Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Ігровий програмний застосунок в жанрі Open-World RPG «Northpoint»
(тема)

Виконав:
здобувач __4__ року навчання,
групи ПЗПІ-21-11

Владислав РАКІТІН
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. Ольга КАЛИНИЧЕНКО
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

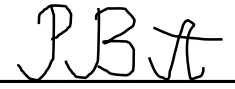
здобувачеві Ракітіну Владиславу Олексійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Ігровий програмний застосунок в жанрі Open-World RPG «Northpoint»
Затверджена наказом по університету від 19.05. 2025р. № 397 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 19.06.2022
3. Вихідні дані до роботи досвід гри у Open-World RPGs, ігрові механіки та дизайн-рішення успішних проектів жанру, фахова література з геймдизайну, технічна документація з C++ та Unreal Engine 5.
4. Перелік питань, що потрібно опрацювати в роботі
Дослідження жанрових особливостей Open-World RPG, формування концепції ігрового засосунку, визначення вимог до ігрової системи та користувацького досвіду, проєктування архітектури програмного застосунку, розробка, тестування і впровадження ігрового середовища.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05 – 22.05	*виконано*
2	Створення специфікації ПЗ	23.05 – 24.05	*виконано*
3	Проектування ПЗ	25.05 – 28.05	*виконано*
4	Розробка ПЗ	29.05 – 03.06	*виконано*
5	Тестування ПЗ	04.06 – 05.06	*виконано*
6	Оформлення пояснювальної записки	06.06 – 09.06	*виконано*
7	Підготовка презентації та доповіді	10.06 – 12.06	*виконано*
8	Нормоконтроль, рецензування, здача роботи у електронний архів	13.06 – 14.06	*виконано*
10	Попередній захист	15.06	*виконано*
11	Допуск до захисту у зав. кафедри	16.06	*виконано*

Дата видачі завдання «20» «травня» 2025р.

Здобувач  (підпис)

Керівник роботи _____ доц. Ольга КАЛИНИЧЕНКО
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 122 стор., 25 рис., 4 табл., 22 джерел.

РОЛЬОВА ГРА З ВІДКРИТИМ СВІТОМ, ІГРОВИЙ ПРОГРАМНИЙ ЗАСТОСУНОК, ГЕЙМДИЗАЙН, UNREAL ENGINE 5, C++, BLUEPRINTS

Об’єкт розробки – ігровий програмний застосунок в жанрі Open-World RPG «Northpoint».

Мета розробки – спроєктувати та реалізувати ігровий застосунок, що поєднує відкритий ігровий світ, інтерактивні механіки, систему розвитку персонажа, бойову систему, діалогову взаємодію та квестові завдання.

Метод рішення – середовище розробки Unreal Engine 5.5, мова програмування C++, система візуального скриптування Blueprints.

У результаті розробки створено повноцінний ігровий застосунок з реалізованими механіками керування персонажем, бойовою системою, взаємодією з ігровим середовищем та персонажами, інвентарем, системою прогресії навичок, діалогами, квестами та можливістю збереження ігрового прогресу.

ABSTRACT

OPEN-WORLD RPG, GAME APPLICATION, GAME DESIGN, UNREAL ENGINE 5, C++, BLUEPRINTS

The object of development is a game application in the Open-World RPG genre called “Northpoint”.

The purpose of the development is to design and implement a game application combining an open-world environment, interactive gameplay mechanics, character development systems, combat mechanics, dialogue interactions, and quest objectives.

Solution method – Unreal Engine 5.5 development environment, C++ programming language, and Blueprints visual scripting system.

As a result of the development, a complete game application was created, featuring character control mechanics, a combat system, interactive environmental and NPC interactions, inventory management, skill progression systems, dialogues, quests, and game progress saving functionality.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface

FPS – Frames Per Second

UML – Unified Modeling Language

UI – User Interface

UX – User Experience

ВСТУП

Ігри жанру Open-World RPG вже багато років залишаються одними з найбільш популярних та перспективних напрямів розвитку сучасної індустрії інтерактивних розваг. Головною причиною цього є можливість створення максимально занурюючого ігрового досвіду, що дозволяє гравцям відчути себе повноцінними мешканцями великого, відкритого та живого віртуального світу. Успішні приклади таких ігор, як «The Elder Scrolls V: Skyrim», «The Witcher 3: Wild Hunt», «Fallout 4» та інші, демонструють величезний потенціал цього жанру, що приваблює як ігрову аудиторію, так і розробників.

Актуальність розробки саме ігрового застосунку «Northpoint» обумовлена зростаючим попитом на інтерактивні розваги, які поєднують у собі високий рівень свободи дій, деталізований ігровий світ та захоплюючі механіки розвитку персонажа. Він покликаний вирізнятися серед аналогічних ігор завдяки впровадженню безперервного ігрового процесу під час взаємодії з меню інвентаря, перегляду карти чи записів щоденника, використанню природної прогресії персонажів замість класичного автолевелінгу, реалістичній навігації без маркерів завдань, а також детальній механіці прихованості, що враховує слухове та зорове сприйняття персонажів світу гри. Крім того, ігровий процес передбачає поглиблену систему бою, діалогові взаємодії з багатоваріантністю відповідей, комплексну систему розвитку персонажа з різноманітними навичками та ресурсами.

Мета роботи полягає у проєктуванні та реалізації ігрового застосунку «Northpoint», що включає створення відкритого світу, інтерактивної взаємодії з оточенням, розробку системи персонажа, механік бою, діалогів та квестів, які забезпечать цілісний ігровий досвід.

Галуззю застосування результатів роботи є сфера інтерактивних розваг, розробка комп'ютерних ігор та супутні галузі цифрової індустрії. Застосунок «Northpoint» може стати основою для подальшого розвитку комерційного продукту, наочно продемонструвати можливості сучасних технологій у сфері розробки ігор та слугувати прикладом реалізації передових рішень у жанрі Open-World RPG.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

1.1.1 Історія та еволюція рольових ігор (RPG)

Рольові ігри (RPG) займають особливе місце в історії ігрової культури, адже вони відрізняються не лише складністю механік, але й глибоким зануренням гравця у світ та наданням значної свободи дій. Виникнення та еволюція RPG нерозривно пов'язані з історією розвитку людської потреби в моделюванні реальності, створенні альтернативних сценаріїв та випробуванні різних соціальних ролей.

Першою значною подією в історії настільних RPG стало створення гри «Dungeons & Dragons» (D&D) у 1974 році Гері Гігаксом і Дейвом Арнесоном. Ця гра вперше дозволила гравцям управляти власноруч створеними персонажами в інтерактивному ігровому світі, що значно розширило рамки традиційних настільних ігор. D&D встановила фундаментальні принципи рольової гри, включаючи систему розвитку персонажів, боїв, досліджень та інтерактивного оповідання, керованого майстром гри (див. рис. 1.1) [1].

Рисунок 1.1 – Сесія гри «Dungeons & Dragons» (за даними [2])

Перші цифрові рольові ігри з’явилися незабаром після настільних, використовуючи переваги нових технологій. Однією з перших стала гра «Akalabeth: World of Doom» (1979), створена Річардом Гарріотом. Ця гра, хоч і була досить простою за сучасними стандартами, представила ключові RPG–елементи,

такі як дослідження підземель, управління інвентарем і розвиток персонажа [3]. Вона стала попередницею серії «Ultima», яка значно вплинула на розвиток жанру.

У 1980–х роках жанр RPG швидко еволюціонував, розділившись на західні RPG (наприклад, серії «Ultima», «Wizardry») та японські RPG (наприклад, «Dragon Quest», «Final Fantasy»). Західні RPG зазвичай надавали більше свободи у виборі дій та розвитку персонажа, тоді як японські зосереджувалися на глибокому сюжеті та яскравих персонажах [4].

1990–ті роки стали золотим віком для RPG. У цей період з'явилися знакові ігри, такі як «Baldur’s Gate» (1998), «Fallout» (1997) і «Diablo» (1996), що встановили нові стандарти у взаємодії з ігровим світом та в механіках управління персонажами. «Diablo», зокрема, запровадила концепцію action–RPG, що поєднує динамічну бойову систему з глибокою системою розвитку персонажів [5].

Початок 2000–х ознаменувався появою MMORPG (Massively Multiplayer Online RPG), серед яких найбільш відомими стали «Ultima Online» (1997), «EverQuest» (1999) та «World of Warcraft» (2004). Ці ігри відкрили нові можливості для взаємодії, дозволяючи тисячам гравців одночасно взаємодіяти у спільних ігрових світах, що кардинально змінило уявлення про можливості RPG [6].

Із розвитком цифрових технологій з'явилися нові виклики та можливості для жанру RPG. Одним із найбільших викликів стала потреба балансувати між відкритістю ігрового світу та цілісністю сюжетної лінії. Це спонукало розробників створювати складні системи, які дозволяють сюжету гнучко адаптуватися до дій гравця, не втрачаючи при цьому свого драматичного потенціалу [7].

1.1.2 Технологічні особливості Open–World RPG

Жанр Open–World RPG за останні десятиліття значно розширив свої межі завдяки стрімкому розвитку цифрових технологій. Особливістю цього жанру є вільне дослідження гравцем величезного, цілісного ігрового світу, що пропонує практично необмежені можливості для взаємодії та розвитку персонажа. Однак така свобода створює низку технологічних і дизайнерських викликів, які потребують нових, інноваційних рішень.

Головною характеристикою Open–World RPG є нелінійність і відкритість ігрового процесу. Ігри цього жанру надають гравцю можливість вільно пересуватися великими територіями, вибирати послідовність виконання завдань, а також впливати на сюжет власними рішеннями. До класичних прикладів належать «The Elder Scrolls V: Skyrim», «The Witcher 3: Wild Hunt» та «Fallout: New Vegas», які значною мірою визначили стандарти сучасного жанру [5].

Однією з найбільших переваг ігор з відкритим світом є так званий emergent gameplay (виникаючий геймплей). Завдяки складним взаємозв'язкам між численними системами гри, гравці стикаються з непередбачуваними ситуаціями, що не були заздалегідь запрограмовані розробниками. Це створює унікальні та особисті історії, що значно підвищує занурення та реіграбельність ігрового проєкту [8].

Однак відкритість світу породжує значні дизайнерські труднощі, зокрема потребу утримувати баланс між свободою гравця і логічною послідовністю основної сюжетної лінії. Для вирішення цієї проблеми розробники використовують модульний підхід до створення сюжетних ліній, при якому сюжет динамічно адаптується до дій гравця. Такий підхід дозволяє забезпечити високий ступінь свободи, не втрачаючи драматичності та логічної цілісності ігрових подій [7].

Ще одним важливим викликом є необхідність створення величезних обсягів контенту для наповнення відкритого світу. Це вимагає значних часових та фінансових ресурсів, що може спричинити проблему «розширення масштабу» (scope creep). Надмірна кількість контенту може призводити до його поверховості або повторюваності. Вирішення цієї проблеми досягається завдяки ретельному плануванню контенту та застосуванню процедурної генерації [9].

Процедурна генерація стала однією з найважливіших технологій у розробці Open–World RPG. Вона дозволяє автоматично створювати значні обсяги контенту (ландшафти, підземелля, квести, предмети), значно скорочуючи витрати часу на їх створення вручну. Яскравим прикладом використання цієї технології є гра «No Man's Sky», в якій процедурна генерація використовується для створення цілого всесвіту з мільярдами планет, кожна з яких унікальна [10].

Важливу роль у створенні переконливого відкритого світу відіграє штучний інтелект (AI). Сучасні AI–системи дозволяють не тільки створювати реалістичну поведінку неігрових персонажів (NPC), але й забезпечують динамічні взаємодії у світі гри. Одним із прикладів є система Radiant AI у грі «The Elder Scrolls IV: Oblivion», що дозволила персонажам вести автономний спосіб життя, приймаючи незалежні рішення залежно від ситуації в ігровому світі (див. рис. 1.2) [11].

Рисунок 1.2 – Реалізація технології Radiant AI у The Elder Scrolls IV: Oblivion (за даними [12])

Технологічною основою Open–World RPG є сучасні ігрові рушії, такі як Unreal Engine 5 та Unity, що дозволяють створювати великі безшовні світи з високоякісною графікою. Важливою технологією стала система динамічного завантаження (streaming), яка дозволяє завантажувати лише необхідні частини ігрового світу, забезпечуючи плавну гру без помітних завантажень і пауз [13].

Графічні технології також відіграють важливу роль, надаючи гравцю можливість зануритися у реалістично відтворений світ. Використання передових графічних рішень, таких як глобальне освітлення, реалістичні текстури та динамічні ефекти погоди, значно підвищують рівень занурення гравця в ігровий процес [14].

Таким чином, сучасні Open–World RPG є результатом поєднання передових дизайнерських рішень та технологій, що забезпечують глибокий, захоплюючий і унікальний ігровий досвід, який відповідає сучасним вимогам гравців.

1.2 Виявлення та вирішення проблем

1.2.1 Загальні вимоги та ключові виклики Open–World RPG

Рольові ігри з відкритим світом сьогодні є одними з найбільш складних і комплексних типів комп’ютерних ігор. Цей жанр пред'являє до розробників значні вимоги, пов’язані із забезпеченням повноцінної свободи пересування та дій гравця, створенням динамічного та деталізованого ігрового світу, а також розробкою складної системи взаємодії персонажів, сюжетних ліній і внутрішньоігрової економіки.

До основних вимог, які висуваються до ігор жанру Open–World RPG, можна віднести:

- масштабність та відкритість світу, який має бути цікавим та органічним;
- реалістичність взаємодії з об'єктами та навколишнім середовищем;
- глибину та деталізацію сюжету й побічних квестів;
- якісну інтеграцію геймплею з основною сюжетною лінією;
- високий ступінь свободи вибору і варіативності поведінки персонажа;
- продуману та збалансовану бойову систему, розвиток персонажів та економічні механіки.

Водночас жанр Open–World RPG має низку потенційних проблем, серед яких:

- технічна складність реалізації великих відкритих просторів без втрати продуктивності;
- ризик створення монотонного або недостатньо деталізованого ігрового світу;
- труднощі у забезпеченні балансу між свободою гравця та структурованістю сюжету;

- складність створення переконливої системи штучного інтелекту NPC;
- можливі технічні помилки та проблеми оптимізації, які впливають на комфортність ігрового процесу.

Виходячи з цього, важливим завданням для розробника є глибокий аналіз успішних прикладів аналогічних проєктів, визначення їх сильних та слабких сторін, а також формування власних підходів, спрямованих на подолання виявлених проблем.

1.2.2 Порівняльний аналіз аналогів

Для детальнішого розуміння та подальшого визначення напрямів розвитку власного проєкту необхідно проаналізувати успішні ігрові аналоги. Для аналізу обрано дві відомі гри: «The Elder Scrolls V: Skyrim» (жанр Open–World RPG) та «God of War: Ragnarok» (жанр Action–adventure). Незважаючи на відмінності у жанрах, обидві гри характеризуються значною популярністю, мають подібні технічні виклики, а також дозволяють розглянути різні підходи до реалізації ігрових механік, сюжету та дизайну світу, що є важливим для пошуку оптимальних рішень.

Перш за все, розглянемо організацію ігрового світу. У Skyrim світ характеризується максимальною відкритістю, що забезпечує значну свободу пересування, проте відсутність чіткої структурованості може призвести до того, що початок одних квестових ліній унеможливлює старт або завершення інших [15]. У свою чергу, God of War пропонує протилежний підхід: світ більш структурований і коридорний, що забезпечує послідовність сюжету, але суттєво обмежує відчуття свободи гравця [16].

Обидві гри використовують курсор у формі хреста для взаємодії з об'єктами у світі через наведення на них камери. У Skyrim ці дії відбуваються миттєво і незалежно від відстані, що робить їх нереалістичними. У God of War, навпаки, взаємодія проходить більш глибоко та інтенсивно, що інколи негативно впливає на динаміку самої гри.

Крім зазначеного, спільною проблемою обох ігор є реалізація інвентаря. У Skyrim та God of War відкриття інвентаря, карти чи інших елементів інтерфейсу супроводжується повною зупинкою гри. Такий підхід суперечить реалістичності й може виводити гравця з занурення у світ гри. Особливо це помітно у Skyrim, де подібна пауза дозволяє «від'їдатися» у середині бою десятками одиниць їжі, що стало навіть основою для численних жартів та мемів у спільноті гравців.

Щодо реалізації головних персонажів, обидва проєкти демонструють високу якість. Проте, Skyrim містить надмірну кількість навичок, деякі з яких практично не використовуються (наприклад, окремі категорії броні), тоді як у God of War навички зосереджені винятково на бойових аспектах.

Бойова система обох ігор відповідає стандартам ААА–проєктів свого часу, але містить деякі недоліки. У God of War дистанційна зброя здебільшого малоефективна в сутичках і застосовується переважно для розв'язання головоломок, а постійна підтримка напарника–лучника лише частково компенсує цю обмеженість. У Skyrim, навпаки, на пізніх етапах гри рукопашні стилі стають менш результативними порівняно з дальніми атаками, а збірка «Stealth Archer» виходить непропорційно потужною (див. рис. 1.3).

Рисунок 1.3 – Геймплей за прихованого стрільця у The Elder Scrolls V: Skyrim (за даними [17])

Ще однією характерною особливістю є система стелсу, притаманна переважно серії The Elder Scrolls. Вона додає геймплею цікаву тактичну складову,

однак має низку недоліків. На ранніх етапах гри NPC часто виявляють персонажа навіть тоді, коли він діє обережно, тоді як на пізніх – майже не реагують навіть на його очевидні дії. Крім того, персонаж може безкарно обшукувати чужі речі прямо перед очима NPC, що підриває логіку поведінки останніх, а прокачаний персонаж ще вільно їх обікрасти [18].

Також помітні проблеми у системі прогресії персонажів: у Skyrim гравець швидко стає занадто могутнім, що знижує інтерес до подальших викликів, а в God of War ключові механіки орієнтування стають доступними надто пізно, що ускладнює навігацію (див. рис. 1.4). Схожі труднощі, хоча й іншого характеру, виникають у Skyrim, де недостатнє інформування інтерфейсу навігації нерідко призводить до неочікуваних сутичок із небезпечними істотами.

Рисунок 1.4 – Геймплей перших годин гри God of War: Ragnarok (за даними [19])

Економічні системи в обох іграх також мають свої недоліки. У Skyrim гравець часто стикається з нестачею ресурсів у торговців, що змушує використовувати штучні механіки очікування для поновлення асортименту. God of War, у свою чергу, пропонує надто спрощену систему торгівлі, яка знижує значення ресурсного менеджменту.

Важливим аспектом є NPC. У Skyrim неігрові персонажі виглядають доволі реалістично й природно; однак акцент на глибокій опрацьованості окремих NPC обмежує їх загальну кількість, через що особливо великі міста здаються

малолюдними. У God of War персонажі NPC переважно виступають як елементи сюжету або масовка, без значущої інтерактивності.

Система збереження також має свої плюси та мінуси. Skyrim прагне зберігати практично все, що відбувається у світі, і це часто працює на користь глибокого занурення. Натомість God of War обмежується контрольними точками, що іноді призводить до втрати прогресу. Проте в Skyrim спостерігається зниження продуктивності на пізніх етапах гри: коли збереження накопичують надлишкову кількість змін (наприклад, предметів, викинутих у світі), час на збереження та завантаження суттєво зростає.

Квести та діалоги в обох проєктах реалізовані на високому рівні відповідно до жанрових особливостей. У Skyrim, як рольовій грі з відкритим світом, основний акцент зроблено на варіативності: більшість завдань мають кілька варіантів проходження, що дає змогу гравцеві самостійно формувати хід історії та впливати на події світу. У свою чергу, God of War як представник жанру action–adventure фокусується на глибокому сценарному наративі, який подається через якісно пророблені сценки, діалоги й озвучення, створюючи ефект повного занурення у сюжет.

З огляду на наведений аналіз, метою подальшої розробки є створення ігрового застосунку, який поєднуватиме переваги розглянутих підходів, мінімізуватиме їхні недоліки та надаватиме інструменти для формування збалансованого досвіду з відкритим світом, цілісною історією і глибоким сюжетом.

1.2.3 Шляхи подолання виявлених проблем і вдосконалення успішних рішень

Організація має поєднувати відкритість Skyrim та структурованість God of War. Гравцю надаватиметься повна свобода пересування і можливість виконання завдань у будь–якому порядку, але з урахуванням усіх можливих ситуацій, що виникають під час взаємодії сюжетних ліній. Таким чином, події одного квесту будуть органічно доповнювати або ініціювати етапи іншого, уникнувши конфліктів і логічних порушень.

Система взаємодії з об'єктами передбачає, що після підтвердження дії персонаж автоматично наблизиться до об'єкта, здійснить акт взаємодії, після чого буде виконано кінцеву дію. Цей процес може бути перервано за бажанням гравця чи через зовнішні умови. Також курсор повинен мати динамічну поведінку, реагуючи на доступні інтерактивні об'єкти, та вміючи визначати дистанцію до деяких з них.

Для забезпечення більшої реалістичності та занурення, всі дії в інвентарі, включно з переглядом карти, відбуватимуться в режимі реального часу без зупинки гри. Це вимагає від гравця уважності та швидкості, особливо під час боїв або перебування у небезпечних зонах. Таким чином, використання ресурсів та керування спорядженням стане частиною тактичного планування.

Навички персонажа повинні включати тільки ті, що безпосередньо впливають на геймплей і є доречними в ігровому світі. При цьому базові функції, такі як компас, будуть доступні з початку гри, а спеціальні можливості, наприклад виявлення прихованих ворогів, з’являться як додаткові здібності.

Бойова система має збалансовано підтримувати дистанційну та ближню зброю, забезпечуючи їх ефективність у відповідних ігрових ситуаціях. Окремою особливістю може стати можливість ведення рукопашних боїв, які не залежатимуть від наявності предметів в інвентарі та забезпечуватимуть можливість альтернативного стилю гри на початку проходження.

Механіка прихованого режиму повинна базуватись на реалістичних принципах. NPC будуть реагувати лише на ті дії, які вони безпосередньо чують або бачать, а не на абстрактну «зону виявлення». Якщо персонаж наближається в прихованому режимі до NPC та затримується перед ним, останній негайно реагуватиме. А споріднена механіка крадіжок враховуватиме вагу та цінність предметів, що впливатиме на ймовірність їх успішного викрадення.

Система прогресії повинна забезпечувати збереження інтересу гравця на всіх етапах гри без використання автолевелінгу. NPC матимуть можливість удосконалювати свої навички в процесі гри одночасно з гравцем, що сприяє реалістичному прогресу і розвитку ігрового світу.

Ігрова економіка буде орієнтована на збалансоване ціноутворення та актуальність ресурсів. Предмети матимуть логічну доступність та цінність, а ресурсний менеджмент буде достатньо глибоким для підтримки інтересу гравця без зайвого ускладнення ігрового процесу.

Неігрові персонажі повинні бути реалізовані на основі сучасних технологій штучного інтелекту, щоб забезпечити їхню реалістичну поведінку, адаптивність до змін в ігровому світі та автентичні взаємодії як із гравцем, так і з іншими NPC. Це створить враження живих мешканців світу гри.

Система збереження гри повинна бути розроблена таким чином, щоб фіксувати всі динамічні зміни, що відбуваються у світі, включаючи стан об’єктів, налаштування гравця, положення NPC, виконання квестів тощо. При цьому застосовувати механізми ігнорування зайвих або вже видалених об'єктів, які не впливають на подальший розвиток подій. Це дозволить зберегти точність завантаження без надмірного навантаження на систему.

Діалоги і квести мають поєднувати і варіативність та глибину сюжету Skyrim, і кінематографічну якість подачі God of War, забезпечуючи взаємодію, яка безпосередньо впливатиме на світ гри, а також емоційно насичені сюжетні моменти для гравця.

Таким чином, запропоновані рішення дозволять створити ігровий застосунок, який слугуватиме ефективним інструментом для реалізації повноцінного інтерактивного середовища з відкритим світом, глибоким сюжетом та продуманими механіками, надаючи основу для побудови збалансованого ігрового досвіду.

1.3 Постановка задачі

1.3.1 Зміст задачі та обґрунтування вибору середовища розробки

Під час розробки програмного ігрового застосунку у жанрі Open–World RPG виникає потреба у формуванні цілісної концепції гри, що включає ключові механіки взаємодії, поведінки персонажів, навігації, збереження прогресу та візуального представлення ігрового світу. Для визначення загальної структури

майбутнього проєкту необхідно сформулювати основні положення задачі, які охоплюють принципи функціонування ігрових систем, очікувану логіку дій користувача та способи інтеграції цих елементів у єдину архітектуру.

Користувач керуватиме ігровим персонажем, маючи змогу вільно переміщуватися у відкритому світі в усіх напрямках. Передбачається реалізація кількох режимів пересування: звичайної ходи, бігу з можливістю зручного перемикання між цими двома режимами, тихого руху зі зниженням швидкості та рівня шуму, а також спринту, який дозволятиме швидко долати відстані за рахунок витрати запасу сил і тимчасової неможливості утримувати зброю. Під час спринту має бути передбачена можливість фізичного зіткнення з іншими персонажами або об’єктами, що сприйматиметься як прояв агресії. Крім основного пересування, персонаж матиме змогу стрибати, причому падіння з великої висоти призводитиме до втрати частини здоров'я. Також планується реалізація системи перекатів, які гравець зможе використовувати для ухилення від атак або для безшумного переміщення в режимі прихованості за наявності відповідної навички.

Для забезпечення повноцінного ігрового досвіду у межах Open–World RPG застосунку необхідна реалізація широкого спектра взаємопов’язаних механік, що охоплюють пересування, бойову систему, інвентаризацію, навігацію, діалоги, квести, збереження та інші аспекти взаємодії у відкритому світі.

Персонаж керуватиметься гравцем за допомогою камери. У русі ротація персонажа фіксовано слідуватиме за напрямком камери, тоді як під час зупинки персонаж зберігатиме положення, а камера дозволятиме оглядати його з різних боків. Планується підтримка двох режимів огляду: від третьої особи (за замовчуванням) для зручнішого орієнтування у просторі та від першої особи для створення більш глибокого ефекту присутності. Має бути забезпечене як плавне, так і миттєве перемикання між режимами із запам’ятовуванням останнього положення камери. У певних ситуаціях передбачається використання кінематографічної камери зі зміщенням фокусу на важливі об’єкти або події. Якщо гравець довго не виконуватиме ніяких дій, камера почне повільно обертатися

навколо персонажа, а сам персонаж природно реагуватиме на очікування, наприклад, змінюючи позу або проявляючи дрібні рухи.

За допомогою курсора у вигляді перехрестя, розташованого в центрі екрана, повинна здійснюватися взаємодія гравця з об'єктами навколишнього світу. Наведення курсора на об’єкти дозволятиме здійснювати відповідні дії з ними. Планується створення різноманітних інтерактивних об'єктів, таких як предмети для підняття, сундуки, двері, елементи меблів (стілець, ліжко) та інші неігрові персонажі. Курсор має бути чутливим до об'єктів попереду: у стандартному стані відображатиметься у вигляді крапки, а при можливості взаємодії – змінюватиме форму на перехрестя з динамічною реакцією на присутність об’єкта. Для режиму прихованості форма курсора також змінюватиметься залежно від рівня видимості персонажа, надаючи гравцеві візуальні підказки про його помітність для навколишнього світу.

На ігрових рівнях планується розміщення динамічних об'єктів, з якими персонаж матиме можливість взаємодіяти. Передбачено реалізацію різних типів предметів, які гравець зможе піднімати та переносити до власного інвентаря: зброї та броні, продуктів харчування, зіль, ключів, відмичок, звичайних предметів та ігрової валюти (золота), що використовуватиметься для торгівлі. Окрему категорію становитимуть сундуки, які слугуватимуть для зберігання або вилучення предметів. Двері можуть бути як відкритими, так і зачиненими на замок; для відкриття зачинених дверей потрібен буде відповідний ключ або застосування відмичок у поєднанні з навичкою злому, що у внутрішній логіці світу вважатиметься незаконною дією. Також передбачається інтерактивність меблів, зокрема можливість зайняти стільці для читання, сортування предметів або розвитку навичок, а також використання ліжок для відпочинку та сну протягом ночі. Окрім предметних об'єктів, гравець матиме змогу вступати у взаємодію з іншими персонажами: спілкуватися, здійснювати торгівлю, а у режимі прихованості – вчиняти крадіжку.

Передбачається розробка системи управління предметами через інвентар, який має забезпечувати надійне зберігання всіх підібраних об’єктів. Гравець

повинен мати можливість використовувати предмети або викидати їх перед собою, обираючи попередньо необхідну кількість за допомогою спеціального елементу інтерфейсу. При використанні зброя та броня мають автоматично екіпіруватися у відповідні слоти екіпірування, а застосування зіль повинно надавати тимчасові ефекти персонажу. Окремі предмети мають використовуватися в особливих умовах, залежно від їх функціонального призначення.

У межах ігрового світу планується створити контейнери за принципом інвентаря, які дозволятимуть безпечно зберігати або вилучати предмети. Окремим типом контейнера має стати меню торгівлі, через яке гравець зможе купувати або продавати предмети за внутрішню ігрову валюту – золото. Кожен предмет повинен мати власну вартість, що визначатиме його економічну цінність у грі. Крім цього, предмети матимуть показник ваги: у випадку перевищення допустимої ваги інвентаря у персонажа повинен виникати стан перевантаження, що на початку впливатиме на швидкість пересування та далі обмежуватиме можливість піднімати нові предмети або взаємодіяти з окремими об'єктами. Іншим видом контейнера стане інвентар інших персонажів, який у режимі прихованості гравець зможе обшукувати з метою крадіжки.

Спорідненим до інвентаря елементом інтерфейсу, що має розташовуватися поруч із ним, буде мапа, яка разом із компасом на головному екрані повинна забезпечувати навігацію для гравця у відкритому світі. Мапа має відображати зменшену версію ігрового рівня і бути доступною у двох варіантах відображення: площинному (ортографічному) вигляді за замовчуванням та перспективному вигляді, між якими гравець зможе перемикатися. На мапі повинні відображатися місцезнаходження персонажа, напрямок його руху та позначені локації; ті локації, які вже були відвідані, мають супроводжуватися відповідними підписами. Передбачається можливість одним натисканням миттєво перемістити фокус карти на поточне розташування персонажа та встановлювати власні позначки, що будуть додатково відображатися на компасі.

Компас має бути розташований на головному екрані та слугувати для орієнтування у просторі за допомогою відміток сторін світу і позначень

найближчих локацій. При наближенні персонажа до нової локації має з’являтися візуальне повідомлення про її відкриття. Власні позначки гравця (персональні маркери) також мають бути відображені на компасі й автоматично зникати після досягнення місця призначення.

Оскільки дослідження світу неминуче супроводжується зустрічами з потенційно небезпечними і вороже налаштованими істотами, виникає потреба у впровадженні бойової системи, що має забезпечити гравцеві засоби для захисту та ведення бою. Передбачається реалізація базової бойової системи, що охоплюватиме три основні типи бою: рукопашний, ближній та дальній.

Рукопашний бій має бути доступним без потреби у спеціальному екіпіруванні й забезпечувати базові удари руками. Ближній бій передбачає використання холодної зброї, насамперед мечів, що потребуватиме екіпірування зброї у відповідний слот. Дальній бій має реалізовуватися через використання лука, що потребуватиме як самого лука, так і наявності стріл у інвентарі. У майбутньому можливе розширення системи до четвертого типу – магії. Базові атаки повинні бути доступні для кожного типу бою: удари лівою та правою рукою або мечем для ближнього бою, постріли для дистанційного з лука. Подальший розвиток персонажа має розблоковувати нові види атак та бойові особливості.

Усі успішні атаки, що вражають ціль, спроможну приймати урон, повинні наносити шкоду здоров'ю, атаки по голові завдають подвійного урону. Окрім нанесення шкоди, удари мають супроводжуватися відповідною реакцією на влучення, яка змінюватиметься залежно від типу зброї і напрямку удара. Наприклад, удар мечем повинен переривати процес натягування тятиви на луці під час прицілювання. У ближньому бою гравець повинен мати можливість здійснювати блокування ворожих атак за допомогою холодної зброї, але такий блок буде частково поглинати шкоду, обмежувати пересування персонажа під час його виконання та поступово виснажувати запас сил. У випадку, якщо рівень здоров’я персонажа знижується до нуля або нижче, настає його смерть.

Для забезпечення повноцінного функціонування і прогресії основних систем гри виникає необхідність у введенні характеристик і навичок. Основними

ресурсами персонажа мають стати здоров’я, запас сил та, у перспективі, запас магічної енергії. Загальну силу та захищеність персонажа гравця повинні відображати показники атаки й захисту, а від відіграшу залежити параметри репутації, штрафу та стилю гри.

Передбачається створення системи навичок, що відповідатимуть ключовим механікам гри. До переліку навичок планується додати: витривалість, ближній бій, стрілецька справа, володіння бронею, прихованість, спритність, красномовство, а в майбутньому – руйнування, виклик та відновлення. Прогрес кожної навички має вимірюватися у діапазоні від 0 до 100, де 0 є базовим рівнем персонажа на початку гри, а 100 – максимально можливим.

Рівень розвитку навичок повинен безпосередньо впливати на ефективність відповідних дій у грі. Протягом прогресії планується відкриття особливих здібностей: одна базова та шість функціональних для кожної навички. Для забезпечення загальної системи прогресії персонаж отримає власний рівень: з кожним підвищенням передбачається надання одного очка умінь для розблокування нових здібностей. Досягнення максимального рівня персонажа має відповідати максимальному розвитку всіх навичок і розблокуванню всіх здібностей. Крім того, при кожному підвищенні рівня гравцеві надаватиметься можливість збільшити максимальне значення одного з основних ресурсів: здоров’я, запасу сил або магічної енергії.

Оскільки ігровий світ має створювати враження реального, населеного простору, необхідно забезпечити його взаємодіючими об'єктами не лише у вигляді середовища, а й через присутність інших істот. Гравець у грі є лише однією з частин цього світу, який також населяють численні неігрові персонажі, що мають управлятися системою штучного інтелекту. Для досягнення високого рівня переконливості необхідно, щоб кожен NPC володів усіма основними властивостями, притаманними ігровому персонажу, і був такою ж рівноправною частиною світу.

Штучний інтелект має забезпечувати гнучку та реалістичну поведінку NPC залежно від їхніх ролей та обставин. Зокрема, у бойових ситуаціях неігрові

персонажі повинні мати можливість користуватися всіма видами зброї, доступними гравцеві, обираючи найбільш доцільну залежно від умов бою та вмісту власного інвентаря. Водночас за межами бою NPC мають демонструвати індивідуальні життєві цикли, обумовлені роллю у світі: бути торговцями, з якими гравець може здійснювати торгівлю; стражниками, що забезпечують порядок і реагують на злочини; звичайними мешканцями поселень або агресивно налаштованими супротивниками – бандитами, дикарями та іншими.

Для досягнення динамічності ігрового середовища також має бути впроваджено механізм зміни часу доби з відповідними циклами дня і ночі, що безпосередньо впливатимуть на поведінку NPC, їхню активність та доступні взаємодії з гравцем.

Одним із варіантів такої взаємодії і основним через який вона відбуватиметься стануть діалоги. На початку діалогу перспектива камери персонажа повинна автоматично фокусуватися на співрозмовнику, який поступово промовлятиме свої репліки. Гравцеві надаватиметься можливість обирати варіанти відповідей серед запропонованих, що безпосередньо впливатимуть на подальший перебіг бесіди. Усі варіанти повинні підводити розмову до логічної точки завершення, після чого співрозмовники розходитимуться. Якщо ж гравець перерве діалог передчасно, така дія може бути сприйнята як акт агресії.

Серед варіантів відповідей передбачається наявність реплік різного характеру: запевнень, що можуть допомогти переконати NPC і здобути додаткову інформацію чи послуги; пропозицій оплати для досягнення бажаного результату; а також погроз, які можуть викликати недружню або ворожу реакцію співрозмовника. Діалогова система повинна забезпечувати можливість побудови глибоких бесід як за змістовим наповненням, так і за складністю структурних гілок. Їхній перебіг повинен адаптуватися до характеристик персонажа гравця, його попередніх рішень у розмові, змін у стані світу та особистих властивостей співрозмовника.

Лінії спілкування з неігровими персонажами будуть утворювати рівні завдань або квестові лінійки, що мають відстежуватися в окремому журналі.

Система не має бути жорстко лінійною: гравцеві доведеться самостійно відстежувати розвиток подій, уважно читати записи та орієнтуватися у власних діях на основі отриманої інформації.

Базова структура кожного запису в журналі повинна містити ігрову дату і час створення запису, заголовок, що коротко описує подію або завдання, та розгорнутий текстовий опис. Такий підхід має дозволити не лише реєструвати основні квестові завдання, але й фіксувати окремі важливі події, що стаються з гравцем у світі. Сам журнал повинен бути побудований у вигляді книги, сторінки якої будуть відображатися з двох боків екрану та гортатися за допомогою зручного елемента управління. Передбачається можливість перегортати як окремі сторінки, так і швидко переходити через кілька записів одночасно для зручності пошуку потрібної інформації.

Тепер, для забезпечення збереження прогресу гравця у відкритому світі має бути розроблена система збережень. Оскільки ігровий процес включає численні взаємопов'язані механіки, система збережень повинна фіксувати всі суттєві події та зміни, що відбулися у грі на момент збереження. Передбачається, що система має забезпечувати повне відновлення стану гри: починаючи від місця розташування персонажа та його поточних характеристик, закінчуючи станом різних інших об’єктів, записами в журналі, ходом діалогів і взаємодій з NPC, зміненою структурою світу та іншими важливими аспектами. Гравець повинен мати змогу у будь–який момент повернутися до раніше збереженої точки прогресу, зберігаючи безперервність і цілісність ігрового досвіду.

В останню чергу, для управління основними процесами гри передбачається створення меню паузи, яке має забезпечувати можливість тимчасового припинення ігрового процесу, а також надавати гравцеві доступ до базових функцій керування прогресом: створення нового збереження, завантаження раніше збереженої сесії або вихід з ігрового застосунку. Меню повинно бути доступним у будь–який момент перебігу гри, забезпечуючи зручність та безпеку для гравця під час користування системою збережень та управління ігровим процесом.

З огляду на складність розроблюваних механік та високі вимоги до графічної якості, масштабованості й інтерактивності, для реалізації проєкту було обрано ігровий рушій Unreal Engine 5.5. Його вибір зумовлений наявністю необхідного функціоналу для втілення вказаних систем: рушій підтримує створення великих безшовних світів, використовує сучасні графічні технології (глобальне освітлення, високоякісні текстури, динамічні погодні ефекти) і має вбудовану систему стрімінгового завантаження контенту, що є критично важливим для open-world проєктів. Окрім того, рушій забезпечує ефективну роботу як із мовою програмування C++, так і з візуальним скриптуванням Blueprints, що дозволяє поєднувати низькорівневу продуктивність із високорівневою логікою, спрощуючи розробку та тестування окремих механік.

Запропоновані рішення дозволять створити інтегрований ігровий застосунок, що забезпечуватиме гравцеві глибокий, цілісний та захопливий досвід знаходження у відкритому світі.

1.3.2 Мета роботи та постановка завдання

Метою даної роботи є розробка програмного ігрового застосунку у жанрі Open–World RPG, що вміщуватиме відкритий ігровий світ, стилізований під холодне середньовічне фентезі із засніженими ландшафтами, суворим кліматом та відповідною архітектурою. Відповідно до стилістики, проект отримає назву «Northpoint», яка символізуватиме віддалену та малодосліджену північну територію, що виступатиме основним місцем подій гри.

Для досягнення поставленої мети необхідно виконати такі основні завдання:

- реалізація зручної системи переміщення з різними режимами руху та інтерактивною камерою;
- створення механізму взаємодії з навколишнім середовищем через інтерактивні об’єкти та персонажів;
- розробка функціональної системи управління інвентарем та екіпіюванням;
- побудова інтерактивної карти та системи навігації;

- впровадження бойової системи з підтримкою рукопашного, ближнього і дальнього бою;
- створення механік прихованості та крадіжок з реалістичною поведінкою NPC;
- розробка системи характеристик персонажа, його ресурсів та навичок із поступовою прогресією;
- реалізація переконливої поведінки NPC із використанням штучного інтелекту;
- впровадження інтерактивної діалогової системи з варіативними репліками;
- організація квестової системи з гнучким журналом завдань;
- забезпечення ефективної системи збереження та відновлення ігрового стану;
- створення інтуїтивного інтерфейсу користувача з підтримкою зручного управління ігровим процесом.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Загальна характеристика продукту

Програмний застосунок отримає назву Northpoint (англ. «Північна точка» / укр. «Нортпоінт»), а його символом гри стане емблема у вигляді стилізованого орла, розміщеного в центрі орнаментального кільця зі скандинавськими візерунками на тлі холодної палітри.

Northpoint є однокористувацьким ігровим застосунком у жанрі Open–World RPG, що дозволяє користувачу досліджувати відкритий світ із відчуттям самоти та віддаленості. Застосунок створюється для забезпечення реалістичної інтерактивної взаємодії з оточенням без необхідності постійної підтримки мережевого з'єднання.

Ігровий світ має бути стилізованим під середньовічне фентезі, наближене по духу до скандинавської культури періоду вікінгів. Дія розгортатиметься у вигаданому всесвіті з суворим кліматом, крижаними ландшафтами, віддаленими поселеннями та архітектурою, що відображає простоту і витривалість місцевих народів.

Ключовими особливостями застосунку визначаються:

- виконання ігрових дій (керування інвентарем, перегляд карти, читання щоденника) без зупинки ігрового процесу
- заміна системи автолевелінгу та рівневої адаптації зон; прогресія персонажів і NPC здійснюється природним шляхом через дії у світі;
- відсутність явного навігаційного супроводу для завдань: інформація щодо розвитку подій фіксується у щоденнику без маркерів напрямку;
- реалістична система прихованості з моделюванням зорового й слухового сприйняття NPC, що залежить від реальних факторів видимості та шуму.

Цільова аудиторія застосунку визначається як дорослі користувачі, переважно чоловічої статі, віком від 16 років, зацікавлені у дослідженні фентезійних світів із суворою атмосферою. Досвід користувачів у рольових іграх не є обов'язковим.

З огляду на наявність бойових сцен із відображенням крові та поранень, продукт орієнтується на вікову категорію 16+ відповідно до європейських та міжнародних стандартів рейтингування інтерактивного контенту.

Платформою для запуску застосунку визначаються персональні комп'ютери з операційними системами Windows 10 та Windows 11, із можливістю подальшого розширення підтримки на інші платформи.

2.2 Технічні особливості та вимоги

Розробка програмного застосунку здійснюватиметься на основі рушія Unreal Engine 5.5 із використанням стандартного набору інструментів і технологій. Основними технічними особливостями застосунку визначено:

- використання рендерингового API DirectX 12 як єдиного цільового графічного інтерфейсу;
- активація системи глобального освітлення Lumen для створення реалістичних ефектів непрямого світла й відбиттів у режимі реального часу;
- використання класичної системи рівнів деталізації (LOD) для моделей замість технології Nanite, що обумовлено потребою у підвищеній продуктивності та економії пам’яті в умовах великого відкритого світу;
- організація структури світу за допомогою технології World Partition, яка забезпечує динамічне стрімінгове завантаження сегментів карти без потреби в ручному розділенні рівнів;
- застосування традиційних shadow maps для відтворення тіней замість Virtual Shadow Maps;
- відмова від використання MetaSounds, MetaHumans та інших експериментальних або складних модулів у межах поточного проекту.

Застосунок орієнтується на досягнення стабільної частоти кадрів не менше 60 FPS на цільових системах, при базовій роздільній здатності 1920×1080 (Full HD). При рекомендованій конфігурації передбачається забезпечення комфортного функціонування у вищих роздільностях, включно із 4K.

Для забезпечення стабільної роботи програмного застосунку визначено такі орієнтовні апаратні вимоги:

а) мінімальні системні вимоги:

1) процесор: Intel Core i5–7500 або еквівалентний AMD Ryzen 5 1400;
2) відеокарта: NVIDIA GTX 960;
3) оперативна пам’ять: 8 ГБ;
4) накопичувач: HDD або SSD із вільним обсягом пам’яті не менше 5 ГБ;
5) операційна система: Windows 10 (64–bit).

б) рекомендовані системні вимоги:

1) процесор: Intel Core i5–10400 або еквівалентний AMD Ryzen 5 3600;
2) відеокарта: NVIDIA GTX 1650 або краще;
3) оперативна пам’ять: 16 ГБ;
4) накопичувач: SSD або HDD із вільним обсягом пам’яті понад 10 ГБ;
5) операційна система: Windows 10 або Windows 11 (64–bit).

Запуск застосунку передбачає використання виключно 64–бітних операційних систем та відповідного програмного середовища.

Стандартними засобами керування визначаються клавіатура та миша, із можливістю подальшого розширення підтримки на геймпади стандарту XInput у майбутніх версіях. Для забезпечення повноти ігрового досвіду рекомендованим є використання стереонавушників або гарнітури із підтримкою об'ємного звуку.

Використання окремого інсталятора не передбачається; запуск здійснюватиметься через наданий виконуваний файл у складі проекту. Для роботи застосунку не вимагається постійне підключення до Інтернету, а також відсутні обмеження щодо типу накопичувача, оскільки застосунок не накладає суворих вимог на швидкість завантаження ресурсів.

2.3 Організація користувацького інтерфейсу

Користувацький інтерфейс програмного застосунку має забезпечувати функціональну зручність, простоту сприйняття та візуальну стійкість на всіх підтримуваних роздільностях дисплеїв.

Композиція елементів інтерфейсу повинна підпорядковуватися єдиній логічній структурі з чіткою візуальною ієрархією. Всі елементи мають бути згруповані у межах єдиного простору без надлишкового розкидання по екрану, із дотриманням балансу між інформативністю та чистотою оформлення. Основна інформація повинна розташовуватися у центральній або прилеглій до центральної зоні для мінімізації переміщення погляду користувача.

Інтерфейс має забезпечувати пропорційне масштабування на дисплеях Full HD (1920×1080), WQHD (2560×1440) та 4K (3840×2160) без втрати чіткості шрифтів, піктограм і основних елементів. Всі елементи повинні зберігати правильні співвідношення розмірів незалежно від роздільної здатності.

Текстові елементи інтерфейсу мають бути виконані читабельним беззарубковим шрифтом середньої товщини із базовим розміром не менше 16 пунктів для основного тексту.

Загальна кольорова палітра інтерфейсу повинна базуватися на приглушених холодних тонах із використанням виразних акцентів для підкреслення ключових функціональних елементів. Колірне рішення має бути стабільним та витриманим, уникати перенасичених відтінків і різких переходів, підтримуючи загальну атмосферу стилізації. Фон інформаційних елементів повинен мати затемнену або напівпрозору текстуру, що забезпечує достатню відділеність інформації від динамічного середовища та покращує загальну читабельність у русі.

Інтерфейс повинен гарантувати мінімізацію візуальних ефектів, здатних спричинити дискомфорт, таких як надмірні спалахи світла, сильне розмиття руху або різкі зміни масштабу.

2.4 Доступність застосунку для гравця

Програмний застосунок має забезпечувати доступність для користувачів із різними рівнями когнітивних та фізичних навичок без спрощення основних ігрових механік або зниження рівня загального виклику.

Ігровий світ і механіки мають бути спроєктовані таким чином, щоб гравець міг поступово й природно засвоювати основні правила взаємодії через вивчення

середовища, подій і наслідків власних дій. Замість прямого навчання за допомогою системних спливаючих повідомлень або нав'язливих підказок, застосунок повинен забезпечувати інтуїтивну зрозумілість базових взаємодій через поведінку об'єктів і реакції світу.

Розвиток навичок і прогресія персонажа мають ґрунтуватися на закономірностях, доступних для логічного розуміння, без необхідності запам'ятовування великої кількості спеціальних умов або винятків.

Уся важлива інформація повинна передаватися гравцю через природні елементи ігрового середовища — візуальні зміни, звукові ефекти, поведінку NPC — без потреби в окремих підказках, спливаючих вікнах або системних поясненнях.

Механіки управління персонажем, бойові дії та дослідження світу мають бути реалізовані так, щоб не вимагати високої точності моторних навичок або надшвидкої реакції. Користувач повинен мати змогу виконувати основні дії плавно, без необхідності частої координації складних комбінацій клавіш або інтенсивного часу реакції.

Темп гри має дозволяти гравцю контролювати перебіг подій, розподіляючи увагу між дослідженням, діалогами й боями без примусового поспіху або таймінгових обмежень, які не обумовлені сюжетно.

Сформульовані вимоги забезпечать надійну основу для проектування архітектури програмного застосунку та побудови його ключових функціональних систем.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML–проєктування програмного забезпечення

Для створення чіткої та зрозумілої архітектури програмного застосунку Northpoint на першому етапі проєктування було вирішено застосувати UML–моделювання. Діаграма варіантів використання (Use Case Diagram) була обрана через її здатність наочно і структуровано представляти взаємодію головного актора – гравця, з різними функціями системи. Використання такої діаграми дозволить чітко визначити межі та вимоги до подальших компонентів архітектури гри.

3.1.1 Актори

Головним актором системи є користувач (гравець). Саме він здійснює взаємодію з системою та використовує всі функції і можливості застосунку.

3.1.2 Варіанти використання

Для побудови діаграми були визначені такі основні варіанти використання, засновані безпосередньо на постановці задачі роботи.

а) переміщення у відкритому світі:
   1) переміщення персонажа (звичайна хода, біг, тихий рух, спринт);
   2) використання стрибків і перекатів;
   3) управління камерою (перемикання між режимами першої та третьої особи).

б) взаємодія з об'єктами оточення:
   1) підняття предметів (зброя, броня, їжа, зілля, ключі, відмички, золото);
   2) взаємодія з контейнерами (сундуки, контейнери NPC);
   3) взаємодія з меблями (ліжка, стільці);
   4) відкриття та злам дверей.

в) керування інвентарем та екіпіруванням :
   1) відкриття та перегляд інвентаря;
   2) використання та екіпірування предметів;
   3) викидання предметів з інвентаря.

г) розвиток навичок та здібностей:

1) перегляд характеристик та рівнів навичок персонажа;

2) покращення навичок і здібностей.

д) орієнтування у світі:

1) перегляд карти світу (ортографічний і перспективний режими);

2) встановлення та видалення маркерів на карті;

3) використання компаса для орієнтування.

е) бойові дії:

1) виконання атак (рукопашних, ближнього бою, дальніх);

2) блокування атак.

ж) прихованість та крадіжки:

1) перемикання у режим прихованості;

2) виконання крадіжок (предмети, контейнери NPC).

з) взаємодія з NPC та квестами:

1) початок діалогів з NPC;

2) вибір реплік у діалогах;

3) виконання квестових завдань;

4) перегляд журналу квестів та важливих подій.

и) управління збереженнями та грою:

1) створення точок збереження;

2) завантаження гри з точок збереження;

3) використання меню паузи;

4) вихід з гри.

3.1.3 Побудова діаграми варіантів використання

На основі сформованої структури варіантів використання побудовано Use Case діаграму, яка наочно демонструє основні дії гравця та їх взаємозв'язок із функціональними можливостями програмного застосунку (див. рис. 3.1).

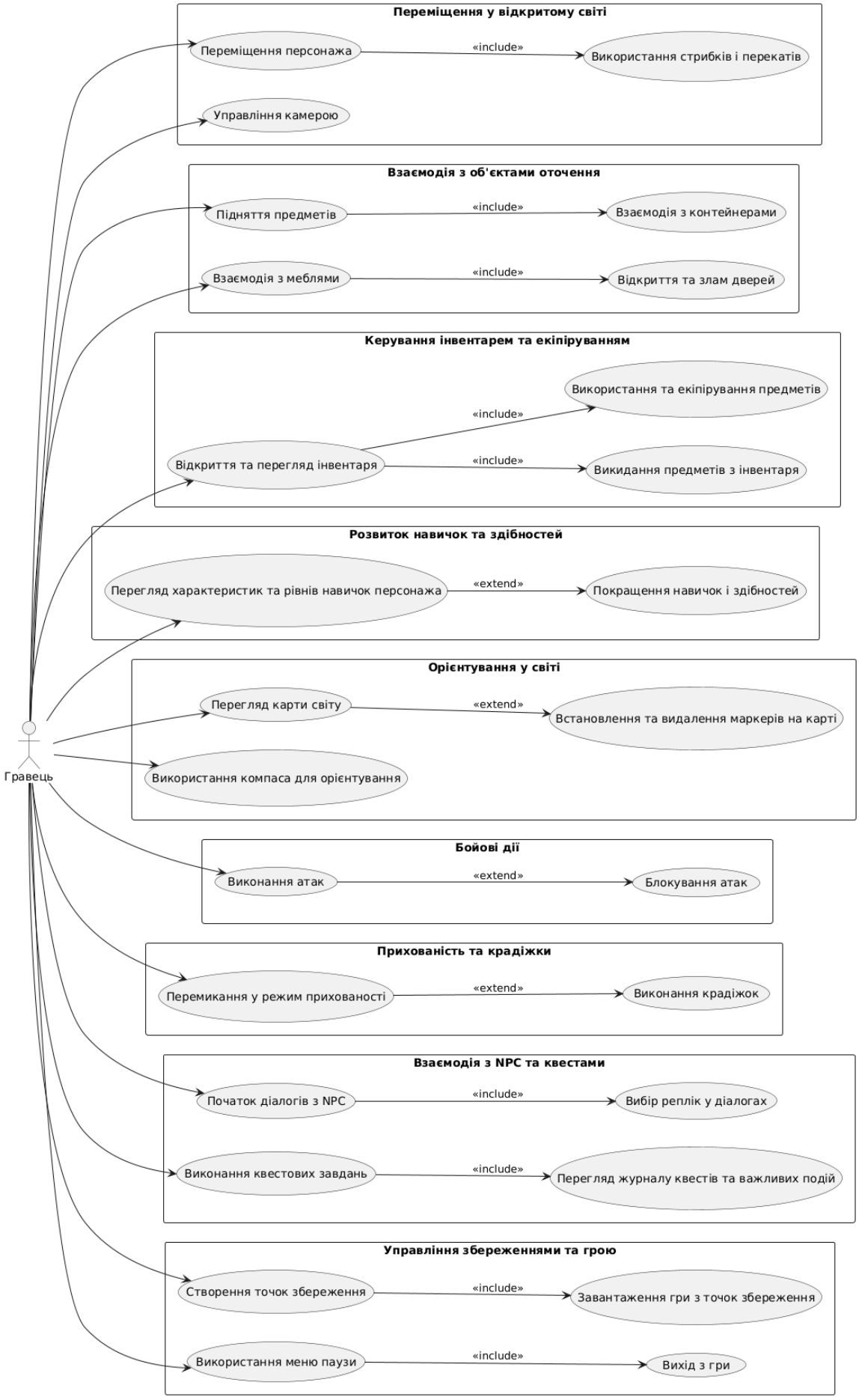


Рисунок 3.1 – Use Case діаграма застосунку (рисунок виконаний самостійно)

Представлена діаграма структурує функціональні можливості програмного застосунку, створюючи базу для подальшого розроблення його архітектури.

3.2 Проєктування архітектури програмного забезпечення

Проєктування архітектури є необхідною умовою для упорядкування функціональності ігрового застосунку в єдину цілісну систему. Враховуючи масштабність задуму – створення відкритого середовища з розвиненою інтерактивністю, бойовими механіками, розвитком персонажа, інвентарем, діалогами, квестами та іншими елементами – архітектура має забезпечити узгоджену роботу всіх складових, стабільність функціонування і можливість подальшого масштабування без необхідності глибоких змін у внутрішній структурі.

Функціональні можливості групуються у логічні підсистеми, кожна з яких охоплює певний напрям роботи застосунку. Орієнтиром для їхнього виділення слугують основні сценарії взаємодії користувача із системою. Такий підхід дозволяє будувати архітектуру послідовно та структуровано, забезпечуючи гнучкість, передбачуваність і незалежність від конкретних технологічних рішень.

У результаті були сформовані тринадцять ключових підсистем, що охоплюють усі аспекти ігрового процесу – від базової поведінки персонажа до керування сесією через користувацький інтерфейс.

3.2.1 Система персонажа (Character System)

Система персонажа визначає архітектуру базових ігрових істот, включаючи гравця та неігрових персонажів. Основою системи є клас Character, який об'єднує декілька основних компонентів: компонент Collision Box для фізичної взаємодії зі світом, Skeletal Mesh для візуального відображення моделі, та компонент Character Movement для обробки переміщення у просторі.

Керування візуальними станами персонажа здійснюється через окрему анімаційну підсистему Animation Blueprint, яка базується на інтерпретації поточних

характеристик руху. Архітектура передбачає автоматичний вибір анімацій відповідно до змінних стану руху без прямого втручання інших систем.

Система ресурсів реалізована як внутрішній підмодуль, що управляє трьома базовими ресурсами: здоров'ям, запасом сил і магічною енергією. Для кожного ресурсу визначено поточне і максимальне значення. Відновлення ресурсів інтегроване через механізм періодичного оновлення на базі таймерів, що дозволяє підтримувати сталість параметрів без активної участі користувача.

Управління змінами стану персонажа організовано через систему подій (Events), що сигналізують про впливи на ресурси чи стани. Реакція на критичні зміни, такі як втрата здоров'я до нуля, інтегрована у внутрішню логіку системи і ініціює перехід персонажа у стан смерті через запуск відповідних анімацій і блокування взаємодії з іншими компонентами.

Архітектура Character System орієнтована на ізоляцію базових властивостей персонажа від зовнішніх систем, забезпечуючи єдину точку відповідальності за фізичну присутність, базові стани і основні характеристики істоти у світі гри.

3.2.2 Система управління персонажем (Player Control System)

Система керування гравцем розширює архітектуру Character, додаючи функціонал, необхідний для прямого управління персонажем користувачем. Вона побудована як окремий логічний рівень, який приймає події Input Events та транслює їх у відповідні дії персонажа.

Основу Player Control System складає обробка традиційних інпутів клавіатури та миші. Натискання клавіш пересування (W, S, A, D) генерує виклики додавання рухового імпульсу Add Movement Input, що дозволяє персонажу переміщатися у просторі. При цьому швидкість руху назад і вбік пропорційно зменшена порівняно з рухом уперед для створення більш природного фізичного відчуття переміщення.

Введення від миші контролює поворот і нахил камери Add Camera Input, що дозволяє гравцю змінювати напрямок огляду у реальному часі. Архітектура системи включає дві камери: перша розташована на Spring Arm та забезпечує огляд

від третьої особи, тоді як друга закріплена безпосередньо у голові персонажа для режиму від першої особи. Відстань огляду регулюється через змінну, значення якої змінюється прокруткою колеса миші. У залежності від цього значення система активує одну з двох камер методом Set Active Camera, забезпечуючи плавний перехід між режимами огляду.

Таким чином, Player Control System забезпечує чіткий зв’язок між діями користувача та базовою логікою переміщення і огляду персонажа, доповнюючи Character System необхідним функціоналом для інтерактивного керування.

3.2.2 Система штучного інтелекту (AI System)

Система штучного інтелекту відповідає за логіку поведінки неігрових персонажів у світі гри. Її архітектура базується на стандартному підході Unreal Engine, у якому рух, сприйняття і реакції NPC реалізуються через набір взаємопов’язаних компонентів та служб.

Для забезпечення навігації у світі використовується об’єм навігаційної сітки Nav Mesh Bounds Volume, який розміщується на рівні так, щоб охопити всі прохідні зони карти. Усередині цього об’єму генерується навігаційна сітка Recast Nav Mesh, що визначає доступні шляхи пересування по поверхні. Вона складається з дискретних клітинок і використовується при побудові маршрутів руху NPC до заданих цілей.

Щоб уникнути надмірного навантаження на систему та не зберігати навігацію для всієї карти одночасно, до базового класу NPC Base Human додається компонент Navigation Invoker. Його завдання – ініціювати локальну генерацію навігаційної сітки лише навколо активного персонажа в межах визначеного радіуса, зазвичай до 5000 одиниць. Таким чином, навігація будується динамічно, лише у тих ділянках, де це необхідно, що дозволяє масштабувати світ без втрати продуктивності.

Кожен NPC управляється спеціальним контролером AI Controller, який відповідає за обробку внутрішньої логіки дій. До складу контролера входить компонент сприйняття AI Perception, що забезпечує виявлення об’єктів у

навколишньому середовищі. Основним сенсорним каналом є зір – Sight Sense, який відстежує присутність інших акторів у полі зору, визначаючи відстань, напрямок і наявність прямої видимості. Об’єкти, які можуть бути виявлені, мають компонент Stimuli Source, що реєструється системою сприйняття як потенційний сигнал.

Щоб NPC міг інтерпретувати побачених акторів, кожному персонажу надається числова змінна, що визначає його роль у світі – наприклад, гравець, мирний житель, стражник чи бандит. На основі значення цієї змінної контролер NPC ухвалює рішення щодо подальших дій у відповідь на сприйняття.

Ключовим елементом архітектури AI System є дерево поведінки Behavior Tree, яке визначає послідовність виконання дій NPC залежно від поточного стану. Behavior Tree прив’язаний до AI Controller і працює у зв’язці з даними з таблиці змінних Blackboard. Основна змінна state у Blackboard зберігає поточний режим поведінки NPC. Наприклад, значення 0 відповідає мирному стану, в якому NPC діє відповідно до своєї рутини: переміщується по заданому маршруту, виконує побутові задачі тощо.

У контролері NPC зареєстровано обробник події On Target Perception Updated для каналу зору Sight Sense. Якщо виявлено персонажа з ворожим статусом, контролер змінює значення змінної state у Blackboard на 2. Це автоматично активує іншу гілку дерева поведінки, у якій NPC переходить до бойового режиму, наприклад, починає атакувати ціль, слідувати за нею або переходити у стан тривоги.

Архітектура AI System побудована як розширення Character System для неігрових агентів. Вона забезпечує гнучку побудову поведінки, яка формується на основі зовнішніх умов, дозволяючи NPC реагувати на події навколишнього середовища без необхідності ручного управління.

3.2.4 Система взаємодії (Interaction System)

Система взаємодії забезпечує можливість інтерактивної роботи персонажа з об’єктами ігрового середовища. Вона інтегрується у контекст гравця і діє на основі постійного сканування простору перед камерою.

У процесі кожного кадру через подію Event Tick здійснюється лінійне трасування Line Trace від центра огляду камери вперед на визначену відстань. Якщо трасування виявляє акторів у своєму шляху, система перевіряє, чи реалізує знайдений об'єкт інтерфейс взаємодії Interact. У разі позитивної відповіді активується відповідний метод взаємодії, передаючи керування конкретному об'єкту. Завдяки такій архітектурі взаємодія з різними типами об'єктів здійснюється уніфікованим способом без потреби знати тип кожного актора заздалегідь.

Кожен персонаж має змінну стану взаємодії is interacting, яка відображає його залучення до процесу взаємодії. У разі активної взаємодії поведінка персонажа обмежується: звичайне керування рухом та іншими діями тимчасово блокується, передаючи контроль логіці об’єкта, з яким здійснюється взаємодія.

Такий підхід забезпечує гнучкість у побудові інтерактивних сценаріїв: від базової взаємодії з предметами до складних механізмів взаємодії з об'єктами середовища та іншими персонажами.

3.2.5 Система інвентаря (Inventory System)

Система інвентарю організовує структуру зберігання, відображення і взаємодії з ігровими предметами у світі гри. Основу системи становить уніфікована структура даних, що описує властивості кожного типу предмета. Вона включає такі поля, як назва, опис, тип предмета, вартість для торгівлі, вага, зображення на дисплеї, тривимірна модель та інші додаткові характеристики, специфічні для окремих видів об'єктів.

Крім загальної структури типу предмета, існує розширена структура Item Info, яка визначає конкретний екземпляр предмета на рівні. Вона містить ідентифікатор предмета, ідентифікатор власника, кількість одиниць та загальну інформацію про об'єкт.

Усі предмети на рівні базуються на класі Base Item. Кожен об'єкт має поле структури item info, яке містить усю необхідну інформацію для взаємодії. При піднятті предмета його дані додаються до масиву предметів персонажа, що утворює вміст інвентаря, після чого фізичний об'єкт у світі знищується.

Для управління інвентарем створено базовий графічний інтерфейс Base Interface, що складається зі слотів Base Inventory Slot. Від цього базового інтерфейсу успадковуються різні спеціалізовані віджети: інвентар персонажа, екіпірування, контейнери для зберігання предметів і торговельні інтерфейси. Окремий віджет Inventory Widget забезпечує гравцеві перегляд власного інвентаря та взаємодію з предметами через відповідні слоти.

Система екіпірування працює через окремі слоти Equipment Slot, куди переносяться активні предмети екіпірування, окремо від основного інвентаря.

Взаємодія з предметами реалізується шляхом обробки дії відповідно до вибраного слота у відкритому інтерфейсі. Тип взаємодії визначається поточним контекстом: використання, екіпірування, обмін або інші доступні операції.

Така архітектура забезпечує централізовану обробку даних предметів, спрощує додавання нових типів інвентарів і дозволяє гнучко керувати взаємодією гравця з об’єктами, не порушуючи логіки зберігання та відображення вмісту.

3.2.6 Система навігації (Map & Compass System)

Система навігації відповідає за орієнтацію гравця у світі гри, поєднуючи інтерактивну карту і компас на головному інтерфейсі. Вона реалізована як окрема підсистема, що не пов’язана безпосередньо з управлінням рухом, але використовує просторові дані для візуального супроводу дослідження світу.

Основним елементом карти є спеціальний інтерфейсний віджет, розташований поруч із вкладками інвентаря, екіпірування та навичок. Візуальна основа карти створюється через Map Capture Actor, розміщений високо над ігровим рівнем. Він містить компонент захоплення сцени Scene Capture Component 2D, який у реальному часі рендерить зону під собою у вигляді текстури, що передається до матеріалу інтерфейсного типу. Результатом є відображення мапи у вигляді зображення на відповідному елементі інтерфейсу.

Область, яку покриває карта, визначається двома контрольними точками Target Point, що задають мінімальні та максимальні межі. Їхні координати використовуються для обчислення масштабу та позиціонування даних на мапі.

У світі присутні спеціальні об’єкти, які успадковуються від базового класу Base Location Point. Під час ініціалізації – зокрема в момент серіалізації через Event OnSerialized – карта разово знаходить усі такі об’єкти, обчислює їх положення відносно власної системи координат і додає відповідні позначки безпосередньо на інтерфейс карти. Таким чином забезпечується візуалізація локацій, до яких гравець може орієнтуватися.

Окрім карти, головний екран містить компас – віджет Compass Widget, розміщений у верхній частині інтерфейсу. Це стрічковий елемент з нанесеними орієнтирами сторін світу, який орієнтується відповідно до обертання гравця по осі Yaw. Значення повороту персонажа змінює положення елементів компасу через обчислення зсуву координат Translation, дозволяючи в режимі реального часу бачити напрямок руху.

Компас також відслідковує об’єкти, успадковані від Base Location Point. Він обчислює кут між напрямком гравця та розташуванням кожної локації, а також її відстань. На основі цих даних віджет компаса відображає іконки локацій у правильному місці стрічки. Гравець також може встановлювати власні мітки, які додаються до компаса на тих самих умовах.

Коли гравець наближається до локації на достатню відстань, об’єкт локації генерує подію, яка повідомляє систему про факт відкриття. Якщо ця локація ще не була відвідана, на головному екрані відображається віджет повідомлення про відкриття разом із назвою відкритої місцевості. Відтоді локація вважається відомою і зберігається як постійно доступна позначка на карті та компасі.

Архітектура Navigation & Map System забезпечує повноцінну просторову орієнтацію у відкритому світі, синхронізуючи візуальні елементи з реальним положенням об'єктів на рівні, а також дозволяє динамічно оновлювати статус локацій у процесі дослідження.

3.2.7 Бойова система (Combat System)

Бойова система реалізує архітектуру обміну ушкодженнями між персонажами та керує процесами завдання й обробки ударів. Усі бойові взаємодії

ґрунтуються на механізмах виявлення зіткнень і обробки подій у визначених компонентах персонажа.

Система включає два основні типи урону: ближній та дальній. Ближній урон реалізується через використання компонентів удару, закріплених на відповідних частинах тіла або зброї (наприклад, кулаки або меч). Ці компоненти налаштовані на перетин з усіма фізичними тілами типу Physics Body та генерують події перетину Overlap Events під час активної фази анімації атаки. У процесі таких подій перевіряється наявність виявлених акторів, що реалізують інтерфейс бойової взаємодії, після чого викликається обробка урону.

Дальній урон реалізується через запуск проєктильного об’єкта (стріли), що рухається в напрямку наведення. При зіткненні стріли з іншими об’єктами система реєструє події зіткнення Hit Events та, у разі виявлення відповідного акту, активує подію Weapon Hit Event, яка ініціює стандартну процедуру обробки урону.

Отримання урону відбувається на стороні актора, який приймає атаку. Після зменшення значення здоров’я активуються супровідні візуальні та анімаційні ефекти. У точці влучання створюється ефект Particle Effect, що імітує результат удару, а при зіткненні з поверхнею – додатково розміщується відповідний слід у вигляді відбитка Decal. Заключною фазою обробки атаки є визначення та програвання анімації реакції на удар. Вибір конкретної анімації залежить від типу використаної зброї, місця влучання та напрямку, з якого був нанесений урон.

Архітектура Combat System забезпечує чітке розділення обов’язків: механізми нанесення урону ініціюються зброєю або діями персонажа, тоді як механізми прийому, візуалізації та реакції обробляються на стороні об’єкта, що зазнає ушкодження. Така структура дозволяє легко масштабувати бойову систему, додаючи нові типи зброї чи урону без зміни базових принципів взаємодії.

3.2.8 Система прихованості (Stealth System)

Система прихованості реалізує альтернативний до відкритого бою підхід до досягнення цілей у грі, зосереджуючись на непомітності, акуратності й тактичності. Вона є логічним розширенням трьох базових систем – Character

System, Player Control System та AI System – і включає спеціалізовану поведінку для гравця і неігрових персонажів.

В основі Stealth System лежить розширення сенсорної моделі штучного інтелекту. До вже наявного каналу зору Sight Sense додаються два додаткові сенсори: слух Hearing Sense та дотик Touch Sense. Сенсор слуху Hearing Sense дозволяє NPC виявляти джерела шуму у визначеному радіусі, з урахуванням гучності сигналу та відстані до нього. Сенсор дотику Touch Sense виявляє безпосередній фізичний контакт із гравцем або об’єктами, які знаходяться в зоні прямого торкання, навіть якщо зорове виявлення відсутнє.

Під час виконання більшості дій персонажі автоматично генерують шум через подію Make Noise. Наприклад, у точках анімації кроків, де ступні контактують із поверхнею, анімаційні сповіщувачі Anim Notifier активують подію створення шуму. Інші приклади – мах зброєю, натягування тятиви лука, крик на допомогу. Усі такі дії реєструються AI Controller NPC, якщо значення гучності перевищує поріг виявлення у межах досяжної зони.

Система пересування підтримує режим скрадання Crouch, який змінює не лише швидкість, а й інтенсивність шумів. Персонажі у такому режимі створюють значно менше шуму, що ускладнює їх виявлення через слуховий сенсор. Поведінка NPC у відповідь на звук залежить від їхньої ролі у світі. Наприклад, стражник Guard, почувши підозрілий шум, переходить у стан пошуку, змінюючи значення змінної state у Blackboard на 1, і прямує до джерела сигналу для перевірки.

До складових Stealth System також входять крадіжки та злом. Якщо гравець перебуває у режимі скрадання та наближається до NPC, який його не бачить і не реагує, з’являється опція взаємодії – крадіжка. При активації відкривається інтерфейс доступу до інвентаря NPC (через віджети Inventory та Container у режимі крадіжки), що дозволяє гравцеві обрати предмети для потенційного викрадення. Успіх дії визначається з урахуванням позиції, стану об’єкта та внутрішньої ймовірності виявлення.

Схожа логіка застосовується до процесу злому. Деякі двері можуть бути зачинені й вимагати ключа для відкриття. Якщо ключа немає, гравець може

використати відмичку – спеціальний предмет в інвентарі. За умови активації відповідної дії та наявності відмички система виконує перевірку успішності злому на основі заздалегідь заданої ймовірності. У разі успіху двері відчиняються, і гравець отримує доступ до закритої зони.

Архітектура Stealth System побудована як надбудова до базових систем і забезпечує паралельну взаємодію з механіками сприйняття, пересування, інвентаря та інтерфейсів. Завдяки цьому приховані дії інтегруються в загальну логіку гри, не конфліктуючи з відкритими бойовими сценаріями, а натомість доповнюючи і збагачуючи їх.

3.2.9 Система діалогів (Dialogue System)

Система діалогів забезпечує організацію інтерактивного спілкування між гравцем та неігровими персонажами у світі гри. Вона діє як розширення стандартної поведінки NPC та включає окрему логіку ініціації розмови, управління її перебігом та обробки вибору гравця.

Діалог починається, коли гравець підходить до NPC, який перебуває у стані готовності до розмови, тобто живий і знаходиться у спокійному режимі поведінки. Після наближення на достатню дистанцію NPC автоматично ініціює початок діалогу: на екран виводиться інтерфейсний елемент Dialogue Widget, а камера Camera Component гравця передає фокус контролю до діалогового режиму, що забезпечує фіксацію уваги на співрозмовнику.

Управління логікою діалогу побудоване на основі окремого Behavior Tree, який створюється для кожного NPC індивідуально. Структура діалогового дерева складається з інстансів базових задач Task – Speak та Reply. Задача Speak використовується для відтворення реплік NPC, тоді як задача Reply обробляє вибір гравця серед можливих варіантів відповіді.

Основні змінні діалогу зберігаються у Blackboard, прив’язаному до Behavior Tree відповідного NPC. Серед них особливу роль відіграє індекс відповіді гравця, який визначає подальший перебіг розмови залежно від зробленого вибору.

Вибір гравця відбувається через інтерфейс Dialogue Widget, що виводить можливі варіанти реплік. При натисканні на відповідну кнопку викликається Event Dispatcher, який врешті–решт передає вибраний індекс відповіді у Blackboard. Повідомлення про зміну стану ініціює продовження Behavior Tree за відповідною гілкою, що забезпечує реактивний перебіг діалогу залежно від рішень гравця.

Такий підхід дозволяє будувати розгалужені сценарії діалогів з NPC, враховуючи поточний та попередній контекст подій.

3.2.10 Система квестів (Quest System)

Система квестів забезпечує фіксацію подій, відстеження завдань і зручний доступ до історії прогресу гравця в квестовому журналі. Її архітектура поєднує зберігання структурованих записів та інтерфейсний механізм їх відображення у вигляді інтерактивного журналу.

Основною структурною одиницею системи є Journal Page – запис, що відповідає окремому завданню, події або етапу проходження гри. Кожна сторінка містить унікальний ідентифікатор ID для забезпечення подальшого доступу та перевірки, дату створення запису, заголовок події та текстовий опис. Це дозволяє зберігати не лише активні завдання, а й минулі події, що мають сюжетне або функціональне значення.

Усі записи зберігаються у масиві структур Journal Page, який є частиною даних гравця. Відображення цього вмісту реалізується через спеціальний інтерфейс Journal Widget. Інтерфейс побудований у вигляді візуального журналу з анімаційною розгорткою, що імітує відкриття сторінок книги.

При активації відповідної події, зокрема натискання кнопки відкриття журналу, віджет Journal Widget розгортається за допомогою віджет–анімації,

Після відкриття гравцеві відображаються дві останні сторінки журналу – по одній на кожному боці. Перемикання між сторінками реалізоване через обробку взаємодії користувача: натискання лівої або правої кнопки миші в межах елемента або використання ползунка Slider для швидкого переходу. Перемикання ініціює зміну відповідного індексу у масиві записів і оновлення вмісту Journal Widget.

Архітектура Quest System дозволяє створювати не лише функціональну систему відстеження завдань, а й стилізований ігровий щоденник, який зберігає логіку світу, подій і дій гравця, підтримуючи цілісність ігрового досвіду.

3.2.11 Система навичок (Skill System)

Система навичок і прогресії забезпечує внутрішню узгодженість усіх ігрових механік, встановлюючи залежність їхньої роботи від рівня розвитку персонажа. З метою побудови єдиної структури прогресії було введено загальний рівень персонажа та сім основних навичок:

- витривалість (Endurance);
- ближній бій (Melee);
- дальній бій (Ranged);
- броня (Melee);
- прихованість (Sneak);
- спритність (Agility);
- красномовство (Speech).

Ці навички відображають ключові напрями активності персонажа та пов’язані з відповідними модулями гри. Окремі навички можуть впливати одночасно на декілька систем або навпаки концентруватися на одному конкретному аспекті.

Рівень розвитку кожної навички визначається в діапазоні від 0 до 100. Прогресія навичок здійснюється через накопичення спеціальних очок, що нараховуються за виконання відповідних ігрових дій. Для розрахунку необхідного рівня для підвищення навички застосовується формула 3.1:

$$\text{level} = \left(1 + \frac{level - 1}{5}\right) pts \tag{3.1}$$

де $level$ – відповідає поточному і наступному рівню розвитку навички;

pts – позначає необхідну кількість очків навички.

Повна прокачка навички з 0 до 100 вимагає рівно 1090 очок навичок, здобутих шляхом виконання дій, пов’язаних із даним напрямом розвитку.

Відповідно загальний рівень персонажа визначається на основі накопичення рівнів навичок, обчислюваних за формулою 3.2:

$$\text{level} = \left(1 + \frac{(level - 1) - 1}{1.8}\right) pts \tag{3.2}$$

де $level$ – відповідає поточному і наступному рівню розвитку гравця;

pts – позначає необхідну кількість очків навички.

Для досягнення максимального рівня 50 потрібно приблизно 702,33 очка рівня або рівня навичок. Додатково передбачено нарахування 0,33 очка рівня за кожен випадок повного розвитку навички до 100, що забезпечує узгодження прогресії всіх 7 навичок із загальним рівнем персонажа.

Розвиток навичок здійснюється через конкретні дії, що відповідають кожній сфері активності, а ефекти від прокачки напряму впливають на характеристики персонажа та змінюють його взаємодію зі світом гри. Всі дії прогресії та залежності навичок наведені Додатку А.

Окрім базових навичок, система прогресії включає механіку здібностей, які відкриваються при досягненні певних рівнів розвитку. Для кожної навички передбачено набір пов’язаних із нею здібностей, що формують ієрархічну або розгалужену структуру. Здібності відкриваються за умови досягнення необхідного рівня відповідної навички та доступності очок розвитку. Їх впровадження реалізоване через систему перевірок у ключових модулях гри методом Has Perk класу персонажа: при наявності відповідного елемента масиву рядків назв Perks виконується спеціалізований гілковий сценарій або активується альтернатива стандартної логіки. Це дозволяє точно масштабувати ігровий досвід відповідно до виборів гравця. Повний перелік здібностей, їх назви, умови відкриття та опис наведено у Додатку Б.

Таким чином, система навичок і прогресії не лише фіксує розвиток персонажа, а й визначає характер роботи інших модулів гри, впливаючи на логіку бою, прихованості, діалогів та інших взаємодій. Вона виступає архітектурним зв’язком між окремими підсистемами, забезпечуючи узгоджене масштабування всіх механік відповідно до досвіду гравця.

3.2.12 Система збереження (Save System)

Система збереження відповідає за збирання, зберігання та відновлення станів усіх інших підсистем гри. Її основне призначення – забезпечити збереження поточного стану ігрового світу, об'єктів, персонажа, прогресії та взаємодій, щоб після перезапуску гри ці дані були відтворені з урахуванням усіх попередніх змін.

Архітектура системи збереження базується на взаємодії трьох ключових компонентів Unreal Engine: Game Mode, Game Instance та Save Game. Кожен із них виконує свою окрему роль. Game Mode відповідає за логіку ігрового режиму та ініціацію збереження. Game Instance є глобальним об’єктом, що існує протягом усієї сесії гри і дозволяє зберігати спільні дані незалежно від рівнів. Save Game – це серіалізований об’єкт, який містить усі потрібні для збереження структури та значення.

Процес збереження починається з виклику методу Save Game у Game Mode. Під час виконання цього методу створюється новий або використовується наявний об’єкт класу Save Game – спеціальний контейнер, у якому зберігається вся необхідна інформація у вигляді окремих змінних, структур та масивів структур, що у подальшому можуть бути збережені на пристрої користувача.

Кожен об’єкт у грі, який має змінний стан і потребує збереження, повинен мати унікальний ідентифікатор ID. У процесі збереження Game Mode отримує список усіх таких об'єктів через метод Get All Actors of Class. Для кожного з них фіксуються ключові змінні, що підлягають збереженню (позиція, стан, зв’язки тощо), після чого дані об’єднуються в масив структур, у якому кожен запис містить ID та відповідні значення. Ці масиви записуються як поля об’єкта Save Game, після чого сам об’єкт серіалізується на диск методом Async Save Game To Slot.

Оскільки Game Instance зберігається протягом усієї сесії гри та не скидається під час зміни рівнів, він використовується як буфер для збережених даних. Після завантаження Save Game із диску його вміст копіюється до Game Instance, звідки кожен об’єкт може отримати свої значення незалежно від рівня або сцени.

Під час завантаження гри, у методі Begin Play відбувається виклик функції Load Game. Через метод Async Load Game From Slot завантажується збереження із зазначеного слоту. Дані десеріалізуються у відповідний об’єкт класу Save Game і записуються до Game Instance для тимчасового зберігання в межах поточної сесії. На цьому етапі кожен окремий актор, який бере участь у системі збереження, під час власного Begin Play звертається до Game Instance, шукає в масиві записів значення з відповідним ID і оновлює свої змінні відповідно до збереженого стану.

Такий підхід дозволяє організувати централізовану та незалежну від рівня систему збереження, яка охоплює всі ключові компоненти ігрового процесу та гарантує їх точне відновлення при наступному запуску гри.

3.2.13 Система меню (Menu System)

Система меню відповідає за керування поточною сесією гри: зупинку, відновлення, збереження, завантаження або завершення, та початок нової. Вона не впливає безпосередньо на внутрішню логіку інших підсистем, але забезпечує централізований доступ до них через інтерфейс, формуючи загальні переходи між ключовими станами виконання.

Головне меню реалізується як окремий віджет інтерфейсу, що викликається натисканням клавіші Esc незалежно від активного стану гри. Після виклику меню гра призупиняється за допомогою функції Set Game Paused, а управління передається інтерфейсним елементам. Щоб забезпечити реакцію на події навіть у режимі паузи, усі обробники натискань у межах віджета налаштовані з параметром Execute Game Paused.

Перша кнопка виконує дію зняття гри з паузи, аналогічну повторному натисканню клавіші Esc. Після активації цієї кнопки гра відновлюється саме з того

моменту, на якому гравець її призупинив, відновлюючи всі події, позиції об'єктів та стани персонажів, які були активними до паузи.

Друга кнопка ініціює запуск повністю нової гри. Вона завантажує дефолтний початковий рівень гри в початковому стані без жодних додаткових параметрів. Це означає, що всі прогреси попередніх ігрових сесій скидаються, а гравець розпочинає гру з початковими налаштуваннями та станом світу.

Третя кнопка виконує збереження поточного стану гри у визначений останній слот збереження. При її натисканні інтерфейс звертається до ігрового інстансу класу BP_ThirdPersonGameMode та активує його метод SaveGame. Після завершення процесу збереження активується подія OnGameSaved, яку прослуховує віджет меню паузи. Отримавши цю подію, віджет відписується від неї та викликає логіку продовження гри — таку саму, як при натисканні кнопки «Продовжити». Перед цим гравець додатково інформується про успішне збереження.

Четверта кнопка призначена для завантаження останнього збереженого стану гри. Вона використовує вбудовану функцію Unreal Engine OpenLevel, приймаючи назву поточного рівня як параметр. Таким чином, гра перезапускається, і поточний рівень завантажується з останньої збереженої точки прогресу.

Остання, п'ята кнопка реалізує повний вихід із гри. Вона викликає стандартну функцію Unreal Engine QuitGame, яка закриває вікно застосунку, завершує ігровий процес і повертає гравця до операційної системи або попереднього вікна запуску гри. Таким чином, після натискання цієї кнопки відбувається повне звільнення ресурсів гри та завершення її виконання.

Menu System виконує функцію зовнішнього керування загальним перебігом гри, надаючи користувачеві інструменти для керування станами сесії без необхідності прямої взаємодії з іншими підсистемами. Вона забезпечує стабільну і контрольовану роботу застосунку в цілому.

3.2.14 Побудова діаграми компонентів

Для візуалізації загальної архітектури на рис. 3.2 подано діаграму компонентів, що ілюструє взаємозв’язки між основними підсистемами.

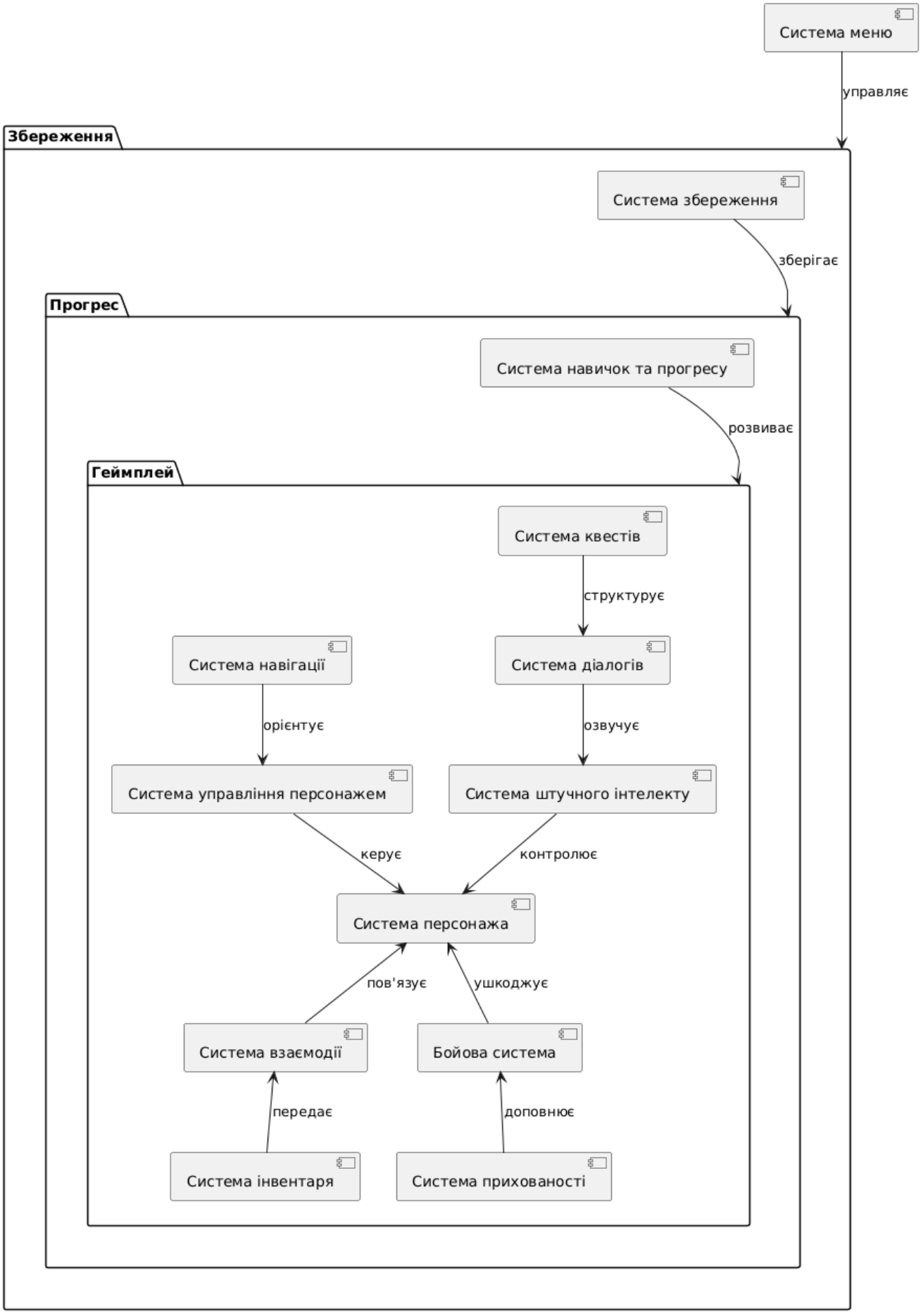


Рисунок 3.2 – Діаграма компонентів застосунку (рисунок виконаний самостійно)

Представлена діаграма компонентів відображає внутрішню архітектуру застосунку, окреслюючи основні підсистеми та їхні взаємозв’язки. Побудована структура забезпечує основу для подальшого проєктування механізмів зберігання та обробки даних у грі.

3.3 Проєктування структури зберігання даних

3.3.1 Проєктування загальної структури збереження ігрового середовища

Збереження стану ігрового світу є стандартним функціональним механізмом рушія Unreal Engine, який реалізується через серіалізацію даних у спеціальні об’єкти класу Save Game. Всі такі об’єкти зберігаються у директорії гри за адресою Saved/SaveGames, а результуючі файли мають розширення .sav. Після створення збереження рушій автоматично формує відповідний файл у вказаній директорії з унікальним іменем слоту.

Клас Save Game виступає універсальним контейнером, що призначений для зберігання будь–яких змінних користувацького типу, в тому числі базових значень, структур і масивів. У межах одного об’єкта допускається серіалізація як простих типів (наприклад, Float, Integer, Boolean), так і складених (наприклад, Transform, Vector, Rotator), а також довільно складених користувачем структур, що містять вкладені масиви або інші структури.

З боку архітектури проєкту передбачено власний нащадок базового класу Save Game, який містить набір полів:

- Date – поточна дата ігрового часу;
- PlayerSaveData – структура збережених даних персонажа;
- ItemsSaveDataArray – масив предметів у світі;
- HumansSaveDataArray – масив неігрових персонажів;
- ChestItemsSaveDataArray – масив вмісту контейнерів;
- DoorsSaveDataArray – масив станів дверей;
- LocationsSaveDataArray – масив відкритих локацій;
- PagesSaveDataArray – масив записів ігрового журналу.

Таке рішення дозволяє зберігати прогрес гравця в повному обсязі: від положення персонажа та його характеристик – до глобального стану світу, стану дверей, контейнерів, виконаних квестів і журналу подій.

Для демонстрації підходу до організації даних розглянемо три окремі типи: одиничну змінну системного типу (Date Time), користувацьку структуру (Player Save Data) та масив структур однотипного формату (Doors Save Data Array).

Перше поле Date має тип Date Time, який є вбудованим типом Unreal Engine та містить повне календарно–часове представлення події. Структура цього типу включає такі поля: рік, місяць, день, година, хвилина, секунда, мілісекунда, а також часовий пояс до них. У межах реалізованої архітектури використовується лише календарна дата та час, хвилина та секунда доби, що слугує базовим механізмом для обчислення внутрішньоігрового часу.

Змінна Date оновлюється періодично відповідно до логіки глобального таймера, який забезпечує симуляцію добового циклу (день/ніч) у реальному часі. Таким чином, кожне збереження зафіксовує момент збереження гри за поточним станом ігрової хронології, що може використовуватись для сценарних та інтерфейсних рішень.

Другим прикладом виступає структура PlayerSaveData, яка реалізується як окрема користувацька структура, що містить усі ключові параметри, необхідні для відновлення стану гравця, включно з базовими характеристиками, навичками, позицією у просторі, станом камери, інвентарем, екіпіруванням, параметрами журналу тощо.

Повний перелік полів цієї структури із зазначенням типу даних та коротким описом функціонального призначення наведено у додатку В.

Для збереження стану інтерактивних об’єктів світу, зокрема дверей, використовується масив структур Doors Save Data Array, кожен елемент якого відповідає одному об’єкту типу дверей у світі. Структура елемента має тип Door Save Data та містить два поля: ID типу Integer та Is Opened типу Boolean. ID формується під час розміщення об’єкта на рівні та є унікальним серед інших

екземплярів даного класу. Поле IsOpened дозволяє точно зберігати статус відкритості двері та доступності відповідної зони (будівлі, кімнати, тунелю тощо).

Така система дозволяє зберігати великий обсяг однотипної інформації з мінімальними витратами пам’яті й забезпечує відновлення всіх змін у світі гри після завантаження збереження.

3.3.2 Проєктування структури бази даних предметів

У межах відкритого світу передбачається значна кількість предметів, з якими гравець зможе взаємодіяти під час дослідження локацій, боїв, торгівлі та керування інвентарем. Незважаючи на різноманітність джерел і форм використання, усі предмети мають спільну змістову структуру, що охоплює базові характеристики – назву, опис, тип, вартість, вагу, зовнішнє представлення, бойові або захисні властивості та інші параметри.

З огляду на це виникає потреба у створенні централізованої бази даних предметів, яка зберігає незмінні значення для кожного типу об’єкта та може бути використана в різних контекстах без дублювання даних. Такий підхід дозволяє скоротити обсяг пам’яті, підвищити узгодженість даних та спростити підтримку і розширення гри.

Для зберігання постійних характеристик предметів у межах програмного застосунку спроєктовано окрему табличну структуру, реалізовану у вигляді бази даних. Як основу обрано механізм Data Table, підтримуваний рушієм Unreal Engine, який дозволяє зберігати набір записів, організованих за єдиною структурою полів. У якості шаблону використовується користувацька структура Item Info, яка описує повний набір атрибутів, необхідних для ідентифікації, класифікації, візуалізації та функціонального використання предметів у системах інвентаря, торгівлі, екіпірування та контейнерів.

Структура Item Info визначає перелік усіх властивостей, які описують кожен предмет, і включає як спільні, так і специфічні для типу об'єкта параметри:

– Name – назва предмета;
– Description – текстовий опис;
– Type – тип предмета;
– Cost – вартість у золотих одиницях;
– Weight – маса предмета;
– Stackable – множинність зберігання;
– Icon – текстура іконки для відображення в інтерфейсі;
– Mesh – 3д-модель для відображення у грі;
– Class – клас предмета, необхідний для його використання;
– WeaponInfo – вкладена структура для предметів типу «зброя», яка включає параметри урону та тип атаки;
– ArmorInfo – вкладена структура для предметів типу «броня», яка включає значення захисту та тип броні;
– SkeletalMesh – ассет Skeletal Mesh для спавну у якості екіпіювання;
– ClassToSpawn – клас предмета для спавну у якості екіпіювання;
– RelatedID – ідентифікатор пов’язаного предмета.

У підсумку отримується централізована таблиця, яка містить усю постійну інформацію про всі предмети в грі та є універсальним джерелом даних для систем інвентаря, екіпірування, контейнерів і торгівлі (див. рис. 3.3).

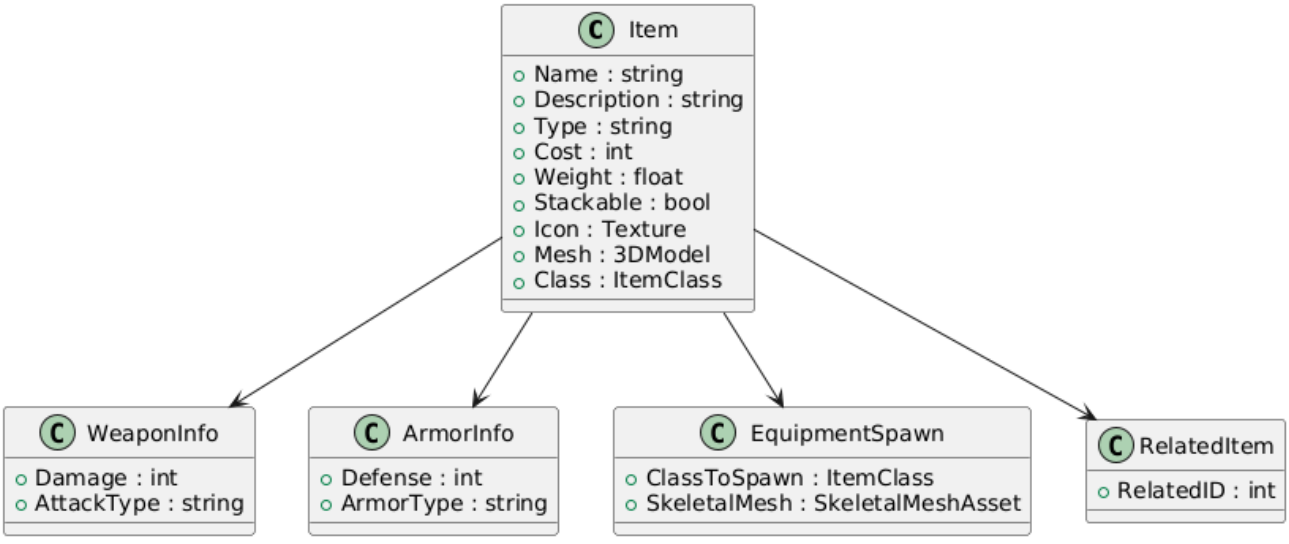


Рисунок 3.3 – Діаграма структури даних предметів застосунку (рисунок виконаний самостійно)

Представлені структури збереження окреслюють загальну архітектуру зберігання ігрових даних, включаючи як динамічні змінні прогресу, так і статичні характеристики об’єктів. Побудована модель забезпечує основу для повноцінного збереження стану світу та підтримки узгоджених даних під час взаємодії з предметами.

3.4 Створення UI / UX дизайну системи

Графічний інтерфейс користувача є одним з основних каналів взаємодії між гравцем та програмним застосунком. Якість реалізації UI / UX напряму впливає на зручність керування, ефективність орієнтування у світі, швидкість прийняття рішень та загальне занурення у процес гри. В системах відкритого світу, які передбачають великий обсяг взаємодій, управління інвентарем, навігацію, моніторинг характеристик і використання предметів, інтерфейс повинен бути не лише функціональним, а й структурно вивіреним, узгодженим і візуально цілісним.

Загальне проєктування інтерфейсу базується на дотриманні принципів ієрархії, модульності та візуальної узгодженості: усі функціональні області розміщуються відповідно до їхньої важливості та частоти використання, кожен компонент оформлюється як окремий віджет із власною логікою, що спрощує повторне використання й обслуговування, стилістика інтерфейсу витримана в єдиній кольоровій гамі з нейтральним фоном і помірним контрастом.

Серед усіх елементів інтерфейсу, які необхідно спроектувати, насамперед розглянемо головний інтерфейс, що поєднує базові функціональні панелі та забезпечує безперервну взаємодію гравця з ключовими підсистемами гри. Він складається з трьох віджетів: панелі гравця, інвентаря та карти.

3.4.1 Створення віджета інвентаря

Віджет інвентаря є основним елементом взаємодії з предметами у грі та розміщується в нижній лівій чверті екрана. У цій області гравець отримує доступ до списку наявних об’єктів, переглядає їхні властивості, кількість, іконки, а також має можливість використовувати або викидати вибраний предмет.

Реалізація інтерфейсу починається зі створення окремого віджета, який успадковується від базового класу Base Interface. Цей клас містить спільні для всіх інтерфейсних модулів параметри, зокрема змінну назви, масив предметів для відображення, а також набір функцій для додавання до ієрархії, керування видимістю та обробки підказок.

Основою розмітки інвентарного віджета є компонент Canvas Panel, що дозволяє позиціювати вкладені елементи у відносних координатах незалежно від розміру екрана. Така структура забезпечує зручне розміщення як декоративних, так і функціональних елементів інтерфейсу.

Усередині Canvas Panel розміщується внутрішня область Main Panel, яка обмежує активну зону віджета. Вона включає затінений фон, реалізований за допомогою компонента Border з напівпрозорим чорним фоном, що забезпечує візуальний контраст без повного затемнення сцени. Передбачено відступи зліва та знизу, що формують візуальне поле панелі та створюють відокремлення її меж від країв екрана. Окремо додано технічний відступ у 124 пікселі знизу для резервування простору під індикатори ресурсів персонажа.

У верхній частині розташовується ще один Border з текстовим блоком, який відображає назву інтерфейсу разом із поточним значенням заповненості. Текст оформлюється з урахуванням кольорового маркування, що може відображати ступінь законності або підозрілої активності взаємодії. У нижній частині панелі розміщується текстовий блок, вирівняний по лівому краю, у якому наводиться підказка доступних дій для вибраного предмета.

Основною функціональною частиною інтерфейсу є масив слотів предметів, що додаються до Canvas Panel у вигляді окремих елементів типу Inventory Slot. Кожен слот є нащадком класу W_BaseSlot — універсального підвіджета розміром 60×60 пікселів. Усередині нього реалізовано базову структуру на основі Overlay, що включає кнопку взаємодії як фоновий елемент, зображення іконки предмета, яке відображається поверх, а також текстовий блок у правому верхньому куті, що показує кількість екземплярів обраного предмета у стеку.

Загальна кількість слотів становить 60, і вони розміщуються один за одним із дотриманням постійних відступів, утворюючи рівномірну сітку (див. рис. 3.4).

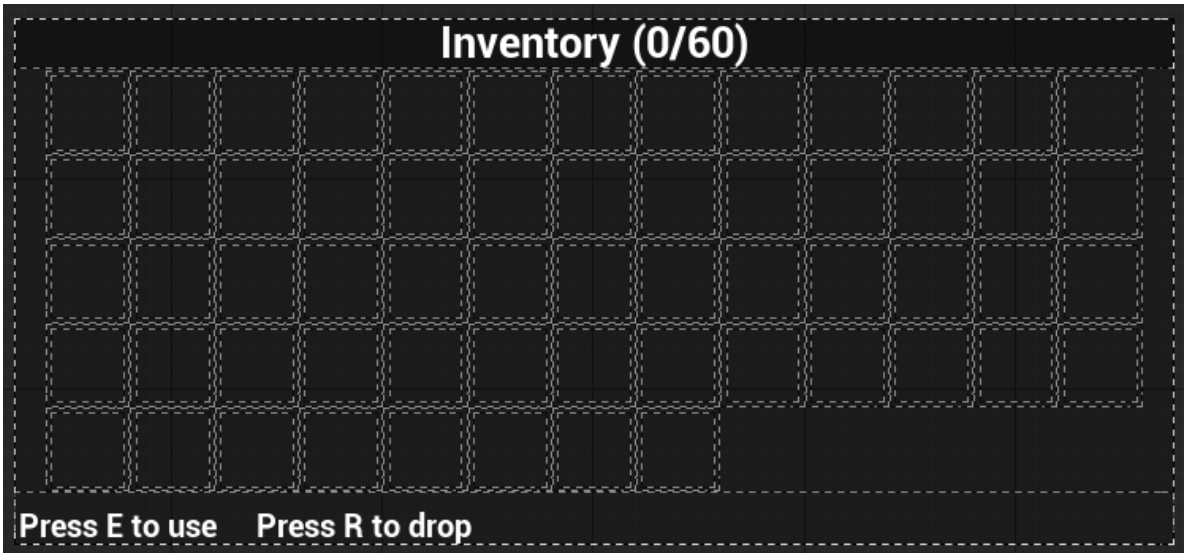


Рисунок 3.4 – Структура віджета інвентаря (рисунок виконаний самостійно)

Таким чином, створений віджет інвентаря поєднує універсальність із відповідністю загальним принципам інтерфейсного дизайну, що дозволяє використовувати його в різних режимах без зміни внутрішньої структури.

3.4.2 Створення віджета гравця

Віджет гравця розміщується безпосередньо над інвентарем у головному інтерфейсі та, як і решта модулів, успадковується від базового класу BaseInterface. Він має відступ у 40 клітин зліва та зверху, додатково передбачено 124 клітини зверху для відображення компаса. У верхній частині розташовано заголовок, у нижній — область для підказок.

Функціонально віджет розділений на дві основні частини — блок екіпірування та блок характеристик. Зліва знаходиться панель екіпірування з ортографічним зображенням персонажа гравця, яке виводиться на основі матеріалу від Render Target 2D камери попереду персонажа. З обох боків цього зображення розміщено слоти екіпірованих предметів: з одного боку — елементи броні (шолом, нагрудник, рукавиці, поножі, взуття), з іншого — спорядження (основна зброя, додаткова зброя, амулет, перший перстень, другий перстень). Над зображенням

персонажа відображається горизонтальний індикатор навантаження — смуга жовтого кольору, яка демонструє співвідношення поточної ваги інвентаря до максимально допустимої.

Справа від блоку екіпірування розміщується панель характеристик, яка поділяється на два підблоки: зліва відображаються ресурси, основні параметри та загальна статистика, а справа — розвиток навичок персонажа. У верхній частині лівого блоку розташовані показники здоров'я, витривалості та магічної енергії, кожен з яких супроводжується назвою, смугою заповнення та кнопкою підвищення, що дозволяє збільшувати максимальне значення при отриманні нового рівня. Нижче виводяться рівень персонажа, кількість доступних очок здібностей, ігровий стиль, сила атаки, захист, репутація та сума штрафів.

Правий блок містить десять навичок: витривалість, володіння зброєю ближнього бою, стрільбу, носіння броні, прихованість, спритність, красномовство, магію руйнування, магію відновлення та магію виклику. Для кожної навички виводиться назва та числове значення поточного рівня (див. рис. 3.5)

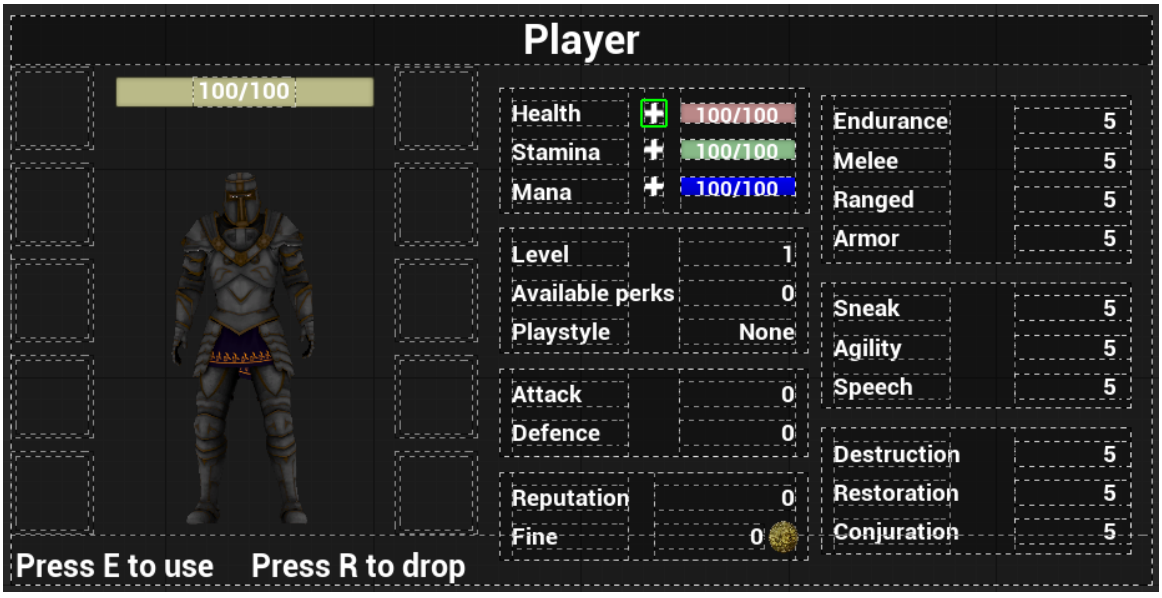


Рисунок 3.5 – Структура віджета гравця (рисунок виконаний самостійно)

Структура віджета повністю охоплює всі функції, пов'язані з відображенням характеристик і екіпірування, та забезпечує максимальну інформативність у межах відведеного простору.

3.4.3 Створення віджета мапи

Віджет мапи має всі характерні ознаки інтерфейсних елементів системи та інтегрується у загальну структуру головного інтерфейсу. Він займає всю праву частину екрана, має фіксовану розмірність 880×880 пікселів і включає зарезервовану верхню область під компас. Така площа відповідає сумарному розміру двох інших віджетів — інвентаря та гравця — і забезпечує симетричне компонування інтерфейсу.

Основна панель MainPanel містить компонент Overlay, що дозволяє накладати дочірні елементи у вигляді окремих шарів. На найнижчому рівні розташований Border, який додає напівпрозоре затемнення в зонах, де карта не покриває фон. Наступним шаром виступає зображення карти — текстура, до якої прив’язано UI-матеріал типу Render Target. Камера, закріплена високо над ігровим рівнем, транслює зображення в реальному часі через Scene Capture, що дозволяє формувати інтерактивне представлення поточного положення гравця у світі.

Вище у ієрархії розміщуються окремі елементи типу W_MapPoint — спеціалізовані віджети, які відображають конкретні позначки: гравця, ціль, досліджені локації, маркери. Вони мають прозоре тло, накладаються на карту поверх зображення та динамічно реагують на зміну позиції камери, зміщуючись у відповідності до її руху.

Структуру віджета наведено на рисунку (див. рис. 3.6).

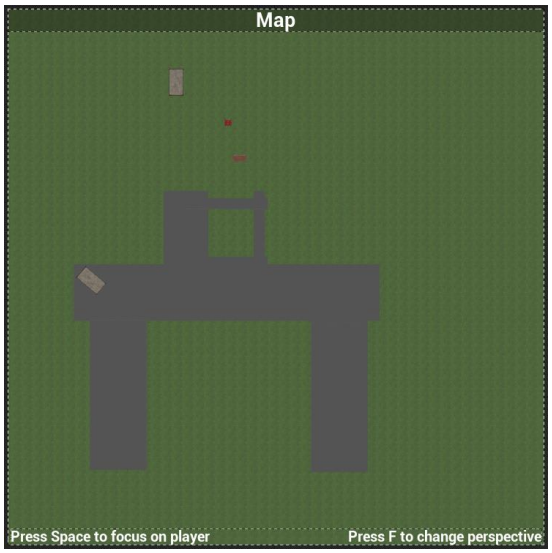


Рисунок 3.6 – Структура віджета мапи (рисунок виконаний самостійно)

Проєктування інтерфейсної системи охоплює всі ключові елементи взаємодії користувача з ігровим середовищем і забезпечує повноцінну логіку відображення, управління та навігації. Побудовані віджети інвентаря, гравця та карти охоплюють основні сценарії використання і разом формують єдину функціональну структуру, здатну підтримувати гнучку й ефективну взаємодію без втрати цілісності візуального оформлення.

3.5 Приклади найцікавіших алгоритмів та методів

У межах проєктування ігрової системи важливе місце займає реалізація механізмів, що забезпечують зв’язок між інтерфейсом користувача та об’єктами ігрового світу. Особливої уваги потребують алгоритми, які забезпечують обробку введення користувача в координатах екрану та його подальшу трансляцію у світ гри.

Один з таких прикладів — метод, який використовується для розрахунку кінцевої точки лінійного трасування (Line Trace), що виходить з позиції камери Scene Capture та спрямовується до точки на карті, куди гравець встановлює маркер натисканням. Алгоритм потрібен для визначення координати на поверхні ігрового рівня, яка відповідає вибраній точці на зображенні карти, що виводиться через Render Target. Проєктування цієї логіки виконується безпосередньо у віджеті карти, як частина обробки взаємодії з її областю.

Основна задача — отримати позицію у світових координатах, яка буде відповідати місцю кліку гравця у межах області мапи на екрані. Для цього обчислюється абсолютна позиція кліку в координатах інтерфейсу, визначаються розміри панелі, межі карти у сцені (верхній лівий і нижній правий кути), а також положення самої Scene Capture камери.

Координата кліку на карті нормалізується відповідно до розмірів панелі. Потім вона масштабується у світові координати, з урахуванням реального розміру ділянки карти, яка захоплюється камерою згори. Отримане значення і є точкою, до якої має бути спрямовано трасування — воно вказує, куди гравець «вказав» на мапі, і дозволяє точно визначити позицію маркера у світі (див. рис. 3.7).

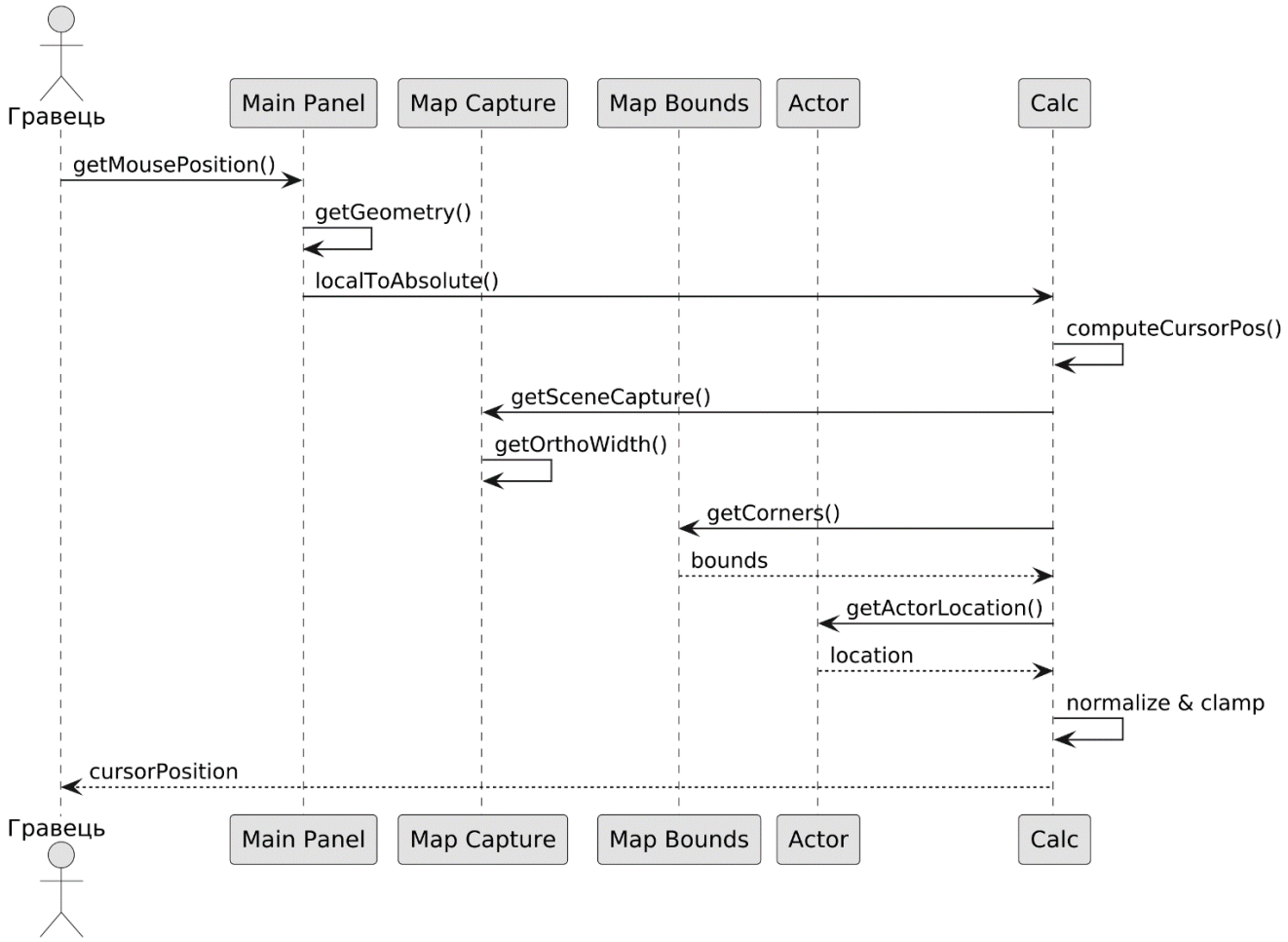


Рисунок 3.7 – Діаграма послідовності обробки позиції курсору на ігровій мапі (рисунок виконаний самостійно)

Другим прикладом є алгоритм, який визначає, чи потрібно програвати анімацію перегортання сторінки в журналі під час переходу між сторінками. У межах інтерфейсу журнал реалізовано як розгортку на дві сторінки, тож зміна сторінки супроводжується візуальним ефектом лише тоді, коли нова сторінка належить до іншої розгортки, ніж поточна.

Функція отримує поточну та нову сторінку. Якщо значення збігаються — анімація не запускається. Якщо різниця між ними перевищує один, вважається, що розгортка змінюється, і ефект перегортання виконується. Якщо ж різниця дорівнює одному, додатково перевіряється, чи сторінки не належать до однієї розгортки — наприклад, ліва та права в межах однієї пари. Якщо це так, анімація не потрібна. В іншому випадку — виконується (див. рис. 3.8).

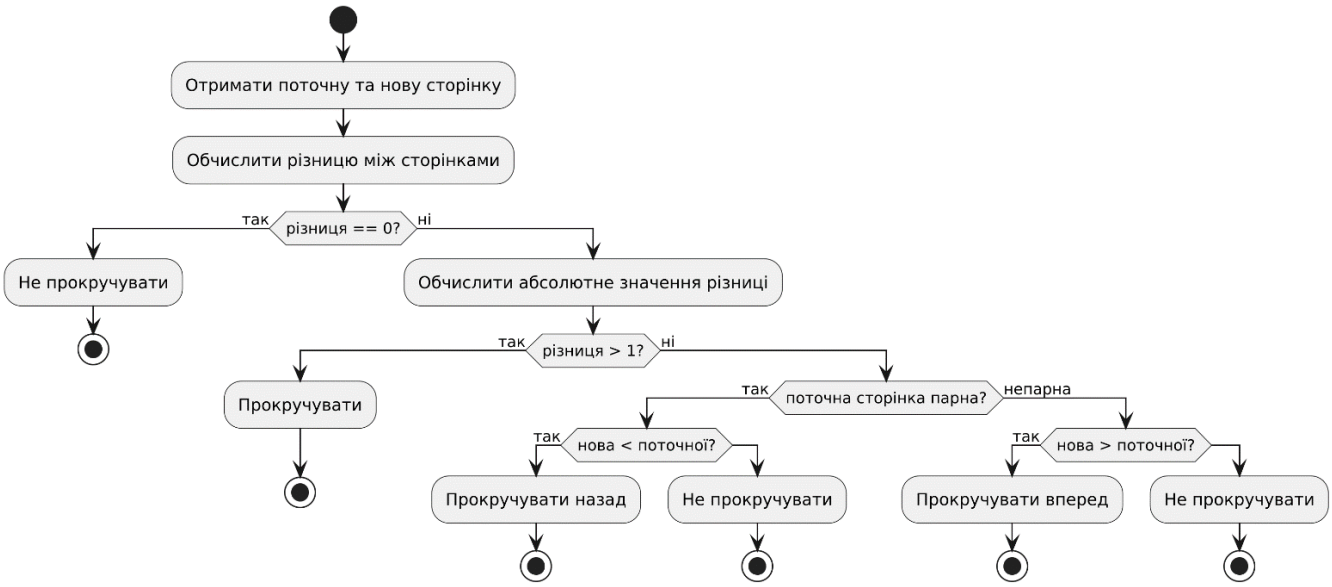


Рисунок 3.8 – Діаграма логіки визначення необхідності перегортання сторінки щоденника (рисунок виконаний самостійно)

В якості третього прикладу візьмемо метод класу персонажа TakeDamage, який викликається при отриманні персонажем шкоди. Він відповідає за зменшення здоров’я, оновлення стану, а також визначення напрямку удару для подальшого використання в анімаційній системі. Для його реалізації попередньо необхідно створити дві допоміжні функції, які використовуються для обчислення кута влучання відносно орієнтації персонажа.

Першою з них є GetYawRotationFromVector. Її завдання — обчислити горизонтальний кут повороту (Yaw) на основі переданого вектора, тобто визначити напрям у площині X–Y, який згодом можна використовувати в логіці персонажа.

Її отримане значення кута повертається та передається у CalculateDirectionBetweenAngles разом з орієнтацією персонажа у просторі. Її завдання — визначити відносний напрям між двома горизонтальними кутами (Yaw), що відповідають орієнтації самого персонажа та напрямку джерела влучання на момент отримання шкоди.

Завершує цей алгоритм метод TakeDamage, який виконує повну обробку отриманої шкоди. Його виклик супроводжується двома параметрами — значенням

завданої шкоди та вектором напрямку удару, що використовується для подальшого розрахунку просторового положення джерела влучання.

Першою дією функція встановлює булеву змінну RecentlyDamaged у значення true, після чого запускається таймер із затримкою 10 секунд, який повертає її до початкового стану. Ця змінна не бере участі у бойових обчисленнях напряму, але є важливою умовою для інших систем, зокрема відновлення здоров’я.

Після цього виконується віднімання шкоди від поточного значення Health. Якщо новий рівень здоров’я менший або дорівнює нулю, а персонаж ще не перебуває в стані смерті, перевірка завершується викликом функції Death.

Для визначення напрямку влучання використовується пара розглянутих допоміжних функцій. Результатом є значення типу EDirection, що вказує сторону удару — спереду, ззаду, зліва або справа. Це значення передається у функцію Death як аргумент і використовується для вибору відповідного анімаційного монтажу залежно від напрямку джерела шкоди (див. рис. 3.9).

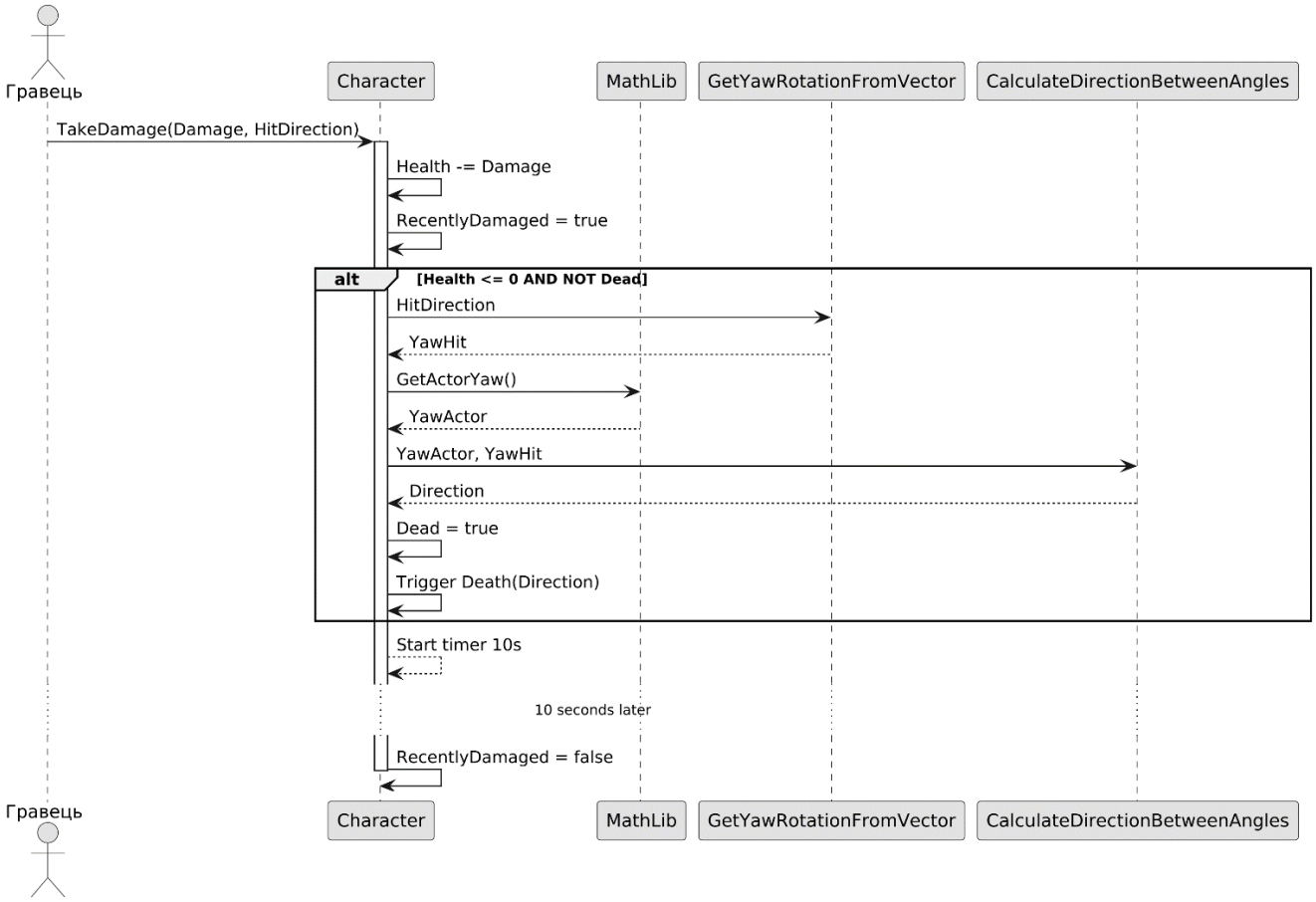


Рисунок 3.9 – Алгоритм обробки отримання шкоди та визначення напрямку влучання (рисунок виконаний самостійно)

Розглянуті приклади демонструють практичну реалізацію окремих елементів логіки ігрових систем, зокрема взаємодію з інтерфейсом, обробку подій і станів персонажа. Наведені методи охоплюють як ізольовані обчислення, так і послідовну обробку з участю допоміжних функцій, що формують завершений функціональний цикл. Усі алгоритми проєктувалися з урахуванням повторного використання, простоти інтеграції та відповідності загальній архітектурі застосунку.

У межах всього розділу було побудовано архітектуру програмного застосунку, розроблено структуру збереження даних, спроєктовано ключові елементи інтерфейсу користувача та наведено приклади найбільш важливих алгоритмів. Усі частини логічно пов’язані між собою й утворюють цілісну систему, придатну до розширення, адаптації та подальшої реалізації.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Організація технічної основи проєкту

Розробка ігрового застосунку «Northpoint» ведеться в середовищі Unreal Engine 5.5, що обрано як основну платформу для реалізації проєкту. На початковому етапі організовується технічна основа проєкту, яка слугує базою для подальшої реалізації конкретних ігрових механік і алгоритмів.

Першочергово було сформовано структуру директорій, що забезпечило чітке і зрозуміле розташування ресурсів, класів та інших елементів проєкту. Це дозволяє ефективно керувати файлами й ресурсами та значно спрощує процес розробки.

У рамках базових налаштувань рушія Nanite було вимкнено для статичних об'єктів в цілях економії ресурсів. Замість цього були використані традиційні LOD-моделі різних рівнів деталізації, які індивідуально налаштовуються для окремих категорій об’єктів — зокрема рослинності, архітектурних елементів і дрібних деталей сцени. Як основну систему глобального освітлення обрано Lumen, що забезпечує реалістичне динамічне освітлення без потреби в попередньому запіканні світла. Разом із Lumen було застосовано метод згладжування TAA (Temporal Anti-Aliasing), який найкраще інтегрується з цією системою й забезпечує м’яке згладжування країв зображення без значного навантаження на продуктивність.

Основний ігровий рівень було визначено як Persistent Level, який є базою для подальшого створення ігрового світу та дозволяє зручно додавати різноманітні компоненти та об'єкти гри. Рівень побудовано як єдину цілісну структуру без розбиття на підрівні, що спрощує редагування середовища та дозволяє безпосередньо взаємодіяти з усіма його елементами у рамках одного простору.

Після створення рівня було реалізовано перший клас ігрового персонажа у вигляді Blueprint-актора під назвою BP_Character. Цей клас став відправною точкою для подальшого впровадження системи керування, анімації та взаємодії з ігровим світом. У налаштуваннях проєкту (Project Settings) BP_Character було встановлено як клас за замовчуванням для гравця, що дозволило одразу почати тестування механік без додаткової конфігурації.

На цьому етапі також було закладено основні принципи взаємодії між C++ та Blueprints, визначено правила розподілу логіки і налаштувань, що сприятиме швидкому та зручному впровадженню змін у майбутньому.

Таким чином, технічна основа проєкту забезпечує необхідну платформу для подальшої розробки всіх запланованих функцій, механік та алгоритмів, пов'язаних з персонажами, навколишнім світом та інтерактивністю гри.

4.2 Розробка ігрового рівня

Заповнення вмісту рівня починається зі створення базового ландшафту розміром 63×63 квадрати (quads), який є оптимальним співвідношенням між продуктивністю і якістю деталізації. Для цього використовується вбудований редактор ландшафту Unreal Engine, що дозволяє швидко формувати різноманітні геометричні форми й природні особливості місцевості.

Наступним кроком стало налаштування основних елементів освітлення й атмосфери. Було додано Directional Light (направлене джерело світла), Sky Light (світло неба), Post Process Volume (об’єм пост-обробки), Sky Atmosphere (атмосфера неба), Volumetric Cloud (об’ємні хмари) та Exponential Height Fog (експоненційний туман). Окремо створена додаткова зона пост-обробки (Post Process Volume), яка розташована під поверхнею води та відповідає за специфічні ефекти в цій зоні.

Для майбутнього циклу зміни дня та ночі створено Actor Blueprint, що керує компонентом Directional Light, забезпечуючи його обертання відповідно до заданих параметрів.

Після формування базової структури рівня було створено автоматичний матеріал для ландшафту в редакторі матеріалів Unreal Engine. Використані текстури для цього матеріалу були взяті з безкоштовного маркетплейсу Epic Games — Fab (раніше відомого як Quixel Bridge). Також звідти були завантажені 3D-моделі природних елементів, таких як скелі, каміння, а також дерева й трава, що розміщені на ландшафті за допомогою Foliage Editor. Крім цього, використані моделі елементів інтер'єру та інших конструкцій, серед яких фрагменти будівель,

паркани та кораблі. Для створення візуальної межі світу було додано Plane, що імітує поверхню води на горизонті.

Заключним етапом стало налаштування освітлення, пост-обробки та атмосферних ефектів, що забезпечило максимально реалістичне й цілісне зображення створеного середовища.

На скриншоті наведено результат розробки рівня в режимі Lit, що демонструє усі застосовані технічні рішення й налаштування в умовах, наближених до реального ігрового оточення (див. рис. 4.1).

Рисунок 4.1 – Вигляд ігрового рівня в режимі Lit (рисунок виконаний самостійно)

Після розташування об'єктів на рівні вручну до кожної 3D-моделі було додано рівні деталізації (LOD), враховуючи початкову кількість полігонів та вимоги до візуальної чіткості. До більшості моделей застосовано пресет «Small Prop», який автоматично створює чотири рівні деталізації, де перший є максимально чітким, а останній — оптимізованим для великих відстаней.

Для наочної демонстрації розподілу полігональної сітки після застосування LOD-моделей, той самий фрагмент рівня відображено в режимі Wireframe (див. рис. 4.2)

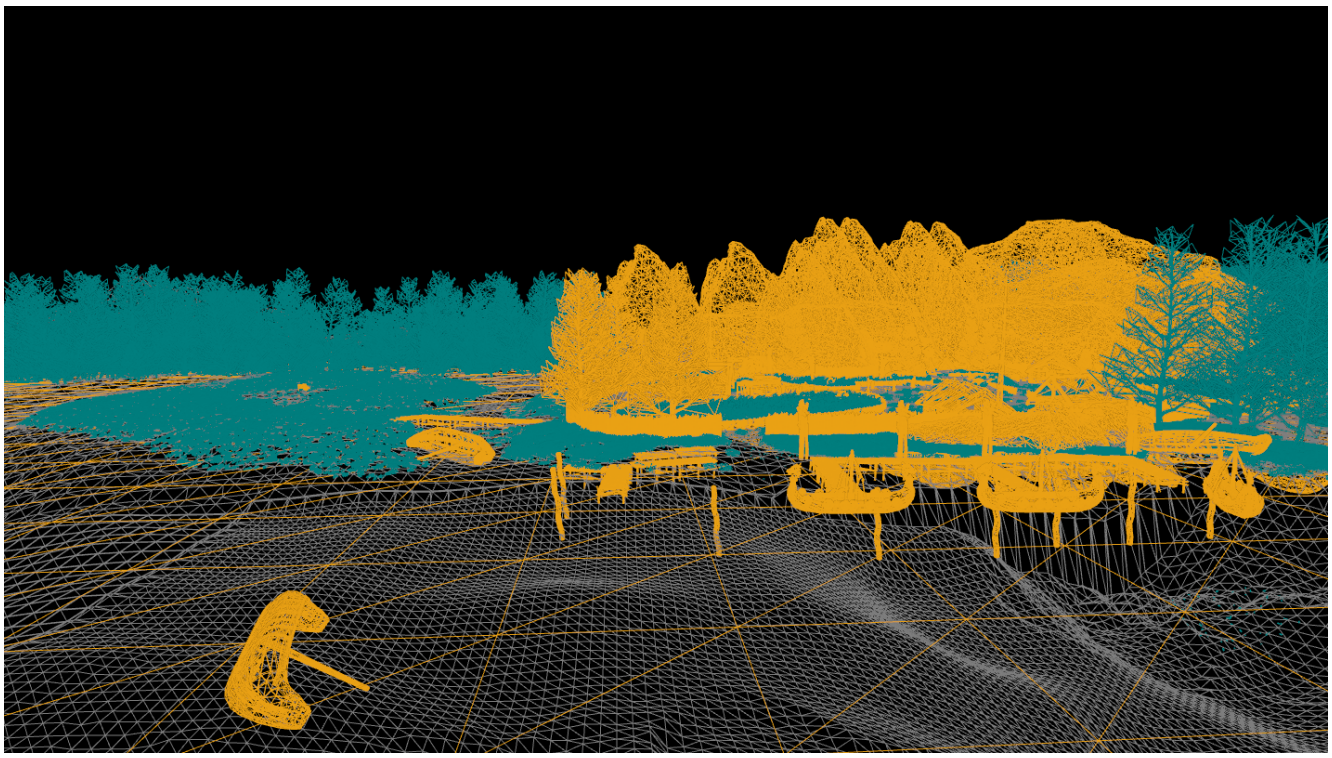


Рисунок 4.2 – Вигляд ігрового рівня в режимі Wireframe після налаштування LOD-моделей (рисунок виконаний самостійно)

Таким чином, було сформовано повноцінний ігровий рівень із базовими елементами середовища, освітленням, природніми ефектами та системою оптимізації через LOD-моделі. Далі опишемо розробку двох основних логічних одиниць ігрової системи — персонажа гравця та неігрових персонажів (NPC), які забезпечуватимуть динаміку й розвиток подій у створеному світі.

4.3 Розробка персонажа гравця

Після створення власного класу персонажа, похідного від стандартного класу Character, було здійснено базові налаштування компонента Character Movement. Встановлено дефолтну швидкість переміщення персонажа по землі на рівні 500 см/с, тоді як для повільної ходьби передбачено значення 200 см/с, а для майбутнього спринту — 800 см/с. Крім цього, параметри Acceleration та Braking Deceleration налаштовані таким чином, щоб імітувати реалістичний набір швидкості та зупинку персонажа під час руху в кожному з визначених режимів.

Наступним етапом стала реалізація анімації переміщення персонажа у просторі. Для цього було створено спеціальний Animation Blueprint з назвою ABP_Human. У ньому на кожне оновлення анімації (подія Blueprint Update Animation) розраховуються швидкості переміщення персонажа в напрямку вперед та у боковому напрямку. Ці значення обчислюються як скалярні добутки вектора швидкості персонажа (Velocity) та напрямків, заданих векторами актора вперед (Actor Forward Vector) і вправо (Actor Right Vector). Отримані величини зберігаються у відповідних змінних Speed Forward та Speed Side.

Для плавного переходу між анімаціями переміщення було створено спеціальний Blend Space. Це інструмент Unreal Engine, що дозволяє здійснювати плавне змішування анімацій залежно від двох параметрів. В даному випадку створений Blend Space має дві осі: горизонтальну зі значеннями від -500 до +500, та вертикальну від -500 до +800, що відповідає передбаченій різниці швидкостей руху персонажа вперед і назад. До кожної з точок цього простору анімацій були прив’язані конкретні анімаційні послідовності, що забезпечує плавний та реалістичний рух залежно від напрямку та швидкості переміщення.

Для отримання необхідних анімацій та вибору моделі персонажа було використано онлайн-платформу Mixamo, яка пропонує готові набори анімацій для тривимірних моделей персонажів (див. рис. 4.4). Звідти було завантажено модель персонажа «Paladin», яка відповідає стилістиці проєкту, а також підібрано анімації руху персонажа в різних напрямках і на різних швидкостях.

Перед імпортуванням отриманої скелетної 3D-моделі та набору анімацій до проєкту необхідно внести одну важливу зміну, що стосується базового кореневого вузла скелета персонажа (root bone). За замовчуванням моделі та анімації з платформи Mixamo не мають спеціального кореневого вузла, який дозволяє використовувати функцію root motion (анімування руху через переміщення кореневої кістки). Root motion особливо важливий для бойових анімацій, таких як перекати, а також анімацій взаємодії, де персонаж має плавно переміщуватися вслід за рухом цієї кореневої кістки.

Щоб вирішити цю проблему, використовується спеціальний плагін до Blender — Mixamo Converter. Цей плагін дозволяє легко редагувати імпортовані з Mixamo файли формату .fbx, додаючи до них спеціально налаштовану кореневу кістку, поведінка якої визначається згідно з налаштуваннями конвертера. Після завершення цієї процедури скелетну модель персонажа разом із повним набором анімацій можна без проблем імпортувати до Unreal Engine.

Імпортована модель персонажа встановлюється як дефолтна для Skeletal Mesh Component класу BP_Character. Після цього завантажені анімації розміщуються у створеному раніше Blend Space відповідно до швидкостей і напрямків руху, для яких вони були призначені. В результаті отримуємо діаграму, що нагадує павутину з чітко визначеними точками прив'язки кожної анімації до відповідних значень швидкості й напрямку руху персонажа (див. рисунок 4.3).

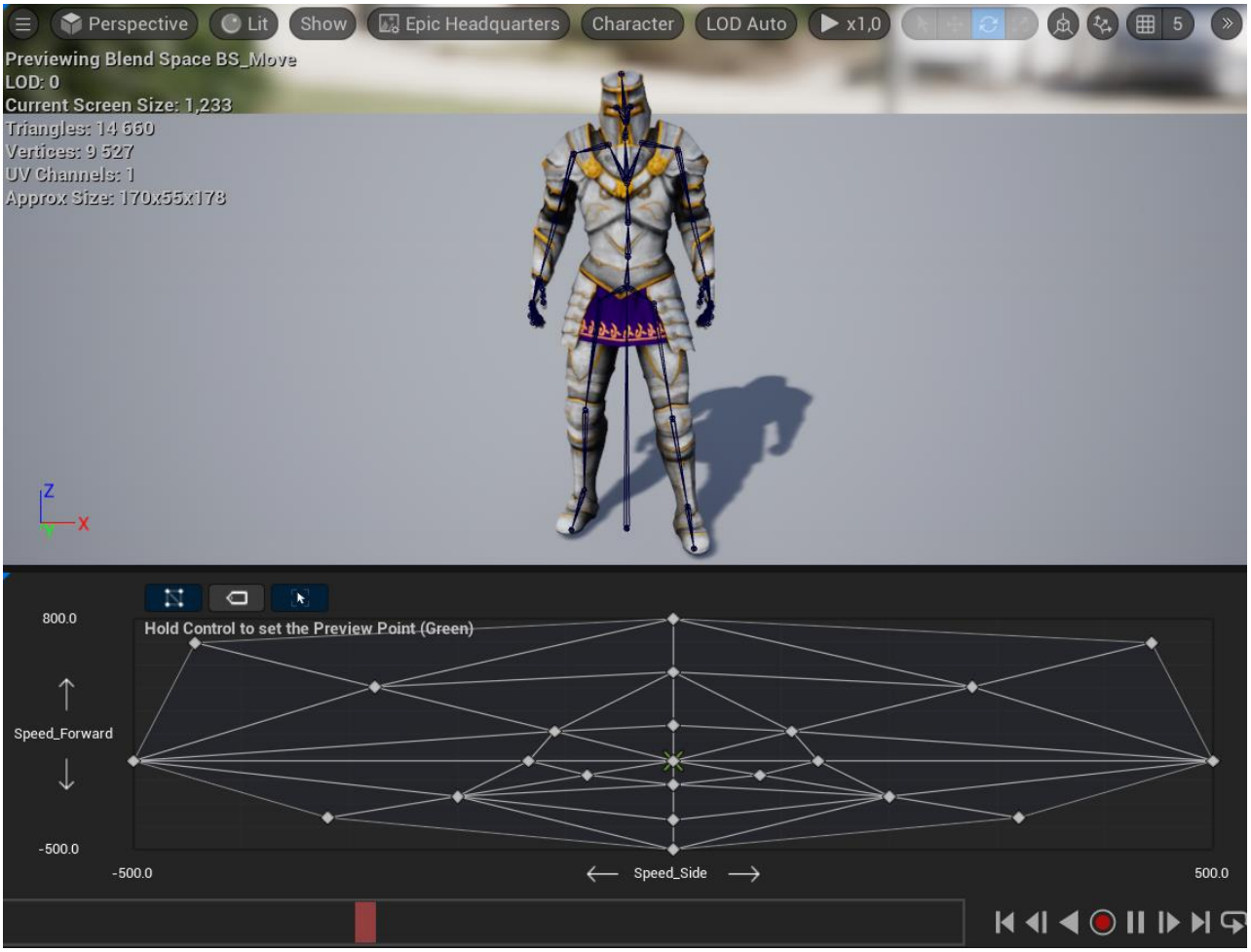


Рисунок 4.3 – Blend Space анімацій руху персонажа (рисунок виконаний самостійно)

На завершальному етапі анімування Blend Space підключаємо до Animation Graph створеного Animation Blueprint. Для цього до Blend Space додаються раніше створені змінні швидкості руху Speed Side та Speed Forward, після чого отриманий результат подається до базового загального слоту (Default Slot), призначеного для анімацій руху. Саме цей вихідний потік (Output Pose) визначає фактичне положення й анімацію скелетної моделі персонажа під час руху на ігровому рівні (див. рисунок 4.4).

Рисунок 4.4 – Animation Graph персонажа з базовим підключенням Blend Space (рисунок виконаний самостійно)

Застосування архітектури, у якій скелетна модель персонажа керується через Animation Blueprint, відкриває широкі можливості для масштабування анімаційної системи. До наявного Blend Space можуть додаватися альтернативні конфігурації для інших типів рухів, а також окремі анімаційні послідовності, які за умовами логіки вибираються та поєднуються функціями Blend Poses by Bool, Layered Blend per Bone, або іншими засобами анімаційного графа. У межах цієї системи можна локально змінювати позиції певних кісток за допомогою Transform (Modify) Bone, реалізовувати обернену кінематику (IK) для взаємодії з поверхнями чи об'єктами, а також програвати окремі анімації без потреби редагувати граф — за допомогою Montage-послідовностей, що викликаються з коду методом Play Anim Montage.

Останнім кроком впровадження системи управління стала реалізація безпосереднього керування персонажем гравця через налаштування відповідних input-подій. У меню Project Settings → Input створено необхідні осьові прив'язки

(Axis Mappings): для руху вперед і назад — Move Forward/Backward, де клавіші W та S отримують значення scale відповідно 1.0 та -1.0, а також аналогічно для руху вліво та вправо — Move Right/Left, де клавішам A та D встановлено scale 1.0 та -1.0. Додатково налаштовано рух камери за допомогою осей миші (Turn Right/Left Mouse та Look Up/Down Mouse), що безпосередньо пов'язані з параметрами MouseX та MouseY.

Для реалізації руху в класі BP_Character створено відповідні input-події, зокрема InputAxis Move Forward/Backward. Ця подія викликається щоразу при оновленні стану відповідної осі, передаючи на вихід актуальне значення (axis value) — від -1.0 до 1.0, що відповідає напрямку й інтенсивності руху гравця. На кожен виклик події виконується функція Add Movement Input, що додає імпульс руху персонажу. Параметром Scale при русі вперед виступає безпосереднє значення осі; при русі назад — значення, поділене на коефіцієнт 1.5; і при одночасному русі вбік — на коефіцієнт 2.5. Параметром напрямку (World Direction) при цьому виступає forward-вектор ротатора, утвореного від значення кута повороту (Yaw) контролера (Control Rotation), що визначає напрямок камери й, відповідно, руху персонажа (див. рис. 4.5).

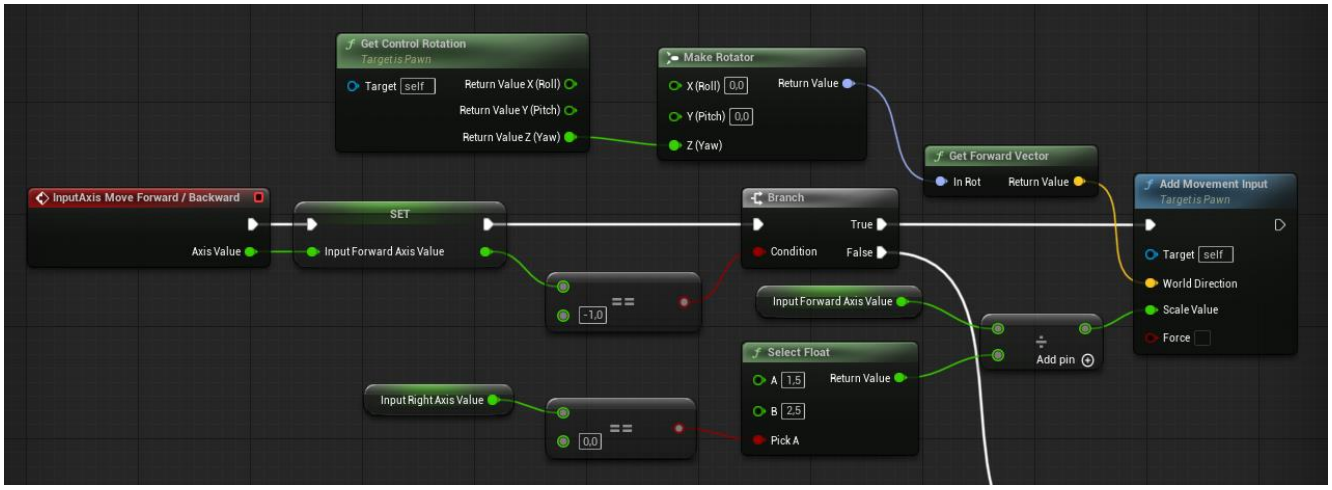


Рисунок 4.5 – Blueprint-схема базової обробки руху персонажа назад (рисунок виконаний самостійно)

Аналогічний підхід реалізовано й для руху в горизонтальній площині (вліво-вправо).

Для управління камерою використовуються події Input Axis Turn Right/Left Mouse та Input Axis Look Up/Down Mouse. У першому випадку викликається метод одного з батьківських класів Pawn — Add Controller Yaw Input, який відповідає за обертання контролера (а отже, і камери) по осі горизонтального повороту (ліворуч-праворуч). У другому — метод Add Controller Pitch Input, який змінює кут огляду по вертикалі, тобто вгору та вниз. Обидві функції забезпечують плавне й пряме управління камерою за допомогою рухів миші.

Керування персонажем реалізується за допомогою контролера, заснованого на стандартному класі PlayerController, який відповідає за обробку всіх вхідних команд актора Pawn. Саме цей контролер інтерпретує input-події та реалізує виклики функцій Add Movement Input, Add Controller Yaw Input та Add Controller Pitch Input, забезпечуючи коректну реакцію персонажа на дії користувача. Додавши до сцени точку появи (Player Start) та призначивши відповідний GameMode, у підсумку отримуємо функціонального персонажа, готового до переміщення по рівню (див. рисунок 4.6).

Рисунок 4.6 – Персонаж гравця на рівні в русі після налаштування керування (рисунок виконаний самостійно)

Отже, сформована структура класу персонажа поєднує гнучку систему управління через input-події та масштабовану анімаційну модель, засновану на

зв'язку між скелетною моделлю та Animation Blueprint. Архітектура класу дозволяє надалі розширювати функціонал шляхом додавання нових компонентів до кореневого об’єкта Capsule Component без необхідності внесення змін до базових зв’язків чи логіки класу. Такий підхід забезпечує необхідну модульність і створює основу для подальшої інтеграції наступних ігрових систем персонажа без потреби змінювати його базову архітектуру.

4.4 Розробка неігрових персонажів

Неігрового персонажа NPC представляє клас BP_BaseHuman, успадкований від того ж базового класу персонажа, що й персонаж гравця BP_Character. Він має аналогічні базові налаштування компонента Character Movement та використовує той самий Animation Blueprint, що забезпечує єдину систему руху та анімації для всіх персонажів у грі. Проте, на відміну від персонажа гравця, управління NPC здійснюється не через інпут-події користувача, які обробляються контролером гравця, а за допомогою окремої логіки, реалізованої через поведінкові дерева (Behavior Trees), що імітують поведінку та прийняття рішень штучним інтелектом. За виконання цієї логіки відповідає спеціалізований контролер типу AIController, призначений для автономного керування персонажами.

Розробку логіки поведінки NPC розпочато зі створення базового дерева поведінки (Behavior Tree) під назвою BT_BaseHuman, яке використовує окремий чорний ящик даних (Blackboard) — BB_BaseHuman. До цього чорного ящика додано змінну-ключ типу int, яка визначає базові стани, у яких перебуває NPC. За замовчуванням вона має значення 0, тоді виконується гілка дерева, де перед будь-якою взаємодією спершу активується таск BTT_SetIdleState, який повертає всі внутрішні параметри NPC у нейтральний стан. Після цього персонаж взаємодіє з різними об'єктами по черзі залежно від часу доби (який перевіряють декоратори BTD_IsDay та BTD_IsNight) та наявності інтеракції з гравцем, що розділяє основну гілку стану 0 на дві підгілки: одна активується, якщо IsTalking = true — у цьому випадку персонаж веде діалог, а інша — коли IsTalking = false, і тоді він переходить до своєї щоденної рутини (див. рис. 4.7).

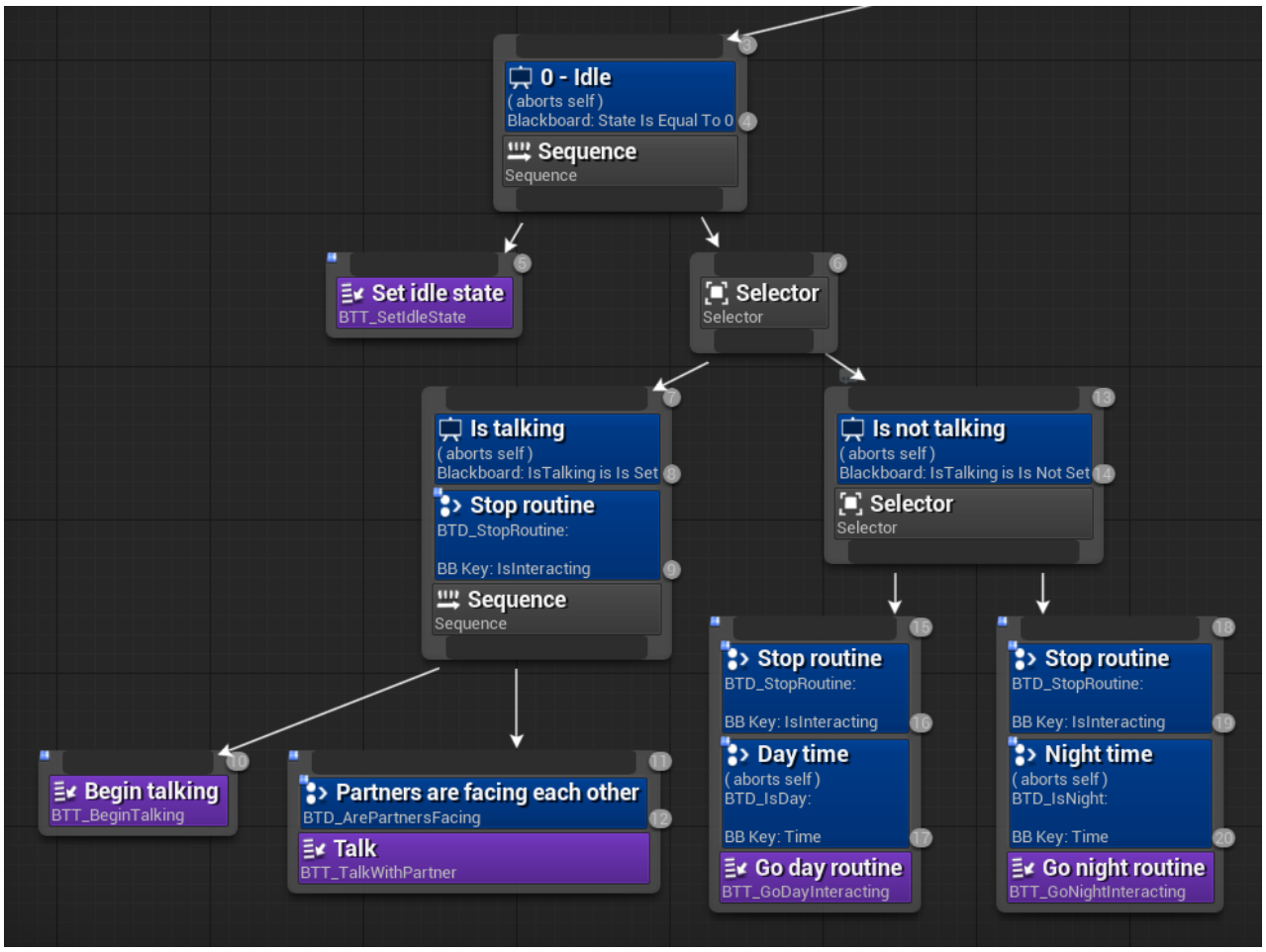


Рисунок 4.7 – Поведінкова гілка дерева BT_BaseHuman у стані 0 (рисунок виконаний самостійно)

Перехід між різними станами поведінки NPC ініціюється безпосередньо через їх спеціалізований AI-контролер, який має компонент AIPerception із відповідними органами сприйняття (Senses). При кожному оновленні будь-якого зі сенсів (подія OnTargetPerceptionUpdated) перевіряються визначені умови, після чого викликається відповідний обробник залежно від типу й достовірності отриманого сигналу. Так, якщо оновлюється сенс слуху (Hearing Sense) і джерело звуку класифікується як ворожий об'єкт, стан NPC встановлюється рівним 1 («пошук»), у якому персонаж негайно рухається до місця, звідки було почуто звук. Якщо на місці джерела персонаж не знаходить нічого підозрілого (не бачить противника через Sight Sense), то він повертається до стану спокою 0 і відновлює стандартну рутину.

Якщо ж на місці джерела NPC виявляє ворога за допомогою сенсу зору (Sight Sense), стан змінюється на значення 2 («бойова готовність»). У цьому стані NPC активує режим бою і намагається ліквідувати загрозу, використовуючи доступну зброю. У випадку, якщо ворог був ліквідований, поведінкове дерево (Behavior Tree), яке контролює бойову поведінку, викликає метод контролера Remove Enemy, що, за відсутності інших активних цілей, повертає NPC у стан спокою. Якщо ціль зникає з поля зору, то стан повертається до пошуку, в якому змінна-ключ state має значення 1, і NPC намагається знайти ворога за останньою відомою позицією. У разі успіху пошуку стан знову змінюється на 2, і персонаж повертається в режим бою; якщо ж і повторний пошук завершується невдачею, персонаж остаточно повертається у стан спокою 0 та відновлює свою звичайну рутину, очікуючи наступного активування одного з його органів сприйняття – або іншої ініціюючої події. Логіку переходів між станами NPC можна наочно представити у вигляді діаграми станів (див. рис. 4.8).

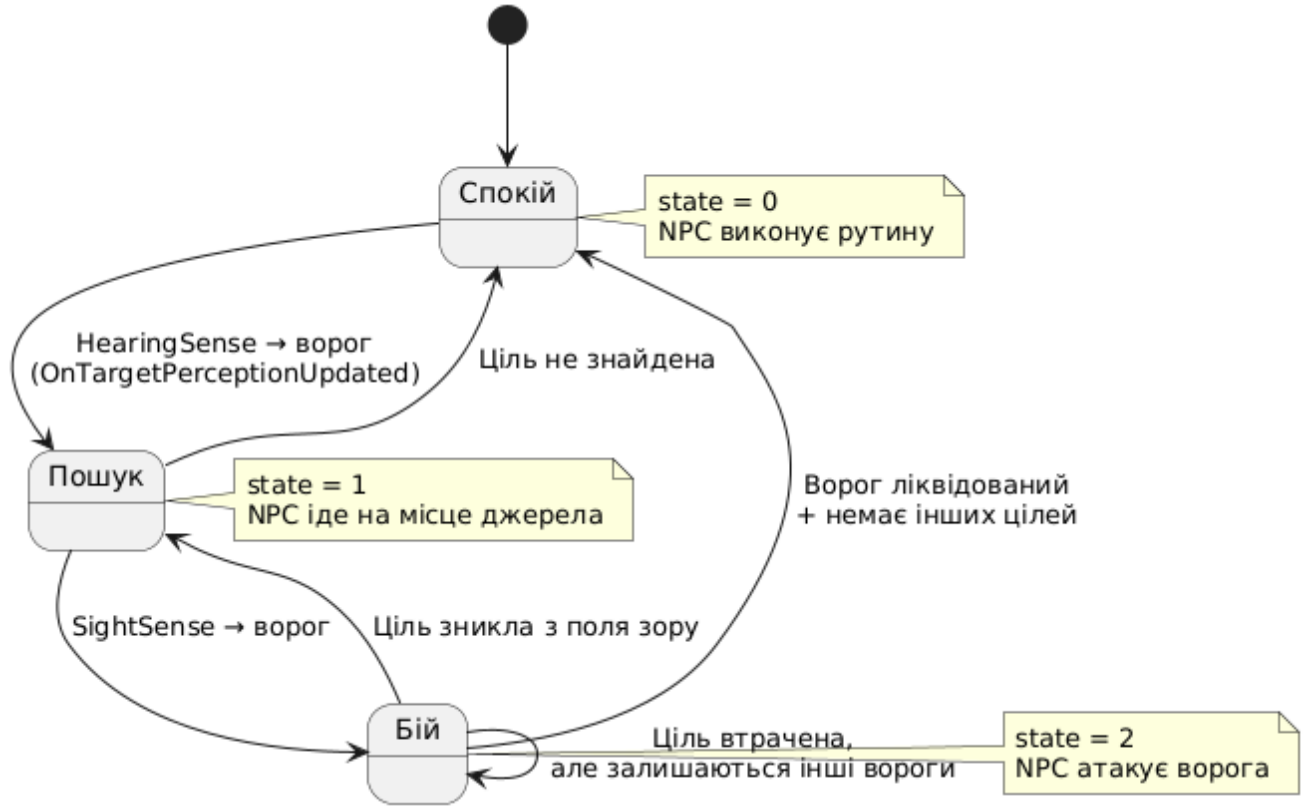


Рисунок 4.8 – Діаграма станів NPC із переходами між спокоєм, пошуком і боєм на основі роботи сенсорної системи (рисунок виконаний самостійно)

Залишається лише запустити поведінкове дерево за допомогою методу Run Behavior Tree у події BeginPlay, що реалізується безпосередньо в AI-контролері NPC, який має постійний доступ до відповідного дерева та пов’язаного з ним сховища. Після цього достатньо створити й призначити персонажу першу умовну точку взаємодії — і, перебуваючи у своєму стані 0, NPC автоматично рухається до неї відповідно до закладеної логіки (див. рис. 4.9).

Рисунок 4.9 – NPC на рівні в русі до точки взаємодії після запуску поведінкового дерева (рисунок виконаний самостійно)

Таким чином, реалізована система поведінки NPC забезпечує йому автономне функціонування в межах ігрового світу. Завдяки поєднанню Behavior Tree, AIController та сенсорної системи, персонаж здатен реагувати на події середовища, змінювати стани відповідно до контексту та взаємодіяти з об'єктами без участі гравця. Така структура дозволяє NPC органічно вписуватися в ігрову екосистему та виконувати складну логіку поведінки на основі закладених сценаріїв.

4.5 Розробка діалогової системи між персонажами

Наступним та фінальним етапом є встановлення взаємодії між персонажем гравця та NPC за допомогою спеціалізованої діалогової системи. Ця система

забезпечує безпосередній обмін інформацією між персонажами, надаючи можливість гравцю вести діалоги з неігровими персонажами, обираючи репліки з запропонованих варіантів відповідей. NPC, своєю чергою, будуть відповідати, обираючи власні репліки залежно від вибору гравця та загального перебігу розмови. Усі діалогові сценарії керуються спеціально розробленими поведінковими деревами Dialogue Behavior Tree зі сховищем змінних BB_Dialogue.

Для забезпечення роботи діалогової системи було створено окремий компонент Dialogue Actor Component, що додається до базового класу NPC — BP_BaseHuman. Цей компонент є технічною основою взаємодії і містить поведінкове дерево для діалогів, параметри якого мають позначки Instance Editable та Expose on Spawn. Це дозволяє встановлювати різні діалогові сценарії для кожного NPC індивідуально безпосередньо в редакторі рівня, при розташуванні персонажа в ігровому просторі. Також у компоненті присутня приватна змінна DialogueController класу AIC_Dialogue. Це спеціальний контролер, що потрібен для функціонування дерева поведінки діалогів та створюється автоматично під час початку гри (Begin Play).

Безпосередньо в момент початку розмови (при взаємодії гравця з NPC) діалогове дерево запускається методом Run Behavior Tree, після чого перебіг розмови контролюється виключно цим деревом. Для реалізації повноцінної логіки діалогу необхідно створити два основних таски поведінкового дерева: DBTT_Speak, що відповідатиме за виголошення реплік з боку NPC, та DBTT_Reply, який формуватиме й пропонуватиме гравцю варіанти відповіді. Крім того, до структури поведінкового дерева діалогів додано службові таски, такі як DBTT_Finish та DBTT_Stop, що забезпечують правильне завершення діалогів та повернення персонажів до нормального ігрового стану.

Почнемо з детального розгляду реалізації таску DBTT_Speak, який відповідає за виголошення реплік NPC у діалозі. Цей таск запускається поведінковим деревом у момент, коли стає активною відповідна гілка діалогу, і в ньому відразу викликається подія Receive Execute AI. Ця подія спершу знаходить віджет діалогу персонажа гравця, після чого прив'язує до його події завершення промови

(OnSpeakFinished) спеціальний делегат, створений всередині таску (FinishSpeak). Після цього викликається метод Speak, якому передається текст репліки NPC, а також встановлюється відповідне значення в чорній дошці (Blackboard), що сигналізує про початок виголошення репліки. По завершенню промови викликається подія FinishSpeak, яка відв'язує раніше встановлений делегат та завершує роботу таску шляхом виклику Finish Execute (див. рис. 4.10).

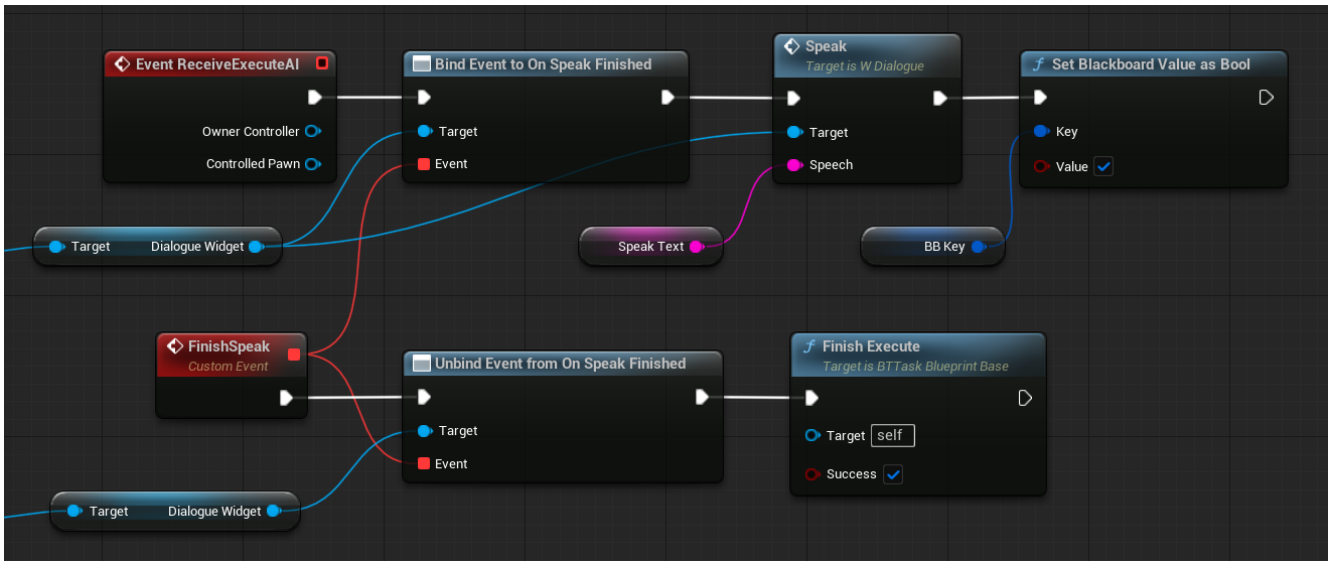


Рисунок 4.10 – Blueprint-схема роботи таску DBTT_Speak (рисунок виконаний самостійно)

Аналогічним чином працює й таск DBTT_Reply, але він має додаткову логіку роботи з масивом можливих відповідей, які пропонуються гравцю. Коли гравець обирає одну з відповідей, спеціальна подія, прив'язана до вибору, записує у чорну дошку відповідні значення: індекс обраної відповіді (ReplyIndex, типу int) та результат цього вибору (ReplyResult, типу bool), який визначає подальший напрямок діалогу. Поведінкове дерево через селектори відповідно до цих значень обирає правильну гілку продовження діалогу.

Завершення роботи діалогового дерева складається з двох окремих етапів. Першим є виклик таску DBTT_Finish, який вручну завершує взаємодію через дизінтеракцію — виклик функції інтерфейсу Interact з істинним значенням параметра stop. Цей етап виконується лише у випадку штатного закінчення діалогу, після вибору відповіді гравцем. Якщо ж взаємодія переривається інакше,

DBTT_Finish не викликається, і виконання дерева одразу переходить на гілку з таском DBTT_Stop, що виступає фінальним деструктором. У ньому встановлюються завершальні значення змінних у blackboard як компонента, так і NPC-власника, після чого логіка дерева повністю зупиняється через виклик Stop Logic компонента Brain контролера.

Базову структуру такого діалогового дерева поведінки показано на схемі (див. рис. 4.11).

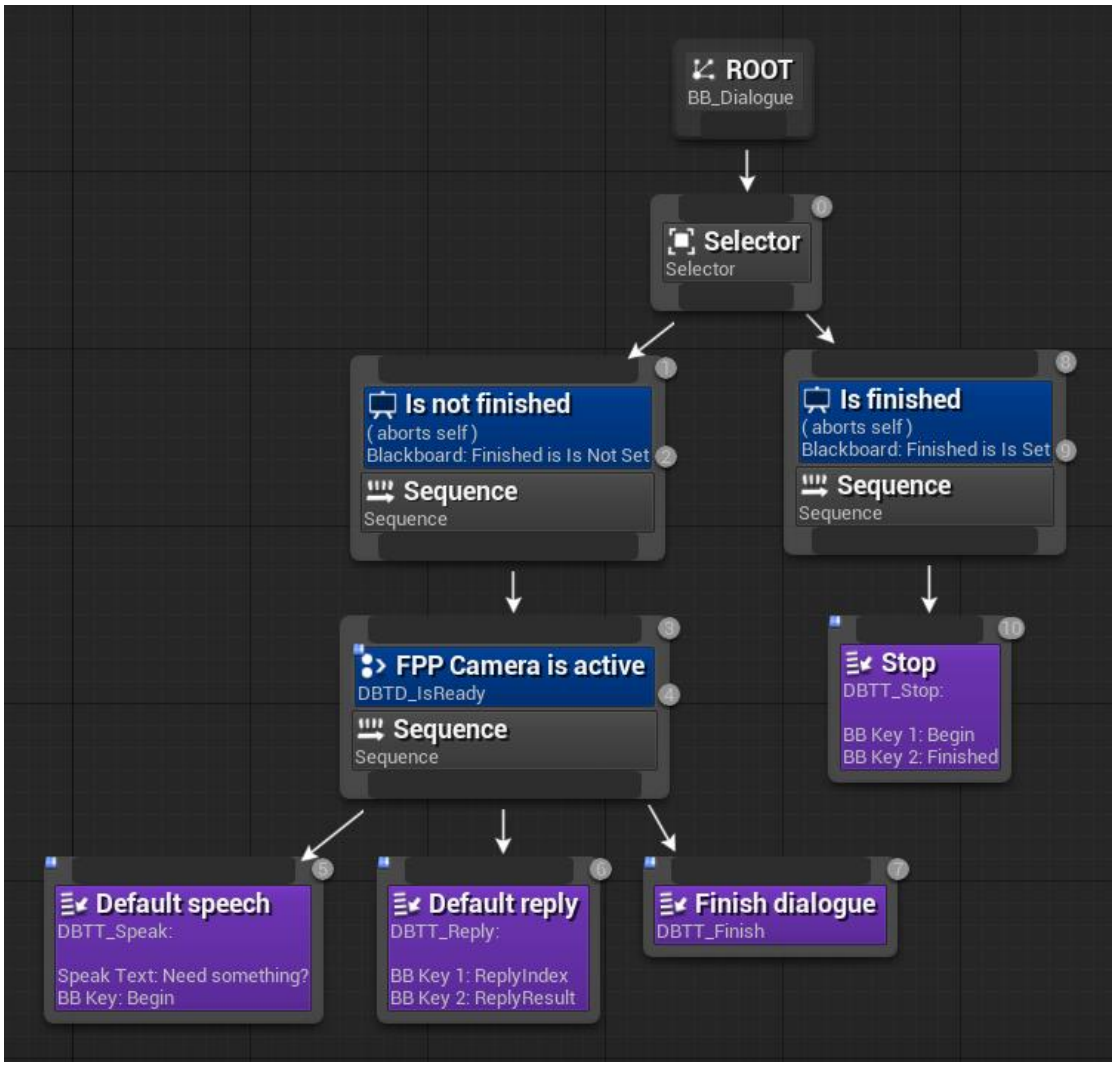


Рисунок 4.11 – Повна структура базового діалогового дерева поведінки для NPC (рисунок виконаний самостійно)

Запропонована діалогова система реалізує повноцінну взаємодію між персонажем гравця та NPC, дозволяючи здійснювати кероване обговорення з динамічним вибором реплік і реакціями на них. Рішення побудовано на

використанні окремого дерева поведінки, що керує перебігом розмови, а також модульного компонента, який забезпечує запуск діалогу й обробку логіки відповіді. Кожен з етапів — ініціація, обробка реплік, завершення — реалізовано через спеціалізовані таски, які можуть масштабуватися під складніші сценарії.

У межах усього розділу було описано реалізацію ключових ігрових систем, що охоплюють побудову ігрового середовища, створення керованого персонажа, розробку автономної логіки NPC та впровадження системи діалогів. Усі компоненти інтегруються у спільну архітектуру проєкту, функціонують у зв’язку один з одним і утворюють завершену функціональну структуру.

5 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування — один із ключових етапів життєвого циклу програмного забезпечення, що забезпечує впевненість у його працездатності, стійкості та відповідності проєктній логіці. У межах розробленого застосунку воно не менш важливе, ніж розробка функціоналу: навіть при чітко сформованій архітектурі складність взаємодії між окремими системами, подієвими зв’язками та інтерфейсами потребує ретельної перевірки всіх варіантів поведінки, включно з прикордонними та нестандартними.

Розробка застосунку в середовищі Unreal Engine 5 дає змогу використовувати кілька різних інструментів для відлагодження та тестування як логіки, створеної через Blueprint Visual Scripting (BVS), так і коду на C++.

Для візуальної логіки основним засобом виступає система брейкпоінтів, яку можна встановити на будь-який вузол у графі подій. У момент досягнення відповідного блоку виконання гри автоматично призупиняється, а розробник отримує змогу переглянути значення змінних у поточному контексті. Навіть без брейкпоінтів, в режимі реального часу Unreal Engine відображає активну послідовність виконання, що допомагає швидко локалізувати помилки у схемах.

Для глибшого аналізу, зокрема під час розробки більш абстрактних або критичних елементів системи, застосовується дебаг C++-коду за допомогою традиційних точок зупинки у середовищі розробки. Такий метод забезпечує розширений доступ до даних, викликів функцій, стеку та пам’яті. Водночас його недоліком є потреба у повній компіляції проєкту після змін, що робить ітерації помітно повільнішими.

У ситуаціях, коли необхідно спостерігати за перебігом подій без зупинки виконання, використовуються інструменти неперервного тестування, найпростіший з яких — виведення діагностичних повідомлень. У випадку BVS це метод Print String, що миттєво показує потрібне значення у вікні гри. У C++ для цього використовується макрос UE_LOG(), який дозволяє гнучко керувати форматуванням повідомлень та їхньою фільтрацією.

Крім локального дебагу, протягом розробки застосовувалися ручні інтеграційні перевірки: кожна функціональна підсистема перевірялась у повному сценарії роботи — наприклад, ініціація діалогу, перемикання поведінкових станів NPC, виявлення ворога сенсором або переходи між рівнями. Значна увага приділялася роботі систем у критичних ситуаціях — швидкісних змінах станів, викликах інтерфейсів з відсутніми посиланнями, або паралельному запуску взаємовиключних подій.

Для документування знайдених помилок створено таблицю виявлених дефектів, що містить наступні поля:

- назва помилки;
- короткий опис;
- компонент, де виявлено проблему;
- рівень критичності;
- пріоритет виправлення;
- покрокова інструкція відтворення;
- поточний результат;
- очікуваний результат.

Ця структура відповідає класичному формату тест-звіту або звіту про виявлені помилки (Bug Report). Усі зафіксовані в таблиці випадки були успішно опрацьовані, підтверджені та виправлені в поточному білді. Таким чином, таблиця водночас виступає валідаційним інструментом та фінальним зведенням пройдених тестів. Повний перелік виявлених помилок наведено у Додатку Г.

Вся розробка застосунку супроводжувалася постійним багаторівневим тестуванням, що охоплювало всі складові: від базових анімацій до логіки діалогу й поведінки NPC. Застосування різних підходів до дебагу, залежно від типу логіки, забезпечило ефективне виявлення й усунення помилок. Оформлення помилок у вигляді структурованого звіту дало змогу не лише контролювати прогрес виправлень, але й оцінити стабільність кожного етапу розробки. У результаті було досягнуто функціональної узгодженості всіх підсистем, що дозволяє вважати програмний застосунок стабільним та готовим до експлуатації.

6 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Процес впровадження програмного забезпечення є завершальним, але надзвичайно відповідальним етапом розробки. Саме на цьому етапі створений у середовищі Unreal Engine проєкт має бути повністю зібраний у самостійний виконуваний застосунок, придатний до запуску без використання рушія. Незважаючи на стабільну роботу проєкту в режимі редактора, саме процедура збирання (або пакування) є найскладнішою технічно: вона виявляє глибші залежності, помилки компіляції, відсутність ресурсів чи порушення структурних зв’язків у проєкті.

Перед безпосереднім збиранням застосунку необхідно забезпечити наявність актуального SDK цільової платформи, тобто комплекту інструментів для розгортання. У нашому випадку — це Windows 10 SDK, оскільки реалізація проєкту орієнтована на ПК із відповідною операційною системою.

У редакторі Unreal Engine через меню Platforms → Windows виконується вибір цільової платформи. Далі обирається конфігурація збірки — Shipping. На відміну від конфігурацій Development чи DebugGame, режим Shipping призначений безпосередньо для фінального розповсюдження. Він виключає налагоджувальну інформацію, зменшує розмір кінцевого виконуваного файлу, оптимізує продуктивність і забезпечує захист внутрішньої логіки гри.

Після вибору конфігурації та запуску процедури Package Project, обирається директорія на диску, де буде створено результати збирання. У процесі виконання системні повідомлення, пов’язані з попередженнями та помилками, виводяться у вікно Output Log. Лише після успішного завершення пакування та відсутності критичних збоїв, з’являється відповідне підтвердження у консолі, а у вказаному каталозі формується підпапка за назвою платформи – Windows, яка містить усі необхідні компоненти зібраного застосунку, зокрема виконуваний файл .exe, підкаталоги з ресурсами рушія та проєкту (Engine, Northpoint), а також супровідні маніфест-файли, необхідні для запуску, перевірки цілісності збірки та забезпечення взаємодії між структурними модулями системи. (див. рис. 6.1).

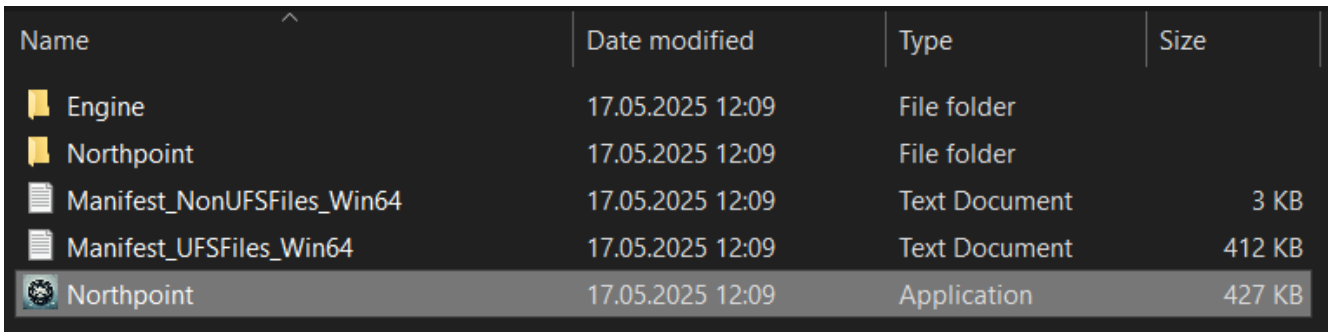


Рисунок 6.1 – Структура директорії зібраного застосунку після успішного пакування проєкту (рисунок виконаний самостійно)

Розглянемо основні елементи створеної структури докладніше. Каталог Engine містить службові дані Unreal Engine, які входять до складу кожного зібраного проєкту незалежно від його вмісту. Це набір інструментальних ресурсів рушія, включаючи стандартні бібліотеки, конфігураційні файли, шаблони запуску, базові системні компоненти для підтримки життєвого циклу програми тощо. Зміст цієї папки залишається практично незмінним між різними збірками й виступає універсальним середовищем виконання.

На відміну від неї, папка Northpoint є унікальною саме для цього проєкту. Тут зосереджено результати компіляції і безпосередньо вміст реалізованого функціоналу. Зокрема, у підкаталозі Binaries знаходяться готові бінарні файли — зібрані компоненти логіки, написаної мовою C++. Скомпільовані ж Blueprints та інші ассети зберігаються у відповідних ресурсних архівах — .pak, .ucas, .utoc. Вони містять усі візуальні, аудіо- та структурні ассети проєкту, упаковані відповідно до обраних налаштувань. Формати .ucas та .utoc забезпечують розподіл та адресацію вмісту, а .pak — сам архів ресурсу. Деякі типи архівів можуть бути зашифрованими або стиснутими, для прикладу використано базову, відкриту структуру пакування без шифрування, що спрощує розгортання.

Файли збереження прогресу за замовчуванням розміщуються за шляхом C:\Users\User\AppData\Local\Northpoint\Saved\SaveGames. Це — системна прихована директорія, яка не потребує прямого доступу з боку користувача. Уся взаємодія із збереженнями реалізована на рівні інтерфейсу гри, а сама структура відповідає стандартам Unreal Engine щодо роботи з локальними даними.

Нарешті, центральним елементом усього зібраного середовища є виконуваний файл Northpoint.exe. Саме з нього розпочинається запуск ігрового застосунку — він ініціює рушій, підключає ресурси, активує логіку поведінки та забезпечує повноцінну взаємодію гравця з ігровим світом. Для користувача це — єдиний необхідний елемент для старту: достатньо подвійного кліку, аби почати гру.

Таким чином, уся згенерована структура є повноцінним і завершеним пакетом застосунку, який може бути переданий як архів або каталог і використовуватись у продакшн-умовах для розповсюдження серед користувачів без залежності від рушія чи середовища розробки.

Підсумовуючи, можна зробити висновок, що процес впровадження програмного забезпечення — це не лише фінальний етап створення проєкту, а й критично важлива частина циклу розробки, що підтверджує працездатність усієї архітектури та забезпечує можливість її самостійного функціонування. У результаті реалізовано повноцінний ігровий застосунок, який готовий до використання кінцевим користувачем.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи бакалавра було спроєктовано та реалізовано ігровий програмний застосунок у жанрі Open-World RPG «Northpoint». Розроблений застосунок повністю відповідає визначеній меті й поставленим завданням роботи.

В результаті проведених досліджень та виконаних робіт були отримані наступні результати:

- розроблено архітектуру ігрового застосунку з відкритим світом, що забезпечує високу продуктивність, масштабованість та можливість інтеграції додаткових ігрових механік у майбутньому;
- реалізовано систему управління персонажем, яка підтримує різноманітні режими руху, дозволяє взаємодіяти з навколишнім середовищем та об’єктами у відкритому світі;
- створено функціональну бойову систему, яка включає в себе різні типи атак і механізми оборони, що забезпечують динамічний ігровий процес;
- впроваджено систему характеристик персонажа та навичок, що дозволяє поступово розвивати ігрового героя, створюючи різноманітні ігрові стилі;
- реалізовано механіки взаємодії з неігровими персонажами (NPC), які мають реалістичну поведінку завдяки застосуванню штучного інтелекту, що підвищує рівень занурення гравця у світ гри
- створено систему діалогів з варіативними відповідями, яка впливає на перебіг ігрових подій, а також систему квестів із зручним журналом завдань для організації ігрового процесу
- розроблено та впроваджено інтерактивну карту і систему навігації, яка значно спрощує орієнтування гравця у великому відкритому світі
- реалізовано систему збереження та завантаження ігрового прогресу, що гарантує стабільність і комфортність використання застосунку гравцями
- створено інтуїтивно зрозумілий інтерфейс користувача, що дозволяє легко взаємодіяти з усіма ігровими системами та механіками

Оцінюючи отримані результати, можна зазначити, що застосунок «Northpoint» відповідає сучасним вимогам до ігор жанру Open-World RPG як з погляду технічного виконання, так і з погляду перспектив подальшого розвитку та розширення контенту. Розроблені рішення є актуальними і можуть бути використані не тільки для подальшої комерціалізації продукту, але і як приклад для створення інших аналогічних проєктів у сфері ігрової індустрії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Пітерсон Дж. Playing at the World: A History of Simulating Wars, People and Fantastic Adventures from Chess to Role-Playing Games. – 2nd ed. – Lake Geneva, WI: Unreason Press, 2012. – 720 с.

2. Narrative Weight in Game Design: The Witcher 3 and Mass Effect [Електронний ресурс]. – Режим доступу: https://gameworldobserver.com/2022/12/16/narrative-weight-in-game-design-witcher-3-mass-effect (дата звернення: 20.05.2025).

3. Бартон М. Dungeons and Desktops: The History of Computer Role-Playing Games. – Boca Raton: A K Peters/CRC Press, 2008. – 648 с.

4. Келер К. Power-Up: How Japanese Video Games Gave the World an Extra Life. – Mineola, NY: Dover Publications, 2016. – 336 с.

5. Шраєр Дж. Blood, Sweat, and Pixels: The Triumphant, Turbulent Stories Behind How Video Games Are Made. – New York: Harper Paperbacks, 2017. – 320 с.

6. Кастронова Е. Synthetic Worlds: The Business and Culture of Online Games. – Chicago: University of Chicago Press, 2005. – 344 с.

7. Бейтман К. Game Writing: Narrative Skills for Videogames. – Boston: Charles River Media, 2007. – 308 с.

8. Джуул Дж. Half-Real: Video Games between Real Rules and Fictional Worlds. – Cambridge, MA: MIT Press, 2005. – 254 с.

9. Адамс Е. Fundamentals of Game Design. – 3rd ed. – Berkeley, CA: New Riders, 2014. – 576 с.

10. Мюррей Дж. Hamlet on the Holodeck: The Future of Narrative in Cyberspace. – Updated ed. – Cambridge, MA: MIT Press, 2016. – 324 с.

11. Шампандар А. AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors. – Indianapolis, IN: New Riders, 2004. – 720 с.

12. Oblivion NPCs Brought Their World to Life – Then They Nearly Killed It [Електронний ресурс]. – Режим доступу: https://www.escapistmagazine.com/oblivion-npcs-brought-their-world-to-life-then-they-nearly-killed-it/ (дата звернення: 20.05.2025).

13. Грегорі Дж. Game Engine Architecture. – 3rd ed. – Boca Raton: A K Peters/CRC Press, 2018. – 1240 с.

14. Акенайн-Меллер Т., Гейнс Е., Гоффман Н. Real-Time Rendering. – 4th ed. – Boca Raton: A K Peters/CRC Press, 2018. – 1198 с.

15. Skyrim Wiki [Електронний ресурс]. – Режим доступу: https://skyrim.fandom.com/wiki/Skyrim_Wiki (дата звернення: 21.05.2025).

16. God of War Ragnarök Wiki [Електронний ресурс]. – Режим доступу: https://godofwar.fandom.com/wiki/God_of_War_Ragnar%C3%B6k (дата звернення: 21.05.2025).

17. Khajiit Stealth Archer Is So OP [Електронний ресурс]. – Режим доступу: https://www.reddit.com/r/skyrim/comments/10iznsm/khajiit_stealth_archer_is_so_op_lol/ (дата звернення: 20.05.2025).

18. When you have no potions mid-combat, so you resort to eating food [Відео] [Електронний ресурс]. – Режим доступу: https://youtu.be/Rn3HEwqq4Mo (дата звернення: 20.05.2025).

19. God of War Ragnarök – The Quest for Tyr Walkthrough [Електронний ресурс]. – Режим доступу: https://www.pushsquare.com/guides/god-of-war-ragnarok-the-quest-for-tyr-walkthrough (дата звернення: 21.05.2025).

20. Unreal Engine Documentation [Електронний ресурс]. – Режим доступу: https://docs.unrealengine.com/ (дата звернення: 29.05.2025).

21. C++ Reference [Електронний ресурс]. – Режим доступу: https://en.cppreference.com/ (дата звернення: 29.05.2025).

22. Ракітін В. О. 2025_B_PI_PZPI-21-11_Rakitin_V_O [Електронний ресурс]. – Режим доступу: https://github.com/NureRakitinVladyslav/2025_B_PI_PZPI-21-11_Rakitin_V_O (дата звернення: 14.06.2025).