Галицький фаховий коледж імені В’ячеслава Чорновола
відділення комп’ютерних технологій
циклова комісія інформатики та комп’ютерних дисциплін

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач відділення
комп’ютерних технологій
Наталія СТЕФУРАК /_____/
підпис
«___» _____ 202_ р.

# ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

освітньо-професійного ступеня «фаховий молодший бакалавр»

зі спеціальності 122 «Комп’ютерні науки»

на тему: «Гра в жанрі roguelike з процедурною генерацією рівнів»

| | | |
|---|---|---|
| Студент групи КН-41 | Михайло МИКИТОВИЧ | _____ (підпис) |
| Керівник роботи | Наталя КУЛЬЧИНСЬКА | _____ (підпис) |
| Консультанти: з техніко-економічного обґрунтування | Любов МЕЛЕНЧУК | _____ (підпис) |
| нормоконтролер | Оксана СИРОТЮК | _____ (підпис) |

Тернопіль – 2025

Галицький фаховий коледж імені В’ячеслава Чорновола

відділення комп’ютерних технологій

циклова комісія інформатики та комп’ютерних дисциплін

ЗАТВЕРДЖУЮ

Завідувач відділення

комп’ютерних технологій

Наталія СТЕФУРАК /_____/

підпис

«___» _____ 202_ р.

ЗАВДАННЯ

на кваліфікаційну роботу

на здобуття освітньо-професійного ступеня «фаховий молодший бакалавр»

студенту Микитовичу Михайлу Петровичу

(прізвище, ім’я та по-батькові студента)

1. Тема роботи Гра в жанрі roguelike з процедурною генерацією рівнів

затверджена наказом по коледжу від «25» листопада 2024 р., № 253а-н

2. Термін здачі студентом завершеної роботи «___» _____202_ р.

3. Вихідні дані до роботи Результати дослідження алгоритмів процедурної генерації у іграх.

4. Перелік питань, які повинні бути розроблені:

а) основна частина Аналіз предметної області. Аналіз вимог та постановка завдання. Проєктування, реалізація та тестування ігрового застосунку.

б) техніко-економічне обґрунтування Проведення розрахунків прибутковості проєкту. Аналіз ситуації на ринку відеоігор.

5. Перелік графічного матеріалу Структурна схема гри. Діаграма послідовностей. Діаграма прецедентів. Макети інтерфейсу.

| Розділ | Консультанти | Підпис, дата | |
|---|---|---|---|
| | | Завдання видано | Завдання прийнято |
| З техніко-економічного обґрунтування | Меленчук Любов Іванівна | | |

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

| № п/п | Найменування етапу | Терміни | |
|---|---|---|---|
| | | початку | завершення |
| 1 | Вибір теми кваліфікаційної роботи | 21.11.2024 | |
| 2 | Аналіз програмних рішень | 25.11.2024 | 06.12.2024 |
| 3 | Вивчення технологій реалізації роботи | 20.01.2025 | 28.02.2025 |
| 4 | Проєктування архітектури ігрового застосунку | 05.03.2025 | 26.03.2025 |
| 5 | Створення системи процедурної генерації рівнів | 31.03.2025 | 11.04.2025 |
| 6 | Реалізація ігрового застосунку | 31.03.2025 | 09.05.2025 |
| 7 | Тестування та оптимізація гри | 29.04.2025 | 20.05.2025 |
| 8 | Аналіз економічної частини | 14.04.2025 | 16.05.2025 |
| 9 | Оформлення пояснювальної записки | 04.06.2025 | 12.06.2025 |
| 10 | Попередній захист кваліфікаційної роботи | 13.06.2025 | |
| 11 | Підготовка до захисту кваліфікаційної роботи | 14.06.2025 | 24.06.2025 |
| 12 | Захист кваліфікаційної роботи | 25.06.2025 | |

Дата видачі «___» _____ 2024 р. Керівник _____/ Кульчинська Н. З.

Завдання прийняв до виконання _____/ Микитович М. П.

# Реферат

Кваліфікаційна робота. Гра в жанрі roguelike з процедурною генерацією рівнів. Микитович Михайло Петрович. Галицький фаховий коледж імені В’ячеслава Чорновола, відділення комп’ютерних технологій. 63 сторінки, 35 рисунків, 12 додатків, 10 джерел.

Об’єкт дослідження – процедурна генерація та її використання в іграх.

Метою роботи є створення ігрового застосунку в жанрі roguelike, рівні якого згенеровані за допомогою алгоритмів процедурної генерації, використовуючи такі засоби як: ігровий рушій Unity, мова програмування C#, середовище розробки Visual Studio, Photoshop та Aseprite.

Ігровий застосунок повинен забезпечити унікальне ігрове середовище для кожного проходження, підтримуючи різноманітність кімнат, ворогів та предметів, що змінюються в залежності від випадково згенерованого рівня. Очікується реалізація повноцінної системи бою з використанням снарядів, здібностей гравця та артефактів, які змінюють ігровий процес. Гравець має взаємодіяти з ворогами та союзниками, що реагують на дії гравця.

Результатом розробки стала функціональна 2D roguelike гра, що включає процедурну генерацію рівнів із різними типами кімнат, бойову систему з використанням артефактів і здібностей, взаємодію з ворогами та союзниками, адаптивний інтерфейс користувача, а також можливість масштабування шляхом додавання нових ігрових механік чи об’єктів.

ІГРОВИЙ ЗАСТОСУНОК, C#, UNITY, АЛГОРИТМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ, ROGUELIKE.

# Abstract

Qualification Work. A game in the roguelike genre with procedural level generation. Mykytovych Mykhailo Petrovych. Halytskyi Professional College named after Vyacheslav Chornovil, Department of Computer Technologies. 63 pages, 35 figures, 12 appendices, 10 sources.

Object of study: procedural generation and its use in games.

The purpose of the work is to develop a game application in the roguelike genre, with levels generated using procedural generation algorithms. The following tools were used: Unity game engine, C# programming language, Visual Studio development environment, Photoshop, and Aseprite.

The game application is intended to provide a unique gameplay experience for each playthrough, supporting a variety of rooms, enemies, and items that change based on randomly generated levels. A fully functional combat system is expected to be implemented, featuring projectiles, player abilities, and artifacts that alter gameplay. The player should interact with both enemies and allies that respond dynamically to the player's actions.

As a result, a functional 2D roguelike game was developed, featuring procedural level generation with various room types, a combat system using artifacts and abilities, interaction with enemies and allies, an adaptive user interface, and scalability through the addition of new game mechanics or objects.

GAME APPLICATION, C#, UNITY, PROCEDURAL GENERATION ALGORITHMS, ROGUELIKE.

ЗМІСТ

# СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ЄСВ – Єдиний соціальний внесок.

ANSI – American National Standards Institute.

ASCII – American Standard Code for Information Interchange.

ESA – Entertainment Software Association.

HP – Health Points.

HUD – Head-Up Display.

MonoBehaviour – Базовий клас Unity для скриптів, що працюють у грі.

PNG – Portable Network Graphics.

RPG – Role-Playing Game.

ScriptableObject – Спеціальний тип об’єкта в Unity для зберігання даних.

UE4 – Unreal Engine 4.

UI – User Interface.

UML – Unified Modeling Language.

UVC – Unity Version Control.

UX – User Experience.

VPS – Virtual Private Server.

# ВСТУП

Динамічний розвиток індустрії відеоігор, зокрема інді-сегменту, стимулює появу нових ігрових проєктів. Серед них особливу популярність здобувають ігри жанру roguelike, що поєднують елементи випадковості, динамічних боїв та високого рівня реіграбельності. Такі ігри вирізняються процедурною генерацією рівнів, яка забезпечує унікальність кожного проходження та значно підвищує інтерес користувачів до геймплею. Актуальність теми кваліфікаційної роботи обумовлена зростаючим попитом на інноваційні ігрові рішення, які можуть забезпечити тривалий користувацький інтерес при мінімальних витратах на створення контенту.

Мета кваліфікаційної роботи полягає в розробці 2D roguelike гри з виглядом зверху, що поєднує процедурну генерацію рівнів, динамічну бойову систему, гнучке управління статус-ефектами та систему об’єктів з різними типами поведінки. Для досягнення поставленої мети необхідно створити компонентно-орієнтовану архітектуру гри, що забезпечить модульність та масштабованість системи, реалізувати алгоритми процедурної генерації рівнів та забезпечити оптимізацію продуктивності для гравців.

Отриманий проєкт можна розвивати в як повноцінний комерційний ігровий продукт. Модульна архітектура та гнучкі системи, розроблені в рамках кваліфікаційної роботи, створять міцну основу для масштабування проєкту до рівня інді-гри з можливістю публікації на провідних ігрових платформах.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ

## 1.1 Опис предметної області

У сучасному світі популярність сфери розваг стрімко зросла, у свою чергу масовий розвиток комп'ютерних технологій та поширення доступу до інтернету серед населення, зробили можливим скористатися нею практично для кожного.

Незважаючи на популярний міф, комп'ютерні ігри не є розвагою лише для дітей та підлітків, багато людей дорослого віку також доволі часто грають у відеоігри. Згідно з опитуванням, яке провели ESA та опублікували у своєму річному звіті «Основні факти про індустрію відеоігор США», станом на 2023 рік лише 26% з усіх геймерів є неповнолітніми (рис. 1.1).

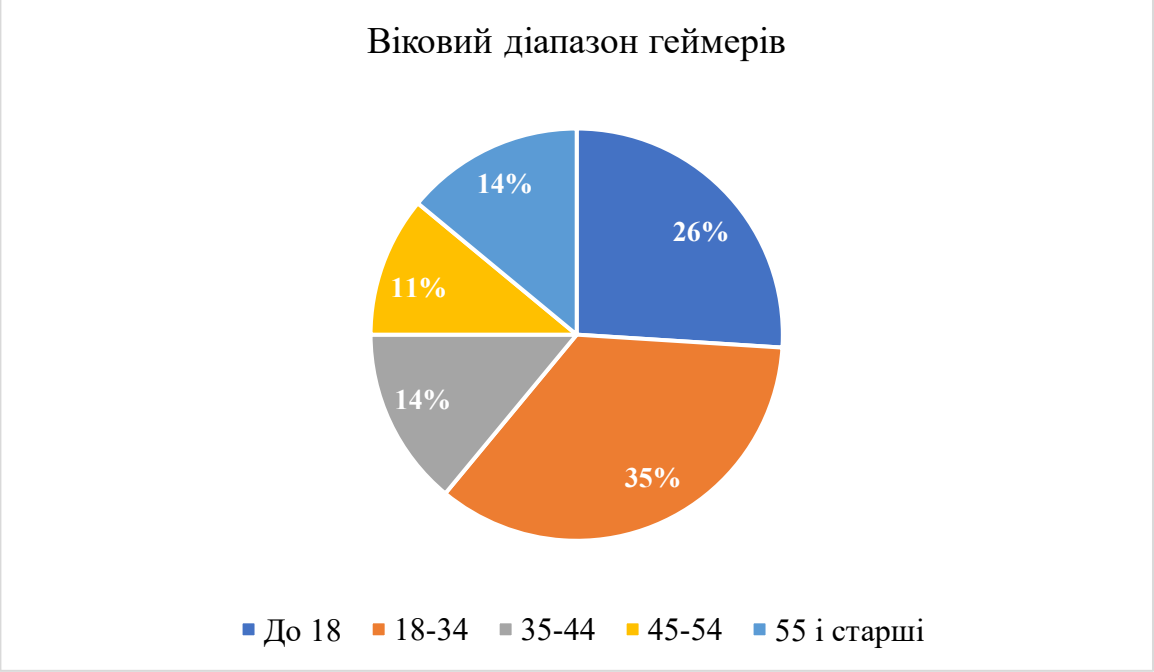


Рисунок 1.1 – Вікова діаграма серед геймерів

А отже ігри перестали бути розвагою лише для дітей, а стали важливим культурним елементом суспільства, доступним для усіх вікових категорій [1].

Комп'ютерні ігри, які колись були нішевим хобі для обмеженого кола ентузіастів, перетворилися на глобальну індустрію, що щороку приносить

мільярди доларів прибутку. Вони стали не лише розвагою, а й потужним інструментом для навчання, соціальної взаємодії та мистецького вираження.

Сьогодні ігри охоплюють безліч жанрів і форматів: від масштабних багатокористувацьких онлайн-ігор створених великими корпораціями накшталт Microsoft чи Tencent до невеликих незалежних проектів, які розробляються невеликими командами або навіть однією людиною. Кожен жанр має свої унікальні особливості та аудиторію, але всі вони об'єднуються спільним бажанням забезпечити гравцям незабутній досвід. Серед цих жанрів особливе місце займає roguelike, який, незважаючи на свою нішевість, знайшов мільйони прихильників по всьому світу завдяки своїй унікальній механіці та глибині.

Основним жанром, до якого належить roguelike, є екшн-RPG (Action Role-Playing Game). Екшн-RPG поєднує в собі елементи рольових ігор, такі як розвиток персонажа, виконання завдань та дослідження світу, з динамічною бойовою системою, яка вимагає від гравця швидкої реакції та стратегічного мислення [2].

На відміну від класичних RPG, де бойова система часто базується на покроковій механіці, екшн-RPG забезпечує безперервний бій, де кожен удар, блок або ухилення залежить від майстерності гравця. Це робить ігри більш інтенсивними та захоплюючими.

У цих іграх гравець не лише розвиває свого персонажа, але й активно взаємодіє з навколишнім світом, вирішуючи головоломки, знаходячи секрети та приймаючи моральні рішення, які впливають на сюжет.

Геймплей таких ігор зазвичай динамічний, вимагає швидкої реакції та тактичного мислення, а також передбачає використання різноманітної зброї, умінь і механік ухилення від атак супротивників.

Крім того, у багатьох екшн-RPG значну роль відіграє дослідження світу та пошук корисних ресурсів, що дозволяє гравцю адаптуватися до нових викликів і створювати унікальні стратегії проходження.

Специфікою жанру roguelike є кілька ключових елементів, які роблять його унікальним та викликають інтерес у гравців. До них належать: процедурна генерація рівнів, висока складність, перманентна смерть персонажа (так звана permadeath) та випадкові події, які разом забезпечують те, що кожне проходження гри стає унікальним досвідом [3].

Процедурна генерація рівнів є однією з найважливіших особливостей жанру. Завдяки цій механіці кожен рівень створюється випадковим чином під час кожного нового запуску гри.

Це означає, що гравець ніколи не зіткнеться з однаковим розташуванням кімнат, коридорів, ворогів чи предметів.

Така система змушує гравця постійно адаптуватися до нових умов, вивчати розташування ворогів та об’єктів, а також експериментувати з різними тактиками. Це робить кожну сесію гри неповторною та спонукає до творчого підходу до проходження.

Висока складність є характерною рисою roguelike, вона досягається за рахунок комбінації різних факторів: численних та різноманітних ворогів, обмежених ресурсів (таких як здоров’я, зброя або зілля), а також необхідності приймати швидкі, але зважені рішення.

Гравець повинен постійно оцінювати ризики, планувати свої дії наперед і швидко реагувати на зміни в ігровому середовищі. Це створює напружену атмосферу та забезпечує глибокий ігровий досвід.

Перманентна смерть персонажа означає, що після загибелі весь прогрес втрачається, і гравцеві доводиться починати спочатку, що додає грі додаткової напруги та заохочує до більш обережного стилю гри.

Випадкові події роблять кожне проходження ще більш унікальним. Вони можуть включати несподівані зустрічі з ворогами, знаходження рідкісних предметів або випадкові зміни в ігровому середовищі, наприклад кімната з головоломкою чи сюрпризом.

Такі події додають елемент непередбачуваності, що підтримує інтерес гравця та стимулює його до повторних спроб.

Жанр виник у 80-х роках з виходом гри «Rogue», що заклала основи для подальших розробок у цьому напрямку. Одні з перших ігор rouglike створювалися у середовищах без графічного інтерфейсу, використовуючи ANSI або ASCII символи (рис. 1.2).


Рисунок 1.2 – Приклад найпростішої rouguelike гри

Умовно рівень поділяється на закриту (недосліджену) частину та вже досліджену частину. Кожен символ має своє значення: стіна, двері, ворог, гравець, золото тощо.

Простота дизайну, разом із цікавими механіками гри, дозволила roguelike стати основою для багатьох сучасних ігор, які розвинули ідеї, закладені ще у минулому тисячолітті.

1.2 Аналіз наявних рішень

Жанр roguelike за останні десятиліття зазнав значних змін, еволюціонуючи від простих текстових ігор до складних проектів із багатим візуальним та ігровим дизайном, а також багатомільйонною базою шанувальників. Проте, незважаючи на різноманітність сучасних рішень, багато ігор зберігають ключові елементи, такі як процедурна генерація, перманентна смерть та висока складність.

Прикладом ігор цього жанру є серія ігор Hades, розробником якої є студія з США Supergiant Games [4]. Приклад геймплею Hades наведено на рисунку 1.3

Рисунок 1.3 – Кадри з гри Hades II

Кожне проходження гри Hades унікальне, оскільки гравець долає низку кімнат, послідовність яких щоразу змінюється.

Випадковим чином визначаються не тільки самі кімнати, а й вороги, що там з’являються, і також нагороди, які можна отримати. Гра містить чотири різні області підземного світу, і в кожній з них на гравця чекають нові випробування та противники.

Однією з ключових особливостей Hades є система бойових умінь, яка дозволяє гравцю вибирати різні зброї та покращувати їх за допомогою богів Олімпу.

Після зачистки кімнати гравець може отримати благословення від одного з богів грецького пантеону (рис. 1.4).


Рисунок 1.4 – Вибір посилення в Hades II

Кожен бог надає унікальні здібності, що дозволяє створювати різні стратегії проходження. Наприклад, Зевс може надати електричні атаки, а Афродіта зосередитися на ослабленні ворогів.

Вибір дару реалізовано у вигляді карточок з посиленнями, гравцю потрібно обрати бажане покращення ігрових характеристик персонажа, не лише змінюючи цифри шкоди, а й дозволяючи відкрити нові унікальні здібності, правильна комбінація дарів є ключем до перемоги, гра дозволяє підбирати різні комбінації дарів, в залежності від ігрових смаків користувача, саме ця система робить кожен запуск гри унікальним і спонукає до експериментів.

Іншим популярним представником roguelike є кросплатформена гра Dead Cells авторства французької компанії Motion Twin [5].

Ігровий процес включає в себе збір скарбів та нової зброї з метою посилення персонажа, паралельно перемагаючи ворогів (рис. 1.5).

Якщо рівень здоров'я персонажа опуститься до нуля, то він втратить усе зароблене під час проходження та почне рівень заново.

Рисунок 1.5 – Кадр з гри Dead Cells

В процесі просування рівнем гравець, з певним шансом, може наткнутися на різні предмети, які певним чином впливають на подальший геймплей.

Roguelike не обов'язково повинен бути основним режимом, деякі ігри використовують його як додатковий режим, щоб гравці мали заняття після проходження основного контенту.

Наприклад у грі Honkai Impact 3rd було створено режим під назвою «Elisian Realm» (рис. 1.6).


Рисунок 1.6 – Режим Elysian Realm в грі Honkai Impact 3rd

Цей режим додає унікальні навички для персонажів, доступні лише у цьому режимі, також незважаючи на лінійність рівнів, вороги в них генеруються випадково [6].

Гравець може обирати між типом кімнат, які він хоче пройти, в залежності від типу змінюються нагороди (рис. 1.7).

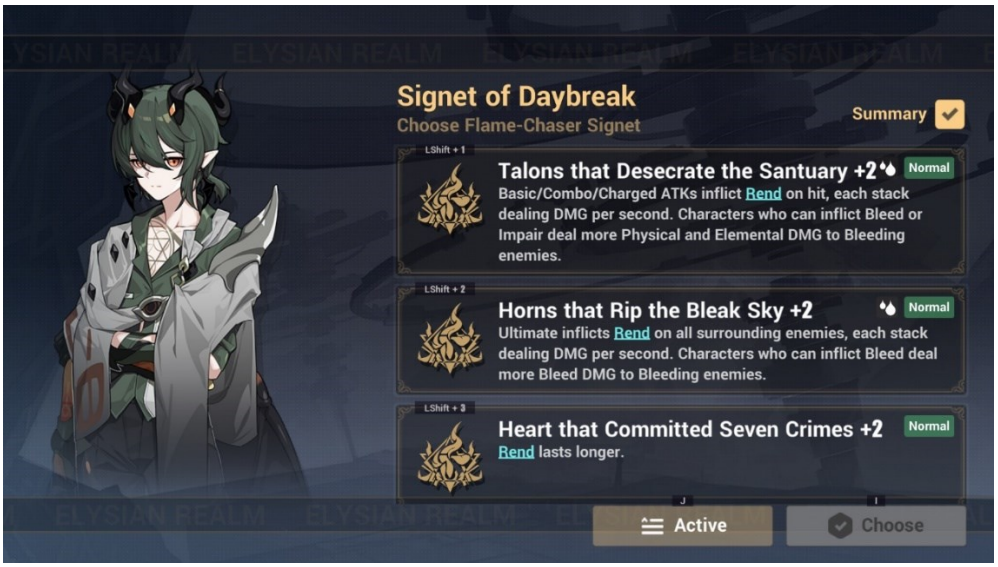


Рисунок 1.7 – Вибір між посиленнями після проходження кімнати

Усі посилення та отримані бонуси діють в межах одного проходження та зникають після завершення рівня.

Від попередніх ігор «Elisium Realm» відрізняється тим, що предмети, які кардинально змінюють геймплей персонажа, обираються на початку

проходження, користувач обирає «Ядро» персонажа, в залежності від обраного ядра ігровий процес може кардинально змінитися.

### 1.3 Аналіз вимог до програмного засобу та постановка завдання

Базуючись на аналізі уже існуючих рішень у предметній області, для створення ігрової програми в жанрі roguelike необхідно реалізувати такі аспекти як: випадкова генерація рівня, а також ворогів і нагород у них, дотримання достатнього рівня складності.

Розробка комп'ютерних ігор є складним і багатокомпонентним процесом, що вимагає врахування як функціональних, так і нефункціональних вимог. У випадку розробки roguelike-гри особливу увагу слід приділити процедурній генерації рівнів, динаміці ігрового процесу та забезпеченню стабільної роботи програми на різних апаратних платформах.

Головною концепцією цієї гри є варіативність та реіграбельність. Завдяки процедурній генерації кожне проходження має бути унікальним, що стимулюватиме гравця повертатися до гри знову і знову. Важливою вимогою є баланс складності: гра повинна бути достатньо складною, щоб кидати виклик користувачеві.

Інтерфейс має бути мінімалістичним, але інтуїтивно зрозумілим, забезпечуючи швидкий доступ до ключової інформації (рівень здоров'я, інвентар, особливі здібності тощо). Гнучкість архітектури передбачає можливість легкого розширення гри, додавання нових ворогів, предметів та механік без значних змін у коді.

Функціональні вимоги охоплюють механіки, необхідні для повноцінного ігрового процесу. Гравець повинен мати можливість керувати персонажем у реальному часі, використовуючи клавіатуру для пересування та атаки. Передбачено використання спеціальних здібностей, що додають тактичної глибини та різноманітності геймплею. Вороги в грі будуть мати різні моделі поведінки, включаючи переслідування, стрільбу або використання особливих атак. Система процедурної генерації рівнів

забезпечить унікальність кожного проходження, дотримуючись логічних правил розташування кімнат та ключових об’єктів. Крім основного геймплею, гра передбачає елементи економіки: у магазині гравець зможе купувати покращення або ресурси, що допоможуть у проходженні.

До нефункціональних вимог належить продуктивність, безпека та апаратні вимоги. Продуктивність гри повинна залишатися стабільною навіть при великій кількості активних об’єктів на екрані, що потребує оптимізації графіки та фізики.

Оскільки гра є 2D-орієнтованою, її апаратні вимоги будуть помірними, що дозволить запускати її на широкому спектрі пристроїв.

Важливо забезпечити підтримку різних дозволів екрана, включаючи як стандартні 16:9, так і більш непопулярні серед користувачів варіанти для сучасних моніторів, наприклад 16:10.

Визначення цільової аудиторії є важливим етапом, що дозволяє адаптувати гру під потреби потенційних гравців. Roguelike-ігри зазвичай приваблюють дві основні категорії гравців:

Прихильники класичних roguelike-ігор це гравці, які цінують складність, випадкові події та високий рівень варіативності. Вони очікують від гри виклику, інтуїтивного, але глибокого геймплею та можливості стратегічного планування.

Важливо також задовільнити потреби гравців, які шукають казуальні розваги, користувачі, які не мають досвіду з roguelike, але хочуть спробувати жанр у спрощеному форматі. Для них важливою є можливість швидкого освоєння гри, зрозумілий інтерфейс та система підказок. Щоб уникнути конфлікту інтересів між гравцями, які прагнуть виклику і казуальними гравцями, необхідно реалізувати систему вибору складності, таким чином кожен зможе обрати варіант для себе.

Завдяки низьким апаратним вимогам, які забезпечує простота графіки, гра буде доступною широкому колу користувачів, включаючи тих, хто має старіші пристрої.

# 2 ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

## 2.1 Опис проєктних рішень

При розробці гри важливим етапом є вибір рушія, який визначає можливості реалізації механік, продуктивність, графічні можливості та зручність розробки. Було розглянуто кілька варіантів рушіїв, зокрема Unity, Unreal Engine 4 та Cocos2d-x, кожен з яких має свої переваги та недоліки.

Unreal Engine 4 є потужним рушієм, який особливо добре підходить для створення високоякісних. Він відзначається передовими технологіями рендерингу, реалістичною фізикою та динамічним освітленням, що робить його більш орієнтованим на 3D-ігри. Одна з ключових переваг UE4 це система Blueprints, яка дозволяє швидко розробляти складні ігрові механіки. Рушій має відкритий вихідний код, що дає змогу змінити його під конкретні потреби.

Проте Unreal Engine 4 має певні недоліки. Він вимагає потужного апаратного забезпечення, що ускладнює розробку на слабких комп'ютерах. Крім того, збірки ігор на UE4 часто виходять об'ємними, що може бути проблемою для інді-ігор, які прийнято створювати невимогливими. Також рушій має вищий поріг входження порівняно з іншими рушіями, особливо для новачків. Використання C++ як основної мови програмування додає додаткові складнощі в процесі розробки, оскільки потребує ретельного керування пам'яттю та оптимізації коду. Проте, Unreal Engine 4 більше орієнтований на розробку 3D-ігор, а його 2D-функціонал менш розвинений [7].

Cocos2d-x — це легкий та швидкий рушій, орієнтований на 2D-розробку, який часто використовується для не AAA проєктів. Він є повністю відкритим, що дозволяє глибоко модифікувати його під свої потреби. Завдяки оптимізованій архітектурі Cocos2d-x забезпечує високу продуктивність на слабких пристроях, що робить його популярним серед незалежних розробників.. Великим плюсом рушія є його низьке споживання ресурсів, що дозволяє запускати ігри навіть на застарілих мобільних пристроях.

Однак у Cocos2d-x є суттєві обмеження. Він не має візуального редактора рівнів, що ускладнює розробку складних процедурно-генерованих світів, таких як у roguelike-іграх. Крім того, рушій вимагає більшої роботи з кодом, що може сповільнити процес розробки, особливо створення складних механік або інтеграцію нових функцій. Спільнота Cocos2d-x менша, ніж у Unity чи UE4, тому знайти готові рішення чи навчальні матеріали може бути важче. Відсутність зручного редактора для створення та тестування рівнів робить Cocos2d-x менш ефективним вибором для розробки ігор, де важлива швидка ітерація та гнучкість у налаштуванні рівнів [8].

Ще одним популярним та легким для освоєння інструментом є Unity, один із найпопулярніших ігрових рушіїв, який підходить як для 2D, так і для 3D-ігор. Він пропонує гнучкі інструменти для створення анімацій, фізики та процедурної генерації рівнів. Основною мовою програмування є C#, яка є відносно простою для вивчення та має велику підтримку спільноти. Unity також відзначається крос-платформенністю, дозволяючи експортувати гру на ПК, мобільні пристрої та консолі з мінімальними змінами коду [9].

Однією з ключових переваг Unity є його вбудований редактор, який дозволяє розробникам створювати рівні, розміщувати об’єкти та тестувати зміни в режимі реального часу. Це значно прискорює розробку та спрощує роботу з контентом. Крім того, Unity має велику бібліотеку Asset Store, де можна знайти готові моделі, звуки, анімації та скрипти, що допомагає скоротити час на розробку гри. Також, використання системи контролю версій Unity Version Control спрощує процес управління змінами в проєкті, оскільки вона інтегрована безпосередньо в редактор Unity, дозволяючи розробникам синхронізувати зміни, відстежувати історію правок та уникнути конфліктів між різними версіями файлів без необхідності використовувати зовнішні інструменти.

Однак Unity не є ідеальним рішенням. Рушій не надає повного доступу до вихідного коду, що обмежує можливості глибокої оптимізації. Деякі просунуті функції, такі як більші обсяги хмарних сервісів чи детальна

аналітика, доступні лише у платних підписках. Окрім того Unity є система роялті (runtime fee), яка передбачає додаткові витрати для розробників після досягнення певного рівня доходу або кількості установок гри.

Після глибокого аналізу доступних ігрових рушіїв було обрано Unity як основномий інструменті для розробки гри. Це рішення було обґрунтоване цілим рядом переваг, які роблять Unity найкрашим варіантом саме для цього проєкту.

Однією з ключових переваг Unity є його зручність у роботі. Інтуїтивно зрозумілий інтерфейс дозволяє ефективно працювати з ігровими рівнями, швидко вносити зміни та тестувати їх у реальному часі без необхідності повної перезбірки проєкту. Це значно прискорює ітераційний процес і підвищує продуктивність розробки.

Важливим фактором стала потужна підтримка 2D-графіки, рушій пропонує цілий набір спеціалізованих інструментів для роботи з двовимірною графікою, включаючи розвинені системи анімації, фізики, виявлення колізій та рендерингу. Ці можливості ідеально підходять для проєкту, де 2D-графіка є основним напрямком.

Unity також відзначається високою гнучкістю, особливо що стосується процедурної генерації контенту. Вбудовані механізми роботи з подіями та фізикою дозволяють створювати складні динамічні системи, необхідні для реалізації наших ігрових механік. Крос-платформенність є ще однією вагомою перевагою, Unity дозволяє без зайвих зусиль адаптувати гру під різні платформи, від ПК і мобільних пристроїв до ігрових консолей.

Немаловажним чинником стала велика активна спільнота Unity. Наявність численних навчальних матеріалів, форумів і готових рішень значно спрощує процес розробки та дозволяє швидко вирішувати виникаючі технічні питання. Крім того, Unity пропонує широкі можливості інтеграції з різними зовнішніми сервісами, що може стати в нагоді на етапі комерціалізації проєкту.

Враховуючи всі ці фактори, рушій Unity 2D є оптимальним вибором для roguelike-гри. Він поєднує у собі потужний функціонал, зручність у використанні та широкі можливості для реалізації наших задумів, дозволяючи створити якісну гру з процедурною генерацією рівнів і складними ігровими механіками без зайвих складнощів у процесі розробки.

Гра розробляється з використанням компонентно-орієнтованої архітектури, що дозволяє створювати незалежні модулі, які можна легко замінювати або розширювати. Усі ігрові об’єкти складаються з окремих компонентів, що відповідають за різні аспекти їх поведінки, такі як рух, взаємодія з іншими об’єктами та візуальне відображення. Процедурна генерація рівнів реалізована через систему випадкового компонування кімнат за певними правилами, що дозволяє створювати унікальні рівні при кожному проходженні гри. Кожна кімната може містити різні елементи, такі як вороги, пастки, скрині з артефактами чи союзниками. Гравець має обмежений огляд кімнат, що означає, що він бачить лише поточну кімнату, а інші залишаються прихованими, доки не буде відкрито двері.

Обрані технології та архітектурні рішення дозволяють створити гнучку, модульну систему, яку легко розширювати та адаптувати під нові механіки. Компонентно-орієнтований підхід у Unity забезпечує зручність у створенні та тестуванні нових ігрових об’єктів, оскільки кожен об’єкт складається з незалежних частин. Використання процедурної генерації дозволяє зробити кожне проходження унікальним, що є важливою характеристикою для жанру roguelike. Використання C# спрощує реалізацію складних алгоритмів, а вбудовані можливості Unity для роботи з фізикою та анімацією дозволяють ефективно реалізувати ігровий процес. Таким чином, всі обрані рішення спрямовані на створення якісного та динамічного ігрового досвіду, який відповідатиме вимогам жанру та очікуванням гравців.

Для роботи з програмним кодом гри використовуватиметься Visual Studio як основне середовище розробки. Цей інструмент є стандартним рішенням для роботи з C# у Unity, оскільки має тісну інтеграцію з ігровим

рушієм та надає всі необхідні функції для ефективного написання скриптів. Інтеграція з Unity дозволяє відкривати скрипти напряму з Unity Editor, швидко переходити до помилок у коді та використовувати вбудований дебагер для аналізу роботи скриптів під час виконання гри. Це особливо важливо для реалізації складних механік, таких як процедурна генерація рівнів, де необхідно постійно тестувати логіку роботи коду. Visual Studio пропонує засоби автодоповнення коду, налагодження та рефакторингу, що значно прискорює процес розробки та допомагає уникнути помилок на ранніх етапах.

Для створення і редагування спрайтів було обрано Aseprite та Adobe Photoshop. Aseprite є спеціалізованим інструментом для малювання у стилі піксель-арт, який надає зручний інтерфейс для анімації, підтримує роботу з палітрами, шарами та кадрами, що підходить для 2D-ігор. Він дозволяє експортувати анімації у форматі .ase або .aseprite, які можна безпосередньо імпортувати в Unity за допомогою плагіну Aseprite Importer, що значно спрощує роботу з анімаціями.

Adobe Photoshop використовується для більш складних графічних елементів, таких як обробка текстур, створення ефектів освітлення та підготовка UI-елементів. Його інструменти для роботи з растровою графікою дозволяють досягти більш деталізованого вигляду спрайтів, якщо це необхідно.

Обидва інструменти добре працюють з Unity та забезпечують гнучкість у роботі з графікою. Вибір саме цих редакторів обґрунтований їхньою спеціалізацією: Aseprite буде використано для спрайтів та швидкої анімації, а Photoshop для доопрацювання деталей та складних ефектів.

Для контролю версій буде використовуватися Unity Version Control, який дозволяє відстежувати зміни, виконувати відновлення даних та уникнути конфліктів між різними версіями файлів. UVC інтегрований безпосередньо в Unity Editor, що робить процес роботи з репозиторієм більш зручним, оскільки не вимагає використання додаткових клієнтів.

2.2 Опис архітектури гри

Архітектура гри складатиметься з кількох ключових модулів, кожен з яких виконуватиме певну роль. Ігрова система буде будуватися на засадах компонентно-орієнтованої архітектури, що забезпечить високу гнучкість у модифікації та масштабованість функціоналу. Кожен ігровий об’єкт реалізуватиметься у вигляді самостійної сутності, до складу якої входитимуть спеціалізовані компоненти, відповідальні за окремі аспекти поведінки. Такий підхід дозволить ефективно керувати об’єктами, спростить внесення змін та забезпечить чітку модульність проєкту.

Крім того, компонентний підхід сприятиме повторному використанню логіки, що значно прискорить процес створення контенту та зменшить ймовірність виникнення технічних помилок.

Взаємодія компонентів гри зображена у структурно-логічній схемі яка наочно демонструє зв’язки між основними елементами, такими як механіки, графіка, звук, інтерфейс користувача та інші (рис. 2.1).

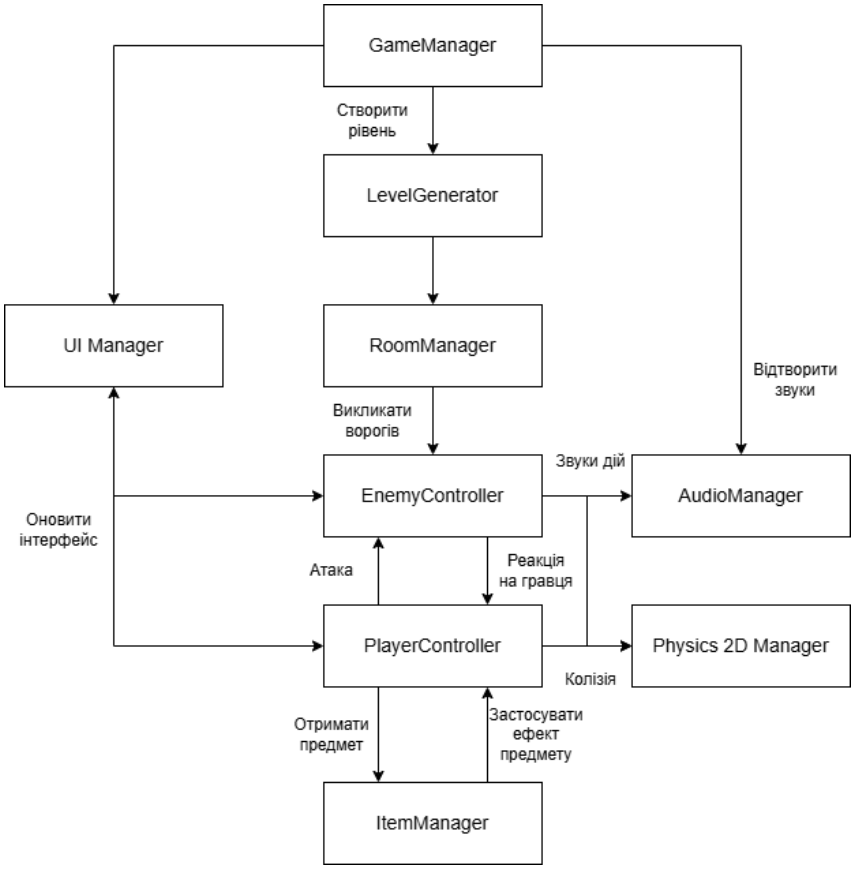


Рисунок 2.1 – Структурна схема гри

Процедурна генерація рівнів реалізовуватиметься за допомогою алгоритмів, які автоматично формують розташування кімнат, визначають їхні типи та зв’язки між ними, забезпечуючи унікальність кожного проходження. Функціонал, пов’язаний із гравцем, включатиме такі механіки, як переміщення, стрільба та використання здібностей.

Артефакти та інші ігрові предмети отримають власну систему характеристик, яка визначатиме їхній вплив на гравця, унікальні ефекти та умови використання. Інтерфейс користувача буде включати інформаційні панелі, що відображають стан здоров’я, енергію, доступні здібності, міні-карту локації та інші важливі елементи управління. Для узгодженої роботи всіх систем передбачається використання моделі взаємодії, що дозволить мінімізувати пряме зв’язування між різними частинами коду та забезпечити стабільність роботи гри навіть при активному розширенні функціоналу.

Кожен модуль взаємодіятиме з іншими за допомогою системи подій Unity або через прямі виклики методів. Наприклад, модуль керування гравцем реагує на введення користувача, оновлює позицію персонажа, а також передає інформацію до модуля фізики для перевірки колізій.

У грі рівні будуть складатися з окремих кімнат, які формуються на основі процедурної генерації [10].

Кожна кімната матиме свою конфігурацію, що включає такі елементи:

— Тип кімнати (звичайна, скарбниця, кімната босів, магазин тощо).
— Набір ворогів із визначеним рівнем складності.
— Вхідні та вихідні двері, що визначають з’єднання кімнати з іншими.
— Інтерактивні об’єкти, які можуть змінювати ігровий процес.

Це дозволяє швидко розширювати наповнення гри, додаючи нові кімнати, ворогів чи предмети.

## 2.3 Алгоритми та логіка взаємодії компонентів проєкту

Алгоритми та логіка взаємодії системи забезпечуватимуть коректну роботу гри, створюючи динамічний ігровий процес, який повинен змінюватися з кожним проходженням. Основна увага приділяється процедурній генерації рівнів, поведінці ворогів, механіці союзників і роботі артефактів.

Гравець буде центральним елементом гри, і його механіки визначатимуть основний ігровий процес. Керування здійснюватиметься за допомогою клавіш WASD для руху та мишки для стрільби. Персонаж матиме такі ключові характеристики: максимальний рівень здоров'я, захист, сила атаки, швидкість руху, швидкість стрільби, усі ці характеристики зможуть змінюватися під впливом артефактів.

Рух гравця відбуватиметься у чотирьох основних напрямках: вперед, назад, ліворуч, праворуч, анімації підтримуватимуть вісім напрямків для більш плавного відображення руху.

Алгоритм стрільби передбачає створення окремих об'єктів снарядів, що будуть рухатися у напрямку, вказаному мишею. При натисканні лівої кнопки миші викликатиметься функція генерації снаряда, де буде визначатися його початкова позиція, напрямок руху та швидкість.

Крім базових атак, гравець матиме особливі здібності, які будуть активовані натисканням певних клавіш. Їхня реалізація залежатиме від отриманих артефактів або стартових характеристик персонажа.

Додатково буде передбачена система взаємодії з об'єктами, наприклад, відкриття дверей, потрапляння у пастку або підбір предметів, що виконується через механізм перевірки колізій. Коли гравець входить у зону взаємодії, йому пропонується натиснути відповідну клавішу, щоб активувати потрібну дію.

Гравець також зможе отримувати ушкодження від ворогів або пасток. Коли персонаж зазнаватиме шкоди, від його здоров'я відніматиметься відповідне значення. Якщо здоров'я досягне нуля, активується алгоритм

завершення гри, який може включати відображення екрану поразки або можливість почати новий забіг.

Логіка викликів методів та взаємодій між об’єктами під час атаки гравця зображено на діаграмі послідовностей (рис. 2.2).

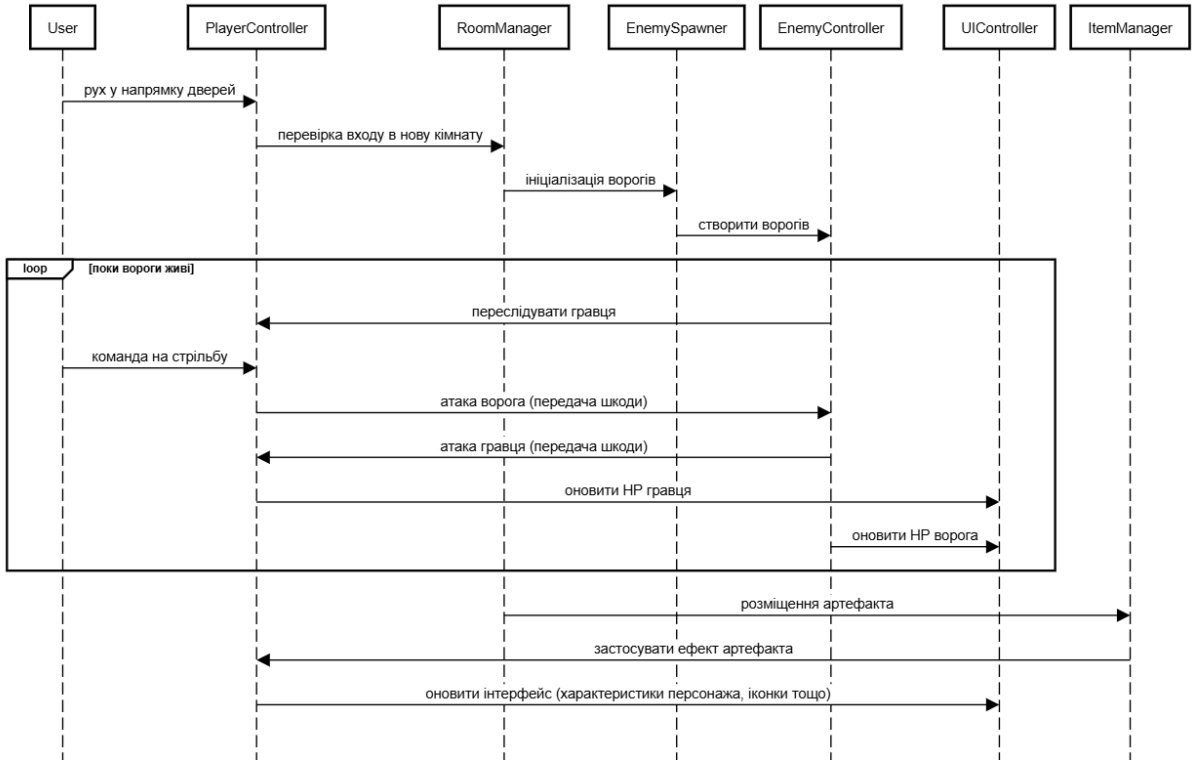


Рисунок 2.2 – Діаграма послідовностей

Процедурна генерація рівня базуватиметься на алгоритмі, що визначає розташування кімнат та їхній взаємозв’язок. Генерація починатиметься з визначення стартової кімнати, яка слугуватиме початковою точкою для гравця. Далі алгоритм визначить загальну кількість кімнат у рівні та поступово додасть їх, враховуючи певні обмеження.

Наприклад, особливі кімнати, такі як скарбниці та магазини завжди повинні бути тупиковими, а кімната боса розміщуватися таким чином, щоб не мати більш ніж один вхід. Під час генерації повинна перевірятися зв’язність усіх кімнат, щоб уникнути ситуацій, коли частина рівня стає недоступною для гравця.

Логіку генерації рівня зображено на рисунку 2.3.


Рисунок 2.3 – Логіка побудови рівня

Коли рівень створено, алгоритм вибору кімнати визначатиме, яка саме кімната буде додана наступною. Цей процес враховуватиме кілька параметрів, серед яких тип кімнати, її положення відносно інших приміщень та ймовірність появи певних кімнат.

Наприклад, система гарантуватиме, що в кожному рівні буде лише один магазин, а кімнати особливого типу, такі як скарбниці, не будуть розміщені поряд.

Після вибору кімнати алгоритм перевірятиме її правильне з’єднання з іншими, після чого додасть відповідні переходи.

Принцип, за яким обирається сама кімната та її позиція, зображено на рисунку 2.4.


Рисунок 2.4 – Логіка вибору кімнати для розміщення

Взаємодія ворогів із середовищем включатиме їх появу у відповідних кімнатах, поведінку та реакцію на дії гравця.

Під час входу в кімнату система перевірятиме її тип і рівень складності, після чого активує механізм генерації ворогів. Вороги можуть мати різні моделі поведінки, зокрема патрулювання, переслідування або атака гравця.

Механіка союзників функціонуватиме аналогічно, проте вони виконуватимуть підтримувальні дії для гравця. Після отримання відповідного артефакту союзник з’явиться біля гравця та слідує за ним, беручи участь у боях. Його поведінка залежатиме від типу викликаного створіння, але основними завданнями союзників будуть відновлення запасу здоров’я гравця та атака ворогів.

Союзники зможуть мати унікальні здібності, наприклад, атаки дальнього бою або здібність тимчасово блокувати ворогів. Як і гравець, вони матимуть свій набір характеристик, який буде здатен змінюватися.

Основні сценарії взаємодії користувача з грою наведено на діаграмі варіантів використання (рис. 2.5), яка демонструє функціональні можливості для гравця.


Рисунок 2.5 – Діаграма прецедентів

Система артефактів буде реалізована через алгоритм випадкової появи предметів у певних кімнатах. В залежності від типу кімнати й рівня складності рівня, шанси отримати той чи інший артефакт змінюються. Після отримання артефакту він застосовуватиме свої ефекти до персонажа, що може змінювати його характеристики або відкривати нові механіки. Артефакти впливатимуть як на самого гравця, змінюючи швидкість руху, силу атаки або здоров’я, так і на його атаки, змінюючи характеристики снарядів або додаючи додаткові ефекти.

Деякі артефакти зможуть взаємодіяти між собою, створюючи комбінації ефектів.

Робота всієї системи базуватиметься на обміні подіями, кожен елемент гри реагуватиме на події, такі як входження гравця до кімнати, отримання пошкоджень, підбір предметів або активація здібностей.

## 2.4 Проєктування інтерфейсу

Інтерфейс користувача є важливою складовою ігрового досвіду, оскільки забезпечує гравцеві доступ до необхідної інформації про стан персонажа, рівень, прогрес та дозволяє взаємодіяти з грою. Елементи інтерфейсу зрозумілими, ненав’язливими, адаптивними до роздільної здатності пристрою, а також органічно поєднаними із візуальним стилем гри.

Основним ігровим інтерфейсом, який бачить гравець під час проходження рівня, є HUD (head-up display). Його структура передбачає чітке розділення інформаційних панелей. Панель здоров’я гравця розміщується окремо у верхній частині екрану.. Індикатори здібностей персонажа розташовуються у нижньому правому куті. З лівого боку екрана передбачено текстову панель, яка відображає розширену інформацію про поточні характеристики персонажа: поточне здоров’я, швидкість, силу атаки, модифікатори та інші атрибути. За потреби, гравець зможе приховати цю панель під час гри, щоб не перевантажувати екран.

У верхній центральній частині екрану розташовується таймер, який показує тривалість поточного проходження рівня. Його оновлення припиняється під час паузи або завершення рівня. У нижній частині інтерфейсу відображаються елементи, пов’язані із наявними предметами або союзникам.

Приклад того, як виглядатиме head-up display в грі під час проходження рівня зображено на рисунку 2.6.


Рисунок 2.6 – Макет HUD під час проходження рівня

При натисканні клавіші паузи гра призупиняється, таймер зупиняється, і відкривається меню паузи. У ньому гравець може переглянути всі наявні характеристики персонажа у зручному форматі, ознайомитися з артефактами, що були зібрані під час проходження рівня, а також прочитати коротку інформацію про ефекти кожного предмета. Меню паузи також надає змогу вийти до головного меню або перейти до налаштувань без втрати поточного прогресу (рис. 2.7).

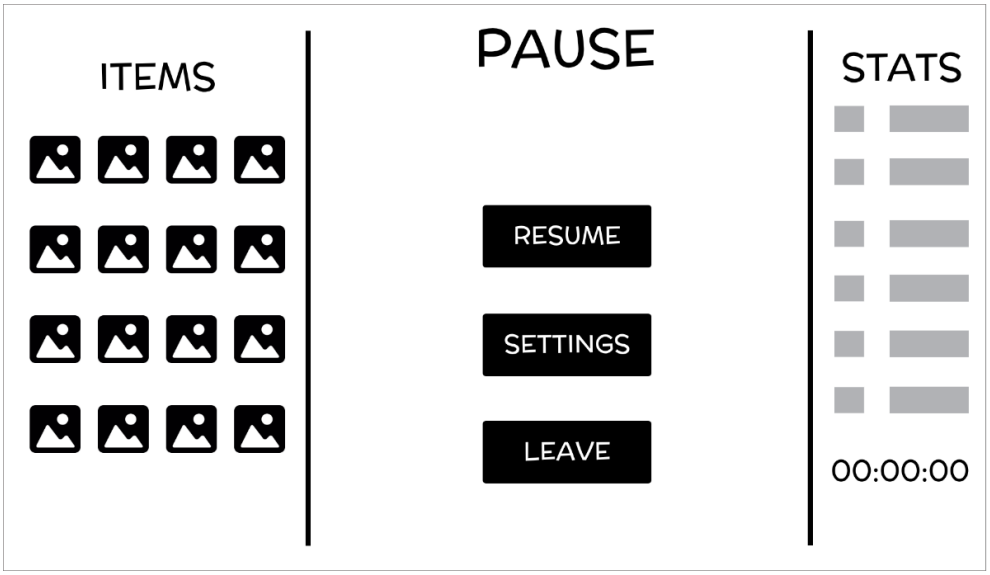


Рисунок 2.7 – Макет меню паузи

Головне меню виступає точкою входу в гру та забезпечує доступ до основних режимів. З нього гравець може розпочати швидку гру або сюжетну кампанію, перейти до налаштувань, а також вийти з гри. Головне меню повинно бути інтуїтивно зрозумілим, із логічним розміщенням кнопок, що забезпечує швидкий доступ до основних функцій.

Важливим елементом також є вкладка налаштувань, де користувач може обрати бажану гучність звуків та музики, змінити розкладку клавіш управління, налаштувати графічні параметри, такі як роздільна здатність екрана, режим вікна або повноекранний режим, а також увімкнути або вимкнути додаткові візуальні ефекти (рис. 2.8).

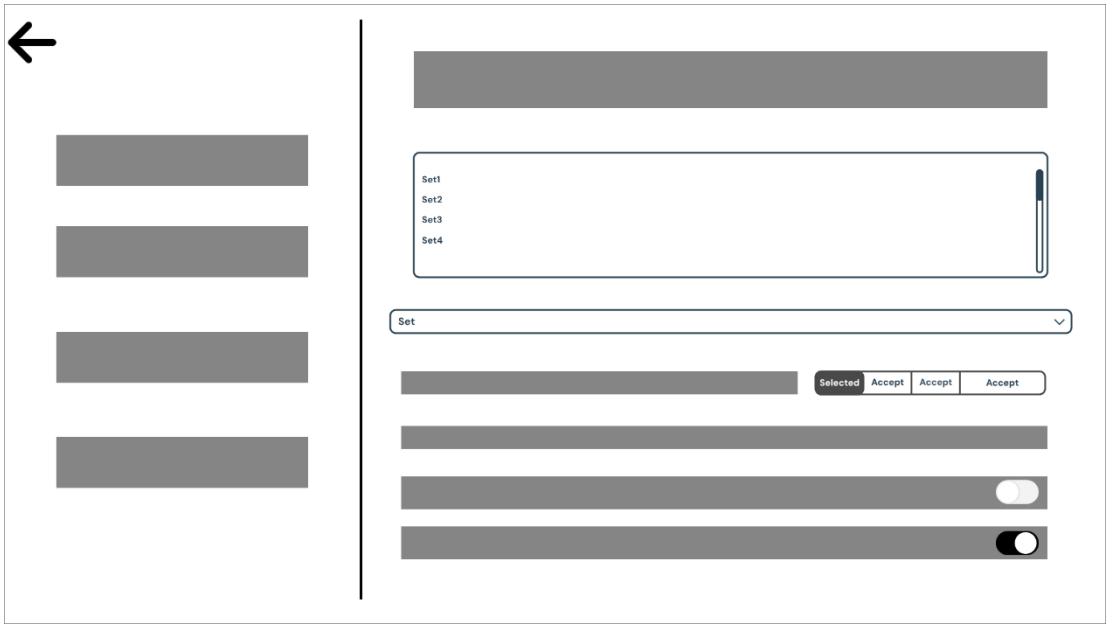


Рисунок 2.8 – Макет вкладки налаштувань

Ще одним важливим елементом інтерфейсу є  екран завершення рівня. Його логіка уніфікована — той самий екран використовуватиметься як у випадку перемоги, так і у випадку поразки, змінюється лише візуальна стилістика та заголовок. Екран відображатиме час проходження, кількість знищених ворогів, список зібраних предметів, а також змінені характеристики персонажа. У нижній частині екрану буде розміщено кнопки для повторного проходження рівня, переходу до наступної глави, якщо це сюжетна кампанія, або виходу в головне меню.

Усі елементи інтерфейсу будуть реалізовані із застосуванням адаптивної верстки. У межах Unity це досягається через Canvas Scaler з параметром Scale With Screen Size, що дозволяє коректно масштабувати інтерфейс залежно від роздільної здатності пристрою. Таким чином, інтерфейс залишатиметься зрозумілим і функціональним як на широкоформатних екранах, так і на меншій роздільній здатності.

## 2.5 Розробка сюжетної складової гри

У процесі створення ігрового світу буде передбачено наявність сюжетної лінії, яка додасть мотивації до проходження гри та створить контекст для хаотичної структури рівнів, притаманної жанру roguelike.

Сценарій буде орієнтований на поєднання науково-фантастичного елементу з особистою історією головного героя, що дозволить надати подіям глибший зміст без необхідності будувати структуровану оповідь. Основна увага буде приділятися атмосфері загадковості, ізольованості та поступового розкриття подій, які сталися до початку гри.

У грі буде присутній сюжет про звичайного жителя маленького містечка, хлопця, що працює у місцевому супермаркеті та життя якого змінюється внаслідок аварії на науковому об’єкті — адронний колайдер. Під час експерименту з моделювання умов раннього всесвіту відбудеться невдале зіштовхування частинок, що в результаті спричинить порушення просторово-часового балансу. Це призведе до утворення нестабільного міжвимірного розлому, в якому почнуть зникати об’єкти з різних реальностей. Головний герой, повертаючись додому, випадково потрапить у цей простір, який згодом отримає назву «Medium inter mundos».

Цей новий вимір буде представлено як нестабільну зону, наповнену уламками різних реальностей, фрагментами світів, об’єктами та істотами, що не мають логічного зв’язку між собою. Такий підхід дозволить обґрунтувати процедурну генерацію рівнів, в яких різноманітні елементи ландшафту, супротивники та стилістика будуть змінюватися випадковим чином.

На сюжетному рівні це буде подано як хаос, спричинений зіткненням світів, що раніше не мали між собою зв’язку.

Сценарій передбачатиме, що головний герой буде прагнути знайти вихід із простору між світами та повернутися додому. Для цього він буде змушений подорожувати крізь так звані зони нестабільності, боротися з ворожими істотами, долати перешкоди та поступово дізнаватися більше про природу розлому. По мірі проходження гравець буде знаходити уламки спогадів світів та артефакти з інших реальностей. Це дозволить поступово розкривати як передумови катастрофи, так і роль самого героя у подіях, що сталися в минулому.

Сюжет буде подаватися фрагментарно, з акцентом на атмосферу та елементи дослідження. В окремих кімнатах або подіях сюжетної кампанії гравець зможе натрапляти на об’єкти, які відкриватимуть нові деталі історії або надаватимуть вибір, що впливатиме на подальше просування сюжету.

Кожен з предметів, які гравець знаходить, матиме не лише ігрову функцію, а й власну історію походження, що належатиме до окремого світу або реальності. Через ці предмети поступово буде розкриватися різноманіття вимірів, фрагменти культур, подій та трагедій, що мали місце в світах, які тепер стали частиною розлому.

У фінальній частині гри буде реалізовано завершення, яке залежатиме від дій гравця протягом гри, зокрема кількості зібраних артефактів та виконаних умов. Це дозволить запропонувати кілька альтернативних кінцівок: повернення додому або залишення в просторі розлому.

Сценарій гри буде побудований так, щоб не обмежувати механіки, але надати світові гри певну логіку та історію, що дозволить гравцеві відчувати себе відкривачем, ізольованим у незнайомому середовищі.

# 3 СТВОРЕННЯ ТА ТЕСТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

## 3.1 Реалізація структури та ігрових механік

Проєкт побудовано на основі компонентної архітектури Unity з використанням MonoBehaviour компонентів та ScriptableObjects для конфігурації ігрових параметрів.

Одним з найважливіших елементів гри є персонаж гравця, який пересується двовимірним світом та атакує ворогів із лука. Рух героя реалізовано за допомогою фізичного рушія Unity, що забезпечує плавність та натуральну інерцію. Гравець керує персонажем за допомогою клавіш WASD, а стрільба за допомогою IJKL. Лістинг програмного коду скрипту для керування персонажем розміщено в додатку А.

Система параметрів реалізована через клас PlayerStats, який містить такі показники, як запас здоров’я персонажа, сила атаки, захист, швидкість переміщення та швидкість польоту стріл. Пошкодження обчислюється з урахуванням захисту цілі, а при отриманні удару гравець миттєво змінює колір спрайту на червоний, що візуально підкреслює отриману шкоду. Лістинг скрипту, що відповідає за керування характеристиками персонажа розміщено у додатку Б.

Система стрільби побудована на основі об’єктів-снарядів у класі Bullet, стріли створюються при натисканні відповідних клавіш керування. Лістинг класу для снарядів зображено в лістингу 3.1. Кожна стріла летить у заданому напрямку зі швидкістю, що визначається параметром швидкості кулі, і знищується після зіткнення з ворогом або через певний час, який визначено в коді.

Лістинг 3.1 – Базовий клас для снарядів персонажа

```
public class Bullet : MonoBehaviour{
     public float lifeTime = 2f;
     public int damage = 10;

    void Start(){
```

```
        Debug.Log($"Знищення стріли");
        Destroy(gameObject, lifeTime);

    }
    void OnTriggerEnter2D(Collider2D collision){
        if (collision.CompareTag("Enemy")){
            EnemyBase                    enemy                    =
collision.GetComponent<EnemyBase>();
            if (enemy != null){
                enemy.TakeEffectDamage(damage);
                Debug.Log($"Нанесено {damage} для {enemy.name}");
            }
            Destroy(gameObject);
        }}}
```

Для реалізації різних типів атак використовується інтерфейс IBulletModifier, який дозволяє артефактам модифікувати властивості снарядів. Це дозволило артефактам додавати стрілам ефекти на кшталт отруєння або просто підвищувати їхню шкоду. Приклад функції, яка використовує цей інтерфейс для передачі ефектів до стріл наведено в лістингу 3.2.

Лістинг 3.2 – Функція для передачі ефектів снарядам

```
public void ModifyBullet(GameObject bullet){
if (Random.Range(0f, 1f) <= stunChance){
var applier = bullet.GetComponent<BulletStatusApplier>();
            if      (applier      ==      null)      applier      =
bullet.AddComponent<BulletStatusApplier>();
            var stunEffect = new StatusEffect(
                EffectType.Stun,
                stunDuration,
                effectColor: stunColor,
                visualEffectPrefab: stunEffectPrefab,
                effectName: "Golden Stun",
                soundEffect: stunSound);
            applier.AddStatusEffect(stunEffect);

            if (bullet.TryGetComponent(out SpriteRenderer sr)){
                sr.color = stunColor;}}}
```

Система анімацій персонажа реалізована за допомогою механізмів Unity Animator , що дозволяє створювати систему анімацій за допомогою станів і переходів між ними. Для головного героя створено анімації для базового переміщення та атак, що забезпечує плавне переключення між цими станами без різких змін у відображенні.

Основу анімаційного контролера складають параметри типу float та bool, які відповідають за напрямок руху та стан персонажа. Параметри «MoveX» і «MoveY» використовуються для визначення напрямку переміщення, тоді як «IsRunning» і «IsIdle» контролюють перемикання між анімаціями бігу та простою. При стрільбі система визначає точний напрямок атаки за допомогою векторного добутку між напрямком пострілу та базовими векторами, що дозволяє точно вибрати одну з восьми можливих анімацій стрільби.

Для реалізації стрільби в різних напрямках створено окремі анімаційні кліпи, кожен з яких відповідає певному напрямку атаки. Активація потрібної анімації відбувається за допомогою тригерів, таких як «Bow_Attack_Up». Усі стани з відповідними тригерами об’єднано в Unity Animator (рис. 3.1).

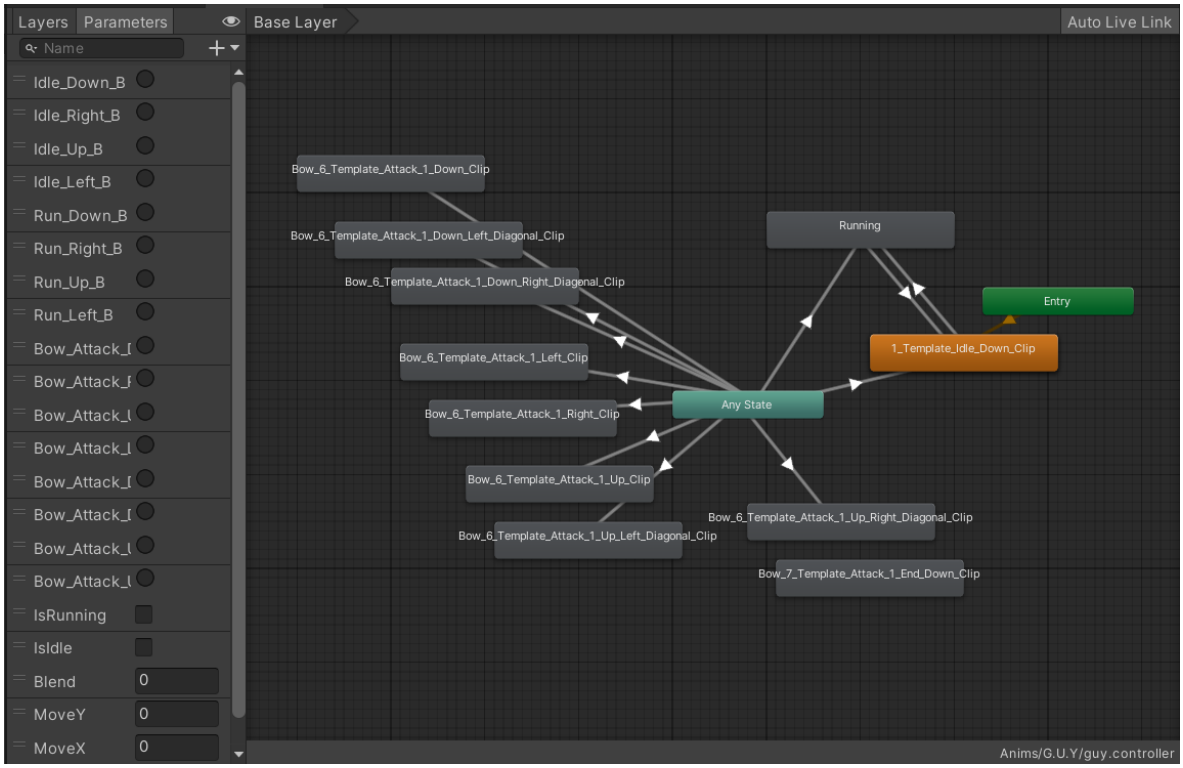


Рисунок 3.1 – Unity Animator з налаштованими анімаціями персонажа

Після завершення анімації система автоматично повертає персонажа до початкового стану.

Артефакти в грі реалізовані через механізм ScriptableObject, що дозволяє легко створювати нові типи артефактів без необхідності змінювати основний код. Кожен артефакт у грі наслідує базовий клас ArtifactBase (лістинг 3.3), який містить загальні властивості, такі як назва, опис та метод OnPickup, що викликається при підборі предмета гравцем.

Лістинг 3.3 – Базовий клас ArtifactBase

```
public abstract class ArtifactBase : ScriptableObject{
    public string artifactName;
    public string description;
    public virtual void OnPickup(PlayerController player) { }
}
```

Гравець отримує артефакти, відкриваючи скрині, розміщені на рівнях. Логіка скринь реалізована в класі ChestController, який при взаємодії з гравцем викликає метод OpenChest, лістинг коду класу ChestController розміщено в додатку В. Після відкриття скриня змінює свій спрайт, відтворює ефект частинок і створює випадковий артефакт із списку ArtifactDatabase. Ця база містить два списки, унікальні артефакти, які можуть зустрічатися лише один раз, і повторювані, які гравець може отримувати багаторазово.

Лістинг 3.4 – Система збереження артефактів

```
[CreateAssetMenu(menuName = " Artifacts/Artifact Database")]
public class ArtifactDatabase : ScriptableObject{
    public List<ArtifactBase> allArtifacts;
    public List<ArtifactBase> repeatableArtifacts;
    private List<ArtifactBase> availableUniqueArtifacts = new
List<ArtifactBase>();
    public void Initialize(){
        availableUniqueArtifacts = new List<ArtifactBase>();
        foreach (var artifact in allArtifacts){
            if (!repeatableArtifacts.Contains(artifact)){
availableUniqueArtifacts.Add(artifact);}}}
    public ArtifactBase GetRandomArtifact(){
        if (availableUniqueArtifacts.Count > 0){
            int index = Random.Range(0,
availableUniqueArtifacts.Count);
            ArtifactBase selected =
availableUniqueArtifacts[index];
            availableUniqueArtifacts.RemoveAt(index);
            return selected;}
        if (repeatableArtifacts.Count > 0){return
repeatableArtifacts[Random.Range(0, repeatableArtifacts.Count)];}
```

Під час підбору артефакт викликає метод AddArtifact у PlayerController, який додає його до набору гравця. Система артефактів забезпечує гнучкість у розширенні можливостей гравця, дозволяючи створювати унікальні комбінації ефектів, які впливають на ігровий процес.

Вороги в грі базуються на класі EnemyBase, лістинг програмного коду скрипту для класу ворогів зображено в додатку Г, який визначає їх основну логіку поведінки, атаки та взаємодії з гравцем.

Кожен ворог має стандартні характеристики, такі як здоров'я швидкість пересування, шкоду, радіус атаки та час відновлення між атаками. Під час ініціалізації ворог знаходить об'єкт гравця за тегом «Player» і зберігає посилання на його трансформ для подальшої взаємодії.

Рух ворога реалізований через фізичний рушій Rigidbody2D. У методі MoveTowardsPlayer ворог постійно обчислює напрямок до гравця та змінює свою швидкість, щоб рухатися до нього. Якщо гравець знаходиться в межах радіусу атаки і минув час перезарядки, ворог викликає метод AttackPlayer, який завдає шкоди гравцевю через компонент PlayerStats.

Особливістю системи є механізм статусних ефектів, таких як приголомшення або сповільнення. Ворог може отримувати ці ефекти через метод ApplyEffect, який додає їх до списку activeEffects. Кожен ефект впливає на поведінку ворога: наприклад, приголомшення зупиняє рух і атаку, а сповільнення зменшує швидкість пересування. Ефекти оновлюються кожен кадр у методі Update, і якщо їх дія закінчується, вони видаляються зі списку.

Для створення ворогів на рівні використовується EnemySpawner, який відповідає за їх появу у заданих точках (лістинг 3.5). Метод SpawnEnemies приймає індекс типу ворога з масиву enemyPrefabs і створює його копії в усіх точках спавну. Це дозволяє легко керувати кількістю і типом ворогів на різних рівнях.

Лістинг 3.5 – Скрипт для появи ворогів у кімнаті

```
public class EnemySpawner : MonoBehaviour{
    public GameObject[] enemyPrefabs;
    public Transform[] spawnPoints;
```

```
    public void SpawnEnemies(int enemyTypeIndex) {
        foreach (Transform spawnPoint in spawnPoints){
            Instantiate(enemyPrefabs[enemyTypeIndex],
spawnPoint.position, Quaternion.identity);
        }}}
```

Коли здоров'я ворога падає до нуля, він знищується. Перед цим може бути додано додаткову логіку, наприклад, поява скринь або виклик події для отримання монет.

Серед усіх інших ворогів, боси в грі є унікальними противниками, які зустрічаються гравцеві наприкінці кожного рівня. Вони створені на основі абстрактного класу BossBase, який визначає їхню базову поведінку, характеристики та механізми взаємодії, лістинг програмного коду абстрактного класу боса розміщено в додатку Ґ. Кожен бос має ім'я, великий запас здоров'я, стан вразливості та систему винагород за перемогу.

Для відображення здоров'я боса використовується інтерфейс BossHealthBar (лістинг 3.6), який показує поточний рівень здоров'я у відсотках та ім'я боса . Цей інтерфейс оновлюється в реальному часі, відображаючи стан боса під час битви.

Лістинг 3.6 – Відображення кількості здоров'я боса через UI

```
public class BossHealthBar : MonoBehaviour{
    public Slider healthSlider;
    public Text bossNameText;
    public GameObject phaseIndicator;
    private BossBase boss;
    public void Initialize(BossBase targetBoss){
        boss = targetBoss;
        bossNameText.text = boss.bossName;
        healthSlider.maxValue = boss.maxHP;
        healthSlider.value = boss.currentHP;
        gameObject.SetActive(true);}
    public void UpdateBossHealthBar(BossBase boss){
        healthSlider.value = boss.GetHealthPercentage();
        bossNameText.text = boss.bossName;}}
```

Боси мають метод «TakeDamage», який враховує стан вразливості. Якщо бос не є вразливим, наприклад, під час певної фази бою, він ігнорує отриману шкоду. При отриманні пошкодження бос може відтворювати анімацію

отримання шкоди через параметр «Hurt» в Animator, що забезпечує візуальний зв’язок для гравця.

Коли здоров’я боса падає до нуля, викликається метод знищення, який відповідає за завершення битви. Під час усунення боса створюються ефекти знищення, а також з’являються винагороди у вигляді артефактів, які гравець може підібрати.

Окрім ворогів, у грі також існують союзники, які представлені абстрактним класом AllyBase, який визначає базову логіку їхньої поведінки, лістинг програмного коду класу союзників розміщено в додатку Д. Вони допомагають гравцеві, автономно пересуваючись за ним та атакуючи ворогів у визначеному радіусі. Кожен союзник має основні параметри, схожі до параметрів інших сутностей: швидкість руху, радіус виявлення ворогів, дальність атаки, час відновлення атаки та базову шкоду.

Союзники використовують метод FollowPlayer для плавного пересування за гравцем, підтримуючи комфортну дистанцію. Для забезпечення природного руху застосовується функція SmoothDamp яка згладжує зміни напрямку та швидкості. Відстань і позиція слідування можуть бути адаптовані через метод GetFollowOffset, що дозволяє створювати різні схеми руху для різних типів союзників.

Пошук ворогів реалізований через абстрактний метод LookForEnemies, який має бути реалізований у кожному конкретному типі союзника. Для пошуку найближчого ворога використовується метод FindNearestEnemy, який сканує область навколо союзника та повертає найближчу загрозу.

Гравець отримує союзника, підібравши відповідний артефакт, який активує їх появу та подальшу підтримку в бою. Для реалізації поведінки їхніх снарядів створено абстрактний клас, який є базовою основою для снарядів союзників. Реалізацію цього абстрактного класу наведено в додатку Е.

3.2 Побудова ігрового процесу

Ігровий світ генерується процедурно за допомогою системи LevelGenerator, яка створює різноманітні рівні з унікальним розташуванням кімнат. Лістинг програмного коду для генератора рівнів наведено у додатку Є. Кожен рівень складається з набору кімнат різних типів: стартова кімната, звичайні кімнати, скарбниці, магазини та кімнати з босом.

Генерація рівня починається зі стартової кімнати, після чого система послідовно додає нові кімнати, розширюючи мапу у випадкових напрямках. Для забезпечення різноманітності та уникнення надто лінійного розташування, кількість напрямків розширення обмежена, що створює природні розгалуження та тупики. Під час генерації система перевіряє, щоб спеціальні кімнати розташовувались у достатньо віддалених одна від одної місцях, що забезпечує баланс ігрового процесу.

Кожна кімната має свої унікальні характеристики, такі як тип та позиція на ігровій сітці. Після генерації всіх кімнат система кімнат автоматично налаштовує двері між ними, активуючи тільки ті, які ведуть до сусідніх кімнат, лістинг класу кімнат наведено в лістингу 3.7.

Лістинг 3.7 – Програмний код класу кімнат на рівні

```
public enum RoomType{Spawn, Regular, Treasure, Shop, Boss, Puzzle}
public class Room : MonoBehaviour{
    public RoomType roomType;
    public Vector2Int position;
    public   Dictionary<Vector2Int,   Room>   neighbors   =   new
Dictionary<Vector2Int, Room>();
    public GameObject doorUp;
    public GameObject doorDown;
    public GameObject doorLeft;
    public GameObject doorRight;
    public void SetDoors(Dictionary<Vector2Int, Room> allRooms){
        if  (allRooms.TryGetValue(position  +  Vector2Int.up,  out
Room upRoom)){
            neighbors[Vector2Int.up] = upRoom;
            doorUp.SetActive(true);
        }
        if  (allRooms.TryGetValue(position + Vector2Int.down, out
Room downRoom)){
            neighbors[Vector2Int.down] = downRoom;
            doorDown.SetActive(true);
        }
```

```
        if (allRooms.TryGetValue(position + Vector2Int.left, out
Room leftRoom)){
            neighbors[Vector2Int.left] = leftRoom;
            doorLeft.SetActive(true);
        }
        if (allRooms.TryGetValue(position + Vector2Int.right, out
Room rightRoom)){
            neighbors[Vector2Int.right] = rightRoom;
            doorRight.SetActive(true);}}}
```

У процесі створення ігрового світу ключову роль відіграє система Tilemap, яка була використана для побудови кімнат для загальної структури рівнів. Tilemap дозволяє ефективно працювати з двовимірним ігровим простором, оскільки забезпечує зручне розміщення плиток на сітці з точністю до пікселя.

Для створення кімнат в грі було створено декілька Tilemap компонентів у межах одного Grid-об’єкта (рис. 3.2). Основний шар містить плитки підлоги, поверх яких розміщуються стіни чи інші елементи, які обмежують рух гравця.

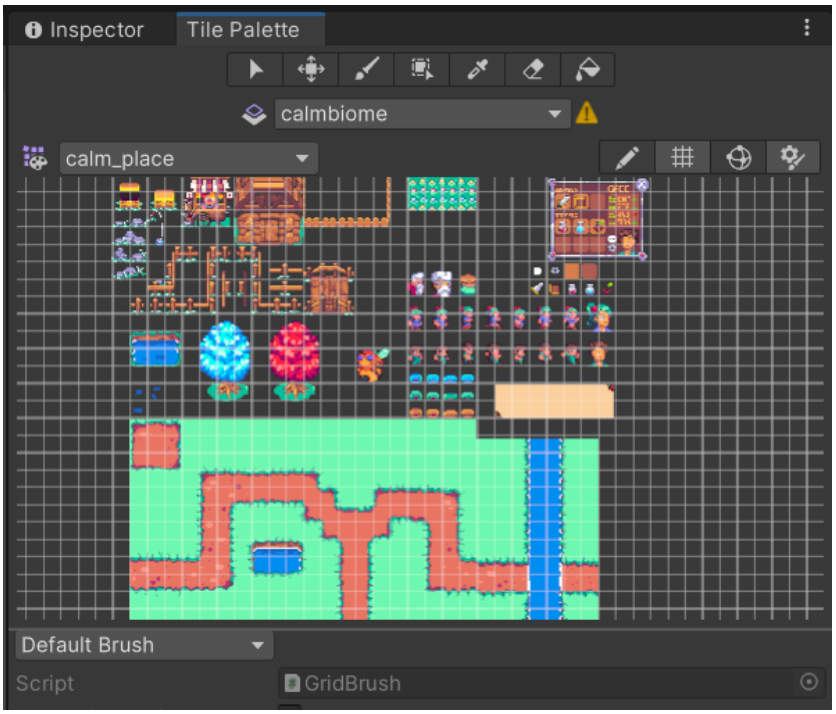


Рисунок 3.2 – Один із створених Tilemap для редагування кімнат

Також використовуються Tilemap Collider і Composite Collider для забезпечення фізичної взаємодії з елементами сцени — наприклад, щоб гравець не міг проходити крізь стіни.

Для формування самих спрайтів було використано програму Adobe Photoshop, яка надала можливість працювати зі спрайтами, враховуючи стилістику гри. (рис 3.3).

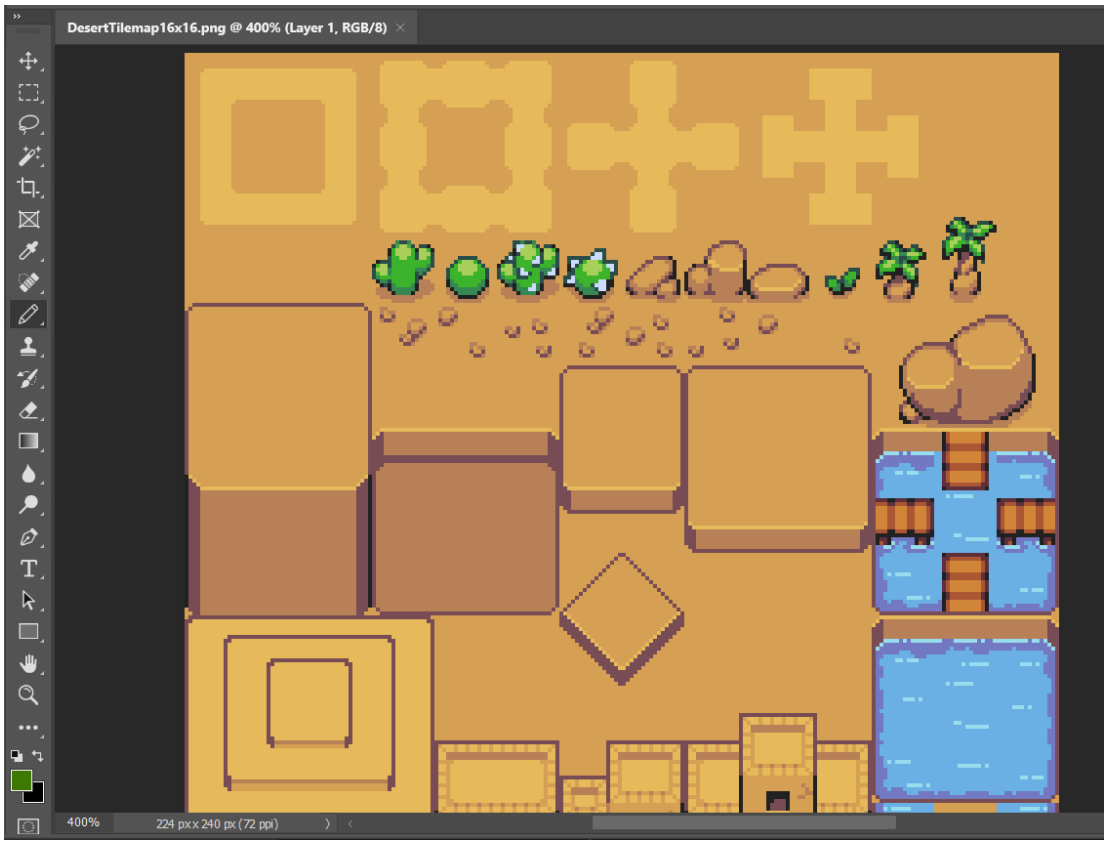


Рисунок 3.3 – Робота зі спрайтами в Adobe Photoshop

Усі графічні ресурси були експортовані у форматі PNG із прозорим фоном. Далі вони були імпортовані в Unity, де розділялися у Sprite Editor на окремі тайли з фіксованим розміром.

Особливістю реалізації є те, що частина спрайтів (наприклад, кутові стіни, перехрестя чи тіньові варіації) має автопідстановку на основі сусідніх тайлів. Це було реалізовано через Rule Tile або ручну логіку розміщення при побудові кімнати в шаблоні. У результаті, незалежно від того, який шаблон кімнати використовується, її візуальна цілісність зберігається, а процес створення нових приміщень стає швидким та гнучким.

3.3 Створення інтерфейсу користувача

Інтерфейс користувача в грі розроблений для забезпечення зручної взаємодії гравця з ігровим світом та надання необхідної інформації в реальному часі. Всі елементи інтерфейсу створені за допомогою Unity UI System, що дозволяє легко змінювати їхній вигляд та функціональність.

Було розроблено головне меню, воно є першим, що зустрічає гравця в грі, тому воно не повинно бути перевантажене елементами. Меню має інтуїтивно зрозумілу структуру та оформлено у візуальному стилі, що відповідає загальній атмосфері гри (рис. 3.4). Воно дозволяє гравцеві розпочати нову гру, почати сюжетну кампанію, перейти до налаштувань або завершити застосунок. Кожен елемент головного меню має адаптивний інтерфейс, який коректно масштабується на різних розширеннях екрана.

Рисунок 3.4 – Створення головного меню в Unity

Одним з найважливіших елементів UI є HUD гравця, він включає основні елементи, такі як панель здоров'я, яка в реальному часі відображає поточний рівень здоров'я та інші характеристики персонажа. Лістинг програмного коду, необхідного для роботи HUD гравця наведено в додатку Ж. Робота панелі здоров'я реалізується у вигляді заповненої шкали або числового значення, яке плавно оновлюється при отриманні шкоди або лікуванні, інші

значення зазвичай відображаються у вигляді текстових полів, розташованих у зручному для ока місці, ліворуч по центру, щоб не перекривати основну зону огляду (рис. 3.5).

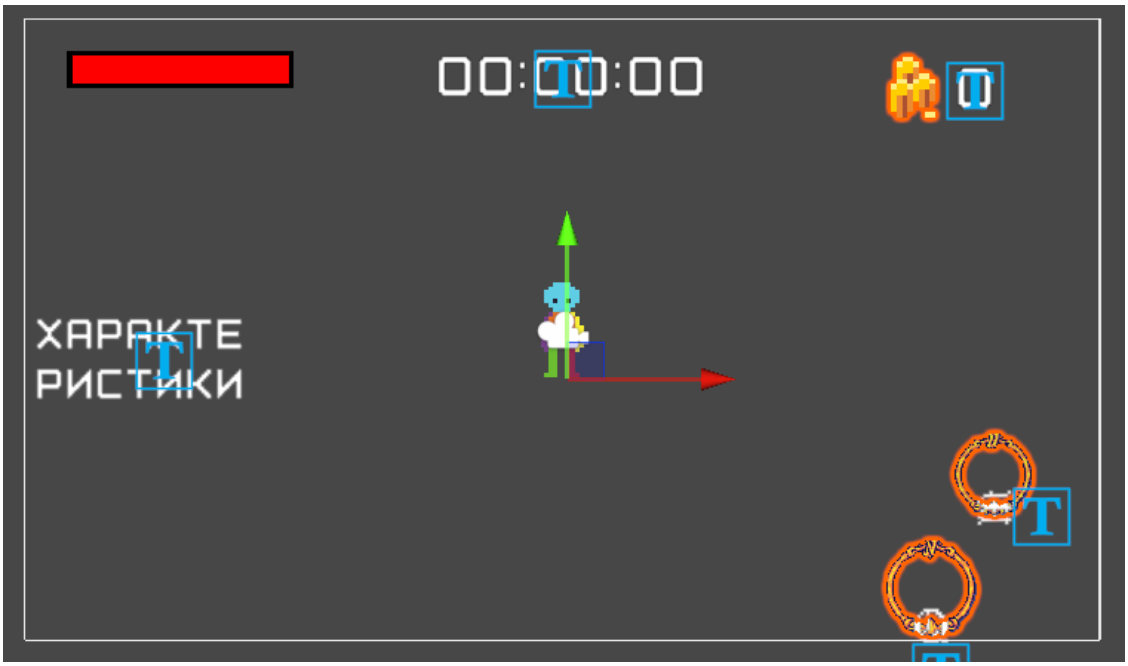


Рисунок 3.5 – Створення HUD в Unity

До складу HUD також входить таймер рівня, який показує, скільки часу гравець провів на поточному рівні, лістинг програмного коду, який використовується для таймера наведено у лістингу 3.8.

Лістинг 3.8 – Програмний код таймеру для HUD

```
public class LevelTimer : MonoBehaviour{
    public TextMeshProUGUI timerText;
    private float levelTime = 0f;
    private bool isRunning = true;
    void Update(){
        if (!isRunning) return;
        levelTime += Time.deltaTime;
        int minutes = Mathf.FloorToInt(levelTime / 60);
        int seconds = Mathf.FloorToInt(levelTime % 60);
        timerText.text = $"{minutes:00}:{seconds:00}";}
    public void StopTimer(){
        isRunning = false;}
    public float GetLevelTime(){return levelTime;}}
```

В Unity було створено вкладку налаштувань, яка дозволяє гравцю змінювати основні параметри гри відповідно до власних уподобань (рис. 3.6).

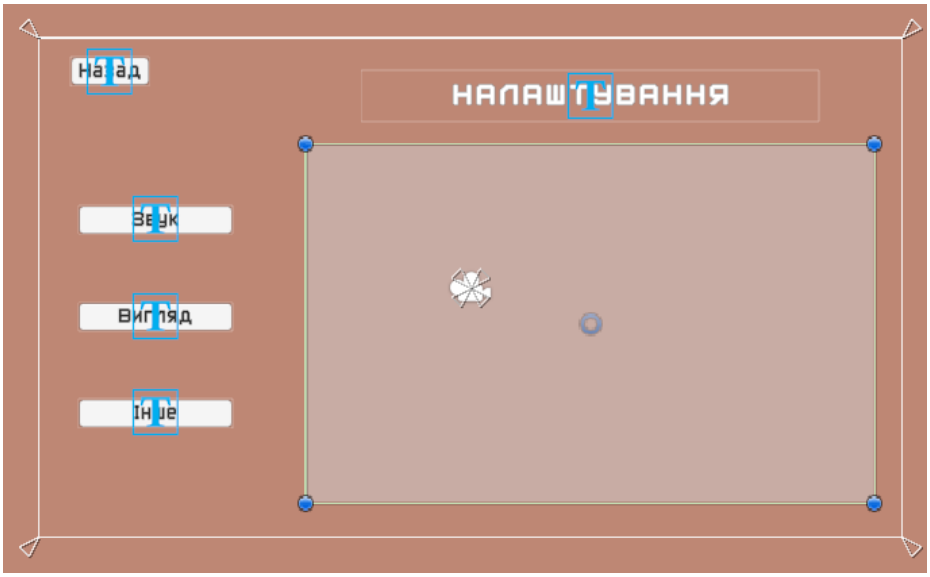


Рисунок 3.6 – Створення вкладки налаштувань

Ця система охоплює налаштування звуку, яскравості, чутливості миші та типу керування. Вона реалізована у вигляді зручного UI-інтерфейсу, доступного як з головного меню. Лістинг програмного коду, що використовується для роботи системи налаштувань наведено у додатку І.

Було створено меню паузи, яке активується натисканням клавіші Escape під час гри (рис. 3.7).


Рисунок 3.7 – Меню паузи в Unity Scene

Меню дозволяє призупинити гру, тимчасово вимкнути управління персонажем та відображає базові кнопки: продовжити гру, повернутися в головне меню, вийти з гри. Пауза працює поверх основної сцени та блокує весь геймплей, доки гравець не вирішить повернутися. При активації меню час у грі призупиняється завдяки встановленню Time.timeScale = 0, що зупиняє усі фізичні та оновлювальні процеси, поверення до гри відновлює час.

У Unity було реалізовано систему магазину (рис. 3.8), яка дозволяє гравцеві витрачати зароблені під час проходження монети на купівлю артефактів. Гроші були додані як окрема змінна в клас PlayerStats, що дозволяє легко керувати додаванням чи зняттям монет. У магазині відображаються чотири випадкові предмети, які беруться зі списку артефактів. Для кожного з них генерується випадкова ціна в межах від 10 до 20 монет. Лістинг програмного коду для роботи магазину наведено в додатку К.

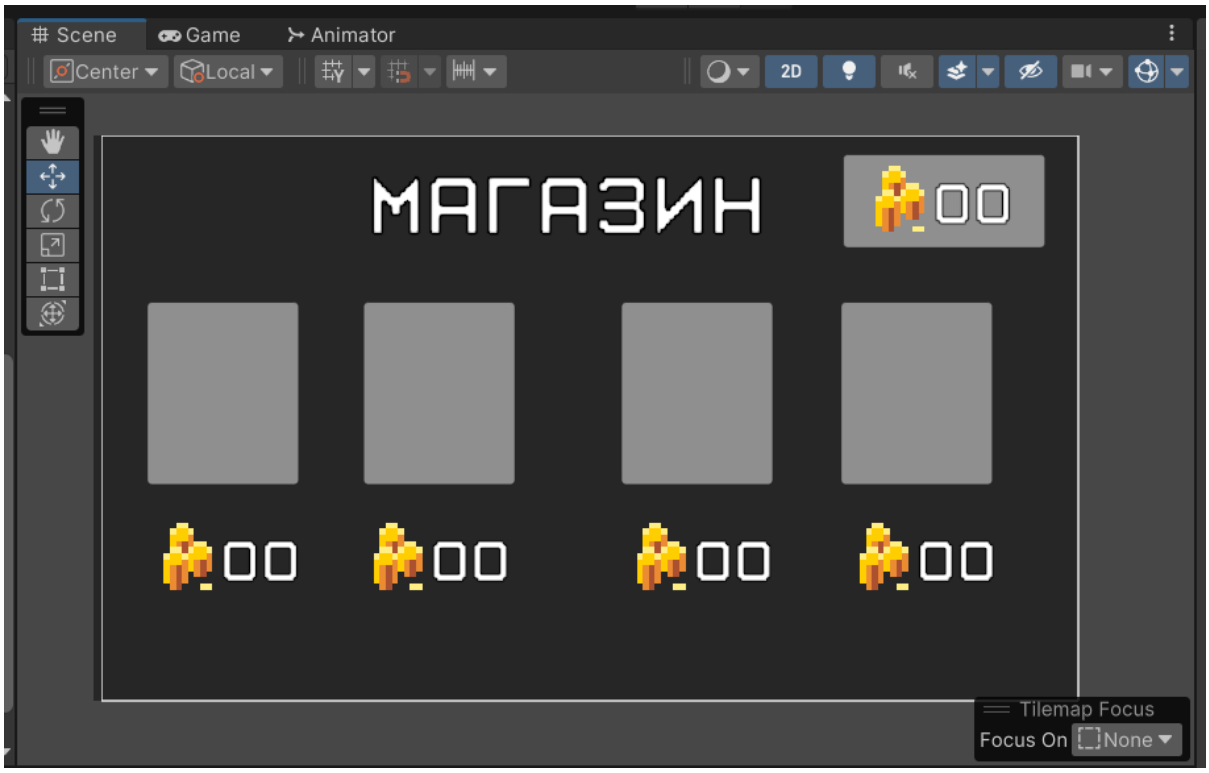


Рисунок 3.8 – Створення меню магазину в Unity

Кожен товар представлений у UI через окремий префаб, що містить іконку, вартість та кнопку для придбання.

Після натискання на кнопку, система перевіряє, чи має гравець достатньо монет, і, якщо так, то артефакт активується, а монети знімаються з балансу.

У Unity було створено систему діалогів, яка забезпечує інтерактивне спілкування між гравцем та неігровими персонажами.

Для цього було реалізовано компонент DialogueManager, який відповідає за відображення обличчя персонажа, його імені та тексту реплік. Лістинг програмного коду для системи діалогів наведено у додатку Л.

Кожен діалог складається з набору реплік, які виводяться послідовно на екран із ефектом поступового друку символів, що підвищує атмосферність сцени (рис. 3.9).

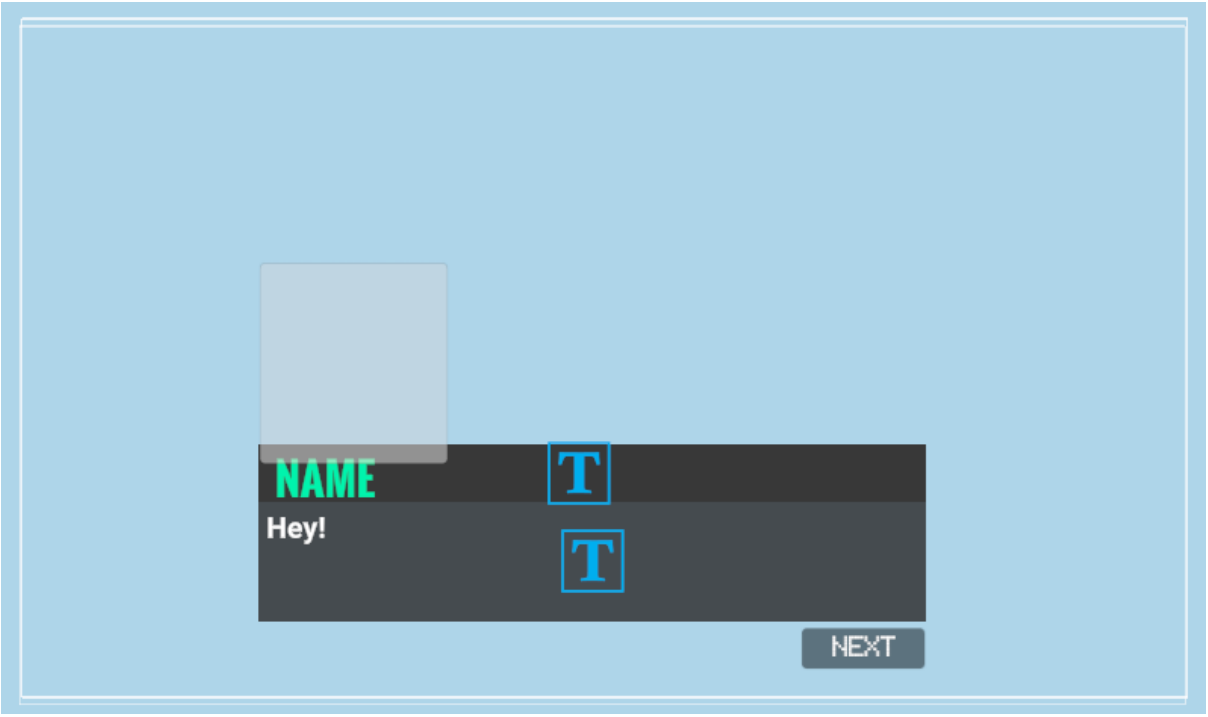


Рисунок 3.9 – Вікно діалогів

Усі репліки зберігаються у черзі, і при кожному виклику методу DisplayNextDialogueLine() відображається наступна з них.

Кожна репліка пов’язана з конкретним персонажем, що дозволяє змінювати іконку та ім’я залежно від того, хто говорить.

3.4 Тестування ігрового застосунку

Першим елементом для перевірки було обрано головне меню (рис. 3.10), оскільки це те, з чим взаємодіє гравець з самого початку.


Рисунок 3.10 – Інтерфейс головного меню гри

З головного меню було здійснено перехід до ігрового режиму, де одразу генерувався рівень із кімнат (рис. 3.11).

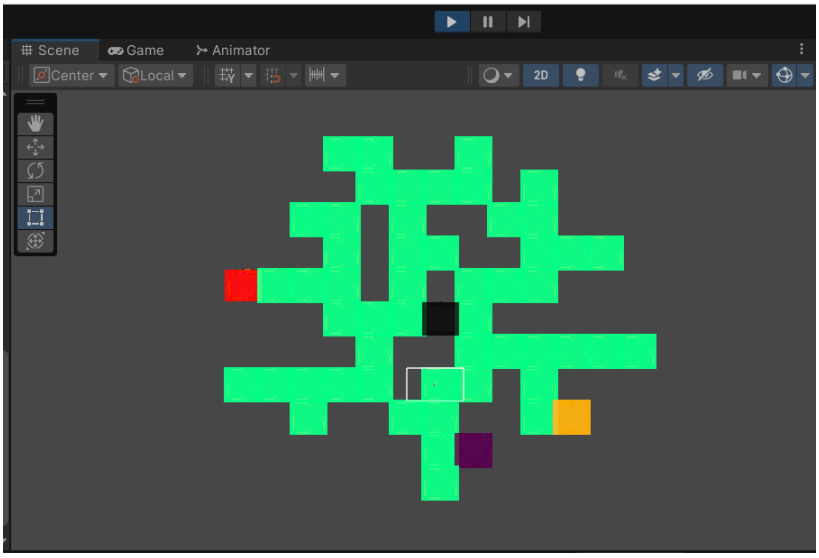


Рисунок 3.11 – Огляд згенерованого рівня зі сцени в Unity

Згенерований рівень містить основні типи кімнат, такі як скарбниці, що позначені жовтим кольором, магазин, який на зображені позначено

фіолетовим, кімнати боса позначено червоним, а зелений колір означає звичайні кімнати.

Перед початком кожного рівня гравцеві пропонується обрати дві базові навички, для використання протягом проходження рівня (рис. 3.12).


Рисунок 3.12 – Меню вибору навичок перед початком рівня

Після вибору двох активних навичок, гравець потрапляє до самого рівня з кімнатами (рис. 3.13).


Рисунок 3.13 – Кімната на рівні з погляду гравця

В більшості кімнат на персонажа будуть нападати монстри, яких він має перемогти, щоб просунутися далі по рівню (рис. 3.14).


Рисунок 3.14 – Приклад кімнати з монстрами

У скарбницях гравець може зустріти скрині з артефактами, відкривши їх і підібравши артефакт, він надасть гравцеві нові посилення (рис. 3.15).

Рисунок 3.15 – Приклад кімнати скарбниці

Деякі предмети додають союзника, який буде допомагати гравцеві у проходження рівня, Приклад союзника, у вигляді привида, що атакує монстрів разом з гравцем наведено на рисунку 3.16.

Рисунок 3.16 – Вигляд союзника у грі

В кінці рівня на гравця чекає бос, якого він повинен перемогти, щоб успішно пройти рівень (рис. 3.17).


Рисунок 3.17 – Битва з босом в кінці рівня

При успішному проходженні усіх кімнат та фінального боса, рівень завершується, а гравця повідомляють про перемогу (рис. 3.18), після цього гравець може спробувати пройти новий рівень або повернутися до головного меню, звідки продовжити гру в іншому режими або вийти із застосунку.


Рисунок 3.18 – Екран перемоги в грі

Якщо ж запас здоров’я гравця опускається до нуля це спричиняє поразку, а перед гравцем з’являється відповідний екран (рис. 3.19).


Рисунок 3.19 – Екран поразки в грі

У сюжетному режимі система діалогів використовується для представлення історії гри, знайомства гравця з персонажами та передачі важливої інформації про завдання (рис. 3.20).

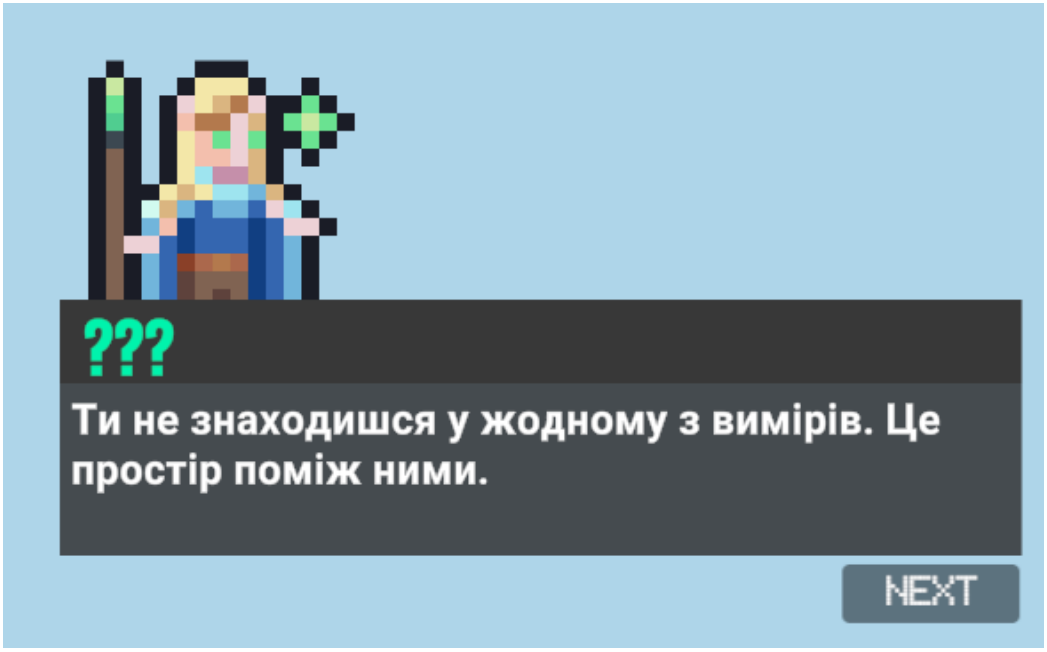


Рисунок 3.20 – Приклад діалогу з сюжету гри

Діалог автоматично запускається при вході гравця в певну зону або після взаємодії з певним об’єктом. Діалоги виводяться по одній репліці за натисканням клавіші або кнопки.

У процесі тестування гри було перевірено ключові аспекти її функціональності та взаємодії. Увагу зосереджено на стабільності геймплею, правильній роботі ігрових механік, зокрема бою, переміщення, системи артефактів, союзників, діалогів та інтерфейсу користувача. Також було протестовано процедурну генерацію рівнів, логіку переходів між кімнатами та поведінку ворогів.

# 4 ТЕХНІКО-ЕКОНОМІЧНЕ ОБГРУНТУВАННЯ

## 4.1. Аналіз ринку збуту та конкуренції

Гра, яка розробляється це 2D roguelike-гра, яка поєднує в собі процедурну генерацію рівнів і нестандартні бойові механіки. Завдяки ігровій системі з динамічним геймплеєм та високою реіграбельностю, проєкт може мати популярність гравців навіть через довгий час після випуску, він орієнтований на молодь і дорослих гравців віком від 14 до 35 років. Ця аудиторія є найактивнішою серед поціновувачів інді-ігор, а також найчастіше купує ігри через цифрові платформи, зокрема Steam та itch.io. Вибір саме цих платформ є стратегічно доцільним, оскільки вони забезпечують глобальний доступ до продукту, мають зручні інструменти для розробників і надають безпосередній зв’язок з аудиторією.

Сучасний ринок інді-ігор демонструє стабільне зростання завдяки низькому порогу входу для нових розробників і популяризації цифрової дистрибуції. Сервіси на зразок Steam Direct і itch.io суттєво спрощують процес публікації ігор, що дало змогу великій кількості незалежних студій виходити на ринок без підтримки великих видавців. Попит на проєкти у жанрі roguelike залишається високим, що підтверджується успіхом таких проєктів, як The Binding of Isaac, Dead Cells та Hades. Ці ігри не лише досягли значних комерційних показників, але й сформували навколо себе стійке гравецьке ком’юніті, що сприяє довготривалому інтересу до жанру.

Гра має низку важливих конкурентних переваг, які відрізняють його від аналогічних проєктів. Основною технічною особливістю є використання компонентно-орієнтованої архітектури, що забезпечує високу модульність системи. Це дозволяє розширювати функціональність гри без порушення основної структури та значно спрощує подальше оновлення контенту. Одним з елементів, які вирізняють гру є гнучка система генерації рівнів, що гарантує унікальність кожного проходження та стимулює повторне проходження гри.

План реалізації проєкту передбачає публікацію гри через сервіс Steam Direct. Вартість розміщення складає 100 доларів США, але вона повертається розробнику після досягнення іграми прибутку у розмірі 1000 доларів, тому ці витрати не вносяться у фінальну кошторисну частину. Основна маркетингова стратегія включає просування гри через спеціалізовані Discord-спільноти, де збирається цільова аудиторія інді-ігор, а також через співпрацю з авторами на платформах Twitch і YouTube, які мають можливість демонструвати ігровий процес живій аудиторії. Крім цього, проєкт буде просуватись у соціальних мережах, зокрема на Reddit та TikTok, що дозволить охопити ширшу аудиторію. Додатковим інструментом популяризації стане участь у фестивалях, присвячених інді-іграм, таких як Steam Indie Festival чи тематичні події на itch.io, де проєкт може отримати перші відгуки, підтримку від гравців і збільшити впізнаваність гри.

## 4.2. Розрахунок витрат на розробку

Детальний аналіз фінансових витрат на розробку гри, а також оцінку економічної доцільності реалізації проєкту на ринку інді-ігор є надзвичайно важливим етапом для кожного проєкту. У процесі створення гри було задіяно чотири спеціалісти з відповідною погодинною оплатою праці: гейм-дизайнер, Unity C# розробник, 2D-художник та тестувальник. Загальна вартість робочого часу склала 190 000 гривень. З цієї суми найбільшу частку зайняла заробітна плата програміста, що пояснюється більш тривалим обсягом його участі та вищою ставкою за годину. Сума заробітної плати кожного члена команди була розрахована за простою формулою: погодинна ставка помножена на кількість годин, відпрацьованих за час реалізації проєкту.

Програміст зайняв найбільшу частку бюджету, а саме 90 000 гривень, що становить майже половину всіх витрат на оплату праці, оскільки саме Unity розробник відповідає за реалізацію ключових механік гри, включаючи складні системи процедурної генерації рівнів, бойову систему та інші технічні аспекти. Високі витрати на цю позицію обумовлені як тривалим терміном

участі, так і високою погодинною ставкою, що відображає високу кваліфікацію необхідну для виконання таких робіт.

Гейм-дизайнер, який відповідав за загальну концепцію гри, баланс ігрових механік та проектування рівнів, відпрацював 150 годин за ставкою 300 гривень на годину, що склало 45 000 гривень. Ця позиція є особливо важливою на початкових етапах розробки, коли формується основа ігрового досвіду. Хоча витрати на гейм-дизайн і поступаються витратам на програмування, їхня роль у створенні успішного продукту не менш важлива.

2D-художник, для створення візуального стилю гри, анімацій та графічних елементів, відпрацював 100 годин за ставкою 400 гривень на годину, що в підсумку склало 40 000 гривень. Вартість роботи художника обумовлена необхідністю створення великої кількості унікальних графічних елементів, які є особливо важливими для 2D-ігор, де візуальна складова грає ключову роль у залученні гравців.

Тестувальник, який забезпечував якість готового продукту, виявивши помилки та недоліки в ігровому процесі, відпрацював 50 годин за ставкою 300 гривень на годину, що склало 15 000 гривень. Хоча ця позиція займає найменшу частку в бюджеті, її важливість не варто недооцінювати, оскільки саме тестування дозволяє випустити на ринок якісний продукт, вільний від критичних помилок.

Погодинні ставки та загальні суми зарплат наведено в таблиці 4.1.

Таблиця 4.1 – Погодинні ставки заробітної плати працівників

| Посада | Ставка (грн/год) | Кількість годин | Сума (грн) |
|---|---|---|---|
| Гейм-дизайнер | 300 | 150 | 45 000 |
| Unity C# розробник | 500 | 180 | 90 000 |
| 2D-художник | 400 | 100 | 40 000 |
| Тестувальник | 300 | 50 | 15 000 |
| Разом | | | 190 000 |

До суми заробітної плати додається обов’язкове нарахування Єдиного соціального внеску (ЄСВ)

Відповідно до чинного законодавства, на заробітну плату нараховується Єдиний соціальний внесок у розмірі 22%.

Свн = Сзп × 0.22 = 190000 × 0.22 = 41 800 грн

Додаткові витрати були пов’язані з придбанням необхідного програмного забезпечення. Одноразова покупка Aseprite обійшлася у 500 гривень, а підписка на Photoshop на період розробки тривалістю три місяці склала 2250 гривень. У таблиці 4.2 наведено загальні витрати на програмне забезпечення, яке використовувалося при створені проєкту.

Таблиця 4.2 – Витрати на програмне забезпечення

| Найменування | Тип ліцензії | Термін | Вартість за період (грн) |
|---|---|---|---|
| Aseprite | Постійна | | 500 |
| Photoshop | Щомісячна | 3 міс. | 750 × 3 = 2 250 |
| Разом | | | 2 750 |

Для забезпечення роботи тестового середовища протягом двох місяців було орендовано віртуальний сервер (VPS), що коштувало 150 гривень на місяць. Усього це склало 300 гривень. Також були закладені інші витрати, до яких відносяться оплата оренди домену, канцелярії тощо. Усі витрати на різні пристрої, необхідні для розробки, наведено у таблиці 4.3.

Таблиця 4.3 – Витрати на електроенергію

| Пристрій | Потужність (Вт) | Час роботи (год) | Тариф (грн/кВт·год) | Сума (грн) |
|---|---|---|---|---|
| Ноутбук x5 | 100x5 | 80x5 | 6.0 | 48.00x5 |
| Маршрутизатор | 40 | 80 | 6.0 | 19.20 |
| Принтер 3 в 1 | 300 | 5 | 6.0 | 9.00 |
| Разом | | | | 268.20 |

Сума таких витрат становила 2850 гривень. Орієнтовні витрати на електроенергію в період активної розробки склали ще 268.20 гривень.

У підсумку, повні витрати на розробку гри становлять 237 946 гривень. Це включає оплату праці, податкові зобов'язання, програмне забезпечення, інфраструктуру та інші операційні витрати (таблиця 4.4).

Таблиця 4.4 – Сумарні витрати на розробку проєкту

| Категорія | Сума (грн) |
|---|---|
| Заробітна плата | 190 000 |
| ЄСВ (22%) | 41 800 |
| Програмне забезпечення | 2 750 |
| Оренда VPS | 300 |
| Інші витрати (звук, домен) | 2 850 |
| Електроенергія | 246 |
| Разом | 237 946 |

Очікуваний обсяг продажів базується на плані реалізації 200 копій гри за ціною 400 гривень. Загальна сума виручки при цьому складає 800 000 гривень. Однак з цієї суми потрібно врахувати комісію цифрової платформи Steam, яка становить 30%. Це дорівнює 240 000 гривень. Залишок, що становить 560 000 гривень, оподатковується на 18% як прибуток. Податкове навантаження в такому випадку дорівнює 57 970 гривень. Таким чином, чистий дохід, який залишиться розробнику після всіх відрахувань, становить 264 084 гривень.

Термін окупності проєкту визначається як відношення загальних витрат до щорічного чистого прибутку. В нашому випадку:

Ток = Срозр / Чистий дохід = 237 946 / 264 084 ≈ 0.9 року (~11 місяців).

Це означає, що проєкт повністю окупить себе менш ніж за рік. Окрім цього, це дозволяє розрахувати коефіцієнт ефективності інвестицій, який визначається як відношення чистого доходу до загальних витрат.

У випадку цього проєкту показник становить:

Кеф = Чистий прибуток / Витрати = 264 084 / 237 946 ≈ 1.11 Це свідчить про те, що кожна вкладена гривня повертає приносить 1.11 прибутку за рік.

4.3 Обґрунтування доцільності розробки

Проєкт представляє собою комплексне рішення, яке поєднує в собі різноманітні аспекти сучасної розробки ігор.

Важливою перевагою проєкту є його модульна архітектура, яка дозволяє легко розширювати функціонал гри в майбутньому. Це відкриває широкі можливості для подальшого вдосконалення продукту, додавання нового контенту або навіть створення цілих доповнень. Такий підхід значно підвищує потенційний термін життя гри на ринку та забезпечує додаткові джерела доходу в довгостроковій перспективі.

З економічної точки зору проєкт демонструє збалансоване співвідношення інвестицій та потенційного прибутку. Термін окупності близько 11 місяців є цілком прийнятним показником для незалежного ігрового проєкту.

Економічна модель проєкту враховує основні ринкові реалії цифрової дистрибуції ігор. Вартість гри встановлена на рівні, який є конкурентоспроможним для інді-сегменту, при цьому дозволяє покрити витрати на розробку та отримати прибуток у разі досягнення запланованих обсягів продажів. Важливим фактором є те, що проєкт не вимагає додаткових інвестицій після виходу на ринок, а всі подальші оновлення можуть фінансуватися з отриманого прибутку.

Соціальний аспект проєкту також вартий уваги. Створення якісної ігрової продукції сприяє розвитку вітчизняної індустрії розваг, підвищує її престиж на міжнародній арені та створює додаткові робочі місця в креативній галузі. Успіх подібних проєктів може стати стимулом для інших молодих розробників, демонструючи, що створення комерційно успішних ігор в Україні є цілком реальним завданням.

# ВИСНОВКИ

У процесі реалізації проєкту було досягнуто поставленої мети, а саме створення функціонального прототипу 2D гри в жанрі roguelike, що забезпечує динамічний геймплей на основі процедурної генерації рівнів і варіативної поведінки ігрових об’єктів. Було виконано основні завдання, необхідні для функціонування гри, розроблено компонентну архітектуру гри, реалізовано логіку гравця, ворогів і союзників, впроваджено систему статус-ефектів, артефактів, інтерфейсу користувача, меню налаштувань, магазину та підтримку масштабування ігрових механік.

Створено систему процедурної генерації рівнів, засновану на використанні заздалегідь підготовлених кімнат у вигляді префабів. Забезпечено гнучке керування параметрами генерації та розширення структури рівня. Для реалізації основної логіки гравця розроблено механізми пересування, стрільби, застосування здібностей, взаємодії з союзниками та артефактами. Поведінка ворогів і союзників базується на окремих моделях та шаблонах дій, що легко масштабується за допомогою спільних базових класів.

У межах системи артефактів реалізовано зміну характеристик гравця, модифікацію візуальних ефектів снарядів та виклик союзників. Система статус-ефектів уніфікована та застосовується до всіх ворогів незалежно від типу. Для покращення взаємодії з користувачем створено адаптивний інтерфейс з відображенням основної інформації, а також меню налаштувань із можливістю конфігурації графіки, звуку, управління та екранного режиму.

Результати проєкту мають практичну цінність як основа для подальшого створення повноцінного ігрового продукту, а також як приклад впровадження компонентно-орієнтованого підходу в розробці ігор на рушії Unity.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. Annual Essential Facts About the U.S. Video Game Industry Report. ESA : вебсайт. URL: https://www.theesa.com/resources/essential-facts-about-the-us-video-game-industry/ (дата звернення: 04.02.2025).

2. What Is A Role-Playing Game? RPG Types Explained. Forbes : вебсайт. URL: https://www.forbes.com/sites/technology/article/what-are-rpg-games/ (дата звернення: 05.02.2025).

3. History of the Roguelike, from Rogue to Hades. Screenran : вебсайт. URL: https://screenrant.com/roguelike-definition-games-rogue-hades-roguelite-dungeon-crawler/ (дата звернення: 07.02.2025).

4. Hades. Supergiant Games : вебсайт. URL: https://www.supergiantgames.com/games/hades/ (дата звернення: 10.02.2025).

5. About Dead Cells. Dead Cells : вебсайт. URL: https://dead-cells.com/ (дата звернення: 12.02.2025).

6. Elysian Realm. Honkai Impact 3 Wiki : вебсайт. URL: https://honkaiimpact3.fandom.com/wiki/Elysian_Realm (дата звернення: 13.02.2025).

7. Get Started with UE4. Epic Games Dev : вебсайт. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/get-started-with-ue4?application_version=4.27 (дата звернення: 27.02.2025).

8. Cocos2d-x - Mature, lightweight, open cross-platform solution. Cocos2d-x : вебсайт. URL: https://www.cocos.com/en/cocos2d-x (дата звернення: 28.02.2025).

9. Unity Engine. Unity : вебсайт. URL: https://unity.com/products/unity-engine (дата звернення: 04.03.2025).

10. Микитович М. Дослідження алгоритмів процедурної генерації та їх застосування. МАТЕРІАЛИ СТУДЕНТ. НАУКОВО-ПРАКТ. КОНФ., м. Тернопіль, 15 травня 2025 р. Тернопіль, 2025.

# ДОДАТКИ

## Додаток А

## Лістинг скрипту PlayerController

```
using System.Collections.Generic;
using UnityEditor.Experimental;
using UnityEngine;
public class PlayerController : MonoBehaviour{
    private Animator animator;
    private Vector2 movement;
    private bool isShooting = false;
    private PlayerStats playerStats;
    private Rigidbody2D rb;
    public GameObject bulletPrefab;
    public List<ArtifactBase> artifacts = new
List<ArtifactBase>();
    void Start(){
        animator = GetComponentInChildren<Animator>();
        playerStats = GetComponent<PlayerStats>();
        rb = GetComponentInChildren<Rigidbody2D>();
        if (rb == null){
            Debug.LogError("Rigidbody2D не знайдено серед
дочірніх об'єктів!");
        }
        if (playerStats == null){
            Debug.LogError("PlayerStats не знайдено!");
        }
    }
    void Update(){
        HandleMovementInput();
        HandleShootingInput();
        UpdateAnimations();
    }
    void FixedUpdate(){
        MovePlayer();
    }
    private void HandleMovementInput(){
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
    }
    private void HandleShootingInput(){
        if (isShooting) return;
        Vector2 shootingDirection = Vector2.zero;
        if (Input.GetKey(KeyCode.I)) shootingDirection +=
Vector2.up;
        if (Input.GetKey(KeyCode.K)) shootingDirection +=
Vector2.down;
        if (Input.GetKey(KeyCode.J)) shootingDirection +=
Vector2.left;
        if (Input.GetKey(KeyCode.L)) shootingDirection +=
Vector2.right;
```

```
        if (shootingDirection.sqrMagnitude > 0){
            shootingDirection.Normalize();
            isShooting = true;
            Shoot(shootingDirection);
            HandleShootingAnimation(shootingDirection);
            Invoke(nameof(ResetShooting), 0.2f);
        }
    }
    private void MovePlayer(){
        if (rb == null || playerStats == null){
            return;
        }
        if (movement.sqrMagnitude == 0){
            rb.velocity = Vector2.zero;
            return;
        }
        rb.velocity = movement.normalized *
playerStats.moveSpeed;
    }
public void AddArtifact(ArtifactBase artifact){
        if (!IsRepeatableArtifact(artifact) &&
HasArtifact(artifact)){
            Debug.Log($"Artifact {artifact.artifactName} already
exists in inventory");
            return;
        }
        artifacts.Add(artifact);
        artifact.OnPickup(this);
}
    private bool HasArtifact(ArtifactBase artifact){
        return artifacts.Exists(a => a.GetType() ==
artifact.GetType());
    }
    private bool IsRepeatableArtifact(ArtifactBase artifact){
        return false;
    }
    private void UpdateAnimations(){
        bool isMoving = movement.sqrMagnitude > 0;
        animator.SetBool("IsRunning", isMoving);
        animator.SetBool("IsIdle", !isMoving);
        if (isMoving){
            animator.SetFloat("MoveX", movement.normalized.x);
            animator.SetFloat("MoveY", movement.normalized.y);
        }
    }
    public void Shoot(Vector2 direction){
        GameObject bullet = Instantiate(bulletPrefab,
transform.position, Quaternion.identity);
        Rigidbody2D bulletRb =
bullet.GetComponent<Rigidbody2D>();
        if (bulletRb != null){
            bulletRb.velocity = direction *
playerStats.bulletSpeed;}
```

```
foreach (var artifact in artifacts){
            if (artifact is IBulletModifier modifier){
                modifier.ModifyBullet(bullet);
            }
        }
    }
    private void HandleShootingAnimation(Vector2 direction){
        string                  animationTrigger                =
GetShootingAnimationTrigger(direction);
        if (!string.IsNullOrEmpty(animationTrigger)){
            animator.SetTrigger(animationTrigger);
        }
    }

    private string GetShootingAnimationTrigger(Vector2 direction)
{
        if  (Vector2.Dot(direction,  Vector2.up)  >  0.9f)  return
"Bow_Attack_Up";
        if  (Vector2.Dot(direction,  Vector2.down)  >  0.9f)  return
"Bow_Attack_Down";
        if  (Vector2.Dot(direction,  Vector2.left)  >  0.9f)  return
"Bow_Attack_Left";
        if  (Vector2.Dot(direction,  Vector2.right)  >  0.9f)  return
"Bow_Attack_Right";
        if  (Vector2.Dot(direction,  new  Vector2(1,  1).normalized)
> 0.9f)  return "Bow_Attack_Up_Right";
        if  (Vector2.Dot(direction,  new  Vector2(-1,  1).normalized)
> 0.9f)  return "Bow_Attack_Up_Left";
        if  (Vector2.Dot(direction,  new  Vector2(1,  -1).normalized)
> 0.9f)  return "Bow_Attack_Down_Right";
        if    (Vector2.Dot(direction,    new    Vector2(-1,    -
1).normalized)  >  0.9f)  return "Bow_Attack_Down_Left";
        return "";
    }

    private void ResetShooting(){
        isShooting = false;
    }
}
```

Додаток Б

Лістинг скрипту PlayerStats

```
using System.Collections.Generic;
using UnityEditor.Experimental;
using UnityEngine;
public class PlayerStats : MonoBehaviour{
    public int maxHealth = 100;
    public int currentHealth;
    public int attackPower = 10;
    public int defense = 5;
    public float moveSpeed = 9f;
    public float bulletSpeed = 10f;
    private SpriteRenderer spriteRenderer;
    void Start(){
        currentHealth = maxHealth;
        spriteRenderer                                                    =
GetComponentInChildren<SpriteRenderer>();
    }
    public void TakeDamage(int damage){
        int damageTaken = Mathf.Max(damage - defense, 0);
        currentHealth -= damageTaken;
        Debug.Log($"Player took {damageTaken} damage. Current HP:
{currentHealth}");
        StartCoroutine(FlashRed());
        if (currentHealth <= 0){
            Die();}
}
    public void Heal(int amount){
        currentHealth   =   Mathf.Min(currentHealth   +   amount,
maxHealth);
        Debug.Log($"Healed to: {currentHealth}");
}
    private void Die(){
        Debug.Log("Player has died.");
}
    private IEnumerator FlashRed() {
        spriteRenderer.color = Color.red;
        yield return new WaitForSeconds(0.2f);
        spriteRenderer.color = Color.white;
    }
}
```

Додаток В

Лістинг коду класу ChestController

```
public class ChestController : MonoBehaviour{
    [Header("References")]
    public SpriteRenderer spriteRenderer;
    public Sprite openedSprite;
    public GameObject artifactSpawnPoint;
    public ParticleSystem openEffect;
    public ArtifactDatabase artifactDatabase;
    public float spawnOffset = 1f;
    private bool isOpened = false;
    private GameObject spawnedArtifact;
    private void OnTriggerEnter2D(Collider2D other){
        if (!isOpened && other.CompareTag("Player")){
            OpenChest();
        }
    }
    private void OpenChest(){
        isOpened = true;
        spriteRenderer.sprite = openedSprite;
        if (openEffect != null){
            openEffect.Play();
        }
        SpawnArtifact();
    }
    private void SpawnArtifact(){
        ArtifactBase                    artifact                    =
artifactDatabase.GetRandomArtifact();
        if (artifact == null) return;
        Vector3 spawnPos = artifactSpawnPoint != null ?
            artifactSpawnPoint.transform.position :
            transform.position + Vector3.up * spawnOffset;
        GameObject          artifactObj          =          new
GameObject(artifact.artifactName);
        artifactObj.transform.position = spawnPos;
        SpriteRenderer                      sr                      =
artifactObj.AddComponent<SpriteRenderer>();
        sr.sprite = artifact.icon;
        CircleCollider2D                collider                =
artifactObj.AddComponent<CircleCollider2D>();
        collider.isTrigger = true;
        ArtifactPickup                  pickup                  =
artifactObj.AddComponent<ArtifactPickup>();
        pickup.artifactData = artifact;
    }
}
```

Додаток Г

Лістинг коду класу EnemyBase

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using System;
public class EnemyBase : MonoBehaviour
{
    public int maxHP;
    public int currentHP;
    public float moveSpeed;
    public int damage;
    public float attackRange;
    public float attackCooldown;
    public Rigidbody2D rb;
    protected Transform deathEffect;
    protected Transform player;
    public float lastAttackTime;
    protected virtual void Start(){
        currentHP = maxHP;
        player                                                    =
GameObject.FindGameObjectWithTag("Player").transform;
        if (player == null){
            Debug.LogError("Player not found!");
        }
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer                                            =
GetComponentInChildren<SpriteRenderer>();
        originalSpeed = moveSpeed;
    }
    protected virtual void Update(){
        if (player == null){
            GameObject                playerObj                   =
GameObject.FindGameObjectWithTag("Player");
            if (playerObj != null) {
                player = playerObj.transform;
            }
            return;
        }
        float deltaTime = Time.deltaTime;
        foreach (var effect in activeEffects.ToList()){
            effect.Update(this, deltaTime);
        }
        if (!isStunned){
            MoveTowardsPlayer();
            TryAttackPlayer();
        }
    }
    private    List<StatusEffect>    activeEffects    =    new
List<StatusEffect>();
    private SpriteRenderer spriteRenderer;
```

```
    private bool isStunned = false;
    private float originalSpeed;
    private GameObject activeVisualEffect;
    public void ApplyEffect(StatusEffect effect){
        ClearEffect(effect.Type);
        activeEffects.Add(effect);
        switch (effect.Type){
            case EffectType.Stun:
                isStunned = true;
                rb.velocity = Vector2.zero;
                break;
            case EffectType.Slowed:
                moveSpeed = originalSpeed * 0.5f;
                break;
        }
       if (spriteRenderer != null && effect.EffectColor.HasValue){
            spriteRenderer.color = effect.EffectColor.Value;
        }
       if (effect.VisualEffectPrefab != null){
            activeVisualEffect                                    =
Instantiate(effect.VisualEffectPrefab, transform);
        }
        if (effect.SoundEffect != null){
            AudioSource.PlayClipAtPoint(effect.SoundEffect,
transform.position);
        }
    }
   public void ClearEffect(EffectType type){
        activeEffects.RemoveAll(e => e.Type == type);
        switch (type){
            case EffectType.Stun:
                isStunned = false;
                break;
            case EffectType.Slowed:
                moveSpeed = originalSpeed;
                break;
        }
        if (spriteRenderer != null){
            spriteRenderer.color = Color.white;
        }
        if (activeVisualEffect != null){
            Destroy(activeVisualEffect);
        }
    }
    protected virtual void MoveTowardsPlayer(){
        if (rb != null && player != null){
            Vector2    direction    =    (player.position    -
transform.position).normalized;
            rb.velocity = direction * moveSpeed;
        }
    }
    protected virtual void TryAttackPlayer(){
```

```
float distanceToPlayer = Vector2.Distance(transform.position,
player.position);
        if (distanceToPlayer <= attackRange && Time.time -
lastAttackTime >= attackCooldown){
            AttackPlayer();
            Debug.Log("Trying to attack player...");
            lastAttackTime = Time.time;
        }
    }
    public virtual void TakeDamage(float amount){
        TakeEffectDamage(amount);
    }
    public event Action OnDeath;
    public virtual void TakeEffectDamage(float amount){
        StartCoroutine(FlashRed());
        currentHP -= Mathf.RoundToInt(amount);
        if (currentHP <= 0){
            Die();
        }
    }
    protected virtual void Die(){
        OnDeath?.Invoke();
        Destroy(gameObject);
    }
    protected virtual void AttackPlayer(){
        Debug.Log("Attacking player!");
        PlayerStats playerStats =
player.GetComponent<PlayerStats>();
        if (playerStats != null){
            playerStats.TakeDamage(damage);
        }
        else{
            Debug.LogError("PlayerStats component not found on
player!");
        }
    }
    private IEnumerator FlashRed(){
        spriteRenderer.color = Color.black;
        yield return new WaitForSeconds(0.18f);
        spriteRenderer.color = Color.red;
        yield return new WaitForSeconds(0.18f);
        spriteRenderer.color = Color.white;
    }
}
```

Додаток Ґ

Лістинг коду скрипту BossBase

```
public abstract class BossBase : MonoBehaviour{
    public string bossName;
    public int maxHP = 500;
    public int currentHP;
    public bool isVulnerable = true;
    public GameObject entranceVFX;
    public GameObject deathVFX;
    public AudioClip entranceSound;
    public GameObject artifactPickupPrefab;
    public int rewardCount = 3;
    protected Animator animator;
    protected Transform player;
    protected virtual void Start(){
        currentHP = maxHP;
        animator = GetComponent<Animator>();
        player                                                    =
GameObject.FindGameObjectWithTag("Player").transform;
        PlayEntranceEffects();}
    private void PlayEntranceEffects(){
        if (entranceVFX != null)
            Instantiate(entranceVFX,          transform.position,
Quaternion.identity);
        if (entranceSound != null)
            AudioSource.PlayClipAtPoint(entranceSound,
transform.position);}
    public virtual void TakeDamage(int damage){
        if (!isVulnerable) return;
        currentHP -= damage;
        if (currentHP <= 0){
            Die();}
        else{
            if (animator != null)
                animator.SetTrigger("Hurt");}}
    protected virtual void Die(){
        if (deathVFX != null)
            Instantiate(deathVFX,             transform.position,
Quaternion.identity);
        SpawnRewards();
        Destroy(gameObject);}
   protected void SpawnRewards(){
        for (int i = 0; i < rewardCount; i++){
            Vector2  spawnPos  =  (Vector2)transform.position  +
Random.insideUnitCircle * 2f;
            Instantiate(artifactPickupPrefab,          spawnPos,
Quaternion.identity);}}
    public float GetHealthPercentage(){
        return (float)currentHP / maxHP;}}
```

Додаток Е

Лістинг коду класу AllyBase

```
public abstract class AllyBase : MonoBehaviour{
    public float moveSpeed = 2f;
    public float detectionRange = 5f;
    public float attackRange = 3f;
    public float attackCooldown = 1.5f;
    public int damage = 10;
    public float followDistance = 1.5f;
    public float smoothTime = 0.3f;
    protected Transform player;
    protected float lastAttackTime;
    protected Vector2 currentVelocity;
    protected virtual void Start(){
player = GameObject.FindGameObjectWithTag("Player")?.transform;
        if (player == null){
            Debug.LogError("Player not found!");
            enabled = false;}}
    protected virtual void Update(){
        if (player == null) return;
        FollowPlayer();
        LookForEnemies();}
    protected virtual void FollowPlayer(){
        Vector2 targetPosition = (Vector2)player.position -
(GetFollowOffset() * followDistance);
        transform.position = Vector2.SmoothDamp(
transform.position, targetPosition, ref currentVelocity,
smoothTime,moveSpeed);}
    protected virtual Vector2 GetFollowOffset(){
        return Random.insideUnitCircle.normalized;}
    protected abstract void LookForEnemies();
    protected virtual bool IsEnemyInRange(Transform enemy, float
range){
        return Vector2.Distance(transform.position,
enemy.position) <= range;}
    protected virtual GameObject FindNearestEnemy(){
        GameObject[] enemies =
GameObject.FindGameObjectsWithTag("Enemy");
        GameObject nearest = null;
        float minDistance = Mathf.Infinity;
        foreach (var enemy in enemies){
            float distance = Vector2.Distance(transform.position,
enemy.transform.position);
            if (distance < minDistance && distance <=
detectionRange){
                minDistance = distance;
                nearest = enemy;}}
        return nearest;}}
```

```
public class AllyProjectile : MonoBehaviour{
    public float speed = 10f;
    public int damage = 10;
    public float lifetime = 2f;
    public GameObject impactVFX;
    private Vector2 direction;
    private Rigidbody2D rb;
    private void Awake(){
        rb = GetComponent<Rigidbody2D>();
        Destroy(gameObject, lifetime);
    }
    public   void   Initialize(Vector2   targetDirection,   int
projectileDamage, GameObject customImpactVFX = null)
    {
        direction = targetDirection.normalized;
        damage = projectileDamage;
        if (customImpactVFX != null){
            impactVFX = customImpactVFX;
        }
    }
    private void FixedUpdate(){
        if (rb != null){
            rb.velocity = direction * speed;}
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Enemy")){
            EnemyBase                    enemy                    =
collision.GetComponent<EnemyBase>();
            if (enemy != null){
                enemy.TakeDamage(damage);
            }
            if (impactVFX != null){
                Instantiate(impactVFX,        transform.position,
Quaternion.identity);
            }
            Destroy(gameObject);
        }
    }
}
```

Додаток Є

Програмний код скрипту LevelGenerator

```
public class LevelGenerator : MonoBehaviour{
    public GameObject spawnRoomPrefab;
    public GameObject regularRoomPrefab;
    public GameObject treasureRoomPrefab;
    public GameObject shopRoomPrefab;
    public GameObject bossRoomPrefab;
    public GameObject puzzleRoomPrefab;
    [Header("Generation Settings")]
    public int maxRooms = 10;
    public float roomSize = 15f;
    public float roomSpacing = 2f;
    private List<Vector2Int> occupiedPositions = new
List<Vector2Int>();
    private Dictionary<Vector2Int, Room> rooms = new
Dictionary<Vector2Int, Room>();
    private Vector2Int[] directions = {
        Vector2Int.up, Vector2Int.down, Vector2Int.left,
Vector2Int.right
    };
    void Start(){
        GenerateLevel();
    }
    void GenerateLevel(){
        bool validLayout = false;
        int attempts = 0;
        const int maxAttempts = 100;
        while (!validLayout && attempts < maxAttempts){
            attempts++;
            ClearExistingLevel();
            Vector2Int start = Vector2Int.zero;
            occupiedPositions.Add(start);
            PlaceRoom(spawnRoomPrefab, start, RoomType.Spawn);
            Queue<Vector2Int> expansionQueue = new
Queue<Vector2Int>();
            expansionQueue.Enqueue(start);
            int roomsCreated = 1;
            while (roomsCreated < maxRooms && expansionQueue.Count
> 0) {
                Vector2Int current = expansionQueue.Dequeue();
                List<Vector2Int> validDirs =
GetValidDirections(current);
                int branches = Mathf.Min(validDirs.Count,
Random.Range(1, 3));
                for (int i = 0; i < branches && roomsCreated <
maxRooms; i++){
                    Vector2Int dir = validDirs[Random.Range(0,
validDirs.Count)];
                    Vector2Int newPos = current + dir;
                    occupiedPositions.Add(newPos);
```

```
                    GameObject          selectedPrefab          =
GetRandomRoomPrefab();
                    PlaceRoom(selectedPrefab,             newPos,
RoomType.Regular);
                    expansionQueue.Enqueue(newPos);
                    roomsCreated++;
                    validDirs.Remove(dir);}}
            if                    (TryPlaceSpecialRooms(start)){
validLayout = true;
                Debug.Log($"Level  generated  successfully  after
{attempts} attempts");}}
        if (!validLayout){
            Debug.LogError("Failed   to   generate   valid   level
layout");}
        SetupAllRoomDoors();}
    void ClearExistingLevel(){
        foreach (Transform child in transform)
            Destroy(child.gameObject);
        occupiedPositions.Clear();
        rooms.Clear();}
    List<Vector2Int> GetValidDirections(Vector2Int current){
        List<Vector2Int> validDirs = new List<Vector2Int>();
        foreach (Vector2Int dir in directions){
            Vector2Int next = current + dir;
            if (occupiedPositions.Contains(next))
                continue;
            if (CountNeighbors(next) <= 1)
                validDirs.Add(dir);}
        return validDirs;}
    List<Vector2Int> FindDeadEnds(Vector2Int start){
        List<Vector2Int> deadEnds = new List<Vector2Int>();
        foreach (var kvp in rooms){
            if (CountNeighbors(kvp.Key) == 1 && kvp.Key != start)
                deadEnds.Add(kvp.Key);}
        return deadEnds;}
    void   PlaceRoom(GameObject   prefab,   Vector2Int   position,
RoomType type){
        if (prefab == null){
            Debug.LogError("Prefab is null!");
            return;}
        Vector3 spawnPos = new Vector3(
            position.x * (roomSize + roomSpacing),
            position.y * (roomSize + roomSpacing),
            0);
        GameObject   roomObj   =   Instantiate(prefab,   spawnPos,
Quaternion.identity, transform);
        Room room = roomObj.GetComponent<Room>();
        if (room == null){
            Debug.LogError("Room component missing!");
            Destroy(roomObj);
            return;}
        room.roomType = type;
        room.position = position;
```

```
        rooms[position] = room;}
    void SetupAllRoomDoors(){
        foreach (var room in rooms){
            room.Value.SetDoors(rooms);}}
    bool Isolated(Vector2Int a, Vector2Int b, Vector2Int c){
        Vector2Int[] group = { a, b, c };
        for (int i = 0; i < group.Length; i++){
            for (int j = i + 1; j < group.Length; j++){
                if ((group[i] - group[j]).sqrMagnitude <= 2)
                    return false; }}
        return true;}
    int CountNeighbors(Vector2Int pos){
        int count = 0;
        foreach (Vector2Int dir in directions){
            if (occupiedPositions.Contains(pos + dir))
                count++;}
        return count;}
    void PlaceSpecialRooms(){
        List<Vector2Int> deadEnds = new List<Vector2Int>();
        foreach (var kvp in rooms){
            Vector2Int pos = kvp.Key;
            Room room = kvp.Value;
            if (room.roomType != RoomType.Regular) continue;
            int neighborCount = 0;
            foreach (var dir in directions){
                if (rooms.ContainsKey(pos + dir))
                    neighborCount++;}
            if (neighborCount == 1)
                deadEnds.Add(pos);}
        Shuffle(deadEnds);
        if (deadEnds.Count < 3){
            Debug.LogWarning("Недостатньо мертвих кутів для
спеціальних кімнат. Згенеруйте більше кімнат.");
            return;}
        PlaceSpecialRoomAt(deadEnds[0],        RoomType.Treasure,
treasureRoomPrefab);
        PlaceSpecialRoomAt(deadEnds[1],            RoomType.Shop,
shopRoomPrefab);
        PlaceSpecialRoomAt(deadEnds[2],            RoomType.Boss,
bossRoomPrefab);}
    private bool TryPlaceSpecialRooms(Vector2Int start){
        List<Vector2Int> deadEnds = FindDeadEnds(start);
        if (deadEnds.Count < 3) return false;
        deadEnds.Sort((a, b) =>
            (b      -      start).sqrMagnitude.CompareTo((a      -
start).sqrMagnitude));
        PlaceSpecialRoomAt(deadEnds[0],            RoomType.Boss,
bossRoomPrefab);
        PlaceSpecialRoomAt(deadEnds[1],        RoomType.Treasure,
treasureRoomPrefab);
        PlaceSpecialRoomAt(deadEnds[2],            RoomType.Shop,
shopRoomPrefab);
```

```
        return true;}
    void  PlaceSpecialRoomAt(Vector2Int  pos,  RoomType  type,
GameObject prefab){
        Destroy(rooms[pos].gameObject);
        PlaceRoom(prefab, pos, type);}
    void Shuffle<T>(List<T> list){
        for (int i = 0; i < list.Count; i++){
            int rand = Random.Range(i, list.Count);
            (list[i], list[rand]) = (list[rand], list[i]);}}
    void PlaceSpecialRoom(RoomType type, GameObject prefab){
        List<Vector2Int>      candidatePositions      =      new
List<Vector2Int>();
        foreach (var kvp in rooms){
            Vector2Int pos = kvp.Key;
            Room room = kvp.Value;
            if (room.roomType != RoomType.Regular && room.roomType
!= RoomType.Spawn)
                continue;
            int neighborCount = 0;
            bool valid = true;
            foreach (Vector2Int dir in directions){
                Vector2Int neighborPos = pos + dir;
                if (rooms.ContainsKey(neighborPos)){
                    Room neighbor = rooms[neighborPos];
                    if (neighbor.roomType != RoomType.Regular &&
neighbor.roomType != RoomType.Spawn){
                        valid = false;
                        break;}
                    neighborCount++;}}
            if (neighborCount == 1 && valid)
                candidatePositions.Add(pos);}
        if (candidatePositions.Count == 0){
            Debug.LogWarning($"No  valid  positions  for  {type}
room.");
            return;}
        Vector2Int                  selectedPos                  =
candidatePositions[Random.Range(0, candidatePositions.Count)];
        Destroy(rooms[selectedPos].gameObject);
        PlaceRoom(prefab, selectedPos, type);}
    GameObject GetRandomRoomPrefab(){
        GameObject[]   roomPrefabs   =   {   regularRoomPrefab,
puzzleRoomPrefab };
        return roomPrefabs[Random.Range(0, roomPrefabs.Length)];
    }
}
```

Додаток Ж

Лістинг коду скрипту необхідного для роботи HUD

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;
public class PlayerUI : MonoBehaviour
{
    public PlayerStats playerStats;
    public Image healthFillImage;
    public TextMeshProUGUI statsText;
    void Update()
    {
        if (playerStats != null)
        {
            UpdateHealthBar();
            UpdateStatsText();
        }
    }

    void UpdateHealthBar()
    {
        if (healthFillImage != null)
        {
            float fillAmount = (float)playerStats.currentHealth /
playerStats.maxHealth;
            healthFillImage.fillAmount = fillAmount;
        }
    }

    void UpdateStatsText()
    {
        if (statsText != null)
        {
            statsText.text = $"АТК: {playerStats.attackPower}\n"
+
                             $"ЗАХ: {playerStats.defense}\n" +
                             $"ШВ:  {playerStats.moveSpeed:F1}\n"
+
                             $"ШВК:
{playerStats.bulletSpeed:F1}";
        }
    }
}
```

Додаток І

Лістинг фрагменту коду системи налаштувань

```
public class SettingsManager : MonoBehaviour{
    public Slider volumeSlider;
    public Toggle fullscreenToggle;
    public Dropdown resolutionDropdown;
    Resolution[] resolutions;
    void Start()
    {
        volumeSlider.value = PlayerPrefs.GetFloat("volume", 1f);
        volumeSlider.onValueChanged.AddListener(SetVolume);
        fullscreenToggle.isOn = Screen.fullScreen;
fullscreenToggle.onValueChanged.AddListener(SetFullscreen);
        resolutions = Screen.resolutions;
        resolutionDropdown.ClearOptions();
        int currentResIndex = 0;
        var              options              =              new
System.Collections.Generic.List<string>();
        for (int i = 0; i < resolutions.Length; i++){
            string  option  =  resolutions[i].width  +  "  x  "  +
resolutions[i].height;
            options.Add(option);
            if              (resolutions[i].width              ==
Screen.currentResolution.width &&
                resolutions[i].height                          ==
Screen.currentResolution.height){
                currentResIndex = i;
            }
        }
        resolutionDropdown.AddOptions(options);
        resolutionDropdown.value = currentResIndex;
        resolutionDropdown.RefreshShownValue();
resolutionDropdown.onValueChanged.AddListener(SetResolution);
    }
    public void SetVolume(float value){
        AudioListener.volume = value;
        PlayerPrefs.SetFloat("volume", value);}
    public void SetFullscreen(bool isFullscreen){
        Screen.fullScreen = isFullscreen;
    }
    public void SetResolution(int index){
        Resolution res = resolutions[index];
        Screen.SetResolution(res.width,              res.height,
Screen.fullScreen);
    }
}
```

Додаток К

Лістинг скрипту для роботи магазину

```
public class ShopSlot : MonoBehaviour{
    public Text nameText;
    public Text priceText;
    public Button buyButton;
    public ArtifactBase artifact;
    public int price;
    private PlayerStats playerStats;
    private PlayerController playerController;
    public void Initialize(ArtifactBase artifactToSell, int cost,
PlayerStats stats, PlayerController controller){
        artifact = artifactToSell;
        price = cost;
        playerStats = stats;
        playerController = controller;
        nameText.text = artifact.artifactName;
        priceText.text = price.ToString();
        buyButton.onClick.RemoveAllListeners();
        buyButton.onClick.AddListener(Buy);
    }
    public void Buy(){
        if (playerStats == null || playerController == null)
return;
        if (playerStats.currentCoins >= price){
            playerStats.currentCoins -= price;
            artifact.OnPickup(playerController);
            buyButton.interactable = false;
            priceText.text = "Sold";
        }
        else{
            Debug.Log("Not enough coins");
        }
    }
}

public class ShopUI : MonoBehaviour{
    public ShopSlot[] slots;
    public ArtifactDatabase artifactDatabase;
    public PlayerStats playerStats;
    public PlayerController playerController;
    public void GenerateShop(){
        for (int i = 0; i < slots.Length; i++){
            ArtifactBase artifact =
artifactDatabase.GetRandomArtifact();
            int price = Random.Range(10, 21);
slots[i].Initialize(artifact, price, playerStats,
playerController);
        }
    }
}
```

Додаток Л

Лістинг коду скрипту системи діалогів

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;
using UnityEngine;
using TMPro;
public class DialogueManager : MonoBehaviour{
    public static DialogueManager Instance;
    public Image characterIcon;
    public TextMeshProUGUI characterName;
    public TextMeshProUGUI dialogueArea;
    private Queue<DialogueLine> lines;
    public bool isDialogueActive = false;
    public float typingSpeed = 50f;
    public Canvas dialogueCanvas;
    public Animator animator;
    private void Awake(){
        if (Instance == null){
            Instance = this;
        lines = new Queue<DialogueLine>();}
    public void StartDialogue(Dialogue dialogue){
        isDialogueActive = true;
        animator.Play("show");
        dialogueCanvas.enabled = true;
        lines.Clear();
        foreach       (DialogueLine       dialogueLine       in
dialogue.dialogueLines){
            lines.Enqueue(dialogueLine);}
        DisplayNextDialogueLine();}
    public void DisplayNextDialogueLine(){
        if (lines.Count == 0){
            EndDialogue();
            return;}
        DialogueLine currentLine = lines.Dequeue();
        characterIcon.sprite = currentLine.character.icon;
        characterName.text = currentLine.character.name;
        StopAllCoroutines();
        StartCoroutine(TypeSentence(currentLine));}
    IEnumerator TypeSentence(DialogueLine dialogueLine){
        dialogueArea.text = "";
        foreach (char letter in dialogueLine.line.ToCharArray()){
            dialogueArea.text += letter;
            yield return new WaitForSeconds(typingSpeed);}}
    void EndDialogue(){
        isDialogueActive = false;
        animator.Play("hide");
        dialogueCanvas.gameObject.SetActive(false);}}
```