Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА 2D TOP DOWN ГРИ «FAIRYTALE ADVENTURE» В ЖАНРІ RPG З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF THE 2D TOP DOWN RPG GAME «FAIRYTALE ADVENTURE» USING UNITY

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-42
Колісніченко Дмитро Віталійович

Керівник:
Ліщина Наталія Миколаївна

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«*20*» *12* 20*24* р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Колісніченку Дмитру Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка 2D Top Down гри «Fairytale Adventure» в жанрі RPG з використанням Unity»*
Керівник роботи: *к.т.н., доцент Ліщина Н. М.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *ігровий рушій Unity, середовище розробки Visual Studio, методичні вказівки до виконання кваліфікаційної роботи бакалавра*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *дослідження предметної області, проведення аналізу існуючих аналогів, визначення вимог, проектування проекту та його розробка, проведення тестування та створення інсталяційного пакету*
5. Перелік графічного матеріалу: *20 графічних рисунків, 4 таблиці, 1 додаток*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Ліщина Н. М.*		
Специфікація вимог до розробленої системи	*Ліщина Н. М.*		
Розробка об'єкта проектування	*Ліщина Н. М.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	1,65 %		
Академічна доброчесність	*Ліщина Н. М.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	Виконано
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	Виконано
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	Виконано
4	Розробка функціонально-структурної схеми роботи об'єкта проектування	до 29.03.2025 р.	Виконано
5	Практична реалізація об'єкта проектування	до 26.04.2025 р.	Виконано
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	Виконано
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	Виконано

Здобувач вищої освіти _____ Дмитро КОЛІСНІЧЕНКО

Керівник кваліфікаційної роботи _____ Наталія ЛІЩИНА

АНОТАЦІЯ

Колісніченко Д. В. Розробка 2D Top Down гри «Fairytale Adventure» в жанрі RPG з використанням Unity. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі здійснено аналіз предметної області, порівняння існуючих аналогів продукту та технологій, сформульовано завдання на кваліфікаційну роботу бакалавра. В другому розділі було сформульовано функціональні та нефункціональні вимоги до проекту, описано проектування продукту, проаналізовано середовища розробки та програмне забезпечення, виконаний їх вибір для виконання поставленого завдання. В третьому розділі здійснено опис процесу розробки проекту та проведення його тестування. У висновках описано результати виконання кваліфікаційної роботи.

Ключові слова: ігровий застосунок, ігровий рушій, ігрові механіки, користувач, клас, прототип, інструмент розробки, скрипт, система, NPC.

ABSTRACT

Kolisnichenko D. Development of the 2D Top-Down RPG Game "Fairytale Adventure" Using Unity. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The first section analyzes the subject area, compares existing analogues of the product and technologies, formulates tasks for the bachelor's qualification work. The second section formulates functional and non-functional requirements for the project, describes the product design, analyzes the development of the environment and software, and selects them to perform the task. The third section describes the process of developing the project and conducting its testing. The conclusions describe the results of the qualification work.

Keywords: game application, game engine, game mechanics, user, class, prototype, development tool, script, system, NPC.

ЗМІСТ

ВСТУП

Актуальність теми. На даний момент, на ринку набирають попит інді-ігри, які пропонують хороший досвід, на відмінну від комерційних проектів великих студій. На противагу великим компаніям ААА-проектів, невеликі студії або самостійні розробники можуть проводити експерименти з механіками ігрового проекту, візуальною складовою та нарративом, які роблять цей сегмент особливо привабливим для дослідження. Проте, обмежені ресурси та відсутність стандартизованих технічних рішень ускладнюють процес розробки стабільної та захоплюючої гри.

Про сучасну ігрову індустрію можна сказати, що вона досить стрімко розвивається, особливо нові технології, які відкривають нові можливості для реалізації широких та інноваційних ідей. Незважаючи на наявність потужних інструментів розробки, подібних до Unity, розробка якісного 2D проекту є досить складним завданням. Це вимагає не тільки високих технічних навичок в сфері розробки програмного забезпечення, але й творчого підходу до реалізації програмного продукту.

Метою кваліфікаційної роботи є розробка 2D Top-Down гри з використанням Unity, що включатиме в себе базові механіки жанру: базова система персонажу, систему взаємодії NPC з гравцем та дослідження ігрового простору.

Об’єктом кваліфікаційної роботи є розробка 2D Top-Down гри на ігровому рушії Unity з елементами жанру RPG.

Предметом кваліфікаційної роботи є сучасні методи та технології створення програмних продуктів в сфері розробки комп’ютерних ігор.

Для досягнення цілей поставленої мети, потрібно виконати такий перелік завдань:

– дослідити предметну область ігрових застосунків; провести аналіз існуючих аналогів та визначити особливості ігрових систем, графіки та анімацій; виконати порівняння одного з алгоритмів ігрових механік;

– визначити функціональні та нефункціональні вимоги до кваліфікаційної роботи, виконати проектування проекту;

– провести порівняльний аналіз ігрових рушіїв та середовищ розробки з визначенням їхніх переваг і недоліків;

– реалізувати ключові механіки проекту з використанням сучасних інструментів розробки;

– розробити штучний інтелект для NPC;

– провести тестування виконаного продукту на факт зручності, простоти геймплею та технічної стабільності;

– створити інтерфейс ігрового застосунку;

– розробити інсталяційний пакет.

Потенційними користувачами програмного продукту є прості користувачі (гравці), які хочуть отримати простий ігровий досвід та розробники в сфері розробки комп’ютерних ігор.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДИЗАЙНУ ВІДЕОІГОР І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

На даний момент, сфера відеоігор демонструє стійку тенденцію розвитку інді-ігор [1], зокрема 2D ігрових продуктів. Це обумовлено кількома важливими факторами: відносною доступністю інструментів розробки та технологій, зростання попиту користувачів на стилістичну ретро-графіку та естетику, реалізацією ігрових механік без необхідності створення нових технічних рішень. Проте, як відображають останні дослідження [2], багато нових ігрових проектів не знаходять комерційного успіху на платформі Steam, що безпосередньо зв’язано з низькою якістю реалізації програмного продукту та її жанрових механік.

Технічний аналіз сучасних ігрових рушіїв для створення 2D-ігор виявляє суттєві відмінності у їх придатності для розробки різних механік, зокрема системи навігації або інші жанрові механіки. Тому перед тим як проектувати програмний продукт, варто визначити вимоги та підібрати ігровий рушій, який найкраще підходить для відповідно поставлених завдань [3].

1.1.1 Особливості жанру

Відео-ігри жанру RPG є одними з найпопулярніших в ігровій індустрії. Класичним прикладом таких ігор є такі проекти як The Binding of Issac, Commando Shooter 2D, Brotato, Stardew Valley та інші. Саме вони створили жанр Top-Down RPG 2D відеоігор. Їхніми особливостями можна легко назвати: вид зверху, управління та унікальну проекцію рівнів, де гравець досліджує ігровий світ. В 2D-реалізації особливого значення набувають рівновага між простотою візуальної подачі та глибиною ігрової взаємодії. Цей жанр використовує Tilemap-системи, NavMesh-навігацію та штучний інтелект ворогів як способи створення динамічного й нелінійного ігрового простору, де рішення гравця мають безпосередні наслідки в реальному часі.

Згодом цей жанр поступово розвивався з двовимірного простору графіки до тривимірного, з додаванням нових механік. Проте в наш час, жанр 2D ігри можуть легко конкурувати з великими проектами. Прикладом таких застосунків є Hollow Knight, Terraria, Dead Cells та інші.

1.1.2 Порівняльний аналіз комерційно успішних аналогів

Порівняльний аналіз успішних комерційних 2D проектів таких як Stardew Valley, Terraria, The Binding of Isaac та інші, дозволяє виявити певні підходи до розробки, що забезпечують їх успіх на ринку:

- застосування технік динамічної та якісної обробки ресурсів та графіки;
- адаптація технічних рішень під різну специфіку ігрових продуктів;
- гравець повинен отримати новий та унікальний для себе досвід.

Особливу увагу заслуговує досвід розробників ігрового проекту Stardew Walley [4], які демонструють ефективність певних технічних рішень, як: соціальна взаємодія NPC між собою, розвинута механіка землеробства, система зміни погоди та пори дня. Гра є симулятором фермерства з відкритим світом з елементами рольової гри та соціальної симуляції. Вона акцентує свою увагу на повільній грі, а саме на розвитку персонажа та середовища навколо нього.

У свою чергу Terraria [5], яку вважають клоном Minecraft завдяки відкритому світу, є пісочницею. Дослідження ігрового світу, будівництво, зіткнення з ворогами та розвиток відбувається в умовах поступово збільшеної складності. Ключовою особливістю даного продукту є можливість відігравання різних ролей, створення необмеженої кількості тактик та методів виживання. Окрім того, процедурна генерація світу та велика кількість контенту в грі надає їй непередбачуваності. Завдяки цьому, при повторному проходженні гри, гравець отримує новий досвід та заставляє його повернутись назад до гри.

The Binding of Isaac [6] представляє один із найвпливовіших експериментів у жанрі roguelike, який поєднує в собі простий візуальний інтерфейс з глибокою ігровою структурою. Гравець керує Ісааком, що втікає до жахливого підземелля, у якому кожна кімната – це не лише бойовий простір, а й потенційний елемент психологічного конфлікту. Ключовим аспектом є випадкове створення простору,

ворогів та предметів, що перетворює кожне проходження на унікальний досвід. Це не лише забезпечує високий рівень геймплею, а й створює постійне напруження невизначеності, до якого гравець змушений пристосовуватися.

1.1.3 Графіка та анімації

2D-графіка в сучасному стані існує в полі постійного поєднання між ретро-естетикою та мінімалістичними трендами цифрового арту, характерними для мобільних платформ. Подібне рішення не є декоративним – воно виконує комунікативну функцію, дозволяючи гравцю інтуїтивно розпізнавати об’єкти, їхню функціональну роль. Зокрема, палітра кольорів у більшості 2D-проектів є стандартною: яскраві кольори сигналізують про взаємодію, контрастні відтінки вказують на загрозу, а нейтральні – на фон чи другорядні елементи. Сама структура зображення у 2D-графіці потребує максимальної точності художника, оскільки кожен піксель має значення: не лише візуальне, а й функціональне.

Спрайти – основне візуальне зображення ігрових об’єктів. Вони імпортуються в проект як набір зображень й анімуються через інструмент Animator, що є перевагою для оптимізації продуктивності. Анімації забезпечують динамічну ідентифікацію станів об’єкта, моделює фізичну поведінку та створює ілюзію живого середовища. Найбільш поширені підходи до анімації в 2D-іграх включають кадрову анімацію (frame-by-frame) та скелетну анімацію (skeletal). Кожен із них має власну область застосування: від плавних переходів між станами персонажа до фізично та коректного руху ігрових об’єктів, анімацій навколишнього середовища.

1.1.4 Алгоритми пошуку шляху

Пошук шляху або алгоритми навігації – це процес побудови маршруту з однієї точки до іншої. В кваліфікаційній роботі використовується метод побудови навігації NavMesh. Даний метод зазвичай використовується в трьовимірних просторах, але його можна адаптувати і під двовимірний простір. До основних задач пошуку шляху відносять знаходження оптимального і найшвидшого маршруту.

NavMesh – це полігональна сітка, яка описує прохідні зони ігрового рівня. Алгоритми шляху використовують цю сітку для пошуку маршруту. Основними перевагами використання цього методу є:

– ефективність на великих рівнях, тобто після попереднього розрахунку рівня, пошук шляху відбувається швидко і автоматично;

– враховуються зміни в ігровому середовищі (наприклад, зруйновані об’єкти) в режимі онлайн, що дозволяє ефективно оновити карту навігації об’єктів;

– враховує рельєф та нахили поверхонь в ігровому середовищі та фізика об’єктів;

– здійснює підтримку анімацій, що дозволяє ігровим об’єктам адаптувати маршрут під нерівності місцевості.

Враховуючи ці переваги, NavMesh є універсальним та практичним методом розробки системи навігації, проте він має свої недоліки, такі як:

– обмежена точність у вузьких просторах, що може привести до некоректної роботи ігрових об’єктів у складних геометріях;

– висока вартість перерахунку, іншими словами динамічні зміни потребують перебудови сітки, що може призводити до проблем з ігровим середовищем;

– не підходить для 3D-навігації.

WaypointGraphs – це система навігації, що заснована на мережі розставлених точок (вейпоінтів), між якими здійснюють свій рух ігрові об’єкти. Перевагами даної системи є простота її реалізації, тобто не потребує складних розрахунків геометрії та низькі накладні витрати. Основними недоліками WaypointGraphs є не адаптивність до змін, тобто динамічні об’єкти не враховуються без написання додаткового коду та вимагають розставлення маркерів вручну, що забирає багато часу та заважає масштабуванню проекту.

Метод FlowFields використовує векторні поля для масового руху об’єктів, приклад її використання є стратегії реального часу. Оптимальний груповий рух ігрових об’єктів без конфліктів та динамічна адаптація до змін в ігровому

просторі є перевагами цього методу. Основними недоліками FlowFields є високі обчислювальні витрати ресурсів системи для оновлення полів та обмеженість роботи тільки в 2D-середовищах.

У результаті, провівши дослідження даних систем навігації було створено порівняльну таблицю, щоб детально розглянути їхні відмінності (табл. 1.1).

Таблиця 1.1 – Порівняльна таблиця методів алгоритмів пошуку шляху

Критерій	NavMesh	Waypoints	FlowFields	Voxel
Швидкість пошуку	Висока	Середня	Низька	Висока
Динамічність	Обмежена	Відсутня	Висока	Обмежена
3D-підтримка	Обмежена	Так	Ні	Так
Груповий рух	Середній	Поганий	Відмінний	Середній
Ресурси	Низькі	Низькі	Високі	Високі

Під час проведення аналізу предметної області, було виконано патентний пошук з метою знайти існуючі технічні рішення або технології, які пов’язані з розробкою 2D ігор, систем навігацій AI та створення рівнів. В результаті пошуку, не було виявлено серйозних обмежень або запатентованих технологій, які могли б завадити розробці цих методів в даній роботі. Варто зазначити, що відсутність суттєвих патентних обмежень в цій сфері пояснюється двома причинами: швидкістю розвитку ігрових технологій та тенденцією до відкритості в індустрії незалежної розробки ігор.

Як висновок, незважаючи на наявність різних інструментів розробки та технологій, створення власних систем для проекту є доцільним з наступних причин:

– можливість створення певних механік, які відповідають цілям кваліфікаційної роботи;

– повний контроль над архітектурою та логікою гри, що дозволить швидко внести зміни в проект та надає можливість розширення;

– поглиблення розуміння процесів розробки та інтеграції різних компонентів гри;

– отримання нових знань та застосування їх на практиці.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Завдання на кваліфікаційну роботу бакалавра полягає в розробці 2D Top-Down гри на рушії Unity, що поєднуватиме в собі класичні ігрові механіки та дизайн. Для виконання цього завдання, потрібно виконати наступні ключові етапи:

1) дослідити предметну область розробленого програмного продукту, виконати аналіз існуючих та популярних аналогів, дізнатись особливості графіки та анімацій, порівняти алгоритми однієї з ігрових механік;

2) визначити функціональні та нефункціональні вимоги до кваліфікаційної роботи, створити UML діаграми для опису класів та схеми зв’язків, виконати проектування UI частини проекту;

3) проаналізувати ігрові рушії та середовища розробки, визначити їхні переваги та недоліки, виконати їх порівняння за допомогою таблиць та здійснити вибір інструментів розробки;

4) розробити ігрові механіки для гнучкої реалізації проекту та здійснити їхній опис;

5) розробити штучний інтелект для поведінки NPC та здійснити його опис;

6) виконати тестування проекту за трьома сценаріями та описати їх: юніт-тестування, інтеграційне тестування та тестування продуктивності;

7) створити інтерфейс ігрового застосунку;

8) розробити інсталяційний пакет проекту.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАЛЬНОЇ ВІДЕОГРИ

2.1 Аналіз, визначення вимог до розроблювального програмного забезпечення та проектування програмного забезпечення

На сьогодні, розробка програмного забезпечення вимагає до себе системного підходу до аналізу та визначення конкретних вимог, що здійснюється після аналізу предметної області та обмежень інструментарію розробки. Цей процес передбачає створення не тільки функціональних вимог до проекту, але й потребує врахування таких нефункціональних вимог як продуктивність, безпека тощо. Провівши дослідження предметної області та аналіз існуючих аналогів, постало завдання сформувати функціональні та не функціональні вимоги до кваліфікаційної роботи. Для 2D Top-Down гри було сформовано наступні вимоги:

Функціональні вимоги:

– базова реалізація механік персонажу з урахуванням фізики (RigidBody2D) та обмеження ігрового простору;

– система атаки на основі колізій (Collider2D) з анімаційною обробкою подій;

– навігація NPC за допомогою методу NavMesh, адаптованого до 2D середовища (використання Collider2D для побудови маршруту);

– відображення інтерфейсу за допомогою UI Canvas;

– генерація рівня через Tilemap або вручну.

Нефункціональні вимоги:

– стабільність роботи на слабких системах;

– модульність архітектури файлів проекту та програмного коду;

– створення інсталяційного пакету.

Проектування розробленого ігрового застосунку розпочнімо з діаграми станів гравця (рис. 2.1), в якій описуються дії гравця в ігровому середовищі.

Рисунок 2.1 – Діаграма станів гравця в ігровому середовищі

Окрім діаграми станів, додатково було створено схеми зв’язків класів (рис.2.2) для об’єкту гравця, які відносяться лише для нього.

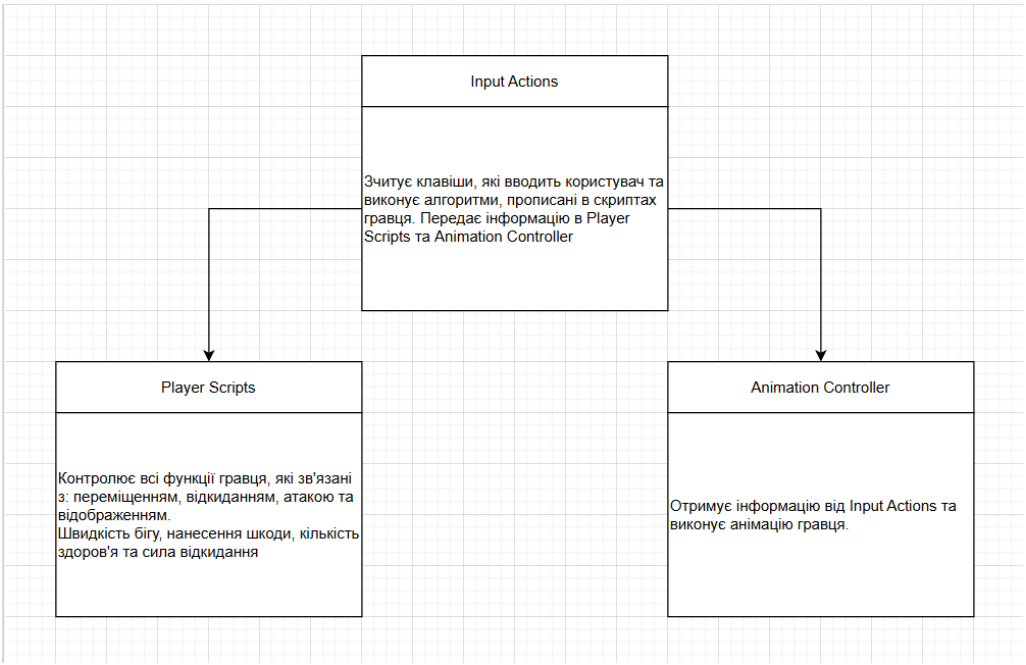

Рисунок 2.2 – Схема зв’язків об’єкту гравця

Діаграма для NPC (рис. 2.3) є дещо складнішою від систем гравця, оскільки в них є більше станів та скриптів, а саме: спокій, активність, стоїть, переміщення, патрулювання, переслідування, атака.

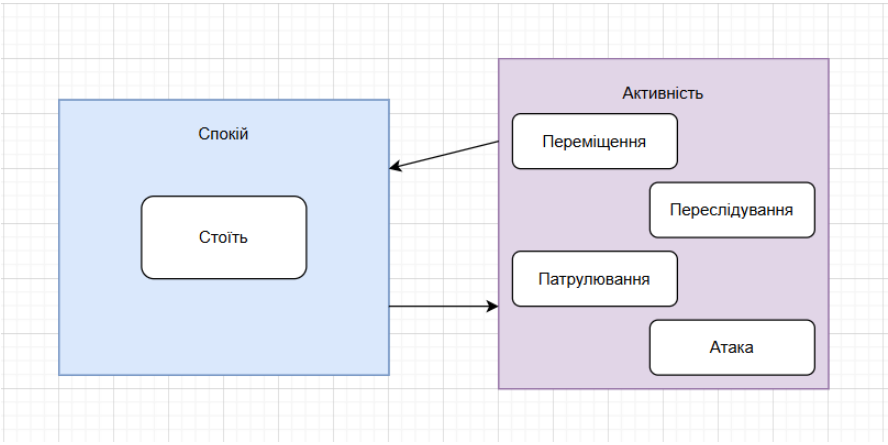

Рисунок 2.3 – Діаграма станів NPC

Спокій – стан, коли NPC знаходиться на одному місці та не може виконувати будь-які інші дії, окрім перехід в стан патрулювання та перевірка на те, чи є гравець в зоні тригеру.

Активність – стан, який активує поведінку NPC в залежності від ситуації, коли шукає гравця. Наприклад: переслідування, атака тощо.

Стоїть – підстан спокою, в якому NPC не виконує ніяких дій та не переміщається. Його наступні дії визначаються через підстани активності.

Переміщення – підстан активності, в якому NPC здійснює переміщення в ігровому середовищі по заданому маршруту.

Патрулювання – підстан активності, в якому NPC здійснює циклічне переміщення в певній зоні ігрового середовища. Зона є обмеженою.

Переслідування – один з підстанів активності, в якому NPC переслідує гравця. Коли гравець покидає зону тригеру, NPC повертається до стану патрулювання.

Атака – підстан активності, де NPC виконує атаку та наносить шкоду здоров’ю гравця.

Наступним чином, потрібно було описати класи, які б описували ці стани. До класів ворогів відносяться:

– Enemy AI;
– Enemy Life;
– EnemyVisual.

Клас Enemy AI відповідає за пошук гравця. В ньому реалізовано систему навігації, стани активності та спокою, події взаємодії з гравцем.

Клас Enemy Life відповідає за характеристики NPC та є зв’язаним з класом Enemy AI.

Клас Enemy Visual відповідає за роботу анімацій на відповідні події в ігровому просторі.

Після створення UML діаграм, наступним кроком стало створення прототипу інтерфейсу ігрового застосунку, а саме: головне меню, меню паузи та екран «смерті». Було проаналізовано сучасні тенденції дизайну в ігрових застосунках та зроблено висновок щодо інтерфейсу продукту: дизайн гри повинен бути простим та інтуїтивно зрозумілим для користувача, оскільки це є одним із ключових факторів привернення уваги користувача до продукту.

Для створення подібного рішення було вирішено використовувати не надто контрастні кольори: темні кольори фонів та світлі кольори для тексту. Такий підхід ґрунтується на принципах психології сприйняття. Темні відтінки фону зменшують навантаження на зір користувача під час тривалого використання застосунку, що особливо актуально для ігор із затяжними сеансами. Світлий колір тексту, навпаки, забезпечує високу контрастність, що сприяє легкому сприйняттю інформації. Крім того, така колірна схема сприяє кращому запам’ятовуванню інтерфейсу, що є важливим фактором для формування позитивного користувацького досвіду. Таке рішення дозволяє користувачеві швидко орієнтуватися в інтерфейсі, не відволікаючись на зайві деталі.

Першим прототипом інтерфейсу, яке було розроблено, стало головне меню гри. Головне меню (рис. 2.4) повинно містити назву гри, фонове зображення та дві кнопки: почати гру та вихід. Інтерфейс повинен відігрівати важливу роль, оскільки користувач вперше ознайомлюється з продуктом і перше враження від нього є важливим. У результаті, було створено елемент інтерфейсу у відповідній стилістиці проекту, його дизайн є простим та мінімалістичним.

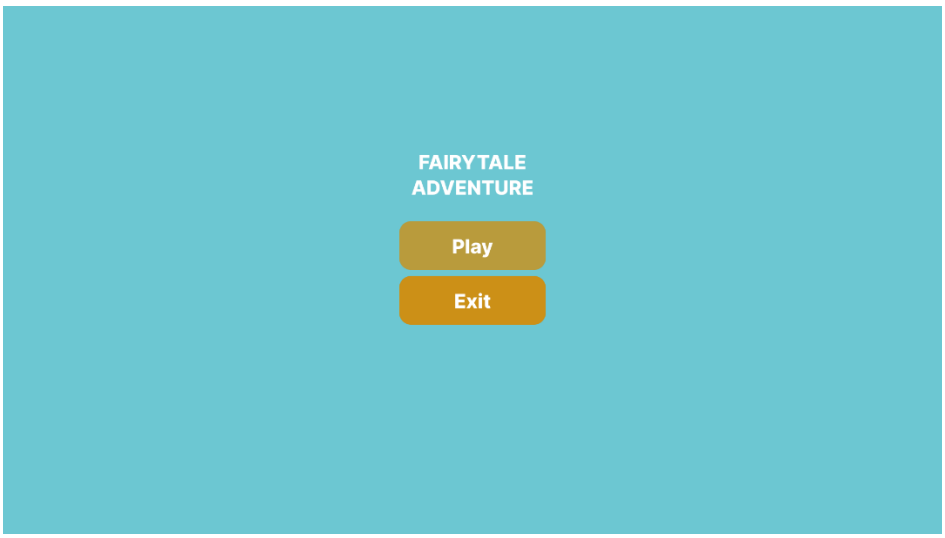

Рисунок 2.4 – Прототип дизайну головного меню

Наступним чином, було створено прототип екрану меню паузи (рис. 2.5), де розміщено фонове зображення ігрового застосунку та дві кнопки, які відповідають за продовження ігрової сесії та повернення до головного меню.

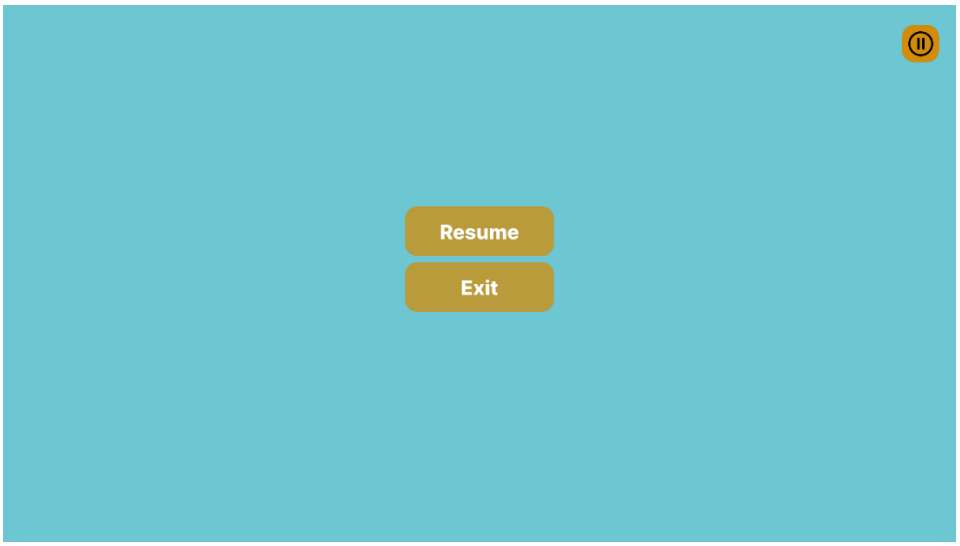

Рисунок 2.5 – Прототип дизайну меню паузи

Останнім елементом дизайну інтерфейсу гри був екран після смерті гравця та відображав два елементи: кнопки «Restart» та «Вихід».

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Сучасні методи розробки програмних продуктів характеризуються великим розмаїттям технічних рішень, кожне з яких має власні переваги для різних аспектів розробки. Враховуючи те, що складність поставлених задач зростає, важливо зробити правильні критерії відбору інструментів створення програмного забезпечення, а важливо враховувати саме: масштабованість, продуктивність, підтримка сучасних технології. У процесі вибору засобів для створення 2D гри, було проаналізовано ігрові рушії Unity, Unreal Engine, Godot, GameMaker, CryEngine.

Інструменти розробки Unity [7] є одним з унікальних ігрових рушіїв за рахунок універсальності, модульності та кросплатформеності – він дозволяє гнучко масштабувати проект та є заснованим на компонентно-орієнтованій моделі, що задовольняє вимоги інді-розробників та великих студій. Одним із ключових його особливостей полягає в модульному підході до створення ігрової логіки. Об’єкти, які розміщенні в середовищі формуються шляхом поєднання декількох компонентів, виконуючи різну роль: фізична взаємодія між собою, аудіо та візуальні елементи, трансформації та виконання скриптів, написаних мовою C#. Це не тільки оптимізує тестування та розробку, а й забезпечить високий рівень відтворення повторюваного коду, що важливо у проектуванні складних проектів. Ще варто виділити інструменти для візуалізації для роботи з 2D та 3D графікою, які пропонує Unity – Universal Render Pipeline та High Definition Render Pipeline [8]. Їхнє призначення це досягнення балансу між продуктивністю та якістю зображення. Окрім того, інтеграція з NavMesh, Tilemap та анімаційними системами, дозволяє створити якісну логіку руху об’єктів, навігації та взаємодії, що важливо для реалізації базових механік в проекті. Незважаючи на переваги цього рушія, він має певні обмеження, що може створити проблеми при розробці програмного продукту.

Відкритість екосистеми Unity надає свободу під час розробки, але вона залежить від зовнішніх пакетів, що можуть виявитись несумісними або застарілими з новими версіями ігрового рушію. Постійні оновлення супроводжуються з радикальними змінами прикладного програмного інтерфейсу (API), тому розробники повинні адаптувати свій код, що є не найкращим рішенням в умовах обмежених ресурсів.

На відмінну від інших ігрових рушіїв, які прагнуть перейти до спрощення процесу розробки на користь доступності, Unreal Engine [9] є складним інструментом для роботи. Його ядро побудоване на основі C++ , дозволяє програмістам працювати з низькорівневими структурами. Це дозволяє відкриває багато можливостей для оптимізації програмного продукту, полегшує проведення тестування та побудови комплексних систем взаємодії.

Однією з переваг, через який використовують Unreal Engine є його реалістичний рендеринг графіки. Цей рендеринг можливий завдяки використанню двох рушіїв – Lumen [10] та Nanite [11]. Lumen призначений для роботи з глобальним освітленням в режимі реального часу, а Nanite є системою віртуалізації геометрії – завдяки ним, можна працювати з моделями високої якості та щільності без використання великого об’єму ресурсів системи, що покращує продуктивність.

Окрім того, здатність Unreal Engine до обробки великого обсягу даних у режимі реального часу, дозволяють його використовувати не тільки для розробки ігор, але й в кінематографії та архітектурної візуалізації. Також він має відкритий код, що не є типово для геймдеву – це дозволяє адаптувати інструмент розробки під вимоги проекту. Однак, даний ігровий рушій не є ідеальним – він вимагає досить високих технічних характеристик, особливо під час тестування. Це може бути проблемою для невеликих команд або інді-розробників. По-друге, велика кількість налаштувань та складність інтерфейсу вимагають від програміста розширених знань в області програмування, розвинених навичок архітектурного моделювання та системного мислення.

Godot Engine [12] є альтернативою традиційної парадигми серед інших програмного забезпечення, таких як Unity. У даному інструменті розробки логічна структура проекту побудована на ієрархічному компонуванні «сцен», кожна з яких містить власний вузол: логічний, візуальний та фізичний. Завдяки цьому, Godot демонструє високу модульність та забезпечує повторне виконання компонентів без певних труднощів, на відмінну від інших ігрових рушіїв з подібною гнучкістю.

Одна з особливостей використання Godot Engine є метод скриптування GDScript [13], який спеціально створений під вимоги рушія та є оптимізований під його архітектуру. Синтаксис GDScript є схожим до мови програмування Python, що робить його доступним та легким для початківців, водночас надає потужність побудови складних логічних структур. Для досвідчених фахівців, ігровий рушій додатково підтримує C#, C++. А з новіших версій – власну систему компіляції компонентів через GDNative та WebAssembly, що робить його корисним у багатоплатформенному використанні з максимальною ефективністю.

З технічної точки зору, Godot може використовуватись як у 2D, так і 3D розробці. Він має один з найкращих побудованих 2D-пайплайнів, який не побудований на основі 3D-пайплайну (як у більшості ігрових рушіїв). Тобто, він має повноцінний рендеринговий модуль з власними алгоритмами освітлення, оброблення анімацій та колізій.

Однак, Godot має деякі обмеження в реалізації деяких функціональних можливостей та невелику команду розробників. По-перше, під час роботи з 3D-рендерингом, VR або штучним інтелектом, він потребує додаткових ресурсів або сторонніх бібліотек, що ускладнює розробку в вузькоспеціалізованих проектах. По-друге, через невелику команду розробників даного ігрового рушія, він змінюється повільніше, на відмінну від аналогів. Це створює проблему для студій, які є орієнтовані на динамічний ринок.

CryEngine [14] асоціюється з високотехнологічними візуальними можливостями, що піднімає стандарти у сфері реалізму в цифровому

моделюванні. Найбільшим успіхом використання цього інструменту розробки, пов’язують такі популярні ігри як серія ігор FarCry та Crysis, які були не лише комерційно успішними проектами, а й були результатом технічної демонстрації нових можливостей рушія на той момент. Завдяки цьому, його вважають рушієм, орієнтованим на високу візуальну досконалість.

Архітектурно CryEngine побудований як низькорівнева система, подібно до Unreal Engine, що забезпечує тісний контроль над рендерингом, фізикою, анімаціями та освітленням. Його модульна структура дозволяє реалізувати складні сцени з високим ступенем фізичної правдоподібності, проте не є адаптованою для початківців, як у Unity та Godot.

Ключовою перевагою даного рушія є його фотореалістичний рендеринг, що використовує просунуті шейдери, PBR (Physically-Based Rendering), HDR і реалістичне постоброблення сцен. Унаслідок цього, CryEngine часто обирають для симуляцій, VR-проектів високої якості та AAA-ігор, орієнтованих на деталізацію середовища. Його фізичний рушій CryPhysics базується на просунутих алгоритмах обробки колізій, динаміки тіл, м’яких об’єктів і руйнувань, що дозволяє моделювати складні взаємодії в реальному часі.

Однак, він вимагає ступеню входу для розробників – від них вимагаються знання C++ та навички з графічного рендерингу. Документація ігрового рушія визнана застарілою та неповно, не зважаючи на недавні оновлення. Це робить його використання непрактичним через обмежену доступність знань.

Окрім того, CryEngine має комерційну ліцензію, що робить його менш привабливим для інді-розробників або маленьких студій, у порівнянні з іншими середовищами розробки. Його зазвичай використовують для великих проектів, де виконується серйозний акцент на якість візуалізації та ефективність оброблення елементів, завдяки багатоядерній обробці та апаратному прискоренню.

GameMaker [15] займає особливе місце, завдяки простоті освоєння з інструментами для побудови складних проектів у 2D-просторі. Він відіграє важливу роль у сфері незалежної розробки. На відміну від універсальних рушіїв

широкого профілю, таких як Unity чи Unreal Engine, GameMaker зосереджений на оптимізації робочого процесу саме з 2D графікою, що суттєво впливає на його архітектуру, функціональну організацію та кінцеву продуктивність.

Архітектурна модель рушія побудована за моделлю, де кожен елемент сцени є об’єктом, який здатний реагувати на події, змінювати свій стан, генерувати логіку та взаємодіяти з іншими об’єктами. Користувачеві пропонується як візуальний редактор подій, так і текстовий синтаксис через GML (GameMaker Language) – скриптову мову, спеціально створену для даного рушія. Її синтаксис містить ряд спеціалізованих конструкцій, оптимізованих саме для ігрової розробки. Наприклад, вбудовані функції для роботи зі спрайтами чи звуковими ефектами дозволяють створити певні функції без додаткового коду. Завдяки цьому GameMaker успішно балансує між доступністю для початківців та достатньою гнучкістю для досвідчених розробників.

Перевагою ігрового рушію є глибока оптимізація процесів, пов’язаних із 2D-графікою. GameMaker забезпечує високий рівень рендерингу спрайтів, ефективне керування зіткненнями та підтримку анімаційних циклів. Також його перевагою є наявність вбудованого візуального редактора кімнат (rooms), що дозволяє оперативно конструювати сцени, керувати шарами, ефектами камери й освітлення.

Однак незважаючи на функціональну завершеність у 2D-середовищі, GameMaker має обмеження у сфері 3D-моделювання, що зумовлено як його архітектурною специфікою, так і профільною орієнтацією.

Наступним чином, було сформовано порівняльну таблицю (табл. 2.1) ігрових рушіїв, щоб розглянути їхні відмінності та обрати його для виконання кваліфікаційної роботи.

Таблиця 2.1 – Порівняльна таблиця ігрових рушіїв

Критерії	Unity	Unreal Engine	Godot	Game Maker	Cry Engine
Ліцензія	Безкоштовна	Безкоштовна	Безкоштовна	Платна	Безкоштовна
Підтримка мов	C#	C++	GDScript	GML	C++, Lua
Графічні можливості	HDRP, URP	Lumen та Nanite,	Вбудований 2D рушій	2D рушій	SVOGI, Ray Tracing

Продовження таблиці 2.1

Критерії	Unity	Unreal Engine	Godot	Game Maker	Cry Engine
Оптимізація	Середня	Висока	Висока	Дуже висока	Висока
Інструменти для 2D	Tilemap, Sprite Editor, Animator	Обмежені	Tilemap	Підтримка піксель-арту	Відсутні
Складність	Середня	Висока	Середня	Дуже низька	Дуже високий

У висновку, для реалізації проекту кваліфікаційної роботи було обрано ігровий рушій Unity як найбільше збалансований рушій з точки зору гнучкості, можливістю розширення, підтримки двовимірної графіки та інструментів для роботи з поведінковими алгоритмами об’єктів. Наступним чином, було проаналізовано середовища для реалізації програмного коду, зокрема Visual Studio, Jetbrains Rider, MonoDevelop.

Visual Studio [16] – це інтегроване середовище розробки від Microsoft, яка використовується для розробки програмного забезпечення. Воно постійно оновлюється відповідно до мов програмування, архітектурних шаблонів та стилів розробки. Завдяки глибокій інтеграції, високому рівню автоматизації та відкритої підтримки великого спектру технологій, Visual Studio є одним із найпотужніших інструментів у корпоративному та академічному середовищі.

Архітектурна побудова Visual Studio вирізняється модульною стратифікацією: ядро середовища доповнюється розширеннями, які забезпечують підтримку мов програмування (зокрема C#, C++, Python, JavaScript тощо), дебагінгу, тестування, рефакторингу, а також DevOps-інструментів. Завдяки такій моделі Visual Studio функціонує не як статична IDE, а як динамічне середовище, яке адаптується до потреб розробника. Центральним елементом функціональної системи є MSBuild, що забезпечує гнучке управління процесом компіляції, конфігурації середовищ виконання та модульності розгортання, дозволяючи реалізовувати складні проекти з великою кількістю залежностей.

Однак, Visual Studio має недоліки пов’язані з його вимогами. Повний об’єм середовища становить декілька десятків гігабайт, що не є правильним для

проектів малого або середнього ступеню складності. Окрім цього, його вимогливість до системних ресурсів та тривалий запуск у великих проектах створює проблеми для користувачів з обмеженим апаратним забезпеченням. Незважаючи, що версія Community має безкоштовну ліцензію, для комерційного або командного використання середовища повинна бути платна ліцензія, що є додатковими витратами для малих студій або команд розробників.

JetBrains Rider [17] – середовище розробки, яке орієнтоване на професійних C#-інженерів, які шукають альтернативу з фокусом на продуктивність, зручність та кросплатформену підтримку.

У своїй основі Rider поєднує двокомпонентну архітектуру: фронтенд, заснований на IntelliJ Platform (відповідає за UI, управління файлами, проектами та навігацією) і бекенд, що базується на ReSharper (відповідає за аналіз коду, рефакторинг і інтелектуальні підказки). Подібна модель забезпечує не лише функціональність, але й продуктивність, особливо при роботі з великими проектами. На відміну від Visual Studio, Rider не потерпає від критичних втрат продуктивності, що особливо цінується в корпоративному та індустріальному середовищі.

Однією з ключових переваг Rider є його підтримка пристроїв. Він є доступним для Windows, macOS і Linux без будь-яких функціональних обмежень, що критично важливо в умовах сучасної розробки, де інженери можуть працювати на різних операційних системах. Крім того, підтримка широкого спектра .NET-технологій – зокрема ASP.NET Core, що робить Rider універсальним інструментом для розробки.

Особливу увагу слід приділити аналітичним можливостям Rider. Інструмент підтримує понад 2500 інспекцій, здатних виявляти помилки, порушення стандартів оформлення коду, а також потенційні проблеми з продуктивністю або безпекою. Потужний механізм рефакторингу, логічна навігація по коду, а також гнучка система плагінів дозволяють глибоко налаштувати середовище під індивідуальний стиль розробника чи команди.

Проте слід визнати й обмеження цього інструменту – Rider є комерційним продуктом, доступним за передплатною моделлю, що унеможливлює його використання безкоштовно у більшості випадків. Також, хоча Rider є інтегрованим з .NET-екосистемою, його початкова конфігурація потребує дещо вищої підготовки від користувача, на відмінно від Visual Studio.

Розроблене в межах ініціативи Mono Project, що мала на меті створення відкритої реалізації .NET Framework для Linux та інших операційних систем, MonoDevelop [18] упродовж певного періоду був чи не єдиною реальною альтернативою Visual Studio.

Архітектурна модель MonoDevelop базується на модульному принципі з можливістю розширення за рахунок засобів, які реалізують підтримку мов програмування, компіляційних процесів, системи контролю версій та візуального дизайну інтерфейсів. Ключовою функціональною особливістю цього середовища була тісна інтеграція з компілятором mcs, який входить до складу Mono, а також з бібліотеками класів .NET Framework, що дозволяло створювати повноцінні C#-додатки на платформах Linux та macOS.

Серед функціональних переваг даного середовища розробки слід виділити підтримку дебагінгу, автодоповнення коду та рефакторингу, а також можливість компіляції та тестування .NET-проєктів без прив’язки до Windows-інфраструктури. Це робило його особливо актуальним у період, коли офіційна підтримка .NET Core ще не була достатньо розвиненою, а Visual Studio ще не досягнув зрілості.

Однак його використання є недоречним – він припинив отримувати підтримку з 2020 року та не відповідає сучасним вимогам і стандартам.

Наступним чином, було сформовано порівняльну таблицю (табл. 2.2) середовищ розробки, щоб розглянути їхні відмінності та обрати її для виконання кваліфікаційної роботи.

Таблиця 2.2 – Порівняльна таблиця середовищ розробки

Критерій	Visual Studio	Jetbrains Rider	MonoDevelop
Ліцензія	Безкоштовна	Платна	Безкоштовна
Інтеграція Unity	Повна	Повна	Обмежена

Продовження таблиці 2.2

Критерій	Visual Studio	Jetbrains Rider	MonoDevelop
Продуктивність	Середня	Висока	Низька
Підтримка мов	C#, C++, Python, JS	C#, VB.Net, JS	C#, VB.Net
Переваги	Підтримка Microsoft	Найкраща продуктивність	Легкий в освоєнні
Недоліки	Займає багато місця	Висока вартість	Немає підтримки

Проаналізувавши інтегровані середовища розробки, для виконання завдань кваліфікаційної роботи було обрано Visual Studio завдяки його інтеграції з Unity, безкоштовному та комфортному використанню.

РОЗДІЛ 3
РОЗРОБКА 2D TOP DOWN ГРИ «FAIRYTALE ADVENTURE»

3.1 Практична реалізація об’єкта проектування

Розробка програмного продукту розпочалась з розробки механік гравця. Потрібно було реалізувати наступні функціональні можливості: звичайне переміщення, атака та відкидання від атаки ворогів. Для зручної роботи, було створено ієрархію об’єктів в проекті, яка зображена на рисунку 3.1.

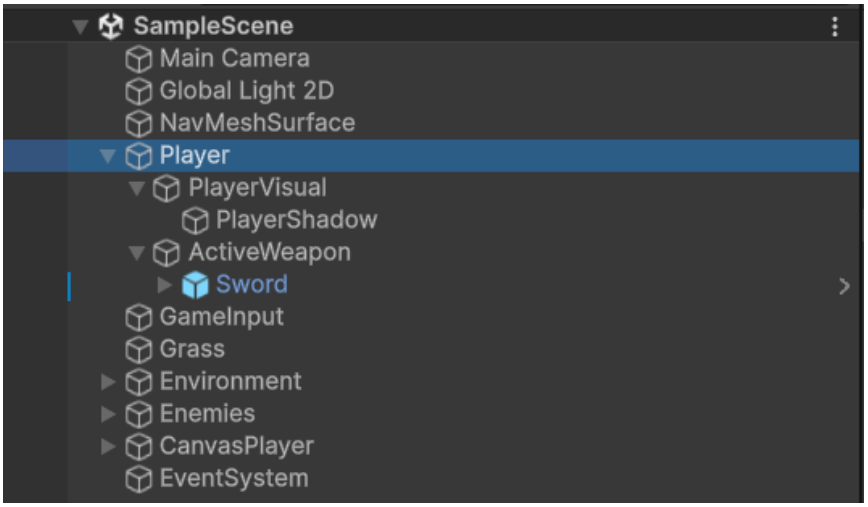

Рисунок 3.1 – Ієрархія проекту в Unity

Реалізація механік гравця була описана в трьох скриптах: Player, GameInput та PlayerKnock. Player – скрипт, який містить в собі основні механіки переміщення, налаштування камери, атаки, системи здоров’я та анімації гравця. GameInput – скрипт, який відповідає за зчитування даних, які вводить користувач через натиск клавіш та відповідає на певний метод скрипта Player. PlayerKnock – допоміжний скрипт до Player, в якому описана система відкидання гравця.

Переміщення гравця було реалізовано через компонент RigidBody2D та Transform. Перший компонент враховує фізику при переміщенні об’єкта по ігровому середовищі, а другий – ні. Вони є вбудованими в ігровому рушії Unity та їх було додано до об’єкту Player. Додатково, було добавлено

CapsuleCollider2D для створення тіла гравця. Наступним кроком, було написання методу переміщення гравця, яке описане в лістингу 3.1.

Лістинг 3.1 – Переміщення гравця

```
public class Player : MonoBehaviour
{
    public static Player Instance { get; private set; }
    [SerializeField] private float _movingSpeed = 5f;
    [SerializeField] private Collider2D _grassBounds;
    Vector2 inputVector;
    private Rigidbody2D rb;
    private float _minMovingSpeed = 0.1f;
    private bool _isRunning = false;

    private void Awake()
    {
        Instance = this;
        rb = GetComponent<Rigidbody2D>();
    }
    private void Update()
    {
        inputVector = GameInput.Instance.GetMovementVector();
    }
    private void FixedUpdate()
    {
        HandleMovement();
    }

    public bool IsRunning()
    {
        return _isRunning;
    }

    private void HandleMovement()
    {
        rb.MovePosition(rb.position + inputVector * (_movingSpeed *
Time.fixedDeltaTime));
        if (Mathf.Abs(inputVector.x) > _minMovingSpeed ||
Mathf.Abs(inputVector.y) > _minMovingSpeed)
        {
            _isRunning = true;
        }
        else
        {
            _isRunning = false;
        }
    }
```

кінець лістингу 3.1

В ігровому рушії Unity, описані розробником методи в скриптах, відображаються наступним чином (рис. 3.2).

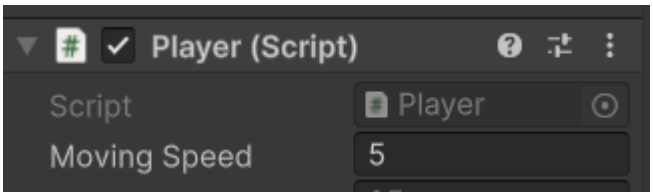

Рисунок 3.2 – Вигляд властивостей об’єкта Player в Unity, описаний розробником

Після перевірки роботи виконання скрипта, далі було створено просту бойову систему. Об’єкт Sword є зброєю головного героя та містить компонент PolygonCollider2D, який буде взаємодіяти з фізичними тілами інших ігрових об’єктів та впливати на них. Метод атаки був описаний в лістингу 3.2.

Лістинг 3.2 – Атака гравця

```
public class Sword : MonoBehaviour
{
    [SerializeField] private int _damageAmount = 2;
    public event EventHandler OnSwordSwing;
    private PolygonCollider2D _polygonCollider2D;

    private void Awake()
    {
        _polygonCollider2D = GetComponent<PolygonCollider2D>();
    }

    private void Start()
    {
        AttackColliderTurnOff();
    }

    public void Attack()
    {
        AttackColliderTurnOffOn();
        OnSwordSwing?.Invoke(this, EventArgs.Empty);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
```

```
        if (collision.transform.TryGetComponent(out EnemyLife
enemyLife))
        {
            enemyLife.TakeDamage(_damageAmount);
        }
    }

    public void AttackColliderTurnOff()
    {
        _polygonCollider2D.enabled = false;
    }

    private void AttackColliderTurnOn()
    {
        _polygonCollider2D.enabled = true;
    }

    private void AttackColliderTurnOffOn()
    {
        AttackColliderTurnOff();
        AttackColliderTurnOn();
    }
```

кінець лістингу 3.2

Наступним чином, було створено зв’язки анімацій персонажу та меча в компоненті Animator, перевірено правильність виконання скриптів. В результаті отримано готового гравця з переміщенням та атакою, який зображено на рисунку 3.3.

Рисунок 3.3 – Готовий вигляд персонажу

Додатково було створено систему регенерації здоров’я. Після отримання шкоди ворогами, персонаж через деякий проміжок часу почав відновлюватись. Нище наведений лістинг 3.3, в якому описана дана система.

Лістинг 3.3 – Система регенерації

```
public class Player : MonoBehaviour
{
    [Header("Health Regeneration")]
    [SerializeField] private float _healthRegenDelay = 5f;
    [SerializeField] private float _healthRegenRate = 1f;
    [SerializeField] private int _healthRegenAmount = 1;

    private int _currentHP;
    private bool _canTakeDamage;
    private bool _isAlive;
    public int GetCurrentHP() => _currentHP;
    public int GetMaxHP() => _maxHP;
    private float _timeSinceLastDamage;
    private bool _isRegenerating;

    private void Start()
    {
        _currentHP = _maxHP;
        _canTakeDamage = true;
        _isAlive = true;
        GameInput.Instance.onPlayerAttack +=
GameInput_onPlayerAttack;
        _timeSinceLastDamage = _healthRegenDelay;
    }

    private void Update()
    {
        inputVector = GameInput.Instance.GetMovementVector();
        HandleHealthRegeneration();
    }

    public bool IsAlive() => _isAlive;

    private void HandleHealthRegeneration()
    {
        if (_currentHP >= _maxHP) return;

        _timeSinceLastDamage += Time.deltaTime;

        if (_timeSinceLastDamage >= _healthRegenDelay &&
!_isRegenerating)
        {
            StartCoroutine(RegenerateHealth());
```

```
        }
    }

    private IEnumerator RegenerateHealth()
    {
        _isRegenerating = true;
        while (_currentHP < _maxHP && _timeSinceLastDamage >=
_healthRegenDelay)
        {
            _currentHP = Mathf.Min(_currentHP + _healthRegenAmount,
_maxHP);
            yield return new WaitForSeconds(_healthRegenRate);
        }
        _isRegenerating = false;
    }

}
```

кінець лістингу 3.3

Останнім, що було виконано, це система відштовхування для надання реалізму об’єкту. Після отримання шкоди, персонаж переміщався назад від напряму атаки NPC з певною силою. Нище наведено лістинг 3.4, в якому описана дана система.

Лістинг 3.4 – Система відштовхування

```
public class Player : MonoBehaviour
{
    private Rigidbody2D rb;
    private PlayerKnock _playerKnock;

    private int _currentHP;
    private bool _canTakeDamage;
    private bool _isAlive;
    public int GetCurrentHP() => _currentHP;

    private void Awake()
    {
        Instance = this;
        rb = GetComponent<Rigidbody2D>();
        _playerKnock = GetComponent<PlayerKnock>();
    }

    private void FixedUpdate()
    {
        if (_playerKnock.isGetKnockPlayer)
```

```
            return;
        HandleMovement();
    }

    public void TakeDamage(Transform damageSource, int damage)
    {
        if (_canTakeDamage && _isAlive)
        {
            _canTakeDamage = false;
            _currentHP = Mathf.Max(0, _currentHP -= damage);
            _timeSinceLastDamage = 0f;
            _playerKnock.GetKnock(damageSource);
            StartCoroutine(HPRecoveryRoutine());
        }
        DetectHP();
    }

    private void DetectHP()
    {
        if (_currentHP == 0 && _isAlive)
        {
            _isAlive = false;
            _playerKnock.StopKnockTime();
            GameInput.Instance.DisableMove();
            OnPlayerDeath?.Invoke(this, EventArgs.Empty);
        }
    }
}
```

кінець лістингу 3.4

3.2 Розробка ШІ для NPC

Для створення штучного інтелекту NPC, потрібно було створити скрипт, в якому здійснюється процес пошуку та взаємодії з гравцем, систему навігації та здійснення переміщення. Для системи навігації було додатково встановлено адаптований NavMesh для двовимірного простору та здійснено налаштування маршруту переміщення ворогів. Для проведення тесту системи навігації, було розміщено декоративні об’єкти на сцені з компонентом CapsuleCollider2D. Результат готової системи навігації зображено на рисунку 3.4, де синя зона – це доступний ігровий простір для переміщення NPC.

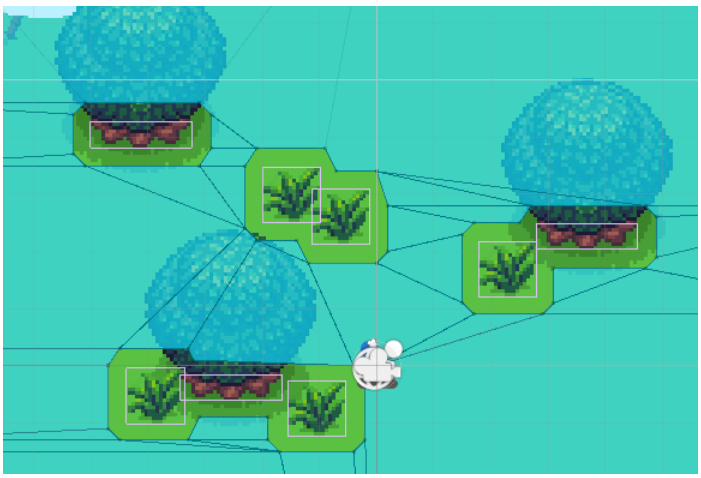

Рисунок 3.4 – Маршрут, прокладений NavMesh

NPC здійснюватиме патрулювання, переслідування по даному маршруту та не виходитиме за його межі. Наступним чином було створено події активності в класі EnemyAI, які описані в додатку А. При наближенні гравця до зони тригеру NPC, ігровий об’єкт починає виконувати відповідний скрипт та реагувати на дії, які прийматиме гравець. На рисунку 3.5 зображено один з результатів цих методів, а саме переслідування гравця – ігровий об’єкт ідентифікує гравця, якщо той попадає в зону тригеру та починає його переслідувати.

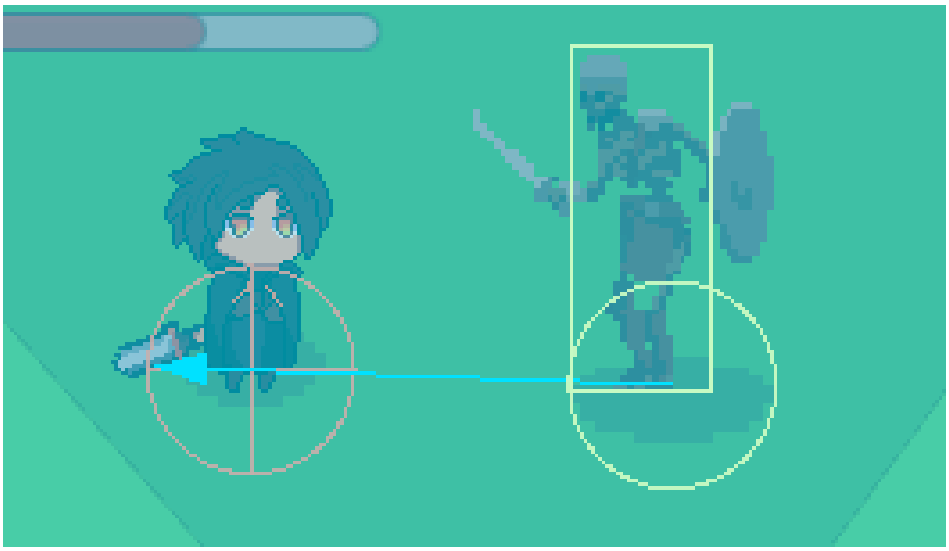

Рисунок 3.5 – Система переслідування NPC

3.3 Тестування та налагодження ігрових механік

Після розробки основної частини проекту, потрібно було виконати тестування. Для кваліфікаційної роботи було здійснено три типи тестів: юніт-тестування, інтеграційне тестування та тестування продуктивності.

Юніт-тестування полягала в перевірці окремих класів на правильність їх виконання. Для даного тестування, я обрав систему переміщення гравця. Після виконання тестування, я отримав такий результат (рис. 3.6).

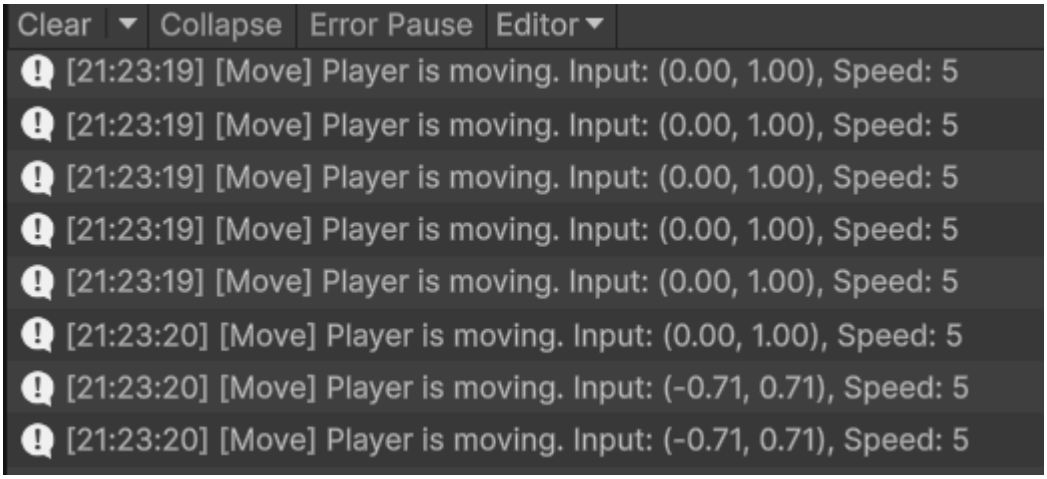

Рисунок 3.6 – Результати юніт-тестування

Якщо брати до уваги, що юніт-тестування стосується лише перевірки певного класу, то інтеграційне тестування здійснює перевірку одразу декількох систем. На цьому етапі можуть виникати проблеми різного характеру, такі як неправильна обробка методів або проблеми з передачею даних. Для їх вирішення застосовуються різні стратегії, зокрема нисхідна, висхідна та гібридна інтеграція, кожна з яких має свої переваги в залежності від архітектури системи. Для проведення інтеграційного тестування було обрано три системи: нанесення шкоди, регенерація та відштовхування. Результати тестування зображені на рисунку 3.7.

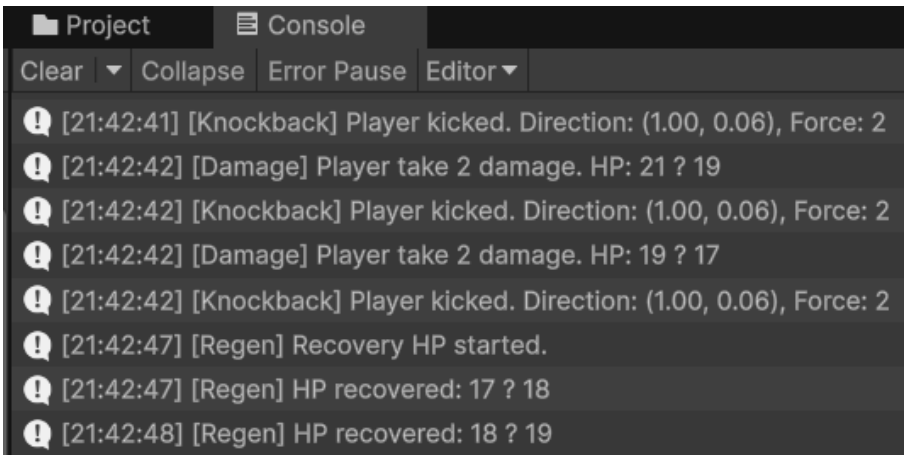

Рисунок 3.7 – Результати інтеграційного тестування

Останнім етапом тестування проекту було проведення тесту продуктивності, мета якого полягає у визначенні відповідності вимог системи щодо швидкодії та стабільності. На відмінну від попередніх тестів, де перевірялись функціональні вимоги, тут аналізуються такі параметри як час відгуку, пропускна здатність та ресурсоспоживання. В ігровому рушії Unity є вбудована статистика продуктивності (рис. 3.8) під час перебуванні в режимі PlayMode, в якій відображаються лише витрати ресурсів та пропускна здатність.

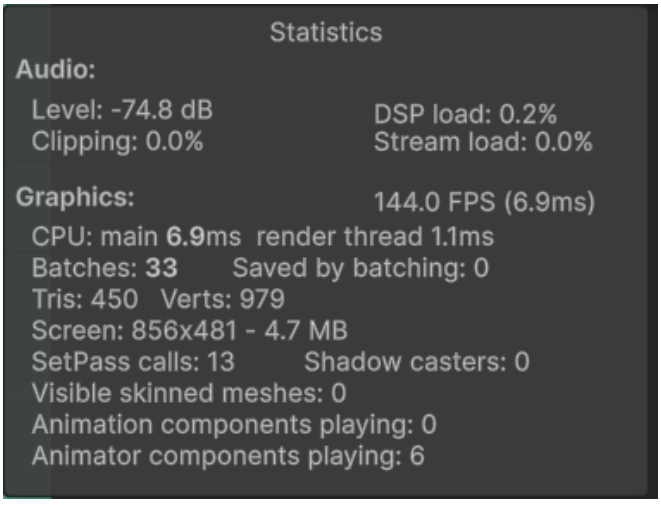

Рисунок 3.8 – Статистика продуктивності під час гри

Для додаткової перевірки продуктивності, було створено об’єкт PerfomanceMonitor та скрипт PerfomanceMonitor.cs. В скрипті було описано метод перевірки продуктивності через debug.log, а результати тестування

виводяться в консоль ігрового рушія Unity. Окрім пропускної здатності, було вирішено описати швидкість запуску ігрової системи, звіти пропускної здатності інтервалом кожні 5 секунд та швидкість запуску методів під час виконання події. Нище наведено лістинг 3.5, в якому описаний тест продуктивності.

Лістинг 3.5 – Метод тесту продуктивності

```
private void Start()
{
  float startTime = Time.realtimeSinceStartup;
  Debug.Log($"[Startup] Scene generated {startTime:F2} s after
launch game");
}
public class PerformanceMonitor : MonoBehaviour
{
    private float deltaTime = 0.0f;
    private float timePassed = 0.0f;
    private int frameCount = 0;
    private float lagThreshold = 0.05f; // 50 мс

    void Update()
    {
        deltaTime += (Time.unscaledDeltaTime - deltaTime) * 0.1f;
        frameCount++;
        timePassed += Time.unscaledDeltaTime;

        // Лаг-фіксація
        if (Time.unscaledDeltaTime > lagThreshold)
        {
            Debug.Log($"[Lag Spike] Time.deltaTime =
{Time.unscaledDeltaTime * 1000f:F2} мс");
        }

        // Кожні 5 секунд — звіт
        if (timePassed >= 5f)
        {
            float fps = frameCount / timePassed;
            float avgFrameTime = (timePassed / frameCount) * 1000f;
            Debug.Log($"[Perf Summary] FPS: {fps:F2}, Avg Frame
Time: {avgFrameTime:F2} мс");
            // обнулення
            frameCount = 0;
            timePassed = 0f;
        }
    }
}
```

кінець лістингу 3.5

Даний скрипт було потрібно прикріпити до пустого об’єкту PerfomanceMonitor та виконати запуск проекту в режимі PlayMode. Результати тестування зображені на рисунку 3.9.

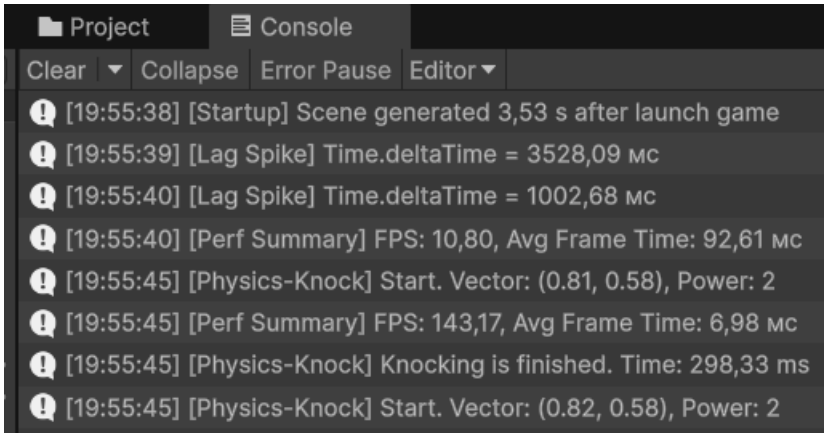

Рисунок 3.9 – Результати проведення тесту продуктивності

Під час проведення тестів, не було виявлено некоректної роботи класів, багів або зависань. Наступним кроком було визначення мінімальних системних вимог для даного ігрового застосунку. На основі результатів даних тестів, було сформовано наступні вимоги, які описані в таблиці 3.1.

Таблиця 3.1 – Мінімальні системні вимоги

Компонент	Характеристика
Операційна система	Windows 10
Процесор	Intel Core i3
Оперативна пам’ять	4 GB
Відеокарта	Intel UHD Graphics 4000+ або GTX 750
Диск	125 MB

3.4 Створення графічного інтерфейсу та побудова ігрового рівня

Після створення, налаштування компонентів гри та її проведення тестування, наступним кроком було створення графічного інтерфейсу. Потрібно було виконати дизайн головного меню, меню паузи та екрану після смерті гравця. Весь UI дизайн елементів меню відбувався в онлайн-редакторі Figma.

Figma [19] – векторний графічний редактор, орієнтований на розробку UI/UX дизайну. На відміну від традиційних графічних редакторів, Figma не потребує прямого встановлення, що усуває необхідність локальної інсталяції та забезпечує доступ до проектів з будь-якого пристрою.

Фон для головного меню (рис. 3.10) було згенеровано за допомогою нейромережі ChatGPT таким чином, щоб він відповідав тематиці гри та був відповідно стилізований. Створені кнопки є на четверть прозорі, містять згладжені кути та відповідно стилізовані: основний колір оранжевий, колір тексту є білим.

Рисунок 3.10 – Вигляд головного меню

Для реалізації головного меню було потрібно створити нову сцену з відповідними UI елементами, які пропонує Unity та написати для них метод переходу в основну сцену гри. Після натиску кнопок в меню, відбувались відповідні методи: перехід до гри або вихід із застосунку. У лістингу 3.6 описаний цей метод.

Лістинг 3.6 – Реалізація головного меню

```
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
```

```
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
1);
    }

    public void ExitGame()
    {
        Application.Quit();
    }
}
```

кінець лістингу 3.6

Після проведення тестування, наступним чином в основній сцені було виконано дизайн меню паузи (рис. 3.11). В інтерфейсі основної сцени було створено панель з відповідними кнопками: пауза, продовження та вихід. Вигляд кнопок виконано в такому ж стилі, як і в головному меню.

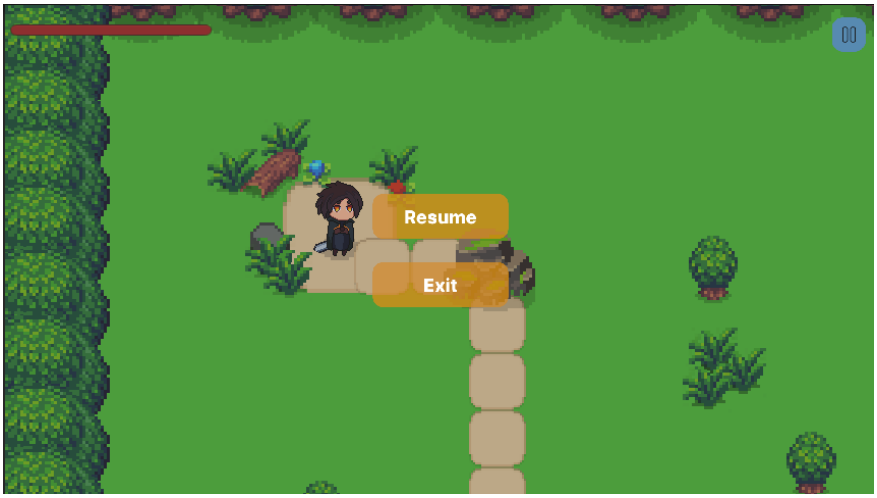

Рисунок 3.11 – Вигляд меню паузи

Останній елемент інтерфейс меню екран смерті було виконано за таким же принципом, як і меню паузи. Після налаштування UI гри, потрібно було створити метод реалізації меню, який описаний в лістингу 3.7.

Лістинг 3.7 – Метод реалізації меню паузи

```
public class PauseMenu : MonoBehaviour
{
    public bool PauseGame;
```

```
    public GameObject pauseGameMenu;

    void Update()
    {

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (PauseGame)
            {
                Resume();
            }
            else
            {
                Pause();
            }

        }
    }

    public void Resume()
    {
        pauseGameMenu.SetActive(false);
        Time.timeScale = 1f;
        PauseGame = false;
    }

    public void Pause()
    {
        pauseGameMenu.SetActive(true);
        Time.timeScale = 0f;
        PauseGame = true;
    }

    public void LoadMenu()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene("Main Menu");
    }
}
```

кінець лістингу 3.7

Наступним чином, після створення інтерфейсу, в ігровому рушії потрібно було використати інструмент Tilemap для генерації рівня. Tilemap – це один з інструментів Unity, який використовується для побудови рівнів на основі ігрових елементів. Разом з цим інструментом виступає сітка (Grid), що визначає

розміщення об’єктів у просторі. Кожен елемент карти може містити не лише графічне зображення, а й додаткові властивості, такі як колізії, анімації тощо.

Перед початком роботи з ним, потрібно створити палітру та помістити в нього спрайти. Додатково, потрібно налаштувати їхні розміщення в інструменті для його коректної роботи. Кінцевим результатом використання даного інструменту зображено на рисунку 3.12.

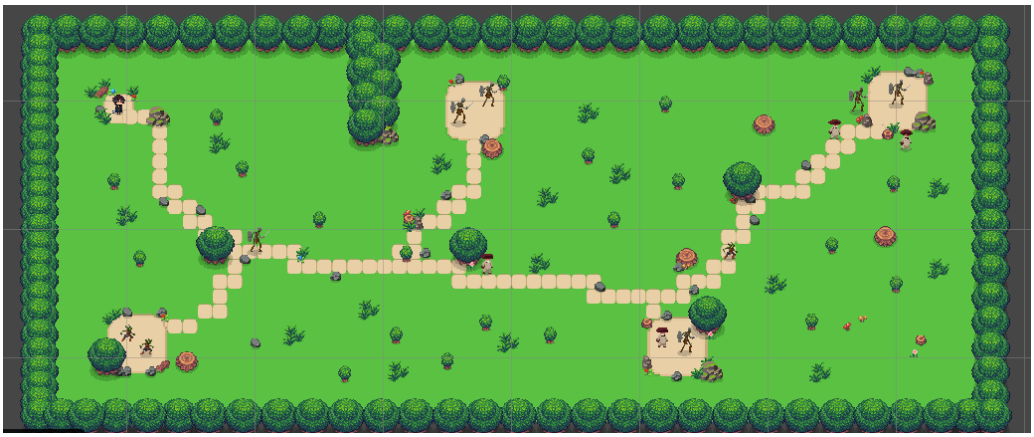

Рисунок 3.12 – Згенерований ігровий рівень за допомогою Tilemap

3.5 Розробка інсталяційного пакета

Після розробки гри, потрібно було виконати її компіляцію. Для цього в ігровому рушії Unity потрібно виконати налаштування збірки. У вікні Build Profiles створити налаштування «Player Settings», а саме: розширення застосунку та режим відображення. На рисунку 3.13 зображено вікно компіляції проекту.

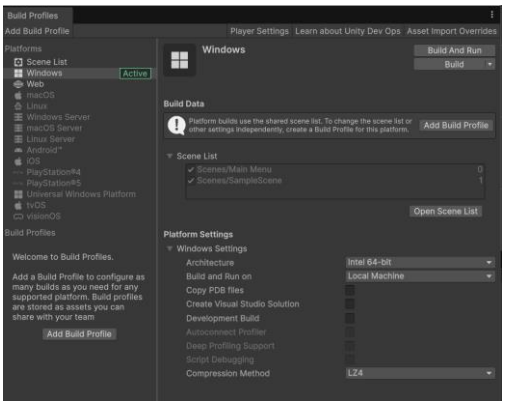

Рисунок 3.13 – Вікно компіляції збірки гри

Після натиску кнопки «Build» відбудеться процес збірки гри. Результатом буде вже готовий ігровий застосунок. Наступним кроком, потрібно було створити інсталятор гри. Для виконання цього завдання, було вирішено використати програму Inno Setup.

Inno Setup [20] – це програмне забезпечення з відкритим кодом, яке спеціалізується на створенні інсталяторів для програм. Класичний інтерфейс програми є не дуже зручним для звичайного користувача, проте він має багато функціональних особливостей: підтримка сучасних версій Windows та дозволяє створити 64-бітний тип програми. Окрім того, він може створити одиночний .exe файл інсталяції, що спрощує встановлення програми та її поширення. Це є важливо для реалізації одного з завдань кваліфікаційної роботи.

Після встановлення Inno Setup, в інтерфейсі програми потрібно виконати наступні дії: відкрити пункт «File» та заповнити форму інсталятора. На рисунку 3.14 зображено процес створення інсталятора.

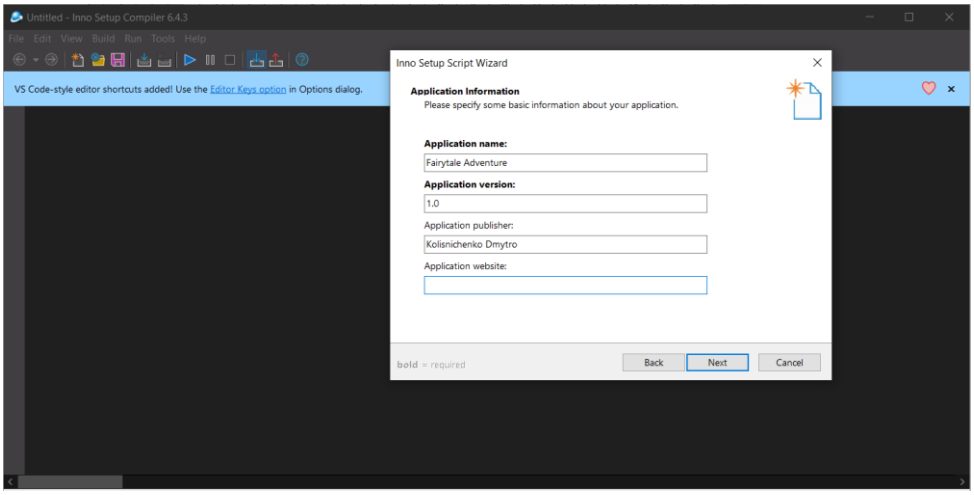

Рисунок 3.14 – Вигляд Inno Setup під час розробки інсталятора

Після заповнення форми, у вікні програми відобразиться скрипт для створення інсталятора. На рисунку 3.15 зображено вигляд програми після заповнення форми.

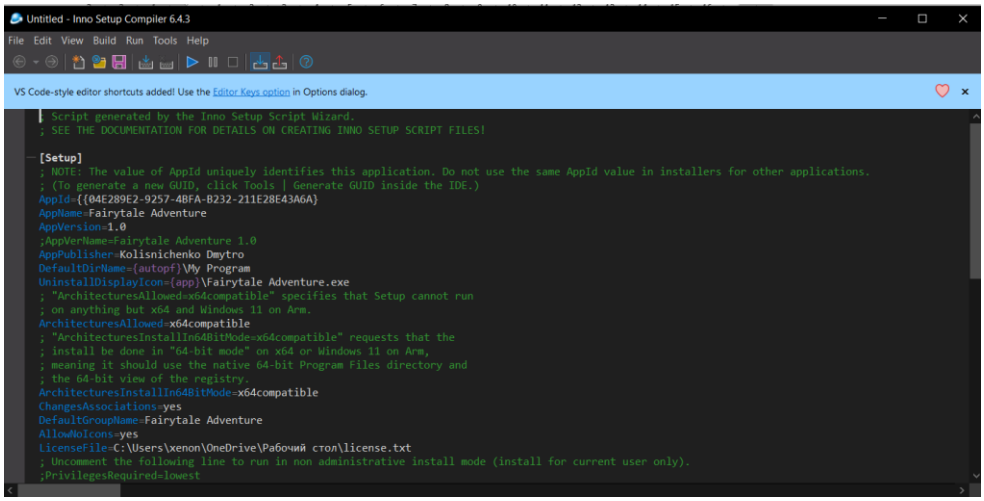

Рисунок 3.15 – Вигляд Inno Setup для створення інсталятора

Запускаємо виконання скрипта і очікуємо певний проміжок часу, доки програма не завершить свою роботу. У результаті отримуємо готовий інсталятор ігрового застосунку.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи бакалавра, було досягнуто наступних цілей:

1) проведено аналіз сучасного стану ігрової індустрії, зокрема жанру 2D відеоігор, що дозволило визначити ключові тенденції та технологічні рішення для виконання завдань кваліфікаційної роботи. Порівняльний аналіз таких ігор, як The Binding of Isaac, Terraria та Stardew Valley виявив їхні особливості, що стало основою для формування власного підходу до розробки;

2) на основі аналізу сформовано функціональні та нефункціональні вимоги до гри, що включають різні ігрові системи: базові системи персонажу, взаємодія з NPC тощо. Для візуалізації структури проекту створено UML-діаграми станів гравця та NPC, а також прототип майбутнього інтерфейсу, що сприяло чіткій організації виконання роботи. Під час проектування було використано програму Figma та веб-платформу draw.io;

3) порівняльний аналіз ігрових рушіїв (Unity, Unreal Engine, Godot, GameMaker, CryEngine) та середовищ розробки (Visual Studio, JetBrains Rider, MonoDevelop) дозволив обґрунтувати вибір Unity та Visual Studio як оптимальних інструментів для реалізації проекту, враховуючи їхню гнучкість, кросплатформеність та інтеграційні можливості;

4) розроблено базові механіки гри такі як переміщення гравця, система атаки, регенерація здоров’я та відштовхування. Використання компонентів RigidBody2D, Collider2D та анімаційних систем забезпечило плавну взаємодію об’єктів у ігровому середовищі. Кожна механіка протестована на стабільність та відповідність поставленим вимогам;

5) на основі адаптованого NavMesh створено систему навігації для NPC, яка включає стани патрулювання, переслідування та атаки. Реалізовано динамічну реакцію NPC на дії гравця, що забезпечує реалістичну поведінку ворогів у ігровому світі;

6) проведено юніт-тестування окремих класів, інтеграційне тестування взаємодії систем та тестування продуктивності. Результати підтвердили стабільність роботи гри на слабких системах, а також відсутність критичних помилок. Визначено мінімальні системні вимоги для комфортного використання продукту;

7) за допомогою Figma розроблено інтуїтивно зрозумілий інтерфейс, що включає головне меню, меню паузи та екран смерті. Використання Tilemap дозволило ефективно побудувати ігровий рівень;

8) завершальним етапом розробки стало створення інсталяційного пакету за допомогою Inno Setup, що дозволило упакувати проект у зручний для користувача формат;

При аналізі предметної області було досліджено особливості жанру подібних відеоігор та окремі механіки, які притаманні даному жанру: вид зверху, відкритий ігровий світ та взаємодія з ігровими об’єктами.

При аналізів алгоритмів штучного інтелекту було досліджено метод NavMesh. Для розробки штучного інтелекту було використано адаптований до двовимірного простору NavMesh та створено власні сценарії поведінки NPC.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. El Output. URL: https://uk.eloutput.com/гра/звітів/Інді-ігри/ (дата звернення: 15.03.2025);

2. Gamedev.dou.ua. URL: https://gamedev.dou.ua/articles/lessons-from-failed-indie-games/ (дата звернення: 15.03.2025);

3. FoxmindEd. URL: https://foxminded.ua/rozrobka-ihor-unity-unreal-engine/ (дата звернення: 15.03.2025);

4. Stardew Valley. URL: https://uk.wikipedia.org/wiki/Stardew_Valley (дата звернення: 17.03.2025);

5. Terraria. URL: https://uk.wikipedia.org/wiki/Terraria (дата звернення: 17.03.2025);

6. Binding of Isaac. URL: https://uk.wikipedia.org/wiki/The_Binding_of_Isaac (дата звернення: 17.03.2025);

7. Unity Development Platform. URL: https://unity.com (дата звернення: 03.04.2025);

8. Unity. Understanding URP, HDRP, and Built-In Render Pipeline. Wayline. URL: https://www.wayline.io/blog/unity-understanding-urp-hdrp-built-in (дата звернення: 03.04.2025);

9. Unreal Engine. URL: https://www.unrealengine.com/en-US (дата звернення: 05.04.2025);

10. Unreal Engine Documentation. Lumen Technical Details. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-technical-details-in-unreal-engine (дата звернення: 05.04.2025);

11. Unreal Engine Documentation. Nanite Virtualized Geometry. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine (дата звернення 05.04.2025);

12. Godot Engine – Free and open-source game engine. Godot Engine. URL: https://godotengine.org (дата звернення: 12.04.2025);

13. GDScript. URL: https://gdscript.com (дата звернення: 12.04.2025);

14. CryEngine. The complete solution for next generation game development by Crytek. URL: https://www.cryengine.com (дата звернення: 15.04.2025);

15. GameMaker. Ultimate 2D Game Engine. URL: https://gamemaker.io/en (дата звернення: 16.04.2025);

16. Visual Studio. IDE and Code Editor for Software Developers and Teams. URL: https://visualstudio.microsoft.com (дата звернення: 20.04.2025);

17. Rider. The Cross-Platform .NET IDE from JetBrains. URL: https://www.jetbrains.com/rider/ (дата звернення: 21.04.2025);

18. MonoDevelop. URL: https://www.monodevelop.com (дата звернення: 21.04.2025);

19. Figma. The Interface Design Tool. URL: https://www.figma.com/ (дата звернення: 15.05.2025);

20. Inno Setup. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Inno_Setup (дата звернення: 15.05.2025).

ДОДАТКИ

Додаток А – методи активності поведінки NPC

```
using UnityEngine;
using UnityEngine.AI;
using FairytaleAdventure.Utils;
using System;

public class EnemyAI : MonoBehaviour
{
    [SerializeField] private State _startingState;
    [SerializeField] private float _roamingDistanceMax = 7f;
    [SerializeField] private float _roamingDistanceMin = 3f;
    [SerializeField] private float _roamingTimeMax = 2f;

    [SerializeField] private bool _isChasingEnemy = false;
    [SerializeField] private float _chasingDistance = 4f;
    [SerializeField] private float _chasingSpeedMultiplier = 2f;

    [SerializeField] private bool _isAttackingEnemy = false;
    [SerializeField] private float _attackingDistance = 2f;
    [SerializeField] private float _attackRate = 2f;
    private float _nextAttackTime = 0f;

    private NavMeshAgent _navMeshAgent;
    private State _currentState;
    private float _roamingTimer;
    private Vector3 _roamPosition;
    private Vector3 _startingPosition;

    private float _roamingSpeed;
    private float _chasingSpeed;

    private float _nextCheckDirectionTime = 0f;
    private float _checkDirectionDuration = 0.1f;
    private Vector3 _lastPosition;

    public event EventHandler OnEnemyAttack;

    public bool isRunning
    {
        get
        {
            if (_navMeshAgent.velocity == Vector3.zero)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }

    private enum State
    {
        Idle,
        Roaming,
        Chasing,
        Attacking,
        Death
    }

    private void Awake()
    {
        _navMeshAgent = GetComponent<NavMeshAgent>();
```

```
        _navMeshAgent.updateRotation = false;
        _navMeshAgent.updateUpAxis = false;
        _currentState = _startingState;
        _roamingSpeed = _navMeshAgent.speed;
        _chasingSpeed = _navMeshAgent.speed * _chasingSpeedMultiplier;
    }

    private void Update()
    {
        StateHander();
        MoveDirectionHandler();
    }

    public void SetDeathState()
    {
        _navMeshAgent.ResetPath();
        _currentState = State.Death;
    }

    private void StateHander()
    {
        switch (_currentState)
        {
            case State.Roaming:
                _roamingTimer -= Time.deltaTime;
                if (_roamingTimer < 0)
                {
                    Roaming();
                    _roamingTimer = _roamingTimeMax;
                }
                CheckCurrentState();
                break;
            case State.Chasing:
                ChasingTarget();
                CheckCurrentState();
                break;
            case State.Attacking:
                AttackingTarget();
                CheckCurrentState();
                break;
            case State.Death:
                break;
            default:
            case State.Idle:
                break;
        }
    }

    private void ChasingTarget()
    {
        _navMeshAgent.SetDestination(Player.Instance.transform.position);
    }

    public float GetRoamingAnimationSpeed()
    {
        return _navMeshAgent.speed / _roamingSpeed;
    }

    private void CheckCurrentState()
    {
        float distanceToPlayer = Vector3.Distance(transform.position,
Player.Instance.transform.position);
        State newState = State.Roaming;

        if (!Player.Instance.IsAlive())
```

```
        {
            _currentState = State.Idle;
            return;
        }

        if (_isChasingEnemy)
        {
            if (distanceToPlayer <= _chasingDistance)
            {
                newState = State.Chasing;
            }
        }

        if (_isAttackingEnemy)
        {
            if (distanceToPlayer <= _attackingDistance)
            {
                newState = State.Attacking;
            }
        }

        if (newState != _currentState)
        {
            if (newState == State.Chasing) {
                _navMeshAgent.ResetPath();
                _navMeshAgent.speed = _chasingSpeed;
            } else if (newState == State.Roaming)
            {
                _roamingTimer = 0f;
                _navMeshAgent.speed = _roamingSpeed;
            } else if (newState == State.Attacking)
            {
                _navMeshAgent.ResetPath();
            }
                _currentState = newState;
        }
    }

    private void AttackingTarget()
    {
        if (Time.time > _nextAttackTime)
        {
            OnEnemyAttack.Invoke(this, EventArgs.Empty);
            _nextAttackTime = Time.time + _attackRate;
        }
    }

    private void MoveDirectionHandler()
    {
        if (Time.time > _nextCheckDirectionTime)
        {
            if (isRunning)
            {
                ChangeFacingDirection(_lastPosition, transform.position);
            } else if (_currentState == State.Attacking)
            {
                ChangeFacingDirection(transform.position,
Player.Instance.transform.position);
            }

            _lastPosition = transform.position;
            _nextCheckDirectionTime = Time.time + _checkDirectionDuration;
        }
    }
```

```
    private void Roaming()
    {
        _startingPosition = transform.position;
        _roamPosition = GetRoamingPosition();
        _navMeshAgent.SetDestination(_roamPosition);
    }

    private Vector3 GetRoamingPosition()
    {
        return _startingPosition + Utils.GetRandomDir() *
UnityEngine.Random.Range(_roamingDistanceMin, _roamingDistanceMax);
    }

    private void ChangeFacingDirection(Vector3 sourcePosition, Vector3 targetPosition)
    {
        if (sourcePosition.x > targetPosition.x)
        {
            transform.rotation = Quaternion.Euler(0, -180, 0);
        }
        else
        {
            transform.rotation = Quaternion.Euler(0, 0, 0);
        }
    }
}
```