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 ДВОВИМІРНОЇ ГРИ «FRAGMENTS OF THE WORLD» В ЖАНРІ ADVENTURE-RPG З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF THE 2D GAME «FRAGMENTS OF THE WORLD» IN THE ADVENTURE-RPG GENRE USING UNITY

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

Виконав: здобувач вищої освіти
групи ІПЗ-41
Штик Олександр Вікторович

Керівник:
Падалко Анатолій Михайлович

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
_____
«10» 12 2024р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Штику Олександру Вікторовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка двовимірної гри «Fragments of the World» в жанрі advanture-RPG з використанням Unity»*
Керівник роботи: *к.ф.-м.н., доцент Падалко А. 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: *Unity, Visual Studio, методичні вказівки до виконання кваліфікаційної роботи бакалавра.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз предметної області, постановка завдання, визначення вимог, реалізація інтерфейсу, логіка, тестування та виправлення помилок.*

5. Перелік графічного матеріалу: *8 рисунків*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Падалко А. М.*		
Специфікація вимог до розробленої системи	*Падалко А. М.*		
Розробка об'єкта проектування	*Падалко А. М.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	0,4 %		
Академічна доброчесність	*Падалко А. 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	Обгрунтування 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 _____ Олександр ШТИК

Керівник кваліфікаційної роботи _____ Анатолій ПАДАЛКО

АНОТАЦІЯ

Штик О. В. Розробка двовимірної гри «Fragments of the World» в жанрі advanture-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 та списку використаних джерел.

У першому розділі здійснено аналіз предметної області, визначено актуальність теми та сформульовано завдання кваліфікаційної роботи. У другому розділі розглянуто вимоги до програмного забезпечення та спроектовано архітектуру майбутньої гри. У третьому розділі описано процес реалізації ігрового застосунку: розробку механік, інтерфейсу, навігації, анімації та тестування. У висновках наведено результати роботи, оцінено ефективність реалізації та перспективи подальшого розвитку проекту.

Ключові слова: Unity, C#, Visual Studio, спрайти, скрипти, 2D-гра, ігровий рушій, програмна архітектура, геймдизайн, анімації.

ABSTRACT

Shtyk O. Development of the 2D "Game Fragments of the World" in the Adventure-RPG Genre Using Unity. Manuscript. Bachelor's qualification thesis of the educational program "Software Engineering" specialty "Software Engineering" Lutsk National Technical University. Lutsk, 2025.

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

The first chapter presents an analysis of the subject area, outlines the relevance of the topic, and formulates the objectives of the thesis. The second chapter discusses the software requirements and presents the architecture design of the future game. The third chapter describes the implementation process of the game application, including the development of gameplay mechanics, user interface, navigation, animation, and testing. The conclusions summarize the results of the work, evaluate the implementation effectiveness, and outline the prospects for further development of the project.

Keywords: Unity, C#, Visual Studio, sprites, scripts, 2D game, game engine, software architecture, game design, animations.

ЗМІСТ

ВСТУП

Актуальність теми. Розробка 2D-ігор залишається актуальною темою у сучасній ІТ-галузі, адже цей напрям активно розвивається як серед професійних студій, так і в середовищі незалежних розробників. Ігровий рушій Unity є одним із найпопулярніших засобів для створення 2D-ігор завдяки своїй простоті, широкому набору інструментів, підтримці різних платформ і великій спільноті користувачів. Проектування гри дозволяє поєднати технічні знання з творчим підходом, охоплюючи програмування, анімацію, дизайн, логіку поведінки об’єктів та взаємодію з гравцем.

Метою даної кваліфікаційної роботи є створення повноцінної 2D-рольової гри «Fragments of the World» із використанням Unity та мови програмування C#. Гра може використовуватися не лише як навчальний приклад, але й як засіб емоційного розвантаження, зокрема для дітей-переселенців, яким подібні інтерактивні проекти можуть допомогти адаптуватися до нових умов через захопливу гру.

Об’єктом кваліфікаційної роботи є процес створення програмного забезпечення для інтерактивної 2D-гри.

Предметом кваліфікаційної роботи є архітектура, логіка та методи реалізації функціоналу гри в середовищі Unity.

Для досягнення поставленої мети були визначені такі завдання:

– аналіз предметної області та актуальних підходів у розробці 2D-ігор, зокрема в жанрі Adventure-RPG. Проведено огляд сучасних ігрових механік, систем управління, структур сцени, поведінки NPC та анімації. Це дозволило виділити ключові компоненти, які необхідно реалізувати в проекті;

– формування функціональних і нефункціональних вимог до гри, включаючи вимоги до продуктивності, адаптивності інтерфейсу, структури рівнів, інтерактивності персонажів, логіки бою та навігації між сценами. Особливу увагу приділено забезпеченню зручності гри для користувачів та можливості масштабування;

– вибір інструментів та технологій, необхідних для реалізації гри. Основними стали Unity (для побудови сцени, анімацій, фізики та UI), мова програмування C#, редактор Visual Studio, а також допоміжні інструменти – Tilemap, Animator, NavMesh та Canvas;

– реалізація управління ігровим персонажем та забезпечення його взаємодії з об’єктами у сцені, включно з пересуванням, орієнтацією, виявленням зіткнень і активацією тригерів. Створення базової бойової системи, яка включає атаки, отримання шкоди, ефект нокауту, відкидання та обробку смерті персонажа;

– розробка логіки ворогів, що передбачає декілька станів: патрулювання, переслідування гравця, атакуюча поведінка, відслідковування дистанції, а також анімацію їхніх дій;

– побудова локацій гри з використанням Tilemap і декількох палет: для трави, землі, води, стін і декоративних елементів. Створення стартової зони, обмежень переміщення та об’єктів оточення;

– створення системи сцен, включаючи головне меню, перехід між рівнями та забезпечення плавного завантаження сцен;

– проведення тестування та налагодження коду на всіх етапах розробки. Перевірка стабільності поведінки ігрових об’єктів, логіки бою, інтерфейсу, переходів між сценами та продуктивності. Застосовано ручне тестування, візуальний дебаг, сценарний аналіз і перевірку крайніх випадків.

РОЗДІЛ 1
АНАЛІЗ СЧАСНИХ РІШЕНЬ У СФЕРІ РОЗРОБКИ ІГОР ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Галузь відеоігор постійно розвивається та модернізується, охоплюючи все ширше коло користувачів і платформ. Сьогодні на ринку домінують 3D-ігри з високим рівнем графіки, реалістичною фізикою та деталізованими світами. Проте попри технічний прогрес, 2D-ігри не втрачають популярності. Навпаки, вони залишаються актуальними завдяки своїй простоті, стилізованому дизайну та гнучкості реалізації.

3D-ігри вимагають значно більших ресурсів як на етапі розробки, так і під час виконання. Створення тривимірних моделей, анімацій, системи колізій, освітлення та рендерингу потребує не лише високої кваліфікації, а й відповідного технічного забезпечення. Це обумовлює складність розробки для невеликих команд або індивідуальних розробників.

2D-ігри, у свою чергу, є більш доступними для реалізації та дозволяють зосередитися на ігровій механіці, сюжеті та художньому стилі. Ігри на кшталт «Hollow Knight», «Celeste», «Dead Cells» доводять, що 2D-геймплей може бути не менш глибоким, ніж у 3D-проектах.

Unity, як один із найпопулярніших ігрових рушіїв, забезпечує широкий спектр інструментів для розробки як 2D, так і 3D-ігор. Вбудовані компоненти для обробки фізики, анімацій, керування колізіями, а також підтримка C# як основної мови програмування роблять його придатним для навчання та комерційної розробки.

Проблеми, що часто виникають при створенні 2D-ігор, включають: забезпечення коректної взаємодії між об’єктами, оптимізацію продуктивності, побудову інтуїтивно зрозумілого інтерфейсу користувача, створення якісної логіки ігрового процесу та механізмів збереження.

У контексті дипломної роботи особливу увагу приділено аналізу технологій, що дозволяють ефективно реалізовувати 2D-гру з сучасною архітектурою, управлінням станами та взаємодією з користувачем.

Індустрія відеоігор є однією з найбільш динамічно розвинутих галузей цифрових технологій. Протягом останнього десятиліття спостерігається значне зростання інтересу до 2D-ігор, зокрема серед інді-розробників. Це обумовлено відносною простотою створення, нижчими технічними вимогами до пристроїв користувача та можливістю зосередитись на геймплейній механіці й візуальному стилі, а не на складній тривимірній графіці.

Багато популярних 2D-ігор, таких як «Hollow Knight», «Celeste», «Dead Cells», доводять, що правильно реалізовані 2D-проекти можуть бути комерційно успішними та викликати інтерес великої аудиторії. Успіх таких ігор зумовлює інтерес до створення авторських 2D-ігор з унікальним сюжетом, стилем і механікою.

Середовище Unity є одним із найбільш популярних рушіїв для розробки 2D-ігор [1]. Завдяки своїй безкоштовній ліцензії для невеликих студій і широким можливостям воно активно використовується як початківцями, так і професійними командами. Платформа підтримує інтеграцію Visual Studio з мовою C# [2], що забезпечує розширюваність, гнучкість і можливість використання готових бібліотек та фреймворків.

Індустрія 2D-ігор залишається актуальною навіть у час стрімкого розвитку 3D-графіки та віртуальної реальності. Простота створення, менші вимоги до апаратного забезпечення, легкість стилізації та висока художня виразність – усе це сприяє зростанню популярності інді-проектів саме у 2D-форматі. На платформах, таких як Steam, Itch.io [3], Google Play та App Store, щоденно з’являються десятки нових 2D-ігор, більшість з яких розробляються малими командами або навіть однією особою.

У цьому контексті важливо зазначити, що створення навіть простого 2D-проекту потребує знання кількох дисциплін: геймдизайну, програмування, анімації, композиції, користувацького досвіду, роботи з ресурсами та

тестування. Вибір рушія Unity є цілком виправданим – він забезпечує готове середовище для створення 2D-проектів, має гнучку систему компонентів, підтримку Tilemap, Animator, системи частинок, UI, NavMesh та багатьох інших інструментів, що значно спрощують реалізацію повноцінної гри.

Наявність інструментів розвитку відіграє ще одну роль у просуванні 2D ігор. Unity пропонує безкоштовну версію з усіма інструментами, необхідними для створення повного продукту. Таким чином, студенти, початківці чи інді-команди можуть реалізувати власні творчі ідеї без будь-яких великих фінансових витрат. Створюючи 2D прототип, ви можете швидше побачити результати і фактично перевірити свої ідеї.

Варто також зазначити, що 2D ігри не багато покладаються на високі технічні можливості пристрою. Це дозволяє таким іграм починати на слабких комп'ютерах та мобільних телефонах, що охоплюють більше глядачів. Завдяки цьому, багато класичних механізмів, таких як платормери, лиходій або захист вежі, розвивалися в двовимірному просторі.

Місцеві проблеми у створенні 2D-ігор включають оптимізацію продуктивності, налаштування фізики та конкуренції та забезпечення стабільної роботи на різних платформах. Необхідна спеціальна обережність при розробці ігрової архітектури, яка дозволяє масштабувати свій проект без коду без суттєвих ускладнень. Крім того, розробникам потрібно звертати увагу на інтерактивність, дизайн інтерфейсу користувача, логіку подій гравця та підтримку прогресу.

Тому дослідження існуючих 2D-ігор та впровадження власних програмних продуктів – це перспективне поле, яке є актуальним та актуальним для створення ступеня бакалавра з інженерії програмного забезпечення.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

У межах даної кваліфікаційної роботи було поставлено завдання створити повноцінну комп'ютерну 2D-гру в жанрі Adventure-RPG під назвою «Fragments

of the World». Реалізація гри здійснювалась у середовищі Unity із застосуванням мови програмування C#. Основною метою було розробити проект, який поєднує дослідження світу, бойову систему, збір предметів та взаємодію з навколишнім середовищем.

Мета роботи зумовила постановку таких завдань:

– аналіз предметної області та актуальних підходів у розробці 2D-ігор, зокрема в жанрі Adventure-RPG. Проведено огляд сучасних ігрових механік, систем управління, структур сцени, поведінки NPC та анімації. Це дозволило виділити ключові компоненти, які необхідно реалізувати в проекті;

– формування функціональних і нефункціональних вимог до гри, включаючи вимоги до продуктивності, адаптивності інтерфейсу, структури рівнів, інтерактивності персонажів, логіки бою та навігації між сценами. Особливу увагу приділено забезпеченню зручності гри для користувачів та можливості масштабування;

– вибір інструментів та технологій, необхідних для реалізації гри. Основними стали Unity (для побудови сцени, анімацій, фізики та UI), мова програмування C#, редактор Visual Studio, а також допоміжні інструменти – Tilemap, Animator, NavMesh та Canvas;

– реалізація управління ігровим персонажем та забезпечення його взаємодії з об’єктами у сцені, включно з пересуванням, орієнтацією, виявленням зіткнень і активацією тригерів. Створення базової бойової системи, яка включає атаки, отримання шкоди, ефект нокауту, відкидання та обробку смерті персонажа;

– розробка логіки ворогів, що передбачає декілька станів: патрулювання, переслідування гравця, атакуюча поведінка, відслідковування дистанції, а також анімацію їхніх дій;

– побудова локацій гри з використанням Tilemap і декількох палет: для трави, землі, води, стін і декоративних елементів. Створення стартової зони, обмежень переміщення та об’єктів оточення;

– створення системи сцен, включаючи головне меню, перехід між рівнями та забезпечення плавного завантаження сцен;

– проведення тестування та налагодження коду на всіх етапах розробки. Перевірка стабільності поведінки ігрових об’єктів, логіки бою, інтерфейсу, переходів між сценами та продуктивності. Застосовано ручне тестування, візуальний дебаг, сценарний аналіз і перевірку крайніх випадків.

Для вивчення підходів до організації поведінки ворогів і станів персонажів використовувались матеріали платформи GameDev.tv [4]. Крім реалізації основної функціональності гри, я поставив перед собою завдання оволодіти технологічною базою для повноцінної розробки ігрового програмного забезпечення. Зокрема, у процесі роботи було:

– вивчено принципи роботи ігрового рушія Unity, його внутрішню архітектуру та засоби інтеграції графіки, фізики й анімацій;

– опановано створення 2D-анімацій за допомогою Animator і Animation Clip;

– застосовано колайдери та тригери для реалізації фізичних і логічних взаємодій у грі;

– організовано програмний код за принципами модульності, повторного використання та розширюваності.

Результатом виконання цих завдань стане інтерактивна 2D-гра, що демонструє базові ігрові механіки та є основою для подальшого розвитку. Незважаючи на домінування 3D-проектів на сучасному ринку відеоігор, 2D-ігри зберігають популярність завдяки стилізованому вигляду, меншій ресурсоємності, доступності до реалізації та великому потенціалу для творчості. Саме тому розробка 2D-гри є актуальною задачею як для освітніх, так і для практичних цілей у сфері інженерії програмного забезпечення.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

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

На початковому етапі розробки гри «Fragments of the World» було важливо визначити як загальні цілі, так і технічні вимоги до проекту. Оскільки гра реалізовується в жанрі 2D-RPG, вона має відповідати як функціональним очікуванням гравця, так і технічним вимогам до стабільної, масштабованої системи.

Перед початком проектування була проведена аналітика аналогічних проектів, які реалізовані в Unity. Зокрема, було вивчено структуру таких популярних 2D-ігор, як Dead Cells, Hollow Knight та Celeste. Кожна з них поєднує просту візуальну естетику з продуманою механікою та плавною взаємодією. При розробці «Fragments of the World» було вирішено використати подібні принципи: модульність, анімаційну плавність, підтримку розширення (нові вороги, предмети, сцени) та мінімалістичний інтерфейс.

Одним з важливих моментів було визначення функціональних вимог. Гравці повинні мати можливість вільно рухатися на ігрових картках, взаємодіяти з NPC та екологічними предметами, боротися з зворотним зв'язком та боротися з ворогами за допомогою простих бойових систем. Крім того, ключовою частиною є система перехідних між місцевостями. Це створити чудовий світ фантазій.

Важливо також було розглянути нефункціональні вимоги. Якщо ви почнете на середньому ПК, вам потрібно мати стабільну продуктивність від логічної структури проекту, яка дозволяє розширити гру без кардинальної обробки коду.

Для реалізації програмного продукту було попередньо спроектовано загальну архітектуру гри, що дозволило чітко розподілити функціональність між

окремими компонентами. Структурну взаємодію основних класів та їхню ієрархію представлено на відповідній UML-діаграмі (рис. 2.1).

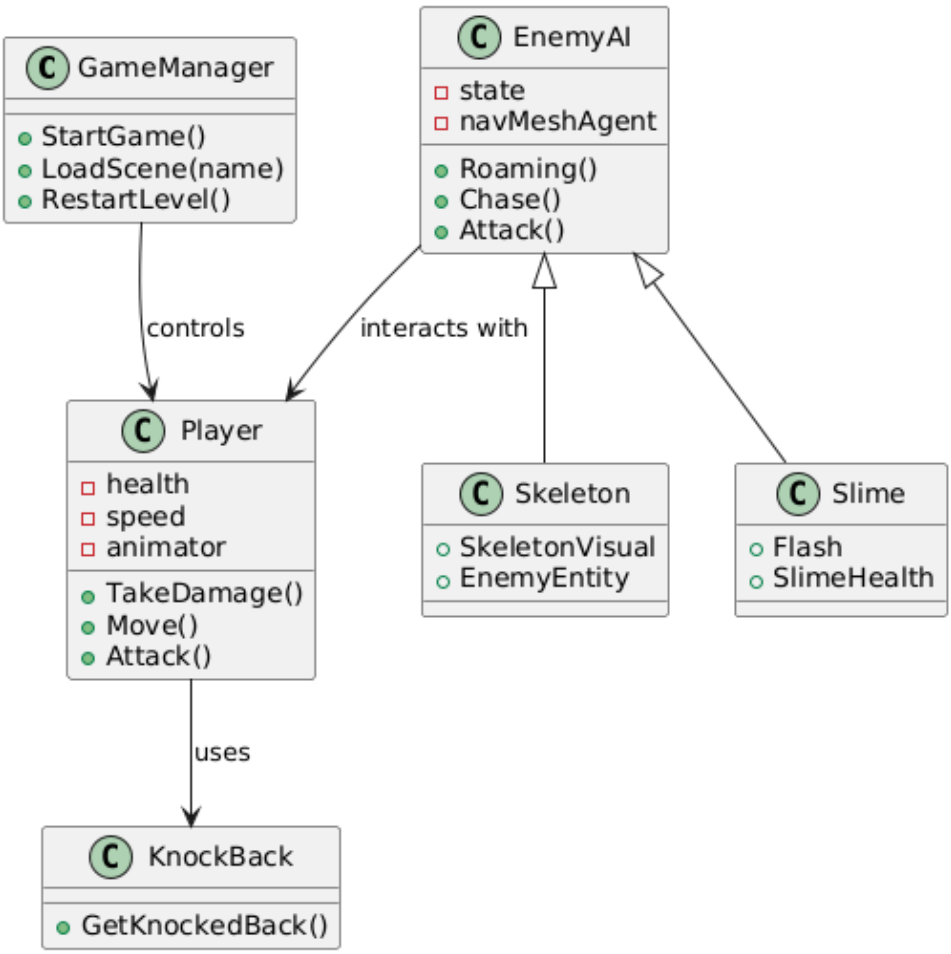

Рисунок 2.1 – Компонентна структура гри

Під час розробки механіки гри окрему увагу було приділено зручності розширення. Наприклад, система ворогів реалізована так, що дозволяє легко додавати нові типи противників із різною поведінкою без зміни основного коду гри. Кожен ворог наслідує скрипт EnemyAI, що зменшує дублювання логіки та полегшує тестування.

При розробці архітектури гри я застосував модульний підхід, при якому основні елементи логіки винесені в окремі скрипти. Зокрема:

– Player – обробка керування та взаємодії з об’єктами;

– PlayerVisual – обробка анімацій та зовнішнього вигляду (рис. 2.2);

– EnemyAI – базова логіка руху, виявлення гравця, патрулювання;

– Grid – сітка, яка складається з чотирьох компонентів для різних видів ландшафту;

– SceneManagement – реалізація переходів між сценами;

– Weapon – управління, зовнішній вигляд та анімація зброї.

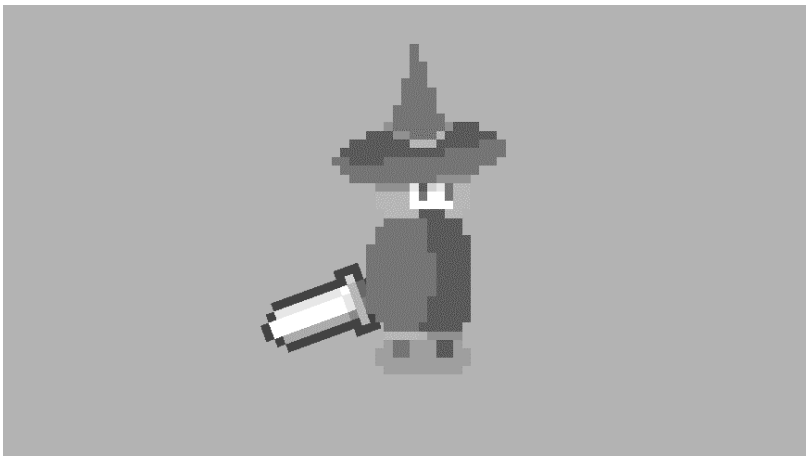

Рисунок 2.2 – Зовнішній вигляд персонажа

У процесі визначення вимог важливо було врахувати, що гра повинна мати логічну структуру сцен. Кожна локація у грі – це окрема сцена, яка завантажується при вході гравця в спеціальну зону переходу. Це дозволяє уникати перевантаження однієї сцени зайвими об’єктами та оптимізувати використання пам’яті. Кожна сцена містить мінімально необхідний набір об’єктів, що дозволяє зберігати продуктивність навіть при наявності кількох ворогів, анімацій та UI-елементів.

Також у проектуванні враховувалась можливість розширення. Логіка гри побудована так, щоб при потребі можна було легко додати нові види ворогів або механіки. Для цього системи проекту реалізовані незалежно одна від одної: UI не залежить від логіки руху, а виявлення зіткнень не пов’язане напряму з системою діалогів. Це дозволяє працювати над кожним компонентом ізольовано, не порушуючи цілісності проекту.

Ініціалізація головного героя при запуску гри (ліст. 2.1). Цей фрагмент коду відповідає за початкове налаштування героя під час запуску

сцени в Unity. Він виконується автоматично на старті об’єкта і забезпечує збереження посилань на ключові компоненти, такі як фізична система та механіка відштовхування. Це дозволяє герою взаємодіяти з ігровим світом, реагувати на зіткнення та інші події.

Лістинг 2.1 – Ініціалізація героя

```
private void Awake() {
    Instance = this;
    _rb = GetComponent<Rigidbody2D>();
    _knockBack = GetComponent<KnockBack>();
}
```

кінець лістингу 2.1

Метою коду є створення основи для роботи героя, уникаючи повторного пошуку компонентів під час гри, що оптимізує продуктивність. Змінна Instance використовується для доступу до об’єкта героя з інших скриптів через патерн Singleton. Фізичний компонент керує рухом і зіткненнями, а система відштовхування обробляє реакцію на удари.

Такий підхід стандартний для Unity, оскільки спрощує організацію коду та полегшує додавання нових функцій, наприклад, анімацій чи здоров’я. Використання GetComponent забезпечує гнучкість і можливість повторного використання коду в різних сценах.

Крім того, ініціалізація в цьому методі дозволяє централізовано керувати залежностями об’єкта героя. Наприклад, якщо в майбутньому знадобиться додати нові компоненти, такі як система інвентарю чи аудіо, їх можна легко інтегрувати, додавши відповідні виклики GetComponent у цей же метод. Це зменшує ймовірність помилок, пов’язаних із відсутністю компонентів, оскільки перевірка їх наявності відбувається на етапі запуску.

Важливою особливістю цього коду є його універсальність. Оскільки він не залежить від конкретних параметрів сцени чи гри, його можна застосовувати до різних типів героїв або навіть інших об’єктів, які потребують подібної ініціалізації. Наприклад, той же підхід може використовуватися для ініціалізації

ворогів чи інтерактивних об’єктів, що мають фізичні властивості та реагують на взаємодію.

Код також сприяє модульності проекту. Завдяки чіткому розподілу обов’язків між компонентами (фізика, відштовхування) розробники можуть легко замінювати або вдосконалювати окремі модулі, не зачіпаючи основну логіку героя. Це особливо корисно в командній розробці, де різні фахівці працюють над різними аспектами гри.

Реалізація переміщення героя (ліст. 2.2) здійснена за допомогою Rigidbody2D.MovePosition. Він забезпечує плавне переміщення персонажа на основі введеного гравцем напрямку, визначеного через вектор. Код також перевіряє, чи персонаж активно рухається, що впливає на зміну анімацій, наприклад, для відображення бігу. Оновлення в методі, призначеному для фізичних розрахунків, гарантує стабільну взаємодію з ігровими об’єктами, запобігаючи ривкам чи неправильним зіткненням.

Лістинг 2.2 – Рух героя

```
private void HandleMovement() {
    _rb.MovePosition(_rb.position + inputVector * (_movingSpeed * Time.fixedDeltaTime));
    _isRunning = inputVector.magnitude > _minMovingSpeed;
}
```

кінець лістингу 2.2

Такий підхід синхронізує рух із фізичною системою Unity, забезпечуючи коректну обробку зіткнень і плавність незалежно від частоти кадрів. Швидкість руху нормалізується множенням вектора введення на відповідний коефіцієнт і часовий інтервал, що забезпечує однакову поведінку на різних пристроях.

Цей метод руху гнучкий і легко адаптується. Наприклад, зміна параметра швидкості дозволяє налаштувати темп персонажа для різних ігрових ситуацій, таких як швидкий біг чи повільна хода. Логіка також підтримує зв’язок із системою анімацій, дозволяючи змінювати візуальний стан персонажа залежно від його руху.

Код ізольовано в окремому методі, що сприяє модульності. Це спрощує додавання нових механік, таких як стрибки чи взаємодія з об’єктами, без зміни основної логіки. Така структура полегшує підтримку та розширення функціоналу в більших проектах.

Використання фізичного підходу до переміщення зменшує ризик помилок, пов’язаних із колізіями. На відміну від прямого змінення позиції, цей метод дозволяє двигуну Unity коректно обробляти взаємодію з перешкодами, такими як стіни чи вороги, забезпечуючи стабільність гри.

Механізм отримання шкоди (ліст. 2.3). Цей фрагмент коду відповідає за обробку отримання шкоди головним героєм. Логіка зменшує значення здоров’я персонажа, активує ефект відштовхування та перевіряє, чи герой залишився живим. Після отримання шкоди вводиться затримка, що тимчасово захищає героя від подальших ударів. Це запобігає швидкій загибелі від численних ударів і дає гравцеві змогу відреагувати чи втекти.

Лістинг 2.3 – Отримання шкоди

```
public void TakeDamage(Transform damageSource, int damage) {
    if (_canTakeDamage && _isAlive) {
        _canTakeDamage = false;
        _currentHealth = Mathf.Max(0, _currentHealth -= damage);
        _knockBack.GetKnockedBack(damageSource);
        StartCoroutine(DamageRecoveryRoutine());
    }
    DetectDeath();
}
```

кінець лістингу 2.3

Код перевіряє, чи персонаж може отримувати шкоду і чи він живий, перед виконанням дій. Здоров’я зменшується на задану величину, але не нижче нуля, що забезпечує правильне обчислення. Ефект відштовхування активується з урахуванням джерела шкоди, що додає реалістичності взаємодії з ворогами. Затримка реалізується через запуск окремої процедури, що блокує нові атаки на певний час.

Такий підхід підвищує ігровий комфорт, надаючи гравцеві шанс адаптуватися до бойових ситуацій. Механізм легко масштабується: наприклад, можна додати модифікатори шкоди чи спеціальні ефекти, такі як візуальні індикатори чи звуки, без зміни основної логіки.

Логіка ізольована в окремому методі, що сприяє модульності коду. Це дозволяє легко інтегрувати додаткові функції, наприклад, ефекти зцілення чи тимчасові бонуси захисту. Перевірка стану персонажа після отримання шкоди забезпечує своєчасне виявлення смерті, що важливо для коректного завершення ігрового циклу.

Код також сприяє стабільності гри, оскільки враховує можливість численних атак і захищає від небажаних сценаріїв, таких як миттєва загибель. Використання умовних перевірок і процедур затримки робить систему надійною і придатною для використання в різних бойових сценаріях.

Ефект фізичного відштовхування головного героя після отримання удару додає реалістичності до бойової системи гри (ліст. 2.4). Він обчислює напрям сили на основі позиції джерела шкоди та застосовує її, щоб персонаж відлетів назад. Така механіка додає боям динамічності й візуальної виразності, роблячи взаємодію з ворогами більш відчутною для гравця. Крім того, відштовхування тимчасово обмежує дії персонажа, що сприяє реалізму та збалансованості бойової системи.

Лістинг 2.4 – Відштовхування головного героя

```
public void GetKnockedBack(Transform damageSourse) {
    IsGettingKnockedBack = true;
    _knockBackMovingTimer = _knockBackMovingTimerMax;
    Vector2 difference = (transform.position -
damageSourse.position).normalized * _knockBackForce;
    _rb.AddForce(difference, ForceMode2D.Impulse);
}
```

кінець лістингу 2.4

Код активує стан відштовхування, встановлюючи таймер для контролю тривалості ефекту. Напрям руху розраховується як нормалізована різниця між

позиціями героя та джерела шкоди, помножена на задану силу. Ця сила застосовується до фізичного компонента, забезпечуючи миттєвий імпульс для відштовхування. Такий підхід гарантує природну поведінку персонажа в ігровому світі.

Механіка відштовхування легко адаптується, наприклад, зміна сили чи тривалості ефекту дозволяє налаштувати її під різні типи атак або ворогів. Це додає гнучкості в дизайні бойових сценаріїв, дозволяючи створювати унікальні взаємодії, наприклад, сильніші відштовхування від босів.

Код ізольовано в окремому методі, що забезпечує модульність і полегшує його інтеграцію з іншими системами, такими як анімації чи звукові ефекти. Обмеження дій персонажа під час відштовхування запобігає небажаним діям гравця.

Зміна напрямку, у який дивиться персонаж (ліст. 2.5), реалізована шляхом порівняння позиції гравця з позицією курсора. Це використовується для коректного відображення спрайта і спрямованості атак. Такий підхід дозволяє реалізувати інтуїтивно зрозуміле управління, де герой завжди дивиться у напрямку дії гравця. Це також важливо для синхронізації з анімаціями та правильного виведення зброї на сцені.

Лістинг 2.5 – Визначення напрямку погляду гравця

```
private void AdjustPlayerFacingDirection() {
    Vector3 mousePos = GameInput.Instance.GetMousePosition();
    Vector3 playerPos = Player.Instance.GetPlayerScreenPosition();
    spriteRenderer.flipX = mousePos.x < playerPos.x;
}
```

кінець лістингу 2.5

Код використовує позицію курсора, отриману через систему вводу, і порівнює її з екранною позицією героя. Залежно від результату порівняння змінюється властивість спрайта, що відповідає за його горизонтальне віддзеркалення. Такий підхід забезпечує миттєву реакцію на дії гравця, що

особливо важливо в динамічних іграх, де швидкість і точність керування впливають на ігровий досвід.

Цей метод гнучкий і може бути адаптований для різних стилів гри. Наприклад, замість позиції курсора можна використовувати позицію цілі чи іншого об’єкта, що дозволить герою орієнтуватися на ворогів або ключові точки в ігровому світі. Це робить код універсальним для проектів із різними механіками взаємодії.

Наступний код відповідає за створення ворога на заданій позиції під час запуску сцени або виклику події (ліст 2.6). Щоб ворог коректно відображався на сцені, необхідно правильно налаштувати його Sorting Layer та Order in Layer. Важливо також розмістити ворога в шарі, відповідному до елементів оточення: наприклад, нижче за дерева, щоб він частково ховався за ними, але вище за землю – для правильного візуального ефекту.

Лістинг 2.6 – Початкова позиція ворогів

```
public void SpawnEnemy(Vector3 spawnPosition) {
    Instantiate(enemyPrefab, spawnPosition, Quaternion.identity);
}
```

кінець лістингу 2.6

Анімація смерті персонажа (ліст. 2.7) активується після виклику події смерті. Встановлюється відповідний прапорець у Animator, що змінює стан анімації. Цей механізм забезпечує плавний перехід до мертвого стану та зупиняє подальшу взаємодію з об'єктом. Також це дозволяє підключити додаткові ефекти, такі як звуковий супровід або затримка перед перезапуском сцени.

Лістинг 2.7 – Візуальна анімація смерті

```
private void Player_OnPlayerDeath(object sender, System.EventArgs e) {
    animator.SetBool(IS_DIE, true);
}
```

кінець лістингу 2.7

Механізм підтримує розширення функціоналу. Наприклад, можна додати відтворення звукових ефектів, запуск візуальних частинок (таких як кров чи спалахи) або таймер для відображення екрана завершення гри. Це робить код універсальним для різних сценаріїв, де смерть персонажа супроводжується додатковими діями.

Атака ворога (ліст. 2.8) реалізована на основі системи подій. Після завершення таймера перезарядки ворог надсилає подію OnEnemyAttack, на яку можуть бути підписані обробники.

Лістинг 2.8 – Атака ворога

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

кінець лістингу 2.8

Логіка атаки перевіряє, чи минув час, необхідний для наступної атаки, порівнюючи поточний час із збереженим моментом останньої атаки. Якщо умова виконується, викликається подія, яка сповіщає підписані компоненти про атаку, і таймер перезарядки оновлюється. Такий підхід дозволяє гнучко обробляти атаки ворога, не прив’язуючи їх до конкретної реалізації.

Використання системи подій робить код модульним і універсальним. Наприклад, обробники можуть активувати анімацію атаки, відтворювати звукові ефекти чи завдавати шкоди гравцеві. Це дозволяє легко додавати нові функції, такі як різні типи атак, без зміни основної логіки.

Орієнтація ворога на гравця (ліст. 2.9) змінюється за допомогою обертання transform.rotation. Це дозволяє візуально спрямувати ворога у бік цілі перед атакою або рухом. Код аналізує горизонтальні координати ворога та цілі. Якщо ціль знаходиться ліворуч, ворог повертається на 180 градусів по осі Y, інакше

орієнтується праворуч. Цей метод забезпечує точне відображення напрямку ворога, що необхідно для анімацій і бойової взаємодії.

Лістинг 2.9 – Напрямок ворога до гравця

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

кінець лістингу 2.9

Механізм ефективний і простий, дозволяючи швидко реагувати на зміну позиції гравця. Його можна вдосконалити, наприклад, додавши обертання за іншими осями для 3D-простору чи врахування вертикального розташування цілі.

Таким чином, розробка гри «Fragments of the World» включала глибокий аналіз вимог і побудову чіткої архітектури, яка забезпечує гнучкість, стабільність та зрозумілу структуру проекту. Це дозволило перейти до реалізації з мінімальними труднощами й забезпечити можливість подальшого розширення гри. Використання модульного підходу до створення лістингів, таких як ініціалізація персонажа, обробка атак чи анімацій, сприяло легкості інтеграції нових функцій. Система подій і чітке розділення логіки підвищили ефективність розробки та спростили тестування. Усі ці аспекти закладають міцну основу для майбутніх оновлень і масштабування проекту.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

У ході реалізації поставленого завдання було необхідно обрати такі інструменти, методи та алгоритми, які забезпечать ефективну розробку 2D-гри з урахуванням сучасних вимог до продуктивності, модульності, зручності користувача та розширюваності проекту.

Основним середовищем розробки було обрано ігровий рушій Unity, який є одним із найпопулярніших засобів для створення 2D- та 3D-ігор. Його перевагами є широкий набір вбудованих інструментів, підтримка фізики, анімацій, системи сцен, UI, Tilemap, а також багата екосистема ресурсів і документації. Крім того, Unity забезпечує інтеграцію з C# – сучасною об’єктно-орієнтованою мовою програмування, яка використовується для реалізації внутрішньої логіки гри.

Для написання програмного коду було обрано середовище Visual Studio, яке забезпечує повноцінну підтримку IntelliSense, автоматичне виявлення помилок, налагодження та зручну навігацію між компонентами проекту. Таке поєднання Unity та Visual Studio дозволило ефективно керувати як ігровою логікою, так і структурою сцени, ресурсами та поведінкою об’єктів.

Під час реалізації гри використовувались такі методи:

– компонентно-орієнтоване програмування – кожен об’єкт у грі є поєднанням компонентів, що реалізують окремі функції. Такий підхід спрощує повторне використання коду та масштабування;

– кодійно-орієнтоване програмування – для взаємодії між об’єктами застосовано делегати, події, callback-методи, що дозволяє уникати жорсткого зв’язування між класами;

– об’єктно-орієнтований підхід – логіка гри структурована за допомогою класів і спадкування, що забезпечує читабельність, повторне використання та розширюваність коду;

– сценарно-керований дизайн – більшість поведінкових механік реалізовано через набори станів (стан руху, атаки, смерті тощо), що дозволяє зручно керувати поведінкою гравця та ворогів.

Я використовував алгоритм навігації NavMesh для впровадження рухів противника та поведінки, що дозволяє об'єктам переходити на карту, щоб вирішити перешкоди. Це забезпечує природну поведінку та гнучкість противника у побудові складних шляхів. Для деяких об'єктів, які NavMesh не

використовував, власний простий алгоритм руху був використаний у змінах випадкової орієнтації.

Бойова система реалізується на основі виявлення зіткнень через Collider2D та тригерні події. Коли гравець контактує з ворогом, він втрачає здоров’я і коли він досягає нуля, починається послідовність смерті. Система пошкоджень містить параметри затримки, безсмертя та візуального зворотного зв'язку (анімація та віддача).

Для створення ігрового середовища була використана система Tilemap, яка дозволяє швидко формувати картки на основі плиток. Кожна поверхня (вода, підлога, трава) мала ще одну палітру, яка зробила б зручною для побудови сцени. Зіткнення непрохідних об’єктів реалізуються за допомогою TilemapCollider2D.

Графічні частини реалізуються в спрайтах та анімаціях через контролер аніматора. Для зменшення навантаження на систему використовували зміни матеріалу для деяких ворогів замість стандартних аніматорів. Інтерфейс був створений інтерфейсом Unity (полотно, зображення, текст) та використовували динамічні оновлення значень за допомогою сценаріїв.

РОЗДІЛ 3
РОЗРОБКА ДВОВИМІРНОЇ ГРИ З ВИКОРИСТАННЯМ UNITY

3.1 Практична реалізація об’єкта проєктування

У процесі реалізації гри «Fragments of the World» було створено повноцінну структуру 2D-гри (рис. 3.1), що охоплює ключові ігрові механіки, логіку взаємодії між об’єктами, а також елементи користувацького інтерфейсу. Основна мета розробки полягала в забезпеченні гравцеві можливості досліджувати світ, взаємодіяти з об’єктами, брати участь у боях та керувати персонажем у зручному та інтуїтивно зрозумілому середовищі.

Рисунок 3.1 – Кінцевий вигляд ієрархії гри

Рух персонажа реалізовано за допомогою обробки введення з клавіатури. Гравець може вільно переміщуватись у чотирьох напрямках, анімація змінюється динамічно залежно від наявності руху. Механіка переміщення розроблена таким чином, щоб забезпечити плавність та природну поведінку персонажа. Особливу увагу приділено руху, який дозволяє відрізнити статичний стан гравця від активного переміщення – це необхідно для правильного відображення анімацій.

Система отримання шкоди є важливим елементом бою. При отриманні удару від ворога гравець не лише втрачає частину здоров’я, а й зазнає відштовхування (knockback). Цей механізм дозволяє зробити бій більш динамічним і візуально зрозумілим. Щоб уникнути повторного миттєвого отримання шкоди, реалізована коротка пауза невразливості. Таким чином, гра забезпечує баланс між складністю й комфортом для гравця.

Особливу увагу під час реалізації проекту було приділено візуалізації персонажів через систему анімацій. Анімації виконують не лише декоративну функцію, а й суттєво покращують зворотній зв’язок для гравця, дозволяючи краще орієнтуватися у стані персонажа та подіях у грі.

Для гравця реалізовано три основні анімаційні стани: анімація покою**,** анімація ходьби та анімація смерті**.** Вони виконані за допомогою компонента Animator в Unity. Анімаційні переходи налаштовуються в анімаційному графі (Animator Controller) на основі логічних змінних «IsRunning» та «IsDie», які змінюються під час гри в залежності від дій користувача або стану персонажа (рис. 3.2). Завдяки цьому гравець бачить плавний перехід між станами: коли герой починає рухатися, змінюється спрайт і запускається циклічна анімація ходьби. У стані спокою він повертається до статичної пози, а при смерті відтворюється відповідна фінальна анімація.

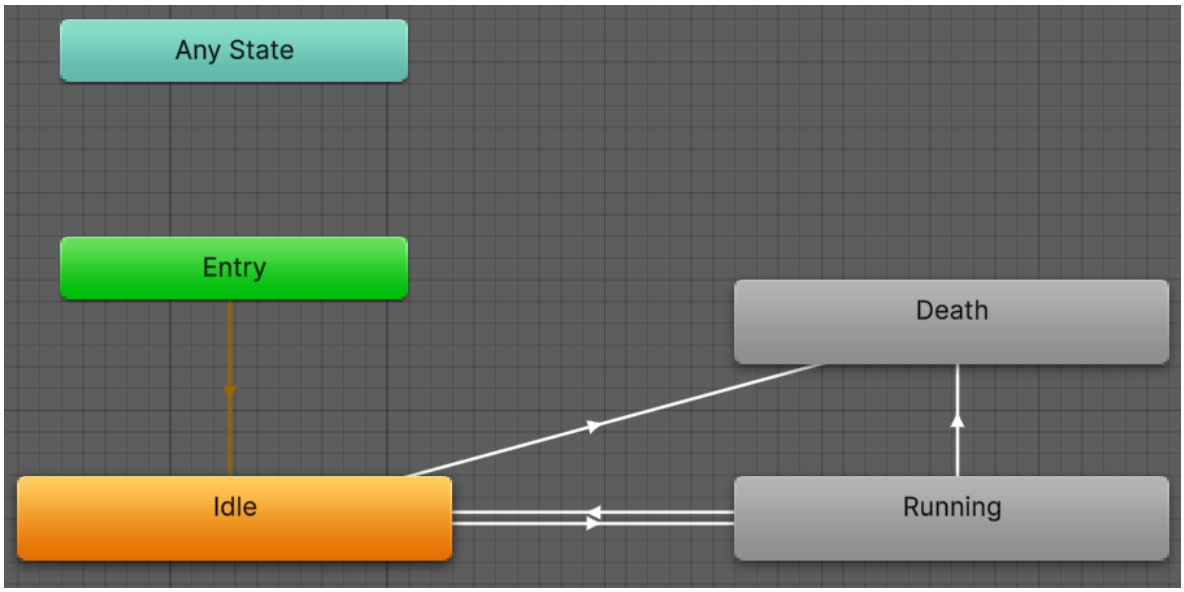

Рисунок 3.2 – Вигляд аніматора PlayerVisual

У межах реалізації гри значна увага приділялася не лише функціональності, а й загальному відчуттю гри**.** Усі анімації були синхронізовані з логікою гри, так щоб кожна дія гравця мала відповідний візуальний відгук. Наприклад, при атаці гравця спочатку вмикається відповідна анімація, і лише після досягнення піку атаки відбувається нанесення шкоди. Це забезпечує точне візуальне узгодження механік.

Також було реалізовано просту систему подій і делегатів, яка дозволяє іншим об’єктам реагувати на дії гравця. Наприклад, при смерті гравця викликається подія «OnPlayerDeath», яка інформує інші системи про зміну стану (відключення управління, запуск анімації смерті). Це дозволяє уникнути жорсткої прив’язки між об’єктами, що відповідає принципам слабкої зв’язаності.

Окрім цього, була реалізована система врахування напрямку погляду, яка динамічно перевіряє положення курсора миші та відображає поворот персонажа вліво або вправо. Це покращує візуальне сприйняття та дозволяє будувати бойову систему, яка враховує орієнтацію гравця. Для багатьох ворогів реалізовані унікальні алгоритми атаки та патрулювання: скелет атакує при наближенні, а слайм – тільки після тривалого затримування в зоні гравця.

Для ворога-скелета використано схожий підхід – анімація також реалізована через Animator. Але при цьому враховано специфіку поведінки ворога: скелет має окремі анімації руху, атаки та смерті (рис. 3.3). Перемикання між ними виконується залежно від поточного стану ворога – патрулювання, переслідування, атака або смерть. Такий підхід дозволяє зробити ворога візуально виразним і живим у контексті гри.

Також для анімацій ворогів було налаштовано зв’язок між логікою станів і анімаційними параметрами, що забезпечує синхронність дій і візуального відображення. Всі дії ворогів, зокрема затримки перед атакою, переслідування або смерть, супроводжуються відповідними анімаційними переходами, що підвищує загальну якість сприйняття гри та занурення у процес.

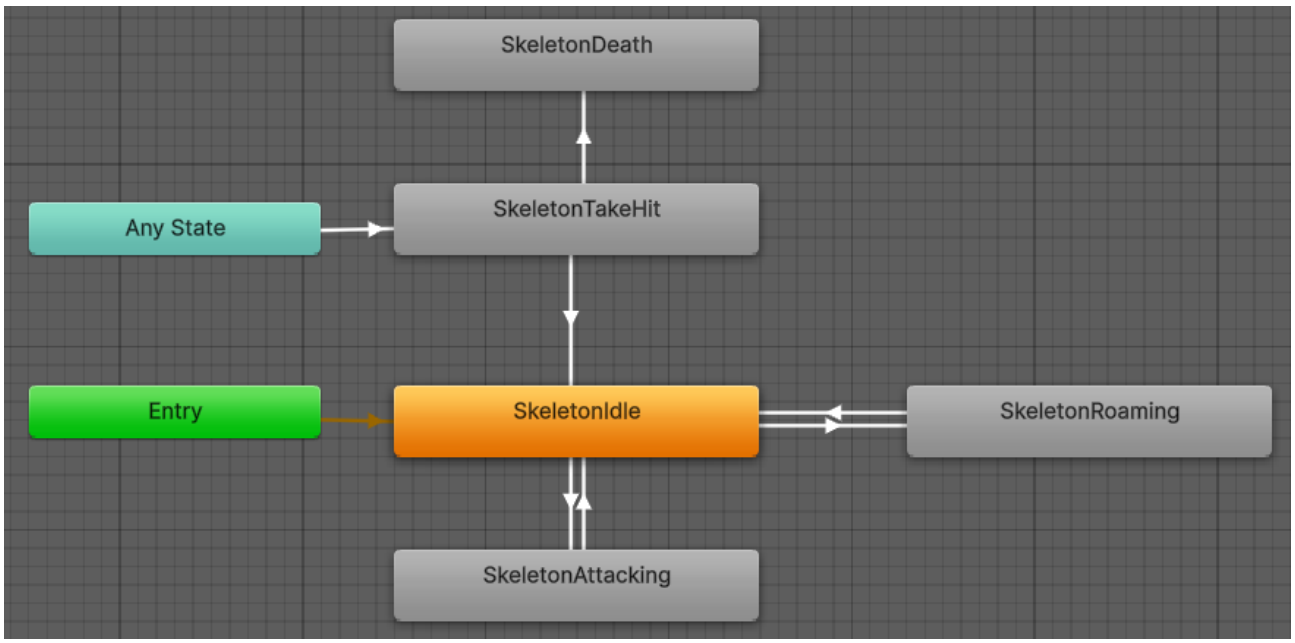

Рисунок 3.3 – Вигляд аніматора SkeletonVisual

Натомість для ворога-слайма було використано інший підхід – анімація реалізована не через Animator, а за допомогою зміни матеріалів. Слайм – це простий об’єкт, який змінює свою візуальну форму за рахунок циклічного оновлення текстур або зміни кольору матеріалу, що дозволяє створити ефект дрижачого слизу, готовий приклад якого було звято з Unity Asset Store і було адаптовано під механіку гри [5]. Така реалізація потребує менше ресурсів і дозволяє швидко змінювати вигляд ворога без складної анімаційної логіки. Це також демонструє гнучкість підходу до анімації в Unity: різні типи об’єктів можуть використовувати різні методи візуалізації залежно від складності та цілей.

Загалом, гнучке поєднання стандартної анімаційної системи Unity (Animator) із легшими методами на основі матеріалів дозволило досягти візуального різноманіття та оптимізувати продуктивність гри. Таке розділення методів є прикладом свідомого дизайну, який враховує ігрову роль кожного персонажа та його місце у загальній структурі проекту.

Для реалізації динамічного стеження за гравцем було використано систему камер Cinemachine [6], яка входить до складу Unity як окремий пакет. Це

потужний інструмент, що дозволяє створювати плавне, кінематографічне слідкування за об’єктами без написання великої кількості коду.

Cinemachine надала мені можливість створити так звану віртуальну камеру, яка автоматично відстежує положення головного героя. У даному проекті віртуальна камера налаштована на слідкування за гравцем, що дозволяє утримувати його у центрі екрану під час руху.

Після створення об’єкта CinemachineCamera, його компоненту було призначено таргет – об’єкт гравця. Завдяки системі налаштувань в редакторі можна було визначити бажану поведінку камери: ступінь згладжування, мертву зону (зону, в якій гравець може трохи рухатися без руху камери), затримку при переміщенні та межі слідкування. Це дозволило досягти ефекту плаваючої камери, яка не миттєво стрибає за гравцем, а слідкує за ним плавно та природно.

Ключовою перевагою Cinemachine стало те, що вона знімає необхідність вручну писати скрипти для позиціонування камери та її інтерполяції. Камера автоматично виконує ці дії з урахуванням фізичних характеристик сцени та руху гравця.

У деяких сценах, наприклад при зміні локацій або під час діалогів, Cinemachine дозволяє змінювати активну віртуальну камеру на іншу. Це дає змогу зробити плавні переходи між різними видами, наприклад, наблизити камеру до персонажа в особливий момент гри або надати огляд сцени під іншим кутом. Віртуальні камери можна перемикати вручну або через тригери, що дає гнучкість у сценічній постановці.

Завдяки інтеграції з основними компонентами Unity, Cinemachine відмінно працює в поєднанні з іншими системами гри, зокрема UI, фізикою та світлом. Крім того, вона дозволяє не лише стежити за гравцем, а й реагувати на зміну орієнтації, масштабувати поле зору, обмежувати зону перегляду, якщо це потрібно для певних локацій.

Вороги реалізовані через просту, але ефективну систему станів. Ворог може знаходитися у стані очікування, патрулювання, переслідування гравця або атаки. Залежно від відстані до гравця, ворог автоматично змінює свою поведінку.

Це створює динамічну взаємодію, де гравець відчуває, що за ним активно спостерігають та реагують. Атаки ворога реалізовані з урахуванням часу, що дозволяє уникнути надмірної частоти ударів.

Система переходів між сценами була реалізована з метою створення враження відкритого світу [7]. Кожна локація є окремою сценою, що завантажується при досягненні гравцем певної зони переходу. Щоб уникнути затримок або візуальних стрибків, перехід супроводжується затемненням екрана, а саме ефектом fade та плавним переміщенням гравця в нову точку входу.

Під час розробки інтерфейсу також було враховано можливість його масштабування. Система Canvas адаптується до різних роздільностей екрана, забезпечуючи стабільне відображення UI-елементів як на великих, так і на малих екранах. Такий підхід дозволяє уникнути розривів або перекриття елементів навіть при зміні розміру вікна гри.

Для створення ігрових локацій у грі, було використано систему Tilemap, яка є частиною 2D Tilemap Tools у Unity. Вона дозволяє ефективно будувати великі карти шляхом розміщення невеликих спрайтів (тайлів) у сітці. Такий підхід значно пришвидшує побудову сцен, спрощує редагування середовища та дозволяє легко вносити зміни в геометрію рівня.

В межах гри було створено кілька окремих Tile Palette**,** кожна з яких відповідала певному типу поверхні (рис. 3.4). Наприклад, для трави використовувалася окрема палітра з варіаціями зелених тайлів, які забезпечують природний вигляд поверхні. Для землі було обрано іншу палітру з коричневими та пісочними тонами, які утворюють підґрунтя та стежки. Окрема палітра була присвячена воді, що імітують рідини, водойми чи річки.

Для ще більшої деталізації навколишнього середовища було додано палітру декоративних елементів, яка включає рослинність, каміння, залишки споруд і бар’єри, що не лише прикрашають рівень, а й впливають на геймплей, виступаючи перешкодами та укриттями.

Рисунок 3.4 – Tile Palette

Кожна палітра містить набір тайлів, які створюють візуально узгоджене середовище. Це дозволяє досягти якісного стилістичного поділу між різними типами поверхні: гравець інтуїтивно розуміє, що трава – це прохідна зона, а вода і земля – перешкода. Завдяки цьому створюється не лише візуальна, а й функціональна структура рівня.

У грі реалізовано шар зіткнень (TilemapCollider2D), який обробляє фізичні обмеження на рівні – гравець не може проходити крізь воду або стіни, адже ці тайли мають відповідні колайдери.

Редагування карти здійснюється безпосередньо у редакторі. Можна просто малювати по Tilemap, вибираючи тайли з палітри. Такий підхід дуже зручний при прототипуванні або зміні рівнів, адже дає змогу швидко адаптувати ігрове середовище без потреби у переписуванні коду.

Для реалізації навігації ворогів по ігровій карті використовується NavMesh-система, вбудована в Unity. Це інструмент, що дозволяє ворогам автоматично обирати найкоротший маршрут до цілі, обходячи перешкоди. Система побудована на алгоритмах навігаційної сітки (navigation mesh), що створюється на основі геометрії сцени.

На етапі підготовки сцени було визначено, які об’єкти є прохідними, а які непрохідними. Поверхні, по яких можуть пересуватися вороги, були позначені як «Walkable», а перешкоди (стіни, вода, декоративні об’єкти) – «Not Walkable».

Кожен ворожий персонаж має компонент NavMeshAgent, який відповідає за переміщення по цій навігаційній сітці. При зміні положення гравця ворог автоматично оновлює свою цільову позицію, і система NavMeshAgent прокладає до неї маршрут. Навіть якщо на шляху виникають об’єкти, ворог буде їх обходити, а не намагатиметься йти напряму, як у примітивних системах.

Це дозволяє уникнути ситуацій, коли вороги застрягають у колізіях або проходять крізь перешкоди. Завдяки NavMesh враховуються реальні межі карти, і ворог не зможе зайти, наприклад, у воду, якщо ця зона позначена як непрохідна. Таким чином, система навігації адаптується до складної геометрії сцени і забезпечує реалістичну поведінку ворогів.

Окрім того, у NavMeshAgent можна регулювати такі параметри, як швидкість, радіус повороту, відстань зупинки, що дозволяє зробити кожен тип ворога унікальним у своєму русі. Наприклад, швидкий слайм може мати високу швидкість і коротку дистанцію зупинки, тоді як повільний скелет рухається повільніше і реагує з затримкою.

У проекті також реалізована перевірка відстані до гравця. Якщо гравець наближається до ворога на задану відстань, ворог переходить зі стану патрулювання у стан переслідування, і його NavMeshAgent починає рух у напрямку гравця. Коли ворог достатньо наближається, він переходить у стан атаки. Якщо ж гравець покидає зону виявлення, ворог повертається до початкової позиції або починає патрулювання.

Завдяки використанню NavMesh навігація ворогів вийшла адаптивною, гнучкою та стабільною. Вороги не лише реагують на положення гравця, а й правильно взаємодіють із геометрією карти. Такий підхід дозволяє легко масштабувати систему.

У цілому, реалізовані механіки демонструють гнучкість використаних технологій, зокрема Unity та C#, а також добре структурований підхід до

побудови архітектури гри. Завдяки модульній структурі всі компоненти проекту можна легко розширити або модифікувати. Такий підхід дозволяє як підтримувати гру в майбутньому, так і масштабувати її, додаючи нові функції, ворогів, локації чи предмети без значного втручання в існуючий код.

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Завершальним етапом розробки гри «Fragments of the World» стало її тестування – критично важлива частина, що дозволяє не лише оцінити працездатність проекту, але й виявити приховані недоліки у логіці, візуальній частині чи функціональності. На цьому етапі гра проходила комплексну перевірку всіх реалізованих механік, взаємодії між об'єктами, коректності роботи інтерфейсу, поведінки ворогів, переходів між сценами та загальної стабільності.

Процес тестування проводився в інтерактивному режимі, безпосередньо в середовищі розробки Unity, а також у зібраній версії гри на цільовій платформі (Windows). Це дозволило виявити як помилки, пов'язані з логікою скриптів, так і технічні недопрацювання, які могли виникати лише під час запуску поза редактором. Оскільки проект розроблявся одноосібно, основна увага приділялась саме ручному тестуванню – перевірці кожного модуля гри у дії.

Першочергово була перевірена поведінка головного героя. Тестувались такі аспекти, як рух у межах сцени, зупинка, зміна напрямку, взаємодія з об'єктами середовища (рослинність, стіни, NPC), відгук на натискання клавіш та реакція на удари ворогів. На цьому етапі вдалося виявити проблему, коли після отримання шкоди гравець міг рухатись, попри заборону керування під час анімації нокауту. Для усунення помилки було додано перевірку прапора «_canMove», що блокує пересування до завершення анімаційного циклу.

Бойова система також потребувала ретельної перевірки. Випробовувалась не лише логіка нанесення шкоди, а й коректність запуску анімацій атаки, застосування knockback-ефекту, відображення візуального фідбеку. Було

встановлено, що при дуже частих ударах виникала колізія – атака могла не зчитуватися, якщо ворог знаходився безпосередньо в точці появи меча. Для вирішення проблеми було внесено зміни у систему виявлення зіткнень, додано компенсацію позиції хітбокса.

Окреме тестування проводилося для ворогів – скелета та слайма. Вони мають різні типи поведінки: один орієнтований на переслідування, інший – на стрибки з затримкою. Було перевірено, чи коректно працює NavMesh, чи ворог орієнтується в просторі, чи не застрягає у стінах або непрохідних об’єктах. Особливу увагу викликала ситуація, коли ворог після створення з’являвся поза навігаційною сіткою – це призводило до помилки при виклику «SetDestination()». Для запобігання подібному випадку в скрипт було додано перевірку navMeshAgent.isOnNavMesh, що гарантує безпечне призначення точки призначення лише у випадку, якщо агент дійсно знаходиться на навігаційній поверхні.

Під час тестування навігації між сценами було виявлено деякі нюанси, пов’язані з некоректним збереженням станів. Наприклад, у разі швидкого перемикання сцен частина ворогів не встигала видалятись, що створювало враження дублювання. Це вирішувалося шляхом додавання затримки перед зміною сцени і очищення об’єктів через функцію «DontDestroyOnLoad()» лише для окремих систем, такої як «GameManager».

Інтерфейс користувача тестувався у двох напрямках: функціональність (правильне оновлення індикаторів, перемикання панелей, робота меню) та адаптивність. Гра запускалась на різних роздільностях екрана, щоб перевірити, чи всі елементи розміщуються належним чином, чи не перекривають один одного та чи читаються тексти. Було виявлено, що без налаштувань «Canvas Scaler’a» інтерфейс міг виглядати некоректно у форматах 4:3 – після відповідної адаптації всі елементи відображались стабільно.

Також проводилося функціональне тестування системи смерті гравця. Була перевірена анімація, блокування керування, запуск сцени поразки та

повернення в головне меню. Аналогічно перевірялась логіка смерті ворогів, а саме знищення об’єкта після завершення анімації.

Усі помилки, знайдені в процесі тестування, були записані в звичайному текстовому документі та поступово видалені. Підтримка стабільності FPS спостерігалася, тим самим уникаючи утворення мертвих зон на стадії та точності структури спрайтів, особливо при одночасних взаємодіях між кількома об’єктами.

Підсумовуючи результати, можна стверджувати, що гра пройшла повний функціональний тест. Це забезпечило продуктивність, стабільну логіку та правильну візуальну конструкцію. Ідентифіковані недоліки були усунені, кажучи, що гра готова до подальшого вдосконалення, розширення або адаптації до нових механіків та платформ.

Методи налагодження (налагодження) активно використовувались для пошуку помилок, які відігравали важливу роль у процесі розробки. Нижче наведено детальне пояснення використовуваних методів, а також реальний приклад ситуації, яку вам довелося працювати.

Найпростішим способом розпізнати помилку було виведення текстового повідомлення на консоль. Це дозволяє певним функціям працювати та здійснювати змінні, які впливають на логіку гри. Наприклад, якщо гравець втрачений, консоль тепер відображає поточний рівень HP, щоб ви могли бачити зміни в динаміці.

Важливу роль у процесі розробки відіграли інструменти візуалізації Unity, зокрема Gizmos, який дозволяє відображати додаткові елементи на сцені для полегшення налагодження та перевірки. Наприклад, Gizmos використовувався для відображення напрямку погляду ворога та зони атаки, що допомагало візуально підтвердити правильність розрахунків. Такий підхід був особливо корисним при реалізації логіки орієнтації ворога на гравця, гарантуючи, що напрямок визначено правильно.

Застосування Gizmos також сприяло оптимізації робочого процесу. Наприклад, відображення радіусів дії атак чи траєкторій руху дозволяло швидко

виявляти помилки в логіці без необхідності запускати повноцінне тестування гри. Це економило час і забезпечувало точність реалізації таких механік, як атака чи відштовхування.

Використання Gizmos у грі підкреслило її цінність як інструмента для підвищення якості розробки. Він не лише спрощував відлагодження, але й дозволяло створювати більш інтуїтивно зрозумілі та стабільні ігрові механіки, що позитивно позначилося на загальному досвіді.

Visual Studio пропонує можливість встановити точки зупинки та запустити програму поетапно. Таким чином мені вдалося зупинити гру в певному місці і побачити значення змінної в певний момент часу. Це було особливо корисно при роботі з умовними структурами, які виконуються лише в певних ситуаціях.

У деяких випадках інформація про поточний стан персонажа або ворога виводилась безпосередньо на екран у вигляді тексту – це давало змогу бачити потрібні дані навіть без перегляду консолі. Такий підхід використовувався під час перевірки станів ворогів: «Idle», «Chasing»,«Attacking»,«Dead».

Під час огляду однієї з сцен було виявлено, що противник не перейшов на гравця після появи. Усі параметри інспектора виглядали правильно, але поведінка була несподіваною. Метод налагодження виявився нижче умов, за яких ворог почав рухатися. Ми виявили, що перевірка доступності або відстані до цілі не пройшла. Для вирішення цього було додано чек, і лише після того, як вона призначить позицію для переміщення.

Ще однією помилкою було те, що ворог розвертався у протилежний бік від гравця. Атака візуально не виконувалась, але технічно вона проводилась, що виглядало неприродно. Перевірка координат і напрямку допомогла знайти проблему – логіка розвороту спрацьовувала некоректно при мінімальній різниці в координатах (ліст. 3.1). Було переписано блок обчислення орієнтації і після цього рух ворога виглядав значно краще. При реалізації використовувались рекомендації з офіційного керівництва Microsoft C# Guide [8]. Подібна практика організації структури проекту спостерігалась у іграх на GitHub [9].

Лістинг 3.1 – Напрямок ворога до гравця

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

кінець лістингу 3.1

Проводячи битву з ворожим гравцем, було виявлено, що ворог атакував майже кілька разів поспіль. Це ускладнило уривки і зробило гру незбалансованою. Це питання було вирішено, запровадивши затримки між атаками. Потім логіка противника була передбачена для гравців та ярмарків.

Тести інтерфейсу показали, що деякі елементи були переміщені або заблоковані на екрані з різними поділами. Це було вирішено шляхом масштабування налаштувань полотна та розміру екрана. Крім того, інтерфейс був перевірений вручну з деякими загальними розмірами екрана.

Для діагностики проблем під час налагодження гри були використані різні підходи: обмін повідомленнями, візуальні підказки, інтерпретація коду кроку та перевірка логіки стану. Це вдалося усунути багато помилок, які впливають на ігровий процес та враження користувача.

3.3 Структура ігрових ассетів та програмного коду

Графічне оформлення 2D-ігор значною мірою грунтується на використанні спрайтів – двовимірних зображень, які відображаються на екрані як візуальні представники ігрових об’єктів. У грі спрайти стали основою для створення світу [10], персонажів, ворогів, інтерфейсу, предметів і навіть частини ефектів. Їх правильне використання не лише створює естетичне враження, а й відіграє важливу роль у функціональній логіці гри.

Кожен елемент у грі, що має візуальне втілення, реалізовано за допомогою одного або кількох спрайтів. Ці спрайти поділяються на кілька категорій за

призначенням та способом використання. Найпростішими є статичні спрайти, які не змінюють свого вигляду з часом. Вони використовуються для елементів фону: рослинності, предметів ландшафту, архітектури. Такі об’єкти зазвичай не мають жодної взаємодії з гравцем і слугують для формування атмосфери світу.

Інша велика категорія – анімовані спрайти. Вони використовуються для гравця та ворогів, а також для об’єктів, які мають стан зміни, наприклад, порталів. Анімація досягається шляхом швидкої послідовної зміни окремих зображень. Це забезпечує ілюзію руху або дії. У Unity цей механізм реалізується через компоненти Animator та Animation Clip, які об’єднують послідовності спрайтів у циклічні чи тригерні анімації (рис. 3.5). Наприклад, для гравця створено три основні анімаційні стани: спокій, ходьба та смерть. Кожен із них має свою власну послідовність спрайтів, а перемикання між ними відбувається автоматично залежно від стану персонажа у грі.

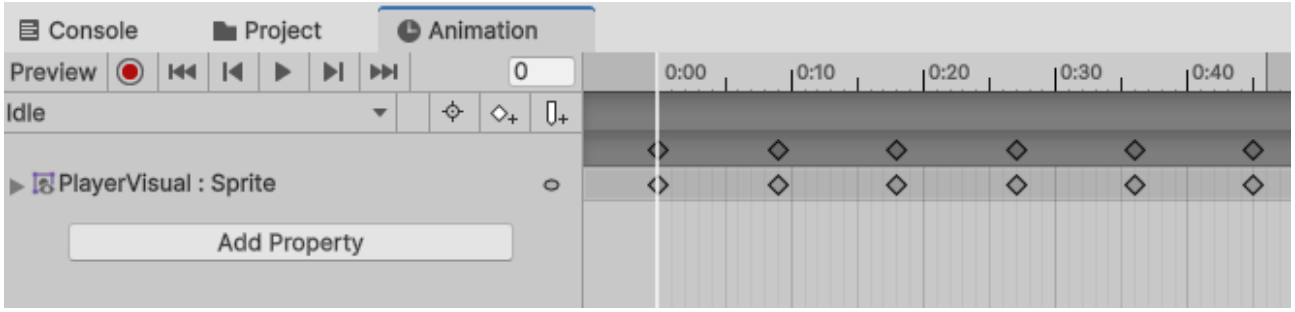

Рисунок 3.5 – Налаштування анімації для PlayerVisual

Не всі персонажі в грі використовують один і той самий підхід до анімації. Якщо гравець і скелет працюють через систему Animator, то слайм був реалізований з використанням змін кольору та матеріалів. Його анімація – це візуальні ефекти, що змінюють вигляд спрайта в реальному часі без використання традиційних кадрів. Це дозволяє створити ефект живого слизу без додаткового навантаження на систему анімацій і без необхідності створювати десятки окремих спрайтів. Такий підхід виявився ефективним для об’єктів з простою візуальною поведінкою.

Важливою особливістю спрайтів є також їх участь у формуванні інтерфейсу. Кожен елемент – це спрайт, який працює в межах системи Canvas. Тут вони використовуються через компоненти Image, які дозволяють масштабування, зміну кольору при взаємодії та адаптацію під різні роздільності екрана.

Усі об’єкти, що мають спрайт, супроводжуються рядом технічних компонентів. Найперший – це SpriteRenderer, який відповідає за відображення графіки в сцені. Через нього задається спрайт, кольорове відтінювання, прозорість, порядок відображення відносно інших об’єктів. Щоб спрайт був інтерактивним або мав фізичні властивості, до нього додається компонент Rigidbody2D – він дозволяє об’єкту взаємодіяти з іншими через фізику Unity. А BoxCollider2D визначає форму зіткнення, що важливо для обробки атак або блокування руху.

В анімованих персонажах зазвичай також присутній Animator, який поєднаний з контролером станів. Крім того були додані спеціальні скрипти, такі як «Player», «EnemyAI», «EnemyHealth», які керують поведінкою об’єкта, обробляють отримання шкоди, пересування, атаку.

Графічні ресурси були імпортовані у форматі PNG, що підтримує прозорість. При імпорті кожен спрайт вручну обрізався для зменшення обсягу пам’яті.

Візуальне сприйняття гри значною мірою залежить саме від спрайтів. Завдяки використанню чіткої художньої стилістики, а саме піксельної графіки, було досягнуто цілісного вигляду, де кожен елемент виглядає доречно і не вибивається з загального стилю. Колірна палітра, товщина ліній, розмір пікселів – усе це підбиралося так, щоб гра виглядала узгоджено як під час руху, так і в моменти взаємодії чи зупинки.

Роль спрайтів у грі неможливо переоцінити. Вони не лише зовнішність об’єктів, а й частина механіки: візуально показують стан здоров’я, виконання дії, напрямок руху, активацію об’єкта. Правильне використання спрайтів, а також

чітка система додаткових компонентів дозволяють створювати складні системи, зберігаючи їх керованими, логічно структурованими та візуально привабливими.

Чітко продумана й структурована організація проекту є однією з основ ефективної розробки гри, її тестування, підтримки та масштабування. У процесі створення гри особливу увагу було приділено саме модульності – як у розміщенні ресурсів, так і в структурі програмного коду. Завдяки такому підходу кожен елемент проекту зберігається в окремому логічному блоці, що значно спрощує навігацію по файлах, повторне використання елементів, внесення змін та пошук помилок. В результаті структура проекту стала не лише зручною, а й гнучкою до змін та масштабування в майбутньому.

Ресурси гри були розподілені за окремими папками (рис. 3.6), назви яких чітко відповідають їхньому вмісту. В папці Animations зберігаються всі анімаційні кліпи та контролери, які відповідають за візуальну поведінку гравця, ворогів і об’єктів середовища. Prefabs містить заздалегідь підготовлені шаблони об’єктів, що використовуються повторно на різних сценах. Це, зокрема, сам гравець, різні види ворогів, предмети, точки переходу між локаціями.

Рисунок 3.6 – Повна структура папок Assets

Всі сцени гри зберігаються в папці Scenes – до них належать ігрові рівні та меню. Папка Scripts містить програмну логіку проекту, при цьому вона розбита на підтеми: управління гравцем, ворогами, обробка інтерфейсу, глобальні системи керування та допоміжні утиліти. Графічні елементи, такі як спрайти персонажів, тайли середовища, зібрані в окремій папці Sprites. Для побудови

карт і ландшафтів використано систему Tilemap, відповідно всі плитки та палітри зберігаються у Tilemap. Матеріали для поверхонь та ефектів – зокрема, тих, що застосовуються до слайма – розміщено у папці Materials.

Завдяки такій структурі любий розробник легко орієнтується у вмісті проекту. Якщо виникає потреба змінити анімацію, додати новий предмет чи оновити логіку ворога – це можна зробити локально, без ризику ненавмисно порушити інші частини гри.

Кодова база гри реалізована згідно з принципами об’єктно-орієнтованого програмування. Основна логіка розбита на окремі класи, кожен з яких виконує конкретну функцію. Наприклад, клас, що відповідає за гравця, включає методи руху, атаки, отримання шкоди, визначення смерті та події взаємодії з іншими об’єктами. Вороги керуються класом «EnemyAI», який реалізує поведінкові стани: переслідування, атака, повернення на місце тощо. Також у грі реалізовано низку глобальних систем – зокрема, «GameManager», «GameInput» – які керують основними процесами на всіх сценах. Усі вони реалізовані за шаблоном Singleton, що дозволяє забезпечити доступ до них з будь-якого місця в проекті.

Механіки, які використовуються кількома об’єктами, винесено в окремі скрипти. Наприклад, відштовхування при отриманні шкоди реалізовано в компоненті «KnockBack.cs» – він прикріплюється до будь-якого об’єкта, що повинен мати відповідну поведінку. Таким чином, логіка не дублюється, а повторно використовується в різних ситуаціях, що зменшує кількість коду і підвищує його підтримуваність.

Окремо варто згадати про системи автоматизації. Наприклад, завантаження сцен відбувається за допомогою спеціального менеджера сцен, який дозволяє динамічно переключатися між локаціями на основі подій гри або взаємодії з об’єктами. Це дає можливість легко додавати нові сцени – достатньо лише зареєструвати їх в загальному списку та створити тригер для переходу.

Загальна архітектура проекту дозволяє не лише ефективно підтримувати наявний функціонал, але й безперешкодно додавати нові модулі: нові види ворогів, нові типи атаки та рівнів. Вся структура побудована так, щоб

забезпечити максимальну адаптивність без необхідності глобального переписування коду або зміни архітектури.

Таким чином, структура ассетів і програмного коду гри відповідає найкращим практикам розробки 2D-ігор у середовищі Unity. Вона забезпечує простоту керування ресурсами, ефективність тестування, легкість масштабування і високу якість підтримки проекту як на етапі розробки, так і після релізу.

Усі скрипти, що реалізують логіку гри, були написані мовою програмування C# у середовищі Visual Studio, яке є стандартним інструментом для розробки проектів на Unity. Це середовище забезпечує зручну інтеграцію з Unity Editor, підтримує інтелектуальну підсвітку синтаксису, автоматичне завершення коду (IntelliSense), зручну навігацію між класами та методами, а також можливість відлагодження в реальному часі. Вибір C# як основної мови програмування обумовлений її глибокою інтеграцією з рушієм Unity та широкими можливостями для об’єктно-орієнтованої розробки. Усі класи та компоненти створювались як окремі скрипти, кожен із яких мав свою функціональність і чітко визначену відповідальність.

Під час написання коду дотримувались базових принципів структурованого програмування: поділ відповідальностей, повторне використання логіки, розділення логіки від представлення (UI), використання подій та делегатів для взаємодії між компонентами. Скрипти були організовані в тематичні папки, що відповідали логіці їх використання.

Крім того, завдяки можливості підключення Visual Studio до Unity, розробка була більш ефективною: усі помилки відображались безпосередньо в редакторі, а за допомогою точок зупину можна було перевірити, як саме працює кожна функція під час гри. Це значно прискорило процес тестування та пошуку помилок, особливо при реалізації складніших механік, таких як бойова система або поведінку ворогів.

ВИСНОВКИ

У межах виконання кваліфікаційної роботи бакалавра було успішно реалізовано повноцінну 2D гру «Fragments of the World» у жанрі Adventure-RPG. Проект розроблявся із використанням сучасного інструментарію Unity та мови програмування C#. Створена гра поєднує в собі механіки дослідження, бою, переходів між сценами, взаємодії з об’єктами, а також підтримує модульну структуру, що дозволяє її легко масштабувати й доповнювати.

Під час аналізу предметної області було обґрунтовано актуальність проекту: незалежна розробка 2D-ігор набула широкої популярності, зокрема в освітньому та інді-сегменті. Гра розроблена з урахуванням простоти освоєння для гравців і водночас включає основні RPG-елементи, що робить її придатною для подальшого розвитку. Окремо було зазначено, що проект може бути використаний як розвивальний ігровий інструмент для дітей-переселенців – у якості способу адаптації та відволікання через інтерактивну взаємодію.

Проведення технічного аналізу дало змогу визначити функціональні та нефункціональні вимоги до гри, побудувати архітектуру, логіку взаємодії компонентів, описати поведінку ворогів, механіки бою, руху та взаємодії з інтерфейсом. Розроблено відповідні UML-діаграми, які відображають ключові взаємозв’язки між елементами коду та ігровими об’єктами.

Аргументовано вибір технологій: рушій Unity було обрано за його гнучкість, підтримку 2D-анімації, тайлових карт, а також широку екосистему. Мова C# забезпечила стабільність та високу читабельність коду. Розробка велась у середовищі Visual Studio, що дало можливість ефективно організувати проектну структуру та налагоджувати код.

Було реалізовано всі основні компоненти гри: рух і управління персонажем, бойову систему, ігрові стани, UI-інтерфейс, а також анімації для гравця. Розроблено окрему логіку для анімованих об’єктів, що використовують як стандартні засоби Animator, так і зміну матеріалів.

У процесі реалізації було повністю створено логіку поведінки ворогів, що охоплює кілька станів: патрулювання, переслідування гравця, атака та відслідковування дистанції. Поведінка супротивників реалізована за допомогою системи станів, яка дозволяє динамічно змінювати дії ворога відповідно до контексту гри. Було створено механізм обробки переходів між станами на основі відстані до гравця та таймерів, що забезпечує плавність реакції. Крім того, кожен стан супроводжується відповідною анімацією, яка активується через Animator Controller, завдяки чому візуальна поведінка супротивників є послідовною та зрозумілою для гравця. Такий підхід дозволив забезпечити більш реалістичну взаємодію з противниками й підвищити загальну динаміку геймплею.

Було створено повноцінну ігрову локацію за допомогою Tilemap, у якій використано окремі палети для трави, землі, води, стін і декоративних елементів. Реалізовано стартову зону, оточення та межі карти, що обмежують переміщення гравця за межі ігрового простору.

У грі реалізовано повноцінну систему сцен, що включає головне меню та ігрові рівні. Перехід між сценами здійснюється плавно завдяки використанню спеціальних тригерів і завантаження сцен через менеджер, що забезпечує безперервність ігрового процесу.

Процес тестування охопив ручне функціональне тестування, перевірку переходів між локаціями, логіку ворогів, атаки, а також коректність інтерфейсу. Виявлені помилки – зокрема, проблеми з пересуванням ворогів або некоректною орієнтацією – були усунені завдяки налагодженню, тестам у редакторі та змінам логіки. Важливу роль у цьому процесі відіграло поступове впровадження перевірок і адаптивних механізмів реагування.

Візуальна складова гри базується на піксельних спрайтах, які структуровано в анімаційні послідовності, тайлові палітри та окремі компоненти інтерфейсу. Було використано кілька типів спрайтів: статичні, анімовані, UI, а також технічні для взаємодії між об’єктами. Графіка адаптована під різні роздільності екрана, що робить гру універсальною для настільних пристроїв.

Результати роботи повністю відповідають вимогам, сформульованим у першому розділі. Реалізовано структуру сцени, основну логіку гравця, поведінку ворогів, графіку, UI, а також налаштування анімації та переходів між локаціями. Усе програмне забезпечення було створено з нуля, унікально для цього проекту, без використання сторонніх шаблонів.

Гра має потенціал для подальшого розширення: реалізації системи діалогів, інвентаря, квестів або підтримки мобільної платформи. Крім того, створений проект може бути використаний як демонстраційний матеріал на курсах з Unity або програмування ігор, а також як стартова платформа для нових студентських ігор.

Отже, поставлені завдання кваліфікаційної роботи виконано повністю. Гра є завершеним, працездатним продуктом, що поєднує ігрову логіку, інтерактивність та візуальне оформлення і має перспективу подальшого розвитку як навчального, так і розважального програмного продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Technologies. URL: https://docs.unity3d.com/ScriptReference/ (дата звернення: 04.02.2025).

2. Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 04.02.2025).

3. itch.io. URL: https://itch.io/game-assets/tag-pixel-art (дата звернення: 10.02.2025).

4. GameDev. URL: https://www.gamedev.tv/p/unity-2d-game-development (дата звернення: 14.02.2025).

5. Unity Asset Store. URL: https://assetstore.unity.com/ (дата звернення: 28.02.2025).

6. Catlike Coding. URL: https://catlikecoding.com/unity/tutorials/ (дата звернення: 10.05.2025).

7. Raywenderlich.com. Unity 2D Tutorials & Resources. URL: https://www.raywenderlich.com/unity (дата звернення: 04.03.2025).

8. Microsoft Learn. C# Guide. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 01.04.2025).

9. GitHub. URL: https://github.com/search?q=unity+2d+game&type (дата звернення: 04.05.2025).

10. Pixel Game Art. URL: https://kenney.nl/assets (дата звернення: 05.03.2025).

11. Microsoft Visual Studio. URL: https://visualstudio.microsoft.com (дата звернення: 10.02.2025).

12. Unity Hub. URL: https://unity.com/download (дата звернення: 10.02.2025).

13. Gamasutra. URL: https://www.gamedeveloper.com (дата звернення: 21.04.2025).

14. Smith J. Game Programming Patterns, Genever Benning, 1 edition, 2021. 354 p.

15. GameDev StackExchange. Unity 2D Questions. URL: https://gamedev.stackexchange.com/questions/tagged/unity2d (дата звернення: 23.04.2025).

16. Red Blob Games. Game development algorithms and mechanics. URL: https://www.redblobgames.com/ (дата звернення: 25.04.2025).

17. Raywenderlich. Unity 2D Tutorials & Resources. URL: https://www.raywenderlich.com/unity (дата звернення: 28.04.2025).