

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**  
**"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**  
**імені ІГОРЯ СІКОРСЬКОГО"**  
**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ**  
**кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ**

"До захисту допущено"  
Завідувач кафедри ЦТЕ  
\_\_\_\_\_  
" " \_\_\_\_\_ Наталія АУШЕВА  
2026 р.

**Дипломна робота**  
**на здобуття ступеня бакалавра**  
**за освітньо-професійною програмою**  
**"Цифрові технології в енергетиці"**  
**зі спеціальності 122 "Комп'ютерні науки"**

**на тему: “Роуглайт гра з механізмами керування часом на базі Unity”**

Виконав:

студент IV курсу, групи ТР-25

ГЛИВЯК Артем Степанович

(ПРИЗВИЩЕ, ім'я, по батькові)

\_\_\_\_\_  
(підпис)

Керівник:

зав. каф, д.т.н., проф. Наталія АУШЕВА

(посада, науковий ступінь, вчене звання, ім'я, ПРИЗВИЩЕ)

\_\_\_\_\_  
(підпис)

Рецензент:

доцент каф. ТАЕ, к.т.н., Артур РАЧИНСЬКИЙ

(посада, науковий ступінь, вчене звання, ім'я, ПРИЗВИЩЕ)

\_\_\_\_\_  
(підпис)

Н.контроль:

ас. Костянтин ЗДОР

(посада, , ім'я, ПРИЗВИЩЕ)

\_\_\_\_\_  
(підпис)

Засвідчую, що у цій дипломній роботі немає  
запозичень з праць інших авторів без  
відповідних посилань.

Студент \_\_\_\_\_

(підпис)

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО”**

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти — перший (бакалаврський)

спеціальність 122 “Ком’ютерні науки”

Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ

Завідувач кафедри ЦТЕ

Наталія АУШЕВА  
“ ” 2026р.

**ЗАВДАННЯ**

**на дипломну роботу студенту**

Гливяку Артему Степановичу

(прізвище, ім’я, по батькові)

1. Тема роботи “Роуглайт гра з механізмами керування часом на базі Unity ”

Науковий керівник Аушева Наталія Миколаївна, д.т.н., проф.

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “28” травня 2026 року № 1992-с

2. Термін подання студентом роботи 04.06.2026

3. Вихідні дані до роботи ігровий рушій Unity 6 LTS, мова програмування C#, допоміжні технології та пакети — Universal Render Pipeline з постефектами, New Input System, Cinemachine, NavMesh, ScriptableObjects; збереження прогресу через PlayerPrefs, система контролю версій Git (GitHub), середовище розробки Visual Studio Code.

4. Перелік питань, які потрібно розробити \_\_\_\_\_

1) проаналізувати жанр survivors-like та наявні реалізації механік маніпуляції часом у відеоіграх;

2) проаналізувати ігри-аналоги та виявити незайняту нішу для розроблюваного продукту;

3) обґрунтувати вибір ігрового рушія, мови програмування та технологічного стеку;

- 4) спроектувати модульну архітектуру програмного забезпечення з відокремленням даних від логіки та реалізувати гру «DimeLoop Survivors» на базі ігрового рушія Unity 6, зокрема систему маніпуляції часом, систему ворогів, прогресію та метапрогрес;
- 5) протестувати працездатність підсистем та оптимізувати продуктивність гри;
- 6) скомпілювати гру для платформи Windows і підготувати презентацію з результатами аналізу та демонстрацією проєкту для захисту дипломної роботи.
5. Орієнтовний перелік ілюстративного матеріалу: діаграма прецедентів, схема архітектури програмної системи, діаграма класів, алгоритм роботи системи відмотування часу, скріншоти ігрового процесу та інтерфейсу користувача.
6. Дата видачі завдання 18.12.2025 р.

### КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів виконання дипломної роботи             | Термін виконання етапів роботи | Примітка |
|-------|-----------------------------------------------------|--------------------------------|----------|
| 1     | Затвердження теми роботи                            | 18.12.25                       | виконано |
| 2     | Вивчення та аналіз задачі                           | 13.04.26–19.04.26              | виконано |
| 3     | Розробка архітектури та загальної структури системи | 20.04.26–26.04.26              | виконано |
| 4     | Розробка окремих підсистем гри                      | 27.04.26–03.05.26              | виконано |
| 5     | Програмна реалізація системи                        | 04.05.26–10.05.26              | виконано |
| 6     | Оформлення пояснювальної записки                    | 11.05.26–13.05.26              | виконано |
| 7     | Захист програмного продукту                         | 14.05.26                       | виконано |
| 8     | Передзахист                                         | 04.06.26                       |          |
| 9     | Подання готової роботи на кафедру                   | 08.06.26                       |          |
| 10    | Захист                                              | 16.06.26                       |          |

Студент

( підпис )

**Артем ГЛИВЯК**

(прізвище та ініціали)

Керівник роботи

( підпис )

**Наталія АУШЕВА**

(прізвище та ініціали)

# АНОТАЦІЯ

Дипломна робота виконана на 55 сторінках, містить 21 ілюстрацій, 2 таблиці, 1 додаток та 21 джерел у переліку посилань.

Мета роботи — проектування та розробка повнофункціональної комп'ютерної гри жанру roguelite-survivors з унікальною механікою маніпуляції часом на базі ігрового рушія Unity 6.

Методи та засоби розробки: ігровий рушій Unity 6 LTS, мова програмування C#, Universal Render Pipeline з постефектами, New Input System, Cinemachine, NavMesh для навігації ворогів, патерни проектування Singleton, Observer, Object Pool та Strategy на базі ScriptableObjects, збереження прогресу через PlayerPrefs.

Результат — комп'ютерна гра DimeLoop Survivors для платформи Windows, що реалізує жанр roguelite-survivors з оригінальною системою маніпуляції часом, яка включає відмотування часу, зупинку часу та локальні часові зони. Гра містить повний ігровий цикл із системою прогресії, метапрогресом між сесіями, модулем навчання та двома ігровими світами. Скомпільовану версію гри протестовано для платформи Windows.

**Ключові слова:** ROGUELITE, SURVIVORS-LIKE, UNITY 6, C#, МАНІПУЛЯЦІЯ ЧАСОМ, REWIND, TIMESTOP, SCRIPTABLEOBJECTS, OBJECT POOL, NAVMESH, URP.

# ABSTRACT

The thesis is 55 pages long, contains 21 figures, 2 tables, 1 appendix, and 21 references.

The purpose of the work is the design and development of a fully functional roguelite-survivors computer game with a unique time manipulation mechanic built on the Unity 6 game engine.

Development methods and tools: Unity 6 LTS game engine, C# programming language, Universal Render Pipeline with post-processing effects, New Input System, Cinemachine, NavMesh for enemy navigation, Singleton, Observer, Object Pool and Strategy design patterns based on ScriptableObjects, PlayerPrefs for progress persistence.

The result is a computer game DimeLoop Survivors for the Windows platform, implementing the roguelite-survivors genre with an original time manipulation system that includes time rewind, time stop, and local time zones. The game features a complete game loop with a progression system, meta-progression between sessions, a shop, a tutorial, and two game worlds. The compiled version of the game has been tested for the Windows platform.

**Keywords:** ROGUELITE, SURVIVORS-LIKE, UNITY 6, C#, TIME MANIPULATION, REWIND, TIMESTOP, SCRIPTABLEOBJECTS, OBJECT POOL, NAVMESH, URP.

## ЗМІСТ

|                                                                         |    |
|-------------------------------------------------------------------------|----|
| ВСТУП.....                                                              | 8  |
| 1 ДОСЛІДЖЕННЯ ЖАНРУ SURVIVORS-LIKE ТА МЕХАНІК МАНІПУЛЯЦІЇ ЧАСОМ .....   | 10 |
| 1.1 Постановка задачі розробки програмної системи .....                 | 10 |
| 1.2 Огляд жанру survivors-like та його ключових механік.....            | 13 |
| 1.3 Аналіз ігор-аналогів та часових механік .....                       | 15 |
| 1.4 Наявні програмні рішення.....                                       | 18 |
| 2 ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ .....                        | 20 |
| 2.1 Вибір ігрового рушія Unity 6 .....                                  | 20 |
| 2.1.1 Переваги Unity 6 для розробки ігор жанру survivors-like .....     | 20 |
| 2.1.2 Розгляд альтернативних варіантів .....                            | 22 |
| 2.2 Вибір мови програмування C#.....                                    | 23 |
| 2.3 Допоміжні технології та пакети.....                                 | 24 |
| 2.3.1 Бібліотека Universal Render Pipeline.....                         | 24 |
| 2.3.2 Система обробки вводу New Input System та навігації NavMesh ..... | 25 |
| 2.3.3 Бібліотека Cinemachine та TextMeshPro .....                       | 26 |
| 2.4 Архітектурні патерни проєктування .....                             | 28 |
| 3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ .....                                    | 30 |
| 3.1 Архітектура програмного забезпечення .....                          | 30 |
| 3.1.1 Загальна структура та ключові компоненти.....                     | 30 |
| 3.1.2 Діаграма класів .....                                             | 33 |
| 3.2 Основний ігровий цикл.....                                          | 34 |
| 3.3 Система ворогів та навігації.....                                   | 37 |

|                                              |    |
|----------------------------------------------|----|
| 3.4 Система маніпуляції часом .....          | 40 |
| 3.5 Система прогресії та збереження .....    | 43 |
| 3.5.1 Система досвіду та покращень .....     | 44 |
| 3.5.2 Система збереження та магазину .....   | 46 |
| 3.5.3 Модуль інтерактивного навчання.....    | 47 |
| 4 ВЗАЄМОДІЯ КОРИСТУВАЧА З СИСТЕМОЮ .....     | 48 |
| 4.1 Системні вимоги та інсталяція.....       | 48 |
| 4.2 Опис інтерфейсу користувача.....         | 49 |
| 4.3 Сценарій роботи користувача .....        | 53 |
| 4.4 Тестування програмного забезпечення..... | 55 |
| ВИСНОВКИ.....                                | 58 |
| СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....             | 59 |
| ДОДАТОК А .....                              | 62 |

## ВСТУП

Жанр survivors-like з'явився як відносно простий експеримент, метою якого було з'ясувати, що відбудеться, якщо прибрати з action-гри необхідність прицілюватись і зосередити увагу гравця виключно на позиціонуванні та виборі покращень. Відповідь виявилась несподіваною, оскільки гра Vampire Survivors у 2022 році зібрала мільйони гравців саме завдяки цій ідеї. Відтоді жанр розрісся і міцно закріпився на ринку, залучаючи дедалі більше розробників, які вбачають у ньому простір для власних ідей.

Питання, яке стало відправною точкою цього проєкту, звучить просто: чи можна додати до жанру щось принципово нове, не зруйнувавши те, що в ньому працює? Маніпуляція часом як ігрова механіка існує давно на прикладах таких проєктів, як Braid, Prince of Persia та Quantum Break, проте жоден із них не поєднував її з динамікою survivors-like. Кожна з цих ігор будувала часові механіки як центральний елемент окремого жанру, тоді як у survivors-like час досі залишається просто фоном. Саме цей перетин і є темою даної роботи.

Актуальність теми зумовлена тим, що попри насиченість ринку іграми жанру survivors-like, інтеграція нелінійних часових механік у цей жанр залишається практично недослідженою нішею як у комерційній індустрії, так і в академічному середовищі. Більшість існуючих робіт розглядають або жанр survivors-like, або маніпуляцію часом окремо, уникаючи їх безпосереднього поєднання. Окрім того, сам жанр є відносно молодим, адже його масове поширення почалось лише у 2022 році, тож дослідницька база навколо нього ще формується.

Метою роботи є проєктування та розробка комп'ютерної гри DimeLoop Survivors жанру roguelite-survivors, де маніпуляція часом виступає не допоміжною функцією, а центральною тактичною механікою, реалізованою засобами Unity 6 та C#. Для досягнення цієї мети були поставлені такі завдання: аналіз існуючих аналогів та підходів до реалізації часових механік у відеоіграх, проєктування модульної архітектури програмного забезпечення, реалізація та тестування всіх ігрових підсистем, компіляція фінальної версії гри для платформи Windows.

Результатом роботи є готова гра для Windows, яка пройшла повний цикл розробки, починаючи від першого прототипу і закінчуючи скомпільованою версією гри. Підходи, застосовані під час розробки, не є суто теоретичними, оскільки кожне рішення перевірялось у дії і коригувалось на основі реального тестування.

Дипломна робота складається з чотирьох розділів. Перший розділ присвячений аналізу предметної області, огляду існуючих аналогів у жанрі survivors-like та реалізації маніпуляції часом у відеоіграх. Другий розділ містить опис технологічного стеку та обґрунтування прийнятих архітектурних рішень. Третій розділ, найбільший за обсягом, описує програмну реалізацію всіх ключових підсистем гри. Четвертий розділ розглядає практичні аспекти взаємодії користувача з готовим продуктом, включаючи системні вимоги та сценарії роботи.

# **1 ДОСЛІДЖЕННЯ ЖАНРУ SURVIVORS-LIKE ТА МЕХАНІК МАНІПУЛЯЦІЇ ЧАСОМ**

Перед початком розробки будь-якого програмного продукту необхідно чітко визначити його призначення, проаналізувати існуючі рішення у відповідній галузі та обґрунтувати доцільність обраного підходу. У цьому розділі розглядається предметна область розроблюваної гри, формулюється постановка задачі, проводиться огляд жанру survivors-like та існуючих аналогів, аналізуються підходи до реалізації маніпуляції часом у відеоіграх, а також обґрунтовується оригінальність розроблюваного продукту на тлі наявних програмних рішень.

## **1.1 Постановка задачі розробки програмної системи**

Об'єктом розробки є комп'ютерна гра DimeLoop Survivors жанру roguelite-survivors для платформи Windows. Предметом розробки є методи та засоби реалізації системи маніпуляції часом, що функціонує як центральна тактична механіка в рамках класичного ігрового циклу жанру.

Призначення системи полягає в наданні гравцеві ігрового досвіду, побудованого на принципах roguelite: коротких сесіях, накопичуваному метапрогресі між ними та варіативності проходжень за рахунок випадкового вибору покращень. Відмінністю від існуючих представників жанру є те, що час у грі перестає бути константою, оскільки гравець отримує над ним обмежений контроль, що формує принципово інший характер прийняття тактичних рішень під час бою.

Система орієнтована на три категорії потенційних користувачів. Першу категорію становлять гравці, знайомі з жанром survivors-like, які шукають нового досвіду в межах звичної формули. До другої групи належать гравці, які цінують механіки маніпуляції часом за прикладами Braid або Prince of Persia і хочуть

побачити їх у новому контексті. Третьою аудиторією є студенти та викладачі профільних дисциплін, для яких гра слугує технічним прикладом реалізації нелінійних часових механік засобами Unity.

Для досягнення поставленої мети були визначені такі завдання:

- провести аналіз жанру survivors-like та існуючих аналогів, виявити незайняту нішу для розроблюваного продукту;
- дослідити підходи до реалізації маніпуляції часом у відеоіграх та обрати технічно обґрунтоване рішення для Unity;
- спроектувати модульну архітектуру програмного забезпечення з відокремленням даних від логіки;
- реалізувати ключові ігрові підсистеми: рух гравця, систему ворогів з NavMesh-навігацією, зброю з Object Pool, систему маніпуляції часом, прогресію та метапрогрес;
- розробити модуль навчання, магазин, два ігрові світи та повний цикл ігрового UI;
- зібрати скомпільовану гру для платформи Windows і провести тестування.

Вхідна інформація поділяється на два типи. Проектна задається розробником у вигляді ScriptableObjects безпосередньо в редакторі Unity: характеристики ворогів, конфігурації рівнів, параметри покращень та предметів магазину. Користувачка надходить у режимі виконання: натискання клавіш керування, позиція курсора миші для прицілювання, вибір покращень при підвищенні рівня та предметів у магазині. Перелік керуючих клавіш наведено у таблиці 1.1.

Таблиця 1.1 — Керуючі клавіші гравця

| Клавіша | Дія                              |
|---------|----------------------------------|
| WASD    | Переміщення персонажа            |
| Миша    | Прицілювання                     |
| ЛКМ     | Ручний постріл                   |
| Space   | Активація уміння (Dash / Sprint) |
| R       | Відмотування часу (Rewind)       |
| E       | Зупинка часу (TimeStop)          |

Кінець таблиці 1.1

|        |                             |
|--------|-----------------------------|
| V      | Перемикання режиму стрільби |
| Escape | Пауза                       |
| F      | Пропуск кроку навчання      |

Вихідна інформація формується підсистемою рендерингу Unity: тривимірне зображення ігрової сцени з моделями гравця, ворогів і снарядів, двовимірний інтерфейс з індикаторами стану, а також збережені дані прогресу в PlayerPrefs операційної системи Windows.

Функціональні можливості системи описуються діаграмою прецедентів, яка відображає взаємодію гравця з основними підсистемами гри (рисунок 1.1).

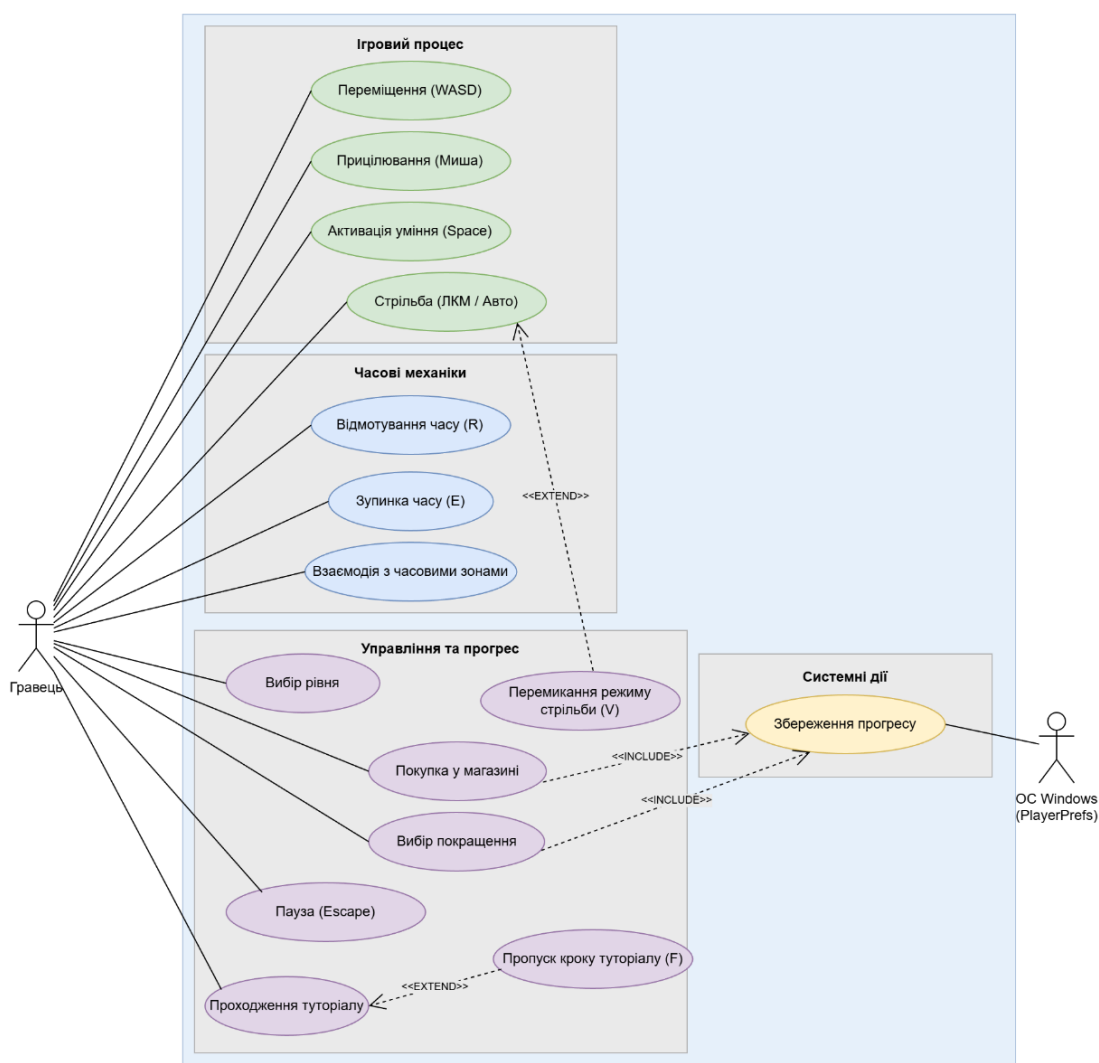


Рисунок 1.1 — Діаграма прецедентів взаємодії гравця з ігровою системою

Наведена діаграма структурована на чотири основні групи прецедентів: ігровий процес, часові механіки, управління та прогрес, а також системні дії. Часові механіки виділено в окрему групу, оскільки вони є центральною унікальною особливістю системи та використовують власний ресурс Time Energy, що обмежує їх використання в межах ігрової сесії.

## **1.2 Огляд жанру survivors-like та його ключових механік**

Survivors-like є піджанром action-ігор, у якому гравець керує персонажем з автоматичною атакою та протистоїть наростаючим хвилям ворогів. Головна відмінність від класичних shoot'em up ігор полягає у зміщенні фокусу: замість прицілювання застосовується позиціонування, а замість одного рівня складності використовується нескінченне масштабування через кількість і щільність ворогів. Жанр популяризувала гра Vampire Survivors у 2022 році, хоча його витоки простежуються до мобільної гри Magic Survival 2018 року [1].

Ключовими механіками жанру є кілька взаємопов'язаних елементів. Автоматична атака звільняє гравця від необхідності прицілюватись і переносить усю увагу на переміщення по арені та уникнення ворогів. Система досвіду і покращень функціонує за принципом roguelite: після знищення ворогів гравець отримує одиниці досвіду, накопичення яких призводить до підвищення рівня та вибору одного з кількох випадкових покращень характеристик. Метапрогрес між сесіями дозволяє витратити зібрані ресурси на постійні покращення, що робить кожну наступну спробу трохи простішою. Сесія обмежена часом або умовою перемоги, за якої гравець або виживає до кінця, або гине і починає заново.

Складність у жанрі зростає не через підвищення небезпеки окремого ворога, а через збільшення їхньої кількості та щільності на екрані. Саме тому позиціонування стає головною навичкою гравця, адже від нього вимагається постійно рухатись, уникати оточення та шукати вигідні маршрути через натовп. Ця

особливість відрізняє survivors-like від більшості інших action-жанрів, де основним викликом є реакція та точність.

Загальний ігровий цикл жанру survivors-like має чітку структуру, яка повторюється протягом кожної сесії. Гравець починає раунд зі слабким персонажем, поступово збирає досвід з вбитих ворогів, підвищує рівень та обирає покращення, після чого цикл повторюється з наростаючою складністю до завершення сесії перемогою або поразкою. Схему ігрового циклу наведено на рисунку 1.2.

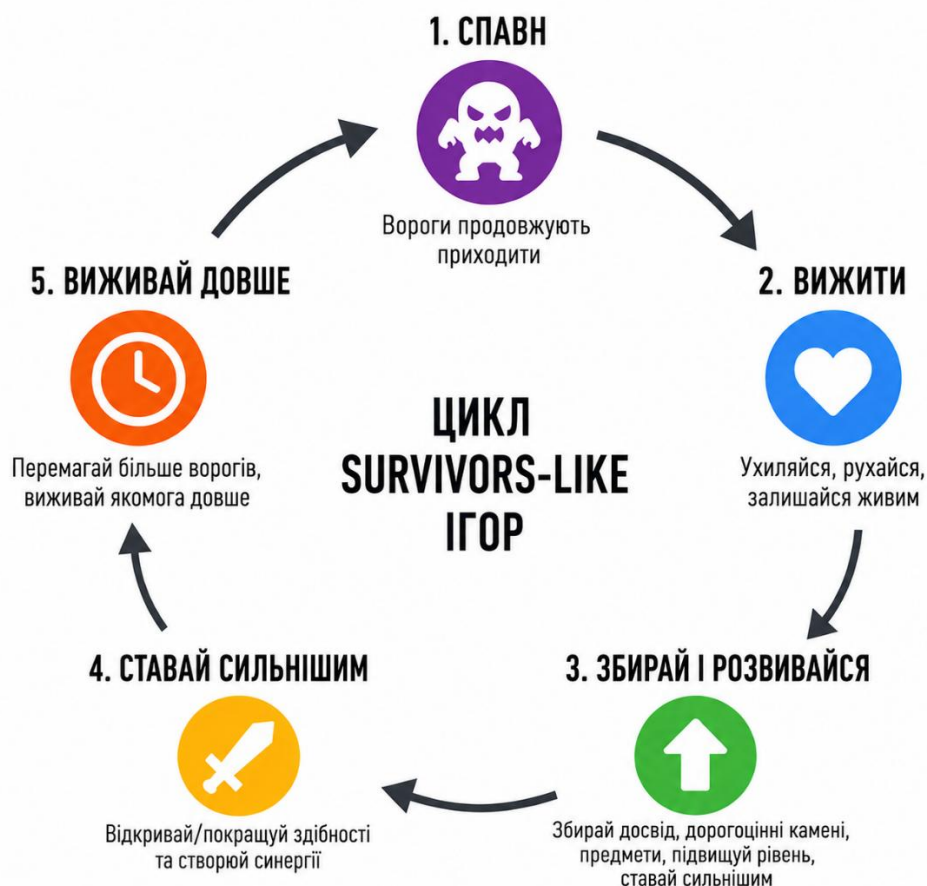


Рисунок 1.2 — Схема ігрового циклу жанру survivors-like

Метапрогрес між сесіями є важливою складовою жанру, що утримує гравця у довгостроковій перспективі. Зібрані за сесію монети або інші ресурси зберігаються між запусками і можуть бути витрачені на постійні покращення або нові предмети у магазині. Таким чином, навіть невдала сесія приносить певний прогрес, що знижує відчуття марності після поразки.

### 1.3 Аналіз ігор-аналогів та часових механік

Для формування концепції DimeLoop Survivors було проведено аналіз ключових представників двох напрямків: ігор з реалізованою маніпуляцією часом та ігор жанру survivors-like. Розглянуті приклади дозволили визначити геймдизайнерські підходи та технічні рішення, що лягли в основу розроблюваного продукту.

Braid (Number None, 2008) є одним із найбільш досліджених прикладів нелінійної маніпуляції часом у відеоіграх. Гра являє собою платформер-головоломку, де кожен світ побудований навколо унікальної варіації часової механіки [2]. На відміну від більшості ігор, де відмотування часу є лише засобом виправлення помилок гравця, у Braid воно є основним інструментом вирішення головоломок. Центральною є концепція time-immune об'єктів: певні елементи рівня не підпадають під дію відмотування і продовжують існувати у поточному часовому стані незалежно від дій гравця. В окремих світах плин часу прив'язаний до позиції гравця: рух вправо прискорює час, рух вліво відмотує його назад (рисунок 1.3).



Рисунок 1.3 —Механіка реверсу часу в грі Braid

Концепція time-immune об'єктів була використана у DimeLoop Survivors: вороги без компонента Rewindable залишаються замороженими під час

відмотування, тоді як вороги з цим компонентом повертаються у попередній стан. Такий підхід дозволяє гнучко визначати, які саме об'єкти беруть участь у часових маніпуляціях.

Prince of Persia: The Sands of Time (Ubisoft, 2003) стала першою масовою грою, яка ввела відмотування часу як центральну ігрову механіку для широкої аудиторії [3]. Ключовою особливістю є система Піску Часу, що являє собою обмежений ресурс, який витрачається при використанні часових здібностей і поповнюється у процесі бою з ворогами. Гравець може відмотати час назад, уповільнити або повністю зупинити його, однак кожна з цих дій потребує певної кількості ресурсу. Дане рішення усуває одну з головних проблем часових механік: якщо відмотування є безмежним, воно знецінює будь-який ризик для гравця (рисунок 1.4).



Рисунок 1.4 — Ігровий процес Prince of Persia: The Sands of Time

Зазначена концепція є прямим прототипом системи Time Energy у DimeLoop Survivors. Обидві здібності Rewind та TimeStop споживають спільний ресурс, що відновлюється з часом. Обмежений ресурс змушує гравця свідомо визначати

момент застосування тієї чи іншої здібності, що формує тактичну глибину в умовах динамічного бою.

Vampire Survivors (Poncle, 2022) є основоположником жанру survivors-like у його сучасному розумінні. Гра вийшла у ранньому доступі у грудні 2021 року і за короткий час сформувала новий піджанр. Ігровий процес побудований на основі такої формули: гравець керує персонажем на арені, автоматично атакує ворогів та намагається вижити протягом відведеного часу. Основою прогресії є система покращень при підвищенні рівня та метапрогрес між сесіями [4] (рисунок 1.5).



Рисунок 1.5 — Ігровий процес Vampire Survivors

Система синергій між предметами у Vampire Survivors стала еталоном жанру і була взята за основу при проєктуванні системи прогресії DimeLoop Survivors. Час у грі є виключно глобальною константою без будь-яких можливостей впливу з боку гравця, що і визначило головну ідею розроблюваного продукту: інтеграція механік маніпуляції часом у перевірену жанрову формулу.

## 1.4 Наявні програмні рішення

Реалізація маніпуляції часом у відеоіграх є нетривіальним технічним завданням, оскільки більшість ігрових рушіїв, включаючи Unity, не мають вбудованої підтримки подібних механік. Існує кілька принципово різних підходів до вирішення цієї задачі, кожен з яких має свої переваги та обмеження.

Перший підхід передбачає використання детермінованого ігрового рушія, що забезпечує повне відтворення стану гри з певної точки шляхом повторного виконання всіх дій гравця у правильній послідовності [5]. Теоретично це найточніший метод, оскільки не потребує зберігання проміжних станів об'єктів. Однак на практиці його реалізація у Unity є вкрай складною через недетермінізм фізичного рушія PhysX: однакові вхідні дані можуть давати різний результат залежно від частоти кадрів та порядку виконання операцій. З огляду на це, зазначений підхід не був обраний для реалізації у даній роботі.

Другий підхід базується на щокadroвому записі стану об'єктів у буфер з подальшим відтворенням у зворотному порядку. Кожен кадр система зберігає знімок стану всіх об'єктів, що беруть участь у відмотуванні: позицію, обертання, стан здоров'я та інші параметри. При активації механіки Rewind система відтворює збережені знімки у зворотному порядку. Головним обмеженням є споживання оперативної пам'яті: зберігання стану десятків об'єктів протягом кількох секунд потребує ретельного управління буфером. У даній роботі цю проблему вирішено через кільцевий буфер з максимальною глибиною 10 секунд, де нові знімки додаються на початок, а старі автоматично видаляються при досягненні ліміту.

Слід зазначити, що вибір глибини буфера у 10 секунд є компромісом між тактичною цінністю механіки та навантаженням на пам'ять: менша глибина обмежує корисність відмотування, тоді як більша суттєво збільшує споживання ресурсів при великій кількості активних об'єктів на сцені.

Третій підхід передбачає використання локального часового масштабу, що дозволяє кожному об'єкту мати власний коефіцієнт плину часу, незалежний від глобального `Time.timeScale`. Стандартний `Time.timeScale` в Unity діє глобально і

зупиняє або сповільнює всю сцену одночасно. Натомість компонент `TimeScaleObject` надає кожному об'єкту власну змінну `localTimeScale`, яка множиться на `Time.deltaTime` при розрахунку переміщення. Це дозволяє одночасно підтримувати об'єкти у різних часових станах: одні заморожені, інші рухаються з нормальною або зміненою швидкістю.

У даній роботі застосовано комбінацію другого та третього підходів. Компонент `Rewindable` реалізує щок кадровий запис у кільцевий буфер, а `TimeScaleObject` забезпечує локальний часовий масштаб для кожного об'єкта сцени. Така архітектура дозволила реалізувати три незалежні часові механіки в рамках однієї системи без конфліктів між ними [6].

## **2 ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ**

У процесі розробки програмного продукту вибір інструментів та технологій є одним із ключових рішень, що визначають якість кінцевого результату та ефективність самого процесу розробки. У цьому розділі обґрунтовано вибір ігрового рушія, мови програмування та допоміжних технологій, що були використані при створенні програмного забезпечення, а також розглянуто альтернативні варіанти та причини їх відхилення.

### **2.1 Вибір ігрового рушія Unity 6**

Ігровий рушій є фундаментальною платформою для розробки будь-якої сучасної комп'ютерної гри. Його вибір впливає на всі аспекти розробки: від графічних можливостей та продуктивності до доступного інструментарію та зручності роботи з кодом [7]. Як основну платформу обрано Unity 6, що є реліз-версією з довгостроковою підтримкою, яка поєднує сучасні технології рендерингу з розвиненим набором інструментів для створення ігор різних жанрів.

#### **2.1.1 Переваги Unity 6 для розробки ігор жанру survivors-like**

Вибір Unity 6 обумовлений кількома ключовими факторами, що безпосередньо відповідають технічним вимогам розроблюваного продукту.

Першим фактором є компонентна архітектура через систему GameObject-Component. Кожен ігровий об'єкт може мати довільний набір прикріплених компонентів поведінки, що дозволяє гнучко комбінувати функціональність без жорсткої ієрархії успадкування. Така архітектура добре відображає структуру action-ігор, де різні типи об'єктів мають спільні та унікальні характеристики.

Другим фактором є комплексна вбудована підтримка суміжних систем: фізики через рушій PhysX, навігації через NavMesh, рендерингу через Universal Render Pipeline з постефектами, системи частинок та звуку. Сукупність цих інструментів покриває більшість технічних потреб ігрових проєктів без необхідності використання зовнішніх рішень.

Третім фактором є підтримка ScriptableObjects, що являють собою серіалізовані класи, які дозволяють зберігати конфігураційні дані незалежно від екземплярів скриптів. Цей механізм забезпечує відокремлення даних від логіки та дозволяє виносити балансні значення у редактор без необхідності змінювати код [8]. Схему передачі даних через ScriptableObjects наведено на рисунку 2.1.

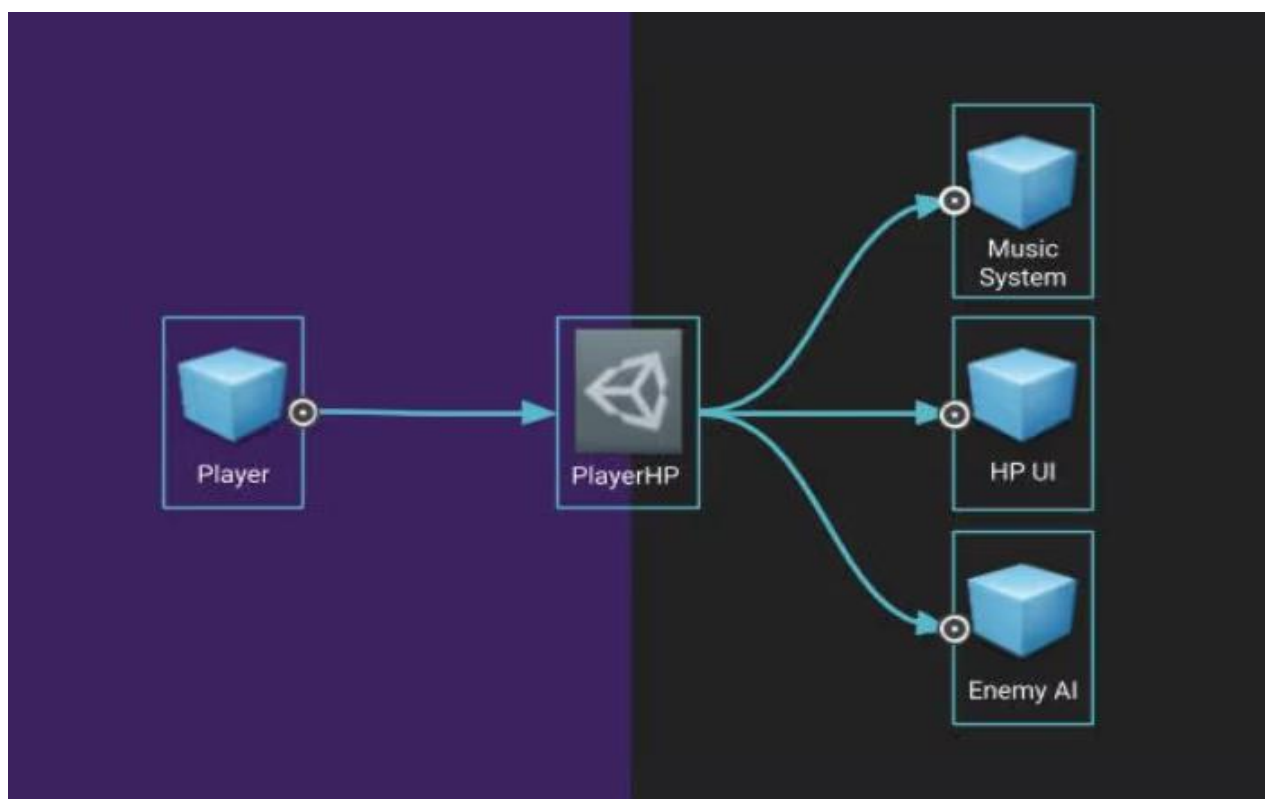


Рисунок 2.1 — Схеми передачі даних через ScriptableObjects

Четвертим фактором є продуктивність при роботі з великою кількістю однотипних об'єктів. Unity 6 представив GPU Resident Drawer, що значно покращує продуктивність при рендерингу великої кількості однотипних об'єктів на сцені. Ця

можливість є критично важливою для жанрів з великою кількістю активних об'єктів одночасно.

Unity також надає розвинену систему управління сценами та ресурсами. Асинхронне завантаження сцен через `SceneManager.LoadSceneAsync` дозволяє уникати затримок при переходах між ігровими екранами. Система `AssetBundle` забезпечує ефективне управління ресурсами з можливістю динамічного завантаження та вивантаження контенту. Вбудований інструментарій `Unity Profiler` дозволяє аналізувати продуктивність у реальному часі, відстежуючи навантаження на процесор, відеокарту та оперативну пам'ять з деталізацією до окремих методів та компонентів.

### **2.1.2 Розгляд альтернативних варіантів**

При виборі ігрового рушія були розглянуті два основні альтернативні варіанти: Unreal Engine 5 та Godot Engine.

Unreal Engine 5 є потужним рушієм з передовими технологіями рендерингу, зокрема Nanite та Lumen. Рушій орієнтований переважно на мову C++, що потребує значно більшого часу на розробку порівняно з C#. Поріг входження є суттєво вищим, що робить його менш доцільним вибором для індивідуального розробника в умовах обмеженого часу. Крім того, для проєктів з акцентом на ігрову логіку, а не на фотореалістичну графіку, можливості Unreal Engine 5 є надлишковими [9].

Godot Engine є відкритим рушієм з активною спільнотою та власною мовою скриптування GDScript. Рушій є цілком придатним для розробки 2D ігор, однак його можливості у сфері 3D рендерингу та постефектів є обмеженішими порівняно з Unity. Зокрема, відсутність повноцінного аналога Universal Render Pipeline обмежує можливості реалізації складних візуальних ефектів [10].

Таким чином, Unity 6 є оптимальним вибором з огляду на поєднання зрілого інструментарію, підтримки C#, розвиненої екосистеми пакетів та достатньої продуктивності.

## 2.2 Вибір мови програмування C#

Мова програмування є визначальним фактором ефективності та гнучкості реалізації задуманих алгоритмів. Для розробки на платформі Unity обрано мову C# як стандартний інструмент реалізації ігрової логіки [11].

C# є об'єктно-орієнтованою мовою з суворим контролем типів, автоматичним збиранням сміття та перевіркою меж масивів. Серед ключових характеристик мови варто виділити:

- суворий контроль типів дозволяє виявляти помилки на етапі компіляції, що є особливо цінним при складних взаємодіях між підсистемами;
- автоматичне збирання сміття звільняє розробника від ручного керування пам'яттю, однак потребує свідомого управління частотою створення об'єктів;
- підтримка інтерфейсів дозволяє описувати контракти між компонентами без жорсткої залежності від конкретних реалізацій;
- механізм подій та делегатів забезпечує слабку зв'язку між підсистемами через патерн Observer;
- співпрограми дозволяють реалізовувати послідовності з затримками без блокування основного потоку;
- узагальнені колекції List, Dictionary та Queue надають ефективні структури даних для різноманітних задач.

Порівняно з альтернативними мовами, C# у контексті Unity є очевидним вибором. Використання C++ потребувало б значно більшого часу на розробку та не надало б суттєвих переваг у продуктивності для ігор подібного масштабу. GDScript є специфічним для Godot Engine і несумісним з Unity.

C# функціонує на платформі .NET, що надає розробнику доступ до стандартної бібліотеки класів з широким набором готових рішень для роботи з колекціями, рядками, файловою системою та асинхронними операціями. Unity використовує власну реалізацію середовища виконання на базі Mono та IL2CPP, що забезпечує сумісність C# коду з різними цільовими платформами. Компілятор

IL2CPP перетворює код на C++ перед компіляцією у нативний код платформи, що підвищує продуктивність виконання порівняно з інтерпретованим підходом.

## 2.3 Допоміжні технології та пакети

Окрім ядра Unity та мови C#, розробка програмного забезпечення спиралась на низку офіційних пакетів та технологій, кожен з яких вирішує конкретну технічну задачу. Використання спеціалізованих пакетів замість власної реалізації є обґрунтованим підходом з точки зору якості та надійності коду: офіційні пакети Unity пройшли ретельне тестування, мають документацію та підтримуються командою розробників рушія. Це дозволяє зосередити зусилля на реалізації унікальної ігрової логіки, а не на вирішенні типових технічних задач, таких як навігація агентів, обробка вводу або керування камерою.

### 2.3.1 Бібліотека Universal Render Pipeline

Universal Render Pipeline є оптимізованим конвеєром рендерингу від Unity, що забезпечує підтримку постефектів та гнучке налаштування візуального стилю [12]. Серед ключових можливостей URP виділяються:

- підтримка постефектів через Volume систему: хроматична аберація, десатурація, віньєтування, bloom та інші;
- програмний доступ до параметрів постефектів під час виконання через API;
- використання `Time.unscaledDeltaTime` для постефектів, що забезпечує коректну роботу незалежно від стану глобального часу сцени;
- оптимізований рендеринг для широкого спектру платформ від мобільних до настільних.

Завдяки модульній архітектурі URP розробник може налаштовувати конвеєр рендерингу під конкретні вимоги проєкту, додаючи або вимикаючи окремі проходи рендерингу. Це забезпечує баланс між візуальною якістю та продуктивністю залежно від цільової платформи.

### **2.3.2 Система обробки вводу New Input System та навігації NavMesh**

New Input System є декларативною системою обробки вводу Unity, що підтримує клавіатуру, мишу та геймпади [13]. Система була введена як заміна застарілій системі вводу з метою усунення її архітектурних обмежень та надання більшої гнучкості розробникам. New Input System надає такі переваги:

- визначення дій незалежно від конкретних пристроїв вводу;
- підтримка одночасного використання кількох пристроїв вводу;
- декларативне налаштування прив'язок клавіш через редактор;
- подієва модель обробки вводу замість постійного опитування стану;
- підтримка складних жестів та комбінацій клавіш через систему взаємодій.

Декларативний підхід New Input System дозволяє відокремити логіку обробки вводу від конкретних пристроїв, що спрощує підтримку різних конфігурацій керування. Подієва модель обробки вводу є більш ефективною порівняно з постійним опитуванням стану, оскільки код виконується лише при фактичній зміні стану пристрою вводу.

NavMesh є вбудованою системою навігації Unity, що забезпечує пошук шляху для агентів на попередньо запеченій навігаційній сітці [14]. Система широко застосовується у іграх з великою кількістю автономних агентів, де кожен з них повинен самостійно визначати маршрут до цілі в обхід перешкод. NavMesh надає такі можливості:

- автоматичне запікання навігаційної сітки на основі геометрії сцени;
- підтримка різних типів агентів з індивідуальними параметрами радіусу, висоти та швидкості;
- динамічне визначення оптимального маршруту через NavMeshAgent;

- підтримка шарів навігації для розмежування зон доступності для різних типів агентів;

- можливість виключення окремих об'єктів або шарів з навігаційної сітки.

Компоненти системи навігації NavMesh наведено на рисунку 2.2.

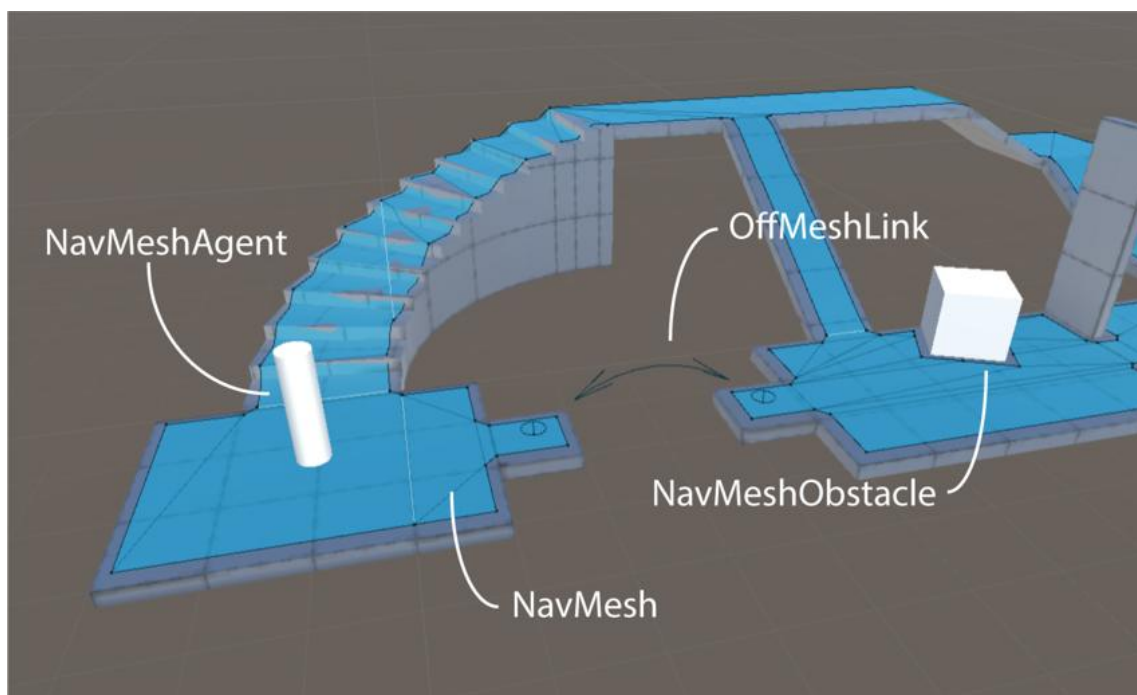


Рисунок 2.2 — Компоненти системи навігації NavMesh в Unity

NavMesh є ефективним рішенням для реалізації навігації в іграх з великою кількістю агентів, оскільки обчислення шляху виконується на попередньо запеченій сітці, що значно знижує навантаження на процесор під час виконання. Алгоритм пошуку шляху базується на методі A\*, що гарантує знаходження оптимального маршруту за прийнятний час. Компонент NavMeshAgent автоматично оновлює маршрут при зміні позиції цілі, що забезпечує коректну поведінку агентів у динамічному середовищі.

### 2.3.3 Бібліотека Cinemachine та TextMeshPro

Cinemachine є системою керування камерою Unity, що забезпечує плавне слідування камери за об'єктом з налаштовуваними параметрами [15]. Система

надає розробникам потужний інструментарій для реалізації складної поведінки камери без необхідності написання власного коду керування. Серед можливостей системи виділяються:

- плавне слідування за цільовим об'єктом з налаштовуваними параметрами демпфування;
- підтримка різних типів камер: від третьої особи, від першої особи, орбітальна;
- система пріоритетів для автоматичного переключення між камерами;
- інтеграція з системою анімацій Unity для синхронізації руху камери;
- підтримка обмежень руху камери для запобігання виходу за межі ігрового простору.

Cinemachine усуває необхідність ручної реалізації логіки руху камери, надаючи готові рішення для типових ігрових сценаріїв. Параметри демпфування дозволяють досягти природного відчуття руху без різких стрибків при швидкому переміщенні об'єкта стеження. Система є неблокуючою та повністю інтегрується з іншими компонентами Unity через стандартний механізм подій.

TextMeshPro є системою рендерингу тексту на основі SDF-технології, що забезпечує чіткість шрифтів при будь-якому масштабуванні. Бібліотека є офіційним стандартом рендерингу тексту у Unity, що прийшов на заміну застарілому компоненту Text. Ключові характеристики бібліотеки:

- використання Signed Distance Field технології для збереження чіткості тексту незалежно від розміру;
- підтримка розширеного форматування тексту через теги;
- можливість застосування ефектів до тексту: тінь, обведення, градієнт;
- висока продуктивність порівняно зі стандартним компонентом Text;
- підтримка локалізації та Unicode для відображення тексту різними мовами.

TextMeshPro є стандартом рендерингу тексту у сучасних проєктах на Unity завдяки суттєвим перевагам у якості відображення та функціональності. SDF-технологія гарантує читабельність тексту незалежно від роздільної здатності

екрана та масштабу елементів інтерфейсу, що є особливо важливим для забезпечення якісного відображення на різних пристроях та роздільних здатностях.

## 2.4 Архітектурні патерни проєктування

При розробці програмного забезпечення вибір архітектурних патернів є не менш важливим рішенням, ніж вибір технологічного стеку. Патерни проєктування є усталеними рішеннями типових задач програмної інженерії, що дозволяють створювати гнучкі, розширювані та підтримувані системи [16].

Патерн Singleton забезпечує існування єдиного екземпляра класу з глобальною точкою доступу. Застосовується для менеджерів глобального стану, де необхідно гарантувати єдину точку керування певною підсистемою. Перевагою патерну є простота доступу до глобальних сервісів без необхідності передавати посилання через ієрархію об'єктів. Водночас надмірне використання Singleton може призвести до прихованих залежностей між компонентами, тому його застосування має бути обґрунтованим.

Патерн Observer реалізується через механізм подій та делегатів. Забезпечує слабку зв'язку між підсистемами через публікацію подій без знання про підписників. Клас-видавець публікує подію при зміні свого стану, а підписники реагують на неї незалежно один від одного. Такий підхід дозволяє додавати нових підписників без модифікації класу-видавця, що відповідає принципу відкритості до розширення та закритості до модифікації.

Патерн Object Pool вирішує проблему частого створення та знищення об'єктів, що призводить до пауз збирача сміття. Суть патерну полягає у попередньому створенні певної кількості об'єктів та їх повторному використанні замість знищення. Менеджер пулу керує колекціями неактивних об'єктів, надаючи їх на запит та повертаючи після використання. Такий підхід є особливо ефективним для об'єктів, що часто створюються та знищуються протягом роботи програми.

Патерн Strategy дозволяє визначати сімейство алгоритмів, інкапсулювати кожен з них та забезпечувати їх взаємозамінність без зміни коду клієнта. Патерн

реалізується через визначення спільного інтерфейсу для всіх варіантів алгоритму та вибір конкретної реалізації під час виконання. Це дозволяє легко розширювати систему новими варіантами поведінки без модифікації існуючого коду, що є важливою перевагою при розробці систем з варіативною поведінкою об'єктів.

Вибір конкретного патерну визначається характером задачі та вимогами до архітектури системи. Singleton доцільно застосовувати лише для справді глобальних сервісів. Observer є оптимальним вибором для систем з динамічною кількістю підписників. Object Pool найбільш ефективний у системах реального часу з інтенсивним створенням та знищенням об'єктів. Strategy забезпечує найбільшу гнучкість у системах з варіативною поведінкою об'єктів. Схему застосованих архітектурних патернів наведено на рисунку 2.3.



Рисунок 2.3 — Архітектурні патерни проєктування

Застосування зазначених патернів у сукупності дозволило створити модульну архітектуру, де кожен компонент має чітко визначену відповідальність та мінімальну залежність від інших підсистем.

## **3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ**

У попередніх розділах було проаналізовано предметну область, обґрунтовано вибір засобів розробки та архітектурних патернів. Даний розділ присвячений детальному опису програмної реалізації ключових підсистем гри DimeLoop Survivors. Розглядаються архітектура програмного забезпечення, основний ігровий цикл, система ворогів з навігацією, унікальна система маніпуляції часом та система прогресії гравця. Особлива увага приділяється технічним рішенням, що забезпечують коректну взаємодію між підсистемами та стабільну роботу гри в умовах великої кількості активних об'єктів на сцені.

### **3.1 Архітектура програмного забезпечення**

Ефективна архітектура є основою для створення масштабованої та підтримуваної програмної системи. Розроблена система побудована за принципом модульності з відокремленням даних від логіки, де кожен компонент відповідає за конкретну частину функціоналу та взаємодіє з іншими через чітко визначені інтерфейси.

#### **3.1.1 Загальна структура та ключові компоненти**

Програмна система складається з набору взаємопов'язаних класів, реалізованих на C# в рамках ігрового рушія Unity 6. Усі менеджери реалізовані за патерном Singleton і доступні з будь-якої підсистеми через статичну властивість Instance [17]. Слабка зв'язка між компонентами забезпечується через C# events та UnityEvent, що дозволяє додавати нових підписників без модифікації класів-видавців.

Сцени гри організовані відповідно до основного потоку: MainMenu з індексом 0, Tutorial з індексом 1, World1 з індексом 2 та World2 з індексом 3. Менеджери AudioManager та SaveManager створюються у сцені MainMenu з прапором DontDestroyOnLoad, що забезпечує їх збереження між сценами протягом усього часу роботи програми.

Ієрархія об'єктів ігрової сцени відображає модульну структуру системи, де кожен менеджер існує як окремий ігровий об'єкт з відповідним компонентом. Усі менеджери згруповані у батьківський об'єкт Managers, що забезпечує зручну навігацію в ієрархії під час розробки та налагодження. Ієрархію об'єктів ігрової сцени наведено на рисунку 3.1.

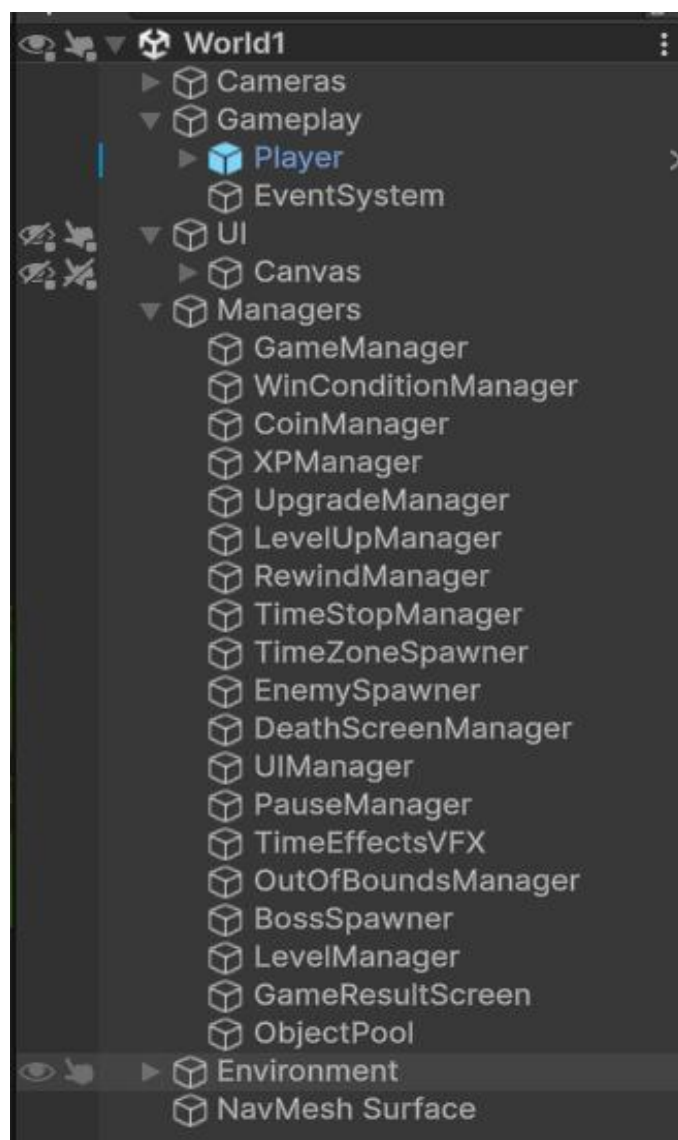


Рисунок 3.1 — Ієрархія об'єктів ігрової сцени World1 в Unity

Конфігураційні дані повністю відокремлені від логіки через ScriptableObjects. Для кожної підсистеми створено окремий тип ScriptableObject: EnemyData для параметрів ворогів, LevelConfig для налаштувань рівнів, UpgradeData для покращень, SkillData для умінь та ShopItemData для предметів магазину. Такий підхід дозволяє змінювати балансні значення безпосередньо в редакторі Unity без необхідності вносити зміни до коду. Загальну архітектуру системи наведено на рисунку 3.2.

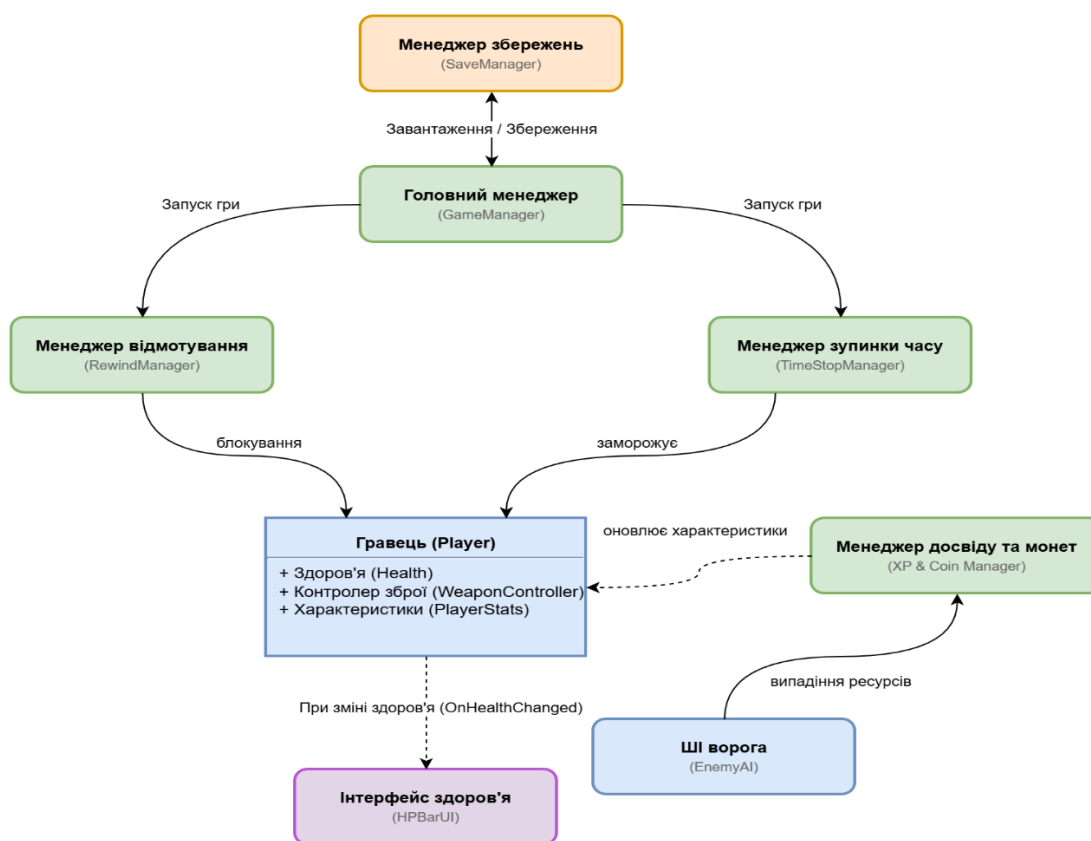


Рисунок 3.2 — Діаграма архітектури програмного забезпечення

Наведена діаграма відображає взаємодію між ключовими менеджерами системи та ігровими об'єктами. GameManager є центральним координатором сесії, що ініціалізує решту менеджерів та керує загальним станом гри. RewindManager та TimeStopManager реалізують часові механіки та використовують спільний ресурс Time Energy. Компонент Health публікує події OnHealthChanged та OnDeath, до яких підписуються HPBarUI, WinConditionManager та DeathScreenManager незалежно один від одного.



патерну Strategy. Компонент Rewindable зберігає кільцевий буфер знімків стану об'єкта та взаємодіє з RewindManager для синхронізації процесу відмотування. RewindManager керує усіма екземплярами Rewindable на сцені через залежність типу 1 до 0..\*, що відображає можливість одночасного відмотування довільної кількості об'єктів. SaveManager та XPManager є незалежними менеджерами, що підключаються до GameManager через асоціацію та забезпечують збереження прогресу і систему досвіду відповідно.

## 3.2 Основний ігровий цикл

Ігровий цикл є центральним механізмом, що координує роботу всіх підсистем гри протягом однієї сесії. Його реалізація побудована на послідовній ініціалізації компонентів та безперервному оновленні стану гри кожен кадр.

Цикл починається у класі GameManager, метод Start якого послідовно викликає два методи: ApplyShopSelections та StartGame. Метод ApplyShopSelections зчитує з PlayerPrefs вибрану гравцем зброю та вміння і передає їх відповідним компонентам гравця через WeaponController та PlayerStats. Метод StartGame ініціалізує решту менеджерів сесії, скидає лічильники досвіду та монет, активує WinConditionManager з параметрами поточного рівня через LevelConfig ScriptableObject. Якщо поточна сцена є навчальною, GameManager передає керування TutorialManager, який тимчасово блокує часові здібності до моменту їх покрокового вивчення гравцем.

Клас PlayerMovement реалізує переміщення персонажа на основі вводу через New Input System з фізичним переміщенням через Rigidbody. Напрямок руху визначається з вісей WASD та нормалізується для забезпечення рівномірної швидкості при діагональному русі. Прицілювання реалізовано через Raycast від позиції камери до шару groundLayer, що дозволяє визначити точку на площині арени, на яку вказує курсор миші. Метод Raycast виконується кожен кадр, а отримана точка перетину використовується для обчислення вектору від позиції

гравця до точки прицілювання, після чого персонаж обертається у цьому напрямку через `Quaternion.LookRotation`. Залежно від обраного `SkillData` активується одна з двох здібностей пересування. `Dash` реалізовано через застосування миттєвого імпульсу до `Rigidbody` у напрямку поточного руху через метод `AddForce` з режимом `ForceMode.Impulse`, після чого запускається таймер часу відновлення. `Sprint` реалізовано через тимчасове множення поточної швидкості переміщення на коефіцієнт `sprintMultiplier` протягом заданої тривалості через корутину.

Клас `WeaponController` реалізує стрільбу для всіх озброєнь гравця одночасно. Три типи зброї описуються через `WeaponData ScriptableObject`, який визначає частоту пострілів, тип кулі, базову шкоду та швидкість снаряда. Підтримуються два режими стрільби з перемиканням клавішею `V`: в автоматичному режимі `WeaponController` стріляє у напрямку курсора з фіксованим інтервалом, у ручному режимі постріл відбувається лише при натисканні лівої кнопки миші. Кулі реалізовані через патерн `Object Pool`: метод `OnEnable` ініціалізує стан кулі при отриманні з пулу, метод `OnDisable` повертає її до пулу при деактивації. Такий підхід усуває необхідність частого виклику `Instantiate` та `Destroy`, що запобігає паузам збирача сміття під час інтенсивної стрільби [18]. Фінальне значення шкоди розраховується як добуток базової шкоди зброї та множника `damageMultiplier` з `PlayerStats`, що дозволяє покращенням впливати на всі зброї одночасно.

Компонент `Health` є спільним для гравця та ворогів і реалізує уніфікований механізм отримання шкоди та смерті. При отриманні шкоди метод `TakeDamage` зменшує поточне здоров'я та публікує подію `OnHealthChanged`, до якої підписані `HPBarUI` та інші компоненти. При досягненні нуля викликається метод `Die`, який публікує подію `OnDeath` та ініціює відповідну реакцію підписників. У навчальному рівні здоров'я гравця автоматично відновлюється при падінні нижче мінімального порогу `minPlayerHP`, що унеможливорює загибель під час навчання.

Система `AudioManager` забезпечує звуковий супровід усіх ігрових подій. Для запобігання накладанню звуків при частих подіях, таких як стрільба та знищення ворогів, реалізовано механізм часу відновлення між відтвореннями. Кожен тип звукового ефекту має власну змінну `lastSoundTime`, що порівнюється з поточним

часом при кожному запиті на відтворення. Для підвищення варіативності звучання застосовується випадкова зміна висоти тону через властивість `pitch` компонента `AudioSource` у заданому діапазоні значень.

Кожен кадр `Update`-цикл координується менеджерами: обробляється ввід гравця, оновлюються таймери здібностей, перевіряються умови перемоги через `WinConditionManager`. `WinConditionManager` підтримує два типи умов завершення сесії: виживання протягом заданого часу та знищення визначеної кількості ворогів. Через рівні інтервали часу гравець отримує бонусні монети за виживання, що стимулює активний ігровий процес. `OutOfBoundsManager` відстежує позицію гравця відносно кругової межі арени та застосовує відсотковий штраф від максимального здоров'я при виході за її межі. Послідовність переходів між фазами ігрової сесії наведено на рисунку 3.4.

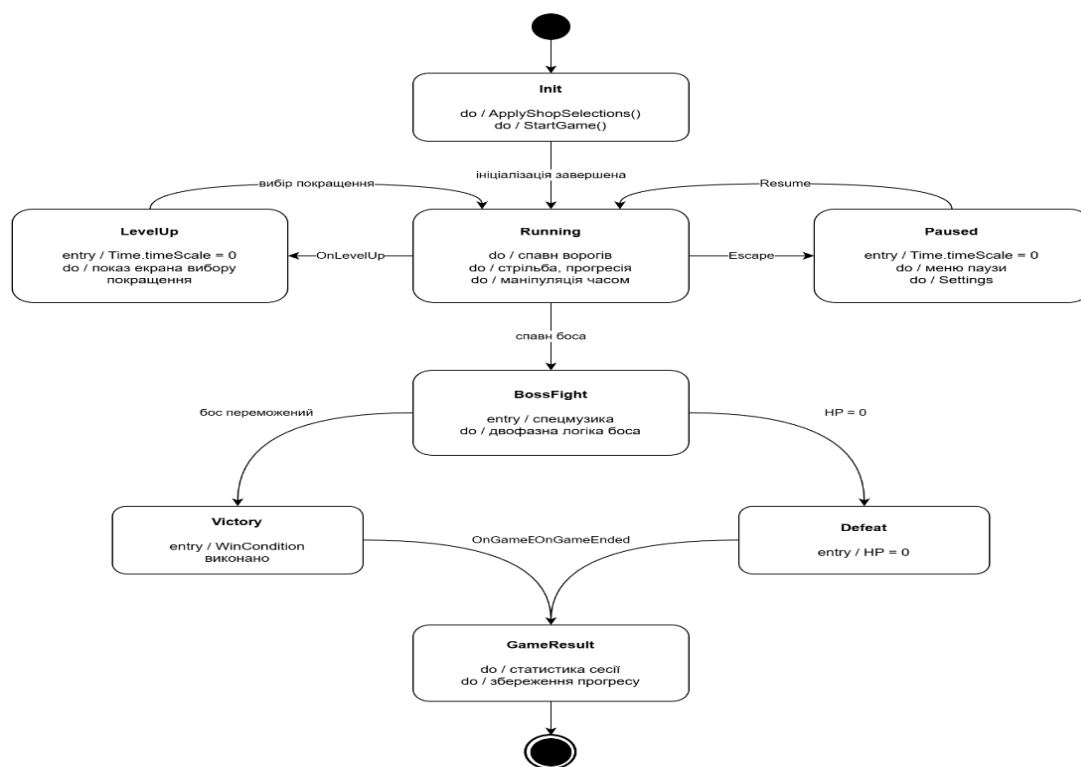


Рисунок 3.4 — Діаграма станів ігрової сесії

Після ініціалізації гра переходить у стан `Running`, де відбувається основний ігровий процес. При натисканні `Escape` активується стан `Paused` із зупинкою `Time.timeScale` та відображенням меню паузи. При підвищенні рівня активується

стан LevelUp із зупинкою часу та відображенням екрану вибору покращень. При появі боса активується стан BossFight з відповідною музикою та двофазною логікою поведінки. Сесія завершується переходом у стан Victory або Defeat, після чого відображається GameResultScreen зі статистикою сесії та збереженням накопиченого прогресу.

### 3.3 Система ворогів та навігації

Система ворогів побудована на принципі єдиного універсального класу з конфігурацією через ScriptableObjects, що дозволяє створювати різні типи ворогів без дублювання коду. Усі вороги реалізовані через спільний клас EnemyAI з конфігурацією через EnemyData ScriptableObject, який визначає характеристики конкретного типу ворога: максимальне здоров'я, швидкість руху, тип атаки, шкоду та значення досвіду при знищенні. Клас EnemyAI успадковує MonoBehaviour та використовує компонентну архітектуру Unity: компонент NavMeshAgent відповідає за навігацію, компонент Rigidbody за фізичне переміщення, компонент Health за стан здоров'я, а компонент TimeScaleObject за підтримку локального часового масштабу.

У EnemyData визначається один з трьох типів атаки через перелік EnemyAttackType. Тип Contact реалізує контактну шкоду при зіткненні з гравцем через метод OnCollisionStay з часом відновлення між атаками. Тип Ranged реалізує дистанційну атаку снарядами через метод FireProjectile при знаходженні гравця в радіусі attackRange. Тип Area реалізує атаку по площі через Physics.OverlapSphere при знаходженні гравця в радіусі areaAttackRange. На основі цього значення клас EnemyAI обирає відповідну стратегію поведінки у методі HandleAttack, що відповідає патерну Strategy. У проєкті реалізовано п'ять типів ворогів: Basic, Fast та Tank з контактною атакою, Shooter з дистанційною та Bruiser з атакою по площі. Конфігурація через ScriptableObjects дозволяє створювати нові типи ворогів без змін у коді — достатньо створити новий екземпляр EnemyData та заповнити

відповідні поля в інспекторі Unity. Це суттєво спрощує процес балансування гри, оскільки всі числові параметри доступні для редагування безпосередньо в редакторі без необхідності перекомпіляції проєкту. Приклад конфігурації EnemyData ScriptableObject в інспекторі Unity наведено на рисунку 3.5.

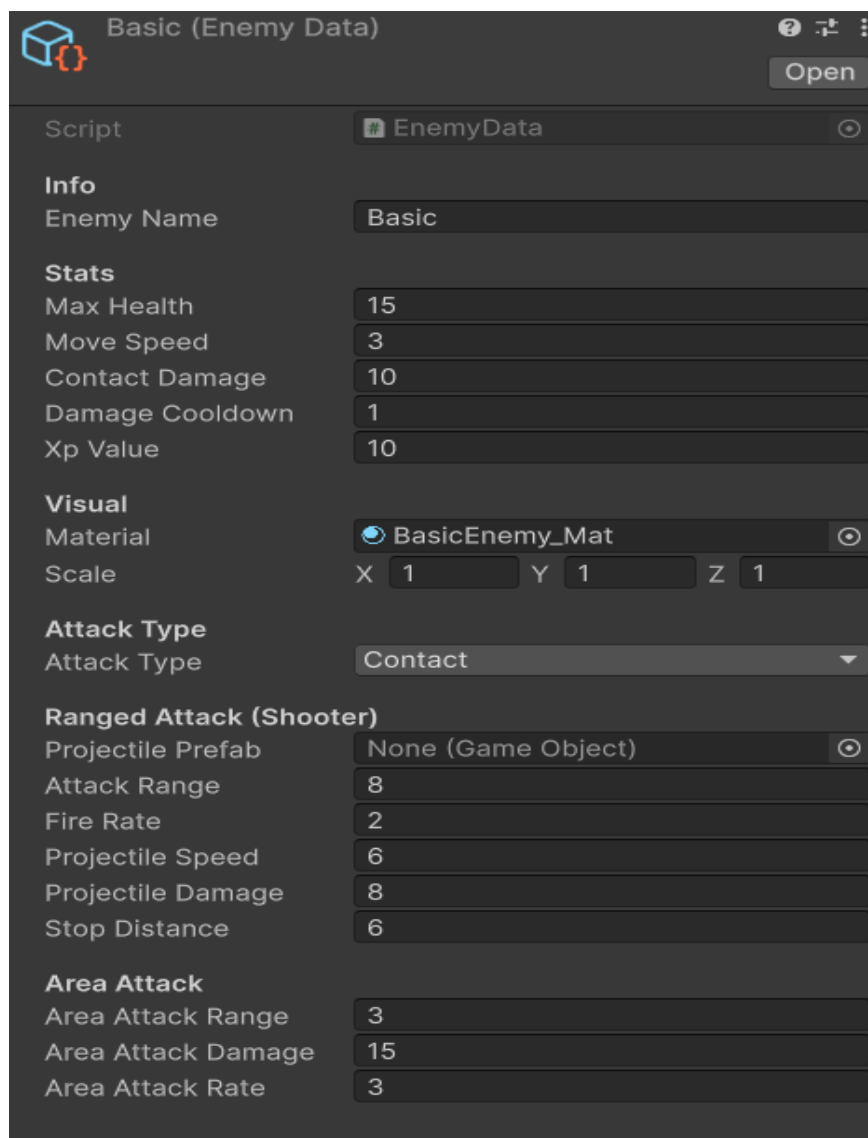


Рисунок 3.5 — Конфігурація EnemyData в інспекторі Unity

Для навігації ворогів використовується система NavMesh, запечена у кожній сцені. Клас EnemyAI містить компонент NavMeshAgent з вимкненими прапорцями `updatePosition` та `updateRotation`, що дозволяє керувати фізичним переміщенням через `Rigidbody` самостійно [19]. У методі `MoveTowardsTarget` агент отримує бажаний вектор руху через властивість `desiredVelocity` та передає його до `Rigidbody`

через `linearVelocity`. Обертання ворога розраховується через `Quaternion.Slerp` у напрямку руху з коефіцієнтом плавності. Такий підхід поєднує переваги `NavMesh`-навігації в обхід перешкод з коректною фізичною взаємодією об'єктів. Позиція агента синхронізується з позицією фізичного об'єкта через присвоєння `agent.nextPosition` після кожного кроку фізики.

Клас `EnemySpawner` відповідає за динамічне створення ворогів протягом сесії. Точки створення визначаються випадково на `NavMesh` у заданому діапазоні відстаней від гравця через метод `NavMesh.SamplePosition` з максимальною кількістю спроб. Якщо жодна спроба не дала валідної точки, створення пропускається. Інтервал між створенням ворогів зменшується з часом відповідно до параметра `intervalDecreasePerMinute`, але не опускається нижче мінімального значення `minSpawnInterval`. Вибір типу ворога здійснюється через систему ваг: кожному типу призначається числова вага у `LevelConfig`, а вибір відбувається через генерацію випадкового числа у діапазоні суми всіх ваг з подальшим кумулятивним порівнянням.

Бос гри реалізований через окремий клас `BossAI` з двофазною логікою поведінки. При здоров'ї понад 50 відсотків від максимального бос використовує повільний одиночний постріл у напрямку гравця з інтервалом `fireRatePhase1`. При досягненні порогу переходу метод `CheckPhaseTransition` запускає корутину `TransitionToPhase2`, яка активує візуальний ефект переходу та змінює колір рендерера боса. У другій фазі швидкість руху збільшується до `moveSpeedPhase2`, а тип атаки змінюється на розсіяний постріл трьома снарядами з кутом `spreadAngle` між ними та зменшеним інтервалом `fireRatePhase2`. Поява боса активується класом `BossSpawner` після знищення гравцем визначеної кількості ворогів, що задається параметром `killsToSpawnBoss` у конфігурації рівня. Снаряди ворогів та боса реалізують підтримку часових механік: при зупинці часу вони застигають зі збереженням вектора швидкості у змінній `frozenVelocity`, який відновлюється після завершення ефекту, а при відмотуванні повертаються до точки створення.

### 3.4 Система маніпуляції часом

Система маніпуляції часом є ключовою унікальною механікою розроблюваного продукту. Її реалізація вирішує нетривіальну технічну задачу: забезпечити незалежний часовий стан для кожного об'єкта сцени без використання глобального `Time.timeScale`, який впливає на всю сцену одночасно.

Базовим компонентом системи є клас `TimeScaleObject`, що надає кожному об'єкту власну змінну `localTimeScale`, незалежну від глобального часу сцени. Властивість `LocalDeltaTime` обчислюється як добуток `Time.deltaTime` та `localTimeScale` і використовується замість стандартного `Time.deltaTime` при розрахунку переміщення об'єктів. Це дозволяє одночасно підтримувати об'єкти у різних часових станах: одні заморожені з `localTimeScale` рівним нулю, інші рухаються з нормальною або зміненою швидкістю. Методи `EnterTimezone` та `ExitTimezone` дозволяють зовнішнім системам змінювати часовий коефіцієнт об'єкта при взаємодії з часовими зонами.

Клас `RewindManager` керує механікою відмотування часу, що активується клавішею `R`. Обидві часові здібності, `Rewind` та `TimeStop`, використовують спільний ресурс `Time Energy`, що унеможливорює їх одночасне використання та змушує гравця обирати між ними залежно від ситуації. `Time Energy` витрачається зі швидкістю `energyCostPerSecond` під час відмотування та відновлюється зі швидкістю `energyRegenPerSecond` у звичайному стані. При повному вичерпанні ресурсу застосовується час відновлення `depletedCooldown` тривалістю три секунди, тоді як при звичайному завершенні здібності час відновлення становить одну секунду.

Кожен об'єкт з підтримкою відмотування має компонент `Rewindable`, що щокадру записує знімок стану у кільцевий буфер через структуру `RewindSnapshot`. Знімок містить позицію, обертання та поточне здоров'я об'єкта. Буфер реалізований через `LinkedList` з обмеженням максимальної кількості знімків, що розраховується як добуток `maxRecordTime` та цільової частоти кадрів 60. При досягненні ліміту найстаріші знімки автоматично видаляються з кінця списку. При відмотуванні

знімки вилучаються з початку списку та застосовуються до об'єкта через плавну інтерполяцію позиції та обертання. Компонент Rewindable є частиною загальної структури компонентів об'єкта гравця, що також включає PlayerMovement, WeaponController, PlayerStats, Health та TimeScaleObject. Структуру компонентів об'єкта гравця в інспекторі Unity наведено на рисунку 3.6.

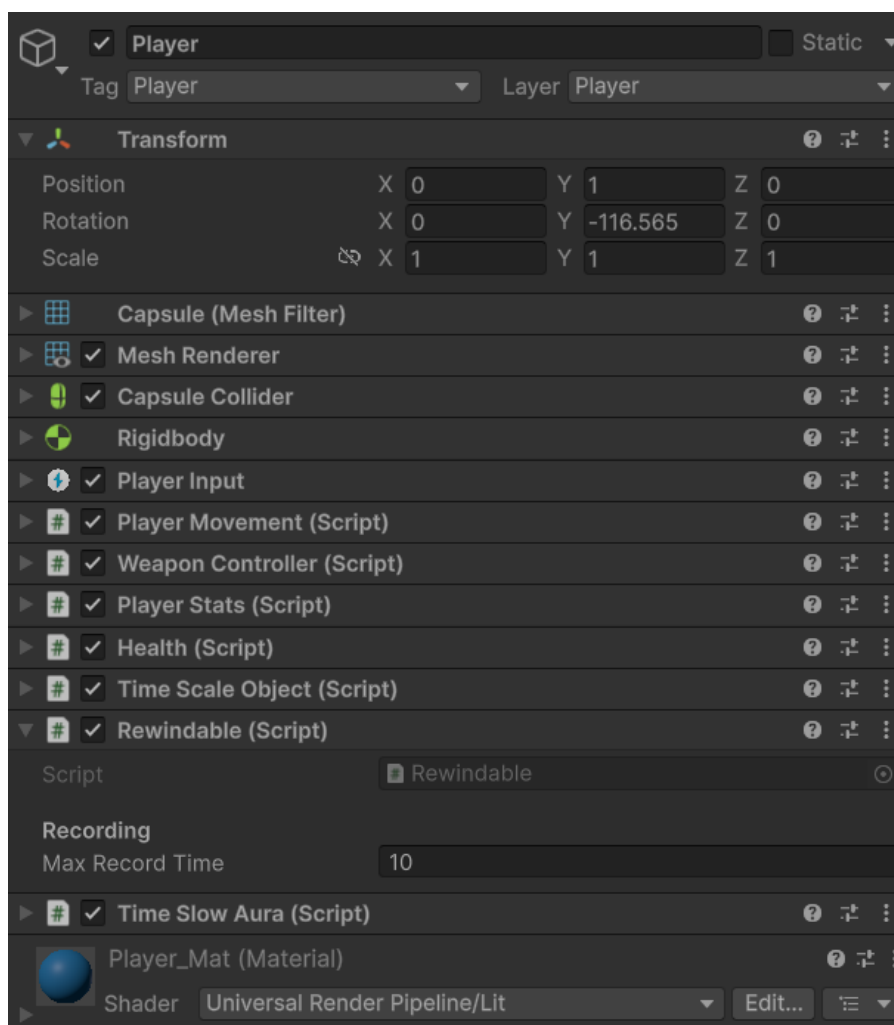


Рисунок 3.6 — Структура компонентів об'єкта гравця в інспекторі Unity

Під час активного відмотування RewindManager знаходить усі екземпляри Rewindable на сцені через FindObjectsByType та викликає їх метод StartRewind. Вороги без компонента Rewindable або з порожнім буфером застигають через вимкнення компонента EnemyAI. EnemySpawner призупиняє створення нових ворогів, стрільба гравця блокується, кулі та снаряди повертаються до точки генерації через застосування вектора у зворотному напрямку до їх Rigidbody. При

досягненні точки появи снаряди повертаються до пулу або знищуються. Обидва менеджери синхронізують час відновлення через метод `SetExternalCooldown`, що гарантує неможливість миттєвого перемикання між здібностями. Алгоритм роботи системи відмотування часу наведено на рисунку 3.7.

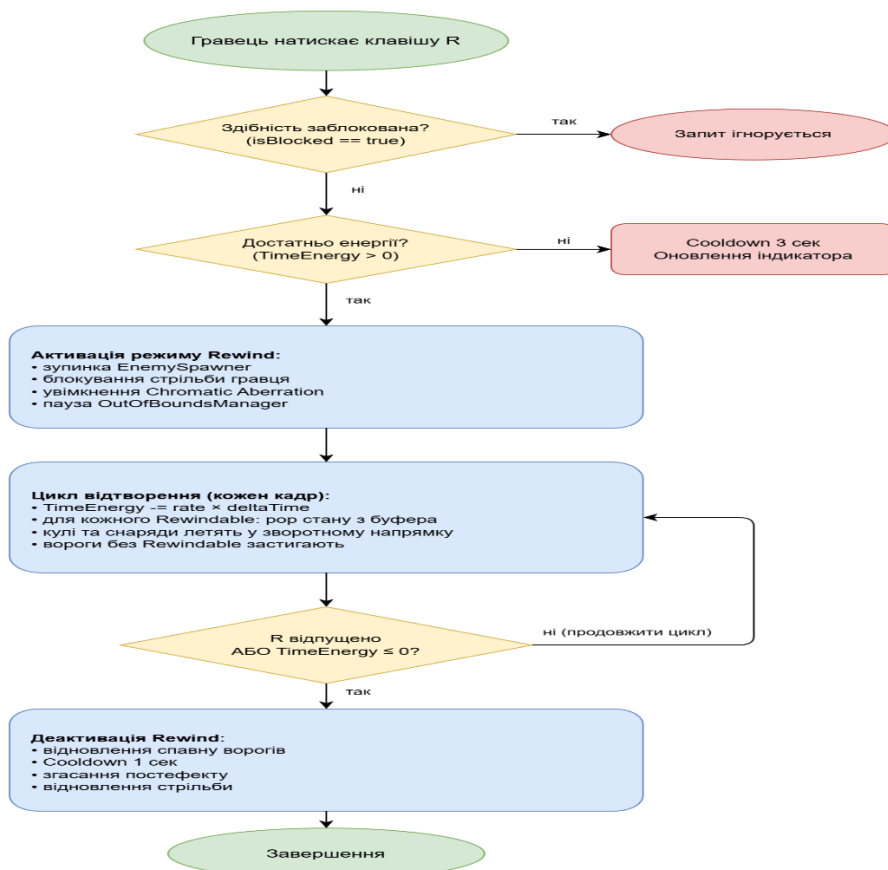


Рисунок 3.7 — Алгоритм роботи системи відмотування часу

Клас `TimeStopManager` керує механікою зупинки часу, що активується клавішею `E`. При активації всі вороги заморожуються через встановлення `localTimeScale` в нуль через компонент `TimeScaleObject`. Снаряди застигають зі збереженням вектора швидкості у змінній `frozenVelocity`, який відновлюється після завершення ефекту. `EnemySpawner` призупиняє роботу, а `OutOfBoundsManager` зупиняє таймер штрафної шкоди. Гравець зберігає повну свободу пересування та стрільби під час зупинки часу, що створює тактичні можливості для проходження крізь замерзлих ворогів.

Локальні часові зони реалізовані через клас `TimeZone`, що являє собою тригерну область з власним часовим коефіцієнтом. При входженні об'єкта з компонентом `TimeScaleObject` у тригер викликається метод `EnterTimeZone` з відповідним коефіцієнтом: значення менше одиниці для зони уповільнення та більше одиниці для зони прискорення. При виході з тригера викликається `ExitTimeZone`, що відновлює стандартний коефіцієнт. Гравець та вороги обробляють часові зони незалежно, оскільки кожен має власний компонент `TimeScaleObject`. Це створює тактичні ситуації, де гравець може розміщувати зони уповільнення на шляху ворогів, зберігаючи власну нормальну швидкість.

Клас `TimeEffectsVFX` керує постефектами URP, що візуально підкреслюють активні часові механіки. При відмотуванні активується хроматична аберація через `ChromaticAberration` компонент профілю `Volume`. При зупинці часу активується десатурація через `ColorAdjustments` та збільшення інтенсивності віньєтування через `Vignette`. Усі переходи між ефектами реалізовані через плавну інтерполяцію з використанням `Time.unscaledDeltaTime`, що забезпечує коректну роботу незалежно від стану глобального часу сцени.

### 3.5 Система прогресії та збереження

Система прогресії забезпечує накопичення та збереження досягнень гравця як в межах однієї сесії, так і між запусками гри. Вона складається з кількох взаємопов'язаних підсистем: досвіду та рівнів, покращень, метапрогресу та магазину. Кожна підсистема реалізована як окремий менеджер з чітко визначеною відповідальністю та слабкою зв'язкою з іншими компонентами через події `C#`. Взаємодія між підсистемами побудована за патерном `Observer`: зміна стану однієї підсистеми публікує відповідну подію, на яку реагують інші без прямої залежності між ними.

### 3.5.1 Система досвіду та покращень

Клас `XPManager` відповідає за накопичення досвіду та підвищення рівня гравця протягом сесії. При знищенні ворога компонент `Health` публікує подію `OnDeath`, після чого створюється орб досвіду `XPOrb` з відповідним значенням. Орб притягується до гравця при входженні в радіус магніту, що визначається параметром `coinMagnetRange` з `PlayerStats`. При підборі кулі з досвідом гравцем метод `AddXP` додає отримане значення до поточного досвіду з урахуванням множника `xrBoostMultiplier`. Для обробки підвищення рівня використовується цикл `while` замість одиночної умови, що коректно обробляє випадок отримання кількох рівнів за один кадр. При кожному підвищенні рівня публікується подія `OnLevelUp`, до якої підписаний `LevelUpManager`.

Клас `LevelUpManager` керує чергою екранів вибору покращень через колекцію `Queue`. Якщо гравець отримує кілька рівнів одночасно, відповідна кількість запитів додається до черги та обробляється послідовно після вибору кожного покращення. При відображенні екрану вибору `Time.timeScale` встановлюється в нуль для зупинки ігрового процесу. Після вибору покращення гравцем `Time.timeScale` відновлюється та перевіряється наявність наступного запиту у черзі.

Клас `UpgradeManager` відстежує кількість взятих накопичень кожного покращення через колекцію `Dictionary` з ключем типу `UpgradeData`. Метод `GetRandomUpgrades` фільтрує покращення з вичерпаним лімітом накопичень та обирає задану кількість випадкових варіантів з доступних. Реалізовано десять типів покращень: шість з обмеженням у три накопичення — підвищення шкоди, швидкості, частоти стрільби, запасу `Time Energy`, пробивання куль та радіуса часової аури; чотири без обмеження накопичень — магніт монет, бонус досвіду, шанс ухилення та додаткове випадання монет. При застосуванні покращення метод `ApplyStatIncrease` викликає відповідний метод `PlayerStats` залежно від значення поля `statType` у `ScriptableObject UpgradeData`.

Кожне покращення конфігурується як окремий екземпляр `ScriptableObject` типу `UpgradeData`, що містить назву, опис, тип характеристики `statType`, значення приросту `increaseAmount` та максимальну кількість накопичень `maxStacks`. Значення `maxStacks` визначається індивідуально для кожного покращення залежно від його впливу на баланс гри. Такий підхід дозволяє додавати нові покращення без змін у коді — достатньо створити новий екземпляр `UpgradeData` та заповнити відповідні поля в редакторі [20]. Конфігурацію `UpgradeData ScriptableObject` в інспекторі Unity наведено на рисунку 3.8.

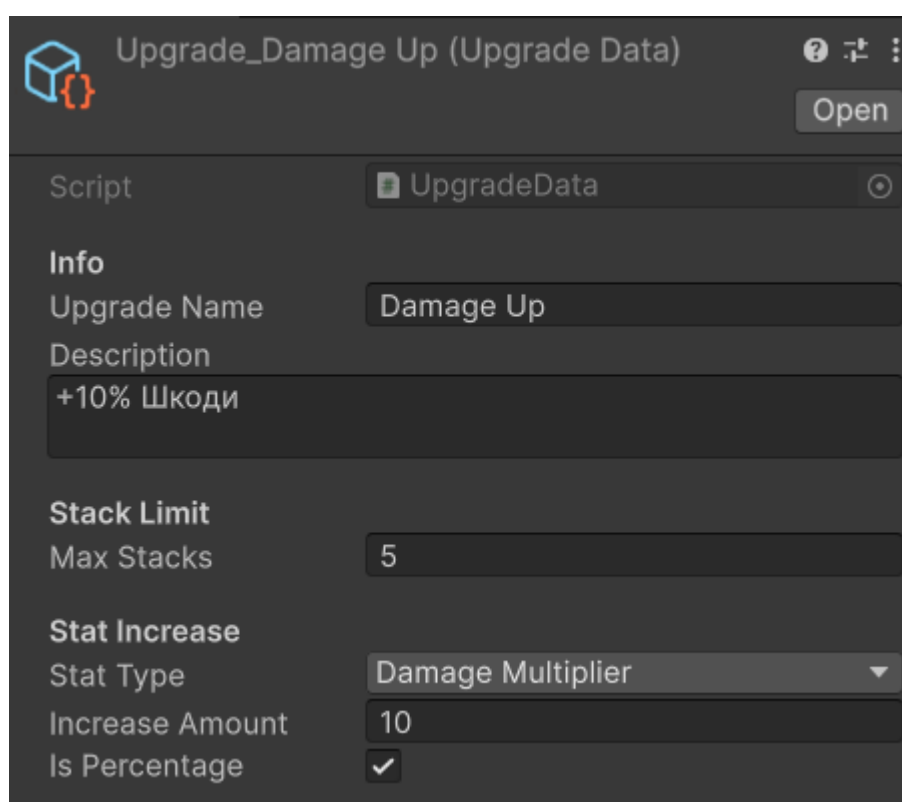


Рисунок 3.8 — Конфігурація `UpgradeData ScriptableObject` в інспекторі Unity

Клас `CoinManager` відповідає за облік монет протягом сесії та взаємодію з системою збереження. Метод `AddCoins` збільшує поточну кількість монет та публікує подію `OnCoinsChanged`, до якої підписаний компонент `CoinUI` для оновлення інтерфейсу. Окремо відстежується кількість монет, зібраних протягом поточної сесії, через змінну `coinsCollectedThisSession`, що використовується для

відображення статистики на екрані результатів. При збереженні прогресу загальна кількість монет передається до SaveManager через метод AddCoins.

### 3.5.2 Система збереження та магазину

Клас SaveManager реалізує збереження та завантаження прогресу між сесіями через механізм PlayerPrefs операційної системи. Зберігаються такі дані: загальна кількість монет через ключ TotalCoins, перелік розблокованих рівнів через ключ UnlockedLevels, перелік придбаних предметів через ключ OwnedItems, вибрана зброя через ключ SelectedWeapon та вибране уміння через ключ SelectedSkill. Метод AddCoins збільшує збережену кількість монет, метод SpendCoins зменшує її після перевірки достатності коштів, метод UnlockLevel додає індекс рівня до переліку розблокованих. SaveManager створюється у сцені головного меню з прапорцем DontDestroyOnLoad, що забезпечує його доступність у всіх наступних сценах без повторної ініціалізації.

Клас ShopManager реалізує магазин між рівнями, де гравець може придбати нові види зброї та уміння за накопичені монети. Картки предметів формуються динамічно на основі списку ShopItemData ScriptableObject при завантаженні сцени магазину. Кожен ShopItemData містить назву предмета, опис, вартість, іконку та посилання на відповідний WeaponData або SkillData. При натисканні кнопки придбання ShopManager перевіряє наявність достатньої кількості монет через CoinManager та відсутність предмета у переліку вже придбаних через SaveManager. При успішній покупці монети списуються, ідентифікатор предмета додається до переліку OwnedItems у PlayerPrefs, а кнопка замінюється індикатором володіння. Вибір активної зброї та уміння зберігається окремо і завантажується GameManager на початку кожної сесії через метод ApplyShopSelections.

Клас LevelSelectManager керує інтерфейсом вибору рівня та відображенням стану доступності кожного рівня. При завантаженні сцени менеджер зчитує перелік розблокованих рівнів з SaveManager та оновлює стан карток рівнів. Картки заблокованих рівнів відображаються з відповідним індикатором та заблокованою

кнопкою переходу. При успішному завершенні рівня WinConditionManager викликає метод UnlockLevel у SaveManager, що розблоковує наступний рівень перед відображенням екрану результатів.

### 3.5.3 Модуль інтерактивного навчання

Клас TutorialManager проводить гравця через дев'ять кроків покрокового знайомства з усіма ігровими механіками. Кожен крок конфігурується в інспекторі Unity через масив рядків з текстом повідомлення та масив булевих значень з прапорцями можливості пропуску. Послідовність кроків реалізована через корутину, що забезпечує зручне управління асинхронними очікуваннями без блокування основного потоку [21].

Клавіша F пропускає поточний крок з обов'язковим очікуванням відпускання клавіші після пропуску, що запобігає ланцюжковому пропусканню кількох кроків за одне натискання. Для кроку з першим ворогом та фінального кроку з чотирма ворогами реалізовано спеціальну логіку очікування знищення визначеної кількості противників. Ворог навчального рівня має збільшене значення досвіду для гарантованого підвищення рівня та гарантований скид монети, що дозволяє продемонструвати відповідні механіки в контрольованих умовах.

Здібності гравця блокуються методами SetSkillBlocked, TimeStopManager.SetBlocked та RewindManager.SetBlocked і розблоковуються поетапно відповідно до прогресу навчання. Це гарантує, що гравець ознайомиться з кожною механікою у визначеній послідовності, не маючи можливості використати здібності до їх пояснення. Здоров'я гравця автоматично відновлюється при падінні нижче мінімального порогу minPlayerHP, що унеможливорює загибель під час навчання. Після завершення модуля навчання SaveManager розблоковує перший ігровий рівень та відображається окремий екран результатів TutorialResultScreen.

## 4 ВЗАЄМОДІЯ КОРИСТУВАЧА З СИСТЕМОЮ

У цьому розділі розглядаються практичні аспекти взаємодії користувача з розробленою грою DimeLoop Survivors. Описуються апаратні та програмні вимоги до середовища запуску, процедура інсталяції, інтерфейс користувача, типовий сценарій роботи від першого запуску до завершення ігрової сесії, а також результати тестування програмного забезпечення.

### 4.1 Системні вимоги та інсталяція

Розроблена гра є програмним продуктом для платформи Windows і не потребує складної процедури встановлення. Цільовою операційною системою є Windows 10 або Windows 11 у 64-розрядній версії. Мінімальні апаратні вимоги для коректного запуску гри включають процесор Intel Core i3-6100 або AMD еквівалент, 4 ГБ оперативної пам'яті, відеокарту з підтримкою DirectX 11 та обсягом відеопам'яті не менше 2 ГБ, наприклад GeForce GTX 660, а також не менше 150-300 МБ вільного місця на диску. Зазначені вимоги є мінімальними і забезпечують стабільну роботу гри на налаштуваннях за замовчуванням.

Оскільки гра реалізована на рушії Unity 6 з використанням Universal Render Pipeline та постефектів, відеокарта є ключовим компонентом для забезпечення прийнятної частоти кадрів. На системах з більш потужним графічним процесором гра працюватиме з вищою частотою кадрів та якістю постефектів, зокрема хроматичної аберації при відмотуванні часу та десатурації при зупинці часу. Підключення до мережі не є обов'язковим, оскільки гра є однокористувацьким продуктом без мережевих функцій.

Інсталяція не потребує нічого специфічного — достатньо розпакувати архів з дистрибутивом гри у будь-який каталог з правом запису. Дистрибутив доступний для завантаження з репозиторію GitHub, а також може бути розповсюджений через платформу itch.io, що є популярним майданчиком для поширення інді-ігор. Каталог

містить виконуваний файл DimeLoopSurvivors.exe, теку DimeLoopSurvivors\_Data з ресурсами гри та супровідні файли UnityPlayer.dll і UnityCrashHandler64.exe. Запуск здійснюється подвійним клацанням на виконуваному файлі без необхідності попередньої інсталяції додаткового програмного забезпечення. Прогрес гравця зберігається автоматично у реєстрі операційної системи Windows через механізм PlayerPrefs, що унеможливлює випадкову втрату накопиченого прогресу між сесіями.

## 4.2 Опис інтерфейсу користувача

Інтерфейс користувача розроблено з урахуванням принципів інтуїтивності та мінімалізму. Усі елементи інтерфейсу реалізовані засобами Unity UI Canvas з використанням TextMeshPro для відображення тексту. Загальний потік навігації між сценами побудований за принципом лінійного прогресу: головне меню — вибір рівня — магазин — ігрова сцена — екран результатів.

Головне меню є першим екраном, який бачить гравець після запуску гри. Зовнішній вигляд головного меню наведено на рисунку 4.1.

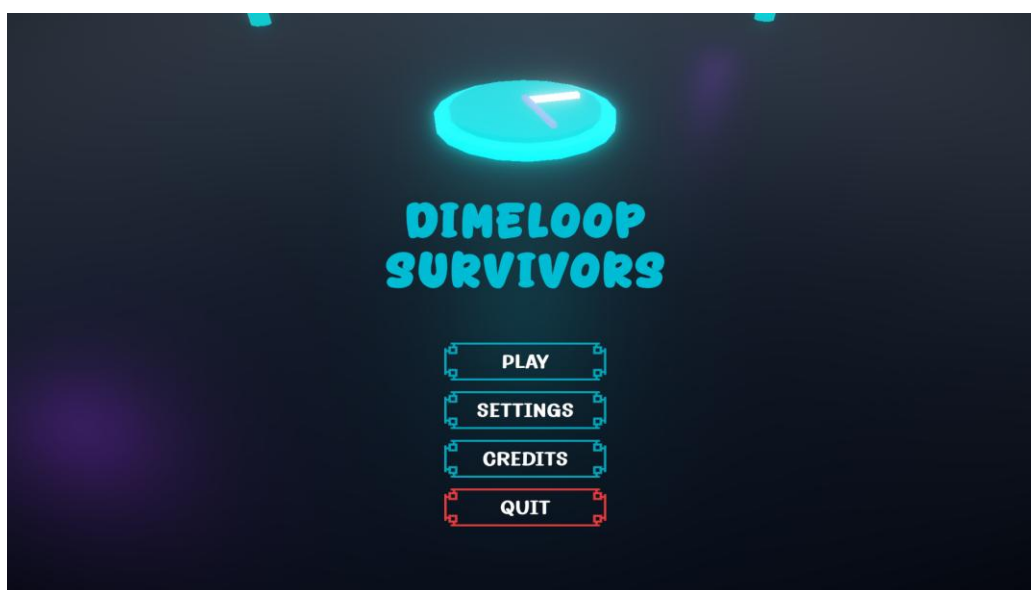


Рисунок 4.1 — Головне меню гри

Воно містить чотири основні кнопки навігації: "Play" для переходу до вибору рівня, "Settings" для відкриття панелі налаштувань, "Credits" для перегляду титрів та "Quit" для завершення роботи програми. Панель налаштувань надає можливість регулювання гучності музики та звукових ефектів через повзунки, а також керування параметрами відображення.

Після натискання кнопки "Play" гравець переходить до сцени вибору рівня, де відображаються картки доступних рівнів: ТUTORІАЛ, Лісова Арена та Темний Ліс. Початково доступним є лише ТUTORІАЛ, решта рівнів розблоковуються послідовно після успішного проходження попередніх. Заблоковані рівні відображаються з відповідним індикатором та неактивною кнопкою переходу. Поряд з кожною картою рівня розміщена кнопка переходу до магазину.

Магазин надає гравцеві можливість придбати додаткові види зброї та уміння за накопичені монети перед початком рівня. Інтерфейс магазину розділений на дві секції: зброя та уміння. Кожен предмет відображається у вигляді картки з назвою, описом та вартістю. Вже придбані предмети позначаються індикатором володіння замість кнопки придбання. Поточна кількість монет відображається у верхньому правому куті екрану. Зовнішній вигляд магазину наведено на рисунку 4.2.

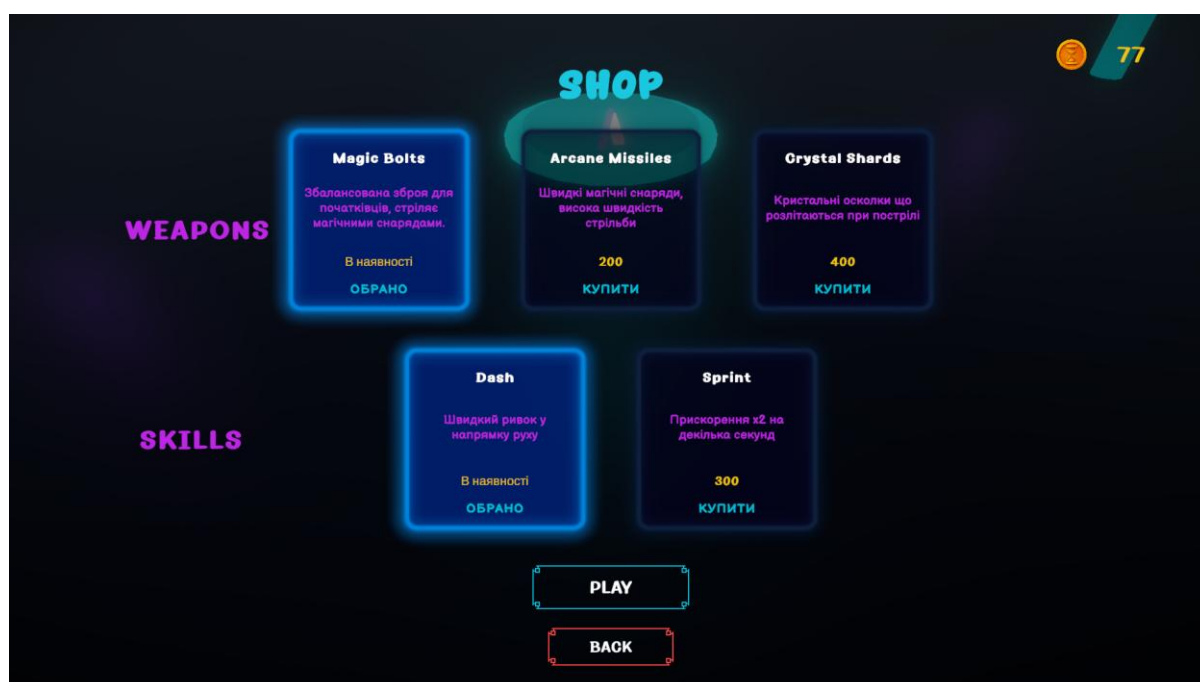


Рисунок 4.2 — Інтерфейс магазину

Ігровий інтерфейс під час сесії відображає всю необхідну інформацію про поточний стан гри. У верхній частині екрану розміщено таймер сесії та лічильник знищених ворогів відносно цілі. У лівій частині відображаються індикатори здоров'я гравця, запасу Time Energy з числовим значенням та індикатор поточного режиму стрільби. Шкала досвіду та поточний рівень гравця розміщені у нижній частині екрану. У правому верхньому куті відображається поточна кількість монет. При активації часових механік у центрі екрану з'являється відповідний текстовий індикатор. Зовнішній вигляд ігрової сцени наведено на рисунку 4.3.



Рисунок 4.3 — Ігрова сцена з активним постефектом часової механіки

При підвищенні рівня ігровий процес зупиняється та відображається екран вибору покращення з трьома випадковими варіантами. Система формує список доступних покращень, виключаючи ті що досягли максимальної кількості накопичень, та обирає три випадкові варіанти з доступних. Кожна картка містить назву покращення, його опис та лічильник поточної кількості накопичень відносно максимально допустимої. Гравець обирає одне покращення, після чого ігровий процес відновлюється. Якщо гравець отримує кілька рівнів одночасно, запити на

вибір покращень додаються до черги та обробляються послідовно. Зовнішній вигляд екрану вибору покращення наведено на рисунку 4.4.

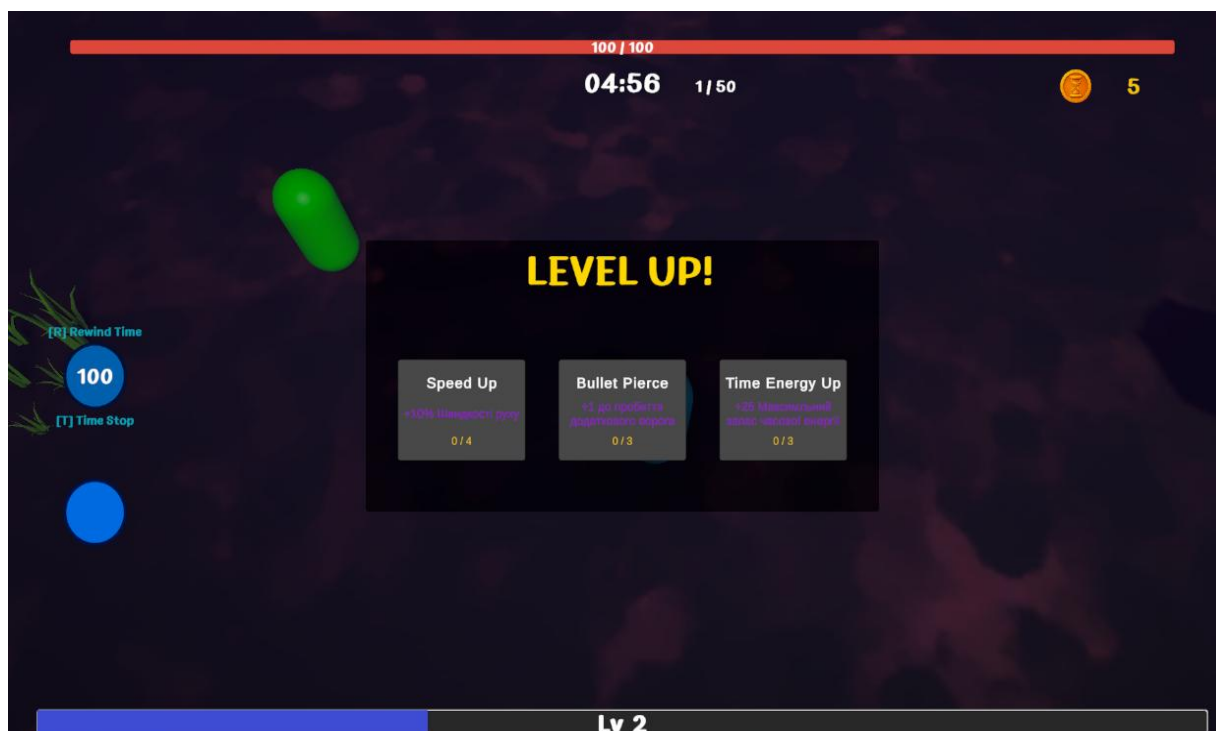


Рисунок 4.4 — Екран вибору покращення при підвищенні рівня

Кожна картка містить назву покращення, його опис та лічильник поточної кількості накопичень відносно максимально допустимої. Гравець обирає одне покращення, після чого ігровий процес відновлюється.

Екран результатів відображається після завершення сесії незалежно від її результату. При перемозі відображається заголовок "VICTORY", при поразці — "LOST". Екран містить статистику сесії: час виживання, кількість знищених ворогів, бонус виживання та загальну кількість зароблених монет. Бонус виживання нараховується за кожні 30 секунд активної ігрової сесії та додається до загальної суми зароблених монет. Зароблені монети автоматично зберігаються між сесіями та можуть бути витрачені у магазині перед наступним рівнем. Для навігації після завершення сесії доступні три кнопки: перехід до наступного рівня, повернення до вибору рівня та повторний запуск поточного рівня. При перемозі

також автоматично розблоковується наступний рівень, якщо він був недоступний раніше. Зовнішній вигляд екрану результатів наведено на рисунку 4.4.



Рисунок 4.4 — Екран результатів ігрової сесії

Таким чином, інтерфейс користувача розробленої гри забезпечує послідовну та інтуїтивну взаємодію з усіма ігровими підсистемами. Кожен екран виконує чітко визначену функцію та містить лише необхідні елементи керування, що дозволяє гравцю зосередитись на ігровому процесі без зайвого навантаження інформацією.

### 4.3 Сценарій роботи користувача

Сценарій роботи користувача починається з моменту першого запуску гри. Після відкриття виконуваного файлу гравець потрапляє до головного меню, звідки розпочинається весь подальший ігровий процес. Оскільки на початку доступним є лише навчальний рівень, рекомендованим першим кроком є його проходження для ознайомлення з усіма ігровими механіками.

Після натискання кнопки "Play" гравець переходить до сцени вибору рівня. На початку доступна лише картка навчального рівня, решта рівнів заблоковані.

Перед запуском будь-якого рівня гравець має можливість відвідати магазин, натиснувши відповідну кнопку поряд з карткою рівня. У магазині можна придбати додаткові види зброї та уміння за накопичені монети, після чого натиснути кнопку "Play" для запуску рівня.

Навчальний рівень є обов'язковим першим етапом для нових гравців. Він складається з дев'яти послідовних кроків, кожен з яких знайомить гравця з окремою механікою: переміщенням, стрільбою, збором досвіду, вибором покращень, використанням уміння, відмотуванням часу, зупинкою часу та часовими зонами. Після успішного завершення модуля навчання розблоковується перший ігровий рівень — Лісова Арена.

Основна ігрова сесія починається з появи гравця на арені з мінімальними характеристиками. Протягом сесії гравець переміщується по арені клавішами WASD, прицілюється мишею та уникає зіткнень з ворогами. Зброя стріляє автоматично у напрямку курсора, або вручну при натисканні лівої кнопки миші залежно від обраного режиму. При накопиченні достатньої кількості досвіду гравець підвищує рівень та обирає одне з трьох запропонованих покращень. Часові здібності активуються клавішами R для відмотування та E для зупинки часу і витрачають спільний ресурс Time Energy, що відображається на відповідному індикаторі.

У процесі сесії складність поступово зростає за рахунок збільшення кількості та різноманітності ворогів. Після знищення визначеної кількості противників з'являється бос рівня, що має дві фази поведінки. У першій фазі бос повільно пересувається та веде одиночний вогонь, у другій, після досягнення 50 відсотків здоров'я, прискорюється та переходить на розсіяний постріл трьома снарядами одночасно. Знищення боса завершує сесію перемогою.

Сесія також може завершитись поразкою у випадку загибелі гравця. Загибель настає при вичерпанні здоров'я від контакту з ворогами, їхніми снарядами або тривалого перебування за межами кругової арени. Межа арени супроводжується візуальним попередженням, а шкода від перебування за її межами зростає з часом.

Після завершення сесії відображається екран результатів зі статистикою та накопиченими монетами. При перемозі гравець може перейти до наступного рівня, повторити поточний або повернутись до вибору рівня. Накопичені монети зберігаються між сесіями та можуть бути витрачені у магазині перед наступним запуском.

#### 4.4 Тестування програмного забезпечення

Тестування є невід'ємною частиною процесу розробки програмного забезпечення, що дозволяє виявити та усунути помилки до випуску готового продукту. У рамках даної роботи проведено функціональне тестування всіх ключових підсистем гри, а також тестування продуктивності для перевірки стабільності роботи в умовах великої кількості активних об'єктів на сцені.

Функціональне тестування проводилось методом ручного тестування з перевіркою кожної ігрової механіки окремо та у взаємодії з іншими підсистемами. Для кожної підсистеми визначались очікувані результати, після чого перевірялась їх відповідність фактичній поведінці системи. Результати функціонального тестування наведено у таблиці 4.1.

Таблиця 4.1 — Результати функціонального тестування

| Підсистема       | Тестовий сценарій                    | Очікуваний результат         | Результат |
|------------------|--------------------------------------|------------------------------|-----------|
| PlayerMovement   | Переміщення у всіх напрямках         | Рівномірний рух без ковзання | Пройдено  |
| WeaponController | Автоматична та ручна стрільба        | Коректне перемикання режимів | Пройдено  |
| RewindManager    | Відмотування при недостатній енергії | Час відновлення 3 секунди    | Пройдено  |

Кінець таблиці 4.1

|                    |                                     |                                  |          |
|--------------------|-------------------------------------|----------------------------------|----------|
| TimeStopManager    | Зупинка часу з повною енергією      | Заморожування всіх ворогів       | Пройдено |
| TimeZone           | Вхід гравця у зону уповільнення     | Зміна localTimeScale гравця      | Пройдено |
| EnemyAI            | Навігація в обхід перешкод          | Коректний маршрут NavMesh        | Пройдено |
| BossAI             | Перехід до другої фази при 50% HP   | Зміна поведінки та кольору       | Пройдено |
| Health             | Загибель гравця при HP = 0          | Відображення екрану результатів  | Пройдено |
| XPManger           | Підвищення кількох рівнів одночасно | Послідовна черга покращень       | Пройдено |
| SaveManager        | Збереження монет між сесіями        | Коректне завантаження значення   | Пройдено |
| TutorialManager    | Пропуск кроку клавішею F            | Очікування відпускання клавіші   | Пройдено |
| OutOfBoundsManager | Вихід за межі арени                 | Наростаюча шкода від відсотку HP | Пройдено |

Окремої уваги під час тестування потребувала система маніпуляції часом через складність взаємодії між її компонентами. Перевірялись сценарії одночасної активності кількох часових механік, коректність відновлення стану об'єктів після відмотування, поведінка снарядів при зупинці та відновленні часу, а також синхронізація часу відновлення між RewindManager та TimeStopManager. Усі виявлені у процесі тестування помилки були усунуті до формування фінального дистрибутиву.

Тестування продуктивності проводилось за допомогою вбудованого інструменту Unity Profiler, який дозволяє відстежувати навантаження на процесор,

відеокарту та оперативну пам'ять у реальному часі. Тестування виконувалось на цільовій платформі Windows з фіксацією показників частоти кадрів при різній кількості активних об'єктів на сцені. Ігрова сцена з максимальною кількістю активних об'єктів під час тестування наведена на рисунку 4.5.

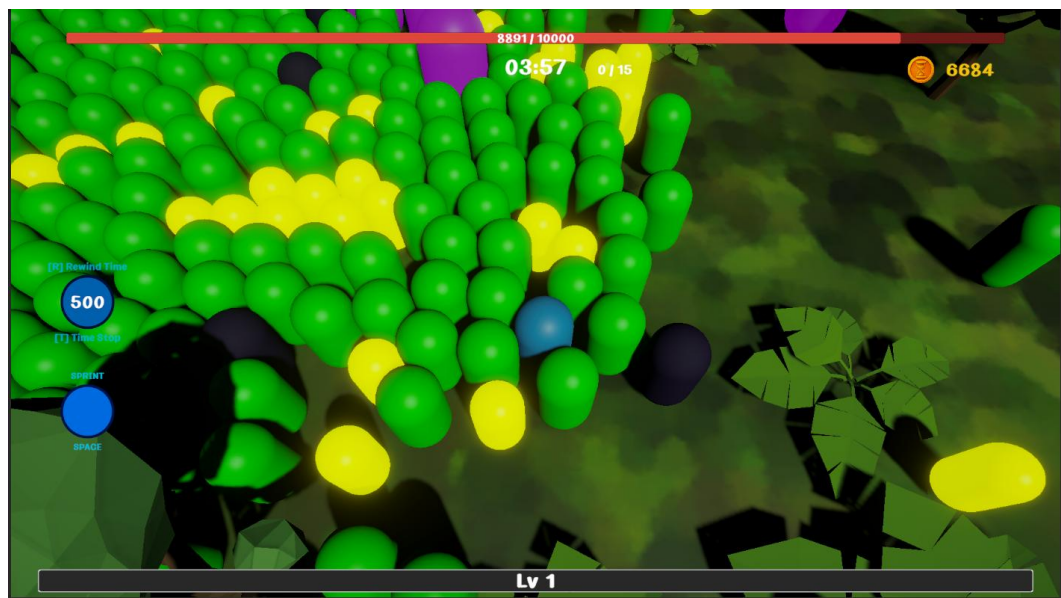


Рисунок 4.6 — Ігрова сцена з максимальною кількістю активних об'єктів

Результати показали стабільну роботу гри з частотою кадрів, що значно перевищує комфортний поріг у 60 кадрів на секунду, навіть при значному завантаженні графічного та центрального процесорів у сценаріях з великою кількістю активних об'єктів на сцені. Найбільше навантаження спостерігається під час активного відмотування часу, що пояснюється необхідністю одночасного оновлення стану всіх компонентів Rewindable. Застосування патерну Object Pool та обмеження глибини буфера відмотування до 10 секунд дозволило утримати продуктивність на прийнятному рівні.

За результатами проведеного тестування підтверджено коректність роботи всіх реалізованих підсистем та відповідність розробленого програмного забезпечення поставленим вимогам.

## ВИСНОВКИ

У результаті виконання дипломної роботи було успішно розроблено комп'ютерну гру DimeLoop Survivors жанру roguelite-survivors з системою маніпуляції часом на базі ігрового рушія Unity 6 та мови програмування C#, завдяки чому поставлену мету досягнуто в повному обсязі, а всі визначені завдання виконано:

1. Теоретичний аналіз аналогів, зокрема Braid, Prince of Persia та Vampire Survivors, дозволив обґрунтувати оригінальність проєкту та обрати комбінацію щокadroвого запису стану об'єктів у кільцевий буфер з локальним часовим масштабом як оптимальне рішення для Unity.

2. На основі проведеного аналізу обґрунтовано вибір технологічного стеку у складі Unity 6 LTS, C#, Universal Render Pipeline, NavMesh, New Input System, Cinemachine та TextMeshPro, а застосування патернів проєктування Singleton, Observer, Object Pool та Strategy забезпечило слабку зв'язку між підсистемами, модульність архітектури та розширюваність системи.

3. Розроблена система є повноцінним програмним продуктом, що реалізує три незалежні часові механіки Rewind, TimeStop та TimeZone, систему ворогів з NavMesh-навігацією та п'ятьма типами противників, двофазного боса, систему прогресії з десятьма типами покращень, метапрогрес через магазин та дев'ятикроковий модуль інтерактивного навчання.

4. Практичним результатом роботи є скомпільована версія гри для Windows, опублікована в репозиторії GitHub, успішне тестування якої підтвердило стабільну продуктивність та коректність функціонування всіх підсистем.

5. Розроблений продукт має значний потенціал для подальшого розвитку: додавання нових ігрових світів та типів ворогів, розширення системи покращень, оптимізації для слабших конфігурацій та дослідження можливостей портування на інші платформи.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Valentine R. We need to decide on a genre name for Vampire Survivors-like games before a really terrible one sticks. URL: <https://www.pcgamer.com/games/roguelike/we-need-to-decide-on-a-genre-name-for-vampire-survivors-like-games-before-a-really-terrible-one-sticks/> (дата звернення: 25.05.2026).
2. Stankovic S. Game Mechanics — Braid. URL: <https://stanislav-stankovic.medium.com/game-mechanics-braid-78c289e95410> (дата звернення: 25.05.2026).
3. McRoberts T. Recreating the time mechanics of Braid. URL: <https://www.gamedeveloper.com/programming/recreating-the-time-mechanics-of-braid-part-1> (дата звернення: 25.05.2026).
4. How Vampire Survivors built its own unlikely genre, the "survivor-like". URL: <https://store.epicgames.com/en-US/news/vampire-survivors-built-genre-autoshooter-bullet-heaven> (дата звернення: 25.05.2026).
5. Time Manipulation Mechanic in Unity — Chapter 1. Medium, 2023. URL: <https://medium.com/@cimenbaran/time-manipulation-mechanic-in-unity-chapter-1-ba16947ff504> (дата звернення: 25.05.2026).
6. Unity Basics - Introduction to C# Scripting. Daniel Ilett, 2021. URL: <https://danielilett.com/2021-01-29-basics-1-intro-to-c-sharp/> (дата звернення: 25.05.2026).
7. Unity 6.3 User Manual. URL: <https://docs.unity3d.com/6000.3/Documentation/Manual/UnityManual.html> (дата звернення: 25.05.2026).
8. Create modular game architecture in Unity with ScriptableObjects. URL: <https://unity.com/resources/create-modular-game-architecture-with-scriptable-objects-ebook> (дата звернення: 25.05.2026).
9. Unity vs. Unreal Engine: Which is Best for Indie Developers? URL: <https://www.cubix.co/blog/unity-vs-unreal-engine-best-indie-developers/> (дата звернення: 25.05.2026).

10. Is Godot a Good Game Engine? URL: <https://gamedesignskills.com/game-design/is-godot-good/> (дата звернення: 25.05.2026).
11. Skeet J. C# in Depth. 4th ed. Shelter Island : Manning Publications, 2019. 528 p. URL: <https://www.manning.com/books/c-sharp-in-depth-fourth-edition> (дата звернення: 25.05.2026).
12. Universal Render Pipeline (URP). URL: <https://docs.unity3d.com/Manual/universal-render-pipeline.html> (дата звернення: 25.05.2026).
13. Unity's new Input System, made easy. Game Dev Beginner, 2021. URL: <https://gamedevbeginner.com/input-in-unity-made-easy-complete-guide-to-the-new-system/> (дата звернення: 25.05.2026).
14. Unity's Navigation and Pathfinding. Dev Genius, 2022. URL: <https://blog.devgenius.io/unitys-navigation-and-pathfinding-f91f4335e5e5> (дата звернення: 25.05.2026).
15. Unity Cinemachine: Advanced Camera Techniques. Wayline, 2023. URL: <https://www.wayline.io/blog/unity-cinemachine-advanced-camera-techniques> (дата звернення: 25.05.2026).
16. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 p. URL: <https://gameprogrammingpatterns.com/> (дата звернення: 25.05.2026).
17. Murray J. W. C# Game Programming Cookbook for Unity 3D. 2nd ed. Boca Raton : CRC Press, 2021. 316 p. URL: <https://www.taylorfrancis.com/books/mono/10.1201/9780429317132/game-programming-cookbook-unity-3d-jeff-murray> (дата звернення: 25.05.2026).
18. Як оптимізувати ігровий проєкт за допомогою патерну ObjectPool. Посібник для Unity-розробників. URL: <https://gamedev.dou.ua/blogs/optimizing-a-game-project-with-object-pool-pattern/> (дата звернення: 25.05.2026).
19. Unity 3D – Smooth Turning with NavMeshAgent AI. URL: <https://keithmaggio.wordpress.com/2019/07/05/unity-3d-smooth-turning-with-navmeshagent-ai/> (дата звернення: 25.05.2026).
20. Unity ScriptableObjects Best Practices. URL: <https://docs.unity3d.com/Manual/class-ScriptableObject.html> (дата звернення: 25.05.2026).

21. Albahari J., Albahari B. C# 12 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2023. 1086 p. URL: <https://www.oreilly.com/library/view/c-12-in/9781098147433/> (дата звернення: 25.05.2026).

## ДОДАТОК А

**Роуглайт гра з механізмами керування часом на базі Unity**

Програмна реалізація системи відмотування часу

НТУУ «КПІ ім. Ігоря Сікорського»

Аркушів 9

Програмні засоби:

- мова програмування — C#;
- ігровий рушій — Unity 6;
- бібліотеки та пакети — UnityEngine, System.Collections.Generic, UnityEngine.InputSystem, UnityEngine.UI, TextMeshPro.

Rewindable.cs:

```
using UnityEngine;
using System.Collections.Generic;

// Records object state every frame and replays it backwards on Rewind.
// Attach to: Player, all enemies, bullets.
[RequireComponent(typeof(TimeScaleObject))]
public class Rewindable : MonoBehaviour
{
    [Header("Recording")]
    [SerializeField] private float maxRecordTime = 5f;

    private LinkedList<RewindSnapshot> snapshots = new LinkedList<RewindSnapshot>();
    private bool isRewinding = false;

    private Health health;
    private Rigidbody rb;

    private void Awake()
    {
        health = GetComponent<Health>();
        rb = GetComponent<Rigidbody>();
    }

    private void Start()
    {
        // Sync max record time with RewindManager
        if (RewindManager.Instance != null)
            maxRecordTime = RewindManager.Instance.GetMaxRewindDuration();
    }
}
```

```

private void Update()
{
    if (isRewinding)
    {
        Rewind();
    }
    else
    {
        Record();
    }
}

// Store one snapshot per frame
private void Record()
{
    float currentHP = health != null ? health.GetCurrentHealth() : 0f;
    bool alive = health != null ? health.IsAlive() : true;

    RewindSnapshot snapshot = new RewindSnapshot(
        transform.position,
        transform.rotation,
        currentHP,
        alive
    );

    snapshots.AddFirst(snapshot);

    // Drop snapshots older than maxRecordTime (~60 fps)
    if (snapshots.Count > Mathf.RoundToInt(maxRecordTime * 60f))
    {
        snapshots.RemoveLast();
    }
}

// Replay snapshots in reverse order
private void Rewind()
{

```

```

if (snapshots.Count == 0)
{
    StopRewind();
    return;
}

RewindSnapshot snapshot = snapshots.First.Value;
snapshots.RemoveFirst();

// Smoothly move towards the recorded transform
transform.position = Vector3.Lerp(
    transform.position,
    snapshot.position,
    Time.deltaTime * 15f
);
transform.rotation = Quaternion.Slerp(
    transform.rotation,
    snapshot.rotation,
    Time.deltaTime * 15f
);

if (health != null && snapshot.isAlive)
{
    health.SetHealthDirect(snapshot.health);
}

// Revive the object if it was alive in the snapshot
if (snapshot.isAlive && health != null && !health.IsAlive())
{
    gameObject.SetActive(true);
}
}

public void StartRewind()
{
    isRewinding = true;

    // Disable physics while rewinding

```

```

        if (rb != null)
        {
            rb.isKinematic = true;
        }
    }

    public void StopRewind()
    {
        isRewinding = false;

        if (rb != null)
        {
            rb.isKinematic = false;
            rb.linearVelocity = Vector3.zero;
        }

        snapshots.Clear();
    }

    public void ClearSnapshots()
    {
        snapshots.Clear();
    }

    public void UpdateMaxRecordTime(float newTime)
    {
        maxRecordTime = newTime;
    }

    public bool IsRewinding => isRewinding;
    public int SnapshotCount => snapshots.Count;
}

```

### RewindManager.cs:

```

using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.UI;
using TMPro;

```

// Manages the Rewind ability: time energy, cooldowns and global rewind state.

```
public class RewindManager : MonoBehaviour
{
    public static RewindManager Instance { get; private set; }

    [Header("Rewind Settings")]
    [SerializeField] private float maxTimeEnergy = 100f;
    [SerializeField] private float energyRegenPerSecond = 15f;
    [SerializeField] private float energyCostPerSecond = 40f;

    [Header("Cooldown Settings")]
    [SerializeField] private float switchCooldown = 1f;
    [SerializeField] private float depletedCooldown = 3f;

    [Header("UI")]
    [SerializeField] private Image timeEnergyBar;
    [SerializeField] private TextMeshProUGUI timeEnergyText;
    [SerializeField] private GameObject rewindIndicator;

    private float currentTimeEnergy;
    private bool isRewinding = false;
    private bool isBlocked = false;
    private bool wasDepletedThisSession = false;

    // Shared cooldown with TimeStopManager
    private float abilityCooldownTimer = 0f;

    private Rewindable[] rewindables;

    private void Awake()
    {
        if (Instance == null) Instance = this;
        else { Destroy(gameObject); return; }
    }

    private void Start()
    {

```

```

currentTimeEnergy = maxTimeEnergy;
UpdateEnergyUI();

if (rewindIndicator != null)
    rewindIndicator.SetActive(false);
}

private void Update()
{
    if (GameManager.Instance != null && !GameManager.Instance.isGameStarted) return;

    if (abilityCooldownTimer > 0)
        abilityCooldownTimer -= Time.deltaTime;

    HandleRewindInput();
    RegenerateEnergy();
    UpdateEnergyUI();
}

private void HandleRewindInput()
{
    if (isBlocked) return;
    if (TimeStopManager.Instance != null && TimeStopManager.Instance.IsTimeStopped) return;
    if (abilityCooldownTimer > 0) return;

    if (Keyboard.current.rKey.isPressed && currentTimeEnergy > 0 && !isRewinding)
    {
        StartRewind();
    }

    if (isRewinding)
    {
        if (!Keyboard.current.rKey.isPressed || currentTimeEnergy <= 0)
        {
            if (currentTimeEnergy <= 0)
                wasDepletedThisSession = true;

            StopRewind();
        }
    }
}

```

```

    }
    else
    {
        currentTimeEnergy -= energyCostPerSecond * Time.deltaTime;
        currentTimeEnergy = Mathf.Max(0, currentTimeEnergy);
    }
}

private void RegenerateEnergy()
{
    if (isRewinding) return;
    if (TimeStopManager.Instance != null && TimeStopManager.Instance.IsTimeStopped) return;

    if (currentTimeEnergy < maxTimeEnergy)
    {
        currentTimeEnergy += energyRegenPerSecond * Time.deltaTime;
        currentTimeEnergy = Mathf.Min(currentTimeEnergy, maxTimeEnergy);
    }
}

private void StartRewind()
{
    isRewinding = true;
    wasDepletedThisSession = false;
    TimeEffectsVFX.Instance?.SetRewindEffect(true);

    rewindables = FindObjectsByType<Rewindable>(FindObjectsSortMode.None);
    foreach (Rewindable r in rewindables)
        r.StartRewind();

    // Freeze enemies that have no rewind history
    foreach (GameObject enemy in GameObject.FindGameObjectsWithTag("Enemy"))
    {
        Rewindable r = enemy.GetComponent<Rewindable>();
        if (r == null || r.SnapshotCount == 0)
        {
            EnemyAI ai = enemy.GetComponent<EnemyAI>();

```

```

        if (ai != null) ai.enabled = false;
    }
}

if (rewindIndicator != null)
    rewindIndicator.SetActive(true);
}

private void StopRewind()
{
    isRewinding = false;
    TimeEffectsVFX.Instance?.SetRewindEffect(false);

    if (rewindables != null)
        foreach (Rewindable r in rewindables)
            if (r != null) r.StopRewind();

    foreach (GameObject enemy in GameObject.FindGameObjectsWithTag("Enemy"))
    {
        EnemyAI ai = enemy.GetComponent<EnemyAI>();
        if (ai != null) ai.enabled = true;
    }

    if (rewindIndicator != null)
        rewindIndicator.SetActive(false);

    abilityCooldownTimer = wasDepletedThisSession ? depletedCooldown : switchCooldown;
    TimeStopManager.Instance?.SetExternalCooldown(abilityCooldownTimer);
}

public void SetExternalCooldown(float duration)
{
    abilityCooldownTimer = duration;
}

public bool IsOnCooldown => abilityCooldownTimer > 0;

public void SetBlocked(bool blocked) => isBlocked = blocked;

```

```

private void UpdateEnergyUI()
{
    if (timeEnergyBar != null)
        timeEnergyBar.fillAmount = currentTimeEnergy / maxTimeEnergy;

    if (timeEnergyText != null)
        timeEnergyText.text = $" {Mathf.RoundToInt(currentTimeEnergy)}";
}

public void ConsumeEnergy(float amount)
{
    currentTimeEnergy -= amount;
    currentTimeEnergy = Mathf.Max(0, currentTimeEnergy);
}

public void AddMaxEnergyBonus(float amount)
{
    maxTimeEnergy += amount;
    currentTimeEnergy = Mathf.Min(currentTimeEnergy + amount, maxTimeEnergy);
    UpdateEnergyUI();

    Rewindable[] allRewindables = FindObjectsByType<Rewindable>(FindObjectsSortMode.None);
    foreach (Rewindable r in allRewindables)
        r.UpdateMaxRecordTime(GetMaxRewindDuration());
}

public float GetRewindDuration() => currentTimeEnergy / energyCostPerSecond;
public float GetMaxRewindDuration() => maxTimeEnergy / energyCostPerSecond;
public bool IsRewinding => isRewinding;
public float EnergyPercent => currentTimeEnergy / maxTimeEnergy;
public float GetCurrentEnergy() => currentTimeEnergy;
}

```