Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій та робототехніки
(повна назва факультету)
Кафедра комп'ютерних та інформаційних технологій і систем
(повна назва кафедри)

Пояснювальна записка
до дипломного проекту (роботи)
магістр
(освітньо-кваліфікаційний рівень)
на тему
Розроблення комп'ютерної гри на рушії Unreal Engine 4

Виконав: студент 6 курсу, групи 601-ТН спеціальності
122 Комп'ютерні науки
(шифр і назва напряму)
Деримарко О.О.
(прізвище та ініціали)
Керівник Беседін В.Ф.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Полтава – 2021 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

спеціальність 122«Комп'ютерні науки»

на тему

«Розроблення комп'ютерної гри на рушії Unreal Engine 4»

Студента групи 601-ТН Деримарка Олександра Олександровича

Керівник роботи
доктор економічних наук,
професор Беседін В.Ф.

Завідувач кафедри
кандидат технічних наук,
доцент Головко Г.В.

Полтава – 2021

РЕФЕРАТ

Кваліфікаційна робота магістра: 123 с., 74 малюнка, 2 додатки, 32 джерела

Об’єкт дослідження: процес практичної розробки комп’ютерних ігор у жанрі Action-RPGз використанням рушія UnrealEngine 4.

Мета роботи: створення комп’ютерної гри та певних для неї видів механік у жанрі Action-RPGз метою подальшого її розвитку в майбутньому, можливість впровадження гри в онлайн-сервіси цифрової дистрибуції у вигляді багатокористувацького режиму з цілью зацікавлення більшої аудиторії.

Методи: використання методів контент-моніторингу, теорії ігор, дослідження ринку відеоігор та метод візуалізації розподілу потреб гри для розробки основної логіки та механіки. Реалізація роботи виконувалася на основі методології RAD (Rapid Application Development) та частково в поєднанні з методом DSDM (Dynamic Systems Development Model).

Ключові слова: комп’ютерна гра, Action-RPG, Unreal Engine 4, ігровий дизайн, одиночна гра.

ABSTRACT

Bachelor's qualification work: 123 pages, 74 drawing, 2 appendices, 32 sources

Object of research: the process of practical development of computer games in the genre of Action-RPG using the Unreal Engine 4 engine.

Purpose: to create a computer game and certain types of mechanics in the genre of Action-RPG for further development in the future, the possibility of introducing the game in online digital distribution services in the form of multiplayer mode to interest a larger audience.

Methods: use of content monitoring methods, game theory, video game market research and method of visualization of the distribution of game needs for the development of basic logic and mechanics. Implementation of the work was performed on the basis of the methodology of RAD (Rapid Application Development) and partly in combination with the method of DSDM (Dynamic Systems Development Model).

Keywords: computer game, Action-RPG, Unreal Engine 4, game design, single player.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення.

ААА-ігри – підмножина відеоігор, мають більше коштів на розвиток.

PC – персональний комп’ютер.

2D – двовимірний.

UE4 – ігровий рушій Unreal Engine 4.

WYSIWYP– попередній перегляд контенту у реальному часі.

DLL – динамічна бібліотека.

3D – тривимірний.

MMORPG – масова багатокористувацька онлайнова рольова гра.

IBMPC – перший масовий персональний комп’ютер.

EAX – технологія для створення звукових ефектів.

HDR – технологія покращення деталізації зображення.

Action-RPG – піджанр комп’ютерних ролевих ігор.

CPU – центральний процесорний пристрій.

LOD – рівень деталізації моделі.

MIDI – цифровий інтерфейс музичних інструментів.

Blueprints – система візуального програмування.

AI– штучний інтелект.

UI– користувацький інтерфейс.

NPC– неігровий персонаж.

Mob– ворожий до гравця персонаж.

ГП – головний персонаж.

Quest – завдання в комп'ютерній грі (зазвичай рольової), що видається неігровим персонажем.

ВСТУП

Актуальність даної роботи зумовлена стрімким розвитком індустрії комп’ютерних ігор. На сьогоднішній день розробка відеогри сприймається як дещо більше, чим просто розвага. За останні роки, ринок відеоігор зростає з неймовірною швидкістю, це обумовлено тим, що багато створених комп’ютерних ігор допомагають людям як фізично, так і морально. Наприклад, існують відеогри які покращують людську пам’ять, реакцію, а також допомагають розслабитись та вийти з депресії. Звісно, потрібно розуміти, що якісно розроблена комп’ютерна гра може принести великий дохід, аудиторію, популярність, адже з кожним днем ринок ігрової індустрії розвивається та відкриває багато можливостей для розробників, а головне наявність комфортного життя та мотивації подальшої своєї діяльності.

У прикладному середовищі комп'ютерні ігри можна використовувати в процесі інновацій та освіті, наприклад, в якості спеціальних ігрових навчальних програм, що використовуються як в ході лекцій, так і в ході заліків, екзаменаційних тестів. Інший продуктивний варіант – використання ігрових навчальних програм студентами під час профорієнтаційної практики.

Тому, створення комп’ютерної гри – актуальна тема для розвитку особистості в різних напрямках, які безперечно будуть корисні протягом усього життя.

Метою проєкту, є розроблення відеогри та певних для неї механік у жанрі Action-RPG з цілью подальшого розвитку та монетизації даного продукту в онлайн-сервісах цифрової дистрибуції, що допоможе забезпечити своє життя та дасть можливість створювати більше проєктів.

Об’єкт дослідження заключається у вивченні процесу практичної розробки комп’ютерної гри за допомогою ігрового рушія UnrealEngine 4, а також аналіз та виділення основних проблем сучасних досліджень, пов’язаних з розробкою відеоігор, і визначення найбільш перспективних напрямків розвитку у цій сфері діяльності.

Очікування результатів від даної роботи заключаються у наступних складових:

- продукт, який представляє в собі набір певних механік для коректної роботи у фінальному етапі розробки;
- працездатність всього функціоналу та логіки створеної через систему візуального програмування Blueprints;
- кінцевий продукт, який зацікавить велику частину майбутніх користувачів;
- можливість додаткового доопрацювання та оновлення продукту, впровадження багатокористувацького режиму у тому випадку, якщо дана гра принесе вагомий дохід.

Розібравшись більш детально, можна зрозуміти, що сучасні комп'ютерні ігри виявляють широкий спектр інноваційних можливостей, які можуть бути актуальні не тільки в світлі технологічної модернізації країни, але і в контексті перспективних соціально-гуманітарних досліджень.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД СТВОРЕННЯ КОМП'ЮТЕРНИХ ІГОР

1.1 Аналіз предметної області

Розробка комп'ютерної гри – це мистецтво створення відеоігор, опис їх дизайну, а також безпосередньо сам процес розробки та кінцевий випуск гри.Гра може бути розроблена кількома людьми з обмеженим бюджетом і фінансуватися видавцем. Тому час та ціна розробки залежать від складності проекту. Розробкою відеоігор займається розробник, який може бути представлений однією людиною і компанією. Зазвичай великомасштабні комерційні ігри створюються командами розробників всередині компанії, що спеціалізується на ПК або консольних іграх. Створення гри включає в себе створення концепції, дизайн, складання, тестування і випуск кінцевого продукту. При розробці гри важливо думати про ігрову механіку, нагороди, внутрішньо-ігрові досягнення (achievements), залучення гравців і дизайні рівнів.

Розробником гри може бути програміст, звукорежисер, художник, дизайнер або багато інших ролей, які доступні безпосередньо в галузі розробки відеоігор. Якщо повністю розроблена комп'ютерна гра дозволяє гравцеві взаємодіяти з контентом, а також надає можливість маніпулюванню елементами гри, то тільки тоді можна називати цей продукт «комп'ютерною грою» або «відеогрою».

Щоб брати участь в процесі розробки ігор, вам не обов'язково потрібно писати код. Наприклад, художники можуть створювати і розробляти потрібні ресурси для гри, в той час як інший співробітник може зосередитися на програмуванні. Тестер може взяти участь у розробці, щоб знайти та виправити помилки або «баги» в грі, та впевнитися, що гра працює коректно, як і очікувалося.

Розробка найбільших бюджетних ігор (так названих «AAA-games»)

може коштувати десятки мільйонів доларів США, і за останні десятиліття ці бюджети неухильно росли, а також кількість команд розробників і час створення. Середній бюджет проєкту AAA, зазвичай випускається найбільшими видавничими компаніями, який продається на фізичних носіях або в цифровому варіанті і часто включається в відому серію з кількох ігор, становить від 18 до 24 мільйонів доларів. Ця ексклюзивна мультибюджетна гра для обох платформ (Xbox 360 та PlayStation 3) мала середню ціну у 2012 році 20 мільйонів доларів США, а для повернення її довелося продати близько 2 мільйонів копій.

Тому за допомогою розширення ринку інді-ігор чимало розробників відеоігор мали можливість працювати над своїми ігровими проєктами без фінансових і юридичних зобов'язань перед видавцями. Інді-ігри – це комп'ютерні ігри, створені окремими розробниками або невеликими командами без фінансової підтримки видавця комп'ютерних ігор. Розподіл здійснюється по цифрових каналах розподілу. Масштаб явищ, пов'язаних з інді-іграми, значно зростає з другої половини 2000-х років, головним чином завдяки розробці нових способів онлайн-розповсюдження, одні з яких дистрибуційні сервіси такі, як Steam, Origin, Battle.net, Uplay, а також іншими інструментами розробки.

Дуже часто виникають ситуації, коли розробники зіштовхуються з багатьма проблемами у процесі створення гри. Для вирішення таких проблем, з якими стикалися ігрові платформи та розробники, були створені такі інструменти, як libGDX і OpenGL. Вони допомогли зробити процес розробки комп’ютерних ігор набагато швидше і простіше, надавши безліч готових функцій і можливостей. Проте, до сих пір все ще важко увійти в індустрію створення відеоігор або зрозуміти цю структуру тим, хто не має взагалі досвід роботи програмістом, що є поширеним випадком на сцені розробки продуктів зв’язаних безпосередньо з комп’ютерними іграми.

1.2 Огляд та порівняння ігрових рушіїв

Ігровий рушій – програмний рушій, основна частина програмного забезпечення будь-якої відеогри, відповідає за всі її технічні аспекти, дозволяючи сприяти розвитку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливою особливістю рушія є можливість створення багатоплатформенних ігор (зараз найпоширеніша для ПК, PS4 та Xbox One одночасно) [1].

Ігровий рушій містить п'ять основних компонентів: основна ігрова програма, яка містить логіку гри; двигун візуалізації, який може використовуватися для генерації 3D-анімованої графіки; звуковий двигун, що складається з алгоритмів, пов'язаних зі звуками; фізичний двигун для впровадження «фізичних» умов у системі, а також штучний інтелект – модуль, призначений для використання інженерами програмного забезпечення зі спеціальним призначенням [1].

Отже, розглянемо та проаналізуємо декілька найпопулярніших ігрових рушіїв за останній час, виділимо переваги та недоліки кожного з них.

UnrealEngine 4

UnrealEngine 4 – це один з найпопулярніших ігрових рушіїв, який використовується для розробки ігор рівня AAA (Triple A). UnrealEngine базується на мові C++, тому якщо ви належним чином розумієте мову, ви можете припинити її вибір, але також є можливість створювати відеоігри, не заглиблюючись у саму мову. Комп’ютерні ігри, які були створені на Unreal Engine, можна публікувати на Xbox One, iOS, Mac, Playstation 4, та звісно PC. У Unreal є все необхідне майже для всіх початківців, а також більш професійних та досвідчених людей, включаючи 3Dмоделювання та ландшафтні роботи. Через велику різноманітність інструментів Unreal Engine складніше опанувати, тому варто знати та готуватися до вивчення багато нової інформації. Але наполеглива праця принесе здатність створювати

справді неймовірні ігри [1].

Перша версія рушія UnrealEngine з'явилася в далекому 1998 році, коли компанія EpicGames випустила шутерUnreal. Уже тоді він демонстрував універсальність, поєднуючи в собі графічний і фізичний рушії, систему штучного інтелекту, управління файлової і мережевий системами, а також включаючи готову середу для розробки ігор. Автори рушія спростили взаємодію з ним, щоб розробники могли зосередитися на створенні основних елементів відеоігор, не відволікаючись на дрібниці на зразок налагодження мережевого коду або обчислення колізій [3].

Розвиток Unreal Engine йшов поступово, рік за роком: рушій змінював версії, зростав новими технологіями – багато в чому, до речі, революційними для свого часу. Кожна версія UE привносила нові вражаючі графічні ефекти. Завдяки простоті використання, а також лояльним умовам ліцензування, рушій використовували багато студій, від інді-команд до найбільших компаній, що випускають дорогі AAA-ігри [3].

Зараз великий обсяг роботи при створенні гри лягає на ігровий, креативний дизайн, і створення контенту. В Unreal є все необхідне для дизайнерів, художників та програмістів, причому весь вихідний код відкритий – будь-яку частину можна налаштувати під свій власний проєкт. Це робить систему більш гнучкою. За рахунок відкритості у UE велика аудиторія, а виправлення знайдених помилок та «багів» самим співтовариством швидко потрапляє в офіційні оновлення [3].

Досить вагомою перевагою Unreal Engine є мультиплеєрна гра «з коробки». Фактично, після запуску редактора в кілька натискань можна створити проєкт, в якому вже є мережева гра. З огляду на зростання багатокористувацьких ігор і ставки великих компаній на онлайн-режими з декількома гравцями, це серйозна перевага, якої, наприклад, немає у Unity [3].

Unreal Engine зараз – один із стандартів в розробці відеоігор, тому для професійного розвитку кожен фахівець ігрової розробки повинен знати його

базовий інструментарій. Весною, у 2020 році компанія-розробник ігрового рушія UnrealEngine, EpicGames, вперше продемонструвала можливості нової версії рушія UnrealEngine 5. Однією з головних цілей його створення було досягнення виняткової фото-реалістичності, порівнянної тільки з AAA-іграми і самим життям [3].

Переваги:

- оскільки безліч розробників використовує цей рушій, то у UnrealEngine, мабуть, найкраща спільнота між суперниками. Десятки годин відеоуроків є цьому доказом;
- компетентна технічна підтримка та оновлення інструментів;
- поява все більшого об'єму новихінструментів;
- великий асортимент інструментів для різних видів моделювання;
- має сумісність з багатьма платформами (Android, Linux, iOS, Windows, Mac та інших);
- нова ліцензійна політика передбачаєщомісячну абонплату в розмірі 19 доларів США. Якщо дохід від гри перевищує 5000 доларів США, розробникам рушія буде надана компенсація у розмірі 5%. Це робить інструмент набагато привабливішим для розробників комп'ютерних ігор, ніж раніше.

Недоліки:

- проблематично створювати великі безшовні світи, розраховані на безліч гравців, що робить скрутним розробку MMORPG і інших MMO-ігор на рушієві.
- необхідність ретельно працювати над проєктами на UE, щоб домогтися плавності картинки.

Комп'ютерні ігри, розроблені на UnrealEngine:

- Batman: Arkham Origins;
- Mortal Kombat X;
- Tekken 7;
- Mass Effect 3;

- Mirror’s Edge;
- Medal of Honor;
- DmC: Devil May Cry
- BioShock Infinite.

Рисунок 1.1 – Логотип ігрового рушія Unreal Engine

Unity

Unity – вважається напевно найкращим існуючим простим середовищем розробки відеоігр, яке дозволяє створювати 2D та 3D ігри зазвичай для будь-якої платформи, включаючи Xbox, Windows, Android, iOS, Playstation тощо. Unity також має змогу підтримувати ресурси у сфері комп’ютерних ігор, створені в Maya, Cinema 4D, Blender, 3ds Max, Softimage та іншому програмному забезпеченні. У порівнянні з конкурентами, він має багато незаперечних переваг, але, ключовим є те, що ви платите за ліцензію лише один раз. Яку б популярність не набрала гра – якщо продукт створений на рушії Unity, то вам не потрібно платити додаткові гроші розробнику цього ж рушія. З фінансової точки зору, це дуже гарний варіант, особливо для початківців розробників та для стартапу [1].

Unity – це не просто програмна платформа для коду, це також сучасний

інструмент редагування. За замовчуванням макет екрана пропонує перегляд сцени, де можна керувати всім своїм рівнем, і переглядом гри, який можливо розпочати в будь-який час для тестування. Загальнодоступні змінні експонуються всередині інструменту, тому доволі легко налаштувати все що потрібно для розробки, не занурюючись у код, навіть під час гри. Редактор Unityдуже розширюється з досить простим у навчанні інтерфейсом для побудови користувальницьких інструментів у редакторі [1].

Ігровий рушій Unity можна використати для створення простих 2D мобільних ігор аж до ігор комп’ютерного або консольного рівня якості AAA-проєктів нового покоління. Майже будь-який дизайн відеоігор може бути створений в Unity, даючи достатньо часу, грошей та роботи. Цей рушій має вдосконалені функції, такі як візуально змінні анімації, які, наприклад поєднують анімації дерев та інші вдосконалені системи частинок [1].

Mecanim, унікально потужна і гнучка система анімації в Unity, яка наділяє ваших персонажів неймовірно природним і плавним рухом.

Завдяки рідній інтеграції Mecanim з рушієм Unity, не потрібно ніяких зусиль для інтеграції з проміжним ПЗ сторонніх розробників. Це дозволяє отримувати всі інструменти і робочі процеси, необхідні для створення і побудови м'язової анімації, анімаційних дерев, кінцевих автоматів і контролерів прямо в рушієві [1].

І оскільки система Mecanim глибоко інтегрована в рушій Unity, з цього випливає змога використовувати Mecanim для анімації всього, чого тільки завгодно, починаючи зі «спрайтів» і закінчуючи формами для змішування і інтенсивністю світла. Крім того, за допомогою AnimationEvents можна викликати будь-яку заскриптовану функцію з відтворенням анімації.

Стабільність і міць Unity в поєднанні з новими оптимізаціями, такими як skinnedmeshinstancing, забезпечує виключно плавну роботу гри [1].

Unity створений навколо складової моделі. Ігрові об'єкти розміщуються в ієрархії сцени і збираються з компонентів, включаючи сценарії, «спрайти», 3D-моделі, частинки об’єктів, джерела звуку, 2D або 3D фізичні тіла та

колайдери. Це дає змогу художникам та дизайнерам працювати в редакторі, для того, щоб робити такі речі, як складання рівнів, при тому не програмуючи.

Мета Unity – «демократизувати розвиток ігор», і їм вдалося зробити такий великий крок та створити таку можливість для людей. Рушій є потужним і гнучким, хоча з ним і легко починати роботу, але все ж таки він має багато різноманітних функцій та можливостей, які можна вивчати роками. Деякі люди вивчають цей рушій 5 і більше років, та все одно знаходять та відкривають багато цікавих речей для себе [1].

Одним з найважливіших кроків, які зробили представники Unity для демократизації розвитку, є створення магазину AssetStore (ассет – цифровий об'єкт, який переважно складається з однотипних даних), масштабний ринок для безкоштовного та платного контенту, який може бути використовуваним у вашій грі, будучи вбудованим прямо в інструмент рушія. В цьому магазині можливо знайти найрізноманітніші речі, включаючи приклади проєктів, створених Unity, які допоможуть навчатись розробці відеоігор, художні засоби, які бувають навіть повністю текстуровані, анімовані персонажі та «моби», плагіни, які надають розширені функції, набори корисних пакетів для створення рівнів, комплекти ігор для початківців які працюють у різних жанрах, і навіть повністю розроблені для вас шаблони відеоігор для розбору та аналізу. Також є багато корисних навчальних посібників та активів, які допоможуть спростити початок роботи в рушії Unity [1].

Переваги:

- політика, яка передбачає ліцензію являється вигідною;
- легкий для використання;
- має сумісність з різноманітними платформами;
- відмінна аудиторія;
- наявність величезної бібліотеки ассетів і плагінів;
- якісна фізика твердих тіл і тканин, система LevelofDetail, колізії між об'єктами, складні анімації;

- великий попит між розробниками (допомагає дуже швидко знаходити та виправляти помилки).

Недоліки:

- має певне обмеження інструментарію для розробників;
- повільна робота на фоні інших конкурентних рушіїв;
- великий розмір додатків для створення ігор на Android.

Комп'ютерні ігри, розроблені на Unity:

- Rust;
- Cuphead;
- Ori and the Blind Forest;
- 7 Days to Die;
- Subnautica;
- Firewatch;
- Inside;
- Hearthstone.

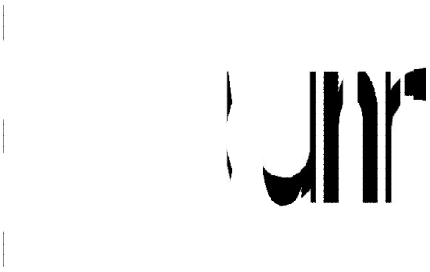


Рисунок 1.2 – Логотип ігрового рушія Unity

CryEngine

CryEngine –неймовірно могутній ігровий рушій, повністю створений

під егідою Crytek, який вперше з'явився у грі Far Cry. Цей рушій застосовується для розробки контенту для консолей та комп'ютерів, охоплюючиXbox One та Playstation 4. Здатністьйого графічних можливостей перевищують Unity, а іноді на крок випереджають Unreal Engine: просунуте освітлення, досить реалістична фізика, передові системи анімації та інше. Якщо зрівнювати з UE та Unity, CryEngine має потужні та інтуїтивно зрозумілі вбудовані функції для проектування більш цікавого та складнішого дизайну та розробки ігор [5].

CryEngine (раніше відомий як CryEngine 3) – це перше комплексне рішення для розробки зі справді масштабними технологіями обчислення та орієнтирів. Завдяки рушієві CryEngine розробники оснащені неймовірно великою кількістю складних для початкових користувачів функцій та інструментів, для створення чудових ігор, використовуючи рушій таких видатних ігор, як Ryse: SonofRome та Crysis [5].

Відомий редактор CryEngineSandbox – це перевірений виробництвом набір інструментів реального часу третього покоління, розроблений та побудований розробниками AAA-продуктів. Всі функції розвитку гри на рушії CryEngine можуть бути вироблені, відредаговані та відтворені відразу ж із системою «те, що ти бачиш, в те, ти і граєш» (WYSIWYP). Двигун займається миттєвою конвертацією та оптимізацією активів у режимі реального часу, що дозволяє робити крос-платформні зміни до всіх елементів процесу створення гри. Це збільшує швидкість та якість розвитку, одночасно набагато знижуючи ризик створення багато-платформних ігор [5].

Незабаром після виходу гри Far Cry всі права на CryEngine були викуплені компанією Ubisoft, яка використовувала рушій для декількох аддонів до шутера. Також він ліг в основу рушія DuniaEngine, на якому були розроблені всі наступні частини серії FarCry. [5].

Crytek тим часом зайнялася створенням рушія CryEngine 2, на якому і був розроблений знаменитий Crysis (а також аддониCrysisWarhead і CrysisWars). Подальші ітерації – CryEngine 3 (зараз належить Amazon),

CryEngine (4-го покоління), CryEngine V – є закономірним розвитком CryEngine 2. Втім, починаючи з 2013 року, привласнення версій рушія порядкових номерів вважається умовною, так як сама Crytek вважає за краще називати його CryEngine, без будь-яких цифр.

Ігри на рушії CryEngine розробляються не тільки студією, що створила його. Спочатку його могли ліцензувати сторонні компанії за фіксовану плату, а освітні установи могли використовувати його безкоштовно, але на некомерційній основі – тільки для навчання студентів. Але починаючи з 2016 року движок і SDK (набір засобів розробки) поширюються безкоштовно для всіх бажаючих, але з умовою виплати Crytek 5% прибутку при доходах, перевищує 5000 доларів / євро (починаючи з версії 5.5, на більш ранніх версіях роялті не виплачується) [5].

Виходячи з вищесказаного, можна зробити висновок, що CryEngine підходить набагато більше досвідченим командам, які мають достатньо коштів і часу для створення дорогих, високоякісних проєктів [5].

Молодим студіям або одиничним розробникам краще звернути увагу на більш доступні (в плані складності розробки) рушії– наприклад, Unity. Втім, з огляду на безкоштовність CryEngine, ніхто не заважає почати створювати відеоігри з його допомогою, ознайомившись з навчальними матеріалами на офіційному сайті. Але в цьому випадку потрібно бути готовим зіткнутися з можливими труднощами в процесі [5].

Переваги:

- функція Flowgraph допоможе прикрасити гру відмінною графікою;
- CryEngine Sandbox: редактор гри в реальному часі, що пропонує зворотний зв'язок «те, що ти бачиш, в те, ти і граєш».;
- модульність: повністю написаний в модульному C++, з коментарями, документацією та розділами в множинних DLL-файлах.
- підтримка самих передових технологій, включаючи DirectX 12, Vulkan API, VR;

- власна технологія трасування променів на рушієві, яка працює на відеокартах AMD і Nvidia і не вимагає потужності графічних чіпів RTX;
- GameSDK– інструмент, на основі якого можна швидко створювати власні ігри, використовуючи в тому числі ассет з офіційного сайту Crytek.

Недоліки:

- складність збірки гри;
- наявність багатьох «багів» в редакторі рушія;
- невеликий вибір ассетів;
- обмеження для розробки мережевих ігор;
- відсутність якісної технічної підтримки;
- відсутність активної аудиторії.

Комп'ютерні ігри, розроблені на CryEngine:

- Crysis;
- Hunt: Showdown;
- Kingdom Come: Deliverance;
- Prey (2017);
- Ryse: Son of Rome;
- Warface;
- Deceit;
- State ofDecay;
- Investigator.

Рисунок 1.2 – Логотип ігрового рушія CryEngine

Rockstar Advanced Game Engine (RAGE)

На даний момент достатньо мало рушіїв, котрі можуть відповідати широкому спектру функцій, наявних у Rage Engine. Rockstar Advanced Game Engine (RAGE) – це ігровий рушій, створений Rockstar San Diego та Rockstar North, які є філіалами американської компанії Rockstar Games. Розширений ігровий механізм Rockstarпризначений для обмеженого використання суто офіційними розробниками, а саме філіями Rockstar Games і не підлягає ліцензіям сторонніх виробників. RAGE вперше був використаний в комп'ютерній грі для настільного тенісу, випущеній Rockstar Games, яка вийшла 23 травня 2006 року. Пізніше рушій був використаний у грі Grand Theft Auto IV та всіх її продовженнях, а також в інших іграх від Rockstar Games. Рушій RAGE підтримує комп'ютери та ігрові консолі, сумісні з IBM PC: Playstation 3, Playstation 4, Xbox 360, Xbox One,Wii та Playstation Portable [5].

У першій половині 2000-х років Rockstar Games використовувала виробничий рушій Criterion Games RenderWare для переважної більшості своїх ігор. Однак літом (2004 рік) видавець з америки,компанія Electronic Arts придбала Criterion Games разом зі своїм рушієм. Це погіршило ліцензійну політику RenderWare, тому Rockstar Games вирішила розробити власний рушій для своїх проєктів. Інформація про намір Rockstar Games розробити власний рушій з'явилася приблизно у вересні 2005 року. Рушій від Rockstar Games заснований на основі рушіяпід назвою AGE (Angel Game Engine), вінвід самого початку створивсязасновниками Angel Studios для таких відеоігор як Midnight Club та декілька іншихпроєктів від Rockstar San Diego для більш новітніх поколінь ігрових консолей. Створення «RAGE» базувалося на основі офіційних співробітників у компанії Rockstar San Diego і Rockstar North під назвою «RAGE Technology Group» [5].

Восени(листопад 2011 рік) Rockstar Games повідомила про розробку

Grand Theft Auto V, в грі використовується движок RAGE. Гра була випущена у вересні (2013 рік) на консолях Playstation 3 і Xbox 360, потім 18 листопада 2014 року гра була випущена на Xbox One і PS4 і 14 квітня 2015 року потрапила вже на ПК [5].

«Rockstar Advanced Game Engine» доволі повнофункціональний ігровий рушій, який містить графічний, фізичний, звуковий рушії, а такожрушій анімації, мережеву взаємодію, та інші компоненти. Оскільки рушій орієнтований на використання в іграх з «відкритим, безшовним світом», основною перевагою рушія є його здатність ефективно обробляти великі ігрові простори. Таким чином, коли персонаж переміщається по рівню, рушій постійно і динамічно додає деякі певні об’єкти і видаляє інші. Згодом розробники повідомили, що для такої технології необхідно було розробити надійний менеджер пам'яті рушія, який міг би постійно розподіляти і видаляти об'єкти з оперативної пам'яті, які б не фрагментували її.RAGE використовує зовнішній рушій фізики Bullet Physics Library, він являється безкоштовним ПЗ.

Розробники приділили багато часу і уваги фізиці автомобілів, які вони намагалися максимально наблизити до реальності. Ця система враховує вагу машини, адгезію шин до поверхні і інші характеристики. Поведінка автомобіля залежить від поверхні, на якій він рухається, а також від погодних умов [5].

Для анімації гуманоїдних персонажів (людей) використовується «ейфорія» – програмний компонент, який автоматично створює анімацію персонажів «на льоту». «Euphoria» була розроблена NaturalMotion і використовувалася в рушії у роліосновного механізму від початку відеогри GTA IV [5].

Переваги:

- швидкий та оптимізований мережевий код;
- великий вибір стилів для геймплея;
- ексклюзивні широкі можливості створення великих світів і

погодних ефектів;

- потужний штучний інтелект (AI).

Недоліки:

- рушій, доступний тільки компанії розробників, тобто відсутність ліцензування для інших компаній;
- рушій, порівняно з другими, відстає по зручності інтерфейсу.

Комп'ютерні ігри, розроблені на RAGE:

- Table Tennis from Rockstar Games;
- GTA IV;
- Max Payne 3;
- Grand Theft Auto V;
- Read Dead Redemption 2;
- Midnight Club: Los Angeles;
- Grand Theft Auto: Episodes from Liberty City.

Рисунок 1.4 – Логотип ігрового рушія RAGE

Frostbite Engine

Frostbite Engine — потужний ігровий рушій, створений шведською компанією DICE,який використовується в серії комп'ютерних ігор і замінив двигун іншої компанії – Refractor Engine. Дана технологія в основному підходить для шутерів від першої особи, ПК з ОС Microsoft Windows та ігрових консолей Playstation та Xbox. Технології можуть відтворювати шкоду довкіллю та навколишньому середовищу (наприклад, деревам, автомобілям,

будівлям) у ігровому світі. Має здатність підтримувати динамічне освітлення та затінення з можливістю HBAO, процедурне затінення, різні пост-ефекти (наприклад, HDR та глибина різкості), системи частинок та технології текстур, таке як рельєфне текстурування. Він також має вбудований власний звуковий рушій, тому немає необхідності використовувати спеціальні інструменти, такі як наприклад EAX.Рушій комплектується ігровим редактором FrostED, написаному на мові програмування C#. Програма призначена для створення рівнів, а також роботи з мешами, шейдерами і об'єктами [5].

Перше покоління Frostbite було розроблено та побудовано спільно з грою Battlefield Bad Company. Спираючись на досвід DICE з широкомасштабними серіями Battlefield відкритого світу, Frostbite був доволі амбітним ігровим рушієм, який давав великі можливість взаємодії в динамічних руйнівних умовах. На сьогоднішній день цей рушій залишається власністю EADICEта не поширюється між іншими розробниками [5].

Переваги:

- високий рівень графіки;
- неймовірна кількість інструментів для розробника;
- багато різноманітних вбудованих модулів;
- якісна оптимізація для багатоядерних процесів.

Недоліки:

- недоступність рушія для інших розробників;
- написання ігор доволі складне на цьому рушії.

Комп’ютерні ігри, розроблені на FrostbiteEngine:

- Battlefield (серія ігор);
- Need for Speed (серія ігор);
- Medal of Honor;
- FIFA (серія ігор);
- Star Wars Battlefront;
- Anthem;

- Mirror’s Edge: Catalyst;
- Madden NFL;
- Shadow Realms;
- Mass Effect;
- Plants vs Zombies (серія ігор);
- Army of Two: The Devil's Cartel;
- Rory McIlroy PGA Tour.

Рисунок 1.5 – Логотип ігрового рушія Frostbite Engine

Source Engine

Source – один із потужних ігрових рушіїв, створений ValveCorporation. Його характеристики: гнучкість та модульна основа, синхронізація голосу та руху губ, механіка вираження емоцій та фізична система, що працює в мережі. Використовується формат моделей рушія .mdl, загальний для продуктів Valve. Система фізикирушія містить частину оновленого коду фізики іншого рушія під назвою «Havok». Source може підтримувати відеокарти DirectX 6-12. Вихідна анімаційна система дозволяє створювати виразних персонажів, міміку з нескінченними емоціями. Ще одним чудовим досягненням є те, що персонажі мають один з найсучасніших штучних

інтелектів, що робить їх дуже досвідченими союзниками та ворогами. За допомогою рушія Source, розробник має можливість та ресурси, щоб легко створити прекрасний і більш реалістичнийігровий світ, який реагує на існування гравця [5].

Трохи історії розробки: почалося все з 1998 року, коли розробники завершуючи роботу над своєю першою грою серії Half-Life, зрозуміли, що в процесі розробки з'явилося безліч напрацювань і деталей, які їм хотілося б впровадити в свій рушій, але тому що гра вже була практично готова, вони не ризикнули вводити нові рішення. У наступні ж роки розробники використовували терміни для назви рушія безпосередньо як «GoldSource» і «Source». GoldSource розвивався з вихідного коду релізной версії рушія, а Source залишилася версією для експериментів і відносилася до майбутньої версії рушія [6].

Таким чином, назву Source стали використовувати для опису нового рушія, а GoldSource став назвою попереднього покоління технології. Варто відзначити, що назва в дослівному перекладі означає «джерело», проте слово source також вживається у словосполученні source code - вихідний код [6].

Першою грою на рушії Source став багатокористувацький шутер Counter-Strike: Source, який вийшов в жовтні 2004 року, він став своєрідною демонстрацією рушія, тому в його назву і вписано назву технології, являючи собою відтворену з новітньої для того часу графікою версію класичного шутера Counter-Strike [6].

Згодом було випущено продовження Half-Life – Half-Life 2, сюжетний науково-фантастичний шутер, який отримав, безліч нагород і відзначений таким чином,що має графіку, яка є однією з найбільш прогресивних для свого часу. В Half-Life 2 дуже активно використовується фізичний рушій, в основу якого ліг Havok, ліцензований Valve. За допомогою Havok Engine побудовані численні головоломки, засновані на грі з фізичними законами. Надалі тематика головоломок з законами фізики була цікаво розвинена в іншому проєкті Valve – Portal випущеному в 2007 році, основною ідеєю гри є

переміщення за допомогою функції телепорту. Крім розвиненої фізичної моделі, Half-Life 2 відрізнялася найбільш передовою для свого часу технологією лицьової анімації. Графічний рушій, що використовує DirectX дев'ятої версії, також відрізнявся сильною оптимізацією і міг працювати на старих моделях відеокарт, знижуючи свою якість графіки і переходячи на більш ранні версії DirectX, аж до шостої версії [6].

Надалі Source, чия структура описана розробниками як вкрай гнучка і модульна, був використаний в більшості ігор компанії, постійно перебуваючи під доопрацюванням і удосконаленням. Було додано безліч сучасних ефектів, а також розширені різноманітні можливості рушія, в тому числі, наприклад, і по роботі з локаціями великих розмірів, додані нові платформи до списку підтримуваних. Спочатку Source був доступний на Windows, пізніше додалися Playstation, Xbox, а вже згодом приєднався Mac (2010 рік). З 2012 року була додана підтримка Linux, а першою портованою грою Valve стала Team Fortress 2 [6].

Фізичний рушій створений на основі Havok. Це рішення дозволяє обчислити набагато більше об’єктів пов’язаних з фізикою, а саме таких, як канати, поверхні, тверді тіла, еластомери тощо. Можна створювати реалістичні транспортні засоби – від автомобілів до морського транспорту на повітряній подушці та пропелерів. Для розрахунку можливої поведінки транспортного засобу на дорозі або в повітрі використовується багато параметрів, наприклад тяга на дорозі, вага автомобіля. Для додання реалістичного руху тіла, використовується фізика «тканинної ляльки»; створена заздалегідь анімація може змішуватися з фізикою реального часу [6].

З розвитком Source, в нього були додані: HDR-рендеринг, затінення з можливостями самозатінення об'єктів і динамічне освітлення, м’які тіні (існує можливість використання традиційного відображення світла), швидкий рендеринг багатоядерних процесорів, а також розвинена система частинок [6].

Ще одною з важливих частин рушія є Source SDK –функціональний набір програм для безкоштовного створення модифікацій на рушії Source доступний через платформу Steam гравцям. У набір входять: редактор карт – Valve Hammer Editor, утиліта для створення лицьової анімації моделей – Faceposer, програма перегляду моделей у форматі .MDL – Model Viewer [6].

На додаток до трьох основних програм, в комплект також входить утиліта для розпакування основнихфайлів при створенні нового ігрового контенту, і звичайноохоплює файли вихідного коду бібліотеки ігор Valve, що дозволяє вручну створювати ігри зі зміненими характеристиками без декомпіляції рушія. Але, потрібно пам’ятати, що для компіляції нових файлів необхідне знання мови C ++ і компілятора.Valve доклали величезних зусиль для того, що б не бути схожими на всіх, в результаті отримали унікальну технологію випереджав свій час [6].

Переваги:

- наявність функції емоцій у персонажів;
- постійна оптимізація та оновлення рушія;
- багато різноманітних вбудованих модулів;
- підтримка мережевих ігор;
- великий захват аудиторії, яка безперечно віддана розробникам;
- всебічні та багатофункціональні різноманітні модулі та елементи для розробки ігор;
- наявність розумного штучного інтелекту.

Недоліки:

- недоступні функції для розробки кросплатформенних ігор та ігор для мобільних операційних систем;
- доволі не простий у використанні початковим користувачам.

Комп’ютерні ігри, розроблені на SourceEngine:

- Counter-Strike: Source
- Left 4 Dead;
- Left 4 Dead 2;

- Half-Life: Source;
- Half-Life 2;
- Portal;
- DOTA 2;
- Portal 2;
- Counter-Strike: Global Offensive;
- Titanfall;
- Team Fortress 2;
- Day of Defeat: Source;
- Alien Swarm.

Рисунок 1.6 – Логотип ігрового рушія Source Engine

Gamemaker

Перш за все, GameMaker – це рушій для новачків-художників, письменників, не досвідчених програмістів, людей з ідеями, які ніколи не писали жодного рядка коду і не знали, з чого почати. Але GameMaker – це набагато більше. Це високоякісний 2D-рушій, ігровий механізм, рушій-платформер-головоломка, рушій піксельного мистецтва. GameMaker був створений YoYo Games 20 років тому з основною метою "спростити процес розробки" за допомогою інструмента візуального сценарію перетягування. В даний час двигун доступний під його останньою ітерацією, GameMaker Studio 2 (GMS2), яка була випущена в березні 2017 року. Більше 1000 користувачів реєструються, щоб користуватися двигуном щодня [18].

У серпні 2021 року YoYo Games спростила ліцензійні параметри для GameMaker, перейшовши від постійних ліцензій до моделі підписки. Замість колишніх ліцензій Creator, Developer, Console та Ultimate, рушій тепер пропонує дві нові опції [18].

Основна сила GameMaker полягає у створенні 2D-ігор, про що наголосив генеральний директор і співзасновник компанії Butterscotch Shenanigans Сет Костер. Співзасновник Crashlands і Levelhead, Костер протягом двох років пройшов шлях від незнання програмування до штатного розробника, і всі вони використовували GameMaker [19].

Ліцензія Indie, що становить 9,99 дол. США щомісяця (або 99,99 дол. США щорічно), надає доступ до всіх неконсольних платформ в одному пакеті. Вона замінює ліцензію розробника рушія, поділену на платформи (настільні, мобільні, веб, UWP). Кожна з них була постійною ліцензією та оцінювалася індивідуально від 99 до 199 доларів [19].

Другий варіант – це ліцензія Enterprise за ціною 79,99 доларів США щомісяця або 799,99 доларів США на рік. Вона надає доступ до всіх платформ. Раніше GameMaker мав ліцензію лише на консоль за таку ж ціну, а також ліцензію Ultimate з усіма платформами за 1500 доларів на рік. Обидва зараз не функціонують. Варто відзначити, що існує також освітня версія GameMaker, призначена для вчителів, які хочуть познайомити своїх студентів з рушієм, з різними варіантами платформи. Ціни починаються з 10 доларів на місяць за місце, при мінімальній купівлі п’яти місць [19].

Переваги:

- можливість обробляти всі жанри та стилі;
- має власну мову, яку легко вивчити;
- спрощує експорт на різні види платформ;
- розміри файлів достатньо добре оптимізовані;
- полегшує створення певних ігрових інструментів для розробників відео-ігор.

Недоліки:

- рушій не безкоштовний;
- рушій не створений для розробки 3D продукту;
- невелика екосистема;
- не підтримує автоматичне розгортання.

Комп’ютерні ігри, розроблені на Gamemaker:

- Undertale;
- Hyper Light Drifter;
- Minit;
- Katana Zero;
- Hotline Miami;
- Shovel Knight.

Рисунок 1.7 – Логотип ігрового рушія Gamemaker

Godot Engine

Godot Engine – це багатофункціональний кросплатформенний ігровий рушій для створення 2D та 3D-ігор з єдиного інтерфейсу. Він надає повний набір поширених інструментів, тому користувачі можуть зосередитися на створенні ігор без необхідності винаходити колесо заново. Ігри можна експортувати в один клік на ряд платформ, включаючи основні настільні платформи (Linux, macOS, Windows), а також мобільні (Android, iOS) та веб-платформи (HTML5) [20].

Godot був спочатку випущений в 2014 році Хуаном Лінієцьким, Аріелем Манзуром, що зробило його молодим рушієм, з яким можна впоратися. Це крос-платформенний двигун (а саме ПК та мобільний), і він

розвивається дуже швидко та досягає високих позицій у забезпеченні свого місця у світі ігор.

Рушій Godot поставляється з мовою програмування під назвою GDScript. Для деяких це не дуже зручно. Часто внутрішні мови або непотрібні, або погано продумані [20].

GDScript з'явився в результаті внутрішнього тестування командою Godot. Замість того, щоб створювати нову мову заради цього, GDScript здійснював ітерацію через інші мови, такі як Python та Lua. Жодна з цих мов не працювала так, як їм хотілося б, тому команда створила GDScript, щоб він був таким же читабельним, як і Python, але при цьому зберігав такі важливі елементи для розробки, як суворе введення тексту, краща інтеграція редактора та більш проста оптимізації швидкості [20].

Багато розробників, які починають з Godot, приємно здивовані тим, як швидко мова засвоюється. Однак, якщо вивчення нової мови немає у вашому списку, є альтернатива.Наразі Godot безпосередньо підтримує C++, C# та GDScript. Вони також працюють над VisualScript, системою програмування без коду на основі вузлів, подібною до системи Blueprint Unreal Engine [21].

Більшість ігрових рушіїв використовують сцени, зазвичай для представлення рівня в грі. Об’єкти існують у цих сценах. В Unity це GameObjects, в Unreal Engine – актори [21].

У Godot сцена – це сукупність вузлів. Кожен вузол є окремим об'єктом, і кожен вузол може успадковувати від будь-якого іншого вузла. Група вузлів називається сценою. Сцени також можуть успадковувати один одного, якщо вони мають спільний кореневий вузол [21].

Godot є повністю безкоштовним і з відкритим кодом та дозволеною ліцензією MIT. Жодних платних функцій, ніяких роялті. Ігри користувачів – їхні, аж до останнього рядка коду рушія. Розробка Godot є повністю незалежною та орієнтованою на спільноту, що дає можливість користувачам формувати свій двигун відповідно до їхніх очікувань. Він підтримується некомерційною організацією Software Freedom Conservancy [21].

Переваги:

- займає не багато місця на жорсткому диску;
- безкоштовний та має відкритий код для розробника;
- кожний ігровий елемент можна анімувати;
- доволі легко знайти навчальні ресурси.

Недоліки:

- не зовсім підходить для складних 3D проєктів;
- не підтримує консолі;
- має невелику кількість громади;
- недостатня кількість функціоналу для великих проєктів.

Комп’ютерні ігри, розроблені на Godot Engine:

- Cruelly Squad;
- Deponia (IOS, Playstation 4);
- Hardcoded;
- Kingdoms of the Dump;
- Carol Reed Mysteries.

Рисунок 1.8 – Логотип ігрового рушія Godot

Порівнявши найпопулярніші на сьогоднішній день ігрові рушії, а також виділивши їх переваги та недоліки, я зупинив свій вибір саме на Unreal Engine 4. В моєму випадку, гра для цього проєкту буде реалізована в форматі 3D, і особисто для себе я вирішив, що найвдаліший рушій за допомогою

якого я зможу якнайбільше розкрити потенціал своєї майбутньої комп’ютерної гри саме UE4. Декілька слів, чому саме цей рушій я обрав, як на мене одна з найголовніших причин вибору UE4 це система Blueprints.

Blueprints – це система візуального скриптинга UnrealEngine 4.Вона є швидким способом створення прототипів ігор. Крім швидкого прототипування, Blueprints також спрощують створення скриптів. Замість порядкового написання коду все можна робити візуально: перетягувати вузли, задавати їх властивості в інтерфейсі і з'єднувати їх «проводи».

Також можна пригадати такі важливі переваги рушія як: вбудований потужний редактор матеріалів, вбудований постпроцесінг, тобто до сцени можна застосовувати bloom-ефект, тонування і згладжування як глобально, так і до окремих її частин (за допомогою компонента PostProcessVolume). Крім цього, в UE4 є такий інструмент як Sequencer,за допомогою якого можливо створюватисінематік та кат-сцени для гри. Це потужний інструмент, що працює за принципом додавання об'єктів на тимчасову шкалу.

Отже, UnrealEngine 4 чудово підходить для створення тривимірних проєктів, а також більш гнучкий порівняно зі своїми конкурентами. Саме тому я і вибрав даний рушій для цього проєкту.

1.3 Вибір жанру для розроблюваної гри

До старту розробки нової гри перед кожним розробником постає завдання вибору жанру і сеттинга для свого майбутнього дітища. Потрібно зрозуміти, що допоможе нам завоювати любов гравців. Тут, звичайно ж, треба виходити з цілого ряду чинників, які варто оцінити перед початком розробки проєкту.

На даному етапі на ігровому ринку помилка з вибором сеттинга і жанру може привести до сумних наслідків, тому до питання потрібно підходити системно і оцінювати всі можливі ризики, які потрібно мінімізувати в майбутньої розробці проєкту. Безліч проєктів не стали успішними саме через

помилки у виборі жанру і сеттинга.

Перед остаточним вибором жанру для майбутньої гри дуже важливо оцінити адекватно свої сили, а саме виділити сильні та слабкі сторони своїх можливостей. Для більшої наочності створимо загальну схему жанрів комп’ютерних ігор, яка відображена в таблиці 1.1.

Таблиця 1.1 – Загальна схема жанрів комп’ютерних ігор

Категорія	**Ігри інформації** (отримання інформації, спілкування, вивчення світу)	**Ігри дій** (переміщення, використання зброї і техніки)	**Ігри контролю** (командування, управління, розподіл коштів)
Гібридні жанри	▪ Action-RPG (бойова ролева гра) ▪ Roguelike (rogue-подібна гра)	▪ MMOFPS (мережевий екшн) ▪ Survival (виживання)	▪ RTS (стратегія-бойовик) ▪ MOBA (багатокористувацька онлайн-арена)
5 елементів	▪ Open RPG (відкрита рольова гра)	▪ Open Action (відкритий бойовик)	▪ Global Strategy (глобальна стратегія)
3 елемента	▪ RPG (рольова гра) ▪ MUD (текстова онлайн-гра) ▪ MMORPG (онлайн рольова гра)	▪ Action (бойовик) ▪ Slasher (бій) ▪ Battle Racing (гонки-битви)	▪ Strategy (локальна стратегія) ▪ SimStrategy (стратегія непрямого контролю) ▪ GlobalWargame (глобальна військова гра)

Продовження таблиці 1.1

2 елемента	▪ Puzzle (головоломка) ▪ Quest (квест) ▪ Browser RPG (браузерні рольові ігри) ▪ Adventure (пригоди)	▪ Platformer (платформер) ▪ Stealth-Action (шпигунський бойовик) ▪ Fighting (поєдинок) ▪ Racing (гонки)	▪ Economical (економічна стратегія) ▪ Tower Defence (захисні вежі) ▪ Wargame (військова гра) ▪ Cardgame (карткова рольова гра)
1 елемент	▪ Education (навчальна гра) ▪ Test (питання, загадки) ▪ Contact (спілкування) ▪ Hero (геройська гра) ▪ Toure (подорож)	▪ Arcade (аркада) ▪ Horror (жахи, виживання) ▪ Shooter (стрільба) ▪ Sport (спорт) ▪ Simulator (симулятор)	▪ Logic (Логічні) ▪ Tactic (Тактична гра) ▪ Micro Control (мікро-контроль) ▪ Building (будівництво) ▪ Life Sim (симулятор життя)
Елементарні жанри	▪ Навчання ▪ Загадки ▪ Спілкування ▪ Роль ▪ Вивчення	▪ Збирання ▪ Ухилення ▪ Знищення ▪ Змагання ▪ Водіння	▪ Турбота ▪ Створення ▪ Контроль ▪ Тактика ▪ Планування

Оцінивши та зваживши свої бажання, вміння і навички, я обрав для своєї майбутньої гри жанр Action-RPG (бойова ролева гра).Цей жанр на стику двох розділів «ігор інформації» і «ігор дій». Тобто суміш двох жанрів: «RPG» і «Slasher». Це все та ж «RPG», але в ній основний упор зроблений на один елемент рольової функції – бій. В наслідок чого інші елементи (розмова, союз, торгівля) атрофувалися і пішли на другий план. Зроблено це за допомогою того, що з усіх елементів гри найбільш цікаво виконаний геймплей битви. Дії також як в «RPG» відбуваються для виконання «квестів». Але в «Action-RPG» мета це не головне, головним стає безпосередньо сам процес.

РОЗДІЛ 2
ПРОЄКТУВАННЯ РОЗРОБЛЮВАНОЇ КОМП'ЮТЕРНОЇ ГРИ

2.1 Короткий опис сценарію гри

Події гри відбуваються у вигаданому всесвіті, за стародавніх часів. Різноманітні створіння, племена, раси, і в тому числі люди ведуть війну за свої інтереси. У грі будуть присутні два головних героя, які потрапили у паралельні виміри, а також ворожі створіння. Перший головний герой – Фріда, жіночий персонаж, людина, яка спеціалізується на ближній та дальній зброї (меч і лук), другий – Аларік, чоловічий персонаж, людина, яка спеціалізується на атаці по області за допомогою магії (атака вогнем по області). В ролі ворогів будуть виступати створіння під назвою «Бойовий мутант-берсерк» та «Павук».

На початку гри, головні персонажі випадковим чином, за допомогою магічних порталів потрапили на поля війни у різні виміри. Фріда та Аларік мають спільну мету, а саме – знайти своїх людей та вибратися з невідомих світів, тобто їх об'єднує одна ціль, але у той же час вони не знають один одного та знаходяться у паралельних вимірах.

Однак, протягом гри, на плечі героїв звалюються різні завдання, якщо Фріда повинна відбиватися від бойових мутантів-берсерків, та намагатись знайти дорогу додому та до своїх рідних, то тим часом Аларік виконує завдання другорядних персонажів (NPC) та намагається дістати інформацію про те, як повернутись назад у свій вимір.

Згодом, головні герої починають розуміти, що все дуже серйозно, ворожі створіння безкінечно з'являються, а завдання мирних людей поповнюються один за одним з кожною хвилиною. Вони усвідомили, що скоріше за все застрягли у нескінченній петлі часу. Герої гри не думають здаватися та падати духом, вони продовжують знищувати ворожих створінь

та допомагати мирним людям у їх справах, цикл за циклом, але в той же час мають надію на те, що їм вдасться розірвати нескінченну петлю часу, та виконати свою головну ціль – дібратися до своїх рідних цілими та неушкодженими.

2.2Проєктування ландшафту

Проєктування нового ландшафту в Unreal Engine 4 може бути виконано двома способами: повністю з нуля, використовуючи при цьому вбудований інструментарій скульптинга або ж завантажити свою карту висот, за допомогою якої буде згенеровано ландшафт. В моєму випадку, це буде проєктування ландшафту повністю з нуля.

Спочатку, перед проєктом ландшафту, потрібно відрити відповідний інструмент, який можна знайти на панелі Modes, котра знаходиться безпосередньо в самому рушію UE4. Потрібний інструмент розташований у вкладці Landscape з іконкою гори.

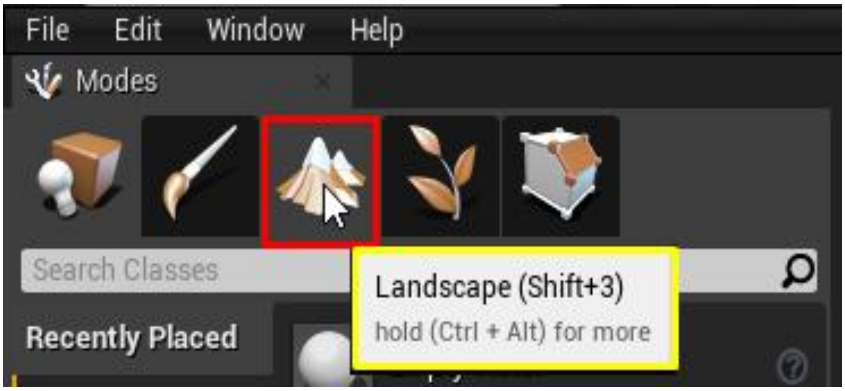


Рисунок 2.1 – Панель Modes в рушії UE4.

Відкривши інструмент Landscape, відразу можна побачити інструментарій по проєктуванню ландшафту і його початкового налаштування (в тому випадку, якщо інструмент був відкритий вперше).

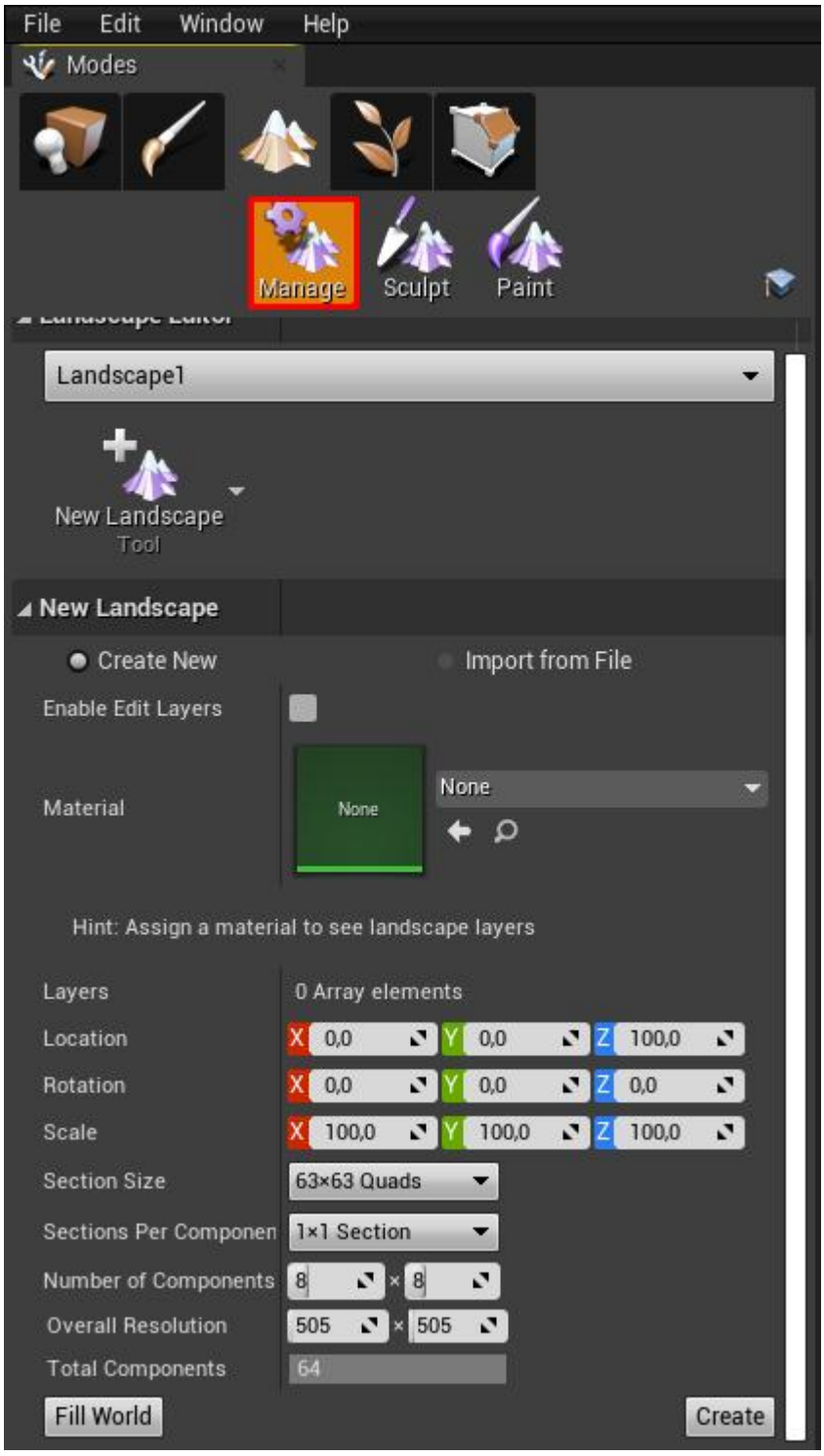


Рисунок 2.2 – Вкладка Manage

Далі, якщо вже є якийсь попередньо спроєктований ландшафт, то вкладка Manage буде відображати меню, що випадає з уже існуючими. В даному меню можна вибрати, з яким ландшафтом ми хочемо працювати.

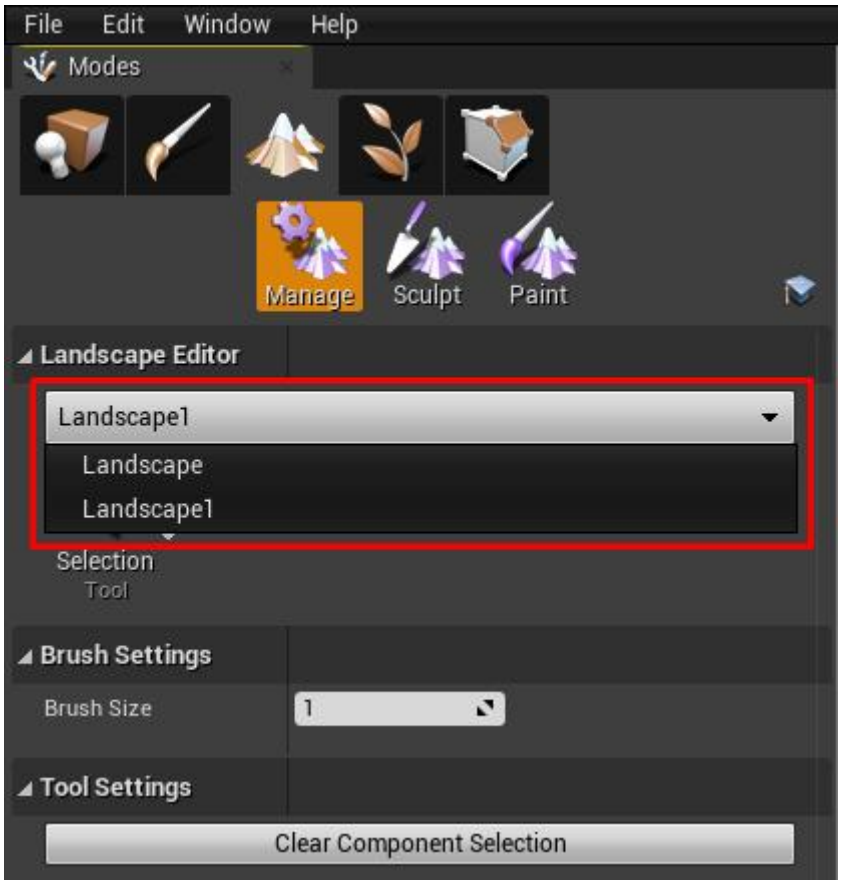


Рисунок 2.3 – Вибір ландшафту

Після цього, перейшовши безпосередньо вже до проєктування ландшафту, потрібно перейти до розділу New Landscape на вкладці Manage. Створюємо новий ландшафт та задаємо потрібні параметри на полях, які розташовані у вікні створення ландшафту.За допомогою різноманітних корисних інструментів, які вбудовані в рушій UnrealEngine, є можливість створити на ландшафті гори, впадини, пагорби та ями. За рахунок цього майбутнє ігрове оточення буде виглядати набаго насичиніше. Це буде відігравати велику роль на атмосфері локації. Також треба враховувати плоскість та розміри створюваного ландшафту, що б не виникали «баги» та помилки при розробці решти компонентів гри.

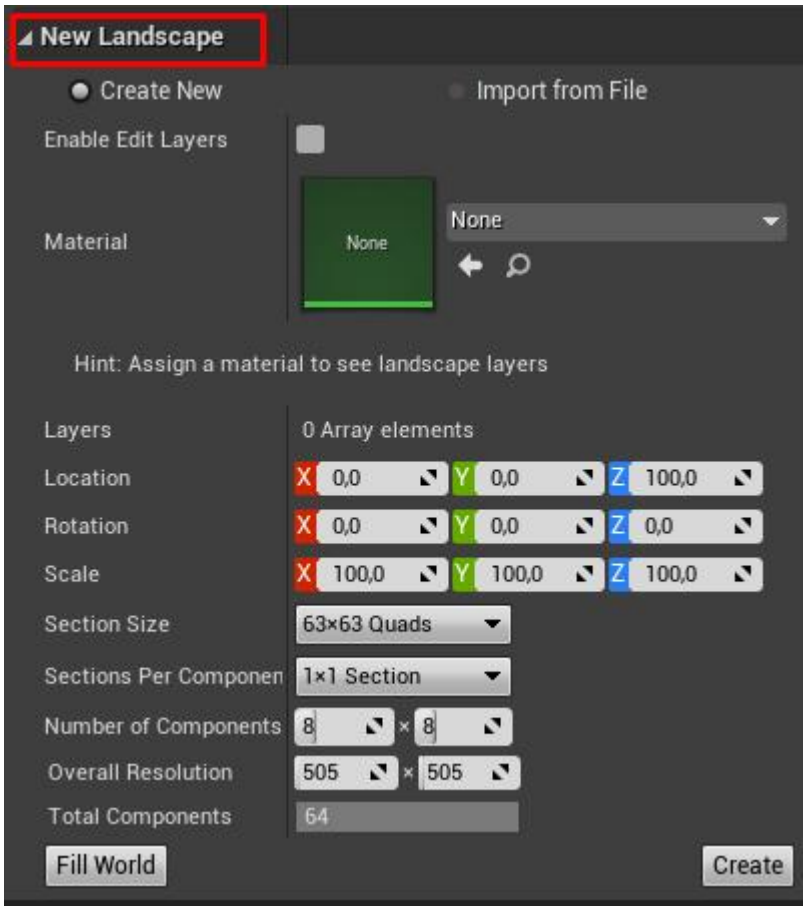


Рисунок 2.4 – Проєктування нового ландшафту

Для більшого розуміння параметрів цього меню, потрібно розглянути їх детальніше:

- Create New – створення нового ландшафту;
- Import from File – імпортування карти висот для створення ландшафту;
- Material – дозволяє відразу накласти матеріал на ландшафт;
- Location – положення створюваного ландшафту;
- Rotation – поворот ландшафту;
- Scale – масштаб ландшафту;
- Section Size – розмір секції, використовується для LOD. Маленький розмір секцій дозволить перемикати LOD більш агресивно, але з більшим навантаженням на CPU. Великі значення навпаки, тому не рекомендується встановлювати маленькі значення при великих розмірах ландшафту в уникненні занадто великого навантаження на CPU;

- Section Per Component – кількість секцій на компонент. Так само допомагає при LOD, так як кожна секція – це, по своїй суті, крок LOD'а. Один компонент може мати 2 на 2 секції, це означає, що компонент буде здійснювати рендер 4-ьох LOD'ів;
- Number of Components – разом з Section Size, впливає на розмір всього ландшафту. Ліміт значення 32x32, так як кожен компонент асоціюється з навантаженням на процесор і більше значення може викликати проблеми з продуктивністю.
- Overall Resolution – кількість клітин площі, використовуваних в ландшафті;
- Fill World – цей параметр робить ландшафт настільки великим, наскільки це можливо;
- Create – створити ландшафт, використовуючи вищеописані налаштування.

Для прикладу, залишаємо налаштування за замовчуванням. Після цього, в головному вікні попереднього перегляду проєкта відобразиться наш новий ландшафт, а налаштування будуть ті, які були обрані раніше (в цьому випадку за замовчуванням).

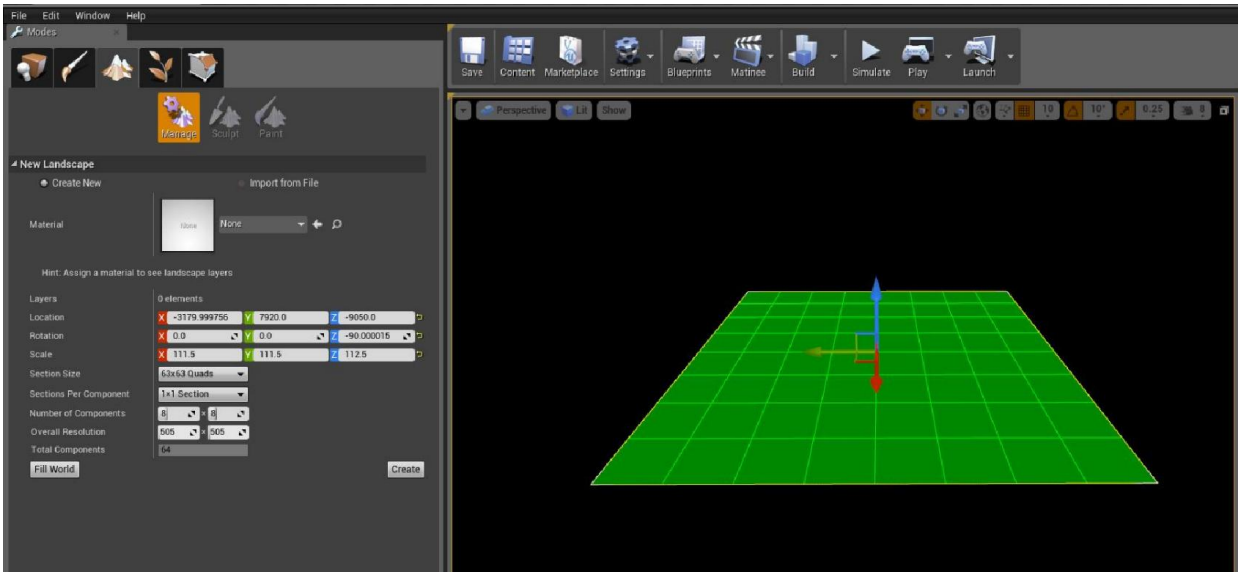


Рисунок 2.5 – Перегляд спроєктованого ландшафту з нуля

Також, у режимі попереднього перегляду є можливість переміщати, повертати, масштабувати наш ландшафт так само, як і звичайний об'єкт. Так

само є можливість потягнути за край, для того що б швидко змінити його розмір. Якщо присутнє бажання, можна відразу застосувати матеріал на ландшафт перед тим, як спроєктували його. Для цього потрібно обрати попередньо створений матеріал (в нашому випадку це матеріал трави) і поруч з пунктом Material натиснути стрілочку.

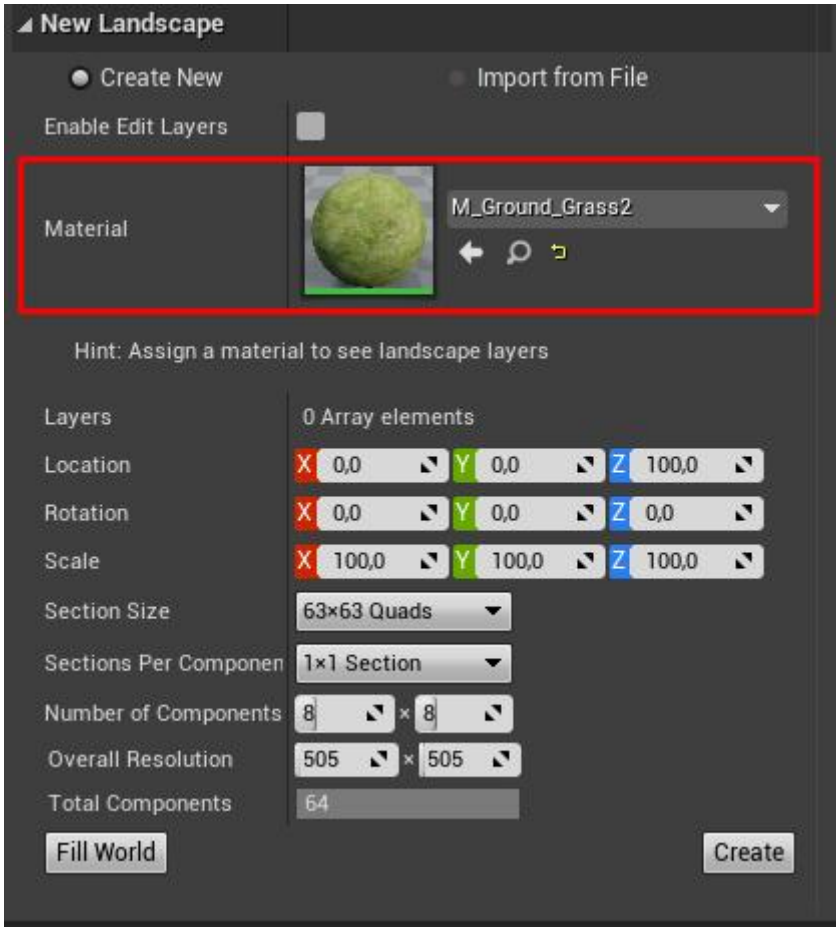


Рисунок 2.6 – Застосування матеріалу на ландшафт

2.3 Проєктування головного персонажу гри

Для проєктування головного персонажу, я обрав редактор тривимірної графіки під назвою Autodesk Maya 2020. Це потужний програмний комплекс для моделювання 3D об'єктів, створення анімацій персонажів, графіки руху об'єктів, проєктування віртуальних середовищ та комп'ютерних персонажів. Взагалі, редактор Maya містить в собі доволі широкий набір функціональних інструментів для різних напрямів діяльності, а саме для художників по

анімації підходить найбільше.

Сам процес проєктування персонажу буде базуватися на основі створеного та налаштованого «скелета». У професійному середовищі процес створення і налаштування скелета персонажа називається skinning, що означає «натягування шкіри», тобто це процес створення системи кісток (скелета) персонажа, приблизно таких же, які мають всі хребетні істоти, а після цього починається зв'язування цих кісток з геометрією тривимірного персонажа.

Кістки тривимірному персонажу потрібні для того ж, для чого вони потрібні людям – вони дають можливість йому рухатися. Сама по собі тривимірна сітка рухатися не може, так само як і не може цього робити наше тіло, якщо його позбавити кісток. Тому саме створення скелета (системи кісток) – це перший крок для нормальної розробки анімації персонажа в майбутньому.

Скелет людиноподібних персонажів або тварин дуже часто створюється на основі анатомічної будови людини або тварини з невеликими змінами. Ці зміни потрібні для врахування особливостей того, як поводяться кістки в тривимірному середовищі. Дуже часто, створивши правильний анатомічний скелет, ми не можемо отримати реалістичну зміна сітки (геометрії) цим скелетом, тому потрібні невеликі доопрацювання, у вигляді додаткових кісток, яких у людини немає, але вони потрібні в тривимірному середовищі для правильної і реалістичною деформації моделі. У фантастичних персонажів скелети можуть сильно відрізнятися від реальних, хоча дуже часто в їх основі знову таки лежить анатомія людського скелета або скелета тварин.

Для створення системи кісток можуть використовуватися як «класичні» кістки, тобто прості об'єкти, які взаємодіють між собою за рахунок ієрархії, або ж більш складні системи кісток, які були спеціально розроблені для створення скелетів, наприклад, двоногих персонажів. Прикладами таких систем можуть служити модулі Biped (двоногий) або ж CAT (Characters

Animation Tools – інструменти для анімації персонажів). Останнім часом навіть з'явився сервіс Mixamo, який дозволяє працювати зі скелетом і анімацією персонажа прямо в інтернет-браузері, минаючи спеціалізоване ПО для цього.

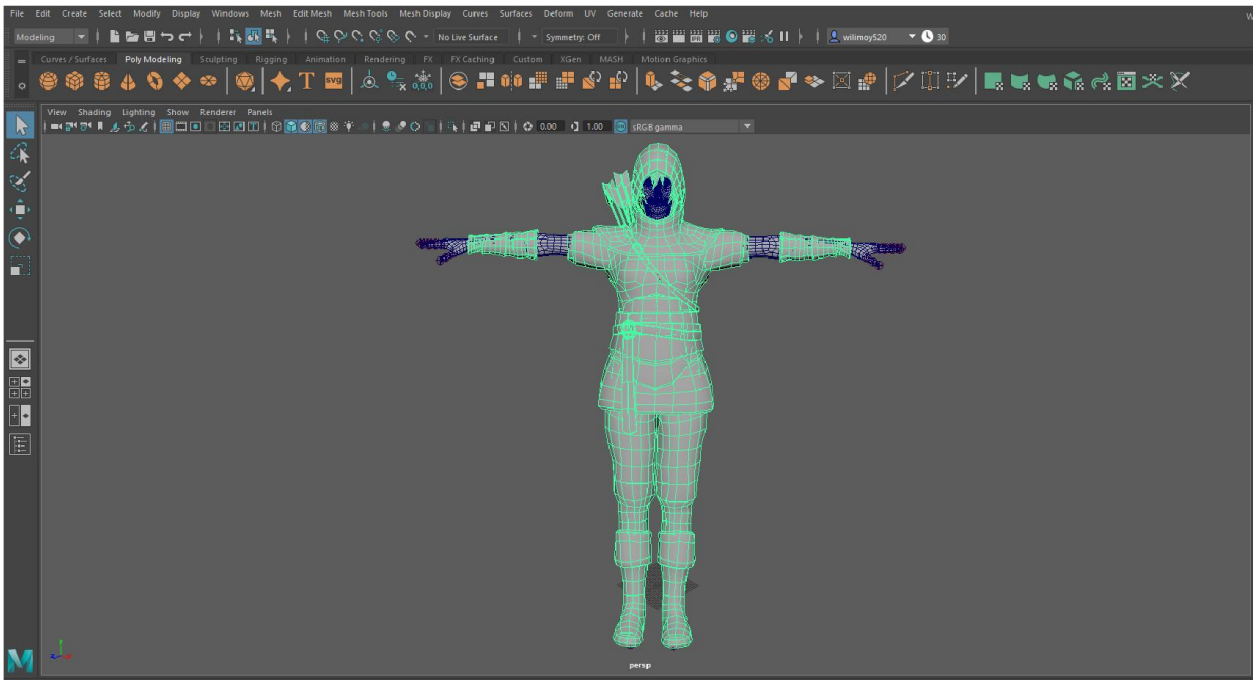


Рисунок 2.7 – Проєктування скелета персонажу

2.4 Проєктування фонової музики для гри

Протягом багатьох років музика в відеоіграх розвивалася так само швидко, як і її технології. Поряд з пристроєм для створення атмосфери в рамках сюжету і обстановки кожного рівня, музика використовувалася для безпосереднього художнього обміну інформацією з гравцями, що виплила в такий унікального аспект для відеоігор як динамічний саундтрек. У музичному плані, ігри, такі як наприклад The Elder Scrolls V: Skyrim, можуть включати в себе красиві делікатні мелодії скрипки, в залежності від місця подій в грі. Все частіше композитори починають виходити за рамки звичайної чутливості композиції, створюючи мелодії, які є досить захоплюючими, щоб витягнути емоції гравців, і досить точними, щоб забезпечити зручний перехід до наступної пісні в будь-який момент.

Відеоігри побудовані на інформації точно так же, як емоції, і ця інформація повинна передаватися творчо. Навіть без складного динамічного саундтрека музика може передавати інформацію.

Відеоігри – це те, що можна назвати «аудіовізуальним» поданням. Тобто візуальні ефекти і звук об'єднуються для створення єдиного, інтерактивного досвіду. Але коли один аспект перестає контактувати з іншим – створюється відчуття відстороненості, гравці бачать одне, але чують при цьому інше.

В аудіовізуальному світі, музика – найкращий спосіб встановити настрій і тон. Вона грає на наших почуттях, закрадається до нас в голову. Це також відображається на психології гравця, не дивно, що люди до сих пір продовжують в багатьох комп’ютерних іграх відчувати себе спустошено і тоскно, коли наприклад розробник створює спеціально для цього відповідну музику.

Звукове оформлення ігор вимагає постійного пошуку взаємовигідних рішень з боку замовника і композитора. Цей процес може включати в себе багатогодинні суперечки про те, як краще вбудувати в атмосферу гри шум автомобілів, техніки та голоси людей, але він залишається невидимий для нас. В цьому і полягає його призначення – зробити навколишній світ більш реалістичним, «живим» і природним для гравця.

На відміну від більшості пісень, які можуть звучати окремо як приємні на слух твори мистецтва, пісні, які використовуються у відеоіграх, повинні виконувати різні специфічні функції. Немає сумнівів в тому, що складання та аранжування музики до відеоігор – складне завдання, але хороший саундтрек завжди зможе зробити з гри – шедевр.

Для проєктування фонової музики та звуків для гри, було обрано аудіо-редактор SoundationStudio.

Soundation Studio пропонує себе як аудіо-редактор, створений майстрами в створенні бібліотек семплів – PowerFX. Дозволяє записувати звук з мікрофона, додавати семпли з бібліотеки семплів, обробляти аудіо-

фрагменти з вбудованими ефектами, імпортувати MIDI файли з жорсткого диска, має вбудований метроном і найпростіший синтезатор. Відрізняється зручним інтерфейсом в дусі Mac OS X і можливістю автоматизації будь-яких параметрів.

Це програмне забезпечення доволі зручно себе позиціонує, не маючи досвіду роботи з аудіо-редакторами, можна за короткий час опанувати початкові знання та вже почати легко орієнтуватися в інтерфейсі даного продукту. Soundation Studio виявився універсальним та зручним редактором для проєктування музики. Головна перевага цього продукту – безкоштовна версія.

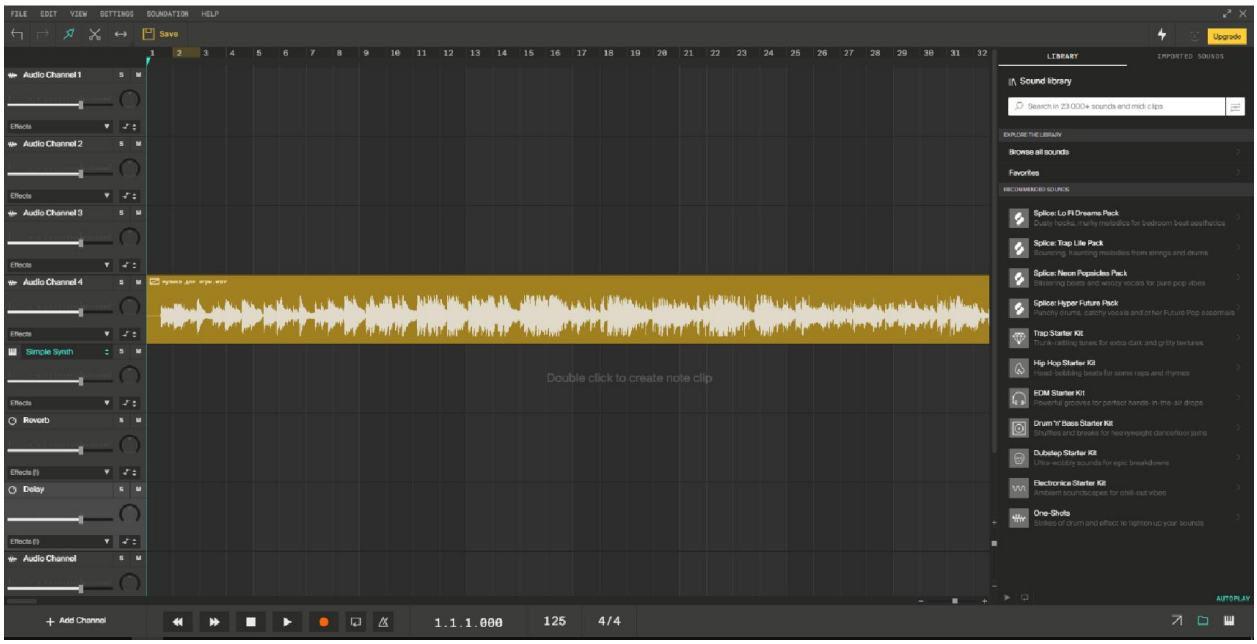


Рисунок 2.8 – Проєктування фонової музики для гри

2.5 Проєктування ігрового штучного інтелекту

У Unreal Engine 4 створювати штучний інтелект можна за допомогою дерев поведінки. Дерево поведінки (behavior tree) – це система визначення поведінки, що використовується штучним інтелектом. Наприклад, у нього може бути поведінка бою, бігу чи видача завдань для головного персонажа гри. Також є можливість створити дерево поведінки, при якому штучний інтелект битиметься з гравцем, якщо його здоров'я вище. Якщо воно нижче

50%, то він тікатиме.

Для початку, потрібно створити дерево поведінки, використовуючи контент браузер безпосередньо у самому рушії. В Content Browser натискаємо кнопку Add New і обираємо Behavior Tree у розділі Miscellaneous.

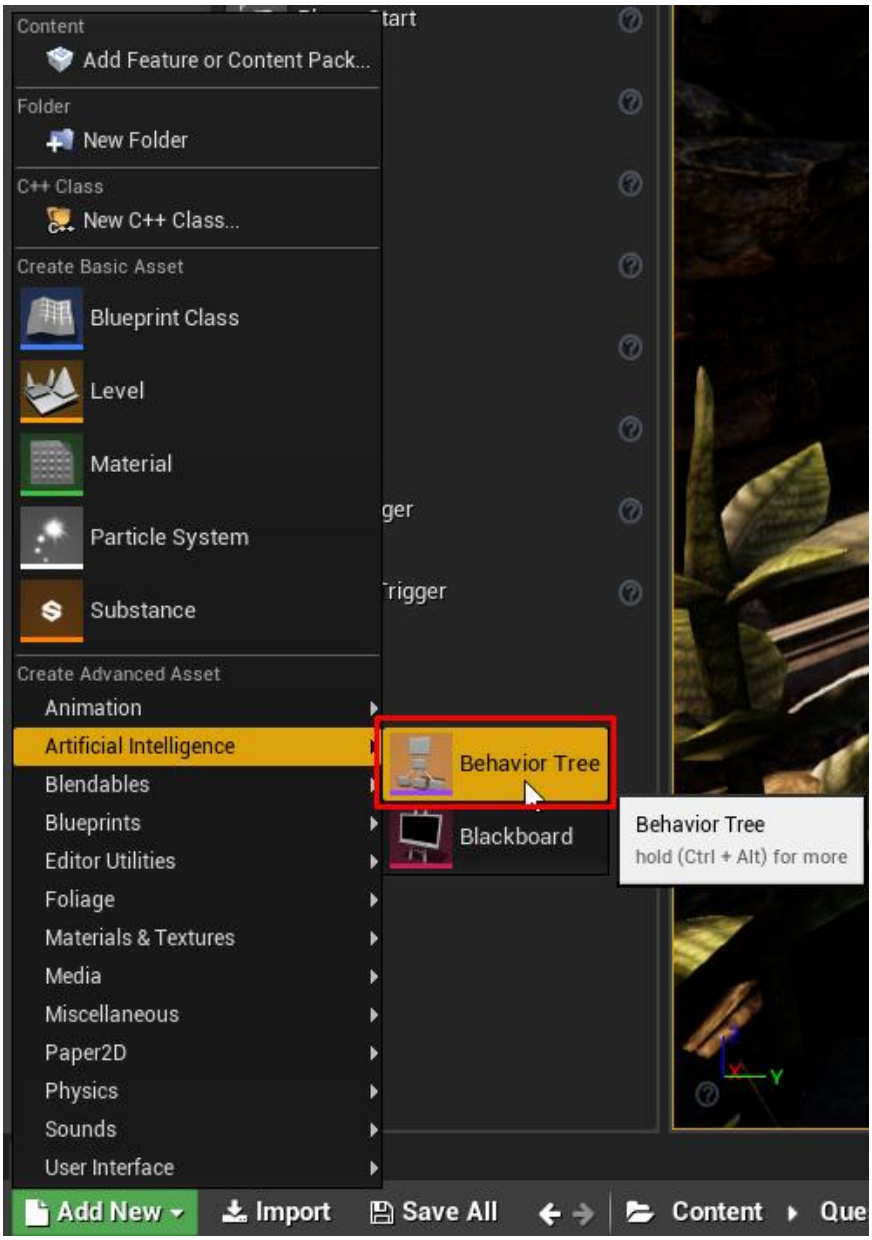


Рисунок 2.9 – Створення дерева поведінки для штучного інтелекту

Наступним кроком, потрібно створити Blackboard, натискаємо на кнопку New Blackboard знаходячись у режимі редагування дерева поведінки. Blackboard – це ресурс, єдине призначення якого полягає у зберіганні змінних (званих ключами (keys)). Можна вважати його пам'яттю штучного інтелекту.Хоча використовувати їх і не обов'язково, blackboard забезпечує

зручний спосіб зчитування та збереження даних. Він зручний, тому що багато вузлів у деревах поведінки можуть отримувати тільки ключі blackboard.

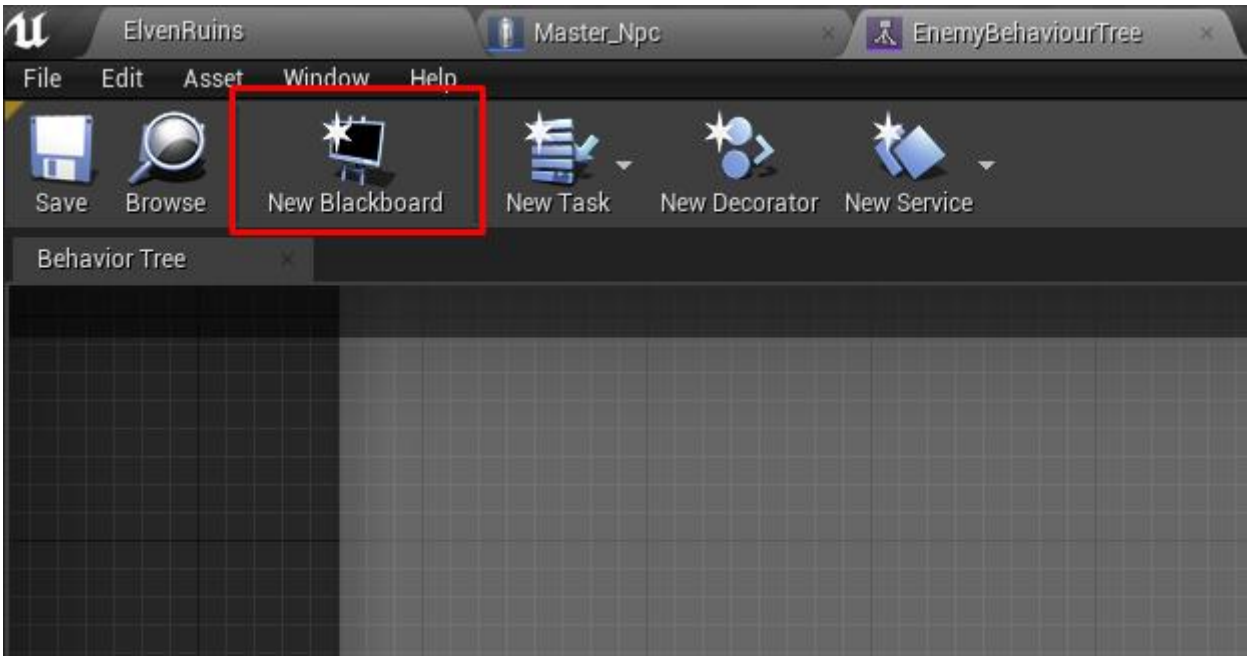


Рисунок 2.10 – Створення Blackboard для дерева поведінки

Тепер, після виконання вищесказаних дій, можна створювати задачі для нашого штучного інтелекту. Існують три основні задачі (Blueprint) для дерева поведінки: task, decorator, service.

Task (завдання) – це вузол, який щось «робить». Це може бути щось складне, наприклад, ланцюжок комбо, або щось просте, наприклад, очікування.

Decorator – вираз, також відомий як «умовний». Його прикріплюють до інших виразів і створюють рішення для відгалуження в дереві поведінки. Декоратори приєднуються до завдань чи композитів. Зазвичай декоратори використовують для виконання перевірок. Якщо результат дорівнює true, декоратор теж повертає true, і навпаки. Завдяки декораторам можна керувати виконанням їхніх родоначальних елементів.

Service (служба) – її прикріплюють до виразів, і виконують з певною частотою доти, доки гілка виконується. Вони часто використовуються для створення перевірок та оновлень Blackboard. Вона займає місце паралельно

іншим виразам у Behavior Tree. Це окремі вузли, вони приєднуються до завдань чи композитам. Завдяки цьому створюється більш упорядковане дерево, тому що тоді доводиться працювати з меншою кількістю вузлів.

РОЗДІЛ 3
РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ

3.1 Створення ігрового оточення

Розробка гри в цілому буде створена за допомогою системи візуального скриптинга Blueprints, тому розглянемо більш детально цю систему.

Система візуальних сценаріїв Blueprints в Unreal Engine – це повна система сценаріїв ігрових процесів, яка заснована на концепції використання інтерфейсу на основі вузла (node) для створення елементів ігрового процесу в редакторі Unreal. Як і у багатьох поширених скриптованих мовах, він використовується для визначення об'єктно-орієнтованих класів або об'єктів в рушії. Ця система надзвичайно гнучка та потужна, оскільки надає можливість дизайнерам використовувати практично повний спектр понять та інструментів, як правило, доступних лише програмістам. Крім того, можливості, характерні для Blueprint також доступні в реалізації C++ Unreal Engine та навпаки, що дозволяє програмістам створювати базові системи або відеоігри, які можуть бути розширені дизайнерами. У своїй базовій формі Blueprints можуть виглядати як доповнення до гри. Поєднуючи вузли, події, функції та змінні з проводами, можна створити складні елементи гри.

Після того, як був спроєктований ландшафт гри, потрібно створити та вдосконалити безпосередньо саме ігрове оточення. Воно буде базуватися та створюватися на основі таких інструментів та функцій:

- blueprint – система візуального програмування в UnrealEngine 4;
- набір текстур, матеріалів, моделей, які доступні в Marketplace Unreal Engine;
- основні вбудовані загальнодоступні засоби Unreal Engine 4.

Перше, що потрібно зробити це задати текстуру нашому спроєктованому ландшафту, для цього потрібно розуміти, що в ігровому рушії немає такого поняття безпосередньо як «текстура», точніше поняття є, але саму текстуру застосувати для об’єктів не можна. Текстури об’єднуються в так звані «матеріали», а вже самі матеріали ми можемо застосувати до різноманітних предметів та об’єктів, включаючи різноманітні частинки та елементи UI.

Як і в реальному світі, в іграх є безліч об'єктів, кожен зі своїм зовнішнім виглядом. В UE4 цей зовнішній вигляд залежить від матеріалів. Який колір має об'єкт? Прозорий він? Блищить чи ні? Всі ці властивості задаються матеріалами.

У нашому випадку, я додав до свого проєкту доступний матеріал з маркету Unreal Engine 4, який має назву summer_terrain. Наступним кроком, я застосував цей матеріал до ландшафту та отримав початковий вид ігрового оточення.

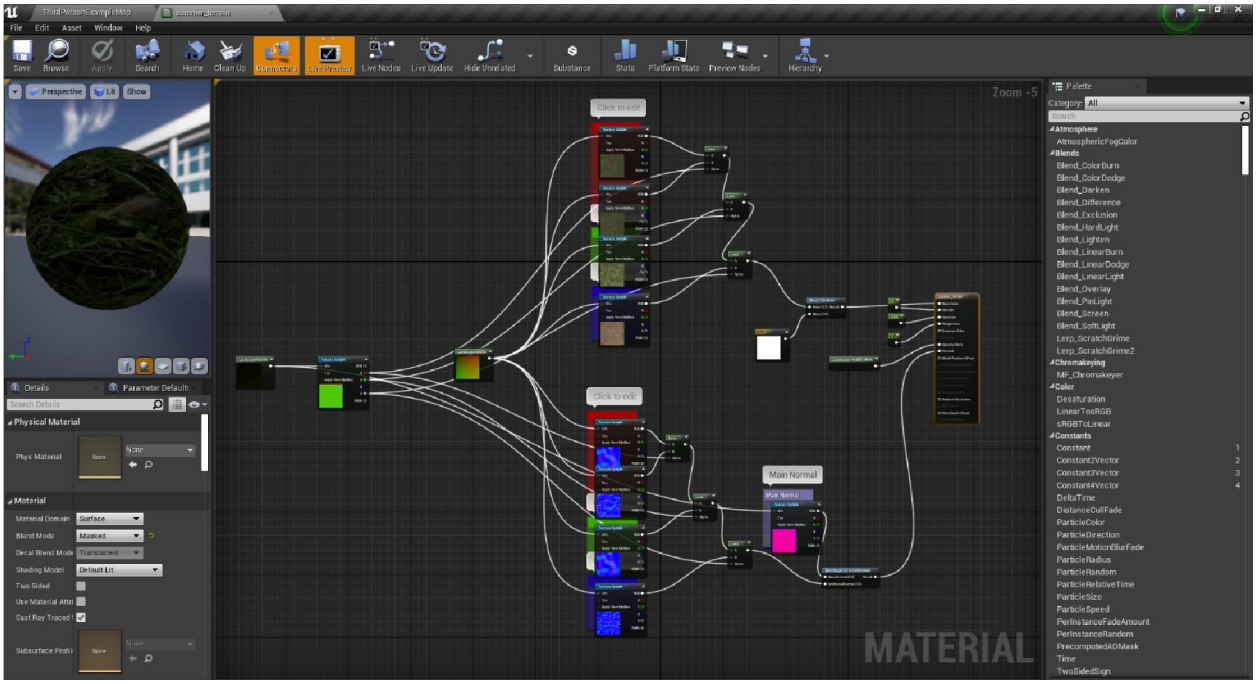


Рисунок 3.1 – Матеріал ландшафту summer_terrain

Далі, додаємо різноманітні об’єкти та предмети для ігрового оточення, які зображені на рисунках 3.2 – 3.10.

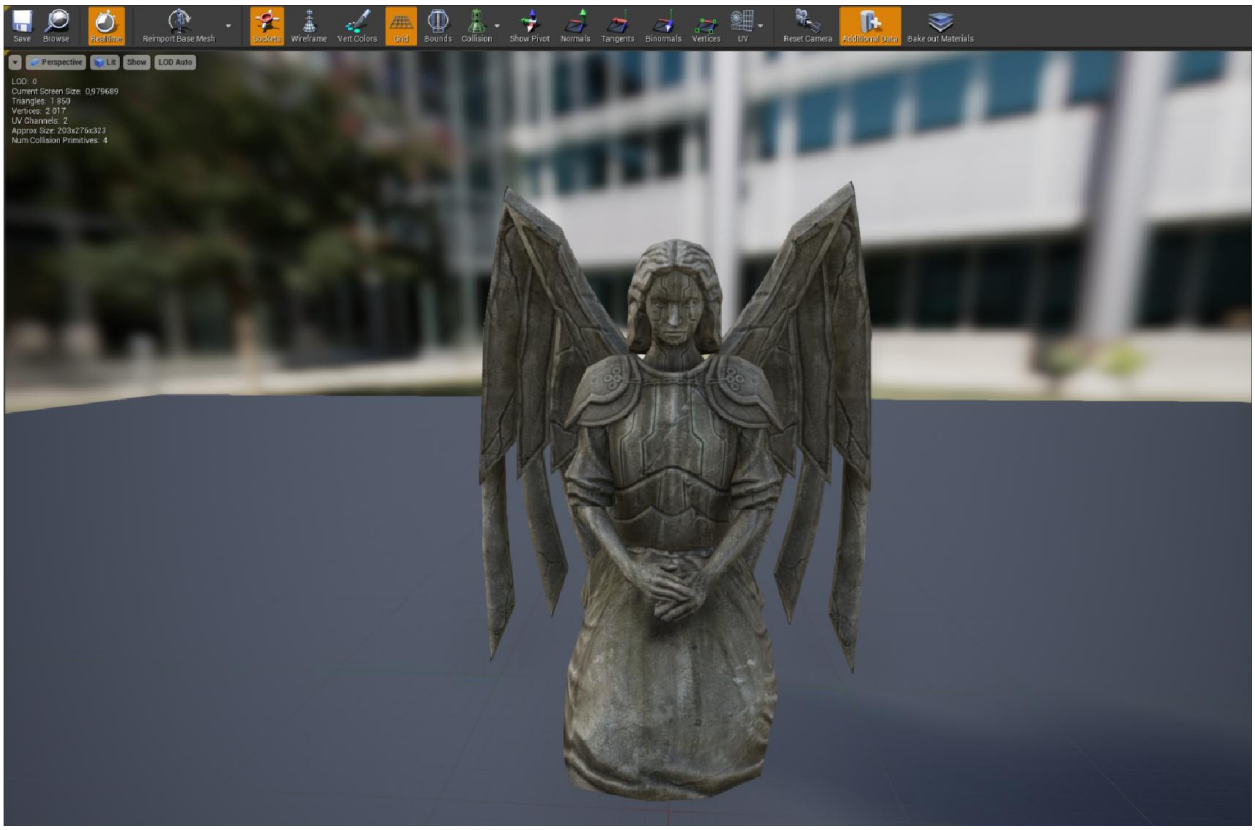


Рисунок 3.2 – Об’єкт статуїянгола angel_statue

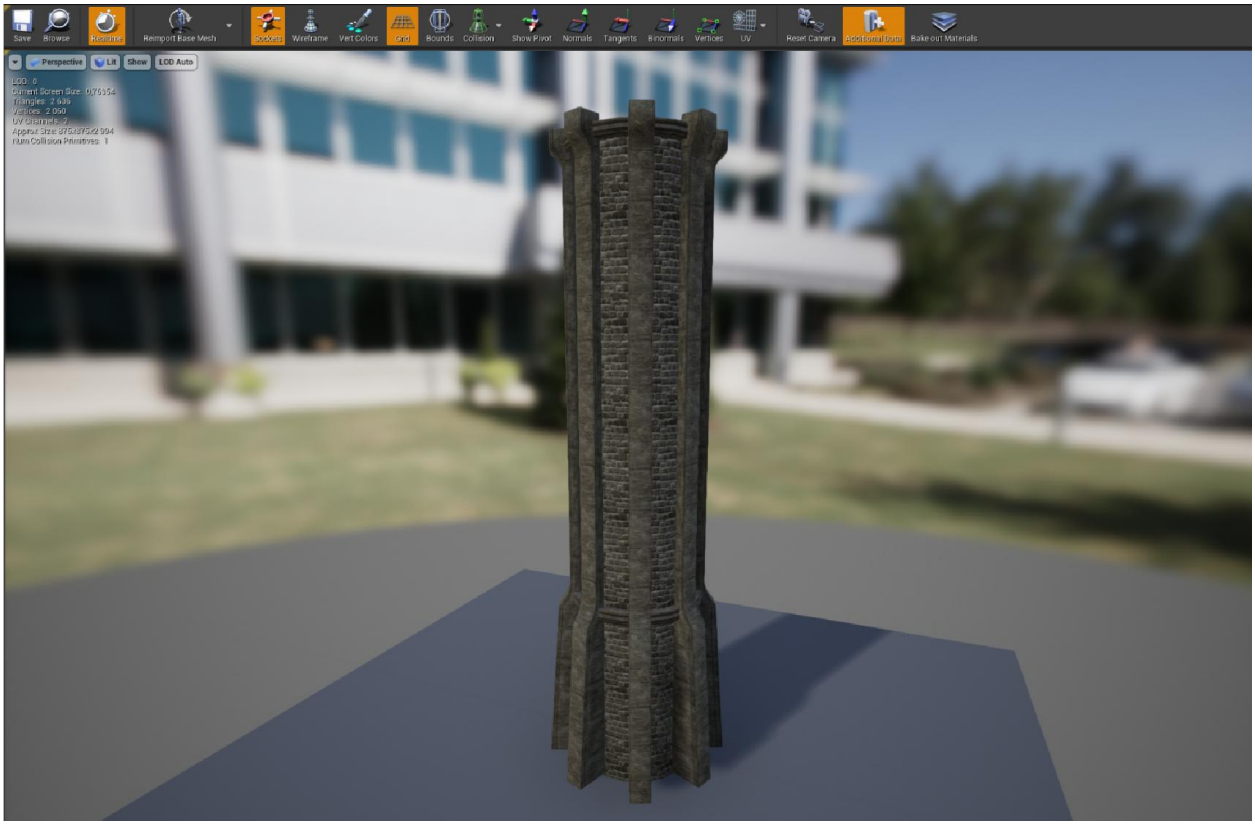


Рисунок 3.3 – Об’єкт вежі замкуplains_castletower

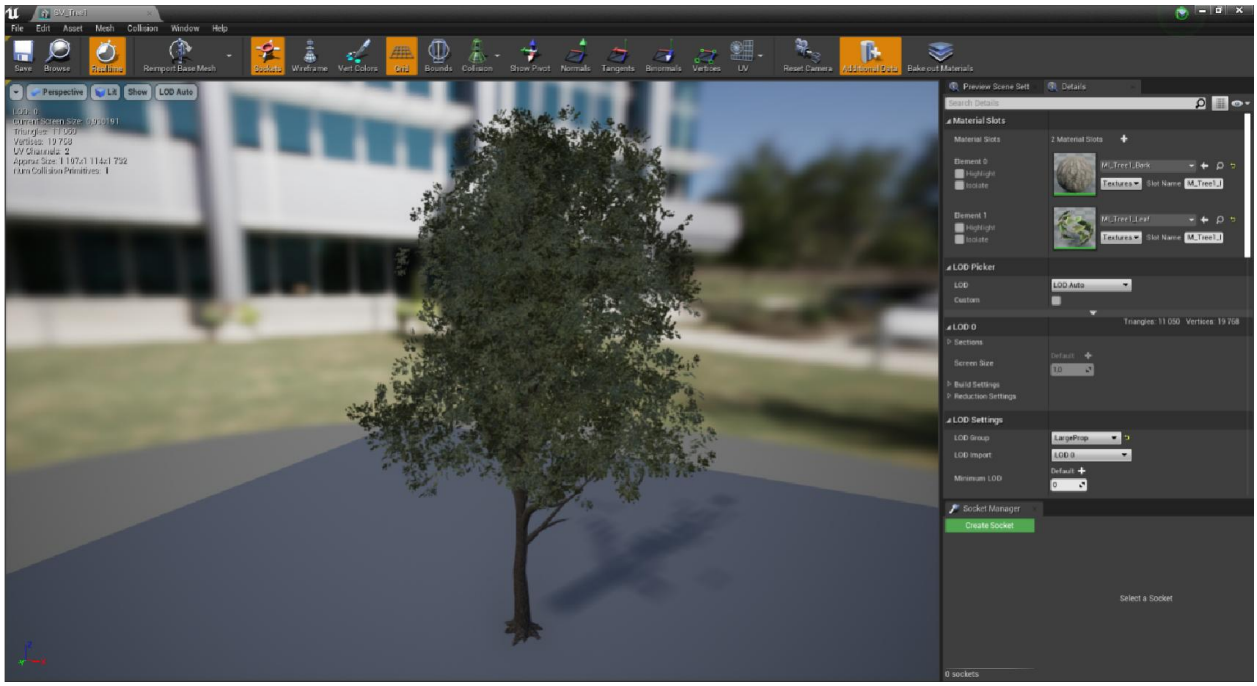


Рисунок 3.4 – Об’єкт дерева sm_tree

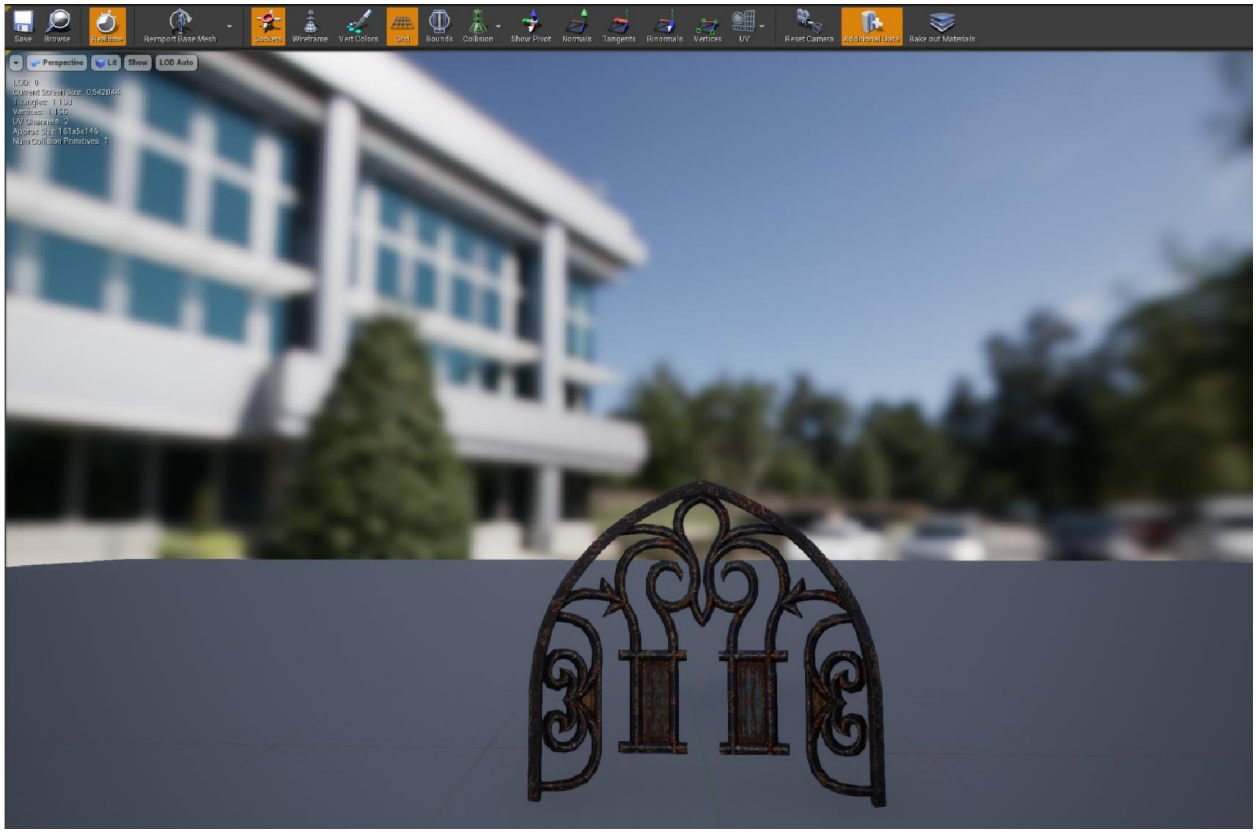


Рисунок 3.5 – Об’єкт залізної аркиcastlearch_iron

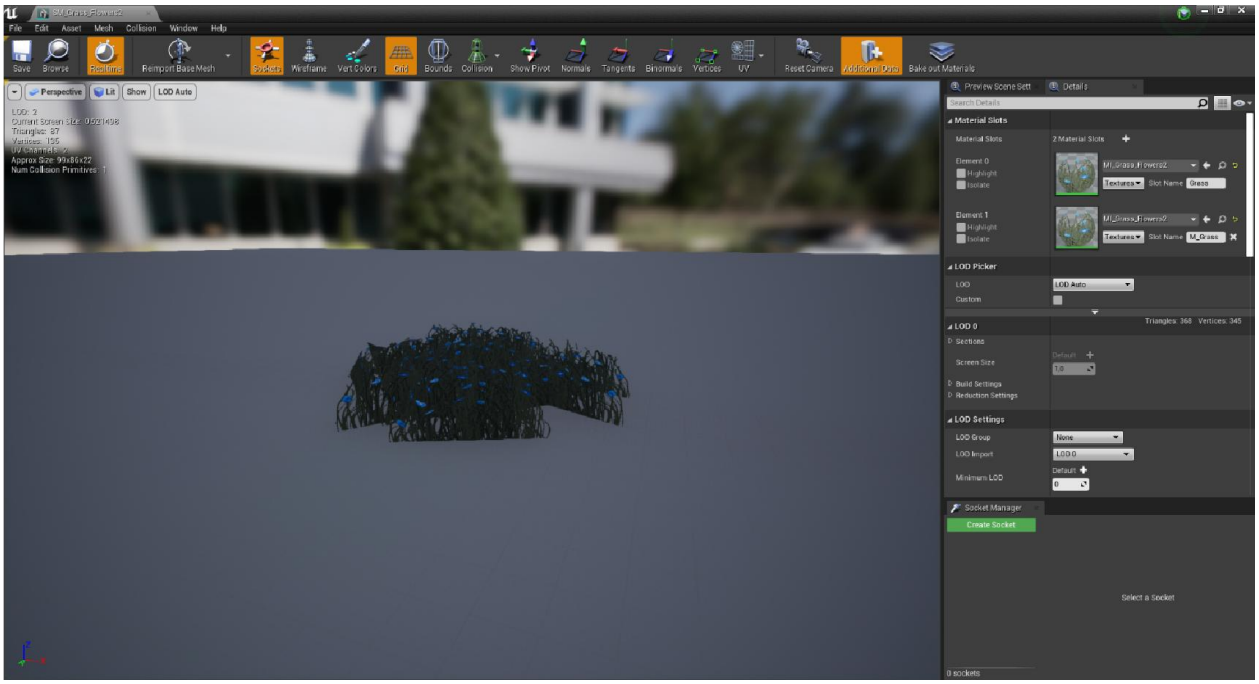


Рисунок 3.6 – Об’єкт квітів grass_flowers

На додаванні об’єктів не закінчуємо, продовжуємо удосконалювати та насичувати ігрове оточення створенням озера та печери. Щоб створити озеро, було використано матеріал води, деякі об’єкти рослин та раніше спеціально підготовлене місце на ландшафті. Печера була створена аналогічними діями, замість матеріалу води були використані спеціальні об’єкти каміння.

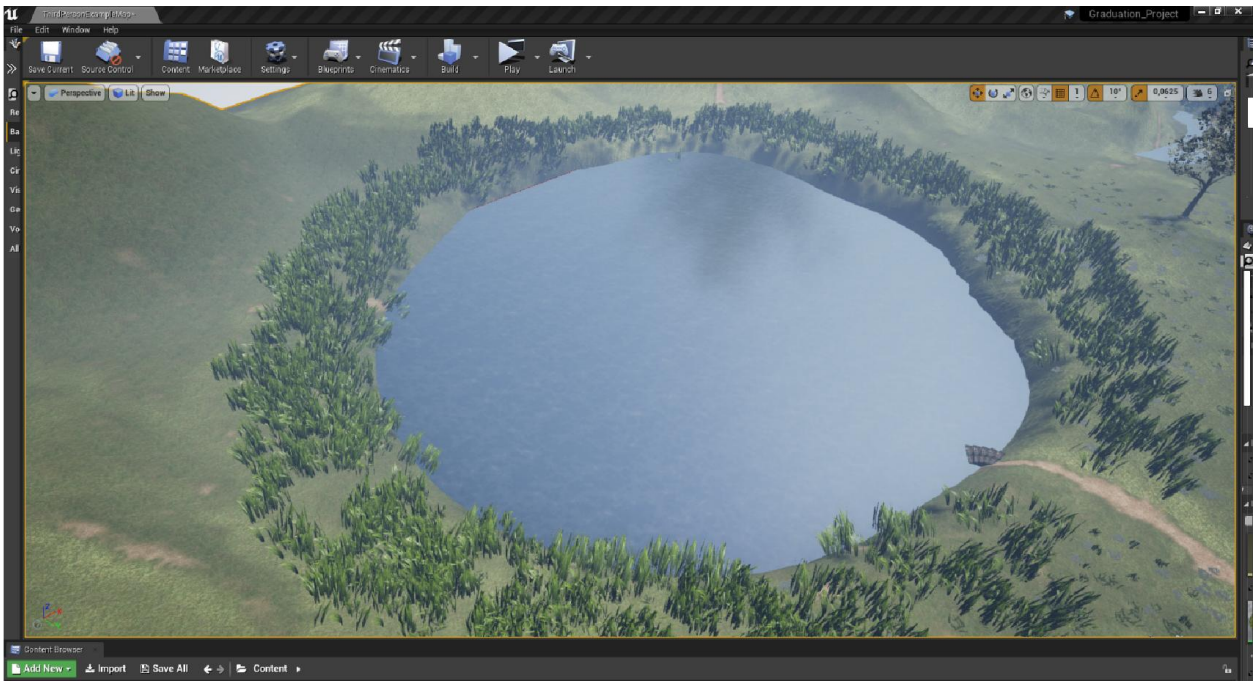


Рисунок 3.7 – Створення озера

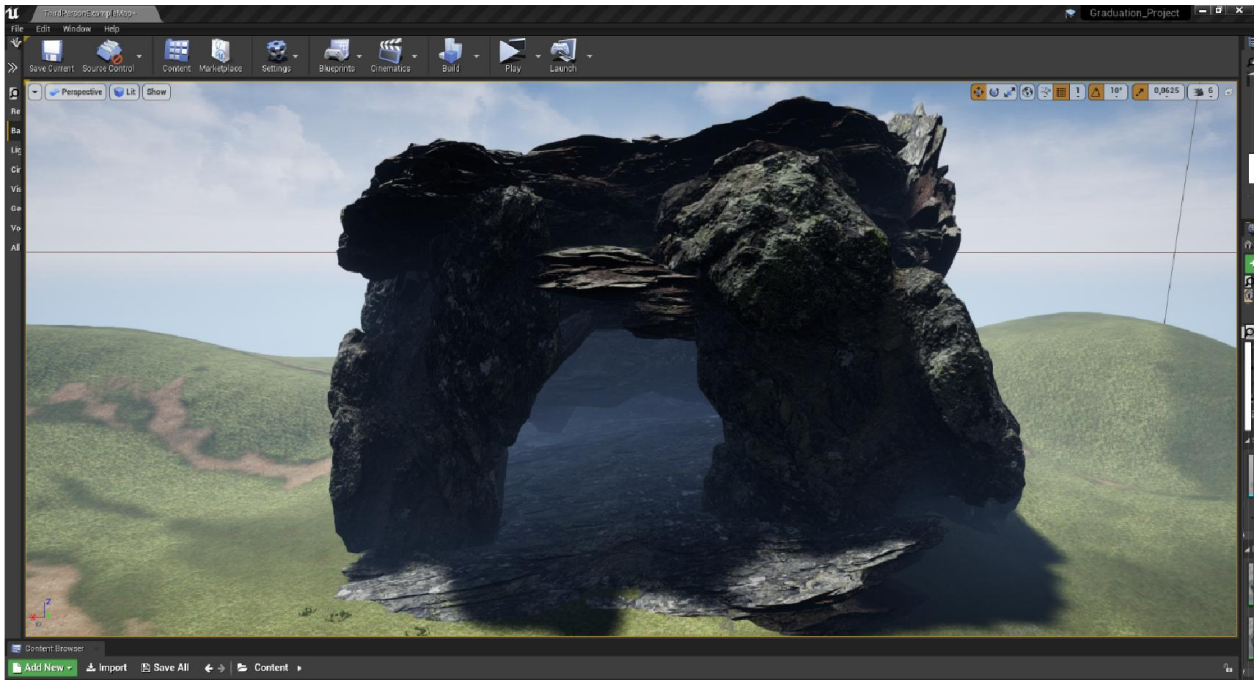


Рисунок 3.8 – Створення печери

Для переходу на наступний рівень гри, потрібно створити ігровий портал, котрий буде переносити головного персонажа в інший вимір, після створення порталу, додаємо його в непримітне місце на локації початкового рівня. На рисунках 3.9 – 3.10 зображено створення порталу та його розміщення на ігровому рівні.

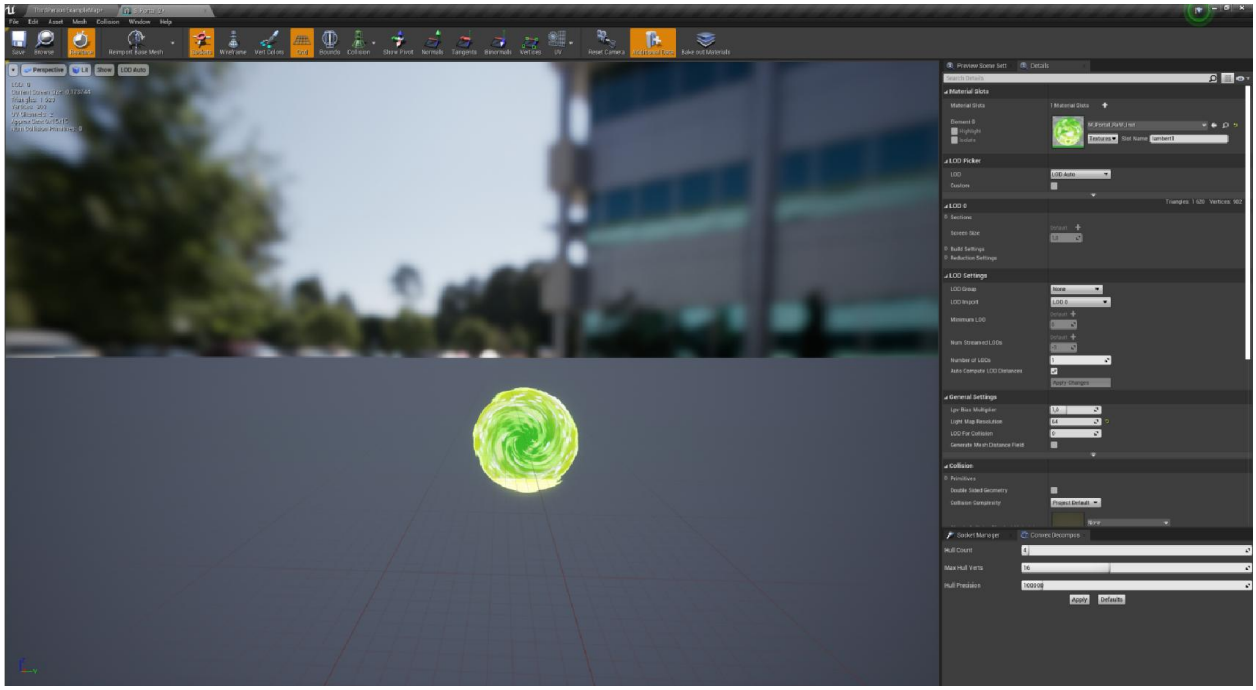


Рисунок 3.9 – Створення порталу

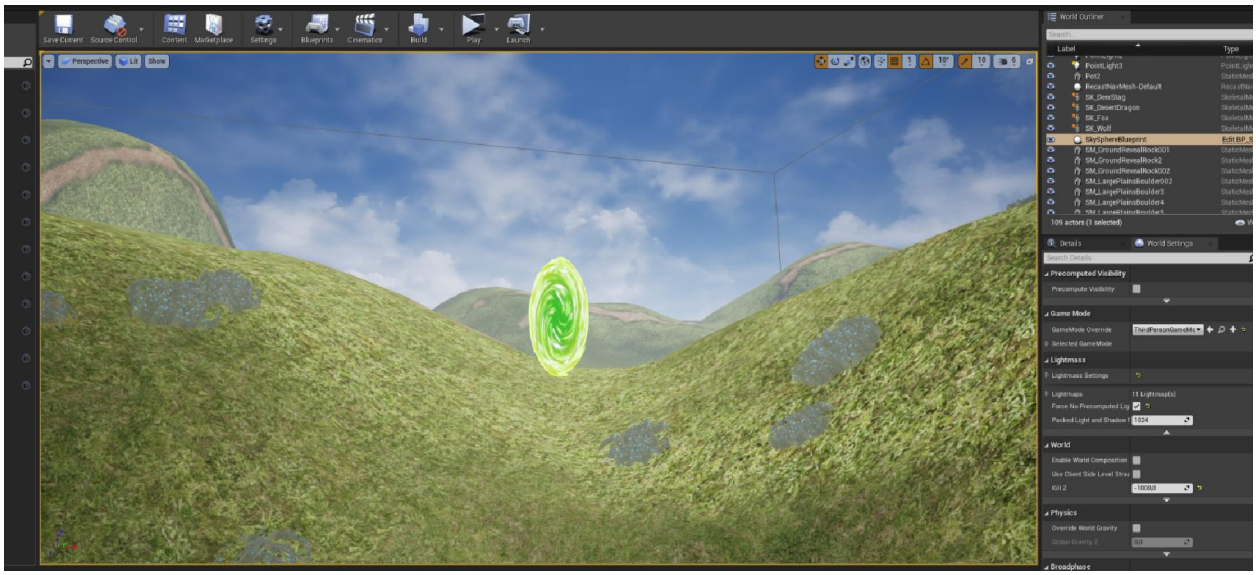


Рисунок 3.10 – Розміщення порталу на ігровому рівні

3.2 Створення інтерфейсу користувача

Практично всі відеоігри на сьогоднішній день мають особистий інтерфейс для своїх користувачів (за дуже рідкими виключеннями можуть не мати). Розробники відеоігор використовують графіку і текст для відображення необхідної інформації, наприклад, здоров'я або лічильника часу. Це називається інтерфейсом користувача (user interface, UI). UI в Unreal Engine 4 створюється за допомогою Unreal Motion Graphics (UMG). UMG дозволяє зручно вибудовувати UI, перетягуючи елементи UI, такі як кнопки і текстові мітки.

Якщо розглядати більш детально, то UMG (Unreal Motion Graphics UI Designer) – це візуальний інструмент для створення елементів призначеного для користувача інтерфейсу, таких як меню, HUD та інше, який відображається поверх всього іншого зображення гри. В основу UMG входять віджети (widget), які представляють собою набір готових функцій (кнопки, чекбокси, слайдери і т.д.), які можуть бути використані для створення інтерфейсу. Ці віджети редагуються в спеціальному редакторі під назвою «Widget Buleprint», який має два основних режими: дизайнер (designer), через який встановлюється візуальна складова інтерфейсу та

графік (graph), в якому проводиться створення логіки, за якою і буде працювати певний віджет.

Для цієї гри інтерфейс буде складатися з наступних частин:

- загальний інтерфейс (Main_HUD);
- віджет зміни дня та часу (Time_HUD);
- віджет здоров'я та мани (HP_MP_HUD);
- віджет плавного затухання екрану гри після відродження, у тому випадку якщо головний персонаж загине (Darkness_HUD);
- віджет, який повідомляє про смерть головного персонажа, а після натискання певної кнопки, відроджує його (Respawn_HUD).

Спершу, створимо загальний інтерфейс, який буде містити в собі подальші віджети, а також створимо логіку до нього.

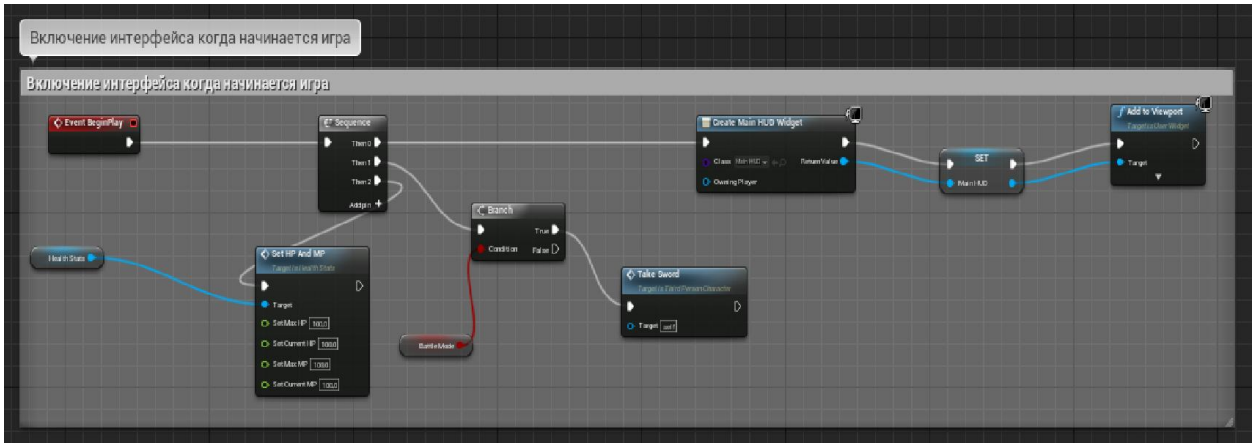


Рисунок 3.11 – Створення логіки ввімкнення загального інтерфейсу

3.2.1 Зміна дня та часу.Далі створюємо віджет зміни дня та часу, за допомогою blueprint класу який був названий TimeSystem, а також логіки в цілому. В цьому класі задаємо потрібний відрізок часу у шаблоні рухання сонця в грі (sun movement template), цей шаблон буде відображати певний час доби, тобто ранок, день, вечір та ніч.

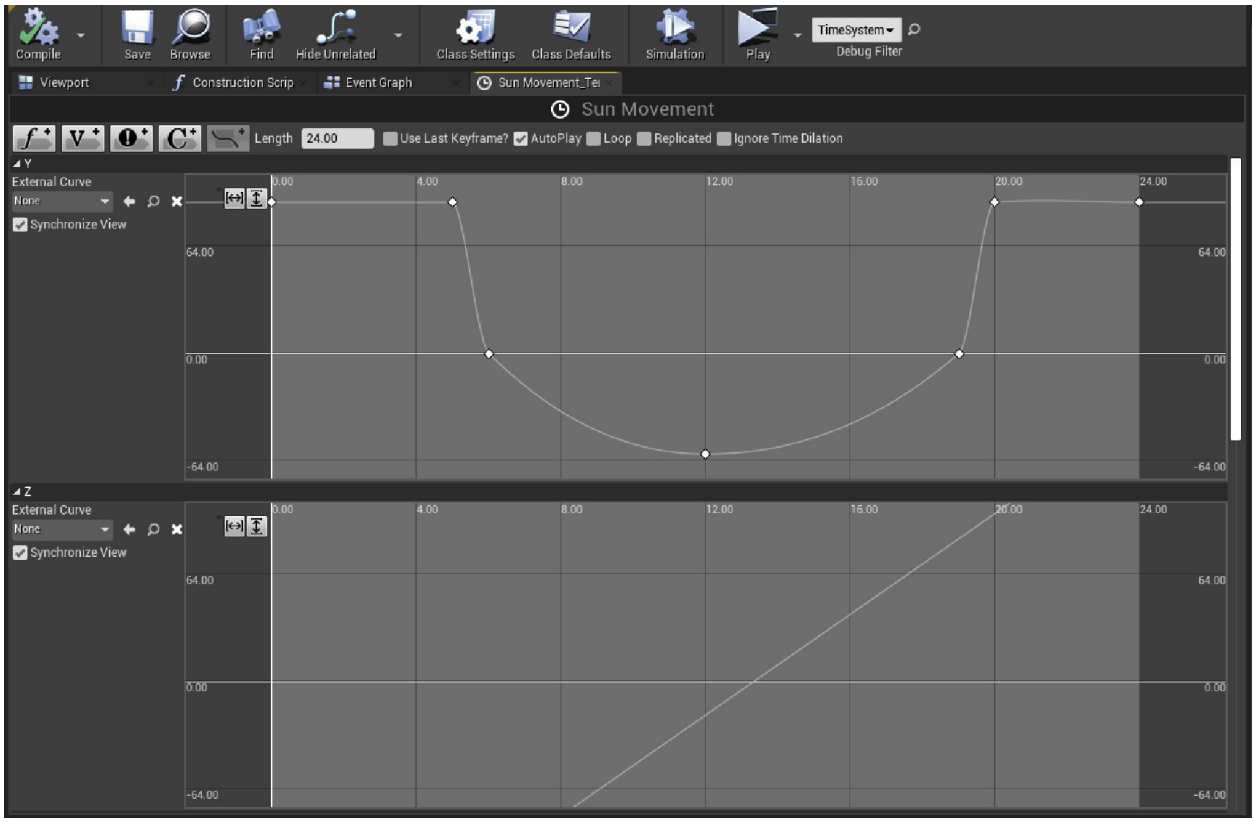


Рисунок 3.12 – Шаблон рухання сонця

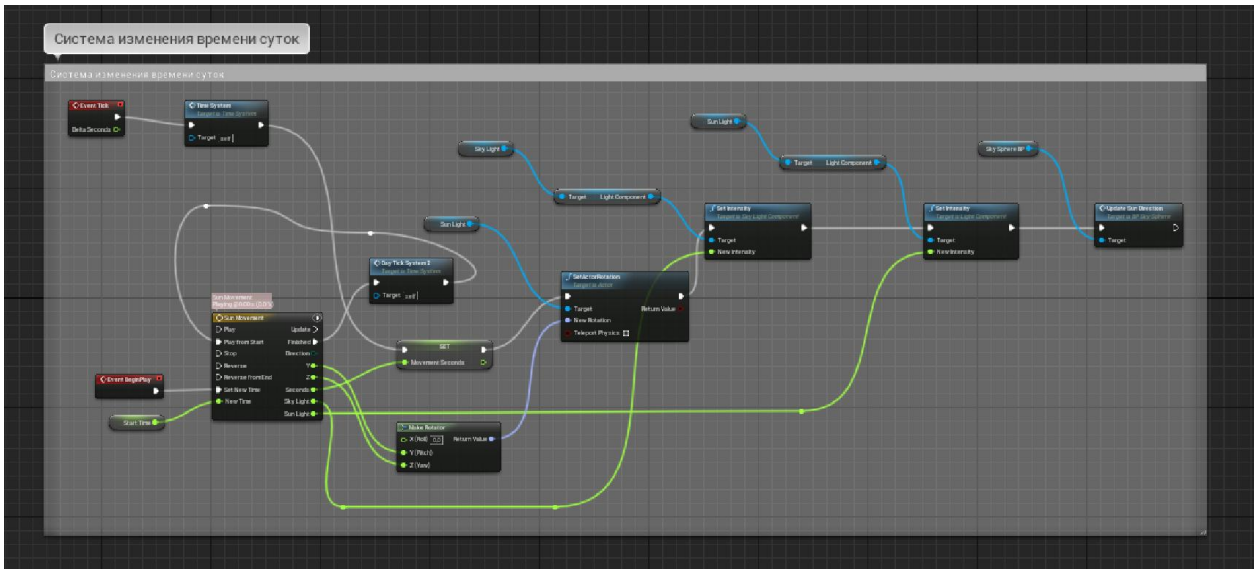


Рисунок 3.13 – Логіка зміни доби та часу

В заключенні, допрацювавши до кінця логіку зміни доби та часу, отримуємо функцію GetText, яка в свою чергу виводить цей віджет на екран гри.

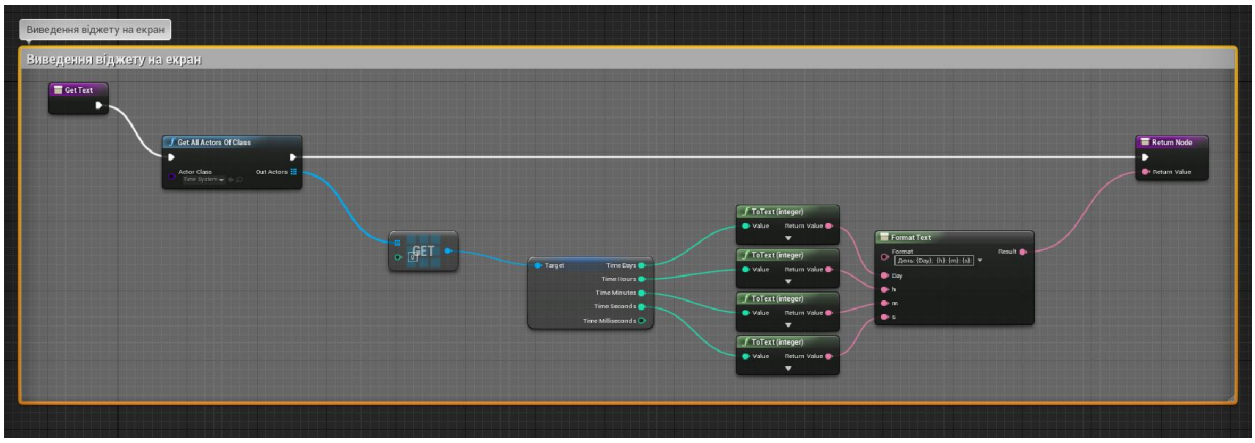


Рисунок 3.14 – Функція Get Text

3.2.2 Показники здоров'я та мани.Переходимо до створення віджету здоров'я та мани, щоб розробити цей віджет треба створити blueprint клас, який назвемо HealthStats та ряд змінних та функцій в ньому, для подальшої коректної роботи.

Перелік створених змінних:

- MaxHP – показник максимального здоров'я;
- CurrentHP – показник здоров'я в даний час;
- MaxMP – показник максимальної мани;
- CurrentMP – показник мани в даний час;
- IsDead? – змінна, яка перевіряє чи помер головний персонаж.

Перелік функцій:

- Get Stats for HUD – функція, котра отримує дані про здоров'я та ману;
- SetHPandMP – функція, котра задає параметри здоров'я та мани;
- DoDamage – функція, котра перевіряє чи отримує головний персонаж шкоду.

Об'єднавши всі ці змінні та функції, отримуємо поточну логіку даного віджету, для прикладу відобразимо декілька функцій та їх логіку, а також кінцевий вид віджету, які відображені на рисунках 3.15 – 3.17.

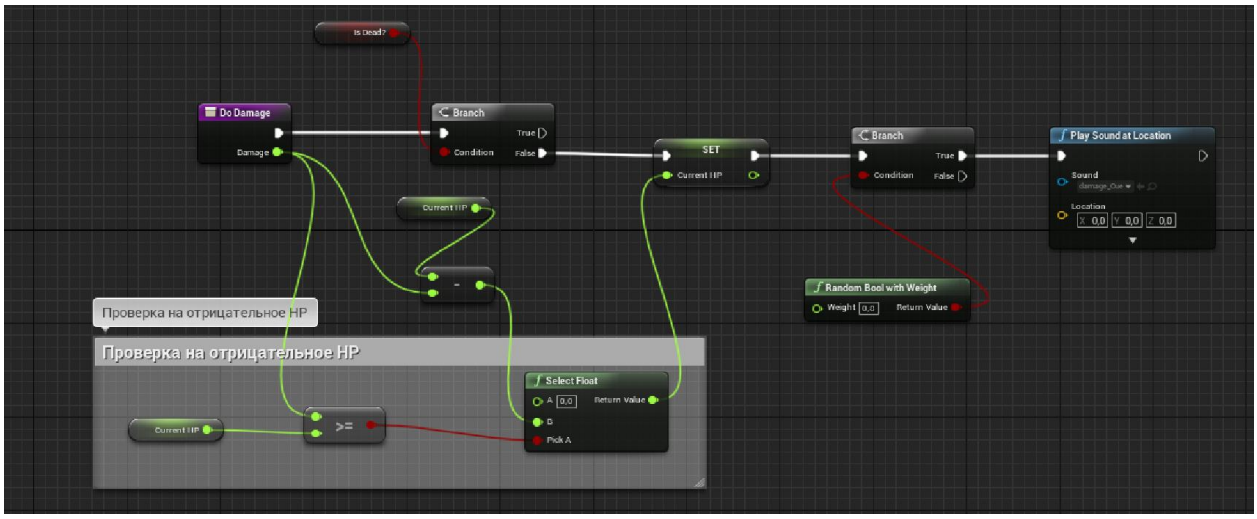


Рисунок 3.15 – Логіка функції Do Damage

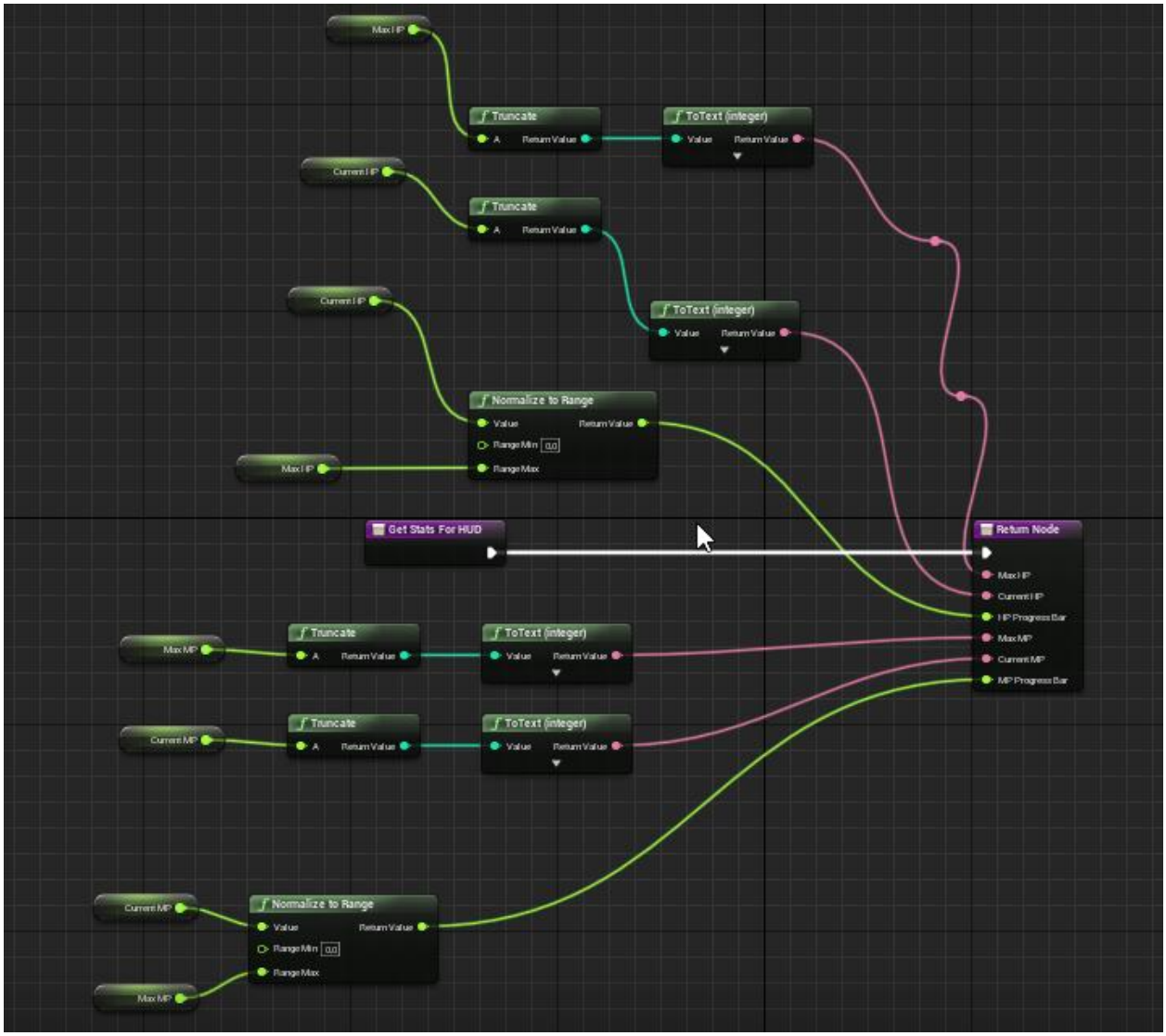


Рисунок 3.16 – Логіка функції Get Stats for HUD

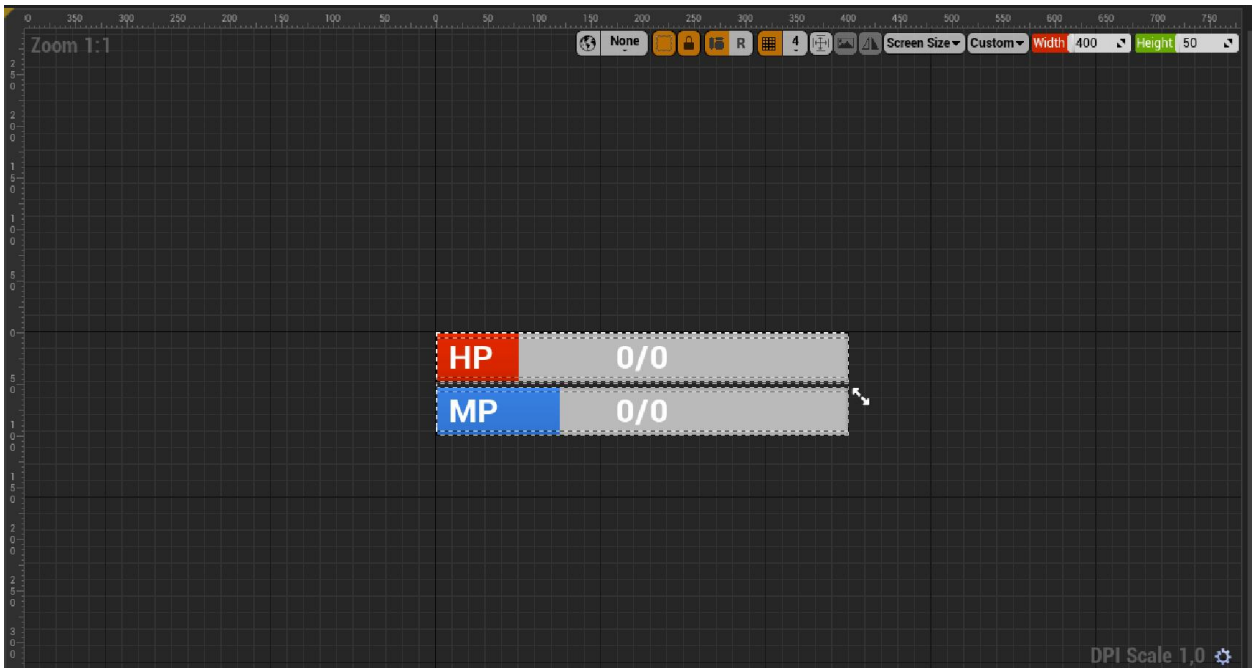


Рисунок 3.17 – Поточний вигляд віджету здоров’я та мани

3.2.3 Віджет затухання екрану та повідомлення про смерть ГП.Останнім кроком по створенню інтерфейсу користувача, буде створення віджету плавного затухання екрану після відродження персонажа, а також віджет повідомлення про смерть головного персонажа. Даний етап виявився найлегшим, все що потрібно – це створити анімацію затухання екрану для першого віджету, а після цього створити кнопку відродження для другого віджету. Також з’єднати все це розробленою логікою. На цьому завершився етап створення інтерфейсу користувача. Відобразимо останні віджети та кінцевий результат повного інтерфейсу на рисунках 3.18 – 3.22.

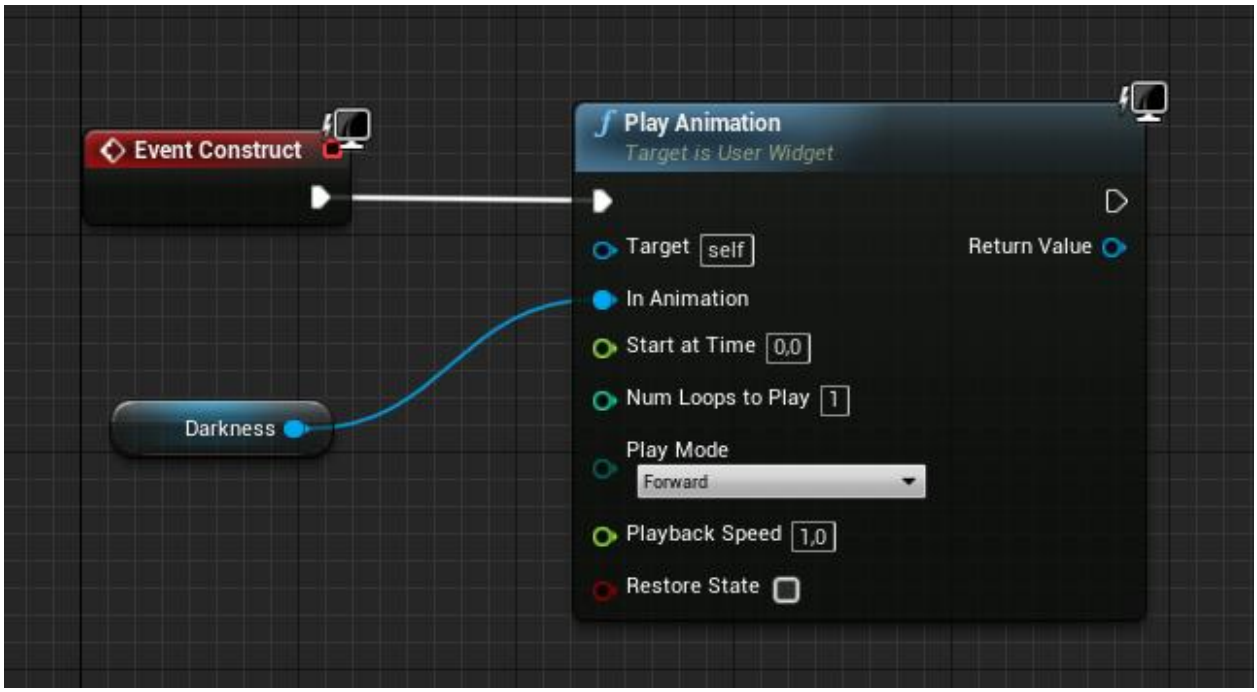


Рисунок 3.18 – Логіка анімації віджету затухання екрану

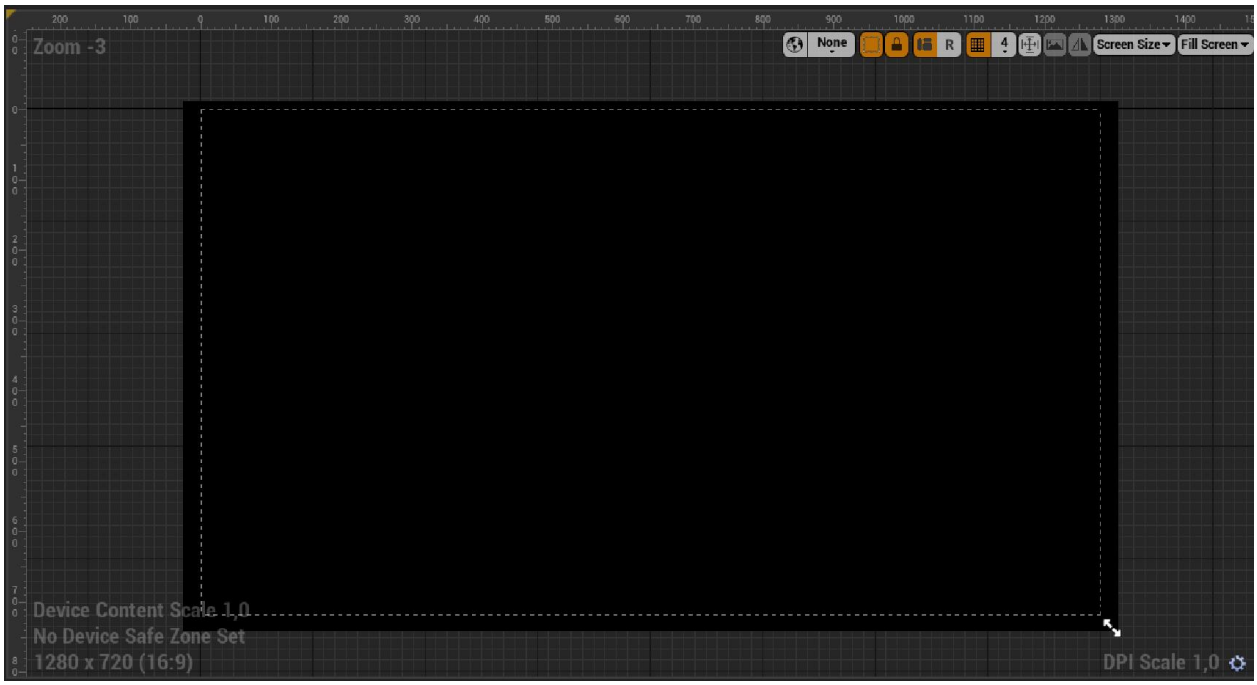


Рисунок 3.19 – Графічний вид віджету затухання екрану

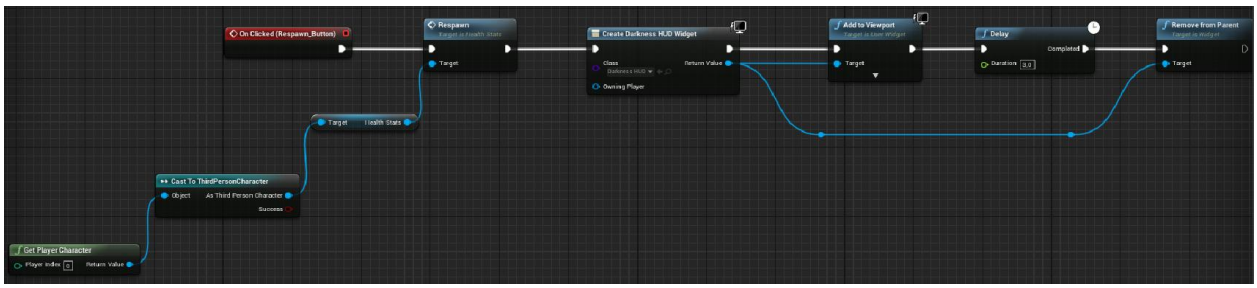


Рисунок 3.20 – Логіка віджету смерті головного персонажа

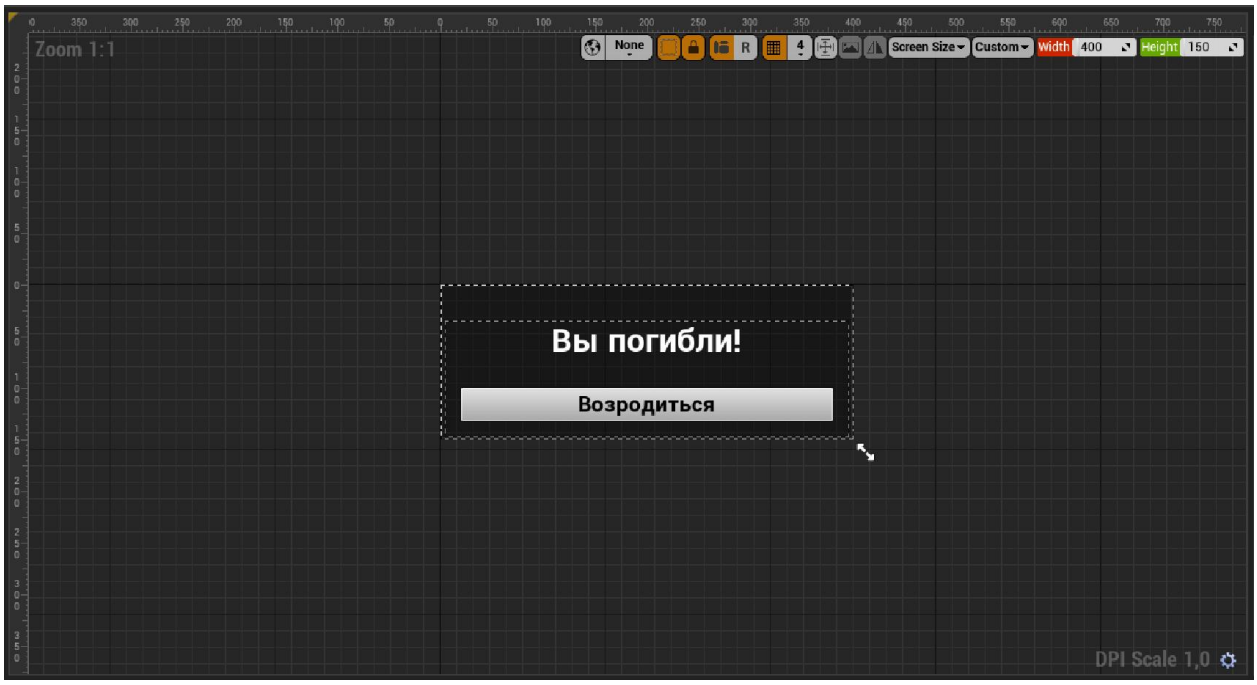


Рисунок 3.21 – Графічний вид віджету смерті головного персонажа

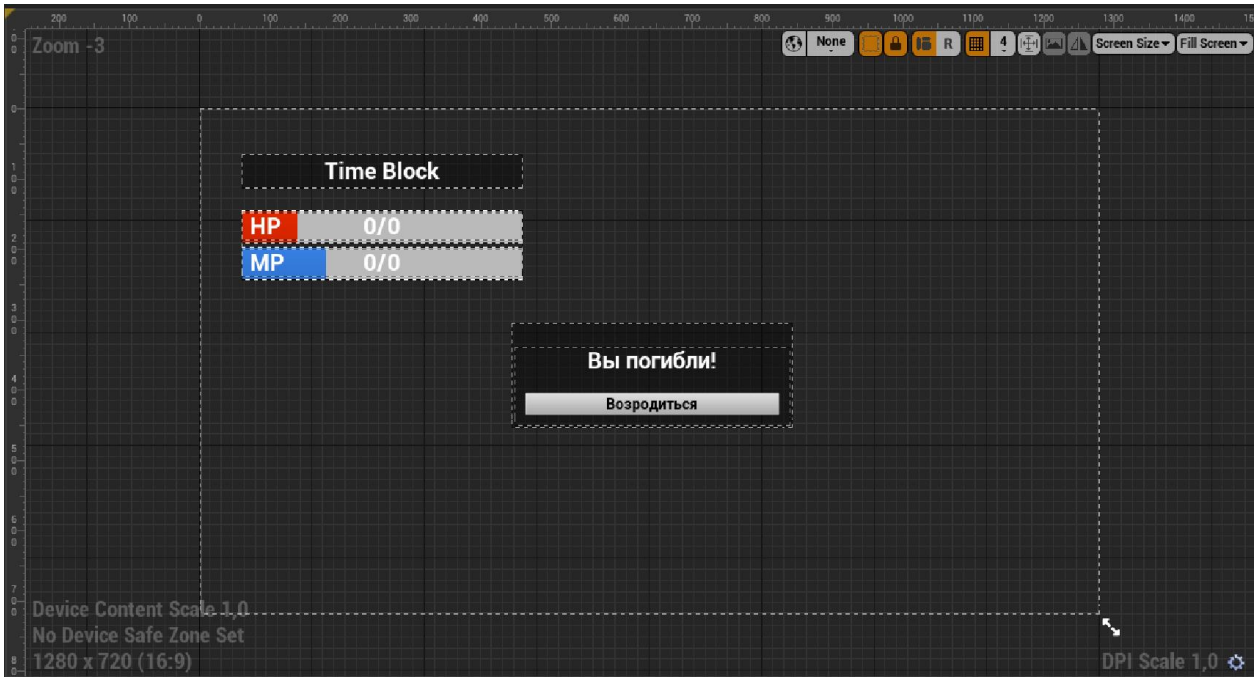


Рисунок 3.22 – Повний інтерфейс користувача

3.3 Створення головного персонажа

Створюючи персонажа, потрібно точно та чітко розуміти ким він буде не тільки зовні, але і внутрішньо: розробити манеру поведінки, характер. Внутрішній світ героя часто відображається у його зовнішності. Тому

потрібно визначитися з його особливостями. Наприклад, чи буде він героїчним? Або ж хитромудрим шахраєм? Або навіть демонічною істотою?

Для початку, виділимо основні етапи створення 3Dперсонажа в UE4:

- створення 3Dмоделі в програмах 3Dмоделювання (3DsMax, Cinema 4D, AutodeskMayата інші);
- текстурування 3Dмоделі (створення та накладання текстур);
- впровадження скелета в 3Dмодель персонажа;
- створення анімацій або набора анімацій для даного скелета;
- програмування анімацій, налаштування програвань анімацій відповідно до дій персонажа.

Також потрібно виділити способи отримання персонажа:

- автоматичне (взяття готового персонажа);
- напівавтоматичне (вибір із попередніх встановлень або моделей чи інструментів);
- ручне (створення моделі з нуля).

Для початкових розробників бажано обирати автоматичний або напівавтоматичний спосіб створення персонажа, тому для створення свого персонажа я обрав саме напівавтоматичний спосіб.Суть цього способу буде полягати в моделі персонажа та його скелеті, фізиці, а також в розробленні логіки анімацій. Першим кроком буде завантаження моделі персонажа в рушій UE4.

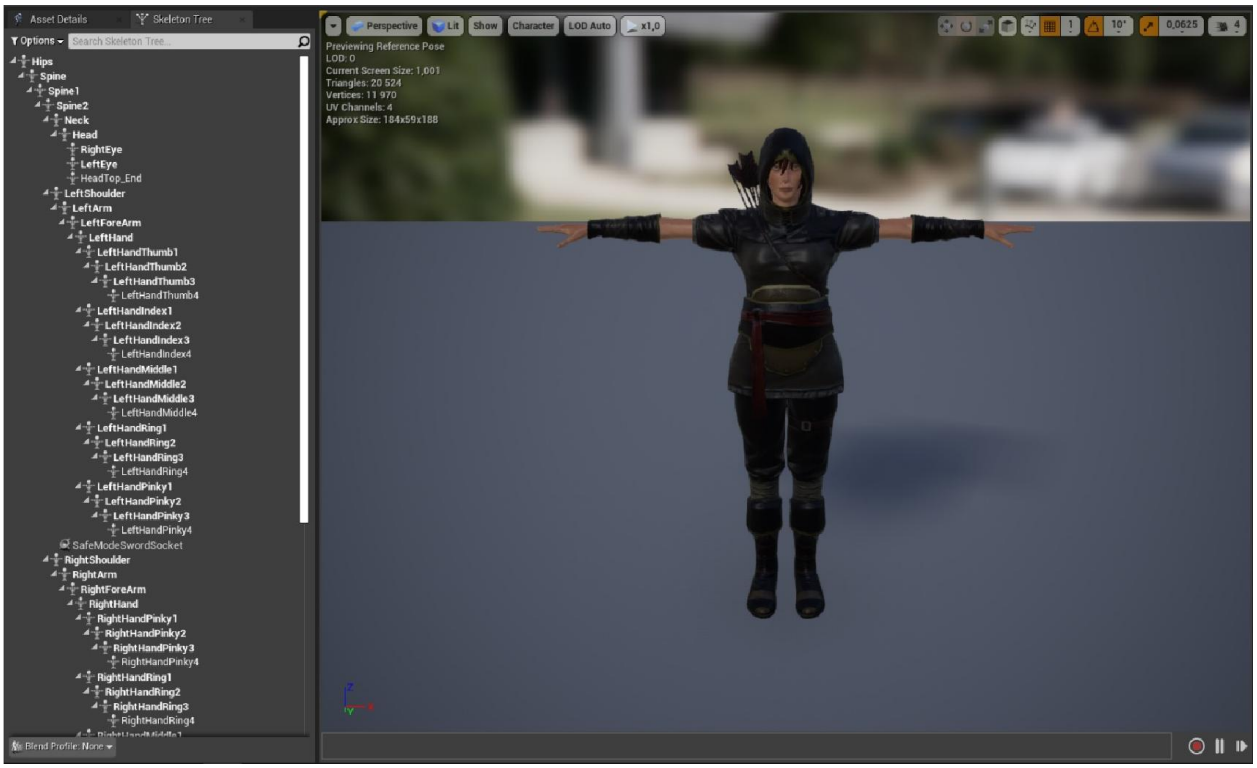


Рисунок 3.23 – Модель головного персонажа

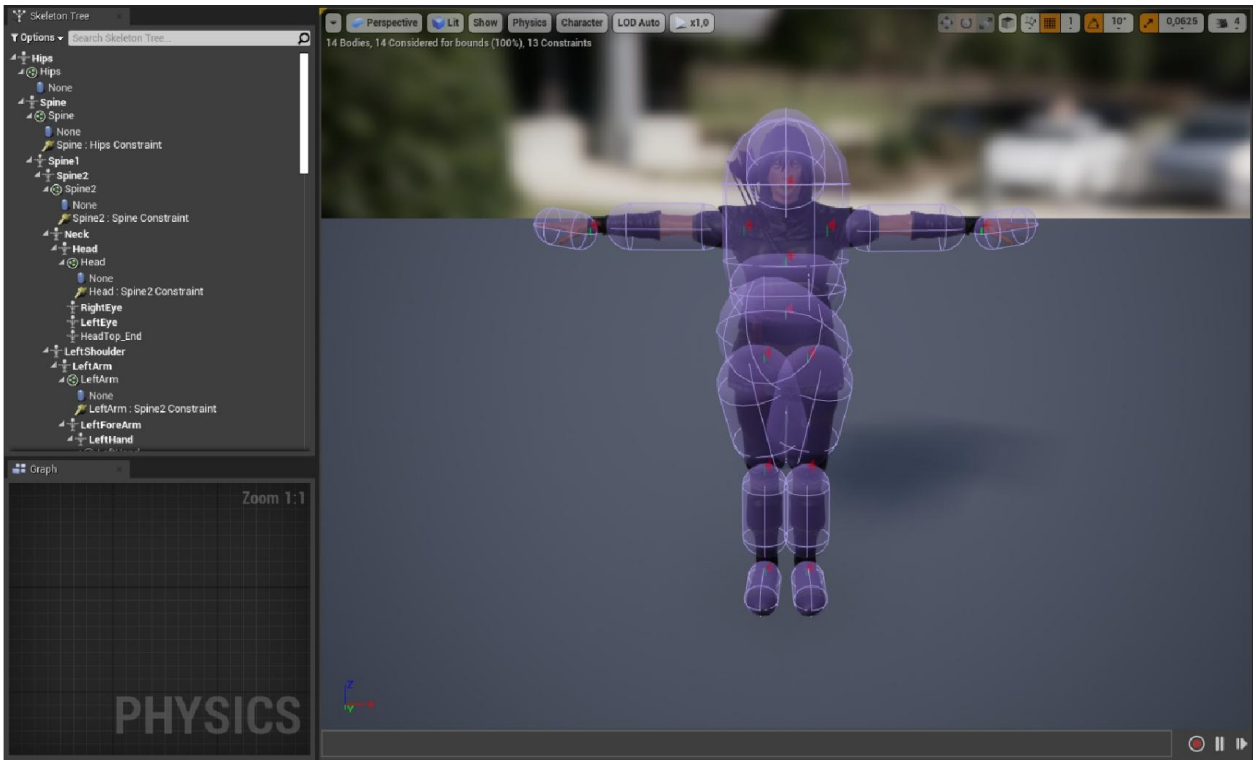


Рисунок 3.24 – Редактор фізики для головного персонажа

Наступним кроком, буде створення загальної логіки для персонажа. Перш за все потрібно створити в цілому логіку руху для головного персонажа.

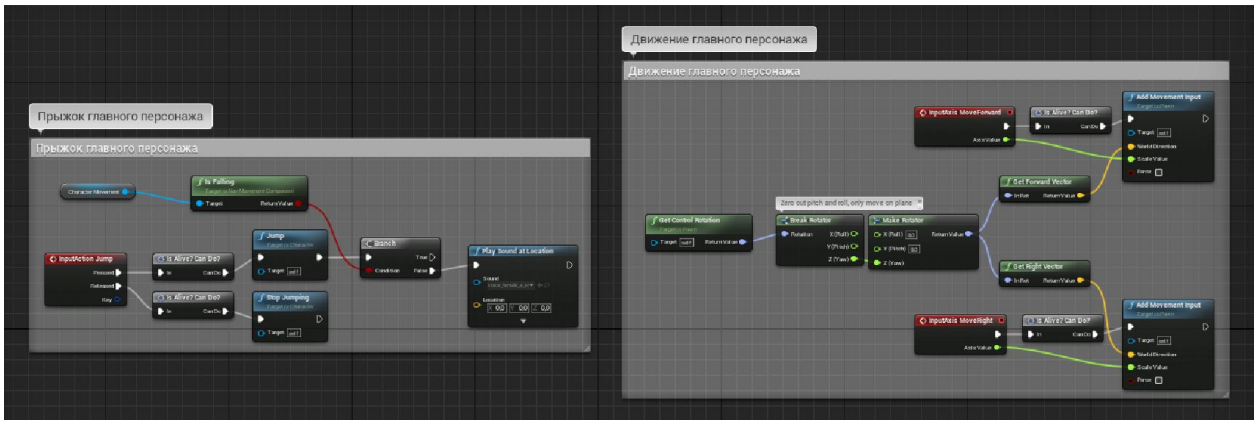


Рисунок 3.25 – Логіка руху для персонажа

Після цього, створюємо логіку перемикання між бігом та ходьбою персонажа за допомогою заздалегідь створених змінних, функцій та макросів, задавши кнопку LeftShiftна виконання цих дій.

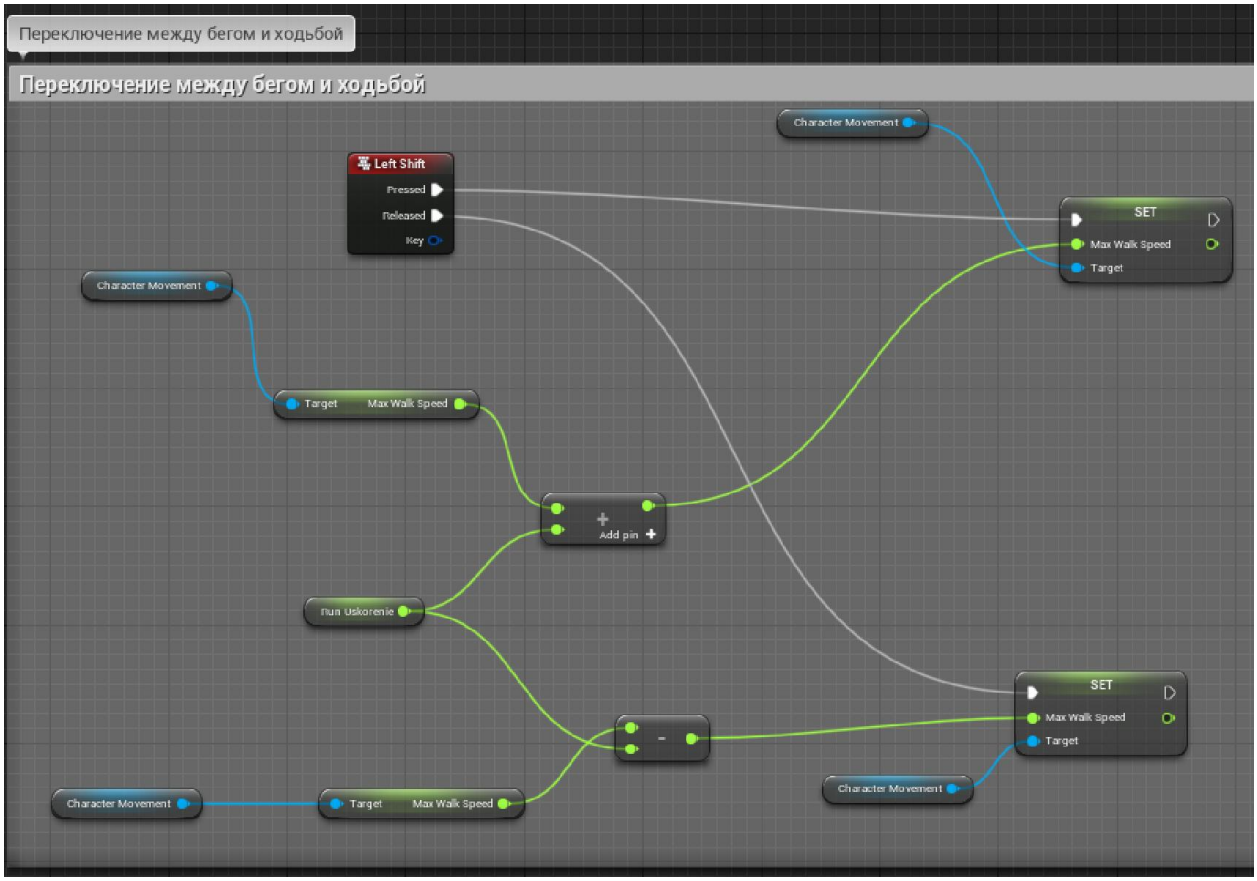


Рисунок 3.26 – Логіка перемикання між бігом та ходьбою персонажа

У грі для цього проєкту буде підтримка зміни відстані камери для персонажа, тобто буде можливість віддалити або приблизи камеру до головного персонажу. Тому, також створюємо логіку для даної опції.

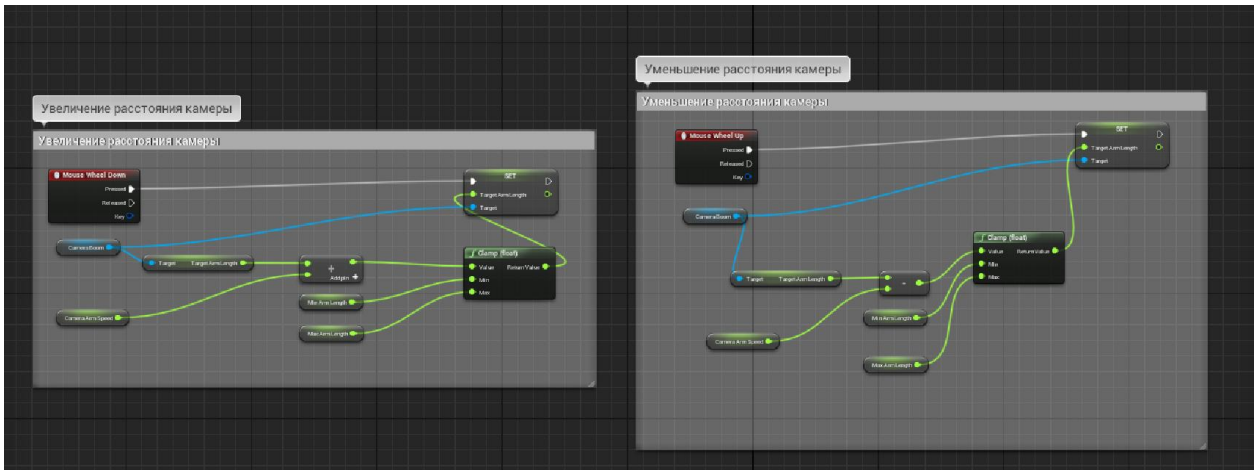


Рисунок 3.27 – Зміна відстані камери для персонажа

Перейдемо до анімацій головного персонажа, після того як були перенесені анімації до рушія UE4, потрібно їх коректно налаштувати. Для цього переходимо в модель персонажа, далі вмикаємо вкладку Animation. Декілька з налаштованих анімацій представлені на рисунках 3.29 – 3.32.

Рисунок 3.28 – Загальний список анімацій для персонажа

Рисунок 3.29 – Анімація ходьби

Рисунок 3.30 – Анімація бігу

Рисунок 3.31 – Анімація стрибка

Рисунок 3.32 – Анімація танця

3.4 Створення бойової системи

Важливою частиною Action-RPG є бойова система. В даному випадку, в залежності від обраного шляху розробки, бойова система може бути абсолютно різною. Складно дати визначення бойовій системі, але якщо говорити узагальнено, то ефективність битв в іграх залежить від характеристик і навичок персонажа. Це можуть бути різні фактори: кількість пошкоджень, шанс потрапляння, шанс критичного удару, можливість використання певного виду зброї і так далі.

Поверхнево, можна зупинитись на тому, що бойова система – частина правил рольової системи, присвячена опису бойових зіткнень.

Залежно від геймплейної частоти битв, бойова система може бути як ретельно пророблено, так і схематичною. У багатьох комп’ютерних іграх розділ бою може займати 50-60% обсягу.

Існує багато різноманітних механік бойової системи, але зазвичай до неї відноситься наступне:

- загальні принципи дії персонажів в бою, сюди входять як розширення і деталізації базового для системи способу розгляду заявок, так і

суто бойові розширення – наприклад, введення поля для тактичних маневрів;

- система пошкоджень;
- розширення загальних правил на бойові ситуації. Наприклад, правила з акробатики можуть бути частиною загальної механіки, а спеціальні правила по відходу від ударів перекатами або дертися по стінах під обстрілом – також є частиною бойової системи, так як не застосовуються ніде, крім бою;
- набір суворо бойових характеристик персонажів і предметів (зброї, броні, фантастичної техніки, імпровізованої зброї і т.д). Один і той же предмет може мати як небойові характеристики, так і бойові. Наприклад, для опису кирки як інструменту важливі додаткові бонуси на перевірку гірничої справи, знос при роботі і вага (для перенесення в інвентарі), а для бойового застосування важливіший показник нанесення шкоди, час замаху (швидкість удару) і прийоми.
- іноді, відносять спеціальні правила по противникам персонажів: генерацію монстрів, спрощені статистики для пересічних злодіїв та інше.

Щоб розробити бойову систему для даної гри потрібно буде виконати наступні дії:

- додати зброю для головного персонажа;
- створити логіку бойової системи(для головного персонажа);
- створити модель ворожого персонажа (моба) та його бойову систему;
- створити штучний інтелект для ворожого персонажа (AIController).

3.4.1 Додавання зброї для головного персонажа.Для додавання зброї була використана модель меча з офіційного маркету UnrealEngine. Після перенесення моделі меча в рушій, треба в моделі головного персонажа налаштувати його, тобто задати вірні координати та колізію для зброї.

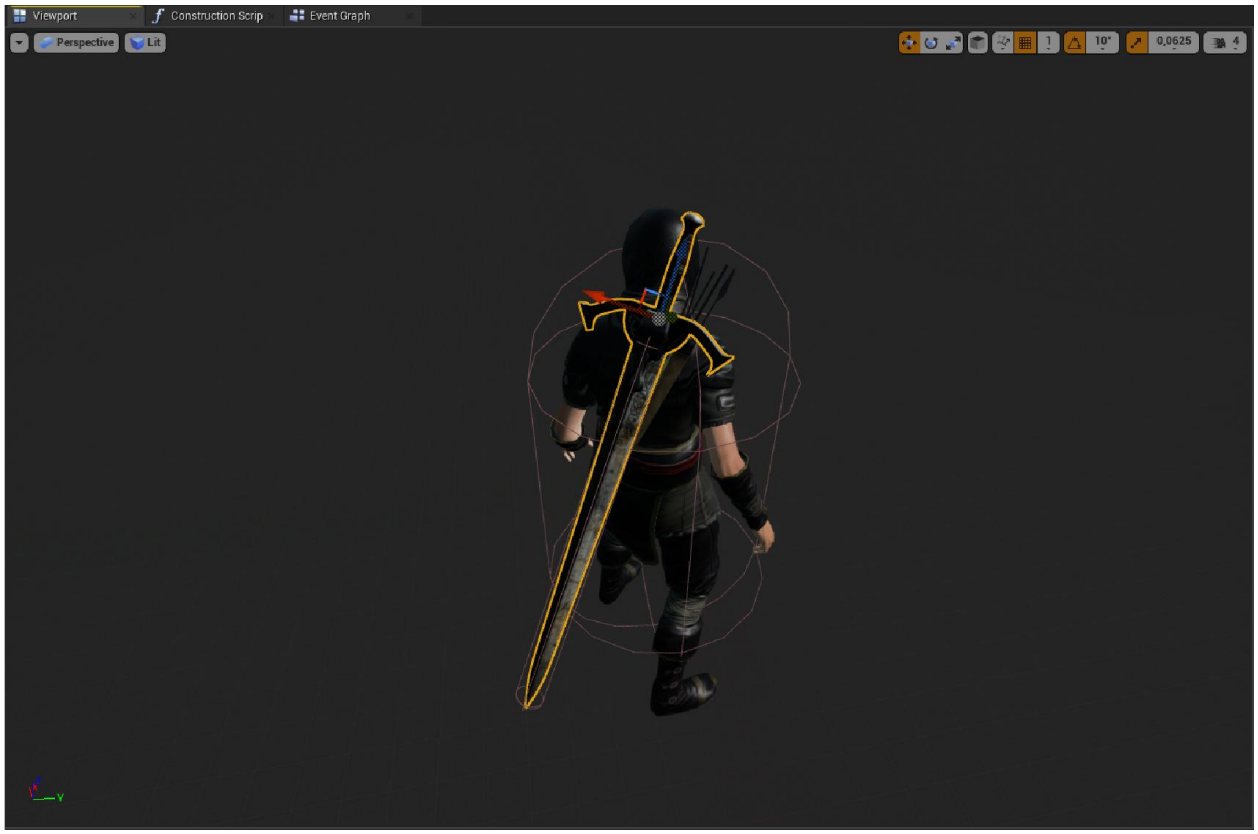


Рисунок 3.33 – Додавання зброї для головного персонажа

Після цього, була розроблена логіка для перемикання бойового режиму, яка була задана на кнопку «2». Для цього було створено 2 макроси: діставання та ховання зброї.

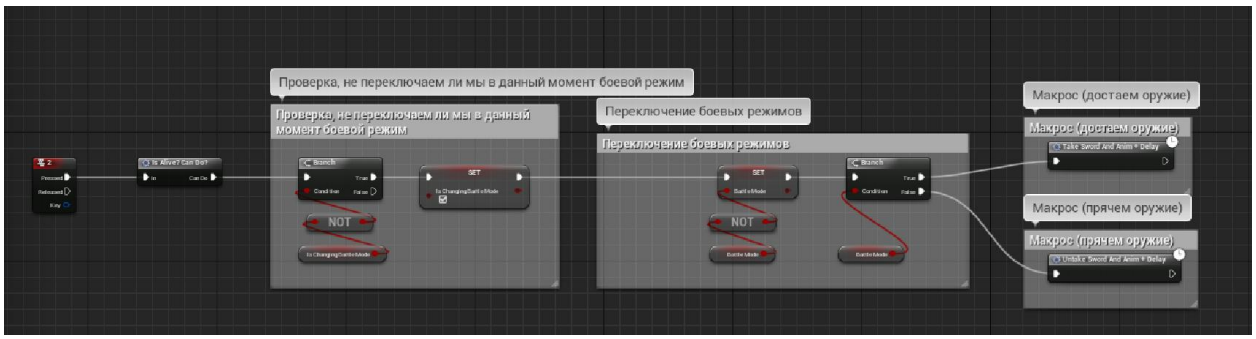


Рисунок 3.34 – Логіка для перемикання бойового режиму

3.4.2 Створення логіки бойової системи для головного персонажа.Спершу, треба задати кнопку удару мечем для нашого персонажа, для цього була використана кнопка «LeftMouseButton». Потім створюємо перевірку на знаходження зброї в руках, додаємо функцію випадкового удару зліва або справа, а також налаштовуємо відсоток шансу удару зліва або справа (після налаштування шанс удару злів – 30%, шанс удару справа – 70%,

сходячись на тому що головний персонаж правша).

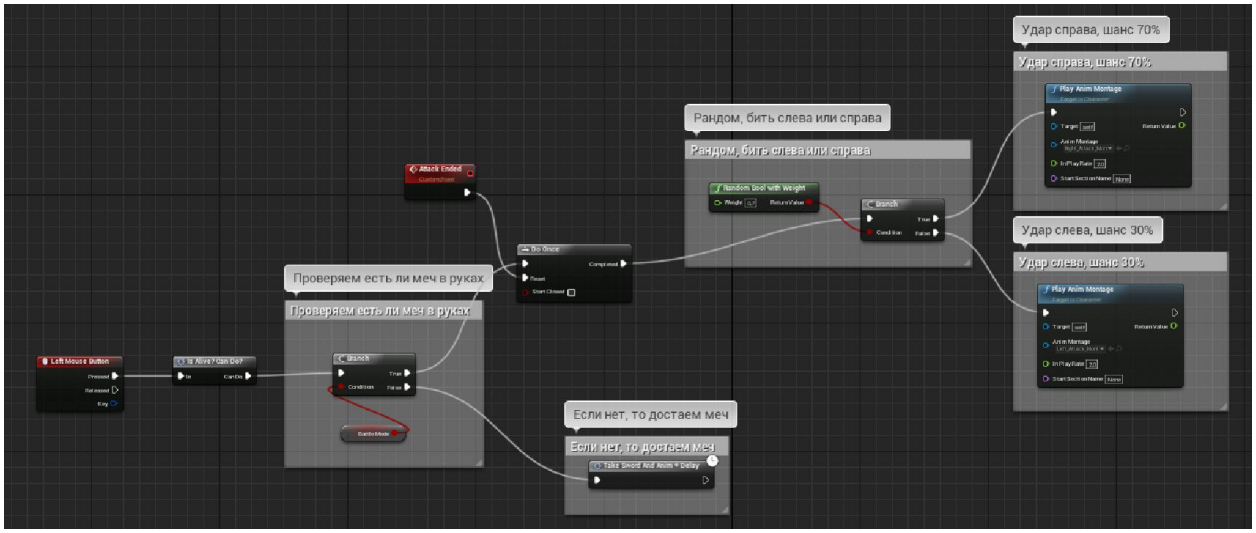


Рисунок 3.35 – Логіка для ударів з різних сторін

Потім, створюємо логіку для виключення шкоди головного персонажа по собі, а також додаємо блокування багаторазового нанесення шкоди для балансу в грі.

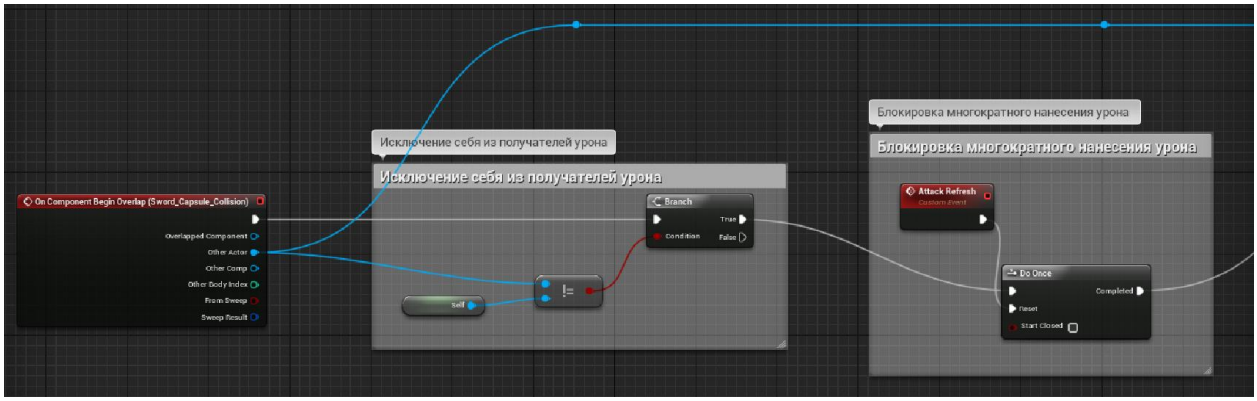


Рисунок 3.36 – Логіка для виключення шкоди по собі та блокування багаторазового нанесення шкоди

В завершенні, задаємо кількість нанесення шкоди нашим головним персонажем для майбутніх ворожих персонажів від 5 до 10 одиниць, а також створюємо шанс нанесення звичайного та критичного удару (шанс звичайного удару – 70%, шанс критичного – 30%), ця логіка відображена на рисунках 3.37 – 3.38.

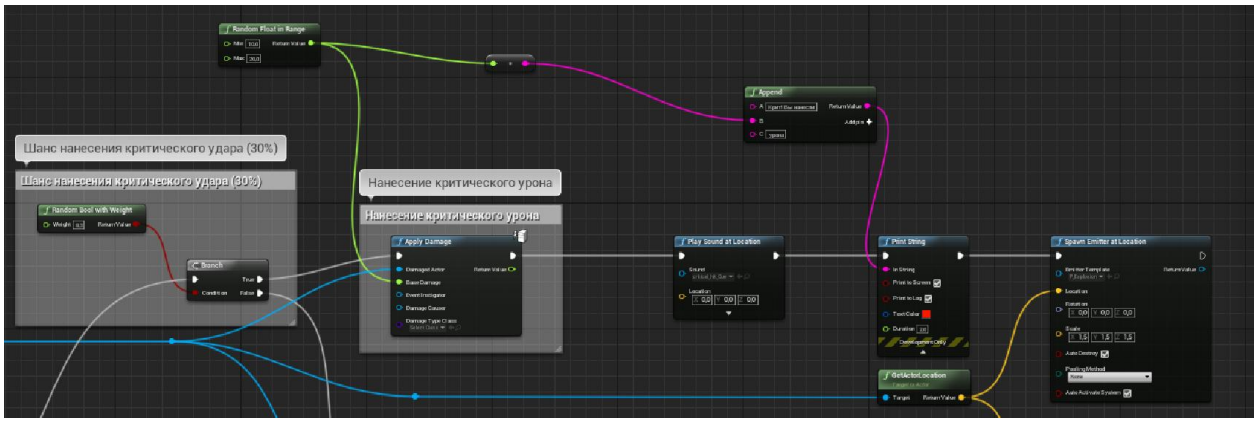


Рисунок 3.37 – Шанс нанесення критичного удару (від 10 до 20 одиниць шкоди)

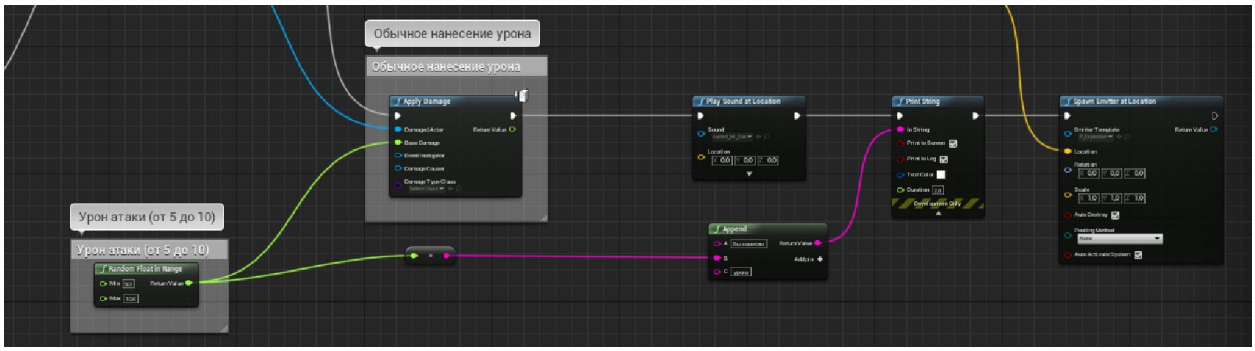


Рисунок 3.38 – Звичайний удар (від 5 до 10 одиниць шкоди)

3.4.3 Створення моделі ворожого персонажа (моба) та його бойової системи.Початкові кроки створення ворожого персонажа аналогічні створенню головного, різниця буде заключатися в зовнішньому виді та іншій логіки в цілому. Спершу створюємо модель і анімацію ворожого моба та завантажуємо їх у рушій. Далі створюємо логіку, яка буде складатися з наступних складових: віджет здоров’я, максимальна швидкість руху, атака, нанесення шкоди, отримання шкоди. Щоб реалізувати всю цю логіку, потрібно створити наступні змінні:

- MaxHP – максимальная кількість здоров’я у ворожого моба;
- CurrentHP – поточна кількість здоров’я у ворожого моба;
- MobName – ім’я ворожого моба;
- IsDead? – перевірка на смерть ворожого моба;
- StartMaxSpeed – максимальна швидкість руху ворожого моба;

- EnemyRotation – рух ворожого моба по ігровому оточенні.

Модель ворожого персонажа та кілька систем його логіки бойової системи зображені на рисунках 3.39 – 3.42.

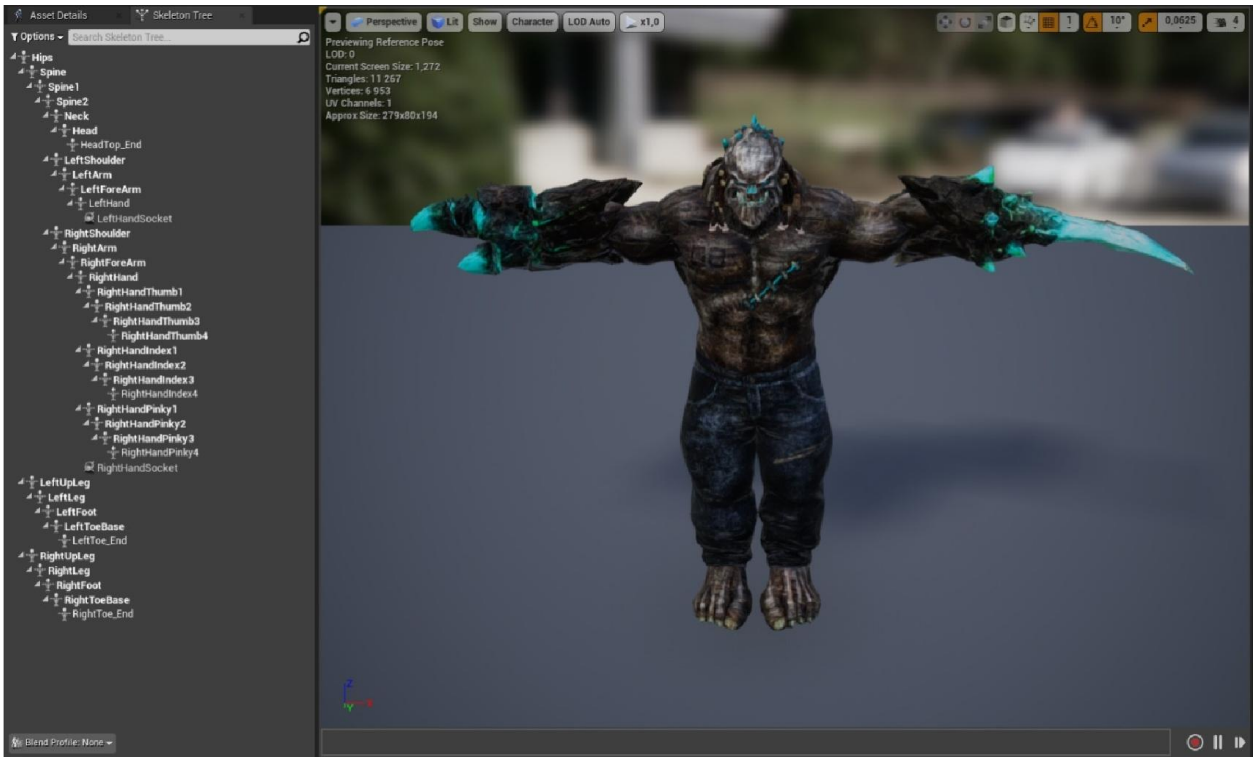


Рисунок 3.39 – Модель ворожого персонажа

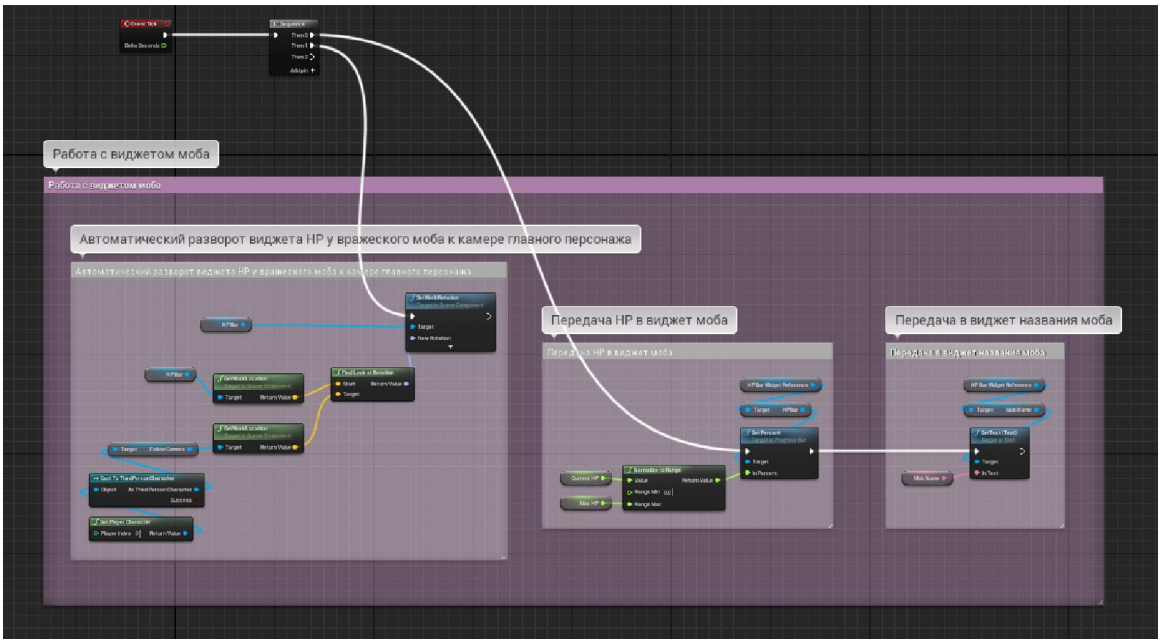


Рисунок 3.40 – Логіка віджету ворожого персонажа

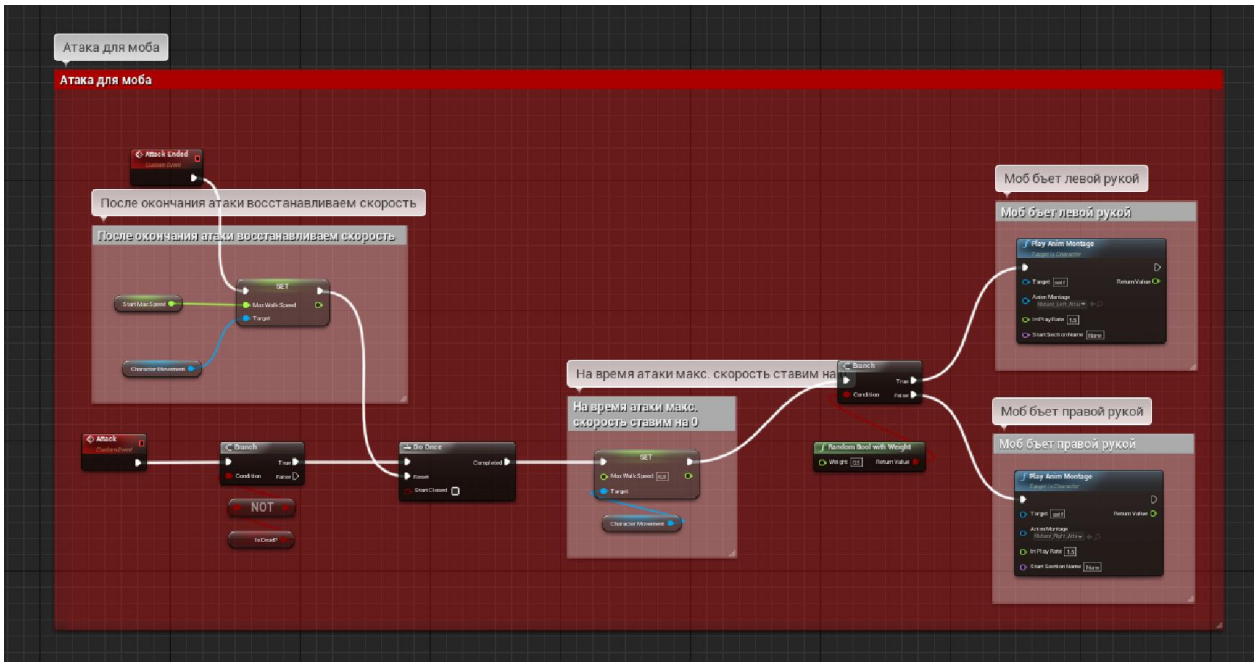


Рисунок 3.41 – Логіка атаки для ворожого персонажа

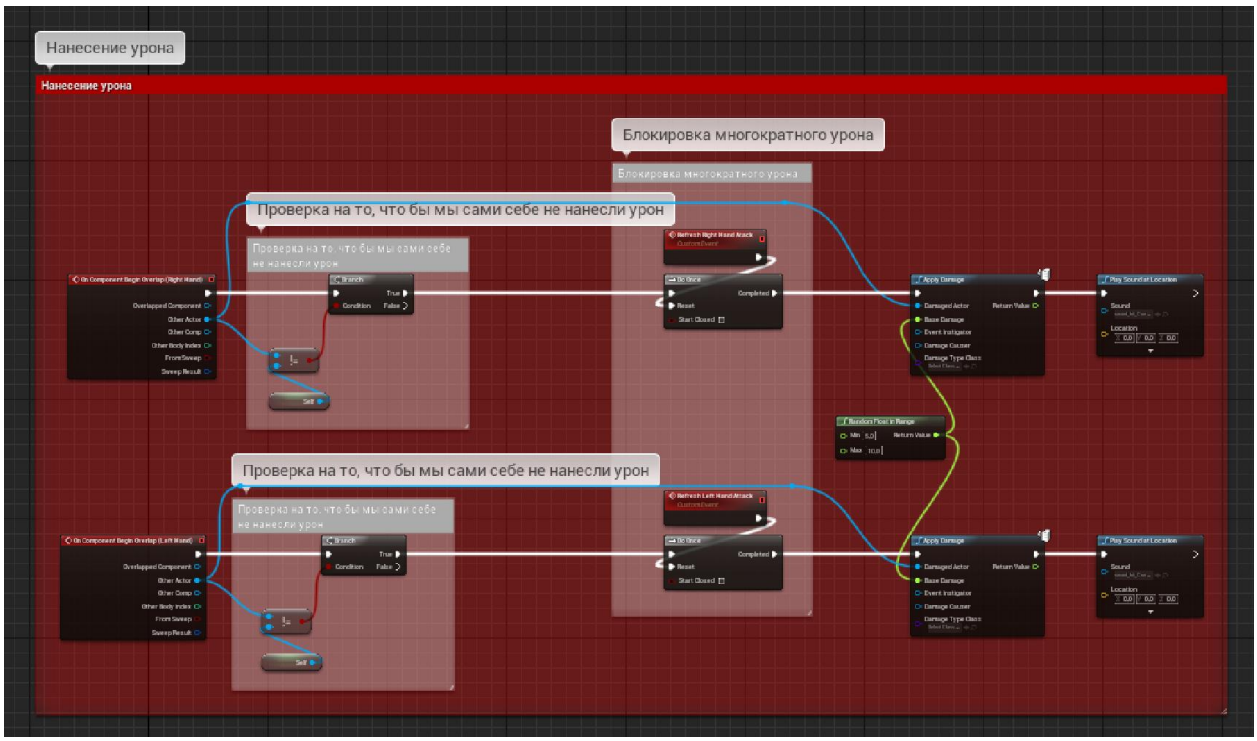


Рисунок 3.42 – Нанесення шкоди ворожим персонажем

3.4.4 Створення штучного інтелекту для ворожого персонажа (AI Controller).В індустрії відеоігор штучним інтелектом (Artificial Intelligence, AI) зазвичай називають процес прийняття рішень не керованими гравцем персонажами. Він може бути простим: ворог бачить гравця і атакує. Або ж

більш складними, наприклад, керований штучним інтелектом противник в стратегії реального часу.

Персонажів комп'ютерних ігор, керованих ігровим штучним інтелектом, ділять на:

- неігрові персонажі (Non-player character, NPC) – як правило, ці персонажі є дружніми або нейтральними до гравця;
- боти (Bot) – ворожі до гравця персонажі, що наближаються за можливостями до ігрового персонажу. Проти гравця в будь-який конкретний момент борються невелика кількість ботів. Боти найбільш складні в програмуванні;
- моби (Mob) – ворожі до гравця «низькоінтелектуальні» персонажі. Моби вбиваються гравцями в великих кількостях заради очок досвіду, артефактів або проходження території.

Для цього проєкту, реалізуємо доволі простий штучний інтелект для ворожого моба. Логіка AIбуде реалізована через так званий AIController, який вбудований в рушій UE4. А саме, якщо моб живий, він буде постійно перевіряти відстань до ворожого персонажа, а якщо ворожий персонаж буде на відстані далі ніж ми вказали в параметрі радіусу агресивності моба або персонаж буде в радіусі, але буде мертвим, то моб буде кожен заданий проміжок часу вибирати випадкову точку в заданому радіусі від місця початкового відродження і йти туди. Таким чином моб патрулюватиме територію.

Як тільки наш персонаж увійде в зону агресивності моба, моб відразу ж відреагує агресивно і почне переслідувати головного персонажа, а якщо він його наздожене, то неодмінно почне атакувати. У тому випадку якщо моб вб’є головного персонажа, він зрозуміє, що він мертвий і повернеться далі патрулювати територію. Функціональна частина логіки буде містити в собі декілька певних змінних та макрос під назвою «AIlookingforenemy», який виконує основну логіку пошуку головного персонажа.Декілька прикладів системи кінцевої логіки штучного інтелекту для ворожого персонажа

наведені на рисунках 3.43– 3.44.

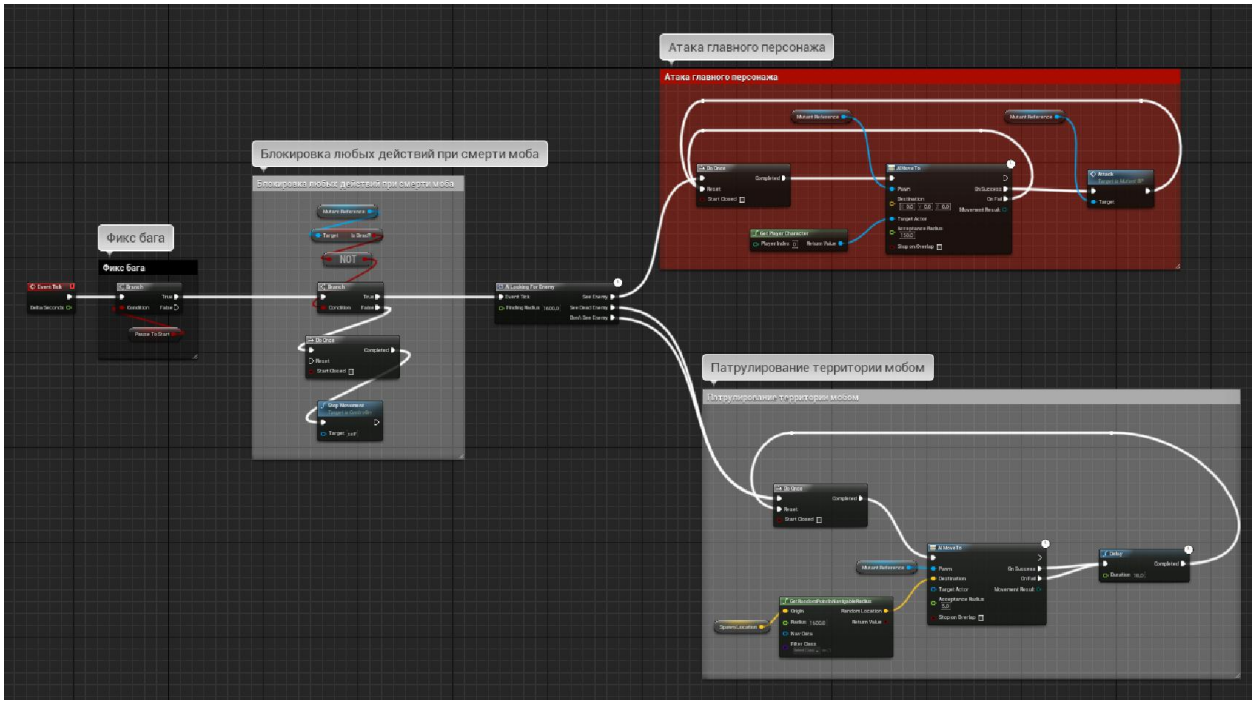


Рисунок 3.43 – Основна логіка штучного інтелекту ворожого персонажа

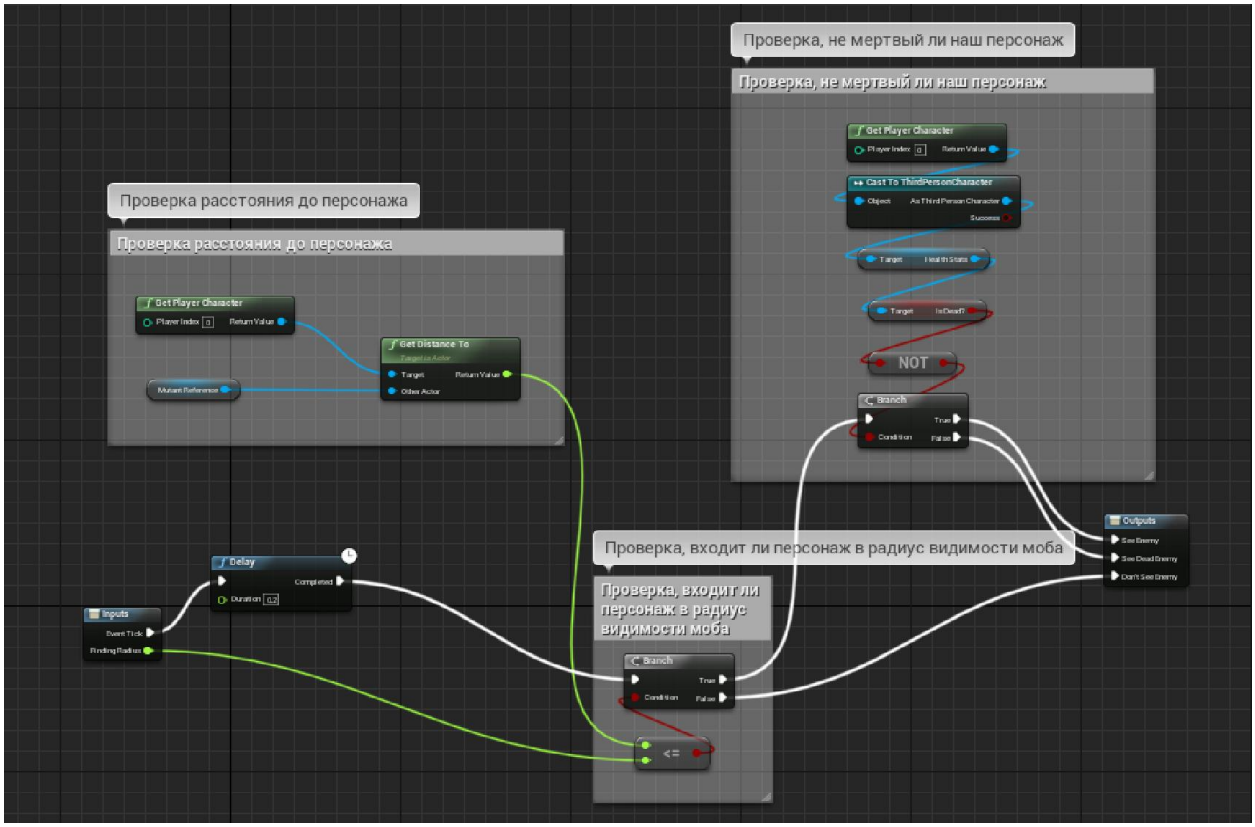


Рисунок 3.44 – Логіка макросу «AIlookingforenemy» для пошуку головного персонажа мобом.

3.5 Створення фонової музики та інших звуків

При розробці комп’ютерних ігор звуковий супровід зазвичай знаходився на другому плані. Багато розробників вважають за краще витратити час на введення новомодних текстур і ефектів для 3D-графіки, тоді як реалізація звуку пускається на другий план. Людей, які не розуміють важливість звукового супроводу, складно переконати витратити час і кошти на якісний звук в грі. Разом з тим, більшість користувачів також більш охоче витратять гроші на новітню відеокарту, ніж на нову звукову карту.

Однак, ситуація змінюється: останнім часом звуку стало приділятися набагато більше уваги і з боку користувачів, і з боку розробників. У сучасних проєктах звуку відводиться до 40 відсотків бюджету і людських ресурсів. Виробники звукових чіпів і розробники технологій 3D-звуку також доклали чимало зусиль, щоб переконати користувачів і розробників додатків в тому, що хороший 3D-звук є невід'ємною частиною сучасного комп'ютера.

Саме поняття «тривимірний звук» має на увазі те, що джерела звуку розташовуються в тривимірному просторі навколо слухача. При цьому кожне джерело являє собою в широкому сенсі будь-який об'єкт у віртуальному ігровому світі, який здатний виробляти звуки.

3D-звук потрібен для більш глибокого занурення користувача в віртуальний світ гри, за рахунок чого посилення реалізму відбувається прямо на ігровій сцені. Для цього використовуються різні технології, адаптовані під поведінку звуку в реальному світі. Наприклад, відбиті звуки, оклюзії (звук, що пройшов через перешкоду), обструкції (звук не пройшов через перешкоду), дистанційне моделювання (вводиться параметр віддаленості джерела звуку від слухача) і маса інших ефектів.

Для цього проєкту, структура створеного звуку реалізована з таких частин:

- фонова музика, тобто додано декілька аудіо треків у гру, які будуть чергуватися між собою;

- звуки природи – спів птахів днем і звуки цвіркунів вночі, тобто коли на дворі буде день співатимуть птахи, а коли почнеться ніч спів птахів закінчиться і почнуться звуки цвіркунів;
- звуки води, які капають в печері – при вході в яку музичний супровід буде затихати, тобто коли головний персонаж буде заходити в печеру, фонова музика буде припинятися, а звук крапель води буде починатися, при виході з печери – навпаки;
- звуки кроків, стрибків, зброї, персонажів.

Загальна логіка створення звуку у грі для даного проєкту базується на системі, яка складається з вбудованих засобів програвання музики та звуків в рушії UnrealEngine 4. Приклад розроблених частин логіки для основних звуків та музики відображений на рисунках 3.45 – 3.46.

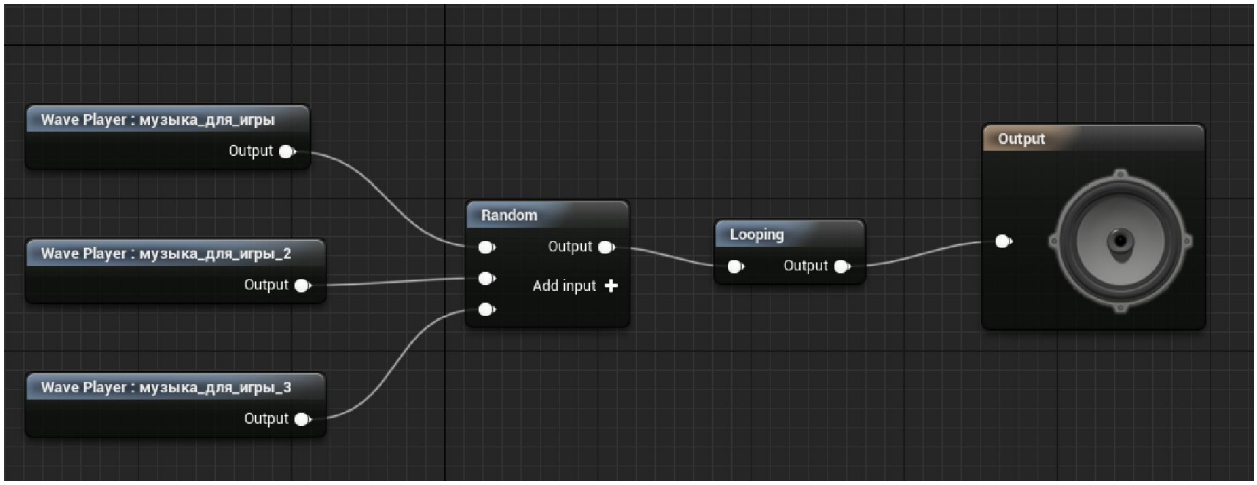


Рисунок 3.45 – Логіка програвання фонової музики

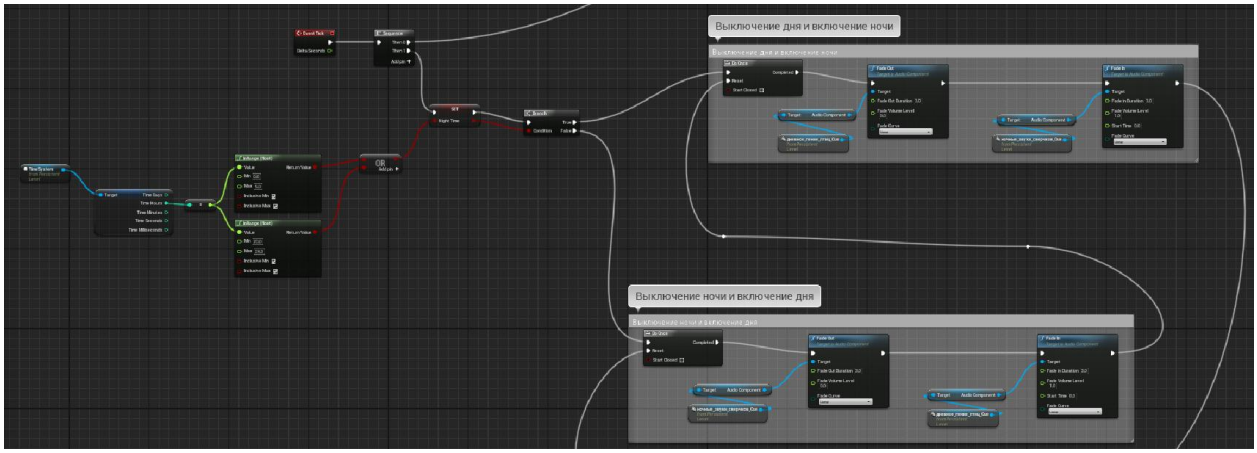


Рисунок 3.46 – Включення та виключення дня та ночі (для розуміння коли буде програватись спів птахів, а коли звук цвіркунів)

3.6 Створення штучного інтелекту для комплексної системи завдань (квестів)

Перед початком створення системи квестів, потрібно зрозуміти що собою являє саме поняття «квест». Чіткого визначення «що таке квест?» не існує, і не дивно, адже квестом можна назвати будь-яке завдання, прохання, або велику частину геймплею будь-якої комп'ютерної гри.Багато фахівців з розробки відеоігор трактують це поняття по-своєму. Але якщо узагальнити, квест – це завдання, яке стоїть перед гравцем. Воно має бути формалізоване, тобто у зрозумілій формі донесено гравцеві: через NPC, що дає завдання, через квест бук(журнал квестів), у ході катсцени, у початкових титрах до рівня... взагалі як завгодно, головне, щоб гравець його отримав і зрозумів, що повинен робити та яка його ціль.

Кожен квест передбачає якийсь геймплей, адже поставлене завдання потрібно вирішити та досягти мети. Іноді цей геймплей є жартом розробників, на кшталт пошуку втраченого коня по залишеним їй купками гною в грі Kingdom Come Deliverance, але він все одно є.Говорячи про історії у квестах, не можна ігнорувати такий важливий чинник, як жанр, адже, якщо жанр гри не сюжетний, то квест насамперед має генерувати комплексний та цікавий геймплей.

Першим кроком до створення комплексної системи квестів буде виділення основних ланок загальної структури механізму, а саме:

- створення додаткового персонажу для гри, довкола якого буде побудована система квестів;
- побудова системи досвіду;
- створення міні-карти;
- налаштування квест-акторів (actors);
- створення віджету для квестів;
- додавання квестів на екран віджету;
- побудова системи відстані і напрямку до цілі;

- налаштування NPC та взаємодія з гравцем;
- додавання областей, в якій знаходяться цілі та спосіб патрулювання NPC;
- анімація списку квестів і перемикання на підцілі;
- створення журналу квестів та його підвіджетів;
- розробка функціоналу для журналу квестів;
- додавання системи здоров’я та створення класу ворога;
- додавання поведінки для ворожих створінь;
- створення життєвих показників для ворожого створіння;
- додавання шаблону пошуку предметів;
- реалізація системи балів престижу та відродження ворожих створінь;
- налаштування логіки на закінченість квестів (успіх або провал);
- додавання можливості відмовитись від квесту;
- створення балів престижу та балів досвіду для отримання квестів від NPC;
- реалізація системизбереження прогресу під час гри.

3.6.1 Взаємодія з NPC та прийняття квеста для його виконання.Щоб почати роботу взаємодії з NPC, перш за все створюємо віджет під назвою «InteractionWidget». Цей віджет буде відповідати за коректну логіку дій з боку неігрових персонажів, тобто спілкуватись та давати квести головному персонажу гри. Наступним кроком, потрібно задати логіку безпосередньо для основного персонажу у грі, який буде взаємодіяти з NPC персонажами за допомогою клавіші «E». Створення віджету, логіки взаємодії NPC з головним персонажем та логіки взаємодії головного персонажу з неігровим зображені на рисунках 3.47 – 3.49.

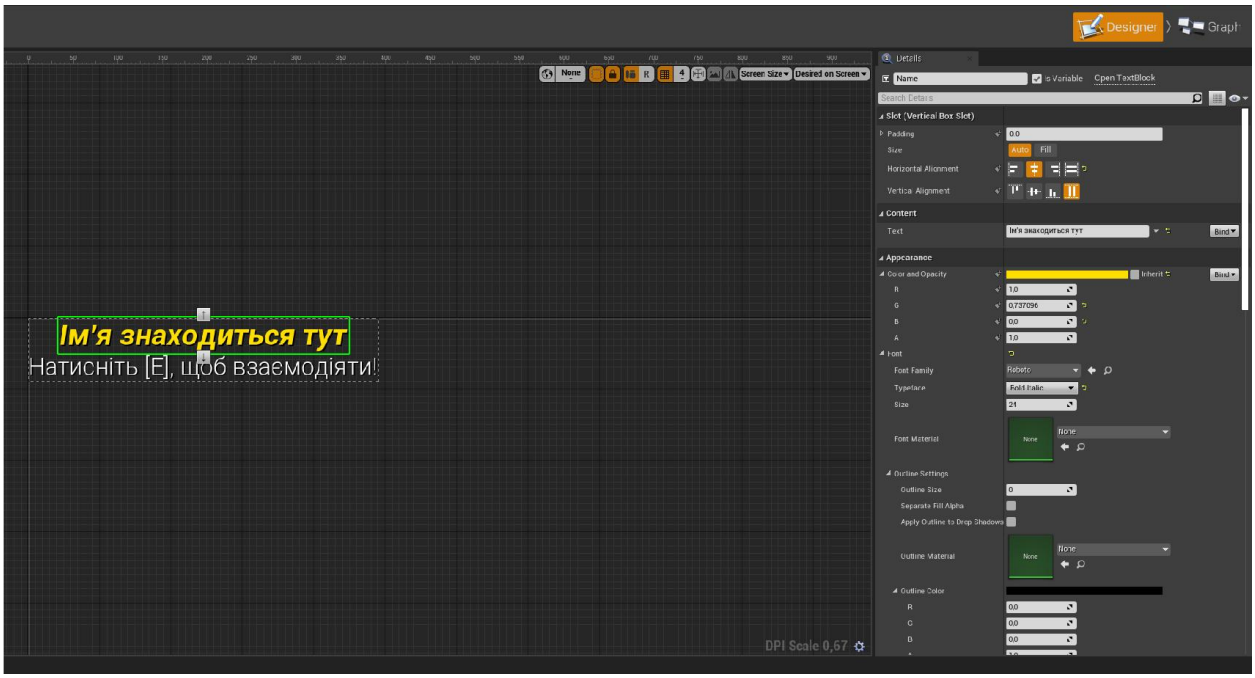


Рисунок 3.47 – Створення віджету «InteractionWidget»

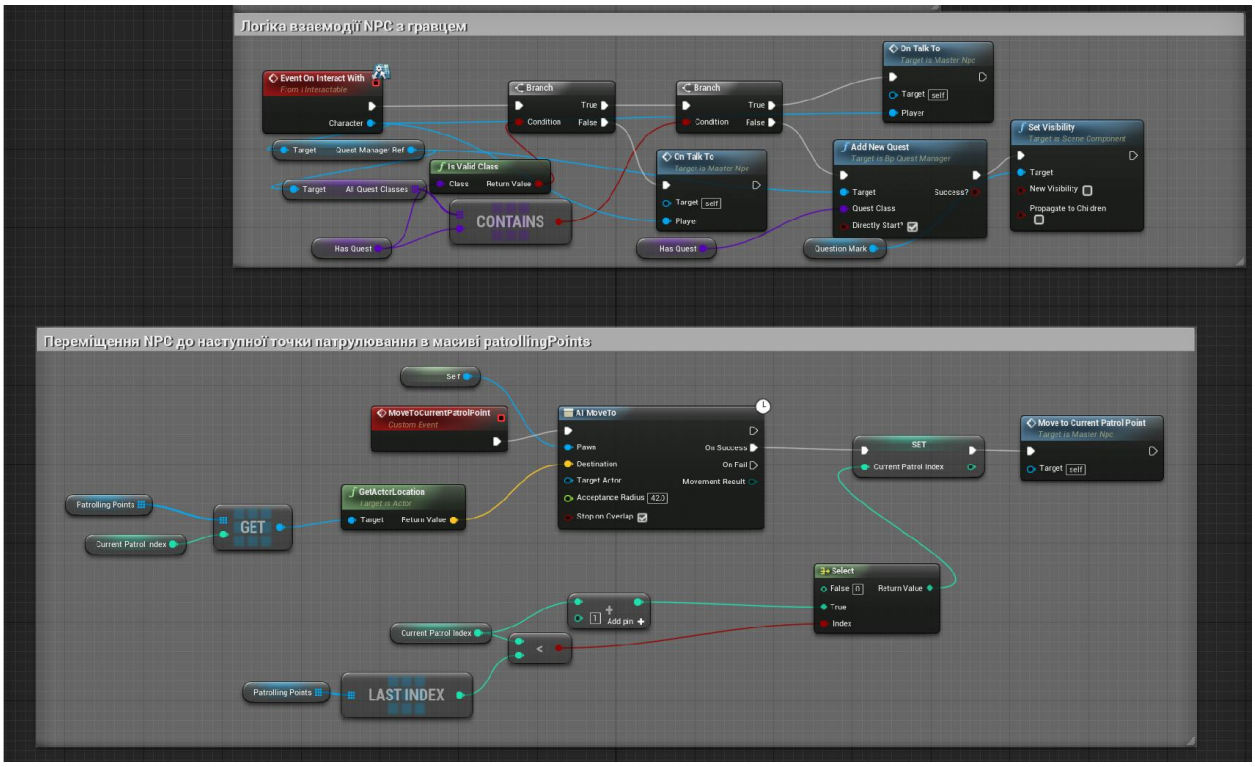
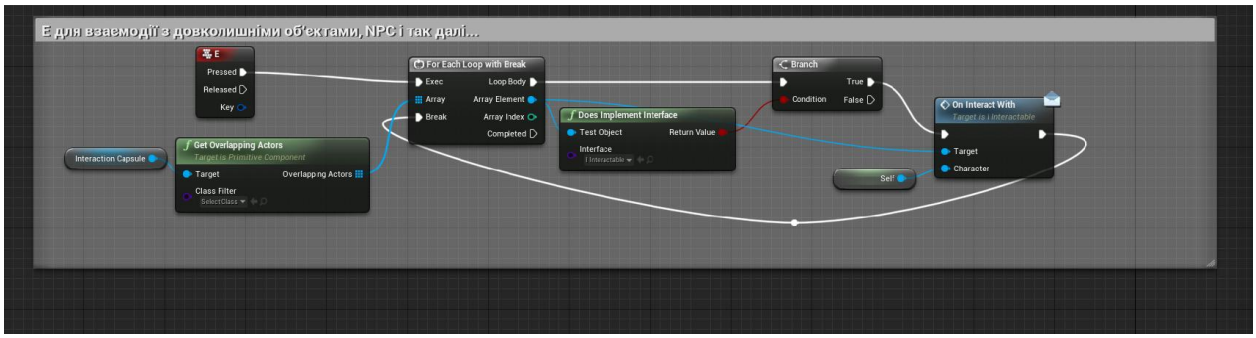


Рисунок 3.48 – Логіка взаємодії NPC з гравцем та його переміщення до наступної точки патрулювання



3.49 – Взаємодія гравця з NPC та довколишнім середовищем

3.6.2 Дизайн журналу квестів та створення його функціоналу.Для розробки дизайну журналу квестів, створимо віджет під назвою «QuestJournal», в якому буде відбуватись конструювання візуальної складової журналу. Він буде містити в собі: список поточних, виконаних та провалених квестів. Також включатиме в себе відображення основного квесту, регіону на якому відбувається виконання квесту, поточний рівень головного персонажу, нагороду за квест (досвід та бали престижу), опис квесту, ціль, а також змога обрати або відмінити квест.

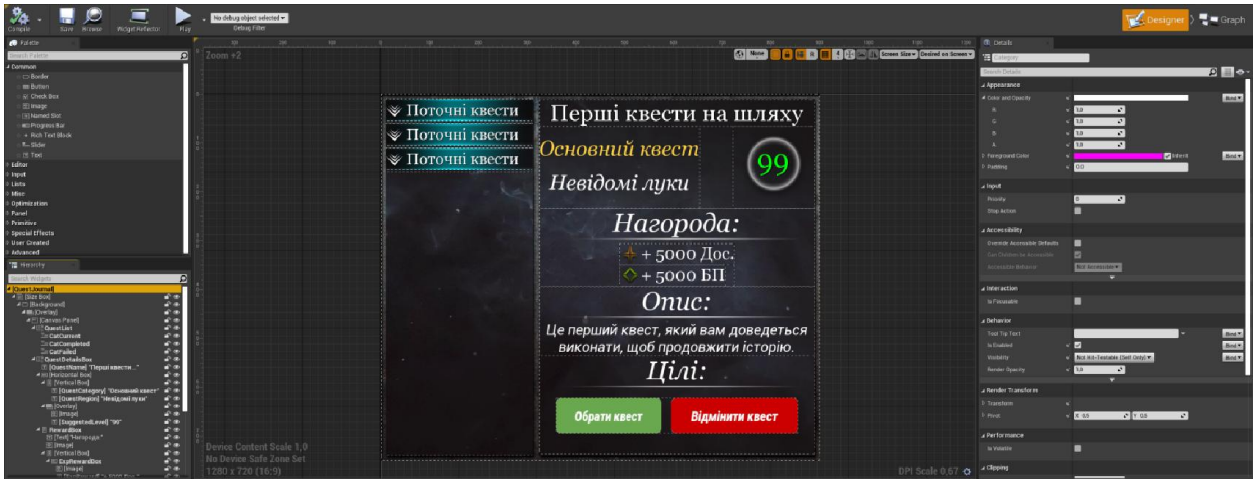


Рисунок 3.50 – Створення дизайну журналу квестів

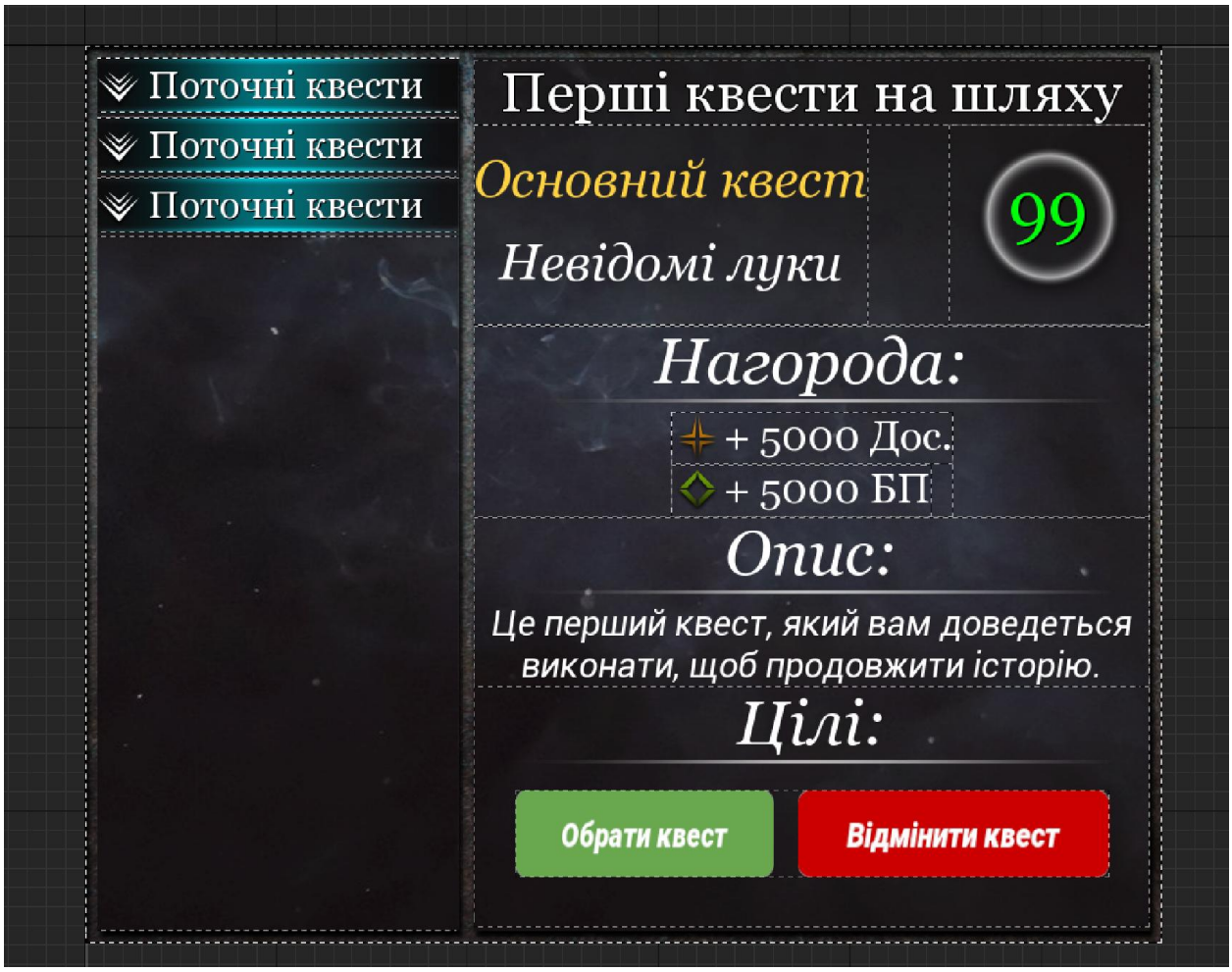


Рисунок 3.51 – Фінальний дизайн журналу квестів

Далі розробляємо функціональну частину журналу квестів, для цього у проєкті створюємо нову структуру під назвою «CompletedGoal». У цій структурі буде відображатися стан квесту залежно від дії гравця, тобто у якому статусі він знаходиться, а саме: виконаний, у процесі виконання, або ж провалений. За цю систему будуть відповідати три змінні, такі як: GoalIndex (тип Integer), GoalInfo (тип S_Goal Info), Succesful? (тип Boolean).Після цьогопереходимо до файлу переліку «e_GoalState»і виставляємо потрібні статуси для квесту.

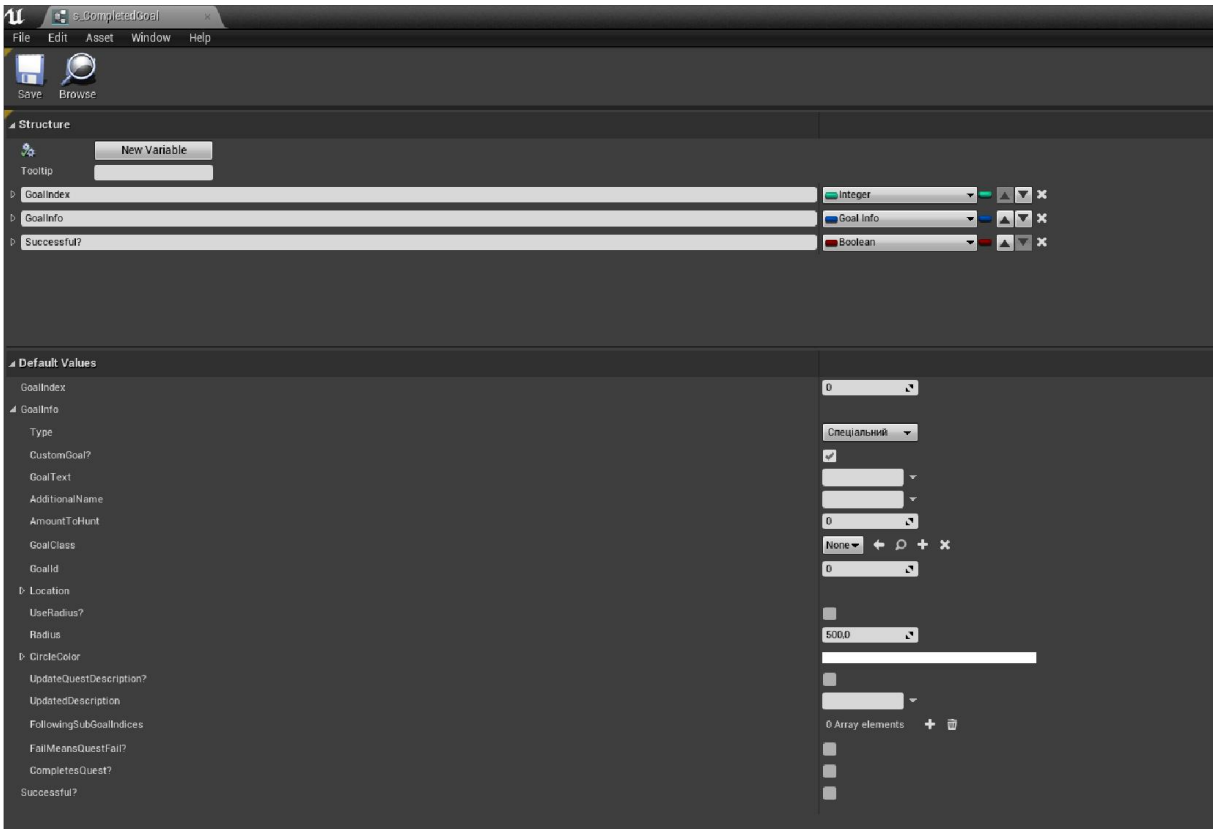


Рисунок 3.52 – Структура «CompletedGoal»

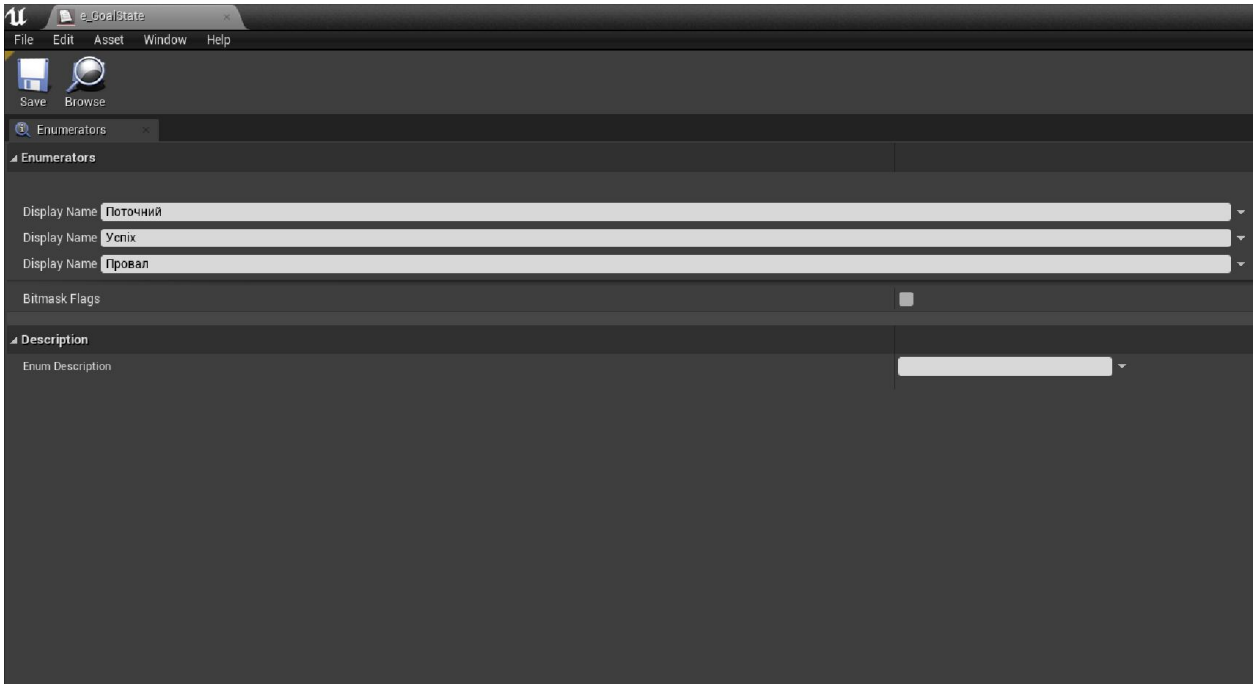


Рисунок 3.53 – Файл переліку статусу квестів «e_GoalState»

Наступним кроком створюємо клас Blueprint під назвою «Bp_QuestManager», у якому буде міститься вся функціональна логіка для журналу квестів. Тому, створюємо функції які відповідають за додавання нового квесту, за відображення поточного квесту, відображення виконаних

завдань, квестів, які були закінчені, скасування квестів, а також функція, яка відповідатиме за отримання інформації всіх NPC за певним ідентифікатором.

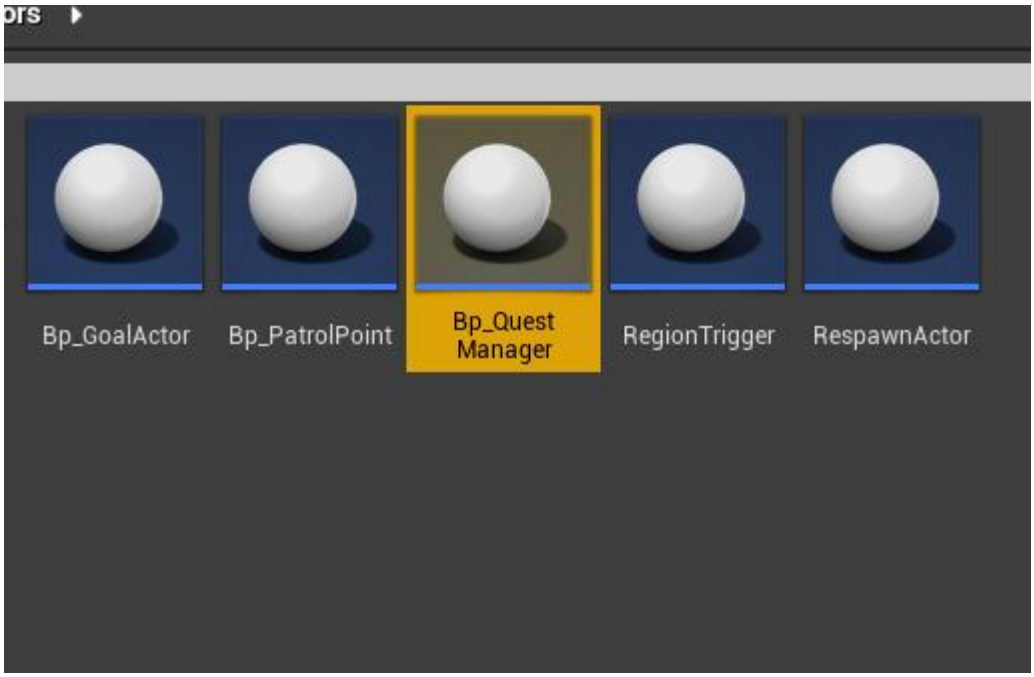


Рисунок 3.54 – Створення Blueprint класу «Bp_QuestManager»

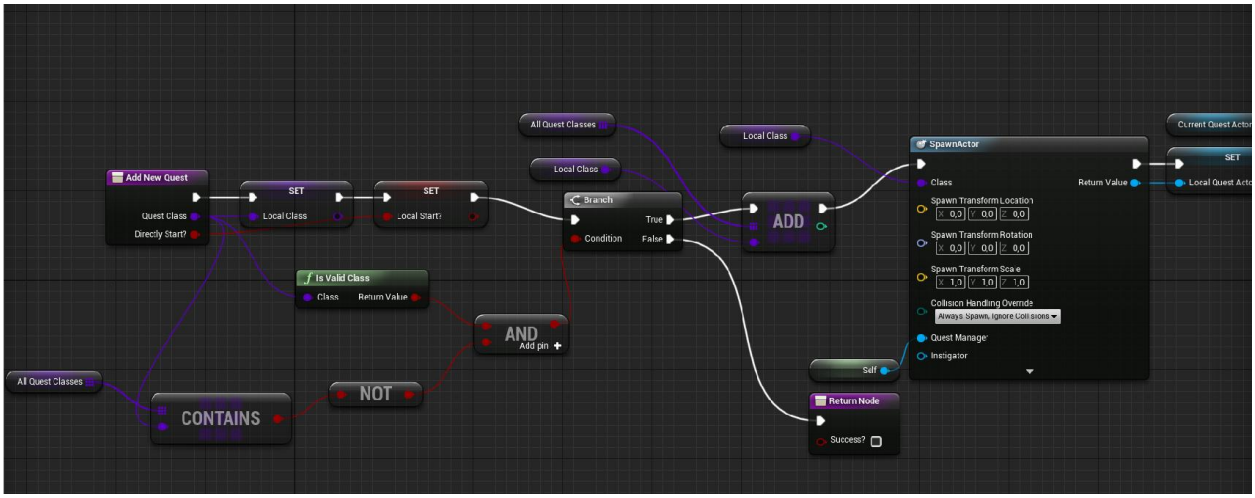


Рисунок 3.55 – Частина логіки для функції додавання квестів

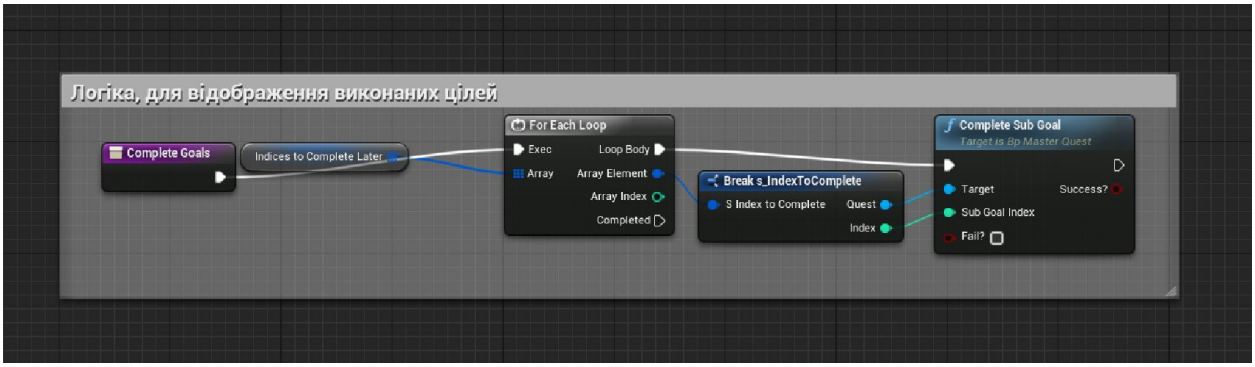


Рисунок 3.56 – Логіка, для відображення виконаних цілей

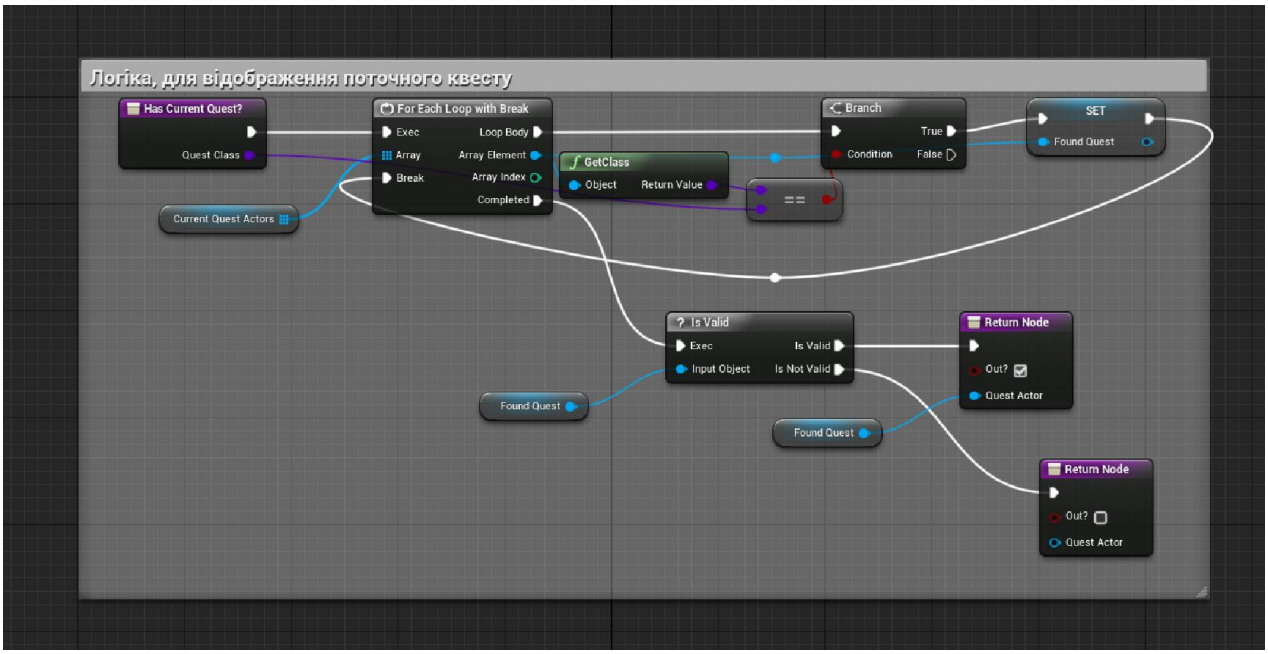


Рисунок 3.57 – Логіка, для відображення поточних квестів

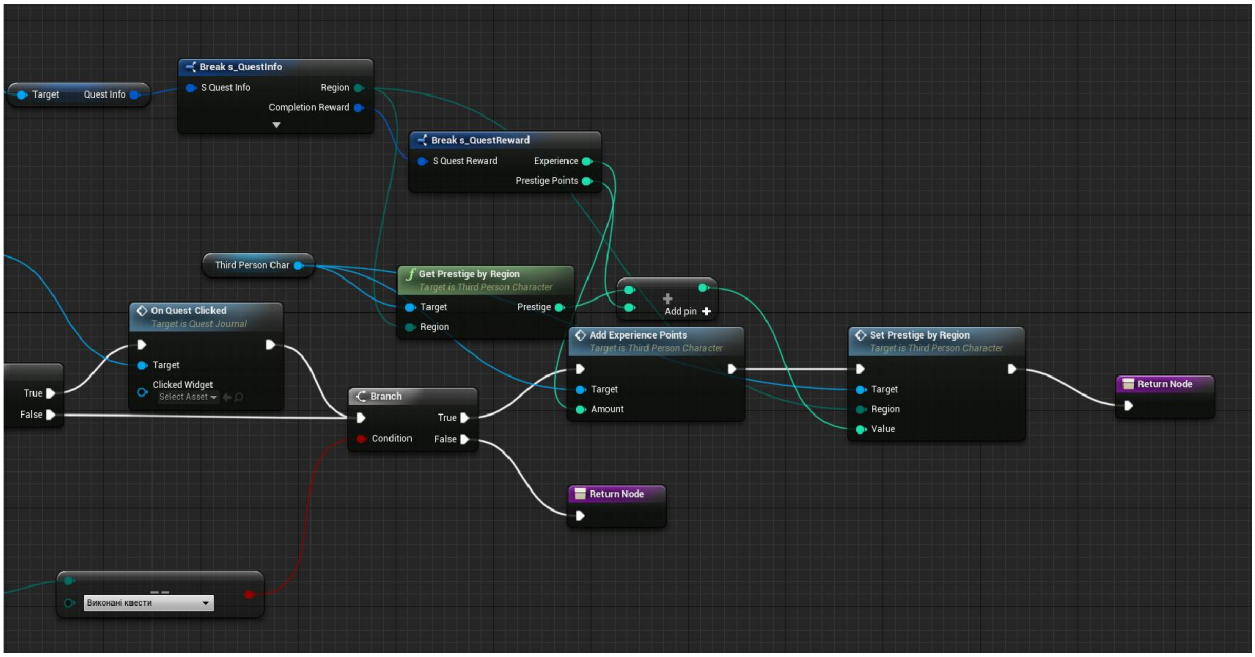


Рисунок 3.58 – Логіка, для відображення виконаних квестів

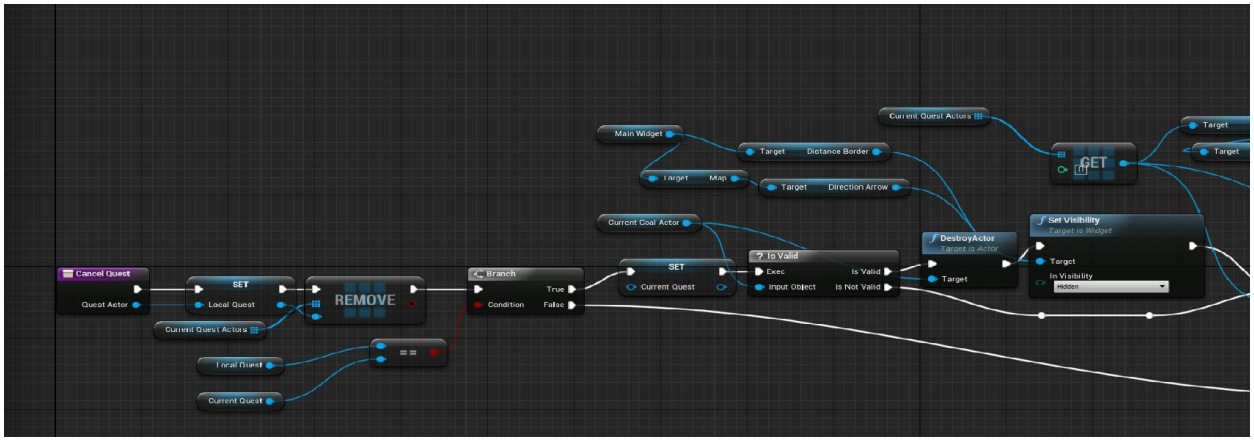


Рисунок 3.59 – Частина логіки, яка відповідає за відміну квестів

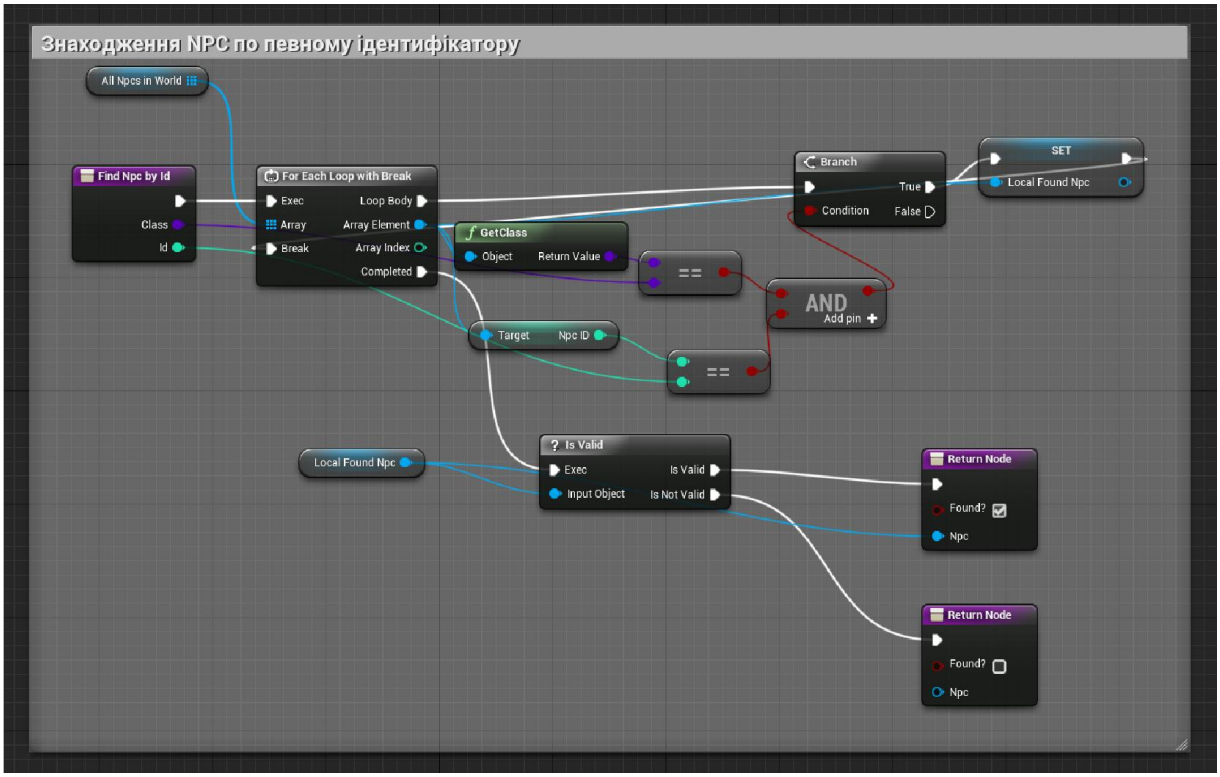


Рисунок 3.60 – Логіка, для знаходження NPC по ідентифікатору

Рисунок 3.61 – Наочний приклад відображення квесту у журналі

3.6.3 Розробка системи життєвих одиниць та підготовка класу ворога.Спочатку створюємо основний віджет під назвою «MainWidget», який буде відображати основну інформацію для гравця. Далі, проєктуємо дизайн шкали життєвих одиниць і за допомогою певних функцій, а також змінних «MaxHealth» та «CurrentHealth» створимо логіку, яка буде коректно працювати протягом гри.

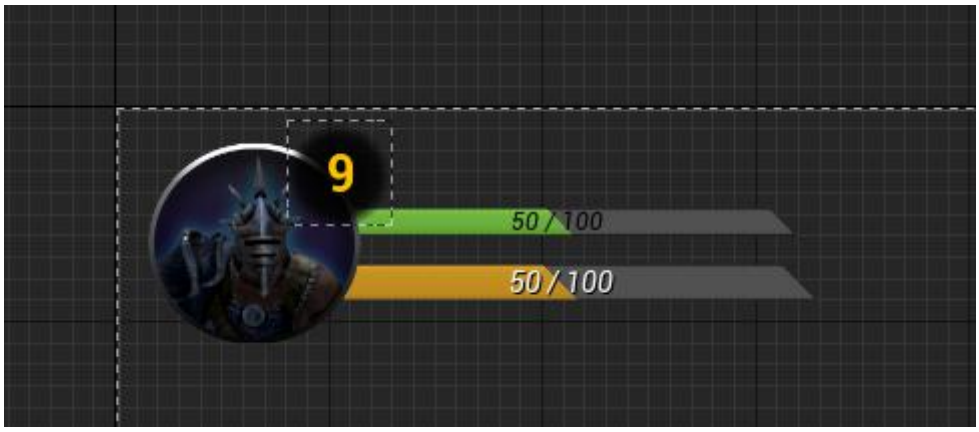


Рисунок 3.62 – Створення дизайну життєвих одиниць для головного персонажа

Потім, переходимо до класу «ThirdPersonCharacter», який відповідає за коректну роботу головного персонажа гри та містить в собі основну частину логіки його дієздатності, і вже там створюємо більш детальні функції та системи для життєвих одиниць.

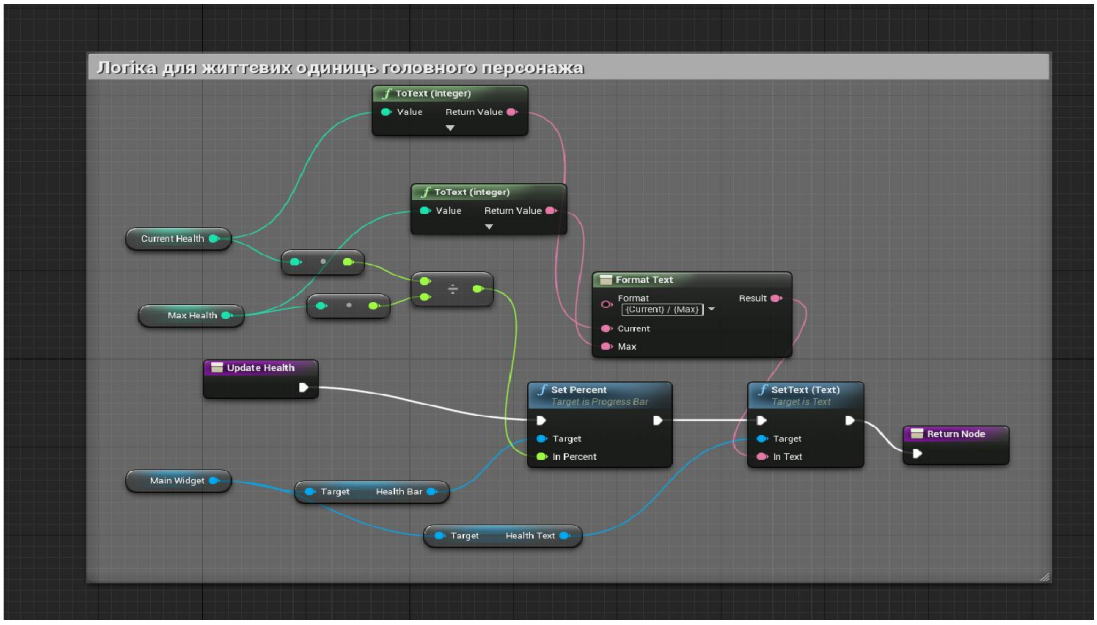


Рисунок 3.63 – Логіка для життєвих одиниць головного персонажа

Переходимо до створення та підготовки класу ворога для загальної системи квестів. Розробляємо клас Blueprint під назвою «Master_Enemy» і відразу ж створюємо там змінну булевого типу «IsDeath?» для подальшого удосконалення логіки. Потім переносимо заздалегідь підготовлені анімації та модель ворожого створіння до класу «Master_Enemy».

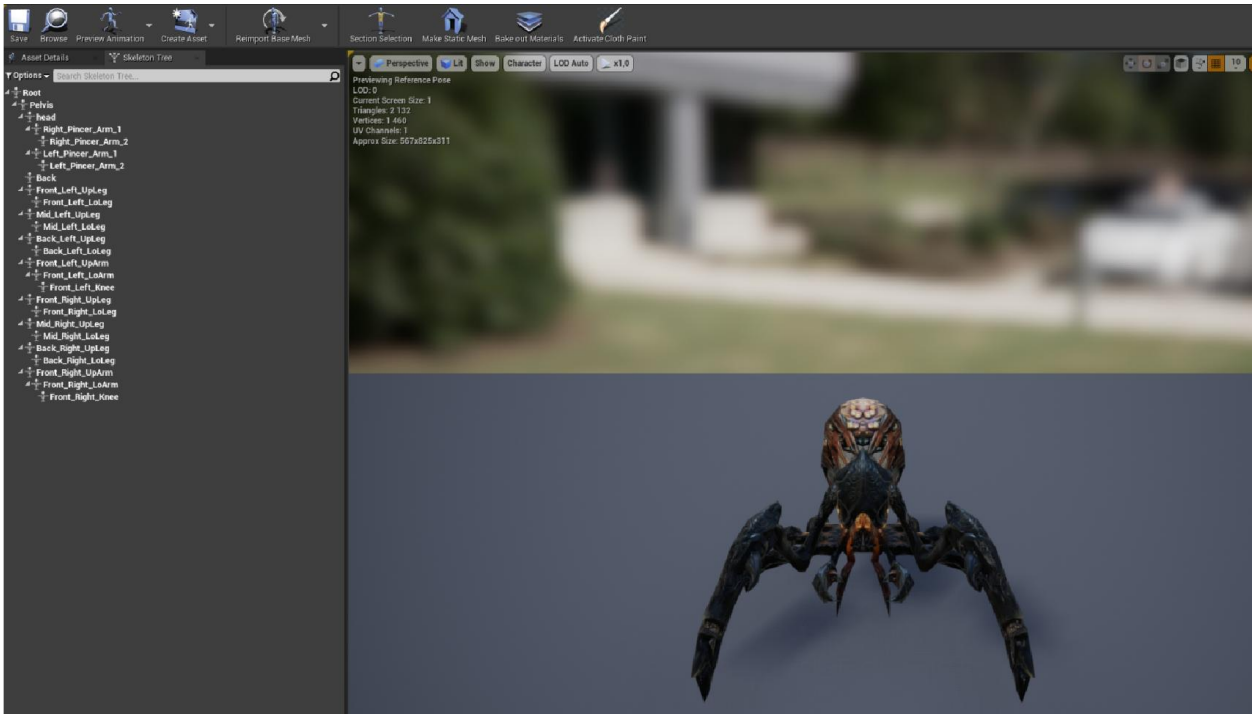


Рисунок 3.64 – Модель ворожого створіння для системи квестів

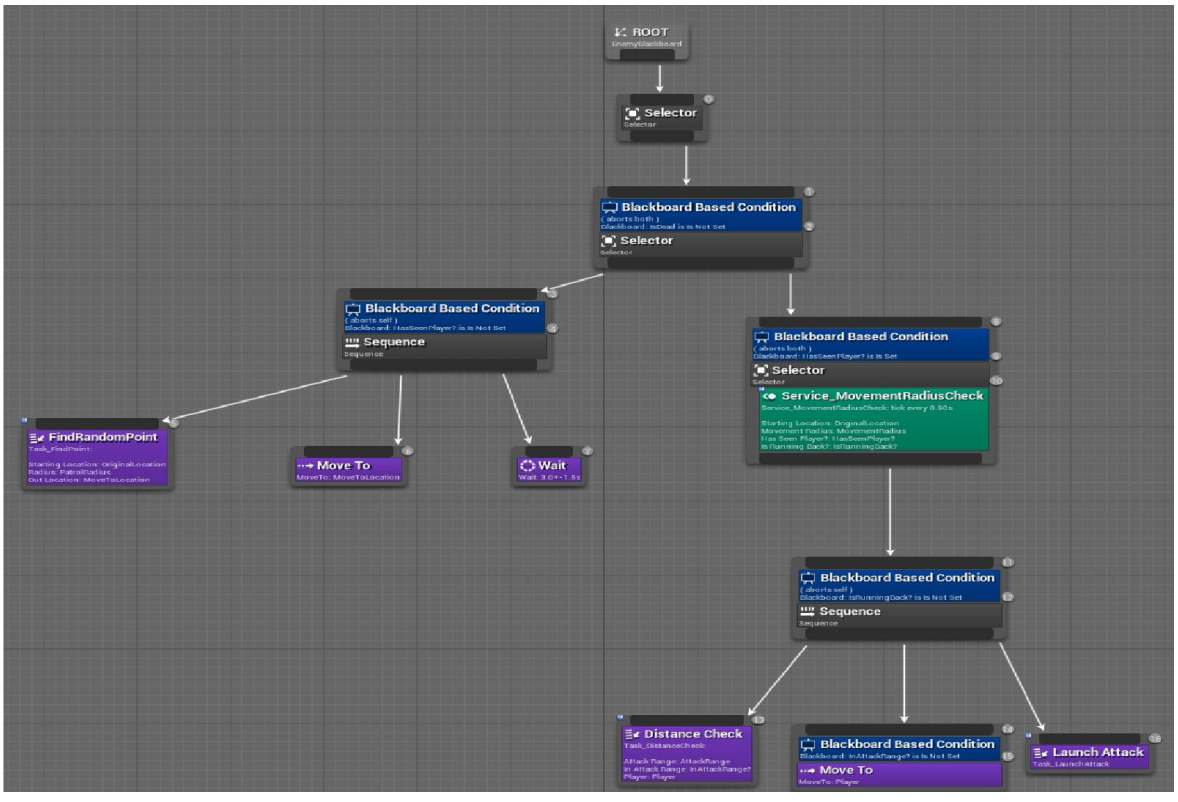


Рисунок 3.65 – Загальна логіка поведінки ворожого створіння

3.6.4 Розробка шаблону пошуку предметів для системи квестів.Для початку знаходимо створений раніше файл «InteractionWidget», потім у ньому ставимо відмітку «Is Variable» для того, щоб можна було у подальшому підняти предмет головним персонажем гри.

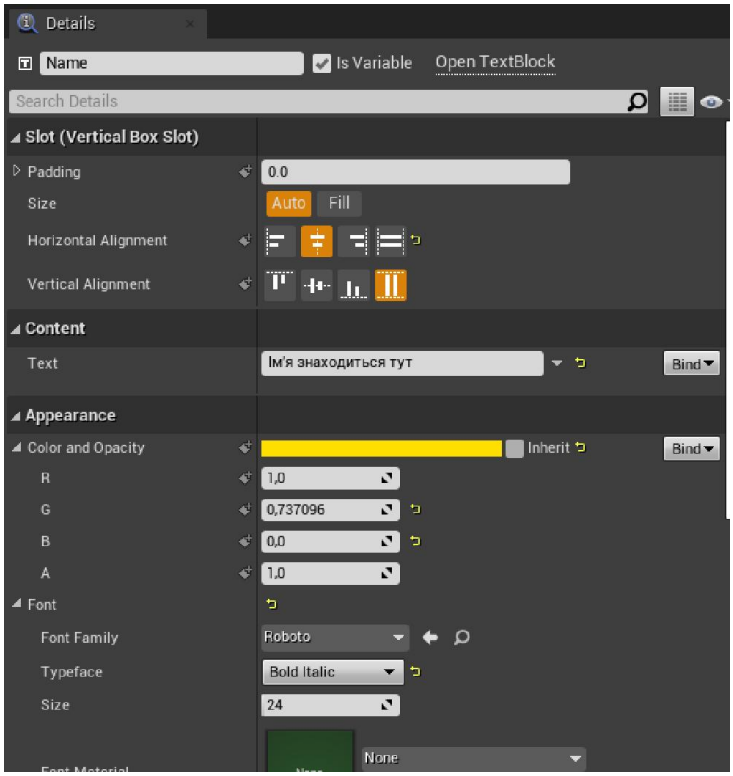


Рисунок 3.66 – відмітка «Is Variable»

Наступним кроком, переходимо до папки з файлами під назвою «Actors» та створюємо новий клас Blueprint під назвою «Master_Object», а потімбудуємо у ньому логіку для підняття предметів головним персонажем. Ця система буде відповідати практично повністю за взаємодію між персонажем гри та різноманітними об’єктами навколо.

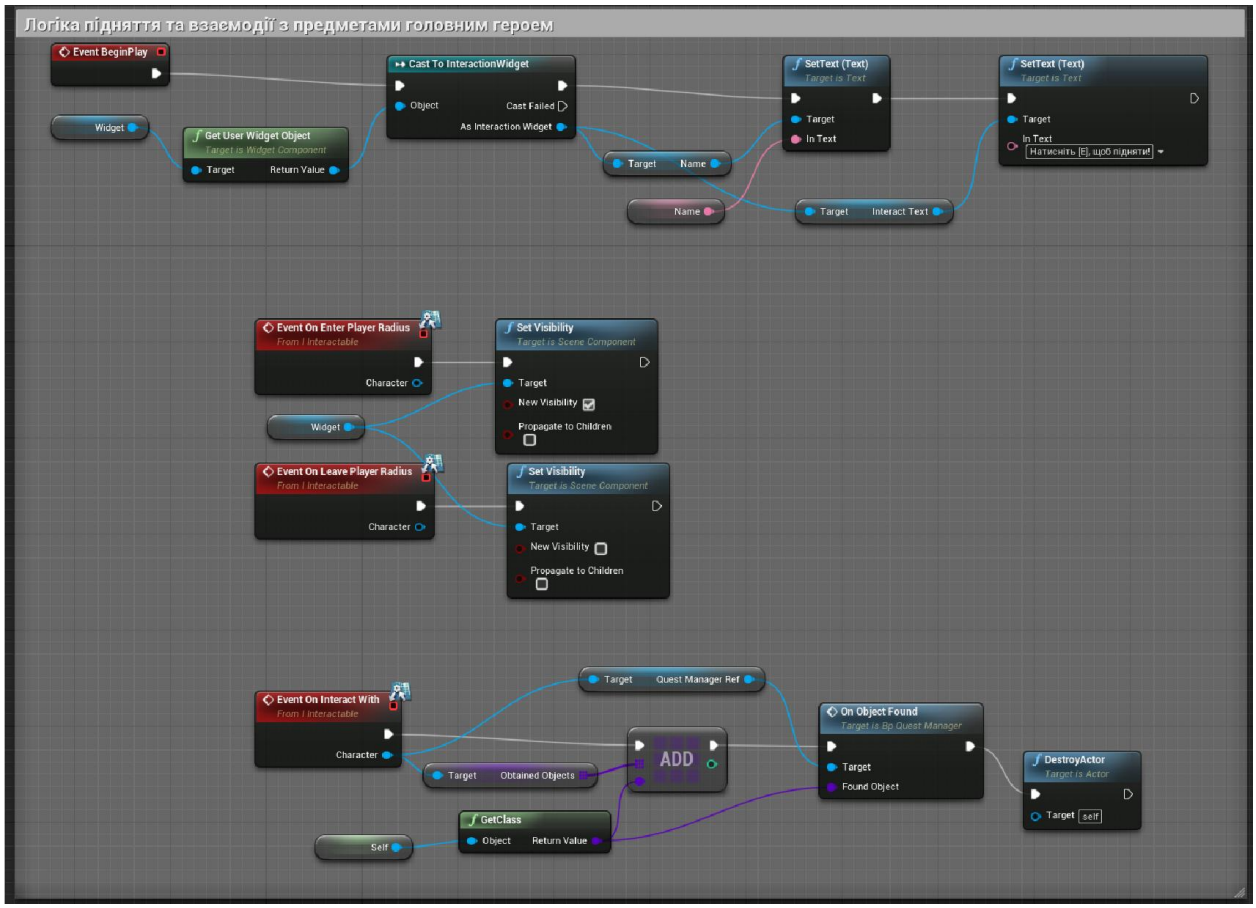


Рисунок 3.67 – Логіка підняття предметів головним персонажем

3.6.5 Реалізація системи балів досвіду та престижу.Суть реалізації системи балів досвіду та престижу полягає у розробці логіки, в якій NPC буде вирішувати, давати головному персонажу квест чи ні в співвідношенні його балів престижу або ж досвіду.

Тому, переходимо до Blueprint класу, який відповідає за систему головного персонажа, та створюємо там логіку балів престижу залежно від регіону у грі та даємо їх назву «Set Prestige by Region».

Далі розробляємо систему підвищення рівня головного персонажа після виконання квесту, а також звуковий та візуальний ефект після підвищення рівня персонажа для більш динамічної гри.

Останнім кроком, буде створення логіки отримання нагороди після виконаного квесту від NPC у вигляді балів престижу, а також отримання досвіду після знищення ворожих створінь.

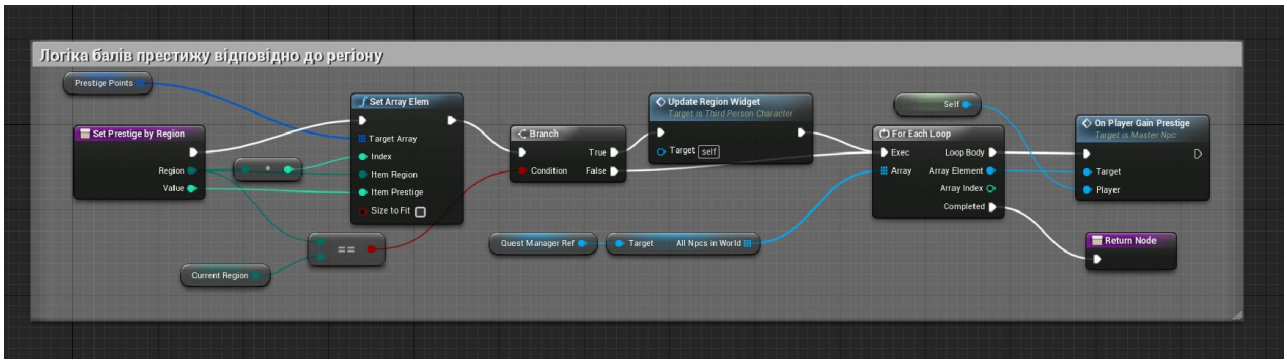


Рисунок 3.68 – Логіка балів престижу відповідно до регіону

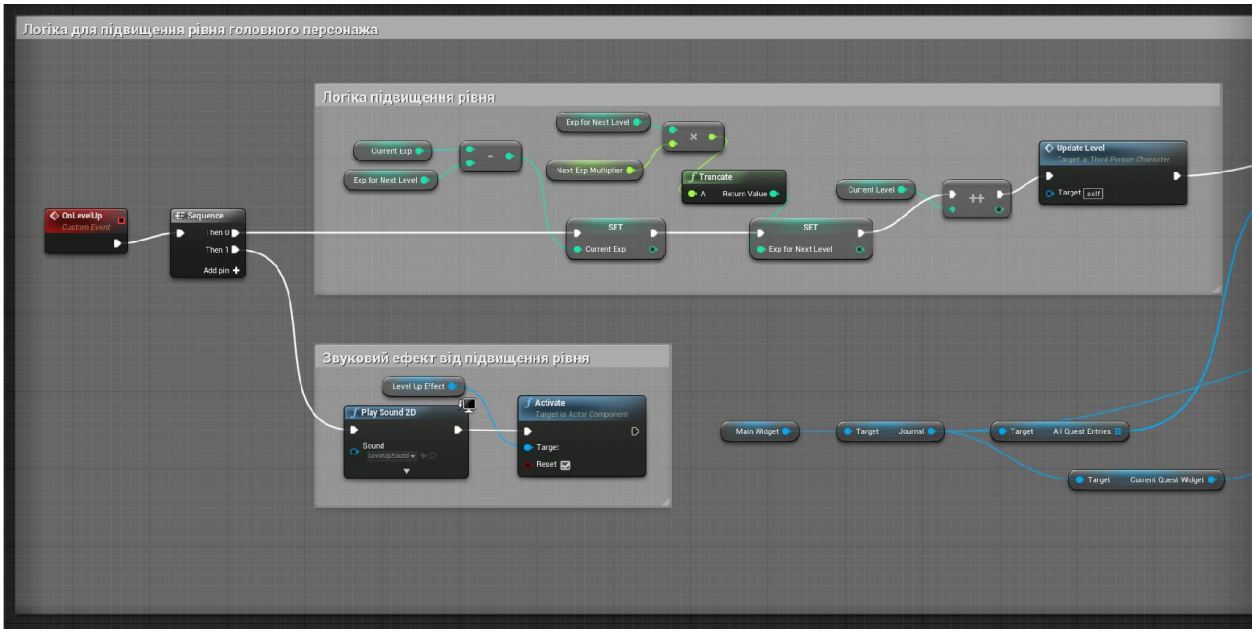


Рисунок 3.69 – Логіка для підвищення рівня головного персонажа

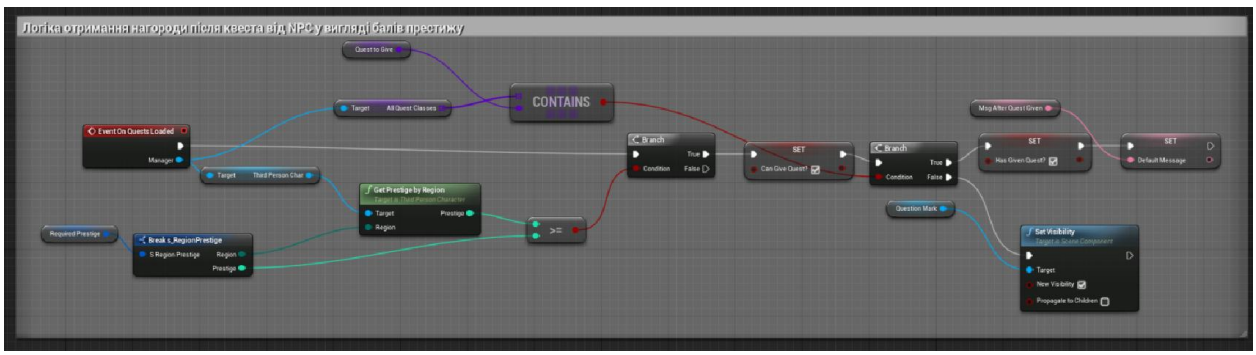


Рисунок 3.70 – Логіка для отримання нагороди після виконання квесту від NPC у вигляді балів престижу

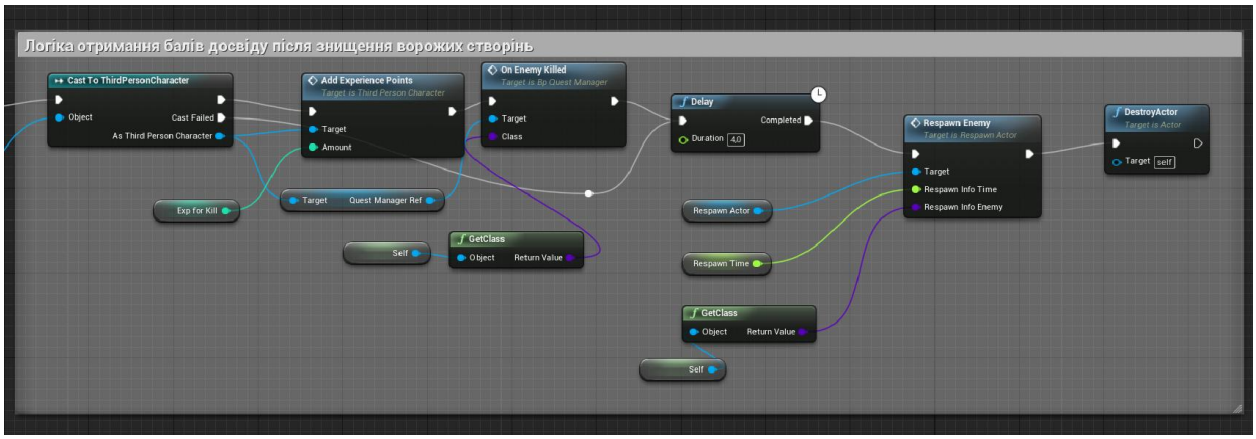


Рисунок 3.71 – Логіка для отримання балів досвіду після знищення ворожих створінь

Після реалізації системи вище, встановлюємо у змінну під назвою «ExpForKill» кількість балів досвіду за одного знищеного створіння, а саме двадцять п’ять одиниць.

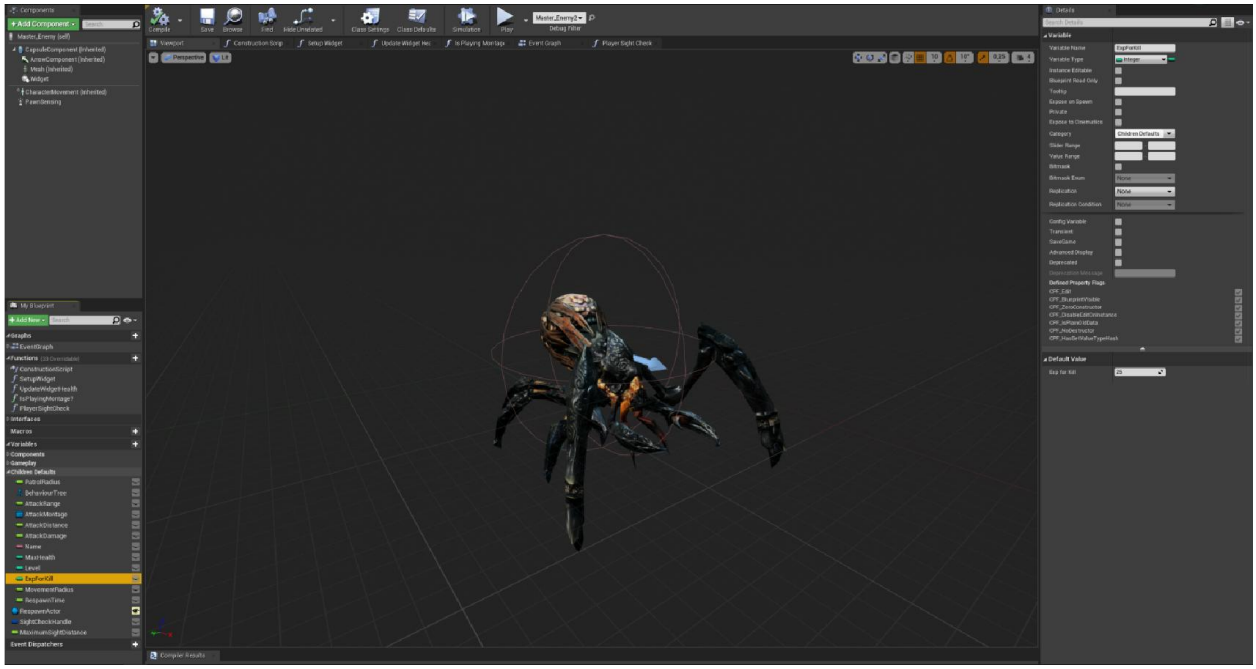


Рисунок 3.72 – Встановлення певної кількості балів досвіду за одного знищеного ворожого створіння

РОЗДІЛ 4
ТЕСТУВАННЯ

Життєво важливою частиною розробки будь-якого нового продукту є фаза тестування, а для відеоігор це важлива частина контролю якості. Основна мета тестування, незалежно від використовуваної методики, полягає у виявленні та документуванні будь-яких дефектів в програмному забезпеченні. Ці дефекти зазвичай називають помилками або «багами». Бути ігровим тестером вимагає високого рівня технічних знань, включаючи комп'ютерні навички, аналітичні та оціночні навички, певна кількість терпіння і витривалості.

У виробництві ігор застосовуються різні типи тестування, кожен з яких призначений для виявлення відповідних типів недоліків. Взагалі, існує багато різних методів тестування для абсолютно різного програмного забезпечення, але тестування комп’ютерних ігор дещо складніше ніж тестування неігрових додатків, як мінімум через те, що відеоігри відрізняються від іншого ПЗ наявністю індивідуальної механіки. Наприклад, у якомусь додатку є певний алгоритм, згідно якому цей додаток працює. У іграх (особливо в жанрі RPG) цього немає, тому що це може зламати всю ідею та систему.

Тестування гри для цього проєкту було виконано двома методами – методом функціонального тестування та методом ручного тестування за допомогою тест-кейсів.

Мета функціонального тестування виявити відхилення необхідної функціональності і помилки, або так званих «багів». Як правило, цей тип тестування не вимагає від тестувальника великих технічних знань, крім розуміння базових концепцій програмування, а також зводиться до багаторазового проходження гри, виявлення неполадок і умов, в яких їх можна відтворити. Проблеми описують в довільній формі, але важливо лише те, щоб текст був зрозумілий розробнику.

Ручне тестування – тестування, в якому тест-кейси виконуються

людиною вручну без використання засобів автоматизації. Незважаючи на те, що це звучить дуже просто, від тестувальника в ті чи інші моменти часу потрібні такі якості, як терплячість, спостережливість, креативність, вміння ставити нестандартні експерименти, а також уміння бачити і розуміти, що відбувається «всередині системи», так як зовнішні впливи на гру трансформуються в її внутрішні процеси.

Основне тестування гри полягало в розробці тест-кейсів, які визначали чи був дійсним очікуваний результат для певної дії в грі для користувача, або ж навпаки, перевірялось, що система не дозволить виконати незаплановані механікою гри дії, які буде намагатися виконати користувач. У цьому випадку буде надіслано повідомлення про помилку в дії.

Структура тестування складалася з таких дій, як:

- перевірка загального балансу у грі;
- перевірка дизайну ігрового оточення, тобто типові помилки такі, як stuck spot (ви застрягли при неможливості вилізти), sticky spot (персонаж застряг, смикається, але в підсумку таки вилазить, витративши багато часу), невидимі стіни (на кшталт пройти можна, а ніби й не можна), текстурні діри (провалились у землю), і відсутність геометрії рівнів (стіна начебто є, але через неї можна легко проскочити);
- перевірка штучного інтелекту ворожих персонажів, тобто коректність виконання дій AIу грі;
- перевірка фізики бойової системи, тобто перевіряється адекватність потрапляння шкоди по цілям, чи є взагалі логічність бойової механіки в цілому;
- перевірка фонової музики та звуків, коректність зміни музики та звуків в залежності від ситуації та місця знаходження;
- перевірка коректної роботи квест-системи.

Набір тест-кейсів, за якими було виконано тестування, наведено у додатку В.

ВИСНОВОК

В рамках даної дипломної роботи було виконано розробку комп’ютерної 3D гри на рушії UnrealEngine 4, жанр якої Action-RPG. Варто відзначити, що саме цей жанр відеоігор знову набирає популярність завдяки своїй гнучкості та великій кількості фанатів по всьому світі.

Під час розгляду різноманітних корисних джерел практичної розробки комп’ютерних ігор були отримані необхідні знання для створення власної гри. Було вивчено певну частину можливостей та інструментів рушія UnrealEngine 4, який став основою розробленої гри, процес створення якої докладно описаний в роботі. Вагому роль у цій роботі зіграв аналіз історії зародження індустрії комп’ютерних ігор, яку можна спостерігати по сьогоднішній день. Була розглянута базова термінологія комп’ютерних ігор, за допомогою якої пришвидшилось створення власної гри.

У ході розробки гри було додано достатньо внутрішнього контенту самого проєкту, до якого відносяться різноманітні механіки, анімації, музика, звуки, персонажі, штучний інтелект, бойова система та інший функціонал який містить в собі індивідуальну логіку.

На даному етапі свого існування гра має багато напрямків розвитку та об’ємний концепт оновлень в майбутньому, що в перспективі дає можливість довгострокової актуальності та попиту для багатьох користувачів впродовж великого відрізку часу.

Була розроблена система штучного інтелекту для внутрішньо-ігрових завдань, і також виконана основна ціль даної дипломної роботи, а саме створення комп’ютерної гри, яка зможе зацікавити користувачів будь-якого віку та інтересів, що внаслідок дасть можливість стрімкого розвитку даної гри та потрапляння її на певні онлайн-сервіси цифрової дистрибуції відеоігор.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Top 10 Video Game Engines [Електронний ресурс] / Jameson Durall, Trey Sharp // Article. – 2020. – Режим доступу до ресурсу: https://www.gamedesigning.org/career/video-game-engines/

2. Blueprints Visual Scripting for Unreal Engine/Marcos Romero, Brenden Sewell., – 2019. – 380 с. – (2nd Edition)

3. What is the best game engine: is Unreal Engine right for you? [Електронний ресурс] / MarieDealessandri //Article. – 2020. – Режим доступу до ресурсу: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unreal-engine-4-the-right-game-engine-for-you.

4. Unreal Engine 4 for Design Vizualization / Tom Shannon., – 2017. – 384с. – (1st Edition).

5. Gameengines: asurvey [Електронний ресурс] / A. Andrade // Article. – 2015. – Режим доступу до ресурсу: https://www.researchgate.net/publication/283657797_Game_engines_a_survey

6. Game Engines and Game History[Електронний ресурс] / Henry Lowood // Article. – 2015. – Режим доступу до ресурсу: https://www.kinephanos.ca/2014/game-engines-and-game-history/

7. How the Video Game Industry Is Changing[Електронний ресурс] / Andrew Beattie // Article. – 2020. – Режим доступу до ресурсу: https://www.kinephanos.ca/2014/game-engines-and-game-history/

8. Game Engine Architecture / Jason Gregory., – 2019. – 1240с. – (3rd Edition).

9. Foundation of Game Engine Development / Eric Lengyel., – 2019. – 412с. – (Book 2).

10. Unreal Engine Blueprints Visual Scripting Projects: Learn Blueprints Visual Scripting in UE4 by bylding three captivating 3D Games / Lauren S. Ferro., – 2019. – 530с. – (1st Edition).

11. Beginning Unreal Game Development: Foundation for Simple to

Complex Games Using UE4 / David Nixon., – 2020. – 389c. – (1st Edition).

12. Role-playingvideogame [Електронний ресурс] / WilliamL. Hosch // Article. – 2016. – Режим доступу до ресурсу: https://www.britannica.com/topic/role-playing-video-game

13. Unreal Engine 4 AI Programming Essentials / Peter L. Newton, Jie Feng., – 2016. – 188c.

14. Unreal Engine 4 Tutorial for Beginners: Getting Started[Електронний ресурс] / Tommy Tran // Article. – 2017. – Режим доступу до ресурсу: https://www.raywenderlich.com/771-unreal-engine-4-tutorial-for-beginners-getting-started

15. Game Audio Implementation: A Practical Guide Using the Unreal Engine / Richard Stevens, Dave Raybould, Clinton Crumpler., – 2015. – 486 c. – (1st Edition).

16. The Many Different Types of Video Games & Their Subgenres[Електронний ресурс] / Vince // Article. – 2018. – Режим доступу до ресурсу: https://www.idtech.com/blog/different-types-of-video-game-genres

17. Unreal Engine 4 Game Development in 24 hours / Aram Cookson, Ryan DowlingSoka, Clinton Crumpler., – 2016. – 496 c. – (1st Edition).

18. What Is GameMaker: Studio? [Електронний ресурс]/ Michael Rohde // Article. – 2020. – Режим доступу до ресурсу: https://www.dummies.com/programming/programming-games/what-is-gamemaker-studio/

19. What is the best game engine: is GameMaker right for you?[Електронний ресурс] / Marie Dealessandri // Article. – 2020. – Режим доступу до ресурсу: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-gamemaker-the-right-game-engine-for-you

20. Godot Docs – 3.3 branch [Електронний ресурс] / Juan Linietsky, Ariel Manzur // Document. – 2014. – Режим доступу до ресурсу: https://docs.godotengine.org/en/stable/#

21. 10 Reasons to Use Godot Engine for Developing Your Next Game

[Електронний ресурс] / Ian Buckley // Article. – 2019. – Режим доступу до ресурсу: https://www.makeuseof.com/tag/reasons-godot-engine-game-development/

22. Godot Engine Game Development Projects: Build five cross-platform 2D and 3D games with Godot 3.0 / Chris Bradfield., – 2018. – 174 с.

23. Game Development with GameMaker Studio 2: Make Your Own Games with GameMaker Language / Sebastiano M. Cossu., – 2019. – 139 с. – (1st Edition).

24. Source SDK Game Development Essentials / Brett Bernier., – 2014. – 312 с.

25. Unreal Engine Blueprints Tutorials – Complete Guide for 2021 [Електронний ресурс] / Daniel Buckley // Article. – 2021. – Режим доступу до ресурсу: https://gamedevacademy.org/unreal-blueprints-tutorial/

26. «Компьютерные игры как искусство» [Електронний ресурс] / Киризлеев Александр // Стаття. – 2009. – Режим доступу до ресурсу: https://gamesisart.ru/TableJanr.html

27. What is a game designer? [Електронний ресурс] / Ryan // Article. – 2020. – Режим доступу до ресурсу: https://www.idtech.com/blog/what-is-a-game-designer

28. Cryengine 3 Game Development: Beginner's Guide / Sean Tracy, Paul Reindell., – 2012. – 117 с.

29. Procedural Generation in Game Design / Tanya Short, Tarn Adams., – 2017. – 313 с. – (1st Edition).

30. Video Game Genres: Everything You Need to Know [Електронний ресурс] / Dwight Pavlovic // Article. – 2020. – Режим доступу до ресурсу: https://www.hp.com/us-en/shop/tech-takes/video-game-genres

31. GAMEDEV: 10 Steps to Making Your First Game Successful / Wlad Marhulets., – 2020. – 253 с.

32. Extra Lives: Why Video Games Matter / Tom Bissell., – 2011. – 256 с.

ДОДАТОК А
ВИХІДНИЙ КОД ОСНОВНИХ КОМПОНЕНТІВ

1. Ландшафт ігрового оточення (Landscape.h)

```
#pragmaonce


#include"CoreMinimal.h"
#include"UObject/ObjectMacros.h"
#include"LandscapeProxy.h"
#include"LandscapeBlueprintBrushBase.h"
#include"Delegates/DelegateCombinations.h"

#include"Landscape.generated.h"

classULandscapeComponent;
classILandscapeEdModeInterface;

namespace ELandscapeToolTargetType
{
    enumType : int8;
};

#if WITH_EDITOR
extern LANDSCAPE_API TAutoConsoleVariable<int32> CVarLandscapeSplineFalloffModulation;
#endif

UENUM()
enum ELandscapeSetupErrors
{
    LSE_None,
           LSE_NoLandscapeInfo,
    LSE_CollsionXY,
    LSE_NoLayerInfo,
    LSE_MAX,
};


enumclassERTDrawingType : uint8
{
    RTAtlas,
    RTAtlasToNonAtlas,
    RTNonAtlasToAtlas,
    RTNonAtlas,
    RTMips
};

enumEHeightmapRTType : uint8
{
    HeightmapRT_CombinedAtlas,
    HeightmapRT_CombinedNonAtlas,
    HeightmapRT_Scratch1,
    HeightmapRT_Scratch2,
    HeightmapRT_Scratch3,

    HeightmapRT_Mip1,
    HeightmapRT_Mip2,
    HeightmapRT_Mip3,
    HeightmapRT_Mip4,
    HeightmapRT_Mip5,
    HeightmapRT_Mip6,
```

```
    HeightmapRT_Mip7,
    HeightmapRT_Count
};

enumEWeightmapRTType : uint8
{
    WeightmapRT_Scratch_RGBA,
    WeightmapRT_Scratch1,
    WeightmapRT_Scratch2,
    WeightmapRT_Scratch3,


    WeightmapRT_Mip0,
    WeightmapRT_Mip1,
    WeightmapRT_Mip2,
    WeightmapRT_Mip3,
    WeightmapRT_Mip4,
    WeightmapRT_Mip5,
    WeightmapRT_Mip6,
    WeightmapRT_Mip7,

    WeightmapRT_Count
};

#if WITH_EDITOR
enumELandscapeLayerUpdateMode : uint32;
#endif

USTRUCT()
struct FLandscapeLayerBrush
{
    GENERATED_USTRUCT_BODY()

    FLandscapeLayerBrush()
#if WITH_EDITORONLY_DATA
          : FLandscapeLayerBrush(nullptr)
#endif
    {}

    FLandscapeLayerBrush(ALandscapeBlueprintBrushBase* InBlueprintBrush)
#if WITH_EDITORONLY_DATA
          : BlueprintBrush(InBlueprintBrush)
          , LandscapeSize(MAX_int32, MAX_int32)
          , LandscapeRenderTargetSize(MAX_int32, MAX_int32)
#endif
    {}

#if WITH_EDITOR
    UTextureRenderTarget2D* Render(bool InIsHeightmap, const FIntRect&
InLandscapeSize, UTextureRenderTarget2D* InLandscapeRenderTarget, const FName&
InWeightmapLayerName = NAME_None);
    ALandscapeBlueprintBrushBase* GetBrush() const;
    boolIsAffectingHeightmap() const;
    boolIsAffectingWeightmapLayer(const FName& InWeightmapLayerName) const;
    voidSetOwner(ALandscape* InOwner);
#endif

private:

#if WITH_EDITOR
    boolInitialize(const FIntRect& InLandscapeExtent, UTextureRenderTarget2D*
InLandscapeRenderTarget);
#endif
```

```
#if WITH_EDITORONLY_DATA
    UPROPERTY()
    ALandscapeBlueprintBrushBase* BlueprintBrush;

    FTransform LandscapeTransform;
    FIntPoint LandscapeSize;
    FIntPoint LandscapeRenderTargetSize;
#endif
};

UENUM()
enum ELandscapeBlendMode
{
    LSBM_AdditiveBlend,
    LSBM_AlphaBlend,
    LSBM_MAX,
};

USTRUCT()
struct FLandscapeLayer
{
    GENERATED_USTRUCT_BODY()

    FLandscapeLayer()
           : Guid(FGuid::NewGuid())
           , Name(NAME_None)
           , bVisible(true)
           , bLocked(false)
           , HeightmapAlpha(1.0f)
           , WeightmapAlpha(1.0f)
           , BlendMode(LSBM_AdditiveBlend)
    {}

    FLandscapeLayer(const FLandscapeLayer& OtherLayer) = default;

    UPROPERTY(meta = (IgnoreForMemberInitializationTest))
    FGuid Guid;

    UPROPERTY()
    FName Name;

    UPROPERTY(Transient)
    bool bVisible;

    UPROPERTY()
    bool bLocked;

    UPROPERTY()
    float HeightmapAlpha;

    UPROPERTY()
    float WeightmapAlpha;

    UPROPERTY()
    TEnumAsByte<enum ELandscapeBlendMode> BlendMode;

    UPROPERTY()
    TArray<FLandscapeLayerBrush> Brushes;

    UPROPERTY()
    TMap<ULandscapeLayerInfoObject*, bool> WeightmapLayerAllocationBlend;
};

structFLandscapeLayersCopyTextureParams
```

```
{
    FLandscapeLayersCopyTextureParams(const FString&InSourceResourceDebugName,
FTextureResource* InSourceResource, const FString&InDestResourceDebugName,
FTextureResource* InDestResource, FTextureResource* InDestCPUResource,
          const FIntPoint&InInitialPositionOffset, int32 InSubSectionSizeQuad, int32
InNumSubSections, uint8 InSourceCurrentMip, uint8 InDestCurrentMip, uint32
InSourceArrayIndex, uint32 InDestArrayIndex)
          : SourceResourceDebugName(InSourceResourceDebugName)
          , SourceResource(InSourceResource)
          , DestResourceDebugName(InDestResourceDebugName)
          , DestResource(InDestResource)
          , DestCPUResource(InDestCPUResource)
          , InitialPositionOffset(InInitialPositionOffset)
          , SubSectionSizeQuad(InSubSectionSizeQuad)
          , NumSubSections(InNumSubSections)
          , SourceMip(InSourceCurrentMip)
          , DestMip(InDestCurrentMip)
          , SourceArrayIndex(InSourceArrayIndex)
          , DestArrayIndex(InDestArrayIndex)
    {}

    FString SourceResourceDebugName;
    FTextureResource* SourceResource;
    FString DestResourceDebugName;
    FTextureResource* DestResource;
    FTextureResource* DestCPUResource;
    FIntPoint InitialPositionOffset;
    int32 SubSectionSizeQuad;
    int32 NumSubSections;
    uint8 SourceMip;
    uint8 DestMip;
    uint32 SourceArrayIndex;
    uint32 DestArrayIndex;
};

UCLASS(MinimalAPI, showcategories=(Display, Movement, Collision, Lighting, LOD,
Input), hidecategories=(Mobility))
classALandscape :public ALandscapeProxy
{
    GENERATED_BODY()

public:
    ALandscape(const FObjectInitializer& ObjectInitializer);

    virtualvoidTickActor(float DeltaTime, ELevelTick TickType, FActorTickFunction&
ThisTickFunction) override;


    LANDSCAPE_API virtual ALandscape* GetLandscapeActor() override;
    LANDSCAPE_API virtualconst ALandscape* GetLandscapeActor() constoverride;
#if WITH_EDITOR


    LANDSCAPE_API boolHasAllComponent();


    LANDSCAPE_API staticvoidCalcComponentIndicesOverlap(const int32 X1, const int32
Y1, const int32 X2, const int32 Y2, const int32 ComponentSizeQuads,
          int32& ComponentIndexX1, int32& ComponentIndexY1, int32& ComponentIndexX2,
int32& ComponentIndexY2);


    LANDSCAPE_API staticvoidCalcComponentIndicesNoOverlap(const int32 X1, const int32
Y1, const int32 X2, const int32 Y2, const int32 ComponentSizeQuads,
```

```
            int32& ComponentIndexX1, int32& ComponentIndexY1, int32& ComponentIndexX2,
    int32& ComponentIndexY2);

       staticvoidSplitHeightmap(ULandscapeComponent* Comp, ALandscapeProxy* TargetProxy =
    nullptr, class FMaterialUpdateContext* InOutUpdateContext = nullptr, TArray<class
    FComponentRecreateRenderStateContext>* InOutRecreateRenderStateContext = nullptr, bool
    InReregisterComponent = true);

       //~ Begin UObject Interface.
       virtualvoidPreSave(constclass ITargetPlatform* TargetPlatform) override;
       virtualvoidPreEditChange(UProperty* PropertyThatWillChange) override;
       virtualvoidPostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent)
    override;
       virtualvoidPostEditMove(bool bFinished) override;
       virtualvoidPostEditUndo() override;
       virtualboolShouldImport(FString* ActorPropString, bool IsMovingLevel) override;
       virtualvoidPostEditImport() override;
       virtualvoidPostDuplicate(bool bDuplicateForPIE) override;
    #endif
       virtualvoidPostLoad() override;
       virtualvoidBeginDestroy() override;
       virtualvoidFinishDestroy() override;



       LANDSCAPE_API boolIsUpToDate() const;

    #endif


public:

#if WITH_EDITORONLY_DATA
       UPROPERTY(EditAnywhere, Category=Landscape)
       bool bCanHaveLayersContent = false;

       DECLARE_EVENT(ALandscape, FLandscapeBlueprintBrushChangedDelegate);
       FLandscapeBlueprintBrushChangedDelegate&OnBlueprintBrushChangedDelegate() { return
LandscapeBlueprintBrushChangedDelegate; }

       DECLARE_EVENT_OneParam(ALandscape, FLandscapeFullHeightmapRenderDoneDelegate,
UTextureRenderTarget2D*);
       FLandscapeFullHeightmapRenderDoneDelegate&OnFullHeightmapRenderDoneDelegate() {
return LandscapeFullHeightmapRenderDoneDelegate; }


       UPROPERTY()
       FGuid LandscapeSplinesTargetLayerGuid;


       FGuid EditingLayer;


       bool bGrassUpdateEnabled;

       UPROPERTY(TextExportTransient)
       TArray<FLandscapeLayer> LandscapeLayers;

       UPROPERTY(Transient)
       TArray<UTextureRenderTarget2D*> HeightmapRTList;

       UPROPERTY(Transient)
       TArray<UTextureRenderTarget2D*> WeightmapRTList;

private:
```

```
	FLandscapeBlueprintBrushChangedDelegate LandscapeBlueprintBrushChangedDelegate;
	FLandscapeFullHeightmapRenderDoneDelegate
LandscapeFullHeightmapRenderDoneDelegate;


	UPROPERTY(Transient)
	TSet<ULandscapeComponent*> LandscapeSplinesAffectedComponents;


	ILandscapeEdModeInterface* LandscapeEdMode;

		struct FLandscapeEdModeInfo
	{
		FLandscapeEdModeInfo();

		int32 ViewMode;
		FGuid SelectedLayer;
		TWeakObjectPtr<ULandscapeLayerInfoObject> SelectedLayerInfoObject;
		ELandscapeToolTargetType::Type ToolTarget;
	};

	FLandscapeEdModeInfo LandscapeEdModeInfo;

		bool bIntermediateRender;

	UPROPERTY(Transient)
	bool bLandscapeLayersAreInitialized;

	UPROPERTY(Transient)
	bool WasCompilingShaders;

	UPROPERTY(Transient)
	uint32 LayerContentUpdateModes;

	UPROPERTY(Transient)
	bool bSplineLayerUpdateRequested;

		class FLandscapeTexture2DArrayResource*
CombinedLayersWeightmapAllMaterialLayersResource;

		class FLandscapeTexture2DArrayResource*
CurrentLayersWeightmapAllMaterialLayersResource;

		class FLandscapeTexture2DResource*
WeightmapScratchExtractLayerTextureResource;


	class FLandscapeTexture2DResource* WeightmapScratchPackLayerTextureResource;

	TArray<FLandscapeLayersCopyTextureParams> PendingCopyTextures;
#endif

protected:
#if WITH_EDITOR
	FName GenerateUniqueLayerName(FName InName = NAME_None) const;
#endif
};

#if WITH_EDITOR
class LANDSCAPE_API FScopedSetLandscapeEditingLayer
{
public:
	FScopedSetLandscapeEditingLayer(ALandscape* InLandscape, const FGuid& InLayerGUID,
TFunction<void()> InCompletionCallback = TFunction<void()>());
```

```
        ~FScopedSetLandscapeEditingLayer();

private:
        TWeakObjectPtr<ALandscape> Landscape;
        FGuid PreviousLayerGUID;
        TFunction<void()> CompletionCallback;
};

    #endif
```

2. Об’єкт статуї янгола (StaticMeshActor.h)

```
    #pragmaonce

    #include"CoreMinimal.h"
    #include"UObject/ObjectMacros.h"
    #include"UObject/UObjectGlobals.h"
    #include"UObject/Object.h"
    #include"Misc/Guid.h"
    #include"Templates/SubclassOf.h"
    #include"Engine/EngineTypes.h"
    #include"UObject/ScriptMacros.h"
    #include"Interfaces/Interface_AssetUserData.h"
    #include"RenderCommandFence.h"
    #include"Components.h"
    #include"Interfaces/Interface_CollisionDataProvider.h"
    #include"Engine/MeshMerging.h"
    #include"Engine/StreamableRenderAsset.h"
    #include"Templates/UniquePtr.h"
    #include"StaticMeshResources.h"
    #include"PerPlatformProperties.h"
    #include"RenderAssetUpdate.h"
    #include"MeshTypes.h"

    #include"StaticMesh.generated.h"

    classFSpeedTreeWind;
    classUAssetUserData;
    classUMaterialInterface;
    classUNavCollisionBase;
    classUStaticMeshComponent;
    classUStaticMeshDescription;
    classFStaticMeshUpdate;
    structFMeshDescription;
    structFMeshDescriptionBulkData;
    structFStaticMeshLODResources;

    UENUM()
    enum ENormalMode
    {
        NM_PreserveSmoothingGroups,
        NM_RecalculateNormals,
        NM_RecalculateNormalsSmooth,
        NM_RecalculateNormalsHard,
        TEMP_BROKEN,
        ENormalMode_MAX,
    };

    UENUM()
    enum EImportanceLevel
    {
        IL_Off,
        IL_Lowest,
```

```
    IL_Low,
    IL_Normal,
    IL_High,
    IL_Highest,
    TEMP_BROKEN2,
    EImportanceLevel_MAX,
};

UENUM()
enum EOptimizationType
{
    OT_NumOfTriangles,
    OT_MaxDeviation,
    OT_MAX,
};

USTRUCT()
struct FStaticMeshOptimizationSettings
{
    GENERATED_USTRUCT_BODY()

          UPROPERTY()
    TEnumAsByte<enum EOptimizationType> ReductionMethod;


    UPROPERTY()
    float NumOfTrianglesPercentage;


    UPROPERTY()
    float MaxDeviationPercentage;

    UPROPERTY()
    float WeldingThreshold;

    UPROPERTY()
    bool bRecalcNormals;

    UPROPERTY()
    float NormalsThreshold;


    UPROPERTY()
    uint8 SilhouetteImportance;


    UPROPERTY()
    uint8 TextureImportance;


    UPROPERTY()
    uint8 ShadingImportance;


    FStaticMeshOptimizationSettings()
    : ReductionMethod( OT_MaxDeviation )
    , NumOfTrianglesPercentage( 1.0f )
    , MaxDeviationPercentage( 0.0f )
    , WeldingThreshold( 0.1f )
    , bRecalcNormals( true )
    , NormalsThreshold( 60.0f )
    , SilhouetteImportance( IL_Normal )
    , TextureImportance( IL_Normal )
    , ShadingImportance( IL_Normal )
```

```
    {
    }


    inlinefriend FArchive&operator<<( FArchive& Ar, FStaticMeshOptimizationSettings&
Settings )
    {
            Ar << Settings.ReductionMethod;
            Ar << Settings.MaxDeviationPercentage;
            Ar << Settings.NumOfTrianglesPercentage;
            Ar << Settings.SilhouetteImportance;
            Ar << Settings.TextureImportance;
            Ar << Settings.ShadingImportance;
            Ar << Settings.bRecalcNormals;
            Ar << Settings.NormalsThreshold;
            Ar << Settings.WeldingThreshold;

            return Ar;
    }


};
```

3. Звук співу птахів (AmbientSound.h)

```
#pragmaonce


#include"CoreMinimal.h"
#include"UObject/ObjectMacros.h"
#include"GameFramework/Actor.h"
#include"AmbientSound.generated.h"

UCLASS(AutoExpandCategories=Audio, ClassGroup=Sounds, hideCategories(Collision, Input,
Game), showCategories=("Input|MouseInput", "Input|TouchInput", "Game|Damage"),
ComponentWrapperClass)
class ENGINE_API AAmbientSound :public AActor
{
    GENERATED_UCLASS_BODY()

private:

    UPROPERTY(Category = Sound, VisibleAnywhere, BlueprintReadOnly, meta =
(ExposeFunctionCategories = "Sound,Audio,Audio|Components|Audio", AllowPrivateAccess =
"true"))
    class UAudioComponent* AudioComponent;
public:

    FString GetInternalSoundCueName();


#if WITH_EDITOR
    virtualvoidCheckForErrors() override;
    virtualboolGetReferencedContentObjects( TArray<UObject*>& Objects ) constoverride;
#endif
    virtualvoidPostRegisterAllComponents() override;


            UFUNCTION(BlueprintCallable, Category="Audio", meta=(DeprecatedFunction))
    voidFadeIn(float FadeInDuration, float FadeVolumeLevel = 1.f);
    UFUNCTION(BlueprintCallable, Category="Audio", meta=(DeprecatedFunction))
    voidFadeOut(float FadeOutDuration, float FadeVolumeLevel);
    UFUNCTION(BlueprintCallable, Category="Audio", meta=(DeprecatedFunction))
    voidAdjustVolume(float AdjustVolumeDuration, float AdjustVolumeLevel);
    UFUNCTION(BlueprintCallable, Category="Audio", meta=(DeprecatedFunction))
```

```
    voidPlay(float StartTime = 0.f);
    UFUNCTION(BlueprintCallable, Category="Audio", meta=(DeprecatedFunction))
    voidStop();


public:


    classUAudioComponent* GetAudioComponent() const { return AudioComponent; }
};
```

4. Скелет головного персонажа гри (SkeletalMesh.h)

```
#pragmaonce

#include"CoreMinimal.h"
#include"UObject/ObjectMacros.h"
#include"UObject/Object.h"
#include"Templates/SubclassOf.h"
#include"Interfaces/Interface_AssetUserData.h"
#include"RenderCommandFence.h"
#include"EngineDefines.h"
#include"Components.h"
#include"ReferenceSkeleton.h"
#include"GPUSkinPublicDefs.h"
#include"Animation/PreviewAssetAttachComponent.h"
#include"BoneContainer.h"
#include"Interfaces/Interface_CollisionDataProvider.h"
#include"EngineTypes.h"
#include"SkeletalMeshSampling.h"
#include"PerPlatformProperties.h"
#include"SkeletalMeshLODSettings.h"
#include"Animation/NodeMappingProviderInterface.h"
#include"Animation/SkinWeightProfile.h"
#include"Engine/StreamableRenderAsset.h"
#include"RenderAssetUpdate.h"

#include"SkeletalMesh.generated.h"

 classUAnimInstance;
 classUAssetUserData;
 classUBodySetup;
 classUMorphTarget;
 classUSkeletalMeshSocket;
 classUSkeleton;
 classUClothingAssetBase;
 classUBlueprint;
 classUNodeMappingContainer;
 classUPhysicsAsset;
 classFSkeletalMeshRenderData;
 classFSkeletalMeshModel;
 classFSkeletalMeshLODModel;
 classFSkeletalMeshLODRenderData;
 classFSkinWeightVertexBuffer;
 structFSkinWeightProfileInfo;
 classFSkeletalMeshUpdate;
#if WITH_APEX_CLOTHING

namespace nvidia
{
       namespace apex
       {
              class ClothingAsset;
       }
```

```
}
    #endif
    USTRUCT()
struct FBoneMirrorInfo
{
       GENERATED_USTRUCT_BODY()


       UPROPERTY(EditAnywhere, Category=BoneMirrorInfo, meta=(ArrayClamp =
"RefSkeleton"))
       int32 SourceIndex;

              UPROPERTY(EditAnywhere, Category=BoneMirrorInfo)
       TEnumAsByte<EAxis::Type> BoneFlipAxis;

       FBoneMirrorInfo()
              : SourceIndex(0)
              , BoneFlipAxis(0)
       {
       }

};

USTRUCT()
struct FBoneMirrorExport
{
       GENERATED_USTRUCT_BODY()

       UPROPERTY(EditAnywhere, Category=BoneMirrorExport)
       FName BoneName;

       UPROPERTY(EditAnywhere, Category=BoneMirrorExport)
       FName SourceBoneName;

       UPROPERTY(EditAnywhere, Category=BoneMirrorExport)
       TEnumAsByte<EAxis::Type> BoneFlipAxis;


       FBoneMirrorExport()
              : BoneFlipAxis(0)
       {
       }

};

USTRUCT()
struct ENGINE_API FSkeletalMeshClothBuildParams
{
       GENERATED_BODY()

       FSkeletalMeshClothBuildParams();


       UPROPERTY(EditAnywhere, Category = Target)
       TWeakObjectPtr<UClothingAssetBase> TargetAsset;


       UPROPERTY(EditAnywhere, Category = Target)
       int32 TargetLod;


       UPROPERTY(EditAnywhere, Category = Target)
       bool bRemapParameters;
```

```
	UPROPERTY(EditAnywhere, Category = Basic)
	FString AssetName;


	UPROPERTY(EditAnywhere, Category = Basic)
	int32 LodIndex;

		UPROPERTY(EditAnywhere, Category = Basic)
	int32 SourceSection;

	UPROPERTY(EditAnywhere, Category = Basic)
	bool bRemoveFromMesh;


	UPROPERTY(EditAnywhere, Category = Collision)
	TSoftObjectPtr<UPhysicsAsset> PhysicsAsset;
};

USTRUCT()
struct FSkeletalMeshLODInfo
{
	GENERATED_USTRUCT_BODY()


	UPROPERTY(EditAnywhere, Category=SkeletalMeshLODInfo)
	FPerPlatformFloat ScreenSize;
	UPROPERTY(EditAnywhere, Category=SkeletalMeshLODInfo, meta=(DisplayName="LOD
Hysteresis"))
	float LODHysteresis;


	UPROPERTY()
	TArray<int32> LODMaterialMap;

#if WITH_EDITORONLY_DATA

	UPROPERTY()
	TArray<bool> bEnableShadowCasting_DEPRECATED;


	UPROPERTY()
	TArray<FName> RemovedBones_DEPRECATED;
#endif


	UPROPERTY(EditAnywhere, Category = BuildSettings)
	FSkeletalMeshBuildSettings BuildSettings;

		UPROPERTY(EditAnywhere, Category = ReductionSettings)
	FSkeletalMeshOptimizationSettings ReductionSettings;


	UPROPERTY(EditAnywhere, Category = ReductionSettings)
	TArray<FBoneReference> BonesToRemove;


	UPROPERTY(EditAnywhere, Category = ReductionSettings)
	TArray<FBoneReference> BonesToPrioritize;


	UPROPERTY(EditAnywhere, Category = ReductionSettings, meta = (UIMin = "0.0",
ClampMin = "0.0"))
	float WeightOfPrioritization;
```

```
		UPROPERTY(EditAnywhere, Category = ReductionSettings)
	class UAnimSequence* BakePose;


	UPROPERTY(EditAnywhere, Category = ReductionSettings)
	class UAnimSequence* BakePoseOverride;


	UPROPERTY(VisibleAnywhere, Category= SkeletalMeshLODInfo, AdvancedDisplay)
	FString SourceImportFilename;

		UPROPERTY()
	uint8 bHasBeenSimplified:1;

	UPROPERTY()
	uint8 bHasPerLODVertexColors : 1;


	UPROPERTY(EditAnywhere, Category = SkeletalMeshLODInfo)
	uint8 bAllowCPUAccess : 1;



	UPROPERTY(EditAnywhere, AdvancedDisplay, Category = SkeletalMeshLODInfo,
meta=(EditCondition="bAllowCPUAccess"))
	uint8 bSupportUniformlyDistributedSampling : 1;

    #if WITH_EDITORONLY_DATA


	UPROPERTY()
	uint8 bImportWithBaseMesh:1;


	FGuid BuildGUID;

	ENGINE_API FGuid ComputeDeriveDataCacheKey(const FSkeletalMeshLODGroupSettings*
SkeletalMeshLODGroupSettings);
#endif

	FSkeletalMeshLODInfo()
		: ScreenSize(1.0)
		, LODHysteresis(0.0f)
		, WeightOfPrioritization(1.f)
		, BakePose(nullptr)
		, BakePoseOverride(nullptr)
		, bHasBeenSimplified(false)
		, bHasPerLODVertexColors(false)
		, bAllowCPUAccess(false)
		, bSupportUniformlyDistributedSampling(false)
#if WITH_EDITORONLY_DATA
		, bImportWithBaseMesh(false)
#endif
	{
#if WITH_EDITORONLY_DATA
		BuildGUID.Invalidate();
#endif
	}

};

USTRUCT()
struct FClothPhysicsProperties_Legacy
```

```
{
	GENERATED_USTRUCT_BODY()

	UPROPERTY()
	float VerticalResistance;
	UPROPERTY()
	float HorizontalResistance;
	UPROPERTY()
	float BendResistance;
	UPROPERTY()
	float ShearResistance;
	UPROPERTY()
	float Friction;
	UPROPERTY()
	float Damping;
	UPROPERTY()
	float TetherStiffness;
	UPROPERTY()
	float TetherLimit;
	UPROPERTY()
	float Drag;
	UPROPERTY()
	float StiffnessFrequency;
	UPROPERTY()
	float GravityScale;
	UPROPERTY()
	float MassScale;
	UPROPERTY()
	float InertiaBlend;
	UPROPERTY()
	float SelfCollisionThickness;
	UPROPERTY()
	float SelfCollisionSquashScale;
	UPROPERTY()
	float SelfCollisionStiffness;
	UPROPERTY()
	float SolverFrequency;
	UPROPERTY()
	float FiberCompression;
	UPROPERTY()
	float FiberExpansion;
	UPROPERTY()
	float FiberResistance;

	FClothPhysicsProperties_Legacy()
		: VerticalResistance(0.f)
		, HorizontalResistance(0.f)
		, BendResistance(0.f)
		, ShearResistance(0.f)
		, Friction(0.f)
		, Damping(0.f)
		, TetherStiffness(0.f)
		, TetherLimit(0.f)
		, Drag(0.f)
		, StiffnessFrequency(0.f)
		, GravityScale(0.f)
		, MassScale(0.f)
		, InertiaBlend(0.f)
		, SelfCollisionThickness(0.f)
		, SelfCollisionSquashScale(0.f)
		, SelfCollisionStiffness(0.f)
		, SolverFrequency(0.f)
		, FiberCompression(0.f)
		, FiberExpansion(0.f)
```

```
        , FiberResistance(0.f)
    {
    }
};
```

5. Фізичний матеріал води (PhysicalMaterial.h)

```
#pragmaonce

#include"CoreMinimal.h"
#include"UObject/ObjectMacros.h"
#include"UObject/Object.h"
#include"Engine/EngineTypes.h"
#include"EngineDefines.h"
#include"PhysxUserData.h"
#include"Vehicles/TireType.h"
#include"PhysicsEngine/PhysicsSettingsEnums.h"
#include"Physics/PhysicsInterfaceCore.h"
#include"PhysicalMaterial.generated.h"

structFPropertyChangedEvent;

namespace physx
{
    classPxMaterial;
}

USTRUCT()
struct FTireFrictionScalePair
{
    GENERATED_USTRUCT_BODY()


    UPROPERTY()
    class UTireType*                        TireType;


    UPROPERTY()
    float                                           FrictionScale;

    FTireFrictionScalePair()
        : TireType(NULL)
        , FrictionScale(1.0f)
    {
    }
};


UCLASS(BlueprintType, Blueprintable, CollapseCategories, HideCategories = Object)
class ENGINE_API UPhysicalMaterial :public UObject
{
    GENERATED_UCLASS_BODY()


    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=PhysicalMaterial,
meta=(ClampMin=0))
    float Friction;


    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = PhysicalMaterial, meta =
(editcondition = "bOverrideFrictionCombineMode"))
    TEnumAsByte<EFrictionCombineMode::Type> FrictionCombineMode;

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PhysicalMaterial)
    bool bOverrideFrictionCombineMode;
```

```
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = PhysicalMaterial,
meta=(ClampMin=0, ClampMax=1))
    float Restitution;


    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = PhysicalMaterial, meta =
(editcondition = "bOverrideRestitutionCombineMode"))
    TEnumAsByte<EFrictionCombineMode::Type> RestitutionCombineMode;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PhysicalMaterial)
    bool bOverrideRestitutionCombineMode;



    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = PhysicalMaterial,
meta=(ClampMin=0))
    float Density;


    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Advanced,
meta=(ClampMin=0.1, ClampMax=1))
    float RaiseMassToPower;

          UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Destruction)
    float DestructibleDamageThresholdScale;

    UPROPERTY
    class UDEPRECATED_PhysicalMaterialPropertyBase* PhysicalMaterialProperty;


    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = PhysicalProperties)
    TEnumAsByte<EPhysicalSurface> SurfaceType;

          UPROPERTY(VisibleAnywhere, Category = Deprecated)
    float TireFrictionScale;

          UPROPERTY(VisibleAnywhere, Category = Deprecated)
    TArray<FTireFrictionScalePair> TireFrictionScales;

public:

    FPhysicsMaterialHandle MaterialHandle;

#if WITH_PHYSX
    FPhysxUserData PhysxUserData;
#endif

    #if WITH_EDITOR
    virtualvoidPostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent)
override;
    staticvoidRebuildPhysicalMaterials();
#endif    virtualvoidPostLoad() override;
    virtualvoidFinishDestroy() override;

    FPhysicsMaterialHandle&GetPhysicsMaterial();


    static EPhysicalSurface DetermineSurfaceType(UPhysicalMaterial const*
PhysicalMaterial);
};
```

6. Модель та анімація вогнища (Emitter.h)

```
#pragmaonce

#include"CoreMinimal.h"
#include"UObject/ObjectMacros.h"
#include"GameFramework/Actor.h"
#include"Emitter.generated.h"

classUPhysicalMaterial;

DECLARE_DYNAMIC_MULTICAST_DELEGATE_FourParams( FParticleSpawnSignature, FName,
EventName, float, EmitterTime, FVector, Location, FVector, Velocity);

DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams( FParticleBurstSignature, FName,
EventName, float, EmitterTime, int32, ParticleCount);

DECLARE_DYNAMIC_MULTICAST_DELEGATE_SixParams( FParticleDeathSignature, FName,
EventName, float, EmitterTime, int32, ParticleTime, FVector, Location, FVector,
Velocity, FVector, Direction);

DECLARE_DYNAMIC_MULTICAST_DELEGATE_NineParams( FParticleCollisionSignature, FName,
EventName, float, EmitterTime, int32, ParticleTime, FVector, Location, FVector,
Velocity, FVector, Direction, FVector, Normal, FName, BoneName, UPhysicalMaterial*,
PhysMat);

UCLASS(hideCategories=(Activation,"Components|Activation",Input,Collision,"Game|Damage
"), ComponentWrapperClass)
class ENGINE_API AEmitter :public AActor
{
    GENERATED_UCLASS_BODY()

    virtualvoidGetLifetimeReplicatedProps(TArray< FLifetimeProperty >&
OutLifetimeProps) constoverride;

private:
    UPROPERTY(Category = Emitter, VisibleAnywhere, BlueprintReadOnly, meta =
(ExposeFunctionCategories =
"Particles|Beam,Particles|Parameters,Particles,Effects|Components|ParticleSystem,Rende
ring,Activation,Components|Activation", AllowPrivateAccess = "true"))
    class UParticleSystemComponent* ParticleSystemComponent;
public:

    UPROPERTY()
    uint32 bDestroyOnSystemFinish:1;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Emitter)
    uint32 bPostUpdateTickGroup:1;


    UPROPERTY(replicatedUsing=OnRep_bCurrentlyActive)
    uint32 bCurrentlyActive:1;

    UPROPERTY(BlueprintAssignable)
    FParticleSpawnSignature OnParticleSpawn;

    UPROPERTY(BlueprintAssignable)
    FParticleBurstSignature OnParticleBurst;

    UPROPERTY(BlueprintAssignable)
    FParticleDeathSignature OnParticleDeath;

    UPROPERTY(BlueprintAssignable)
    FParticleCollisionSignature OnParticleCollide;

#if WITH_EDITORONLY_DATA
```

```
private:
    UPROPERTY()
    class UBillboardComponent* SpriteComponent;

    UPROPERTY()
    class UArrowComponent* ArrowComponent;
public:

#endif

    UFUNCTION()
    virtualvoidOnParticleSystemFinished(class UParticleSystemComponent*
FinishedComponent);


    UFUNCTION()
    virtualvoid OnRep_bCurrentlyActive();


    UFUNCTION(BlueprintCallable, Category=Particles, meta=(DeprecatedFunction))
    voidActivate();
    UFUNCTION(BlueprintCallable, Category=Particles, meta=(DeprecatedFunction))
    voidDeactivate();
    UFUNCTION(BlueprintCallable, Category=Particles, meta=(DeprecatedFunction))
    voidToggleActive();
    UFUNCTION(BlueprintCallable, Category=Particles, meta=(DeprecatedFunction))
    boolIsActive() const;
    UFUNCTION(BlueprintCallable, Category=Particles, meta=(DeprecatedFunction))
    virtualvoidSetTemplate(class UParticleSystem* NewTemplate);
    UFUNCTION(BlueprintCallable, Category="Particles|Parameters",
meta=(DeprecatedFunction))
    voidSetFloatParameter(FName ParameterName, float Param);
    UFUNCTION(BlueprintCallable, Category="Particles|Parameters",
meta=(DeprecatedFunction))
    voidSetVectorParameter(FName ParameterName, FVector Param);
    UFUNCTION(BlueprintCallable, Category="Particles|Parameters",
meta=(DeprecatedFunction))
    voidSetColorParameter(FName ParameterName, FLinearColor Param);
    UFUNCTION(BlueprintCallable, Category="Particles|Parameters",
meta=(DeprecatedFunction))
    voidSetActorParameter(FName ParameterName, class AActor* Param);
    UFUNCTION(BlueprintCallable, Category="Particles|Parameters",
meta=(DeprecatedFunction))
    voidSetMaterialParameter(FName ParameterName, class UMaterialInterface* Param);

    voidAutoPopulateInstanceProperties();


    virtual FString GetDetailedInfoInternal() constoverride;


    virtualvoidPostActorCreated() override;
    virtualvoidPostInitializeComponents() override;
#if WITH_EDITOR
    virtualvoidCheckForErrors() override;
    virtualboolGetReferencedContentObjects( TArray<UObject*>& Objects ) constoverride;
    voidResetInLevel();
#endif
```

ДОДАТОК В
ТЕСТОВІ СЦЕНАРІЇ

Таблиця В.1 – Тестування запуску гри

Опис	Перевірка можливості розпочати гру		
Передумови	Гру запущено		
№	Дія	Очікуваний результат	Фактичний результат
1.	Натиснути кнопку «New Game» після запуску гри	Поява головного персонажа на ігровому оточенні	Головний персонаж з’являється на ігровому оточенні.

ТаблицяВ.2 – Перевірка кнопки діставання/хования зброї

Опис	Перевірка працездатності кнопки діставання та хования зброї головного персонажа		
Передумови	Гру запущено та головний персонаж знаходиться в ігровому оточенні		
№	Дія	Очікуваний результат	Фактичний результат
1.	Натиснути кнопку «2» під час сесії гри	Головний персонаж повинен дістати зброю	Головний персонаж дістав зброю
2.	Повторити натискання кнопки «2» під час сесії гри	Головний персонаж повинен сховати зброю	Головний персонаж сховав зброю

ТаблицяВ.3 – Перевірка переключання руху головного персонажа

Опис	Перевірка коректного перемикання руху між бігом та ходьбою головного персонажа
Передумови	Гру запущено та головний персонаж знаходиться у русі, а також виконує паралельно перемикання режиму між бігом та ходьбою

Продовження таблиці В.3

№	Дія	Очікуваний результат	Фактичний результат
1.	Під час руху головного персонажа натиснути та затримати кнопку «LeftShift»	Головний персонаж повинен розпочати швидкий рух	Головний персонаж розпочинає швидкий рух
2.	Під час руху головного персонажа відпустити кнопку «LeftShift»	Головний персонаж повинен повернутися в початковий звичайних рух	Головний персонаж повертається в початковий звичайний рух

ТаблицяВ.4 – Перевірка переключання руху головного персонажа

Опис	Перевірка відображання віджетів інтерфейсу гравця під час гри		
Передумови	Гру запущено та головний персонаж знаходиться на ігровому оточенні		
№	Дія	Очікуваний результат	Фактичний результат
1.	Звернути свою увагу на верхній лівий кут екрану	Повинні відображатись віджети інтерфейсу, що показують день, котра година та кількість здоров’я та мани	Відображаються віджети інтерфейсу, що показують день, котра година та кількість здоров’я та мани

ТаблицяВ.5 – Перевірка бойової системи головного персонажа

Опис	Перевірка працездатності бойової системи головного персонажа
Передумови	Гру запущено та головний персонаж знаходиться на ігровому оточенні

Продовження таблиці В.5

№	Дія	Очікуваний результат	Фактичний результат
1.	Періодично натискати ліву кнопку миші	Повинна відображатись анімація удару мечем головним персонажем по цілі, а також за її відсутності	Відображається анімація удару мечем головним персонажем по цілі, а також за її відсутності

ТаблицяВ.6 – Перевірка появи вікна відродження після смерті персонажа

Опис	Перевірка коректної роботи кнопки відродження після появи вікна смерті головного персонажа
Передумови	Гру запущено та головний персонаж знаходиться на ігровому оточенні під час бойової дії

№	Дія	Очікуваний результат	Фактичний результат
1.	Провести бойову дію з ворожим персонажем програвши бій та втративши все здоров’я	Повинна відбутися смерть головного персонажа та з’явитися вікно відродження	Смерть головного персонажа та поява вікна відродження
2.	Після появи вікна відродження натиснути лівою кнопкою миші на кнопку «Возродиться»	Головний персонаж повинен відродитися після плавного затухання екрану	Головний персонаж відроджується після плавного затухання екрану

ТаблицяВ.7 – Перевірка спрацювання тригеру звуку при вході в печеру

Опис		Перевірка на коректну зміну звуку при вході в печеру головним персонажем	
Передумови		Гру запущено та головний персонаж знаходиться у печері	
№	Дія	Очікуваний результат	Фактичний результат
1.	Персонаж заходить у печеру та знаходиться деякий час в ній	Спрацювання тригеру зміни звуку, починає програватись звук капель води	Спрацьовує тригер зміни звуку, починає програватись звук капель води
2.	Персонаж виходить з печери та знаходиться поза неї	Спрацювання тригеру зміни звуку, звук капель води перестає програватись	Спрацьовує тригер зміни звуку, звук капель води перестає програватись

ТаблицяВ.8 – Перевірка прийняття завдання від NPC

Опис		Перевірка на коректне прийняття квесту при взаємодії головним героєм з NPC	
Передумови		Гру запущено та головний персонаж знаходиться біля NPC	
№	Дія	Очікуваний результат	Фактичний результат
1.	Персонаж натискає кнопку «E» біля NPC	NPC повинен коректно взаємодіяти з головним персонажем	NPC коректно взаємодіє з головним персонажем
2.	Персонаж починає взаємодію з NPC	Завдання повинно коректно видаватись головному персонажу	Завдання коректно видається головному персонажу