КВАЛІФІКАЦІЙНА РОБОТА

КРФМБ.122.042.053.2026.ПЗ

ЗАКРЕНИЧНОГО ОЛЕКСАНДРА ОЛЕКСАНДРОВИЧА

ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ

Відділення інженерної інфраструктури та комп'ютерних наук

Циклова комісія спеціальності «Комп'ютерні науки»

Реєстраційний № _____
Дата реєстрації _____

РОЗРОБКА ГРИ ЖАНРУ SURVIVOR-LIKE НА UNITY

Пояснювальна записка
до кваліфікаційної роботи
здобувача фахової передвищої освіти
галузі знань 12 Інформаційні технології
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми Комп'ютерні науки
кваліфікації фахового молодшого бакалавра з комп'ютерних наук
групи П-42
ЗАКРЕНИЧНОГО Олександра Олександровича

Керівник:
МОЖАРОВСЬКА Людмила Володимирівна

2026

ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ

Відділення інженерної інфраструктури та комп'ютерних наук
Циклова комісія спеціальності «Комп'ютерні науки»
Галузь знань 12 Інформаційні технології
Спеціальність 122 Комп'ютерні науки
Освітньо-професійна програма Комп'ютерні науки
Кваліфікація фаховий молодший бакалавр з комп'ютерних наук

ЗАТВЕРДЖУЮ
Голова циклової комісії
_____Сергій МОЖАРОВСЬКИЙ
2 березня 2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ОСВІТИ

ЗАКРЕНИЧНОМУ Олександру Олександровичу

1. Тема роботи: **Розробка гри жанру Survivor-like на UNITY**

Керівник роботи: **МОЖАРОВСЬКА Людмила Володимирівна**
Затверджена наказом коледжу від **10 березня 2026 року, №51-н**
2. Строк подання здобувачем освіти роботи: **29 травня 2026 року**
3. Вихідні дані до роботи: Процес проєктування, моделювання та програмної реалізації мовою C# на рушії Unity двовимірної гри жанру Survivor-like у стилі готичного піксель-арту, включаючи розробку систем автоматичного бою, генерації хвиль ворогів, Roguelike-покращень та оптимізації рендерингу великої кількості об'єктів.
4. Зміст розрахунково-пояснювальної записки (перлік питань, які потрібно розробити) огляд та аналіз технологій розробки програмного продукту, проєктування та розробка програмного продукту, керівництво використання програмного продукту
5. Перелік графічного матеріалу (з точним зазаначенням обовязкових крелень):

6. Консультанти розділів роботи:

Розділ	Консультант	Дата, підпис	
		Завдання видала	Завдання прийняла
1	Людмила МОЖАРОВСЬКА	20.03.2026	10.04.2026
2	Людмила МОЖАРОВСЬКА	10.04.2026	24.04.2026
3	Людмила МОЖАРОВСЬКА	24.04.2026	01.05.2026

Дата видачі завдання: **16 березня 2026 року**

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Примітка
1	Детермінування мети та цілей роботи, встановлення об’єкта та предмета дослідження, визначення завдань розробки програмного забезпечення	20.03.2026	Виконано
2	Окреслення проєкту розробки	27.03.2026	Виконано
3	Формулювання технічного завдання та огляд літератури	03.04.2026	Виконано
4	Проєктування структури	10.04.2026	Виконано
5	Створення коду, його оптимізація та відлагождення	17.04.2026	Виконано
6	Тестування та розгортання системи	24.04.2026	Виконано
7	Обробка результатів роботи, написання та оформлення ПЗ	01.05.2026	Виконано

Здобувач освіти _____ Олександр ЗАКРЕНИЧНИЙ

Керівник роботи _____ Людмила МОЖАРОВСЬКА

РЕФЕРАТ

Записка: 102 стор., 25 рис., 1 табл., 1 додаток, 20 джерел.

ГЕЙМДИЗАЙН, ГОТИЧНИЙ ПІКСЕЛЬ-АРТ, ДВОВИМІРНА ГРА, ІГРОВІ МЕХАНІКИ, ООП, ПРОГРАМНА РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ПЗ, ACTION ROGUELIKE, C#, GUI, SURVIVOR-LIKE, UI/UX, UNITY.

Об’єкт дослідження: процес проєктування та програмування двовимірної гри жанру Survivor-like з художнім оформленням у стилі готичного піксель-арту.

Мета роботи: проєктування та програмна реалізація двовимірної гри жанру Survivor-like з використанням механік автобою, прогресії та художнього оформлення у стилі готичного піксель-арту.

Методи дослідження:

- розгляд сучасних підходів до розробки відеоігор;
- дослідження особливостей жанру Survivor-like;
- моделювання ігрових механік;
- проєктування архітектури програмного продукту;
- розробка та інтеграція 2D-графіки;
- функціональне тестування ПЗ.

Результати: розроблено та протестовано інтерактивну 2D-гру Necrocide у жанрі Survivor-like з реалізацією системи автобою, спавну хвиль ворогів, випадкових покращень персонажа та художнього оформлення у стилі готичного піксель-арту.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 4
ВСТУП 6
1 ОГЛЯД ТА АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ ПРОГРАМНОГО ПРОДУКТУ 9
1.1 Постановка задачі розробки програмного продукту 9
1.2 Аналіз та порівняння подібних додатків 11
1.3 Вибір та обґрунтування технологій розробки програмного продукту ... 15
1.4 Висновки до розділу 1 17
2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ 18
2.1 Функціональні можливості та використання програмного продукту 18
2.2 Проєктування діаграм активностей, послідовностей та класів 20
2.3 Етапи розробки системи. Розробка алгоритму конвертації програмного продукту 23
2.4 Висновки до розділу 2 32
3 ОПИС РОБОТИ ПРОГРАМНОГО ДОДАТКУ, ЙОГО ВСТАНОВЛЕННЯ, КОРИСТУВАННЯ ТА ТЕСТУВАННЯ 34
3.1 Керівництво встановлення програмного продукту 34
3.2 Керівництво користування програмним продуктом 36
3.3 Тестування роботи програмного продукту 41
3.4 Висновки до розділу 3 43
ВИСНОВКИ 44
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ 46
ДОДАТОК А Вихідний код гри 48

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПК (Персональний комп’ютер) – індивідуальна обчислювальна станція, що використовується для автоматизації робочих процесів, навчання, програмування чи мультимедійних розваг користувача.

ПЗ (Програмне Забезпечення) – цілісний комплекс інструкцій, цифрових кодів, скриптових сценаріїв та бібліотек, які реалізують виконання цільових задач обчислювальною системою.

ОС (Операційна Система) – фундаментальний пакет програмних інструментів, що здійснює пряме адміністрування апаратних ресурсів пристрою та формує екосистему для запуску прикладного софту.

ТЗ (Технічне Завдання) – вихідний нормативний документ, який детально описує функціональну архітектуру, ліміти, мету та базові критерії якості розроблюваного IT-продукту.

UI (User Interface) – система інтерактивних візуальних компонентів (клавіш, меню, панелей, індикаторів), за допомогою яких людина безпосередньо взаємодіє з додатком.

UX (User Experience) – суб'єктивне сприйняття, рівень ергономіки, комфорту та загальних вражень, які формуються у користувача під час оперування інтерфейсною частиною продукту.

QA (Quality Assurance) – планомірний процес контролю та тестування, орієнтований на перевірку працездатності ПЗ, локалізацію дефектів коду (багів) та оцінку загальної технічної стабільності.

NPC (Non-Player Character) – керований алгоритмами штучного інтелекту або фіксованими скриптами внутрішньоігровий юніт, з яким взаємодіє користувач.

FPS (Frames Per Second) – кількісний показник генерації графічних кадрів за одиницю часу (секунду), що визначає ступінь плавності візуального ряду та продуктивність обчислень.

IDE (Integrated Development Environment) – комплексне програмне середовище інженера, що консолідує текстовий редактор, засоби компіляції та модулі для пошуку й усунення дефектів у коді.

API (Application Programming Interface) – архітектурний інтерфейс взаємодії, що надає набір готових функцій та структур для обміну даними між відокремленими компонентами софту.

Pixel Art – стилістичний напрям двовимірного цифрового мистецтва, де формування зображень відбувається через ручне попіксельне малювання з акцентом на дискретну растрову естетику.

Sprite (Спрайт) – ізольований двовимірний графічний об'єкт (статичний малюнок або анімаційний лист текстури), що виступає окремим елементом сцени чи персонажем.

Game Engine (ігровий рушій) – базове програмне ядро (середовище), що оптимізує та автоматизує створення відеоігор шляхом централізованого керування рендерингом, фізикою твердих тіл та звуковими потоками.

2D (Two-Dimensional) – двовимірна координатна система відображення графіки, параметри простору якої обмежуються вимірами ширини та висоти об'єктів.

ВСТУП

Актуальність теми. Індустрія геймдеву на сьогодні є однією з найбільш динамічних інноваційних гілок у сфері комп'ютерних технологій, що гармонійно поєднує інженерію програмного забезпечення та сучасне візуальне мистецтво. Відеоігри давно перестали бути виключно засобом розваги, перетворившись на вагомий соціокультурний та технологічний феномен, який за рівнем впливу на суспільство стоїть в одному ряду з кінематографом та літературою. Як профільні дослідники, так і активні користувачі констатують стрімке масштабування грального ринку, його гнучку адаптацію до мінливих запитів аудиторії та інтеграцію передових практик розробки ПЗ у найрізноманітніші жанрові ніші.

Особливий сплеск популярності останніми роками демонструє жанр Survivor-like (справжнім каталізатором якого став комерційний успіх проєктів на кшталт Vampire Survivors), де ключовий фокус зміщено на виживання в умовах безперервного тиску хвиль ворогів, автоматизацію бойових дій та варіативність синергетичних ефектів. Специфіка цього напряму полягає у вдалому поєднанні механік Action Roguelike, концепції поступового масштабування складності, еволюції характеристик персонажа під час забігу, а також протистояння з колосальною кількістю супротивників, що формує високу реіграбельність та утримує увагу аудиторії.

Додатково актуальність цієї праці підтверджується стрімким розвитком інді-сегменту та незалежних студій, для яких рамки Survivor-like стали чудовим полем для експериментів. Сучасний досвід невеликих команд доводить можливість створення глибокого, захопливого та фінансово вигідного продукту з автентичним візуальним стилем та нетривіальними алгоритмами прогресії навіть за обмежених ресурсів. У наукових розвідках щодо еволюції ігрових трендів зазначається, що саме такі проєкти є ідеальним полігоном для тестування нестандартних художніх концепцій, похмурої атмосфери та нових підходів до дизайну ігрових сесій.

Водночас створення інтерактивного застосунку такого типу вимагає ретельного проєктування архітектури, налагодження стійких зв'язків між програмними модулями, оптимізації обчислювальних ресурсів під час

рендерингу сотень об'єктів та менеджменту спавну супротивників. Наявні методичні джерела та практичні дослідження в інженерії мультимедійного ПЗ акцентують на технологічних викликах розробки подібних систем, високих стандартах оптимізації, вимогах до графічного та звукового контенту, а також забезпеченні стабільної якості (QA) та позитивного користувацького досвіду (UX).

Метою роботи є реалізація повного життєвого циклу розробки та безпосереднє програмування двовимірної гри жанру Survivor-like із впровадженням класичних жанрових механік, алгоритмів випадкового вибору модифікаторів, систем автобою та стилізованого графічного оформлення під готичний піксель-арт.

Об’єкт дослідження – процес архітектурного проєктування та програмної реалізації двовимірної комп'ютерної гри жанру Survivor-like у похмурій готичній стилістиці.

Предмет дослідження – алгоритми моделювання ігрового балансу, методи процедурної генерації супротивників та інструментальні засоби створення ПЗ, зокрема системи автоатаки, обробки колізій та оптимізації графіки.

Методи дослідження:

- аналіз сучасних тенденцій та інженерних рішень у межах обраного ігрового жанру;
- дослідження архітектурних особливостей успішних Survivor-like проєктів;
- обґрунтування вибору технологічного стеку та інструментів розробки;
- застосування принципів об'єктно-орієнтованого програмування (ООП);
- проєктування та модульна побудова програмної архітектури застосунку;
- алгоритмічне моделювання ключових механік та циклів прогресії;
- комплексне функціональне тестування фінального програмного продукту.

Структура кваліфікаційної роботи. Дана праця складається зі вступної

частини, 3-х взаємопов'язаних розділів основної текстової частини, висновків до кожного з них, загальних підсумків за результатами дослідження, бібліографічного списку, переліку прийнятих термінів і термінологічних скорочень, а також комплексу додатків. У пояснювальній записці детально вивчено теоретичний базис жанру, формалізовано математичні та логічні моделі механік, описано етапи написання програмного коду гри Necrocide, інтеграції візуальних активів, а також наведено аналітичні результати верифікації та тестування готового застосунку.

1 ОГЛЯД ТА АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ ПРОГРАМНОГО ПРОДУКТУ

1.1 Постановка задачі розробки програмного продукту

Паралельно зі стрімким розширенням глобальної ігрової аудиторії спостерігається пропорційне зростання кількості розробників, що загострює конкуренцію та змушує шукати унікальні рішення для залучення користувачів. Попри те, що жанр Survivor-like (Action Roguelike), який базується на автоматизації бойових дій, виживанні проти нескінченних хвиль супротивників та варіативній системі покращень, сформувався порівняно нещодавно, сьогодні він переживає пік своєї затребуваності. Проте проектування таких застосунків є складним інженерним викликом, що вимагає системного підходу до архітектури, точного математичного балансування механік та утримання високої планки оптимізації графічної підсистеми [8]. Саме тому ключовим завданням є створення програмного забезпечення, здатного підтримувати високу інтерактивність, стабільну частоту кадрів при рендерингу сотень об'єктів та автентичну візуальну стилістику [9].

Головна мета даного дослідження полягає у проектуванні та програмній реалізації двовимірного ігрового продукту «Necrocide» у жанрі Survivor-like з візуальним оформленням у стилі похмурого готичного піксель-арту. Основна геймплейна концепція проекту передбачає керування магом-некромантом, який протистоїть масовим навалам ворогів, використовує заклинання з автоматичним наведенням, збирає ресурси для моментального підвищення характеристик під час забігу та розблокує постійні апгрейди [10]. Ігровий простір планується реалізувати у вигляді просторої арени типу «занедбане кладовище» з динамічним ускладненням зон, а решта функціоналу базуватиметься на класичних канонах аренних роглайків.

Для успішного виконання накреслених завдань життєвий цикл розробки комп'ютерної гри має містити такі послідовні етапи [8]:

- аналіз ключових механік та архітектурних рішень у сучасних Survivor-like проектах;
- дослідження наявного інструментарію та рушіїв для створення двовимірних ігор;
- формування переліку функціональних та технічних вимог до програмного продукту;
- проектування модульної структури застосунку та логіки взаємодії між підсистемами;
- програмна реалізація базового контролера переміщення та позиціонування гравця;
- розробка алгоритмів автоматичного ведення вогню та розрахунку радіуса ураження;
- проектування штучного інтелекту хвиль супротивників та поведінки фінального боса;
- створення інтерфейсу користувача (GUI) та елементів динамічної мапи;
- інтеграція графічного контенту, витриманого в єдиній концепції готичного піксель-арту;
- проведення всебічного контролю якості (QA), включаючи пошук помилок та оптимізацію коду.

У межах програмної реалізації першочергову увагу необхідно приділити стабільності обробки колізій, менеджменту динамічного створення та видалення об'єктів (системі спавну), а також базовим елементам інтерфейсу керування. Важливе місце в роботі посідає графічний складник, оскільки дотримання заявленої теми вимагає автентичного відображення готичного піксель-арту, що безпосередньо впливає на занурення користувача в атмосферу світу. Крім того, критично важливими критеріями проектування є продуктивність програми під високим обчислювальним навантаженням, раціональне споживання ресурсів пристрою, коректна фізика взаємодії та загальний комфорт ігрового процесу.

Задля ефективного вирішення поставленої інженерної задачі початковим кроком є детальний аналіз сучасних методологій розробки, готових рушіїв та

засобів оптимізації 2D-графіки, що дозволить мінімізувати архітектурні ризики та структурувати подальший процес написання коду.

1.2 Аналіз та порівняння подібних додатків

На етапі проєктування нового програмного забезпечення одним із базових та найбільш критичних кроків є дослідження наявних на ринку рішень від інших розробників. Результати вивчення таких застосунків дозволяють чітко аргументувати доцільність випуску нового продукту, виявити сильні та слабкі сторони чинних систем і сформувати фінальний пул технічних вимог. Зокрема, детальний розбір успішних Survivor-like проєктів допомагає визначити найбільш життєздатні методи програмування ключових геймплейних циклів, архітектури генерації хвиль, графічного виконання та взаємодії гравця з інтерфейсом. Крім того, компаративний аналіз аналогів відкриває можливість запозичити або модифікувати вдалі інженерні рішення під час написання власного ПЗ.

Беззаперечним флагманом та найбільш відомим представником сучасних ігор у жанрі Survivor-like є проєкт Vampire Survivors, який вирізняється колосальною кількістю одночасних об'єктів на екрані, похмурою атмосферою, мінімалістичним, але затягуючим ігроладом та варіативною системою еволюції зброї. Особливий акцент автори зробили на оптимізації обчислювальних процесів та динаміці ігрової сесії, що створює високу щільність подій під час проходження арени. Приклад оформлення стартового екрана та загальної стилістики візуалу гри Vampire Survivors продемонстровано на рисунку 1.1.

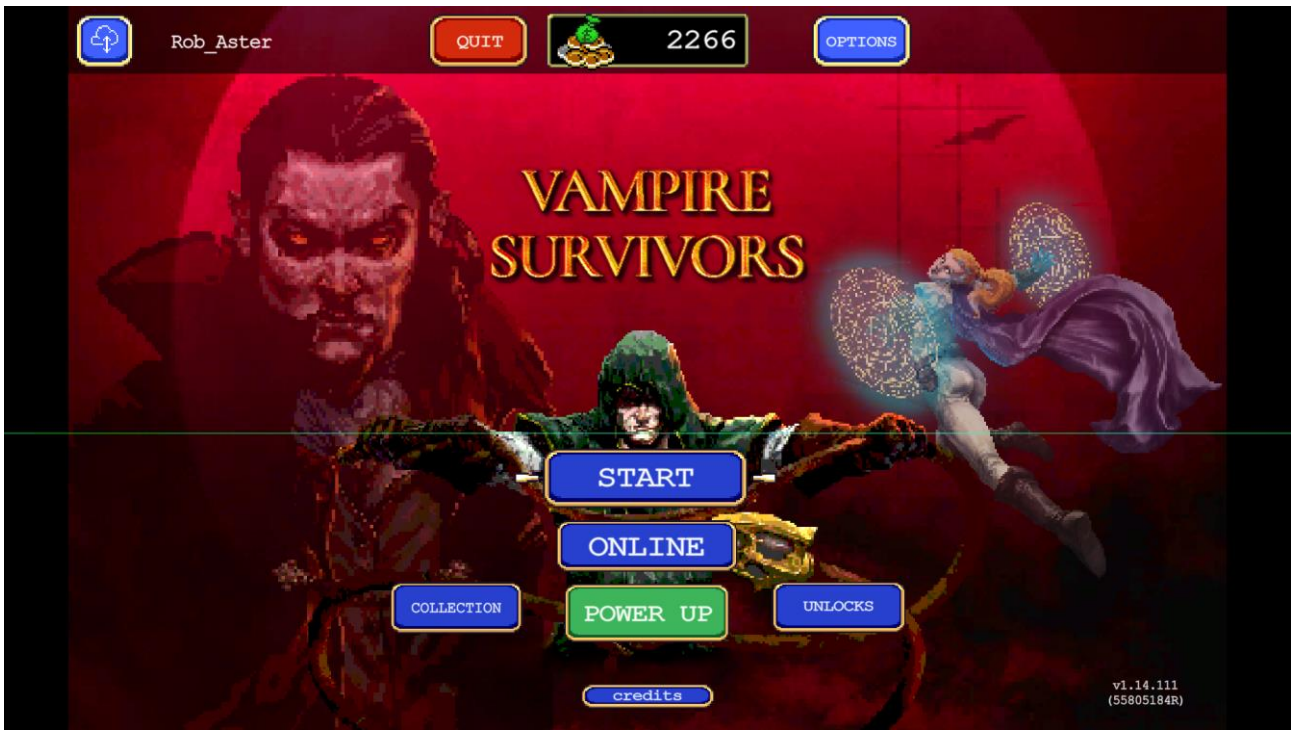

Рисунок 1.1 Головне меню гри Vampire Survivors

Яскравим прикладом якісної інженерної роботи та комерційного успіху в межах цього напряму є гра Brotato, де реалізовано синергію аренних шутерів, замкнених локацій та глибокої бойової системи з акцентом на збирання білдів. Brotato наочно демонструє високий функціональний потенціал створення варіативних ігрових механік за допомогою плоскої 2D-графіки та чітко прорахованих математичних моделей прокачки. Гра суттєво виділяється серед конкурентів гнучким налаштуванням характеристик персонажа та високою динамікою бойових зон (рисунок 1.2).

Рисунок 1.2 Битва з хвилею ворогів у грі Brotato

Окрім аналізу ігрових механік безпосередньо, дуже важливо правильно налаштувати особливості GUI та організацію взаємодії з користувачем. Наприклад, як показано на рисунку 1.3.

Рисунок 1.3 Екран вибору зброї у Vampire Survivors

Для сучасних Survivor-like ігор є базовою нормою мати реалістичний інтерфейс, який є мінімалістичним і не перевантажує ані меню, ані ігровий екран

великою кількістю елементів. Це дозволяє візуально збільшити бойову арену та зосередити увагу гравця на маневруванні серед натовпу ворогів, контролі радіуса атак та загальній динаміці забігу. Яскраві приклади реалізації ігрових інтерфейсів наведено на рисунках 1.3 та 1.4.

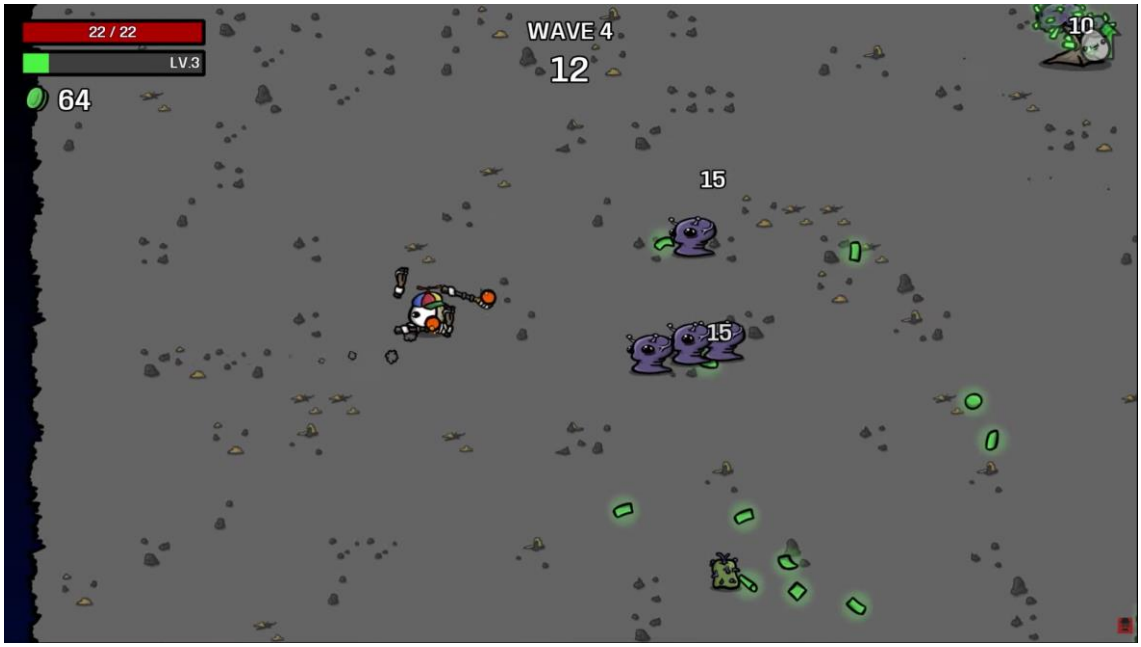

Рисунок 1.4 Бойовий інтерфейс користувача (HUD) у грі Brotato

Також під час аналізу аналогів було детально розглядено візуальні та художні особливості піксель-арту, які нерідко використовуються в незалежних ігрових проєктах. Користування лише обмеженою палітрою кольорів, великих контрастів у освітленні та фонів з різними рівнями деталізації дозволяє створити індивідуальний візуальний стиль навіть у рамках обмеженої графіки, і навіть у 2D форматі. Для цього слугував, у числі інших, розгляд стилізації гри Boneraiser Minions, приклади локації та боса якої наведено на рисунку 1.5.

Рисунок 1.5 – Стилізація гри Boneraiser Minions

Таким чином, розгляд подібних проєктів показав, що більшість сучасних успішних Survivor-like ігор мають такі спільні риси: інтенсивний аренний геймплей; систему розблокування випадкових модифікаторів героя, виокремлене індивідуальне візуальне оформлення та комбінацію виживання в натовпі ворогів з елементами швидкої динаміки та мета-прогресу. Отримані результати аналізу були повністю чи частково враховані під час формування концепції та проєктування гри Necrocide.

1.3 Вибір та обґрунтування технологій розробки програмного продукту

Життєвий цикл створення будь-яких мультимедійних додатків, включаючи індустрію геймдеву, базується на впровадженні інструментарію та засобів розробки, що здатні гарантувати стабільну працездатність системи, точне відтворення закладеного функціоналу та ергономічність ігроладу з високим рівнем UX. Обґрунтування технологічного стеку виступає базовим фундаментом архітектурного проєктування ПЗ, оскільки обране рішення безпосередньо визначає швидкість обробки даних, якість візуалізації, загальну оптимізацію

продукту, а також гнучкість та оперативність самого написання коду.

Для технічного втілення проєкту **Necrocide** було залучено спеціалізовану платформу **Unity**, яка наразі займає позицію одного з лідерів серед сучасних ігрових рушіїв для створення інтерактивного софту у 2D та 3D просторах [11-13]. Ключовими плюсами цього середовища є розвинений набір інструментів для плоскої графіки, інтегровані системи обробки фізичних колізій, модулі керування спрайтами та анімаційними треками, а також велика база готових бібліотек для програмування логіки. До того ж Unity пропонує розробнику зручний функціонал миттєвого налагодження та тестування коду безпосередньо у вікні редактора з можливістю подальшої кросплатформної компіляції під різні ОС.

Застосування середовища Unity є повністю раціональним, зважаючи на його орієнтацію на компонентну архітектуру побудови програмних об'єктів [12]. Такий інженерний підхід дозволяє декомпозувати загальну логіку застосунку на відокремлені, слабопов'язані класи, що суттєво полегшує подальший рефакторинг та масштабування коду за потреби додавання нових фіч. Для створення алгоритмів переміщення протагоніста, механіки автоматичного наведення атак, тригерів спавну ворогів та їхніх візуальних станів доцільно застосувати базові модулі рушія, зокрема компоненти фізичної взаємодії та систему Animator Controller.

Головним інструментом програмування було обрано об'єктно-орієнтовану мову **C#**, яка має бездоганну інтеграцію з Unity, гнучкий синтаксис та високі показники швидкодії під час виконання складних обчислень [14-15]. Вагомим додатковим фактором на користь цього вибору є наявність розвиненої екосистеми, детальної технічної документації та масштабної спільноти розробників на профільних ресурсах, що значно прискорює пошук рішень та етап QA.

Процес написання, структурування та ревізії вихідного коду здійснювався в інтегрованому середовищі розробки **Visual Studio** [16]. Це дозволило оптимізувати робочий час розробника, автоматизувати рутинні операції та

підвищити ефективність пошуку програмних помилок на стадії налагодження.

З огляду на те, що візуальний концепт гри передбачає застосування стилістики похмурого готичного піксель-арту, окрему увагу було приділено підбору спеціалізованого софту для створення та підготовки растрових об'єктів. Основним робочим інструментом для малювання графічних ассетів став редактор Aseprite, розроблений безпосередньо для попіксельного дизайну та покадрової анімації.

Для організації та проведення контролю якості (QA) розробленого софту застосовано методологію функціонального тестування, спрямовану на перевірку відповідності ключових механік базовим критеріям обраного жанру.

На основі вищесказаного можна стверджувати, що затверджений комплекс програмних засобів та технологій формує надійну основу для реалізації Survivor-like гри з виразною атмосферою, стабільною архітектурою модулів та високим потенціалом для подальшого розвитку. Використання екосистеми Unity, мови C# та супутніх графічних редакторів є повністю збалансованим та виправданим кроком для успішного релізу проєкту Necrocide [17].

1.4 Висновки до розділу 1

У межах першого розділу було детально досліджено специфіку відеоігор жанру Survivor-like, окреслено базові технічні вимоги до програмного продукту та сформовано загальну геймплейну концепцію проєкту **Necrocide**. Компаративний аналіз існуючих аналогів дозволив виділити найбільш раціональні інженерні підходи до побудови бойових систем, логіки арен та оптимізації графіки. Крім того, було аргументовано вибір програмного стеку та засобів реалізації, включно з використанням екосистеми Unity, об'єктно-орієнтованої мови C# та графічного інструментарію Aseprite. Затверджений комплекс технологій надає повний спектр можливостей для створення якісної інтерактивної 2D-гри з автентичним готичним піксель-арт дизайном.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1 Функціональні можливості та використання програмного продукту

Створюваний програмний продукт є аренним 2D Survivor-like проєктом у піксельному стилі з похмурими неоготичними елементами оточення. Ключовим користувачем системи виступає індивідуальний гравець, який здійснює позиціонування головного героя, виживає в замкненому просторі, протидіє масовим навалам супротивників, збирає ресурси й динамічно комбінує пасивні та активні модифікатори.

Функціональні можливості гри:

- Керування персонажем: користувач має змогу вільно переміщувати некроманта за допомогою осей координат (вгору, вниз, ліворуч, праворуч), маневрувати між натовпом, активувати короткочасний тактичний ривок (dash) після вибору відповідного треку покращень.
- Бойова система: головний герой атакує автоматично випущеними базовими згустками магії, а також використовує додаткові заклинання кругового ураження, виклик кістяних шипів та прокляття уповільнення (активація та потужність окремих типів зброї залежить від обраних під час сесії перків: Soul Orb, Bone Spikes або Death Aura).
- Система життєдіяльності персонажа: під час фізичного контакту з юнітами ворога або їхніми снарядами маг втрачає бали стійкості. У момент отримання шкоди спрацьовує фрейм тимчасової невразливості, що супроводжується графічним блиманням спрайту. Повна втрата здоров'я запускає алгоритм фіксації смерті, відображення екрана статистики та повернення до головного меню.
- Система рандомного вибору апгрейдів (ротація): накопичення досвіду, вибір випадкових карт модернізації, підвищення швидкості атак, розширення зони збору предметів та еволюція зброї $\rightarrow$ жанр Survivor-

like (ігровий прогрес та сила білду визначаються комбінуванням випадкових елементів).

- Взаємодія з супротивниками: кожен ворожий юніт наділений власним пулом HP, реагує на отримання шкоди зміною кольору, підлягає повній деструкції та генерує кристали досвіду або лікувальні сутності на місці своєї ліквідації.
- Поведінка юнітів: скелет-піхотинець рухається за вектором найкоротшої траєкторії до гравця, тоді як привид ігнорує перешкоди, відстежує поточні координати цілі та прискорюється при наближенні на критичну відстань.
- Битва з фінальним лідером хвиль: при досягненні певного часового таймера на полі з'являється унікальний бос (при цьому блокуються межі поточної локації, припиняється стандартний спавн рядових мобів, а тріумф над лідером приносить рідкісний артефакт). Бос функціонує в рамках кількох поведінкових патернів: кругової стрільби, ближнього тарана та призову елітних міньйонів.
- Комплексна обробка колізій та шкоди.

З наведеної діаграми варіантів використання на рисунку 2.1 чітко видно, що користувач володіє базовими функціями позиціонування та автоматичного ведення бою, тоді як значна частина геймплейних дій реалізована як розширення, що активуються під час вибору випадкових покращень у процесі забігу.

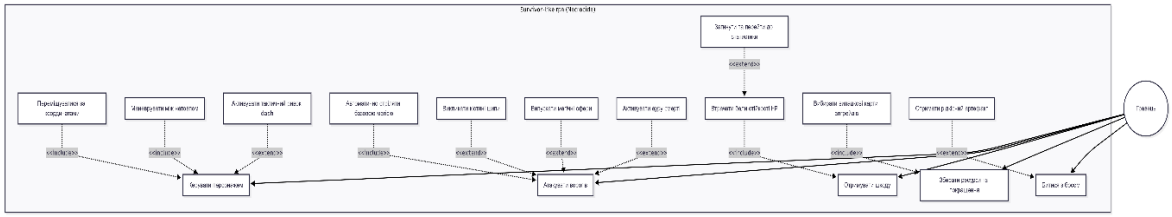

Рисунок 2.1 – Діаграма варіантів використання гри Necrocide

Сформована логіка повністю відповідає концептуальній структурі Action Roguelike застосунку, де інтеграція нових пасивних та активних умінь відкриває шляхи до побудови унікальних бойових стратегій та ефективного стримування агресивного середовища.

2.2 Проєктування діаграм активностей, послідовностей та класів

Діаграма активностей (рисунок 2.2) ілюструє базовий ігровий цикл (Core Loop) застосунку: обробку введення для переміщення мага, автоматичне функціонування зброї, алгоритми отримання пошкоджень, генерацію хвиль, підбирання кристалів досвіду та фінальний поєдинок із головним босом арени [18].

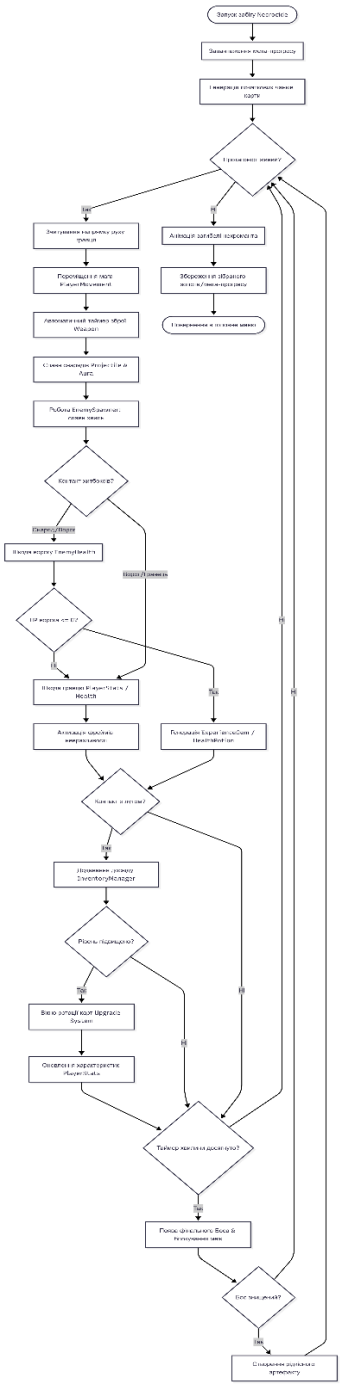

Рисунок 2.2 Діаграма активностей

Діаграма послідовностей (рисунок 2.3) деталізує динамічну взаємодію в часі між користувачем, модулями обробки інвентарю та характеристик, генератором супротивників, компонентами розрахунку колізій хитбоксів і менеджером мета-прогресії під час бойової сесії та левелапу.

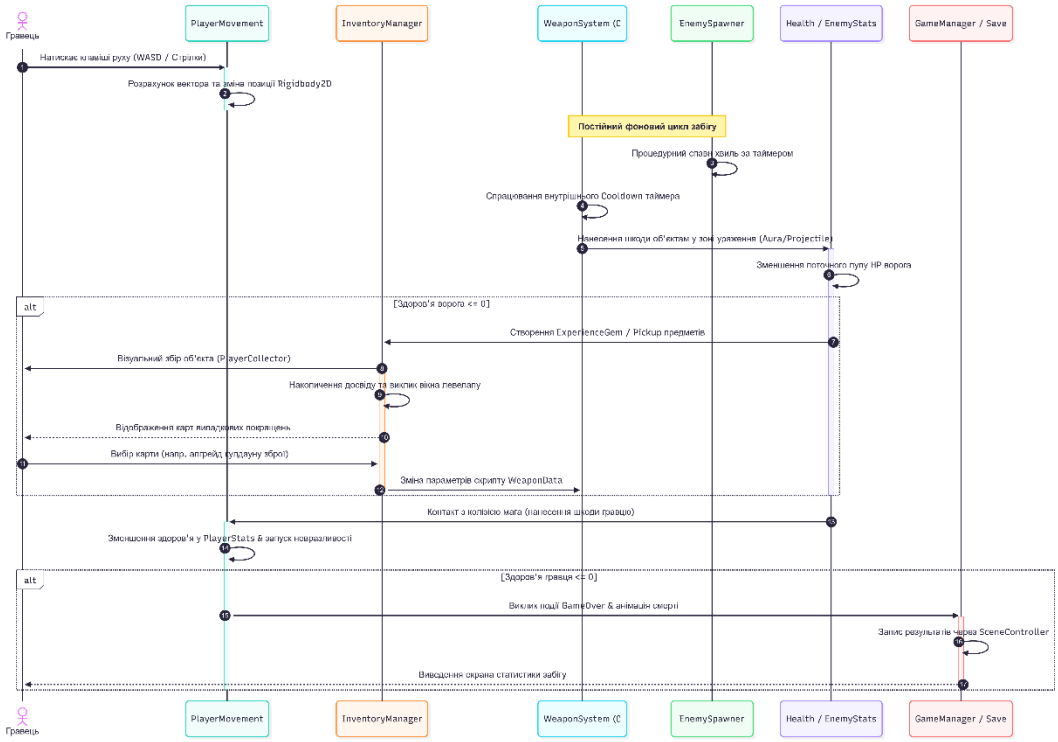

Рисунок 2.3 Діаграма послідовностей

У діаграмі компонентів (рисунок 2.4) наочно представлено архітектурний поділ програмного продукту на ізольовані функціональні пакети: ядро гравця (Player Core), систему штучного інтелекту ворогів (Enemy AI), менеджер випадкових генерацій мапи (Map Controller), інтерфейсний HUD та підсистему збереження даних.

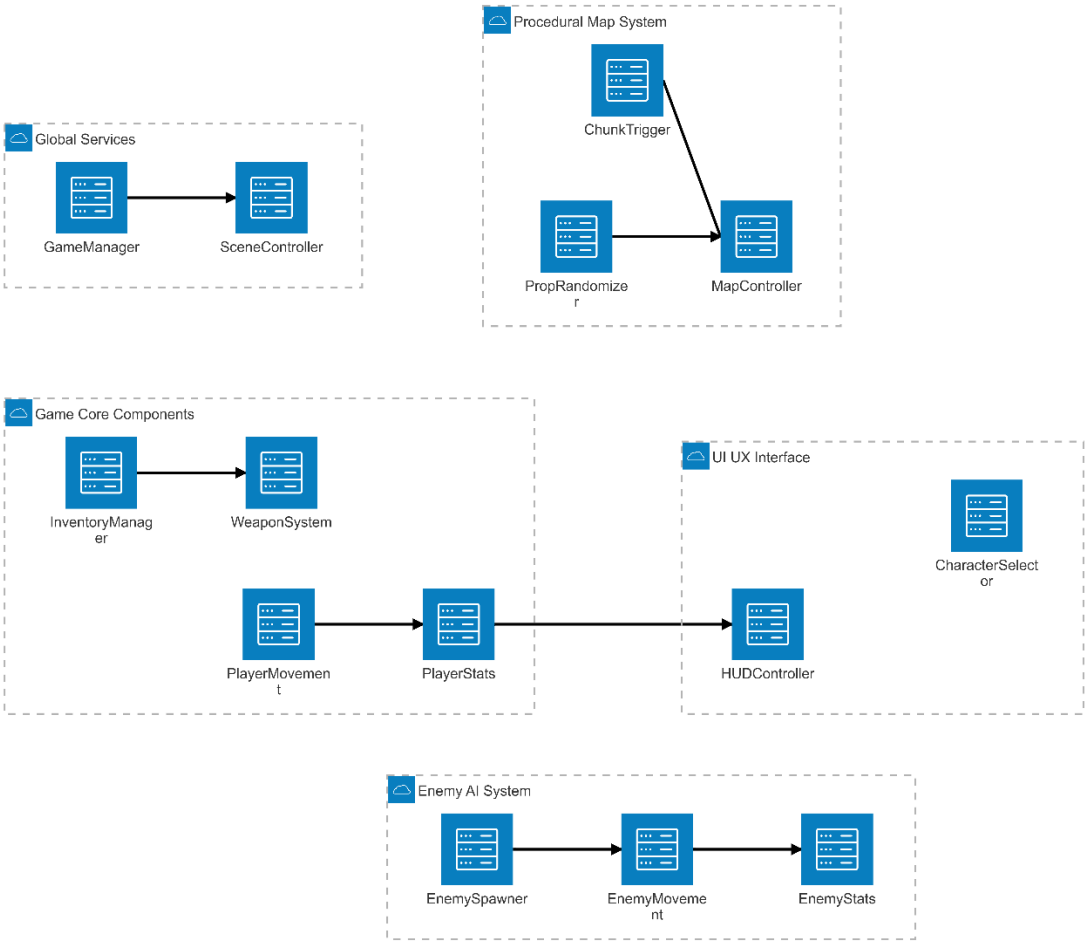

Рисунок 2.4 Діаграма компонентів гри

Діаграма класів (рисунок 2.5) відображає об’єктну структуру кодової бази, описуючи ключові класи, їхні внутрішні змінні, методи та типи наслідування чи асоціацій. Архітектурна схема спроєктована в точній відповідності до реалізованих C# скриптів: контролерів переміщення, менеджерів зброї, систем збору луту (*Pick-Ups*) та обробників динамічних чанків карти.

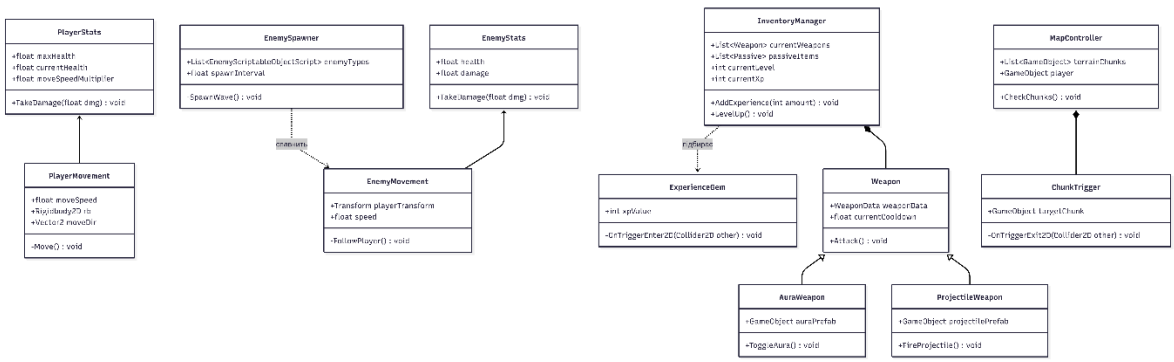

Рисунок 2.5 Діаграма класів

2.3 Етапи розробки системи. Розробка алгоритму конвертації програмного продукту

Створення інтерактивного ігрового додатка жанру Survivor-like з візуальним оформленням у стилі похмурого неоготичного піксель-арту здійснювалося як послідовний процес проєктування ігрової архітектури, що об'єднує програмні модулі, розробку графічних ассетів, алгоритми взаємодії об'єктів та інструменти контролю геймплейних циклів. Кожен робочий крок фокусувався на побудові окремого функціонального вузла та покроковому зведенні цілісного програмного забезпечення.

Першочерговим етапом стало комплексне дослідження специфіки Action Roguelike напряму та аналітичний огляд комерційних аналогів цього класу. Було сформовано базові параметри жанру:

- обмежений або замкнений простір арен;
- процедурна генерація та нескінченний спавн ворогів;
- накопичення досвіду та випадкова ротація перків під час левелапу;
- автоматизоване ведення вогню та контроль радіусів атак;
- балансування щільності хвиль супротивників;
- наявність елітних лідерів (босів) та розвинена мета-прогресія гравця.

Також було детально вивчено художній концепт ретро-стилістики, що диктує використання обмеженої 2D-палітри, затіненого простору, деталізованих спрайт-лістів та гнітючої атмосфери локацій.

Підсумком цього аналітичного кроку стало затвердження дизайн-документа, фіксація ключових геймплейних механік та візуального коду майбутнього інтерактивного софту.

До переліку функціональних вимог увійшли:

- позиціонування та переміщення протагоніста;
- розрахунок векторів руху та фізичних зсувів;
- автоматична бойова система (активні скіли);
- обробка параметрів життєдіяльності та міцності;

- інтерфейс випадкового вибору модифікаторів;
- логіка штучного інтелекту хвиль мобів;
- скрипти фазових битв із босами;
- збереження накопичених ресурсів та прогресу;
- динамічне оновлення елементів HUD;
- інтеграція звукових доріжок та аудіоефектів (SFX).

До категорії нефункциональних вимог належали:

- висока оптимізація при великій кількості об'єктів;
- кадрова плавність відтворення анімацій;
- мінімальний інпут-лаг при зчитуванні введення;
- модульна структура кодової бази;
- архітектурна гнучкість для подальшого масштабування;
- кросплатформна сумісність з архітектурою ПК.

Після деталізації вимог було спроєктовано загальну ієрархію модулів програмного продукту та сформовано склад ключових систем.

Наступним кроком стало формування загальної архітектури проєкту. Головним завданням виступала ізоляція логічних блоків у незалежні класи для полегшення кодингу, юніт-тестування та додавання майбутнього контенту.

Було архітектурно виділено такі підсистеми:

- модуль переміщення та фізики гравця;
- підсистема автоматичного оперування зброєю;
- блок моніторингу життєвих показників;
- менеджер інвентарю та левелапів;
- алгоритми поведінки штучного інтелекту ворогів;
- обробник подій появи босів;
- графічний інтерфейс користувача;
- звуковий мікшер та аудіоменеджер;
- сервіс серіалізації даних (збереження).

Для програмування кожної підсистеми було написано відповідні C# класи:

- PlayerMovement реалізує навігацію некроманта;
- InventoryManager та Weapon оперують білдом і боєм;
- PlayerStats і Health контролюють показники життєдіяльності;
- EnemyStats та EnemyMovement керують параметрами мобів;
- EnemySpawner відповідає за розрахунок хвиль ворогів;
- MapController та ChunkTrigger забезпечують безшовність карти.

На етапі програмування навігаційних алгоритмів було закладено фундамент контролю головного героя.

Було створено та налагоджено механізми:

- всеспрямованого руху по осях X та Y;
- швидкого ухилення від атак (ривок/dash);
- розрахунку дистанції зближення з ворогами;
- перевірки меж поточного чанку карти;
- інверсії спрайтів залежно від вектора руху;
- тригерів покадрової анімації;
- підбирання предметів магнітним радіусом.

Для розрахунку фізичного шару задіяно компоненти Rigidbody2D та систему колайдерів Unity. Особлива увага приділялася лінійній інтерполяції руху, чутливості керування та відсутності затримок між анімаційними станами.

Крім того, було впроваджено алгоритм відкидання (knockback) супротивників при ударах, що підвищує тактичну глибину бойових зіткнень.

Одним із центральних етапів стала побудова алгоритмів автоматизованого бою.

У структурі класів було реалізовано:

- спавн базових магічних снарядів (Projectile);
- кругову зону ураження (AuraWeapon);
- генерацію кістяних шипів під ворогами;
- уповільнюючі прокляття;
- таймери перезарядки зброї (Cooldown);

- часткові візуальні ефекти (VFX) та звуки заклинань.

Бойові алгоритми розроблені так, щоб нові типи зброї та пасивні предмети інтегрувалися випадковим чином під час проходження. Це дозволило втілити ключову для Survivor-like ігор систему синергії білдів.

Окремо було запрограмовано логіку розрахунку радіуса збору досвіду (PlayerCollector), який зчитує наближення об'єктів за допомогою тригерів.

Для протагоніста та юнітів орди створено диференційовані системи життєдіяльності.

Клас моніторингу стану мага включає:

- обчислення отриманих пошкоджень;
- фрейми імунітету до шкоди;
- візуальний відгук (миготіння матеріалу);
- обробку критичного стану (смерть);
- регенерацію за рахунок підібраних еліксирів;
- рендеринг слайдерів на екрані.

Після фіксації колізії з ворогом вмикається короткий таймер невразливості, що унеможливлює моментальне зчитування шкоди на кожному кадрі.

Для звичайних ворожих юнітів впроваджено:

- віднімання балів стійкості (HP);
- колірну індикацію влучання снаряда;
- деструкцію та повернення об'єкта в пул;
- розрахунок шансу випадіння досвіду (DropRateManager).

Для забезпечення прогресії всередині забігу реалізовано алгоритм підвищення рівня на основі досвіду.

Ця система оперує такими елементами:

- збір кристалів досвіду (ExperienceGem);
- калькуляція порогу наступного рівня;
- призупинення ігрового часу (Pause);
- випадкова вибірка карт із пулу WeaponData;

- модифікація пасивних параметрів (PlayerStats).

При виборі карти покращення система миттєво перезаписує змінні швидкості атак або шкоди. Цей механізм є ядром Survivor-like геймплею, оскільки він створює унікальність кожної ігрової сесії.

Для підвищення динаміки забігу було спроєктовано кілька класів штучного інтелекту ворогів із різними патернами руху:

- пряме переслідування цілі за найкоротшим вектором;
- обхідні траєкторії для літаючих юнітів;
- прискорення мобів при наближенні на певну дистанцію;
- завдання шкоди через перекриття хитбоксів;
- ускладнена поведінка елітних супротивників.

Окрему увагу приділено оптимізації поведінкових скриптів через обмеження частоти оновлення координат цілі (оптимізація методу Update), що гарантує стабільний FPS.

Поява фінального боса є кульмінаційною подією ігрової сесії.

Для цього етапу було запрограмовано:

- часовий тригер виклику боса;
- активацію статичних бар'єрів на краях екрана;
- комбіновані фази атак лідера;
- алгоритм призову міньйонів на арену;
- зміну фонової аудіодоріжки;
- генерацію фінального артефакту перемоги.

Скрипт боса динамічно змінює частоту використання заклинань залежно від залишку його здоров'я, що робить фінальну битву непередбачуваною.

Для оперативного інформування гравця було розроблено мінімалістичний HUD.

Інтерфейсна панель відображає:

- шкалу поточного та максимального здоров'я;
- індикатор прогресу до наступного рівня;

- іконки поточної зброї та рівень її прокачки;
- таймер тривалості поточного забігу.

Графічний HUD забезпечує безперервний зворотний зв'язок, дозволяючи користувачу моментально оцінювати ефективність поточного білду.

Паралельно було реалізовано систему мета-прогресу та збереження результатів.

Архітектура збереження фіксує:

- кількість зібраного за забіг золота;
- розблокованих персонажів;
- рівень постійних пасивних покращень;
- рекорди виживання за часом;
- стан глобальних тригерів досягнень;
- загальну статистику облікового запису.

Це дає можливість гравцеві витрачати накопичені ресурси в головному меню для постійного посилення стартових характеристик мага.

Для створення цілісної атмосфери виконано художнє та звукове наповнення в єдиній неоготичній стилістиці.

Було розроблено:

- анімовані спрайти некроманта;
- покадрові цикли руху та атак ворогів;
- безшовні текстури для підкладки карти;
- частки візуальних ефектів магії та вибухів;
- налаштування шарів сортування (Sorting Layers);
- фонові декорації та руйновані об'єкти (BreakableProps).

Додатково ігровий простір було насичено аудіокомпонентами:

- похмурим фоновим ембієнтом;
- звуковими ефектами заклинань;
- аудіовідгуком при кроках та ривках;
- звуками поглинання досвіду та отримання шкоди;

- ексклюзивним звуковим супроводом для бос-фази.

Структурована декомпозиція запланованих до інтеграції ігрових сутностей та їхніх взаємозв'язків відображена на концептуальній ER-подібній схемі, що наведена на рисунку 2.6.

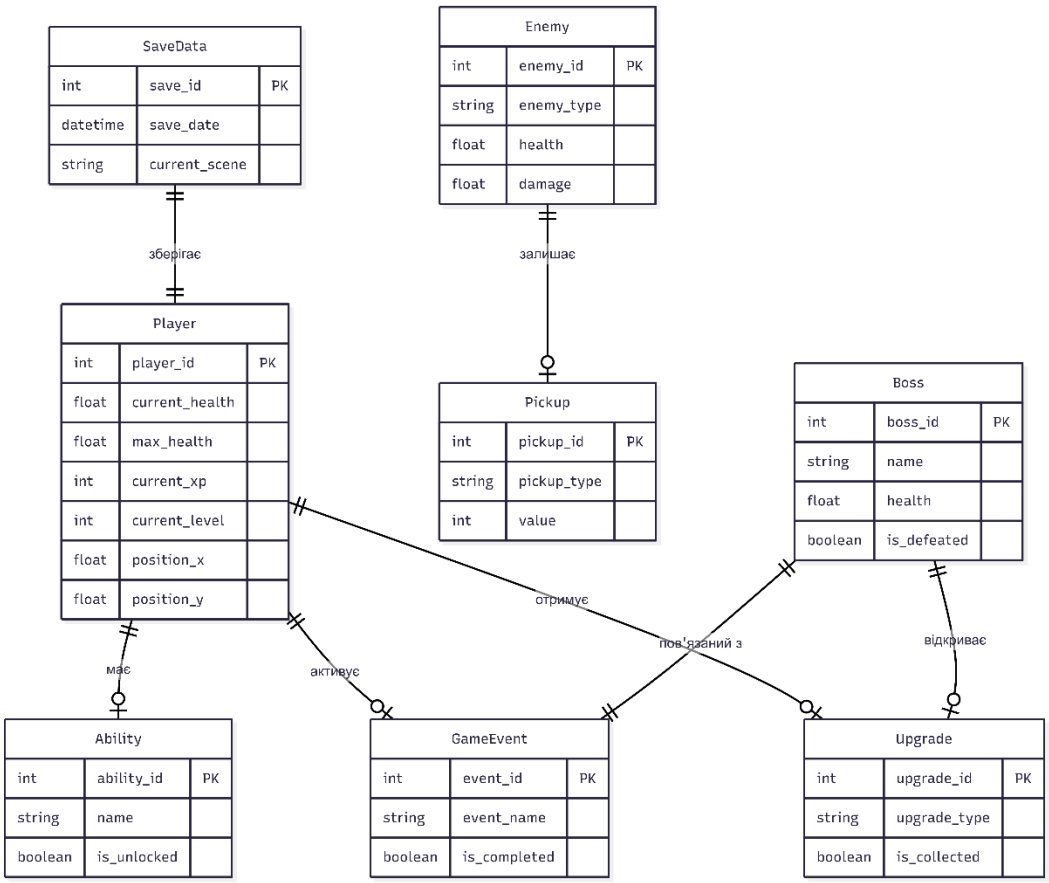

Рисунок 2.6 ER-подібна схема даних

Послідовність розгортання та алгоритм комерційної конвертації готового програмного продукту (ілюстрований за допомогою блок-схеми на рисунку 2.7) складається з таких кроків:

1. Консолідація та санація компонентів проєкту. Здійснюється ревізія наявних цифрових активів: чанків арен, кодової бази, елементів спрайт-лістів, звукових пресетів, систем часток та префабів. Проводиться повна деструкція налагоджувальних об'єктів сцени й детекція ізольованих посилань.

2. Валідація конфігурації ігрових просторів. У менеджері розгортання Unity формується черговість завантаження локацій. Стартове вікно вибору

персонажа, головне меню та основна ігрова арена заносяться до загального реєстру Build Settings.

3. Параметризація десктопної архітектури. Обирається фінальне середовище виконання (наприклад, ОС Windows). Задаються базові ліміти роздільної здатності дисплея, режими рендерингу (повноекранний/віконний), завантажується фірмова іконка застосунку та профілі графічної оптимізації.

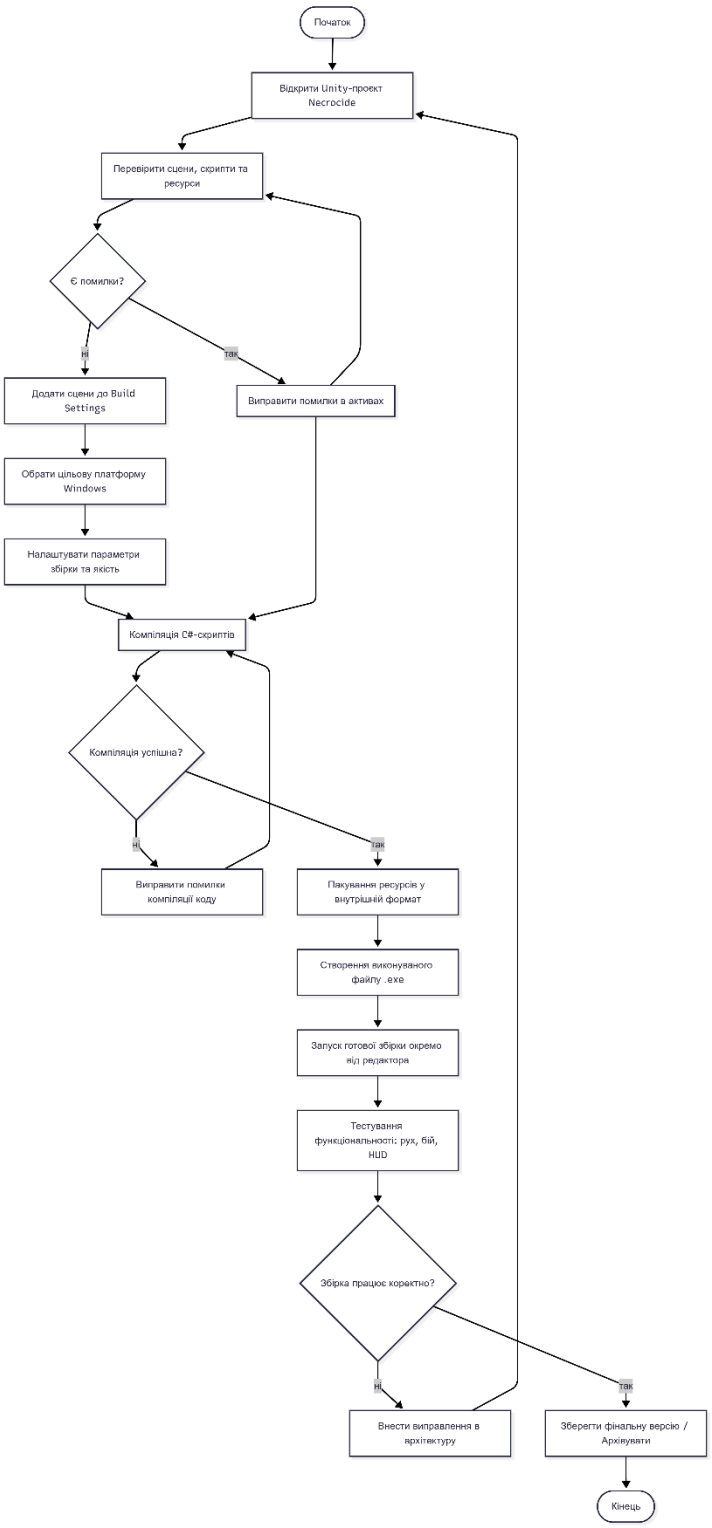

Рисунок 2.7 Блок-схема алгоритму конвертації

1. Детекція внутрішніх та зовнішніх залежностей. Автоматизовані інструменти середовища перевіряють цілісність ресурсного пакунка: скриптів автоматичної атаки, аудіодорожок, матеріалів, анімаційних контролерів та файлів локального збереження.

2. Трансляція та компіляція програмного коду. Модулі, написані мовою C#, трансформуються середовищем у високоефективний машинний код. На цьому етапі локалізуються помилки типізації, архітектурні конфлікти класів та неіснуючі неймспейси.

3. Агрегація та бінаризація ассетів. Рушій Unity здійснює стиснення й пакування текстурних атласів, аудіофайлів та системних конфігів у власні оптимізовані бінарні контейнери, призначені для десктопного запуску.

4. Генерація релізного дистрибутива. Після безпомилкового завершення фази компіляції середовище формує фінальний виконуваний файл з розширенням .exe та супровідну директорію зі зкомпільованими ресурсами гри.

5. Автономне верифікаційне тестування. Створений білд запускається ізольовано від редактора Unity. Напряму перевіряється коректність систем навігації, циклів автоатаки, динамічного спавну хвиль, оновлення HUD, механізмів підбору досвіду та логіки боса.

6. Кінцева дистрибуція застосунку. Після успішного проходження тестів папка з грою підлягає фінальному стисненню в архів або копіюється у виділений репозиторій для подальшого розгортання користувачем.

2.4 Висновки до розділу 2

У другому розділі кваліфікаційного проєкту було успішно здійснено архітектурне проєктування та програмну реалізацію двовимірної Action Roguelike гри Necrocide, виконаної в оригінальній стилістиці похмурого неоготичного піксель-арту. У межах програмного рішення розроблено чутливі алгоритми навігації протагоніста, автоматизовані бойові модулі, комплексну систему моніторингу показників життєдіяльності, дерево випадкових модифікаторів білду, штучний інтелект орди супротивників, скрипти фазових аренних битв, графічний HUD-інтерфейс та сервіс серіалізації даних мета-прогресу. Для формалізації внутрішньої архітектури й функціональних взаємозв'язків софту було побудовано структурні UML-діаграми та

концептуальну ER-схему даних.

У процесі виробництва ключовим інструментарієм виступило середовище Unity спільно з об'єктно-орієнтованою мовою програмування C#. Створене програмне забезпечення демонструє високу оптимізацію обчислювальних потоків при великому скупченні об'єктів на екрані, базується на принципах модульного кодингу та повністю готове до подальшого масштабування шляхом додавання нових чанків мапи, типів зброї, класів монстрів і додаткових механік виживання.

3 ОПИС РОБОТИ ПРОГРАМНОГО ДОДАТКУ, ЙОГО ВСТАНОВЛЕННЯ, КОРИСТУВАННЯ ТА ТЕСТУВАННЯ

3.1 Керівництво встановлення програмного продукту

Створений програмний комплекс Necrocide представляє собою двовимірний екшен-рогалик жанру Survivor-like, спроєктований для розгортання на персональних комп'ютерах (ПК) під управлінням операційної системи Windows. За необхідності архітектура проєкту дозволяє здійснити кросплатформну компіляцію під інші актуальні платформи. Для безпомилкового функціонування гри вимагається наявність стандартного програмно-апаратного забезпечення, що відповідає технічним лімітам сучасних інтерактивних додатків середньої ланки, побудованих на базі екосистеми Unity. Ключовий пріоритет інсталяційного процесу полягає у забезпеченні безперебійної роботи геймплейних циклів, коректного виведення динамічного HUD-інтерфейсу, а також повноцінної працездатності вхідних обчислювальних механік та систем автоатаки [19].

Для комфортної ігрової сесії рекомендується використовувати обчислювальні станції із встановленою ОС Windows 10 або новіших ітерацій, оснащені двоядерним процесором, об'ємом оперативної пам'яті щонайменше 4 ГБ та графічним прискорювачем із інтеграцією архітектури DirectX 11 (бажано). Додатково на жорсткому диску чи твердотільному накопичувачі ПК має бути зарезервовано не менше 1,5 ГБ вільного дискового простору для розміщення бінарних релізних пакунків, медіа-бібліотек та допоміжних конфігурацій рушія.

Перед безпосереднім розгортанням необхідно виконати скачування стиснутого архіву із релізною версією дистрибутиву. Кінцевий користувач здійснює декомпресію архівних даних у будь-який зручний каталог файлової системи. Для успішної активації інтерактивного софту потрібно виконати таку послідовність дій:

- перейти до кореневої директорії з розархівованою грою (рисунок

3.1);

- здійснити запуск виконуваного релізного файлу Necrocide Core.exe;
- зачекати завершення ініціалізації та появи стартового екрана гри;
- за допомогою графічного інтерфейсу активувати команду «Start Game».

Під час первинного сеансу програмний додаток в автоматичному режимі проводить калібрування системних змінних, виконує завантаження текстурних атласів, аудіо-сплітів, а також генерує локальну структуру конфігураційних файлів профілю. За умови відсутності базових системних компонентів, операційне середовище може видати запит на інсталяцію додаткового програмного забезпечення чи фреймворків, необхідних для стабільної роботи Unity-білдів.

У моменти розпакування та стартового запуску програми слід пересвідчитися, що брандмауер, антивірусний екран чи інші захисні служби операційної системи не блокують запуск виконуваного коду. За потреби, додаток додається до переліку виключень локальної системи безпеки десктопа.

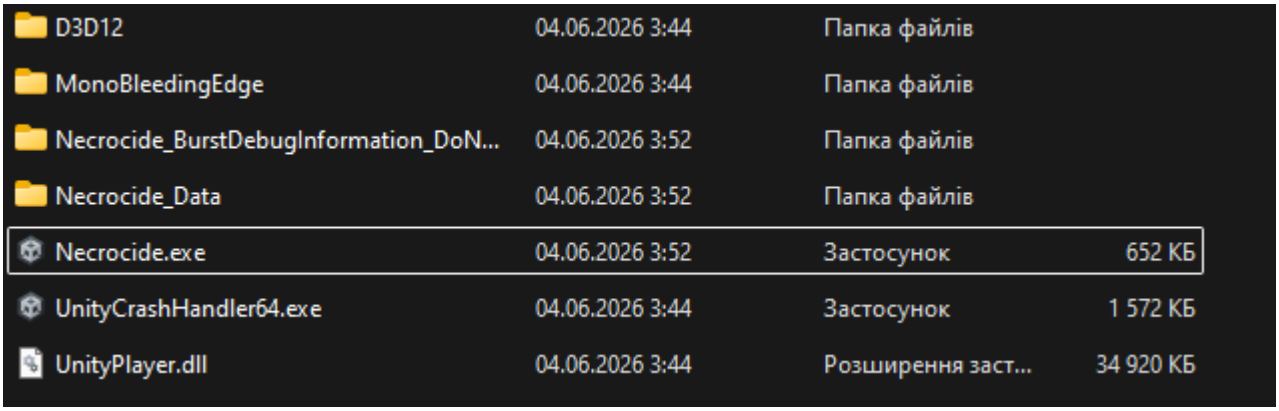

Рисунок 3.1 Виконуваний файл у теці дистрибутиву

Після коректного проходження етапу завантаження користувачеві відкривається функціонал головного меню, параметри конфігурації та безпосередній геймплей. Керування повністю оптимізовано під використання стандартного набору клавіатури й миші та базується на класичних схемах взаємодії людини з комп'ютером. Усі фундаментальні механіки, включаючи

всеспрямований рух некроманта, кругові аури ураження, підбір кристалів досвіду та вибір карт покращень, функціонують одразу «із коробки» без залучення ручних інструментів конфігурування, відповідно до закладених алгоритмів.

3.2 Керівництво користування програмним продуктом

Після завершення ініціалізації та старту екшен-рогалика Necrocide користувачеві відкривається візуальний простір стартового екрана додатка, де надається можливість запустити новий бойовий забіг через команду «Start Game», активувати збережену мета-прогресію за допомогою вкладки «Upgrades» або повністю закрити клієнт («Exit»). Графічна оболонка спроєктована у фірмовому похмурому стилі з домінуванням темно-багрових та неоготичних тонів із застосуванням деталізованих піксельних текстур, що підкреслює загробну атмосферу ігрового процесу. Композиційне наповнення головного меню проілюстровано на рисунку 3.2.

Рисунок 3.2 Головне меню гри Necrocide

Після кліку на клавішу початку забігу система перенаправляє користувача до вікна вибору стартового класу некроманта, де кожен герой володіє

унікальними базовими характеристиками швидкості та міцності (рисунок 3.3). Далі завантажується процедурна бойова локація. Гравець бере під контроль обраного мага, вільно переміщується безкінечними чанками арени, маневрує між перешкодами та збирає ресурси.

Рисунок 3.3 Вікно вибору стартового персонажа

Переміщення протагоніста координується за допомогою стандартної розкладки клавіатури, тоді як запуск заклинань та кругових аур відбувається повністю автоматично згідно з внутрішніми таймерами зброї. Стартове позиціонування мага на полі бою та логіка його пересування продемонстровані на рисунку 3.4.

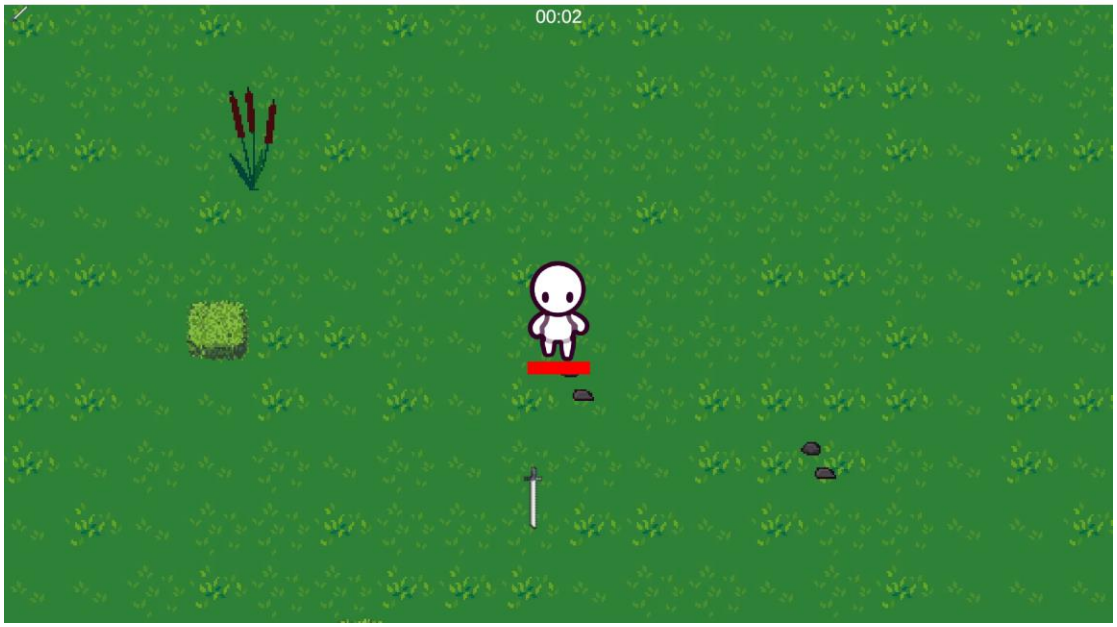

Рисунок 3.4 Перший ігровий екран забігу Necrocide

Фундаментом геймплею є виживання в умовах безперервного натиску орди мобів. Щільність ворожих хвиль динамічно зростає кожну хвилину, що відповідає класичним канонам Survivor-like проєктів та перетворює маневрування на арені на складну координаційну задачу. Приклад початкового бойового зіткнення та перша поява базових супротивників-скелетів наведені на рисунку 3.5.

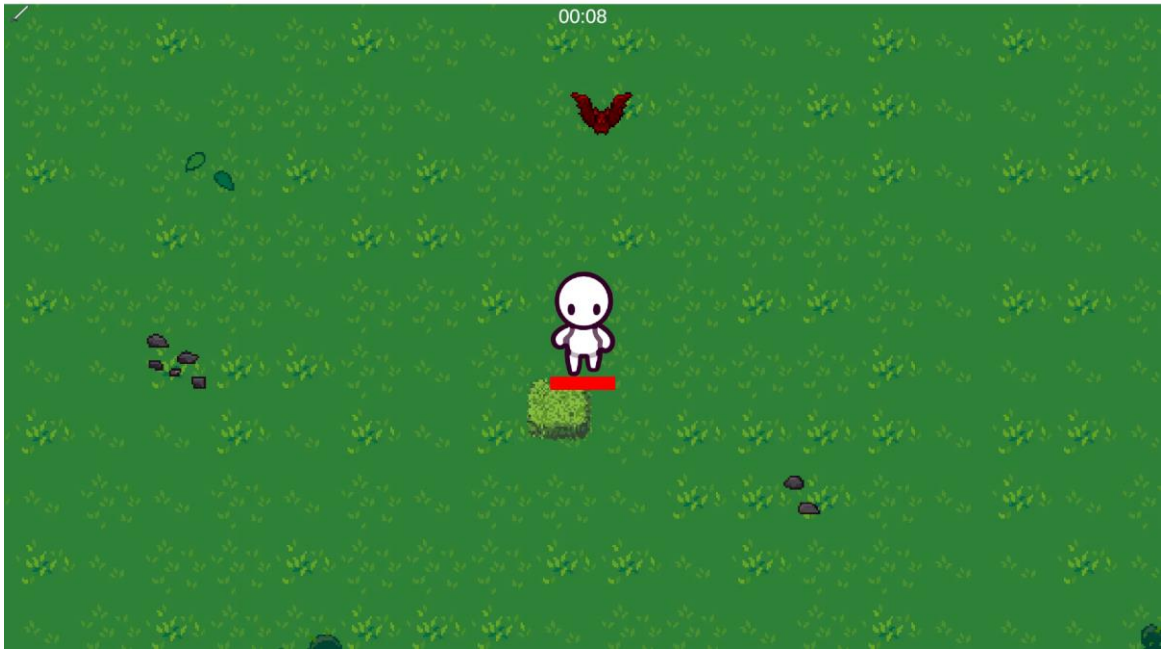

Рисунок 3.5 Зустріч з першим ворогом на арені

Важливою геймплейною віхою є механіка накопичення досвіду за рахунок підбирання магічних кристалів, що випадають із повержених істот. При заповненні шкали прогресу гравець отримує новий рівень, у момент якого бойовий час зупиняється і відкривається спеціальний інтерфейс ротації випадкових карт. У цьому вікні можна обрати нове магічне заклинання або покращити параметри поточних пасивних перків. Візуалізацію моменту підвищення рівня та інтерфейс вибору апгрейдів продемонстровано на рисунку 3.6.

Рисунок 3.6 Вікно підвищення рівня персонажа

У будь-який момент сесії користувач може скористатися меню паузи, яке тимчасово призупиняє розрахунок фізики рушія Unity. Вікно паузи дозволяє оцінити поточний геймплейний білд, переглянути статистику завданої шкоди, змінити гучність аудіосупроводу або достроково завершити поточну спробу виживання. Приклад активації режиму паузи наведено на рисунку 3.7.

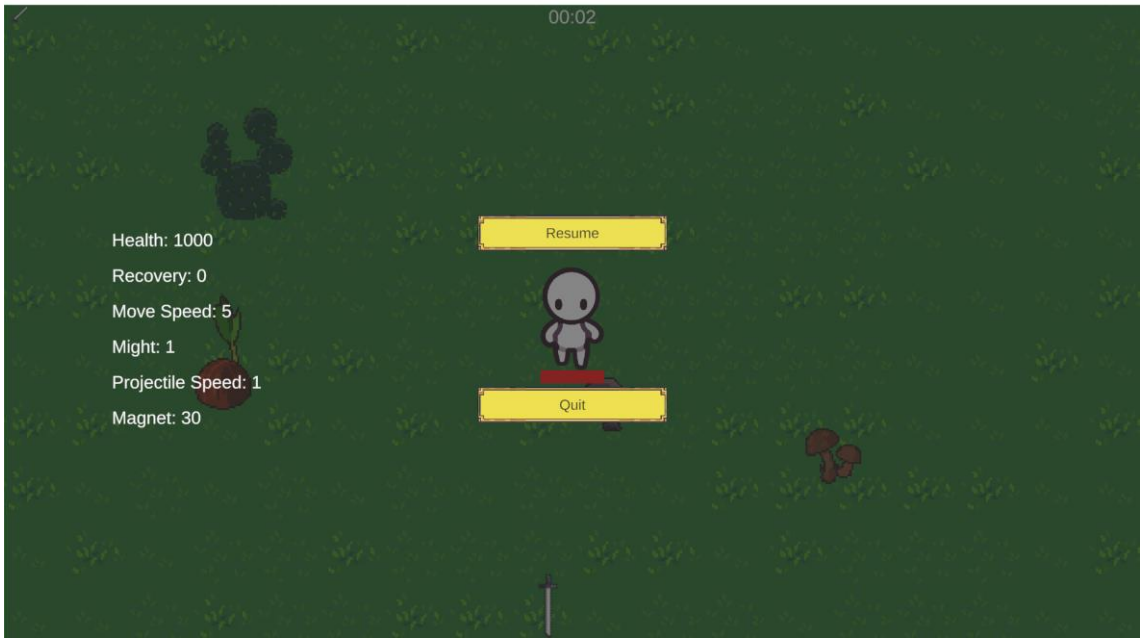

Рисунок 3.7 Ігрове меню паузи

Для підтримки високого рівня динаміки, на арені періодично з'являються масштабні боси з індивідуальними паттернами атак. Протагоніст втрачає очки здоров'я (поточний запас відображається червоною смугою HUD у верхній частині екрана) при прямих колізіях із монстрами, після чого на мить отримує імунітет до повторних ударів. Якщо показник HP падає до нуля, забіг переривається, а система проводить калькуляцію зароблених ресурсів. Фінальний екран результатів із детальною статистикою виживання, часом забігу, кількістю знищених монстрів та зібраного золота зображено на рисунку 3.8.

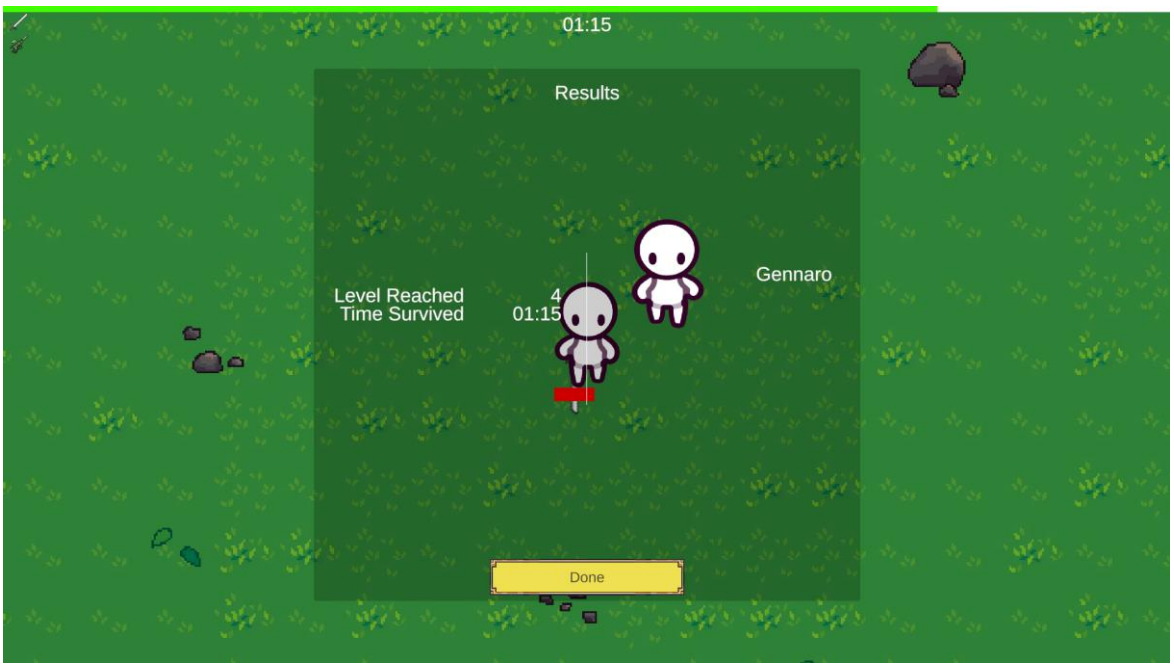

Рисунок 3.8 Екран результатів забігу (Game Over)

Таким чином, розроблений програмний продукт Necrocide гарантує користувачеві насичений геймплей, успішно комбінуючи елементи процедурного виживання, Roguelike-прогресії, автоматизованого бою та менеджменту бойових білдів. Зручна конфігурація управління та адаптивні інтерфейси HUD дозволяють повноцінно взаємодіяти з ігровим світом без необхідності ручного налаштування графічних чи навігаційних параметрів.

3.3 Тестування роботи програмного продукту

Фінальним, проте критично значущим етапом розробки екшен-рогалика Necrocide є всебічний контроль якості інтерактивного софту. Задля досягнення цієї мети було реалізовано комплексне тестування ігрового додатка, спрямоване на перевірку стабільності обчислювальних процесів, коректності функціонування взаємопов'язаних підсистем рушія та відповідності розроблених механік сформованому технічному завданню [20]. Верифікація є невід'ємною частиною життєвого циклу програмного забезпечення, оскільки вона дозволяє оперативно детектувати баги, оцінити рівень інтеграції між ізольованими скриптами та закласти надійний фундамент для позитивного користувацького досвіду (UX).

Під час проведення перевірочних сесій вектор уваги було сфокусовано на алгоритмах всеспрямованого позиціонування мага, механіці швидкого ривка (dash), логіці автоматичного функціонування зброї, а також на інтерфейсі випадкового вибору карт покращень під час підвищення рівня. Додатково аналізу піддавалися поведінкові фактори штучного інтелекту монстрів, процедурна генерація та безшовне зведення нових чанків карти, динамічне віднімання балів стійкості (HP) та коректність модифікації характеристик протагоніста після активації перків.

Тестування здійснювалося як безпосередньо в інтегрованому середовищі розробки Unity, так і в межах ізольованого релізного десктопного білду. Для оцінки роботопридатності застосовувалися різноманітні сценарії поведінки користувача на полі бою, що імітували тривалі ігрові сесії, інтенсивне маневрування між щільними хвилями ворогів та одночасну активацію кількох високотехнологічних заклинань і кругових аура-ефектів. Окремий акцент було зроблено на перевірці відсутності інпут-лагу та точності рендерингу елементів HUD-інтерфейсу за умов критичного скупчення об'єктів на екрані. Результати проведеного функціонального аналізу систематизовано у таблиці 3.1.

Таблиця 3.1 – Результати тестування гри «Necrocide»

№	Функція або механіка	Очікуваний результат	Результат перевірки
1	Навігація протагоніста	Некромант безперешкодно рухається по осях X та Y	✔
2	Меню левелапу та апгрейдів	Час зупиняється, відкривається ротація випадкових карт	✔
3	Автоматична бойова система	Снаряди генеруються за таймером Cooldown зброї	✔
4	Еволюція та кругові аури	Супер-форми заклинань активуються за умови синергії білду	✔
5	Фіксація пошкоджень	Пул здоров'я зменшується з активацією невразливості	✔
6	Спавн хвиль та ШІ монстрів	Орда процедурно з'являється та переслідує координати мага	✔
7	Графічний HUD та меню	Екрани результатів та показники UI відображаються коректно	✔

Зведені дані наочно свідчать, що ключові геймплейні модулі софту

функціонують безперебійно, стабільно та повністю задовольняють висунуті вимоги. У процесі тестування було виявлено поодинокі візуальні артефакти в анімаційних циклах, мікрозатримки при деструкції ворожих спрайтів та крайові помилки розрахунку тригерів колізій чанків, які було успішно усунено на етапі фінального доопрацювання архітектури коду.

Отримані практичні показники повністю доводять загальну працездатність створеного екшен-рогалика, високу точність взаємодії між незалежними C# скриптами та потенційну здатність системи до подальшої модернізації шляхом інтеграції додаткових типів озброєння, класів супротивників та нових алгоритмів мета-прогресії.

3.4 Висновки до розділу 3

У третьому розділі було детально проаналізовано специфіку розгортання, практичного використання та верифікації працездатності екшен-рогалика Necrocide. У межах даного етапу описано алгоритм первинного запуску інтерактивного софту, базові компоненти комунікації між користувачем та програмою, а також функціонування систем всеспрямованої навігації мага, моніторингу життєдіяльності й фіксації ушкоджень та процедурного виживання на арені.

Разом із тим було реалізовано комплексне тестування ключових функціональних вузлів цифрового додатка, у підсумку якого підтверджено повну коректність роботи модулів руху, автоматичного бою, випадкової ротації карт апгрейдів під час левелапу, а також ШІ-алгоритмів поведінки орди мобів та рендерингу графічного HUD-інтерфейсу. Отримані практичні показники наочно засвідчують високий ступінь стабільності та відмовостійкості архітектури створеного геймплейного коду.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було здійснено повний цикл проєктування та програмної реалізації інтерактивного 2D екшен-рогалика Necrocide у жанрі Survivor-like з оригінальним візуальним оформленням у похмурому неоготичному піксель-арті. У межах проведеного дослідження було детально вивчено геймплейні та технічні особливості кращих сучасних представників Action Roguelike напряму. Систематизація ігрових циклів та механік, властивих цьому інтенсивному геймдизайнерському вектору, стала надійною базою для формування алгоритмів процедурного спавну та побудови бойового оточення в розроблюваному застосунку.

У першому розділі було виконано аналітичний огляд ринкових аналогів розроблюваної гри, досліджено специфіку масштабування ворожих хвиль, структури нескінченних арен, ретро-стилістики та побудови циклу геймплею (Core Loop). Спираючись на результати проведеного аналізу, сформовано концептуальну модель програмного софту та зафіксовано перелік ключових вимог до функціональності й оптимізації майбутнього проєкту. Також було науково обґрунтовано вибір технологічного стеку для втілення мети роботи, зокрема ігрового рушія Unity, об'єктно-орієнтованої мови програмування C# та спеціалізованого графічного інструментарію.

У другому розділі спроєктовано модульну архітектуру застосунку, визначено функціональні ліміти обчислень і створено комплекс UML-діаграм та ER-моделей, які детально описують логіку взаємодії між ізольованими скриптами системи. Було програмно втілено базові механіки всеспрямованої навігації некроманта, автоматизовані алгоритми кругових аур та далекобійної зброї, розрахунок пулу здоров'я, інтерфейс випадкового вибору карт покращень (левелап), штучний інтелект хвиль монстрів, процедурне зведення чанків карти та фазові тригери битви з босом. Крім того, розроблено архітектуру динамічного HUD та логіку збереження мета-прогресу.

У третьому розділі покроково описано процеси десктопного розгортання, первинного запуску, конфігурації та практичної експлуатації готового програмного продукту. Разом із тим було реалізовано серію верифікаційних тестів, у ході яких експериментально підтверджено безперебійну роботоздатність бойових алгоритмів, високу відмовостійкість коду при критичному навантаженні та точність відображення елементів графічного інтерфейсу користувача.

У результаті виконання кваліфікаційної роботи було створено повноцінний працездатний релізний прототип Survivor-like гри під назвою «Necrocide» із розвиненими елементами тактичного маневрування, Roguelike-прогресії, синергії озброєння та інших систем функціоналу. Розроблена програма наочно демонструє ефективність комбінування сучасних методів оптимізації 2D-об'єктів із обраним аудіовізуальним супроводом у готичному стилі та може слугувати гнучким фундаментом для подальшого масштабування кодової бази, інтеграції нових класів магів, видів заклинань, розширення бестіарію ворогів та розвитку проєкту на інших платформах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Шелест Х. В. Методи та технології розробки тривимірних та двовимірних ігор на базі сучасних ігрових рушіїв. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2021. № 42. С. 84–91.

2. Бондарчук А. О., Коваленко О. М. Особливості архітектурного проєктування відеоігор жанру Action Roguelike в середовищі Unity. Вісник ХНУ, серія «Технічні науки». 2023. Т. 2, № 4 (312). С. 115–122.

3. Гай С. В. Проєктування та розробка ігрових застосунків мовою C# в Unity: навч. посіб. Київ : ТНТУ, 2022. 248 с.

4. Шаллоуей А., Тротт Дж. Шаблони проєктування. Новий підхід до об'єктно-орієнтованого аналізу та проєктування. Львів : БаК, 2020. 360 с.

5. Фаулер М. UML. Основи. Коротка подорож за стандартною мовою об'єктного моделювання. 4-е вид. Харків : Факт, 2021. 192 с.

6. Unity Real-Time Development Platform: official website. URL: https://unity.com/ (дата звернення: 12.04.2026).

7. Unity Documentation: Manual and Scripting API. Unity Technologies Reference. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 18.04.2026).

8. Скієн С. С. Алгоритми. Керівництво з розробки. 3-є вид. Київ : Діалектика, 2022. 816 с.

9. Миронов Є. В., Сидоренко Д. В. Використання паттерну «Об'єктний пул» (Object Pool) для оптимізації мобільних та десктопних ігор на Unity. Сучасні інформаційні технології. 2024. № 11. С. 45–51.

10. Буч Г., Якобсон А., Рамбо Дж. UML. Керівництво користувача. 2-е вид. Київ : Діалектика, 2021. 496 с.

11. Роджерс С. Level Up! Керівництво з чудового геймдизайну. 2-е вид. Харків : Ранок, 2022. 512 с.

12. Гіні П. Математика у відеоіграх: вектори, матриці та трансформації для 2D та 3D графіки. Дніпро : Середняк Т. К., 2023. 310 с.

13.Торн А. Процедурна генерація рівнів та об'єктів в Unity за допомогою C#. Одеса : Астропринт, 2021. 185 с.

14.Рассел С., Норвіг П. Штучний інтелект: сучасний підхід. 4-е вид. Київ : Діалектика, 2023. 1216 с. (Розділ: Поведінкові патерни штучного інтелекту у відеоіграх).

15.Марченко В. А. Паттерни проєктування ігрових систем: State, Component, Strategy та Command в архітектурі Unity ECS/MonoBehaviour. Комп'ютерні системи та мережі. 2024. Вип. 18. С. 77–86.

16.Аллен С., Тейлор М. Розробка баз даних та ER-моделювання концептуальних схем. Львів : Видавництво Львівської політехніки, 2020. 412 с.

17.Майєрс Г., Сандлер Т., Баджетт Т. Мистецтво тестування програмного забезпечення. 4-е вид. Київ : Вільямс, 2022. 272 с.

18.Ковтуненко О. В. Методологія стрес-тестування та верифікації графічних інтерфейсів (UI/HUD) ігрових систем на базі Unity Profiler. Вісник Академії наук. 2025. № 3. С. 98–104.

19.Microsoft Docs. .NET Fundamentals and C# Programming Guide. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 01.05.2026).

20.Сомервілль І. Інженерія програмного забезпечення. 10-е вид. Харків : Видавництво «Ранок», 2021. 848 с.

ДОДАТОК А

Вихідний код гри

```
// -- FILEPATH: Assets/Scripts/Enemy/Bat.cs ---
using UnityEngine;

public class Bat : MonoBehaviour
{
    EnemyStats enemy;
    Transform player;

    // Викликається перед першим кадром оновлення (Update), після створення MonoBehaviour
    void Start()
    {
        enemy = GetComponent<EnemyStats>();
        player = FindAnyObjectByType<PlayerMovement>().transform;
    }

    // Оновлення викликається раз на кадр
    void Update()
    {
        // Переміщення кажана в бік гравця з урахуванням швидкості ворога та часу
        transform.position = Vector2.MoveTowards(transform.position, player.transform.position,
enemy.currentMoveSpeed * Time.deltaTime);
    }
}
// -- FILEPATH: Assets/Scripts/Enemy/EnemyScriptableObjectScript.cs ---
using UnityEngine;

[CreateAssetMenu(fileName = "EnemyScriptableObjectScript", menuName = "Scriptable
Objects/EnemyScriptableObjectScript")]
public class EnemyScriptableObjectScript : ScriptableObject
{
    [SerializeField]
    float moveSpeed;
    // Швидкість переміщення ворога
    public float MoveSpeed { get =>  moveSpeed; private set => moveSpeed = value;}

    [SerializeField]
    float maxHealth;
    // Максимальний запас здоров'я ворога
    public float MaxHealth { get => maxHealth; private set => maxHealth = value; }

    [SerializeField]
    float damage;
    // Шкода, яку завдає ворог
    public float Damage { get => damage; private set => damage = value; }
}
// -- FILEPATH: Assets/Scripts/Enemy/EnemySpawner.cs ---
using NUnit.Framework;
using System.Collections;
using System.Collections.Generic;
```

```
using Unity.VisualScripting;
using UnityEngine;

public class EnemySpawner : MonoBehaviour
{
    [System.Serializable]
    public class Wave
    {
        public string waveName;             // Назва хвилі
        public List<EnemyGroup> enemyGroup; // Групи ворогів у цій хвилі
        public int WaveQuota;               // Загальна кількість ворогів, які з'являться в цій хвилі
        public float spawnInterval;         // Інтервал, через який з'являються вороги
        public int spawnCount;              // Кількість ворогів, які вже з'явилися в цій хвилі
    }

    [System.Serializable]
    public class EnemyGroup
    {
        public string enemyName;            // Назва типу ворога
        public int enemyCount;              // Кількість ворогів, які з'являться в цій хвилі
        public int spawnCount;              // Кількість ворогів цього типу, які вже з'явилися в цій хвилі
        public GameObject enemyPrefab;      // Преаб ворога
    }

    public List<Wave> waves;                // Список усіх хвиль у грі
    public int currentWaveCount;            // Індекс поточної хвилі

    [Header("Spawner attributes")]
    float spawnTimer;                   // Таймер для спавну
    public int enemiesAlive;                // Кількість живих ворогів на карті
    public int maxEnemiesAllowed;           // Максимально дозволена кількість ворогів одночасно
    public bool maxEnemiesReached = false;  // Чи досягнуто ліміту ворогів
    public float waveInterval;              // Інтервал між хвилями

    [Header("Spawn position")]
    public List<Transform> relativeSpawnPoints; // Відносні точки для спавну навколо гравця

    Transform player;                   // Посилання на трансформ гравця

    void Start()
    {
        // Знаходимо трансформ гравця через його скрипт характеристик
        player = FindAnyObjectByType<PlayerStats>().transform;
        CalculateWaveQuota();
    }

    void Update()
    {
        // Якщо поточна хвиля ще не почала спавнитись, запускаємо корутину нової хвилі
        if(currentWaveCount < waves.Count && waves[currentWaveCount].spawnCount == 0)
        {
            StartCoroutine(BeginNextWave());
```

```
        }

        spawnTimer += Time.deltaTime;
        // Якщо таймер перевищує інтервал спавну, створюємо ворогів
        if (spawnTimer >= waves[currentWaveCount].spawnInterval)
        {
            spawnTimer = 0f;
            SpawnEnemies();
        }
    }

    // Корутина для початку наступної хвилі після затримки
    IEnumerator BeginNextWave()
    {
        yield return new WaitForSeconds(waveInterval);
        if(currentWaveCount < waves.Count - 1)
        {
            currentWaveCount++;
            CalculateWaveQuota();
        }
    }

    // Рахує загальну кількість ворогів для поточної хвилі
    void CalculateWaveQuota()
    {
        int currentWaveQuota = 0;
        foreach(var enemyGroup in waves[currentWaveCount].enemyGroup)
        {
            currentWaveQuota += enemyGroup.enemyCount;
        }
        waves[currentWaveCount].WaveQuota = currentWaveQuota;
        Debug.LogWarning(currentWaveQuota);
    }

    // Логіка створення ворогів на карті
    void SpawnEnemies()
    {
        if (waves[currentWaveCount].spawnCount < waves[currentWaveCount].WaveQuota &&
!maxEnemiesReached)
        {
            foreach (var enemyGroup in waves[currentWaveCount].enemyGroup)
            {
                if (enemyGroup.spawnCount < enemyGroup.enemyCount)
                {
                    // Перевірка, чи не перевищено ліміт живих ворогів
                    if(enemiesAlive >= maxEnemiesAllowed)
                    {
                        maxEnemiesReached = true;
                        return;
                    }

                    // Спавн ворога у випадковій точці навколо гравця
```

```
                Instantiate(enemyGroup.enemyPrefab, player.position +
relativeSpawnPoints[Random.Range(0, relativeSpawnPoints.Count)].position, Quaternion.identity);

                enemyGroup.spawnCount++;
                waves[currentWaveCount].spawnCount++;
                enemiesAlive++;
            }
        }
    }
    // Скидаємо прапорець ліміту, якщо кількість ворогів зменшилась
    if (enemiesAlive < maxEnemiesAllowed)
    {
        maxEnemiesReached = false;
    }
}

// Викликається, коли ворога вбито
public void OnEnemyKilled()
{
    enemiesAlive--;
}
}
// -- FILEPATH: Assets/Scripts/Enemy/EnemyStats.cs ---
using UnityEngine;
using UnityEngine.UIElements;

public class EnemyStats : MonoBehaviour
{
    public EnemyScriptableObjectScript enemyData; // Дані ворога з Scriptable Object

    [HideInInspector]
    public float currentMoveSpeed;  // Поточна швидкість переміщення
    [HideInInspector]
    public float currentDamage;     // Поточна шкода
    [HideInInspector]
    {
    [HideInInspector]
    public float currentHealth;     // Поточне здоров'я

    public float despawnDistance = 20f; // Відстань, на якій ворог деспавниться/переноситься
    Transform player;

    void Awake()
    {
        // Ініціалізація початкових характеристик із Scriptable Object
        currentDamage = enemyData.Damage;
        currentMoveSpeed = enemyData.MoveSpeed;
        currentHealth = enemyData.MaxHealth;
    }

    void Start()
    {
```

```
        player = FindAnyObjectByType<PlayerStats>().transform;
    }

    private void Update()
    {
        // Якщо ворог занадто далеко від гравця, повертаємо його ближче
        if (Vector2.Distance(transform.position, player.position) >= despawnDistance)
        {
            ReturnEnemy();
        }
    }

    // Отримання шкоди ворогом
    public void TakeDamage(float dmg, Vector3 source)
    {
        currentHealth -= dmg;
        if (currentHealth <= 0)
        {
            Kill();
        }
    }

    // Знищення об'єкта ворога при смерті
    public void Kill()
    {
        Destroy(gameObject);
    }

    // Завдання шкоди гравцю при постійному контакті (колізії)
    private void OnCollisionStay2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            PlayerStats player = collision.gameObject.GetComponent<PlayerStats>();
            player.TakeDamage(currentDamage);
        }
    }

    // Повідомляємо спавнер про смерть ворога, коли об'єкт знищується
    private void OnDestroy()
    {
        EnemySpawner es = FindAnyObjectByType<EnemySpawner>();
        if (es != null)
        {
            es.OnEnemyKilled();
        }
    }

    // Повертає ворога ближче до гравця, якщо той втік за межі радіусу (оптимізація замість видалення)
    public void ReturnEnemy()
    {
```

```
        if (!gameObject.scene.isLoaded)
        {
            return;
        }
        EnemySpawner es = FindAnyObjectByType<EnemySpawner>();
        transform.position = player.position + es.relativeSpawnPoints[Random.Range(0, es.relativeSpawnPoints.Count)].position;
    }
}
// -- FILEPATH: Assets/Scripts/Interfaces/Icollectible.cs ---
public interface Icollectible
{
    // Метод, який викликається при зборі предмета
    void Collect();
}
// -- FILEPATH: Assets/Scripts/Map/MapController.cs ---
using System.Collections.Generic;
using System.Collections.Hashtable;
using System.Collections;
using UnityEngine;
using System.Linq;

public class MapController : MonoBehaviour
{
    public Camera referenceCamera;      // Камера для відстеження меж видимого світу
    public float checkInterval = 0.5f; // Інтервал перевірки карти (в секундах)

    [Header("Chunk Settings")]
    public PropRandomizer[] terrainChunks; // Масив префабів чанків землі
    public Vector2 chunkSize = new Vector2(20f, 20f); // Розмір одного чанку
    public LayerMask terrainMask = 1;      // Маска шару для визначення наявності чанку
    public bool deleteCulledChunks = false;// Чи видаляти чанки, які вийшли з поля зору

    // Зберігає попередню позицію та розмір камери для оптимізації перевірок
    Vector3 lastCameraPosition;
    Rect lastCameraRect;
    float cullDistanceSqr;

    void Start()
    {
        // Виведення помилок у консоль, якщо важливі змінні не призначені
        if (!referenceCamera)
            Debug.LogError("MapController не може працювати без посилання на камеру (Reference Camera).");

        if (terrainChunks.Length < 1)
            Debug.LogError("Не призначено жодного чанку землі (Terrain Chunks), динамічна генерація карти неможлива.");

        // Запуск корутини періодичної перевірки карти
        StartCoroutine(HandleMapCheck());
        HandleChunkSpawning(Vector2.zero, true);
```

```
    }

    void Reset()
    {
        referenceCamera = Camera.main;
    }

    // Корутина, яка періодично перевіряє позицію камери та створює нові ділянки карти
    IEnumerator HandleMapCheck()
    {
        for (; ; )
        {
            yield return new WaitForSeconds(checkInterval);

            // Оновлюємо карту лише якщо камера змістилася або змінився її розмір
            Vector3 moveDelta = referenceCamera.transform.position - lastCameraPosition;
            bool hasCamWidthChanged = !Mathf.Approximately(referenceCamera.pixelWidth -
lastCameraRect.width, 0),
                 hasCamHeightChanged = !Mathf.Approximately(referenceCamera.pixelHeight -
lastCameraRect.height, 0);

            if (hasCamWidthChanged || hasCamHeightChanged || moveDelta.magnitude > 0.1f)
            {
                HandleChunkCulling();
                HandleChunkSpawning(moveDelta, true);
            }

            lastCameraPosition = referenceCamera.transform.position;
            lastCameraRect = referenceCamera.pixelRect;
        }
    }

    // Отримує прямокутник (Rect), що відображає область ігрового світу, яку бачить камера
    public Rect GetWorldRectFromViewport()
    {
        if (!referenceCamera)
        {
            Debug.LogError("Камеру посилання не знайдено. Використовується Main Camera.");
            referenceCamera = Camera.main;
        }

        Vector2 minPoint = referenceCamera.ViewportToWorldPoint(referenceCamera.rect.min),
                maxPoint = referenceCamera.ViewportToWorldPoint(referenceCamera.rect.max);
        Vector2 size = new Vector2(maxPoint.x - minPoint.x, maxPoint.y - minPoint.y);
        cullDistanceSqr = Mathf.Max(size.sqrMagnitude, chunkSize.sqrMagnitude) * 3;

        return new Rect(minPoint, size);
    }

    // Отримує всі точки координат, у яких необхідно перевірити наявність чанків
    public Vector2[] GetCheckedPoints()
    {
```

```
        Rect viewArea = GetWorldRectFromViewport();
        Vector2Int tileCount = new Vector2Int(
            (int)Mathf.Ceil(viewArea.width / chunkSize.x) + 1,
            (int)Mathf.Ceil(viewArea.height / chunkSize.y) + 1
        );

        HashSet<Vector2> result = new HashSet<Vector2>();
        for (int y = -1; y < tileCount.y; y++)
        {
            for (int x = -1; x < tileCount.x; x++)
            {
                result.Add(new Vector2(
                    viewArea.min.x + chunkSize.x * x,
                    viewArea.min.y + chunkSize.y * y
                ));
            }
        }

        return result.ToArray();
    }

    // Керує генерацією (спавном) нових чанків на основі руху камери
    void HandleChunkSpawning(Vector2 moveDelta, bool checkWithoutDelta = false)
    {
        HashSet<Vector2> spawnedPositions = new HashSet<Vector2>();
        Vector2 currentPosition = referenceCamera.transform.position;

        // Перевіряємо всі необхідні точки в межах видимості камери
        foreach (Vector3 vp in GetCheckedPoints())
        {
            if (!checkWithoutDelta)
            {
                // Перевірка напрямку руху по осі X
                if (moveDelta.x > 0 && vp.x < 0.5f) continue;
                else if (moveDelta.x < 0 && vp.x > 0.5f) continue;

                // Перевірка напрямку руху по осі Y
                if (moveDelta.y > 0 && vp.y < 0.5f) continue;
                else if (moveDelta.y < 0 && vp.y > 0.5f) continue;
            }

            // Округляємо (прив'язуємо) перевірену позицію до сітки чанків
            Vector3 checkedPosition = SnapPosition(vp);

            // Якщо на цій позиції ще немає об'єкта, створюємо новий чанк
            if (!spawnedPositions.Contains(checkedPosition) &&
!Physics2D.OverlapPoint(checkedPosition, terrainMask))
                SpawnChunk(checkedPosition);

            spawnedPositions.Add(checkedPosition);
        }
    }
```

```
    // Округляє вектор до найближчих координат згідно з розміром чанку (прив'язка до сітки)
    Vector3 SnapPosition(Vector3 position)
    {
        return new Vector3(
            Mathf.Round(position.x / chunkSize.x) * chunkSize.x,
            Mathf.Round(position.y / chunkSize.y) * chunkSize.y,
            transform.position.z
        );
    }

    // Створює чанк у вказаній позиції (випадковий або конкретний варіант)
    PropRandomizer SpawnChunk(Vector3 spawnPosition, int variant = -1)
    {
        if (terrainChunks.Length < 1) return null;
        int rand = variant < 0 ? Random.Range(0, terrainChunks.Length) : variant;
        PropRandomizer chunk = Instantiate(terrainChunks[rand], transform);
        chunk.transform.position = spawnPosition;
        return chunk;
    }

    // Керує приховуванням або видаленням чанків, які знаходяться занадто далеко від камери
    void HandleChunkCulling()
    {
        for (int i = transform.childCount - 1; i >= 0; i--)
        {
            Transform chunk = transform.GetChild(i);
            Vector2 dist = referenceCamera.transform.position - chunk.position;
            bool cull = dist.sqrMagnitude > cullDistanceSqr;
            chunk.gameObject.SetActive(!cull);

            // Якщо увімкнено видалення віддалених чанків — знищуємо їх
            if (deleteCulledChunks && cull) Destroy(chunk.gameObject);
        }
    }
}
// -- FILEPATH: Assets/Scripts/Map/PropRandomizer.cs ---
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class PropRandomizer : MonoBehaviour
{
    public List<GameObject> PropSpawnPoints; // Точки на чанку, де можуть з'явитися об'єкти (декорації)
    public List<GameObject> PropPrefabs;    // Префаби декорацій/об'єктів (камені, дерева, кущі тощо)

    void Start()
    {
        SpawnProps();
    }
```

```
    void Update()
    {

    }

    // Спавнить випадкові декорації у визначених точках чанку
    void SpawnProps()
    {
        foreach (GameObject sp in PropSpawnPoints)
        {
            int rand = Random.Range(0, PropPrefabs.Count);
            // Створення об'єкта у позиції точки спавну
            GameObject prop = Instantiate(PropPrefabs[rand], sp.transform.position,
Quaternion.identity);
            // Робимо створений об'єкт дочірнім до точки спавну
            prop.transform.parent = sp.transform;
        }
    }
}
// -- FILEPATH: Assets/Scripts/Passive Items/Passive.cs ---
using UnityEngine;

/// <summary>
/// Клас, який приймає PassiveData і використовується для збільшення
/// характеристик (статів) гравця при отриманні предмета.
/// </summary>
public class Passive : Item
{
    public PassiveData data;
    [SerializeField] CharacterData.Stats currentBoosts;

    [System.Serializable]
    public struct Modifier
    {
        public string name, description; // Назва та опис модифікатора
        public CharacterData.Stats boosts; // Бонуси до характеристик
    }

    // Для динамічно створених пасивних предметів викликайте Initialise, щоб усе налаштувати.
    public virtual void Initialise(PassiveData data)
    {
        base.Initialise(data);
        this.data = data;
        currentBoosts = data.baseStats.boosts;
    }

    public virtual CharacterData.Stats GetBoosts()
    {
        return currentBoosts;
    }
```

```
    // Підвищує рівень пасивного предмета на 1 і перераховує відповідні характеристики.
    public override bool DoLevelUp()
    {
        base.DoLevelUp();

        // Забороняємо підвищення рівня, якщо вже досягнуто максимуму.
        if (!CanLevelUp())
        {
            Debug.LogWarning(string.Format("Не вдалося підвищити рівень {0} до {1}, максимальний рівень {2} вже досягнуто.", name, currentLevel, data.maxLevel));
            return false;
        }

        // В іншому випадку додаємо характеристики наступного рівня до нашого предмета.
        currentBoosts += data.GetLevelData(++currentLevel).boosts;
        return true;
    }
}
// -- FILEPATH: Assets/Scripts/Passive Items/PassiveData.cs ---
using UnityEngine;

/// <summary>
/// Заміна для класу PassiveItemScriptableObject. Ідея полягає в тому, що ми хочемо зберігати всі
/// дані про рівні пасивного предмета в одному об'єкті, замість створення кількох об'єктів для
/// одного пасивного предмета (що довелося б робити, якби ми продовжували використовувати PassiveItemScriptableObject).
/// </summary>
[CreateAssetMenu(fileName = "Passive Data", menuName = "2D Top-down Rogue-like/Passive Data")]
public class PassiveData : ItemData
{
    public Passive.Modifier baseStats; // Базові характеристики на 1-му рівні
    public Passive.Modifier[] growth;  // Налаштування приросту характеристик для наступних рівнів

    public Passive.Modifier GetLevelData(int level)
    {
        // Беремо характеристики для наступного рівня.
        if (level - 2 < growth.Length)
            return growth[level - 2];

        // Повертаємо порожнє значення та попередження, якщо рівень не налаштований.
        Debug.LogWarning(string.Format("Для пасивного предмета не налаштовані характеристики підвищення до рівня {0}!", level));
        return new Passive.Modifier();
    }
}
// -- FILEPATH: Assets/Scripts/Pick-Ups/ExperianceGem.cs ---
using UnityEngine;

public class ExperianceGem : Pickup
```

```
{
    public int jackpotExperienceGranted; // Кількість досвіду, яку дає цей самоцвіт

    public override void Collect()
    {
        // Перевірка, чи предмет уже підібраний, щоб уникнути подвійного збору
        if (hasBeenCollected)
        {
            return;
        }
        else
        {
            base.Collect();
        }

        // Нараховуємо досвід гравцю
        PlayerStats player = FindAnyObjectByType<PlayerStats>();
        if (player != null)
        {
            player.IncreaseExperience(jackpotExperienceGranted);
        }
    }
}
// -- FILEPATH: Assets/Scripts/Pick-Ups/Health potion.cs ---
using UnityEngine;

public class Healthpotion : Pickup
{
    public int healthToRestore; // Кількість здоров'я для відновлення

    public override void Collect()
    {
        // Перевірка, чи предмет уже підібраний
        if (hasBeenCollected)
        {
            return;
        }
        else
        {
            base.Collect();
        }

        // Відновлюємо здоров'я гравцю
        PlayerStats player = FindAnyObjectByType<PlayerStats>();
        if (player != null)
        {
            player.RestoreHealth(healthToRestore);
        }
    }
}
// -- FILEPATH: Assets/Scripts/Pick-Ups/Pickup.cs ---
using UnityEngine;
```

```
public class Pickup : MonoBehaviour, Icollectible
{
    protected bool hasBeenCollected = false; // Прапорець, який вказує, чи був предмет уже підібраний

    public virtual void Collect()
    {
        hasBeenCollected = true;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        // Якщо в зону тригера потрапляє гравець, знищуємо об'єкт предмета
        if (collision.CompareTag("Player"))
        {
            Destroy(gameObject);
        }
    }
}
// -- FILEPATH: Assets/Scripts/Pick-Ups/TreasureChest.cs ---
using UnityEngine;

public class TreasureChest : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D col)
    {
        PlayerInventory p = col.GetComponent<PlayerInventory>();
        if (p)
        {
            // Випадкове логічне значення (true/false) для визначення тиру скрині
            bool randomBool = Random.Range(0, 2) == 0;

            OpenTreasureChest(p, randomBool);

            // Знищуємо скриню після відкриття
            Destroy(gameObject);
        }
    }

    public void OpenTreasureChest(PlayerInventory inventory, bool isHigherTier)
    {
        // Перебираємо кожен слот зі зброєю, щоб перевірити, чи може вона еволюціонувати.
        foreach (PlayerInventory.Slot s in inventory.weaponsSlots)
        {
            Weapon w = s.item as Weapon;
            if (w == null || w.data.evolutionData == null) continue; // Ігноруємо зброю, якщо вона не може еволюціонувати.

            // Проходимо по всіх можливих еволюціях цієї зброї.
            foreach (ItemData.Evolution e in w.data.evolutionData)
            {
```

```
            // Намагаємося еволюціонувати зброю лише якщо умовою є відкриття скрині (treasureChest).
            if (e.condition == ItemData.Evolution.Condition.treasureChest)
            {
                bool attempt = w.AttemptEvolution(e, 0);
                if (attempt) return; // Якщо еволюція пройшла успішно, зупиняємо перебір.
            }
        }
    }
}

// -- FILEPATH: Assets/Scripts/Player/CameraMovement.cs ---
using UnityEngine;

public class CameraMovement : MonoBehaviour
{
    public Transform target; // Ціль, за якою слідує камера (зазвичай гравець)
    public Vector3 offset;   // Зміщення камери відносно цілі

    void Start()
    {

    }

    void Update()
    {
        // Оновлюємо позицію камери слідом за ціллю з урахуванням зміщення
        transform.position = target.position + offset;
    }
}
// -- FILEPATH: Assets/Scripts/Player/CharacterData.cs ---
using UnityEngine;

[CreateAssetMenu(fileName = "Character Data", menuName = "2D Top-down Rogue-like/Character Data")]
public class CharacterData : ScriptableObject
{
    [SerializeField]
    Sprite icon;
    // Іконка персонажа для інтерфейсу
    public Sprite Icon { get => icon; private set => icon = value; }

    [SerializeField]
    new string name;
    // Ім'я персонажа
    public string Name { get => name; private set => name = value; }

    [SerializeField]
    WeaponData startingWeapon;
    // Стартова зброя персонажа
    public WeaponData StartingWeapon { get => startingWeapon; private set => startingWeapon =
```

```
value; }

    [System.Serializable]
    public struct Stats
    {
        // Основні характеристики: макс. здоров'я, регенерація, швидкість руху,
        // сила (might), швидкість атак/снарядів (speed), радіус магніту (magnet).
        public float maxHealth, recovery, moveSpeed;
        public float might, speed, magnet;

        public Stats(float maxHealth = 100f, float recovery = 0f, float moveSpeed = 1f, float might = 1f,
float speed = 1f, float magnet = 30f)
        {
            this.maxHealth = maxHealth;
            this.recovery = recovery;
            this.moveSpeed = moveSpeed;
            this.might = might;
            this.speed = speed;
            this.magnet = magnet;
        }

        // Перевантаження оператора додавання для зручного складання статів персонажа та
бонусів
        public static Stats operator +(Stats s1, Stats s2)
        {
            s1.maxHealth += s2.maxHealth;
            s1.recovery += s2.recovery;
            s1.moveSpeed += s2.moveSpeed;
            s1.might += s2.might;
            s1.speed += s2.speed;
            s1.magnet += s2.magnet;
            return s1;
        }
    }

    public Stats stats = new Stats(1000);
}
// -- FILEPATH: Assets/Scripts/Player/PlayerAnimator.cs ---
using UnityEngine;

public class PlayerAnimator : MonoBehaviour
{
    Animator am;
    PlayerMovement pm;
    SpriteRenderer sr;

    void Start()
    {
        am = GetComponent<Animator>();
        pm = GetComponent<PlayerMovement>();
        sr = GetComponent<SpriteRenderer>();
    }
```

```
    void Update()
    {
        // Якщо є рух по будь-якій осі, вмикаємо анімацію руху та перевіряємо напрямок спрайту
        if (pm.MoveDir.x != 0 || pm.MoveDir.y != 0)
        {
            am.SetBool("Move", true);
            SpriteDirectionChecker();
        }
        else
        {
            am.SetBool("Move", false);
        }
    }

    // Перевіряє напрямок останнього руху по горизонталі та повертає (фліпає) спрайт
    void SpriteDirectionChecker()
    {
        if (pm.lastHorizontalVector < 0)
        {
            sr.flipX = false; // Дивиться вліво
        }
        else
        {
            sr.flipX = true;  // Дивиться вправо
        }
    }
}
// -- FILEPATH: Assets/Scripts/Player/PlayerCollector.cs ---
using System.Xml.Serialization;
using UnityEngine;

public class PlayerCollector : MonoBehaviour
{
    PlayerStats player;
    CircleCollider2D playerCollector;
    public float pullSpeed; // Швидкість притягування предметів до гравця

    private void Start()
    {
        player = FindAnyObjectByType<PlayerStats>();
        playerCollector = GetComponent<CircleCollider2D>();
    }

    private void Update()
    {
        // Радіус тригера збору динамічно змінюється залежно від показника магніту гравця
        playerCollector.radius = player.CurrentMagnet;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
```

```
        // Якщо об'єкт можна підібрати (реалізує інтерфейс Icollectible)
        if(collision.gameObject.TryGetComponent(out Icollectible icollectible))
        {
            Rigidbody2D rb = collision.gameObject.GetComponent<Rigidbody2D>();
            // Вираховуємо напрямок сили для притягування предмета до гравця
            Vector2 forceDirectin = (transform.position - collision.transform.position).normalized;

            // Прикладаємо силу до предмета, щоб він летів до гравця
            rb.AddForce(forceDirectin * pullSpeed);

            // Викликаємо метод збору предмета
            icollectible.Collect();
        }
    }
}

// -- FILEPATH: Assets/Scripts/Player/PlayerMovement.cs ---
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
    Rigidbody2D rb;

    [HideInInspector]
    public Vector2 MoveDir; // Напрямок руху гравця

    [HideInInspector]
    public float lastHorizontalVector; // Останній ненульовий напрямок по горизонталі

    [HideInInspector]
    public float lastVerticalVector;   // Останній ненульовий напрямок по вертикалі

    [HideInInspector]
    public Vector2 lastMoveVector;     // Загальний останній вектор руху

    PlayerStats player;

    void Start()
    {
        player = GetComponent<PlayerStats>();
        rb = GetComponent<Rigidbody2D>();
        lastMoveVector = new Vector2(0f, -1f); // Початковий напрямок вниз
    }

    void Update()
    {
    }

    void FixedUpdate()
    {
        Move();
```

```
    }

    // Метод нової системи введення (Input System) для отримання вектору руху
    public void OnMove(InputValue value)
    {
        // Якщо гра закінчена, ігноруємо введення
        if (GameManager.instance.isGameOver)
        {
            return;
        }

        MoveDir = value.Get<Vector2>();

        // Зберігаємо останні вектори напрямку, якщо гравець рухається
        if (MoveDir != Vector2.zero)
        {
            if(MoveDir.x != 0) lastHorizontalVector = MoveDir.x;
            if(MoveDir.y != 0) lastVerticalVector = MoveDir.y;
            lastMoveVector = MoveDir;
        }
    }

    // Фізичне переміщення гравця через Rigidbody2D
    private void Move()
    {
        if (GameManager.instance.isGameOver)
        {
            return;
        }
        // Встановлюємо швидкість руху з урахуванням поточних статів гравця
        rb.linearVelocity = MoveDir * player.CurrentMoveSpeed;
    }
}
// -- FILEPATH: Assets/Scripts/Player/PlayerInventory.cs ---
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class PlayerInventory : MonoBehaviour
{
    [System.Serializable]
    public class Slot
    {
        public Item item;
        public Image image;

        // Призначає предмет у слот та оновлює його іконку в інтерфейсі (UI)
        public void Assign(Item assignedItem)
        {
            item = assignedItem;
```

```
            if (item is Weapon)
            {
                Weapon w = item as Weapon;
                image.enabled = true;
                image.sprite = w.data.icon;
            }
            else
            {
                Passive p = item as Passive;
                image.enabled = true;
                image.sprite = p.data.icon;
            }
            Debug.Log(string.Format("Предмет {0} призначено гравцю.", item.name));
        }

        // Очищає слот інвентаря
        public void Clear()
        {
            item = null;
            image.enabled = false;
            image.sprite = null;
        }

        public bool IsEmpty() { return item == null; }
    }

    // Слоти для зброї та пасивних предметів (максимум по 6)
    public List<Slot> weaponsSlots = new List<Slot>(6);
    public List<Slot> passiveSlots = new List<Slot>(6);

    [System.Serializable]
    public class UpgradeUI
    {
        public TMP_Text upgradeNameDisplay;        // Відображення назви покращення
        public TMP_Text upgradeDescriptionDisplay; // Відображення опису покращення
        public Image upgradeIcon;                  // Іконка покращення
        public Button upgradeButton;               // Кнопка для вибору цього покращення
    }

    [Header("UI Elements")]
    public List<WeaponData> availableWeapons = new List<WeaponData>();   // Список доступної для вибору зброї
    public List<PassiveData> availablePassives = new List<PassiveData>(); // Список доступних пасивних предметів
    public List<UpgradeUI> upgradeUIOptions = new List<UpgradeUI>();          // Елементи інтерфейсу для вибору апгрейдів на екрані

    PlayerStats player;

    void Start()
    {
        player = GetComponent<PlayerStats>();
```

```
}

// Перевіряє, чи є в інвентарі предмет певного типу
public bool Has(ItemData type) { return Get(type); }

public Item Get(ItemData type)
{
    if (type is WeaponData) return Get(type as WeaponData);
    else if (type is PassiveData) return Get(type as PassiveData);
    return null;
}

// Шукає пасивний предмет за його даними
public Passive Get(PassiveData type)
{
    foreach (Slot s in passiveSlots)
    {
        Passive p = s.item as Passive;
        if (p != null && p.data == type)
            return p;
    }
    return null;
}

// Шукає зброю за її даними
public Weapon Get(WeaponData type)
{
    foreach (Slot s in weaponsSlots)
    {
        Weapon w = s.item as Weapon;
        if (w != null && w.data == type)
            return w;
    }
    return null;
}

// Видаляє зброю з інвентаря та (опціонально) з пулу можливих покращень
public bool Remove(WeaponData data, bool removeUpgradeAvailability = false)
{
    if (removeUpgradeAvailability) availableWeapons.Remove(data);

    for (int i = 0; i < weaponsSlots.Count; i++)
    {
        Weapon w = weaponsSlots[i].item as Weapon;
        if (w != null && w.data == data)
        {
            weaponsSlots[i].Clear();
            w.OnUnequip();
            Destroy(w.gameObject);
            return true;
        }
    }
```

```
        return false;
    }

    // Видаляє пасивний предмет з інвентаря
    public bool Remove(PassiveData data, bool removeUpgradeAvailability = false)
    {
        if (removeUpgradeAvailability) availablePassives.Remove(data);

        for (int i = 0; i < passiveSlots.Count; i++)
        {
            Passive p = passiveSlots[i].item as Passive;
            if (p != null && p.data == data)
            {
                passiveSlots[i].Clear();
                p.OnUnequip();
                Destroy(p.gameObject);
                return true;
            }
        }
        return false;
    }

    public bool Remove(ItemData data, bool removeUpgradeAvailability = false)
    {
        if (data is PassiveData) return Remove(data as PassiveData, removeUpgradeAvailability);
        else if (data is WeaponData) return Remove(data as WeaponData, removeUpgradeAvailability);
        return false;
    }

    // Додає нову зброю у вільний слот
    public int Add(WeaponData data)
    {
        int slotNum = -1;

        for (int i = 0; i < weaponsSlots.Count; i++)
        {
            if (weaponsSlots[i].IsEmpty())
            {
                slotNum = i;
                break;
            }
        }

        if (slotNum < 0) return slotNum;

        Type weaponType = Type.GetType(data.behaviour);

        if (weaponType != null)
        {
            GameObject go = new GameObject(data.baseStats.name + " Controller");
            Weapon spawnedWeapon = (Weapon)go.AddComponent(weaponType);
            spawnedWeapon.Initialise(data);
```

```
            spawnedWeapon.transform.SetParent(transform);
            spawnedWeapon.transform.localPosition = Vector2.zero;
            spawnedWeapon.OnEquip();

            weaponsSlots[slotNum].Assign(spawnedWeapon);

            if (GameManager.instance != null && GameManager.instance.choosingUpgrade)
            {
                GameManager.instance.EndLevelUp();
            }

            return slotNum;
        }
        else
        {
            Debug.LogWarning(string.Format("Вказано некоректний тип скрипту зброї для {0}.",
data.name));
        }

        return -1;
    }

    // Додає новий пасивний предмет у вільний слот
    public int Add(PassiveData data)
    {
        int slotNum = -1;

        for (int i = 0; i < passiveSlots.Count; i++)
        {
            if (passiveSlots[i].IsEmpty())
            {
                slotNum = i;
                break;
            }
        }

        if (slotNum < 0) return slotNum;

        GameObject go = new GameObject(data.baseStats.name + " Passive");
        Passive p = go.AddComponent<Passive>();
        p.Initialise(data);
        p.transform.SetParent(transform);
        p.transform.localPosition = Vector2.zero;

        passiveSlots[slotNum].Assign(p);

        if (GameManager.instance != null && GameManager.instance.choosingUpgrade)
        {
            GameManager.instance.EndLevelUp();
        }

        player.RecalculateStats();
```

```
        return slotNum;
    }

    public int Add(ItemData data)
    {
        if (data is WeaponData) return Add(data as WeaponData);
        else if (data is PassiveData) return Add(data as PassiveData);
        return -1;
    }

    // Підвищує рівень зброї у вказаному слоті
    public void LevelUpWeapon(int slotIndex, int upgradeIndex)
    {
        if (weaponsSlots.Count > slotIndex)
        {
            Weapon weapon = weaponsSlots[slotIndex].item as Weapon;

            if (!weapon.DoLevelUp())
            {
                Debug.LogWarning(string.Format("Не вдалося підвищити рівень для {0}.",
weapon.name));
                return;
            }
        }

        if (GameManager.instance != null && GameManager.instance.choosingUpgrade)
        {
            GameManager.instance.EndLevelUp();
        }
    }

    // Підвищує рівень пасивного предмета у вказаному слоті
    public void LevelUpPassiveItem(int slotIndex, int upgradeIndex)
    {
        if (passiveSlots.Count > slotIndex)
        {
            Passive p = passiveSlots[slotIndex].item as Passive;
            if (!p.DoLevelUp())
            {
                Debug.LogWarning(string.Format("Не вдалося підвищити рівень для {0}.", p.name));
                return;
            }
        }

        if (GameManager.instance != null && GameManager.instance.choosingUpgrade)
        {
            GameManager.instance.EndLevelUp();
        }

        player.RecalculateStats();
    }
```

```
    // Формує та відображає варіанти покращень при підвищенні рівня (Level Up)
    void ApplyUpgradeOptions()
    {
        List<WeaponData> availableWeaponUpgrades = new List<WeaponData>(availableWeapons);
        List<PassiveData> availablePassiveItemUpgrades = new
List<PassiveData>(availablePassives);

        foreach (UpgradeUI upgradeOption in upgradeUIOptions)
        {
            if (availableWeaponUpgrades.Count == 0 && availablePassiveItemUpgrades.Count == 0)
                return;

            int upgradeType;
            if (availableWeaponUpgrades.Count == 0)
            {
                upgradeType = 2; // Тільки пасивні
            }
            else if (availablePassiveItemUpgrades.Count == 0)
            {
                upgradeType = 1; // Тільки зброя
            }
            else
            {
                upgradeType = UnityEngine.Random.Range(1, 3); // Випадково зброя (1) або пасивний
предмет (2)
            }

            // Логіка для генерації апгрейду активної зброї
            if (upgradeType == 1)
            {
                WeaponData chosenWeaponUpgrade =
availableWeaponUpgrades[UnityEngine.Random.Range(0, availableWeaponUpgrades.Count)];
                availableWeaponUpgrades.Remove(chosenWeaponUpgrade);

                if (chosenWeaponUpgrade != null)
                {
                    EnableUpgradeUI(upgradeOption);

                    bool isLevelUp = false;
                    for (int i = 0; i < weaponsSlots.Count; i++)
                    {
                        Weapon w = weaponsSlots[i].item as Weapon;
                        if (w != null && w.data == chosenWeaponUpgrade)
                        {
                            if (chosenWeaponUpgrade.maxLevel <= w.currentLevel)
                            {
                                DisableUpgradeUI(upgradeOption);
                                isLevelUp = true;
                                break;
                            }
```

```
                    upgradeOption.upgradeButton.onClick.AddListener(() => LevelUpWeapon(i, i));
                    Weapon.Stats nextLevel = chosenWeaponUpgrade.GetLevelData(w.currentLevel
+ 1);
                    upgradeOption.upgradeDescriptionDisplay.text = nextLevel.description;
                    upgradeOption.upgradeNameDisplay.text = nextLevel.name;
                    upgradeOption.upgradeIcon.sprite = chosenWeaponUpgrade.icon;
                    isLevelUp = true;
                    break;
                }
            }

            if (!isLevelUp) // Якщо зброї ще немає в інвентарі, додаємо як нову
            {
                upgradeOption.upgradeButton.onClick.AddListener(()                    =>
Add(chosenWeaponUpgrade));
                upgradeOption.upgradeDescriptionDisplay.text                    =
chosenWeaponUpgrade.baseStats.description;
                upgradeOption.upgradeNameDisplay.text = chosenWeaponUpgrade.baseStats.name;
                upgradeOption.upgradeIcon.sprite = chosenWeaponUpgrade.icon;
            }
        }
    }
    // Логіка для генерації апгрейду пасивного предмета
    else if (upgradeType == 2)
    {
        PassiveData                    chosenPassiveUpgrade                    =
availablePassiveItemUpgrades[UnityEngine.Random.Range(0,
availablePassiveItemUpgrades.Count)];
        availablePassiveItemUpgrades.Remove(chosenPassiveUpgrade);

        if (chosenPassiveUpgrade != null)
        {
            EnableUpgradeUI(upgradeOption);

            bool isLevelUp = false;
            for (int i = 0; i < passiveSlots.Count; i++)
            {
                Passive p = passiveSlots[i].item as Passive;
                if (p != null && p.data == chosenPassiveUpgrade)
                {
                    if (chosenPassiveUpgrade.maxLevel <= p.currentLevel)
                    {
                        DisableUpgradeUI(upgradeOption);
                        isLevelUp = true;
                        break;
                    }

                    upgradeOption.upgradeButton.onClick.AddListener(() => LevelUpPassiveItem(i,
i));
                    Passive.Modifier                    nextLevel                    =
chosenPassiveUpgrade.GetLevelData(p.currentLevel + 1);
                    upgradeOption.upgradeDescriptionDisplay.text = nextLevel.description;
```

```
                    upgradeOption.upgradeNameDisplay.text = nextLevel.name;
                    upgradeOption.upgradeIcon.sprite = chosenPassiveUpgrade.icon;
                    isLevelUp = true;
                    break;
                }
            }

            if (!isLevelUp) // Якщо пасивного предмета ще немає, додаємо як новий
            {
                upgradeOption.upgradeButton.onClick.AddListener(() =>
Add(chosenPassiveUpgrade));
                Passive.Modifier nextLevel = chosenPassiveUpgrade.baseStats;
                upgradeOption.upgradeDescriptionDisplay.text = nextLevel.description;
                upgradeOption.upgradeNameDisplay.text = nextLevel.name;
                upgradeOption.upgradeIcon.sprite = chosenPassiveUpgrade.icon;
            }
        }
    }
  }

  // Очищає слухачі кнопок перед новим вибором
  void RemoveUpgradeOptions()
  {
    foreach (UpgradeUI upgradeOption in upgradeUIOptions)
    {
      upgradeOption.upgradeButton.onClick.RemoveAllListeners();
      DisableUpgradeUI(upgradeOption);
    }
  }

  public void RemoveAndApplyUpgrades()
  {
    RemoveUpgradeOptions();
    ApplyUpgradeOptions();
  }

  void DisableUpgradeUI(UpgradeUI ui)
  {
    ui.upgradeNameDisplay.transform.parent.gameObject.SetActive(false);
  }

  void EnableUpgradeUI(UpgradeUI ui)
  {
    ui.upgradeNameDisplay.transform.parent.gameObject.SetActive(true);
  }
}

// -- FILEPATH: Assets/Scripts/Player/PlayerStats.cs ---
using Unity.VisualScripting;
using UnityEngine;
using System.Collections;
```

```
using System.Collections.Generic;
using Unity.Mathematics;
using UnityEngine.UI;
using TMPro;

public class PlayerStats : MonoBehaviour
{
    CharacterData characterData;
    public CharacterData.Stats baseStats;
    [SerializeField] CharacterData.Stats actualStats;
    float health;

    #region Геттери та Сеттери для Характеристик Гравця
    public float CurrentHealth
    {
        get { return health; }
        set
        {
            if (health != value)
            {
                health = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentHealthDisplay.text = string.Format(
                        "Health: {0:F0} / {1:F0}",
                        health, actualStats.maxHealth
                    );
                }
            }
        }
    }

    public float MaxHealth
    {
        get { return actualStats.maxHealth; }
        set
        {
            if (actualStats.maxHealth != value)
            {
                actualStats.maxHealth = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentHealthDisplay.text = string.Format(
                        "Health: {0:F0} / {1:F0}",
                        health, actualStats.maxHealth
                    );
                }
            }
        }
    }

    public float CurrentRecovery
```

```
    {
        get { return Recovery; }
        set { Recovery = value; }
    }

    public float Recovery
    {
        get { return actualStats.recovery; }
        set
        {
            if (actualStats.recovery != value)
            {
                actualStats.recovery = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentRecoveryDisplay.text = "Recovery: " +
actualStats.recovery;
                }
            }
        }
    }

    public float CurrentMoveSpeed
    {
        get { return MoveSpeed; }
        set { MoveSpeed = value; }
    }

    public float MoveSpeed
    {
        get { return actualStats.moveSpeed; }
        set
        {
            if (actualStats.moveSpeed != value)
            {
                actualStats.moveSpeed = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentMoveSpeedDisplay.text = "Move Speed: " +
actualStats.moveSpeed;
                }
            }
        }
    }

    public float CurrentMight
    {
        get { return Might; }
        set { Might = value; }
    }

    public float Might
```

```
    {
        get { return actualStats.might; }
        set
        {
            if (actualStats.might != value)
            {
                actualStats.might = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentMightDisplay.text = "Might: " + actualStats.might;
                }
            }
        }
    }

    public float CurrentProjectileSpeed
    {
        get { return Speed; }
        set { Speed = value; }
    }

    public float Speed
    {
        get { return actualStats.speed; }
        set
        {
            if (actualStats.speed != value)
            {
                actualStats.speed = value;
                if (GameManager.instance != null)
                {
                    GameManager.instance.currentProjectileSpeedDisplay.text = "Projectile Speed: " +
actualStats.speed;
                }
            }
        }
    }

    public float CurrentMagnet
    {
        get { return Magnet; }
        set { Magnet = value; }
    }

    public float Magnet
    {
        get { return actualStats.magnet; }
        set
        {
            if (actualStats.magnet != value)
            {
                actualStats.magnet = value;
```

```
            if (GameManager.instance != null)
            {
                GameManager.instance.currentMagnetDisplay.text = "Magnet: " + actualStats.magnet;
            }
        }
    }
}
#endregion

SpriteRenderer SR;
private Coroutine flashCoroutine;
public ParticleSystem damageEffect;

[Header("Experience/level")]
public int Experience = 0;
public int level = 1;
public int experienceCap;

[System.Serializable]
public class LevelRange
{
    public int startLevel;
    public int endLevel;
    public int experienceCapIncrease;
}

[Header("I-Frames")]
public float InvicibilityDuration; // Тривалість невразливості
float invicibilityTimer;
bool isInvicible;

public List<LevelRange> levelRanges;
PlayerInventory inventory;

public int weaponIndex;
public int passiveItemIndex;

[Header("UI")]
public Image healthBar;
public Image expBar;
public TextMeshProUGUI levelText;

public void Awake()
{
    SR = GetComponent<SpriteRenderer>();
    SR.color = Color.white;

    characterData = CharacterSelector.GetData();
    CharacterSelector.instance.DestroySingleton();

    inventory = GetComponent<PlayerInventory>();
```

```
        baseStats = actualStats = characterData.stats;
        health = actualStats.maxHealth;
    }

    private void Start()
    {
        inventory.Add(characterData.StartingWeapon);
        experienceCap = levelRanges[0].experienceCapIncrease;

        // Початкове налаштування текстових полів на екрані паузи
        GameManager.instance.currentHealthDisplay.text = "Health: " + CurrentHealth;
        GameManager.instance.currentRecoveryDisplay.text = "Recovery: " + CurrentRecovery;
        GameManager.instance.currentMoveSpeedDisplay.text = "Move Speed: " +
CurrentMoveSpeed;
        GameManager.instance.currentMightDisplay.text = "Might: " + CurrentMight;
        GameManager.instance.currentProjectileSpeedDisplay.text = "Projectile Speed: " +
CurrentProjectileSpeed;
        GameManager.instance.currentMagnetDisplay.text = "Magnet: " + CurrentMagnet;

        GameManager.instance.AssignChosenCharacterUI(characterData);

        UpdateHealthBar();
        UpdateExpBar();
        UpdateLevelText();
    }

    private void Update()
    {
        if (invicibilityTimer > 0)
        {
            invicibilityTimer -= Time.deltaTime;
        }
        else if (isInvicible)
        {
            isInvicible = false;
        }

        Recover();
    }

    // Перераховує сталі характеристики персонажа з урахуванням бонусів від пасивних
предметів
    public void RecalculateStats()
    {
        actualStats = baseStats;
        foreach (PlayerInventory.Slot s in inventory.passiveSlots)
        {
            Passive p = s.item as Passive;
            if (p)
            {
                actualStats += p.GetBoosts();
            }
```

```
        }
    }

    // Збільшує досвід та викликає перевірку на Level Up
    public void IncreaseExperience(int amount)
    {
        Experience += amount;
        LevelUpChecker();
        UpdateExpBar();
    }

    // Перевіряє, чи достатньо досвіду для отримання нового рівня
    void LevelUpChecker()
    {
        if (Experience == 0 && level == 1) return;
        if (Experience >= experienceCap)
        {
            level++;
            Experience -= experienceCap;
            int experienceCapIncrease = 0;
            foreach (LevelRange range in levelRanges)
            {
                if (level >= range.startLevel && level <= range.endLevel)
                {
                    experienceCapIncrease = range.experienceCapIncrease;
                    break;
                }
            }
            experienceCap += experienceCapIncrease;
            UpdateLevelText();
            GameManager.instance.StartLevelUp(); // Відкриваємо вікно Level Up
        }
    }

    // Нанесення шкоди гравцю
    public void TakeDamage(float amount)
    {
        if (!isInvicible)
        {
            CurrentHealth -= amount;
            if (damageEffect) Destroy(Instantiate(damageEffect, transform.position,
Quaternion.identity), 5f);

            invicibilityTimer = InvicibilityDuration;
            isInvicible = true;

            if (flashCoroutine != null)
            {
                StopCoroutine(flashCoroutine);
            }
            flashCoroutine = StartCoroutine(DamageFlashRoutine());
```

```
            if (CurrentHealth <= 0)
            {
                Kill();
            }
            UpdateHealthBar();
        }
    }

    // Корутина для візуального блимання червоним кольором при отриманні шкоди
    IEnumerator DamageFlashRoutine()
    {
        if (SR != null)
        {
            SR.color = Color.red;
            yield return new WaitForSeconds(0.3f);
            SR.color = Color.white;
        }
    }

    // Викликається при смерті гравця
    public void Kill()
    {
        if (!GameManager.instance.isGameOver)
        {
            GameManager.instance.AssignLevelReachedUI(level);
            GameManager.instance.GameOver();
        }
    }

    // Відновлення здоров'я (наприклад, банками)
    public void RestoreHealth(float amount)
    {
        if (CurrentHealth < actualStats.maxHealth)
        {
            CurrentHealth += amount;
            if (CurrentHealth > actualStats.maxHealth)
            {
                CurrentHealth = actualStats.maxHealth;
            }
            UpdateHealthBar();
        }
    }

    // Пасивна щосекундна регенерація здоров'я персонажа
    void Recover()
    {
        if (CurrentHealth < actualStats.maxHealth)
        {
            // ВИПРАВЛЕНО ПОМИЛКУ: Видалено дублюючий рядок додавання регенерації, тепер рахується правильно
            CurrentHealth += Recovery * Time.deltaTime;
```

```
            if (CurrentHealth > actualStats.maxHealth)
            {
                CurrentHealth = actualStats.maxHealth;
            }
            UpdateHealthBar();
        }
    }

    // Оновлення графічних елементів інтерфейсу
    void UpdateHealthBar()
    {
        if (healthBar != null && actualStats.maxHealth > 0)
        {
            healthBar.fillAmount = CurrentHealth / actualStats.maxHealth;
        }
    }

    void UpdateExpBar()
    {
        if (expBar != null && experienceCap > 0)
        {
            expBar.fillAmount = (float)Experience / experienceCap;
        }
    }

    void UpdateLevelText()
    {
        if (levelText != null)
        {
            levelText.text = "LV " + level;
        }
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/Aura.cs ---
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Аура — це ефект періодичної шкоди (damage-over-time), який діє на певну область із заданими інтервалами часу.
/// Використовується для реалізації функціоналу Часнику (Garlic), а також може застосовуватися для створення ефектів Святої води (Holy Water).
/// </summary>
public class Aura : WeaponEffect
{
    Dictionary<EnemyStats, float> affectedTargets = new Dictionary<EnemyStats, float>();
    List<EnemyStats> targetsToUnaffect = new List<EnemyStats>();

    // Метод Update викликається раз на кадр
    void Update()
    {
        Dictionary<EnemyStats, float> affectedTargsCopy = new Dictionary<EnemyStats,
```

```
float>(affectedTargets);

        // Перебираємо кожну ціль під дією аури та зменшуємо час перезарядки
        // дії аури для цієї цілі. Якщо час перезарядки досягає 0, завдаємо шкоди.
        foreach (KeyValuePair<EnemyStats, float> pair in affectedTargsCopy)
        {
            affectedTargets[pair.Key] -= Time.deltaTime;
            if (pair.Value <= 0)
            {
                if (targetsToUnaffect.Contains(pair.Key))
                {
                    // Якщо ціль помічена для видалення, прибираємо її зі списків.
                    affectedTargets.Remove(pair.Key);
                    targetsToUnaffect.Remove(pair.Key);
                }
                else
                {
                    // Скидаємо час перезарядки та завдаємо шкоди.
                    Weapon.Stats stats = weapon.GetStats();
                    affectedTargets[pair.Key] = stats.cooldown;
                    pair.Key.TakeDamage(GetDamage(), transform.position);
                }
            }
        }
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.TryGetComponent(out EnemyStats es))
        {
            // Якщо аура ще не діє на цю ціль, додаємо її
            // до нашого списку вражених цілей.
            if (!affectedTargets.ContainsKey(es))
            {
                // Завжди починаємо з інтервалу 0, щоб ціль отримала
                // шкоду вже на наступному тику (кадрі) Update().
                affectedTargets.Add(es, 0);
            }
            else
            {
                if (targetsToUnaffect.Contains(es))
                {
                    targetsToUnaffect.Remove(es);
                }
            }
        }
    }

    void OnTriggerExit2D(Collider2D other)
    {
        if (other.TryGetComponent(out EnemyStats es))
        {
```

```
            // Не видаляємо ціль одразу після того, як вона залишає зону,
            // оскільки нам усе ще потрібно відстежувати її час перезарядки.
            if (affectedTargets.ContainsKey(es))
            {
                targetsToUnaffect.Add(es);
            }
        }
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/AuraWeapon.cs ---
using UnityEngine;

public class AuraWeapon : Weapon
{
    protected Aura currentAura;

    // Метод Update викликається раз на кадр
    protected override void Update() {
        base.Update();
    }

    public override void OnEquip()
    {
        // Спроба замінити поточну ауру зброї на нову.
        if (currentStats.auraPrefab)
        {
            if (currentAura) Destroy(currentAura);
            currentAura = Instantiate(currentStats.auraPrefab, transform);
            currentAura.weapon = this;
            currentAura.owner = owner;
            currentAura.transform.localScale = new Vector3(currentStats.area, currentStats.area,
currentStats.area);
        }
    }

    public override void OnUnequip()
    {
        if (currentAura) Destroy(currentAura);
        if (currentAura) Destroy(currentAura.gameObject);
    }

    public override bool DoLevelUp()
    {
        if (!base.DoLevelUp()) return false;

        // Якщо до цієї зброї прикріплена аура, ми оновлюємо її параметри.
        if (currentAura)
        {
            currentAura.transform.localScale = new Vector3(currentStats.area, currentStats.area,
currentStats.area);
        }
        return true;
```

```
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/Item.cs ---
using NUnit.Framework.Interfaces;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Базовий клас як для пасивних предметів (Passive), так і для зброї (Weapon). Основне призначення —
/// обробка еволюції предметів, оскільки ми хочемо, щоб і зброя, і пасивні предмети могли еволюціонувати.
/// </summary>
public abstract class Item : MonoBehaviour
{
    public int currentLevel = 1, maxLevel = 1;
    protected ItemData.Evolution[] evolutionData;
    protected PlayerInventory inventory;
    protected PlayerStats owner;

    public virtual void Initialise(ItemData data)
    {
        maxLevel = data.maxLevel;
        // Зберігаємо дані про еволюцію, оскільки нам потрібно відстежувати,
        // чи є всі каталізатори в інвентарі для здійснення еволюції.
        evolutionData = data.evolutionData;

        // У майбутньому потрібно знайти кращий спосіб отримання посилання на інвентар гравця,
        // оскільки поточний варіант є неефективним.
        inventory = FindAnyObjectByType<PlayerInventory>();
        owner = FindAnyObjectByType<PlayerStats>();
    }

    public virtual bool CanLevelUp()
    {
        return currentLevel <= maxLevel;
    }

    // Щоразу, коли предмет підвищує рівень, здійснюється спроба змусити його еволюціонувати.


    // Ефекти, які ви отримуєте під час екіпірування предмета.
    public virtual void OnEquip() { }

    // Ефекти, які видаляються при знятті предмета.
    public virtual void OnUnequip() { }

    public virtual ItemData.Evolution[] CanEvolve()
    {
        List<ItemData.Evolution> possibleEvolutions = new List<ItemData.Evolution>();
```

```
        // Перевіряємо кожну вказану еволюцію та наявність необхідних умов в інвентарі.
        foreach (ItemData.Evolution e in evolutionData)
        {
            if (CanEvolve(e)) possibleEvolutions.Add(e);
        }

        return possibleEvolutions.ToArray();
    }

    // Перевіряє, чи можлива конкретна еволюція.
    public virtual bool CanEvolve(ItemData.Evolution evolution, int levelUpAmount = 1)
    {
        // Неможливо еволюціонувати, якщо предмет ще не досяг необхідного рівня.
        if (evolution.evolutionLevel > currentLevel + levelUpAmount)
        {
            Debug.LogWarning(string.Format("Еволюція не вдалася. Поточний рівень {0}, рівень еволюції {1}", currentLevel, evolution.evolutionLevel));
            return false;
        }

        // Перевірка, чи всі каталізатори знаходяться в інвентарі.
        foreach (ItemData.Evolution.Config c in evolution.catalysts)
        {
            Item item = inventory.Get(c.itemType);
            if (!item || item.currentLevel < c.level)
            {
                Debug.LogWarning(string.Format("Еволюція не вдалася. Відсутній {0}", c.itemType.name));
                return false;
            }
        }
        return true;
    }

    // AttemptEvolution створює нову зброю для персонажа та видаляє всі
    // предмети, які мають бути поглинуті в процесі еволюції.
    public virtual bool AttemptEvolution(ItemData.Evolution evolutionData, int levelUpAmount = 1)
    {
        if (!CanEvolve(evolutionData, levelUpAmount))
            return false;

        // Чи потрібно поглинати пасивні предмети / зброю?
        bool consumePassives = (evolutionData.consumes & ItemData.Evolution.Consumption.passives) > 0;
        bool consumeWeapons = (evolutionData.consumes & ItemData.Evolution.Consumption.weapons) > 0;

        // Перебираємо всі каталізатори та перевіряємо, чи слід їх поглинати.
        foreach (ItemData.Evolution.Config c in evolutionData.catalysts)
        {
            if (c.itemType is PassiveData && consumePassives) inventory.Remove(c.itemType, true);
```

```
            if (c.itemType is WeaponData && consumeWeapons) inventory.Remove(c.itemType, true);
        }

        // Чи маємо ми поглинути самі себе також?
        if (this is Passive && consumePassives) inventory.Remove((this as Passive).data, true);
        else if (this is Weapon && consumeWeapons) inventory.Remove((this as Weapon).data, true);

        // Додаємо нову еволюціоновану зброю до нашого інвентарю.
        inventory.Add(evolutionData.outcome.itemType);

        return true;
    }

    public void EvolveItem(Item item, ItemData.Evolution evolution)
    {
        // Якщо конфігурація еволюції вимагає поглинання зброї або пасивних предметів,
        // ми можемо використовувати функцію EvolveItem.
        if (item.AttemptEvolution(evolution))
        {
            // Оновлюємо інтерфейс підвищення рівня, якщо ми щойно еволюціонували предмет.
            Debug.Log(string.Format("Еволюціоновано {0} у {1}", item.name,
evolution.outcome.itemType.name));
        }
    }

    public virtual bool DoLevelUp()
    {
        if (evolutionData == null) return true;

        // Спроба еволюціонувати в кожну доступну еволюцію цієї зброї,
        // якщо умовою еволюції є підвищення рівня.
        foreach (ItemData.Evolution e in evolutionData)
        {
            if (e.condition == ItemData.Evolution.Condition.auto)
                AttemptEvolution(e);
        }

        return true;
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/ItemData.cs ---
using UnityEngine;

/// <summary>
/// Базовий клас для всіх даних про зброю / пасивні предмети. Цей базовий клас
використовується для того,
/// щоб WeaponData та PassiveItemData можна було використовувати взаємозамінно за потреби.
/// </summary>
public abstract class ItemData : ScriptableObject
{
    public Sprite icon;
    public int maxLevel;
```

```
    [System.Serializable]
    public struct Evolution
    {
        public string name;

        public enum Condition { auto, treasureChest }
        public Condition condition; // Умова еволюції: автоматично або через скриню зі скарбами

        [System.Flags] public enum Consumption { passives = 1, weapons = 2 }
        public Consumption consumes; // Що поглинається: пасивні предмети, зброя чи обидва

        public int evolutionLevel;
        public Config[] catalysts;
        public Config outcome;

        [System.Serializable]
        public struct Config
        {
            public ItemData itemType;
            public int level;
        }
    }

    public Evolution[] evolutionData;
}
// -- FILEPATH: Assets/Scripts/Weapons/Projectile.cs ---
using UnityEngine;

/// <summary>
/// Компонент, який прикріплюється до всіх префабів снарядів. Усі створені снаряди летітимуть у напрямку,
/// куди вони повернуті, і завдаватимуть шкоди при зіткненні з об'єктом.
/// </summary>
[RequireComponent(typeof(Rigidbody2D))]
public class Projectile : WeaponEffect
{
    public enum DamageSource { projectile, owner };
    public DamageSource damageSource = DamageSource.projectile; // Джерело шкоди: сам снаряд чи гравець (для розрахунку відкидання)
    public bool hasAutoAim = false;
    public Vector3 rotationSpeed = new Vector3(0, 0, 0);

    protected Rigidbody2D rb;
    protected int piercing;

    // Метод Start викликається перед першим кадром
    protected virtual void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        Weapon.Stats stats = weapon.GetStats();
        if (rb.bodyType == RigidbodyType2D.Dynamic)
```

```
        {
            rb.angularVelocity = rotationSpeed.z;
            rb.linearVelocity = transform.right * stats.speed;
        }

        // Запобігаємо тому, щоб масштаб (area) дорівнював 0, оскільки це приховає снаряд.
        float area = stats.area == 0 ? 1 : stats.area;
        transform.localScale = new Vector3(
            area * Mathf.Sign(transform.localScale.x),
            area * Mathf.Sign(transform.localScale.y),
            1
        );

        // Встановлюємо, скільки пробиття (piercing) має цей об'єкт.
        piercing = stats.piercing;

        // Знищуємо снаряд після закінчення часу його життя.
        if (stats.lifespan > 0) Destroy(gameObject, stats.lifespan);

        // Якщо у снаряда увімкнено автоприцілювання, автоматично знаходимо відповідного ворога.
        if (hasAutoAim) AcquireAutoAimFacing();
    }

    // Якщо снаряд самонаводяться, він автоматично знайде відповідну ціль для руху до неї.
    public virtual void AcquireAutoAimFacing()
    {
        float aimAngle; // Нам потрібно визначити, куди цілитися.

        // Знаходимо всіх ворогів на екрані.
        EnemyStats[] targets = FindObjectsOfType<EnemyStats>();

        // Вибираємо випадкового ворога (якщо є хоча б один).
        // Інакше вибираємо випадковий кут.
        if (targets.Length > 0)
        {
            EnemyStats selectedTarget = targets[Random.Range(0, targets.Length)];
            Vector2 difference = selectedTarget.transform.position - transform.position;
            aimAngle = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg;
        }
        else
        {
            aimAngle = Random.Range(0f, 360f);
        }

        // Повертаємо снаряд у напрямку, куди ми цілимося.
        transform.rotation = Quaternion.Euler(0, 0, aimAngle);
    }

    // Метод FixedUpdate викликається з фіксованим інтервалом часу
    protected virtual void FixedUpdate()
    {
```

```
        // Керуємо рухом самостійно лише в тому випадку, якщо Rigidbody є кінематичним (Kinematic).
        if (rb.bodyType == RigidbodyType2D.Kinematic)
        {
            Weapon.Stats stats = weapon.GetStats();
            transform.position += transform.right * stats.speed * Time.fixedDeltaTime;
            rb.MovePosition(transform.position);
            transform.Rotate(rotationSpeed * Time.fixedDeltaTime);
        }
    }

    protected virtual void OnTriggerEnter2D(Collider2D other)
    {
        EnemyStats es = other.GetComponent<EnemyStats>();
        BreakableProps p = other.GetComponent<BreakableProps>();

        // Взаємодіємо лише з ворогами або об'єктами, які можна зламати.
        if (es)
        {
            // If there is an owner, and the damage source is set to owner,
            // we will calculate knockback using the owner instead of the projectile.
            // Якщо є власник і джерелом шкоди вказано власника, ми розраховуємо відкидання від гравця, а не від снаряда.
            Vector3 source = damageSource == DamageSource.owner && owner ? owner.transform.position : transform.position;

            // Завдає шкоди та зменшує показник пробиття снаряда.
            es.TakeDamage(GetDamage(), source);

            Weapon.Stats stats = weapon.GetStats();
            piercing--;
            if (stats.hitEffect)
            {
                Destroy(Instantiate(stats.hitEffect, transform.position, Quaternion.identity), 5f);
            }
        }
        else if (p)
        {
            p.TakeDamage(GetDamage());
            piercing--;

            Weapon.Stats stats = weapon.GetStats();
            if (stats.hitEffect)
            {
                Destroy(Instantiate(stats.hitEffect, transform.position, Quaternion.identity), 5f);
            }
        }

        // Знищуємо цей об'єкт, якщо у нього закінчився ліміт пробиття після влучань.
        if (piercing <= 0) Destroy(gameObject);
    }
}
```

```
// -- FILEPATH: Assets/Scripts/Weapons/ProjectileWeapon.cs ---
using UnityEngine;

public class ProjectileWeapon : Weapon
{
    protected float currentAttackInterval;
    protected int currentAttackCount; // Кількість разів, яку ця атака має повторитися.

    protected override void Update()
    {
        base.Update();

        // Інакше, якщо інтервал атаки падає нижче нуля, ми також викликаємо атаку.
        if (currentAttackInterval > 0)
        {
            currentAttackInterval -= Time.deltaTime;
            if (currentAttackInterval <= 0) Attack(currentAttackCount);
        }
    }

    public override bool CanAttack()
    {
        if (currentAttackCount > 0) return true;
        return base.CanAttack();
    }

    protected override bool Attack(int attackCount = 1)
    {
        // Якщо префаб снаряда не призначений, виводимо попередження.
        if (!currentStats.projectilePrefab)
        {
            Debug.LogWarning(string.Format("Префаб снаряда не встановлено для {0}", name));
            currentCooldown = data.baseStats.cooldown;
            return false;
        }

        // Чи можемо ми атакувати?
        if (!CanAttack()) return false;

        // Розраховуємо кут та зміщення (offset) нашого створеного снаряда.
        float spawnAngle = GetSpawnAngle();

        // Створюємо копію снаряда.
        Projectile prefab = Instantiate(
            currentStats.projectilePrefab,
            owner.transform.position + (Vector3)GetSpawnOffset(spawnAngle),
            Quaternion.Euler(0, 0, spawnAngle)
        );

        prefab.weapon = this;
        prefab.owner = owner;
```

```
        // Скидаємо час перезарядки лише в тому випадку, якщо ця атака була викликана за кулдауном.
        if (currentCooldown <= 0)
            currentCooldown += currentStats.cooldown;

        attackCount--;

        // Чи потрібно нам виконати ще одну атаку в цій серії?
        if (attackCount > 0)
        {
            currentAttackCount = attackCount;
            currentAttackInterval = data.baseStats.projectileInterval;
        }

        return true;
    }

    // Визначає, в який бік має дивитися снаряд при створенні.
    protected virtual float GetSpawnAngle()
    {
        return Mathf.Atan2(movement.lastMoveVector.y, movement.lastMoveVector.x) * Mathf.Rad2Deg;
    }

    // Генерує випадкову точку для створення снаряда та повертає її відповідно до кута spawnAngle.
    protected virtual Vector2 GetSpawnOffset(float spawnAngle = 0)
    {
        return Quaternion.Euler(0, 0, spawnAngle) * new Vector2(
            Random.Range(currentStats.spawnVariance.xMin, currentStats.spawnVariance.xMax),
            Random.Range(currentStats.spawnVariance.yMin, currentStats.spawnVariance.yMax)
        );
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/Weapon.cs ---
using UnityEngine;

/// <summary>
/// Компонент, який має бути прикріплений до всіх префабів Зброї. Префаб зброї працює разом із
/// ScriptableObjects типу WeaponData для керування та виконання поведінки всієї зброї в грі.
/// </summary>
public abstract class Weapon : Item
{
    [System.Serializable]
    public struct Stats
    {
        public string name, description;

        [Header("Visuals")]
        public Projectile projectilePrefab; // Якщо прикріплено, снаряд з'являтиметься щоразу, коли зброя перезаряджається.
```

```
        public Aura auraPrefab; // Якщо прикріплено, аура з'являтиметься під час екіпірування зброї.
        public ParticleSystem hitEffect;
        public Rect spawnVariance;

        [Header("Values")]
        public float lifespan; // Якщо 0, ефект триватиме вічно.
        public float damage, damageVariance, area, speed, cooldown, projectileInterval, knockback;
        public int number, piercing, maxInstances;

        // Дозволяє використовувати оператор + для додавання двох структур Stats разом.
        // Це дуже важливо для подальшого підвищення характеристик зброї при підвищенні рівня.
        public static Stats operator +(Stats s1, Stats s2)
        {
            Stats result = new Stats();
            result.name = s2.name ?? s1.name;
            result.description = s2.description ?? s1.description;
            result.projectilePrefab = s2.projectilePrefab ?? s1.projectilePrefab;
            result.auraPrefab = s2.auraPrefab ?? s1.auraPrefab;
            result.hitEffect = s2.hitEffect == null ? s1.hitEffect : s2.hitEffect;
            result.spawnVariance = s2.spawnVariance;
            result.lifespan = s1.lifespan + s2.lifespan;
            result.damage = s1.damage + s2.damage;
            result.damageVariance = s1.damageVariance + s2.damageVariance;
            result.area = s1.area + s2.area;
            result.speed = s1.speed + s2.speed;
            result.cooldown = s1.cooldown + s2.cooldown;
            result.number = s1.number + s2.number;
            result.piercing = s1.piercing + s2.piercing;
            result.projectileInterval = s1.projectileInterval + s2.projectileInterval;
            result.knockback = s1.knockback + s2.knockback;
            return result;
        }

        // Отримання завданої шкоди (з урахуванням випадкового розкиду).
        public float GetDamage()
        {
            return damage + Random.Range(0, damageVariance);
        }
    }

    protected Stats currentStats;

    public WeaponData data;

    protected float currentCooldown;

    protected PlayerMovement movement; // Посилання на скрипт руху гравця.

    // Для динамічно створеної зброї викликайте initialise для початкового налаштування.
    public virtual void Initialise(WeaponData data)
```

```
    {
        base.Initialise(data);

        this.data = data;
        currentStats = data.baseStats;
        movement = GetComponentInParent<PlayerMovement>();
        currentCooldown = currentStats.cooldown;
    }

    protected virtual void Awake()
    {
        // Призначаємо характеристики заздалегідь, оскільки вони будуть використовуватися іншими скриптами пізніше.
        if (data) currentStats = data.baseStats;
    }

    protected virtual void Start()
    {
        // Не ініціалізуємо зброю, якщо дані зброї (WeaponData) не призначені.
        if (data)
        {
            Initialise(data);
        }
    }

    protected virtual void Update()
    {
        currentCooldown -= Time.deltaTime;
        if (currentCooldown <= 0f) // Щойно час перезарядки стає менше або дорівнює 0 — атакуємо
        {
            Attack(currentStats.number);
        }
    }

    public override bool DoLevelUp()
    {
        base.DoLevelUp();
        // Забороняємо підвищення рівня, якщо ми вже досягли максимального рівня.
        if (!CanLevelUp())
        {
            Debug.LogWarning(string.Format("Неможливо підвищити рівень {0} до рівня {1}, макс. рівень {2} вже досягнуто.", name, currentLevel, data.maxLevel));
            return false;
        }

        // Інакше додаємо характеристики наступного рівня до нашої зброї.
        currentStats += data.GetLevelData(++currentLevel);
        return true;
    }

    public virtual bool CanAttack()
```

```
    {
        return currentCooldown <= 0;
    }

    // Виконує атаку зброєю.
    // Повертає true, якщо атака пройшла успішно.
    // Тут базовий метод нічого не робить. Ми маємо перевизначити його у дочірніх класах.
    protected virtual bool Attack(int attackCount = 1)
    {
        if (CanAttack())
        {
            currentCooldown += currentStats.cooldown;
            return true;
        }
        return false;
    }

    // Отримує кількість шкоди, яку має завдати зброя.
    // Враховуються характеристики зброї (включно з розкидом шкоди), а також характеристика Сили (Might) персонажа.
    public virtual float GetDamage()
    {
        return currentStats.GetDamage() * owner.CurrentMight;
    }

    // Для отримання поточних характеристик зброї.
    public virtual Stats GetStats() { return currentStats; }
}
// -- FILEPATH: Assets/Scripts/Weapons/WeaponData.cs ---
using UnityEngine;

[CreateAssetMenu(fileName = "Weapon Data", menuName = "2D Top-down Rogue-like/Weapon Data")]
public class WeaponData : ItemData
{
    [HideInInspector] public string behaviour;
    public Weapon.Stats baseStats;
    public Weapon.Stats[] linearGrowth;
    public Weapon.Stats[] randomGrowth;

    // Повертає приріст характеристик / опис для наступного рівня.
    public Weapon.Stats GetLevelData(int level)
    {
        // Беремо характеристики для наступного рівня лінійного зростання.
        if (level - 2 < linearGrowth.Length)
            return linearGrowth[level - 2];

        // Інакше вибираємо одну з характеристик з масиву випадкового зростання (якщо лінійні закінчилися).
        if (randomGrowth.Length > 0)
            return randomGrowth[Random.Range(0, randomGrowth.Length)];
```

```
        // Повертаємо порожнє значення та попередження, якщо нічого не налаштовано.
        Debug.LogWarning(string.Format("Для зброї не налаштовано характеристики підвищення рівня для рівня {0}!", level));
        return new Weapon.Stats();
    }
}
// -- FILEPATH: Assets/Scripts/Weapons/WeaponEffect.cs ---
using UnityEngine;

/// <summary>
/// Ігровий об'єкт (GameObject), який створюється як ефект від пострілу або дії зброї, наприклад: снаряди, аури, пульсації.
/// </summary>
public class WeaponEffect : MonoBehaviour
{
    [HideInInspector] public PlayerStats owner;
    [HideInInspector] public Weapon weapon;

    public float GetDamage()
    {
        return weapon.GetDamage();
    }
}
// -- FILEPATH: Assets/Scripts/BreakableProps.cs ---
using UnityEngine;

public class BreakableProps : MonoBehaviour
{
    public float health;

    public void TakeDamage(float damage)
    {
        health -= damage;
        if (health < 0)
        {
            Kill();
        }
    }

    public void Kill()
    {
        Destroy(gameObject);
    }
}
// -- FILEPATH: Assets/Scripts/CharacterSelector.cs ---
using UnityEngine;

public class CharacterSelector : MonoBehaviour
{
    public static CharacterSelector instance;
    public CharacterData characterData;
```

```
    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Debug.LogWarning("Знайдено дублікат синглтону " + this + ". Об'єкт знищено.");
            Destroy(gameObject);
        }
    }

    public static CharacterData GetData()
    {
        if (instance && instance.characterData)
            return instance.characterData;
        else
        {
            // Якщо жодних даних персонажа не призначено, ми обираємо одного випадковим чином.
            CharacterData[] characters = Resources.FindObjectsOfTypeAll<CharacterData>();
            if (characters.Length > 0)
            {
                return characters[Random.Range(0, characters.Length)];
            }
        }
        return null;
    }

    public void SelectCharacter(CharacterData character)
    {
        characterData = character;
    }

    public void DestroySingleton()
    {
        instance = null;
        Destroy(gameObject);
    }
}
// -- FILEPATH: Assets/Scripts/DropeRateManager.cs ---
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class DropeRateManager : MonoBehaviour
{
    [System.Serializable]
    public class Drops
    {
```

```
        public string name;
        public GameObject itemPrefab;
        public float dropRate;
    }

    public List<Drops> drops;

    private void OnDestroy()
    {
        // Перевірка, чи сцена зараз завантажена, щоб уникнути спавну предметів під час виходу з гри чи зміни сцени.
        if (!gameObject.scene.isLoaded)
        {
            return;
        }

        float randomNumber = UnityEngine.Random.Range(0f, 100f);
        List<Drops> possibleDrops = new List<Drops>();

        foreach (Drops rate in drops)
        {
            if (randomNumber <= rate.dropRate)
            {
                possibleDrops.Add(rate);
            }
            if (possibleDrops.Count > 0)
            {
                Drops drops = possibleDrops[UnityEngine.Random.Range(0, possibleDrops.Count)];
                Instantiate(rate.itemPrefab, transform.position, Quaternion.identity);
            }
        }
    }
}
// -- FILEPATH: Assets/Scripts/SceneController.cs ---
using UnityEngine;
using UnityEngine.SceneManagement;

public class SceneController : MonoBehaviour
{
    public void SceneChange(string name)
    {
        SceneManager.LoadScene(name);
        Time.timeScale = 1;
    }

    public void ExitGame()
    {
        // Ця команда закриває гру (працює у фінальному білді)
        Application.Quit();

        // Цей рядок потрібен лише для перевірки в самому Unity Editor,
        // оскільки Application.Quit() не закриває сам редактор.
```

```
        Debug.Log("Гра закрилася!");
    }
}
// -- FILEPATH: Assets/Scripts/GameManager.cs ---
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    public static GameManager instance;

    // Визначаємо різні стани гри
    public enum GameState
    {
        Gameplay,
        Paused,
        GameOver,
        LevelUp
    }

    // Зберігаємо поточний та попередній стани гри
    public GameState currentState;
    public GameState previousState;

    [Header("Screens")]
    public GameObject pauseScreen;
    public GameObject resultsScreen;
    public GameObject levelUpScreen;

    [Header("Current Stat Displays")]
    public TMPro.TMP_Text currentHealthDisplay;
    public TMPro.TMP_Text currentRecoveryDisplay;
    public TMPro.TMP_Text currentMoveSpeedDisplay;
    public TMPro.TMP_Text currentMightDisplay;
    public TMPro.TMP_Text currentProjectileSpeedDisplay;
    public TMPro.TMP_Text currentMagnetDisplay;

    [Header("Results Screens Displays")]
    public Image chosenCharacterImage;
    public TMPro.TMP_Text chosenCharacterName;
    public TMPro.TMP_Text levelReachedDisplay;
    public TMPro.TMP_Text timeSurvivedDisplay;
    public List<Image> chosenWeaponsUI = new List<Image>(6);
    public List<Image> chosenPassiveItemsUI = new List<Image>(6);

    [Header("Stopwatch")]
    public float timeLimit; // Ліміт часу в секундах
    float stopwatchTime; // Поточний час, що минув з моменту запуску секундоміра
    public TMPro.TMP_Text stopwatchDisplay;
```

```
public bool isGameOver = false;

public bool choosingUpgrade;

public GameObject playerObject;

void Awake()
{
    if (instance == null)
    {
        instance = this;
    }
    else
    {
        Debug.LogWarning("EXTRA " + this + " DELETED");
        Destroy(gameObject);
    }
    DisableScreens();
}

void Update()
{
    switch (currentState)
    {
        case GameState.Gameplay:
            CheckForPauseAndResume();
            UpdateStopwatch();
            // Код для стану ігрового процесу (ігри)
            break;

        case GameState.Paused:
            CheckForPauseAndResume();
            // Код для стану паузи
            break;

        case GameState.GameOver:
            // Код для стану завершення гри (програшу/виграшу)
            if (!isGameOver)
            {
                isGameOver = true;
                Time.timeScale = 0f; // Зупиняємо гру
                Debug.Log("GAME IS OVER");
                DisplayResults();
            }
            break;

        case GameState.LevelUp:
            if (!choosingUpgrade)
            {
                choosingUpgrade = true;
                Time.timeScale = 0f; // Поки що ставимо гру на паузу
                Debug.Log("Upgrades shown");
```

```
                levelUpScreen.SetActive(true);
            }
            break;
        default:
            Debug.LogWarning("STATE DOES NOT EXIST");
            break;
    }
}

public void ChangeState(GameState newState)
{
    currentState = newState;
}

public void PauseGame()
{
    if (currentState != GameState.Paused)
    {
        previousState = currentState;
        ChangeState(GameState.Paused);
        Time.timeScale = 0f; // Зупиняємо гру
        pauseScreen.SetActive(true);
        Debug.Log("Game is paused");
    }
}

public void ResumeGame()
{
    if (currentState == GameState.Paused)
    {
        ChangeState(previousState);
        Time.timeScale = 1f; // Відновлюємо гру
        pauseScreen.SetActive(false);
        Debug.Log("Game is resumed");
    }
}

void CheckForPauseAndResume()
{
    // Метод перевіряє, чи була натиснута клавіша Escape для паузи або відновлення гри
    if (Keyboard.current != null && Keyboard.current.escapeKey.wasPressedThisFrame)
    {
        if (currentState == GameState.Paused)
        {
            ResumeGame();
        }
        else
        {
            PauseGame();
        }
    }
}
```

```
    void DisableScreens()
    {
        pauseScreen.SetActive(false);
        resultsScreen.SetActive(false);
        levelUpScreen.SetActive(false);
    }

    public void GameOver()
    {
        timeSurvivedDisplay.text = stopwatchDisplay.text;
        ChangeState(GameState.GameOver);
    }

    void DisplayResults()
    {
        resultsScreen.SetActive(true);
    }

    public void AssignChosenCharacterUI(CharacterData chosenCharacterData)
    {
        chosenCharacterImage.sprite = chosenCharacterData.Icon;
        chosenCharacterName.text = chosenCharacterData.Name;
    }

    public void AssignLevelReachedUI(int levelReachedData)
    {
        levelReachedDisplay.text = levelReachedData.ToString();
    }

    public void AssignChosenWeaponsAndPassiveItemsUI(List<Image> chosenWeaponsData,
List<Image> chosenPassiveItemsData)
    {
        if (chosenWeaponsData.Count != chosenWeaponsUI.Count || chosenPassiveItemsData.Count
!= chosenPassiveItemsUI.Count)
        {
            Debug.Log("Chosen weapons and passive items data lists have different lengths");
            return;
        }

        // Оновлюємо відображення іконок обраної зброї в інтерфейсі результатів
        for (int i = 0; i < chosenWeaponsUI.Count; i++)
        {
            if (chosenWeaponsData[i].sprite)
            {
                chosenWeaponsUI[i].enabled = true;
                chosenWeaponsUI[i].sprite = chosenWeaponsData[i].sprite;
            }
            else
            {
                chosenWeaponsUI[i].enabled = false;
            }
```

```
        }

        // Оновлюємо відображення іконок обраних пасивних предметів в інтерфейсі результатів
        for (int i = 0; i < chosenPassiveItemsUI.Count; i++)
        {
            if (chosenPassiveItemsData[i].sprite)
            {
                chosenPassiveItemsUI[i].enabled = true;
                chosenPassiveItemsUI[i].sprite = chosenPassiveItemsData[i].sprite;
            }
            else
            {
                chosenPassiveItemsUI[i].enabled = false;
            }
        }
    }

    void UpdateStopwatch()
    {
        stopwatchTime += Time.deltaTime;
        UpdateStopwatchDisplay();
        if (stopwatchTime >= timeLimit)
        {
            playerObject.SendMessage("Kill");
        }
    }

    void UpdateStopwatchDisplay()
    {
        // Розраховуємо кількість хвилин і секунд, що минули
        int minutes = Mathf.FloorToInt(stopwatchTime / 60);
        int seconds = Mathf.FloorToInt(stopwatchTime % 60);

        stopwatchDisplay.text = string.Format("{0:00}:{1:00}", minutes, seconds);
    }

    public void StartLevelUp()
    {
        ChangeState(GameState.LevelUp);
        playerObject.SendMessage("RemoveAndApplyUpgrades");
    }

    public void EndLevelUp()
    {
        choosingUpgrade = false;
        Time.timeScale = 1f; // Відновлюємо гру
        levelUpScreen.SetActive(false);
        ChangeState(GameState.Gameplay);
    }
}
```