КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет автоматизації інформаційних технологій

Кафедра інформаційних технологій

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

на тему:

Комп'ютерна гра в жанрі Rogue-like на основі ігрового двигуна Unity.
Ч.1. Розробка скриптів та ігрових механік, їх впровадження

Цись Максим Володимирович

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ЗАТВЕРДЖУЮ
Завідувачка кафедри ІТ
д.т.н., доцент Гончаренко Т.А.

„___” _____2025 року

КВАЛІФІКАЦІЙНА РОБОТА ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему: «**Комп'ютерна гра в жанрі Rogue-like на основі ігрового двигуна Unity. Ч.1. Розробка скриптів та ігрових механік, їх впровадження**»

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач
Цись Максим Володимирович
122 «Комп’ютерні науки»
(спеціальність)
Інформаційні управляючі системи і технології
(освітня програма)
Групи КН-21-2
Керівник Рябчун Ю.В.
(прізвище та ініціали)
Доктор філософії
(вчене звання, науковий ступінь)
Рецензент к.т.н., доц. Баліна О.І.
(Прізвище та ініціали)

Ідентичність підтверджую

Київ, 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет:	Автоматизації інформаційних технологій
Випускова кафедра:	Інформаційних технологій
Освітній ступінь:	Бакалавр
Спеціальність:	Комп’ютерні науки
Освітня програма:	Інформаційні управляючі системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТ

Тетяна ГОНЧАРЕНКО

_____

„____”_____2025 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

	Цись Максиму Володимировичу
Тема роботи Комп'ютерна гра в жанрі Rogue-like на основі ігрового двигуна Unity. Ч.1. Розробка скриптів та ігрових механік, їх впровадження	

затверджена наказом ректора КНУБА № 235/23/25 від «14» лютого 2025 року

2. Керівник роботи	Рябчун Юлія Володимирівна, PhD

3. Строк подання Здобувачем роботи до захисту_____

4. Зміст пояснювальної записки за розділами:

Р.1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ.

Р.2 ПРОЄКТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Р.3 РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Р.4 ВИМОГИ, ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ СИСТЕМИ

5. Графічний матеріал за розділами:

Р.1_____

Р.2_____

Р.3_____

Р.4_____

6. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1. Аналіз предметної області та постановка задачі.	30.12.2024
Розділ 2. Проєкт програмного забезпечення	20.01.2025
Розділ 3 Результати розробки програмного забезпечення	15.02.2025
Розділ 4 Вимоги, тестування та валідація	20.04.2025
Остаточне оформлення роботи	10.05.2025
Направлення роботи для перевірки на плагіат	25.05.2025
Попередній захист роботи на випусковій кафедрі	26.05.2025
Направлення роботи на рецензування	26.05.2025

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта	Перевірив	
		дата	підпис
Розділ 1	Рябчун Ю.В.,доц.каф.ІТ	30.12.2024	
Розділ 2	Рябчун Ю.В.,доц.каф.ІТ	20.01.2025	
Розділ 3	Рябчун Ю.В.,доц.каф.ІТ	15.02.2025	
Розділ 4	Рябчун Ю.В.,доц.каф.ІТ	20.04.2025	

8. Дата видачі завдання 02.12.2024 р.

Зав. кафедри			Гончаренко Т.А.
	(підпис)		(прізвище та ініціали)
Керівники			Рябчун Ю.В.
	(підпис)		(прізвище та ініціали)
Здобувач			Цись М.В.
	(підпис)		(прізвище та ініціали)

АНОТАЦІЯ

Цись М.В. Комп'ютерна гра в жанрі Rogue-like на основі ігрового двигуна Unity. Ч.1. Розробка анімації, моделей персонажів та сцен.

Кваліфікаційна випускна робота бакалавра за спеціальністю 122 «Комп’ютерні науки», освітня програма «Інформаційні управляючі системи і технології». – Київський національний університет будівництва та архітектури. – Київ, 2025.

Дана робота присвячена розробці гри в стилі Rogue-like за допомогою ігрового двигуна Unity. Основна увага зосереджена на створенні динамічного ігрового досвіду, що поєднує в собі елементи процедурної генерації рівнів, використання унікальної механіки "жаб’ячого зору" та забезпечення захоплюючого геймплею.

Процедурна генерація рівнів дозволяє створювати непередбачувані карти, що сприяє підвищенню реіграбельності гри. Динамічний ігровий процес забезпечує взаємодію гравця з адаптивним ігровим середовищем, що змінюється в залежності від дій гравця.

У роботі розглядаються етапи проектування, реалізації та тестування гри, з акцентом на інтеграцію основних механік. В рамках проекту створено прототип гри, що демонструє адаптивний ігровий процес, який відповідає стандартам жанру. Результати проекту можуть бути використані для подальшого розвитку гри або адаптації механік у нових проектах.

Розробка має потенціал для подальшого розвитку та може бути цікавою як для гравців, так і для інді-розробників, що шукають нові ідеї для створення ігор.

Робота викладена на …

Ключові слова: гра, жанр, Rogue-like, Unity, НПС (неігровий персонаж), механіки, ігровий процес, геймплей.

SUMMARY

Tsys M.V. Development of a PC game in Rogue-like style. Bachelor's thesis in the specialty: 122 "Computer Science," specialization: "Automation and information technologies." – Kyiv National University of Construction and Architecture. – Kyiv, 2024.

This thesis is dedicated to the development of a Rogue-like game using Unity. The game is based on procedural level generation, frog-vision mechanics and dynamic gameplay.

The work analyzes the key characteristics of the Rogue-like genre, considers technical aspects and selects mechanics according to the genre. Also, the interaction of the player with the NPC (non-player character) is implemented. In the ramps of the project, a prototype of the game was created, demonstrating adaptive gameplay that meets the standards of the genre. The results of the project can be used for further development of the game or adaptation of mechanics in new projects.

Зміст

ВСТУП 8

Розділ 1. 10

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ 10

1.1. Постановка та аналіз проблеми 10

1.2. Огляд літератури та готових існуючих готових рішень 12

1.3. Дерево цілей 23

1.4. Вимоги та особливості проєктування системи 26

Розділ 2. 27

ПРОЄКТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 27

2.1 Ескізний проєкт 27

2.1.1 Контекстна діаграма 28

2.2 Технічний проект 29

2.2.1 Логічна модель 30

2.2.3 Діаграма класів 31

2.2.4 Діаграма діяльності 32

2.3 Робочий проект 32

2.3.1 Вибір засобів розробки 32

Розділ 3. 36

РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМИ 36

3.1 Структура проєкту та сцен 36

3.2 Впровадження механік гри 37

3.3 Огляд механік гри 39

3.3.1 Механіка процедурної генерації рівнів 39

3.3.2. Механіка розміщення сутностей 41

3.3.3. Механіка розміщення декорацій 42

3.3.4. Базова механіка для бойової системи 44

3.3.5. Механіка поведінки сутностей 45

3.3.5.1. Механіки поведінки гобліна 46

3.3.5.2. Механіка поведінки привида. 48
3.3.5.3. Механіка поведінки боса. 50
3.3.5.4. Механіка предметів та інтерактивних об’єктів. 52
Розділ 4. 54
ВИМОГИ, ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ СИСТЕМИ 54
4.1 Вимоги до гри та основні підходи до проєктування. 54
4.2. Основні характеристики, що враховуються при розробці гри. 54
4.2. Опис проєктованого продукту 55
4.3. План виробництва додатку 56
ВИСНОВКИ 59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 61
ДОДАТОК А. Код розробки 64

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Unity – платформа для розробки.

НПС – неігровий персонаж.

Sprite – елемент для відображення зображень в ігровому середовищі.

ПГР – процедурна генерація рівнів;

ГГ – головний герой.

Prefab - це спеціальний тип об’єкта, який дозволяє зберігати готовий шаблон GameObject разом із усіма його компонентами, налаштуваннями та дочірніми об’єктами для багаторазового використання у сцені або в інших проектах.

GameObject — це базовий об’єкт у Unity, який представляє будь-який елемент у сцені (персонаж, світло, камеру, 3D-модель тощо). Сам по собі він не має ніякої функціональності, але може містити компоненти (Components), які надають йому певну поведінку або властивості.

Скрипт — це програмний код, написаний на C#, який визначає поведінку GameObject або взаємодію між об’єктами в грі. Він додається до об’єкта як компонент і може керувати його рухом, анімацією, логікою гри тощо.

Rogue-like – жанр комп'ютерних ігор. Характерними особливостями класичного rogue-like є рівні, покроковість і незворотність смерті персонажа, що генеруються випадковим чином – у разі його загибелі гравець не може завантажити гру і повинен почати її заново.

Метроїдванія — піджанр пригодницького бойовика з набором елементів ігрової механіки та ігровим процесом, подібним до серій Metroid і Castlevania (починаючи з Castlevania: Symphony of the Night). Жанр також відомий під назвами «метроїд», «ігаванія» (на честь геймдизайнера Кодзі Ігарасі), «кастельроїд».

Пригодницький бойовик (англ. Action-adventure) – це жанр відеоігор, який поєднує характерні особливості пригодницьких відеоігор та ігор жанру бойовик.

Ігровий процес, або геймплей (англ. Gameplay) – термін, яким називають особливості взаємодії людини з відеогрою.

Комп'ютерна рольова гра (англ. computer role-playing game , позначається абревіатурою **CRPG** або **RPG**) – жанр комп'ютерних ігор, в якому гравець управляє одним або декількома персонажами, кожен з яких описаний набором чисельних характеристик, списком здібностей та умінь; прикладами таких характеристик можуть бути окуляри здоров'я (англ. hit points, HP), показники сили, спритності, інтелекту, захисту, ухилення, рівень розвитку тієї чи іншої навички тощо.

Завантажуваний контент (англ. downloadable content), який часто скорочується до **DLC** (читається «ді-ел-сі») — додатковий контент, створений для вже випущеної відеоігри , що розповсюджується через інтернет видавцем гри. Він може бути доданий безкоштовно або може бути формою монетизації відеоігор, що дозволяє видавцеві отримувати додатковий дохід від гри після її покупки, часто використовуючи будь-який тип системи мікротранзакцій .

Фентезі (англ. **fantasy** – фантазія) – піджанр фантастики – одного з жанрів сучасного мистецтва, дія якого відбувається у вигаданому світі, де чудеса і вигадка нашого світу є реальністю. Світ є подібним до Середньовіччя, властивою основою світу є магія, він наповнений чарівними істотами, оповідь епічна, тому час зациклений і йде по колу (є зміна пір року, однак нема історичного розвитку епохи). Ці риси визначальні для фентезі і поєднують жанр із літературною казкою.

Темне фентезі (англ. **dark fantasy** - «темне, похмуре фентезі») – піджанр художніх творів, що включає елементи жахів і готики , дія в якому відбувається в антуражі традиційного фентезі.

Світле фентезі (англ. **light fantasy**) – повна протилежність темному фентезі.

Дисбаланс (англ. **imbalance**) – незбалансований персонаж, артефакт або інший ігровий об'єкт - зазвичай занадто сильний.

Бафф (англ. **buff**) – поняття в комп'ютерних іграх , що означає тимчасове посилення гравця, як правило, під дією спеціального заклинання.

Дебафф (англ. **debuff**) – відноситься будь-який негативний вплив на гравця або НПС, відмінне від прямого нанесення шкоди.

Action-RPG, рольова гра в жанрі «екшн» або рольовий екшн (англ. action role-playing game, action RPG, ARPG) – піджанр комп'ютерних ігор, в якому поєднуються ключові елементи жанрів екшн і рольових ігор.

Ізометрична проекція (др.-грец. ἴσος «рівний» + μετρέω «вимірюю») – це різновид аксонометричної проекції , при якій у відображенні тривимірного об'єкта на площину коефіцієнт спотворення (відношення довжини спроектованої на площину відрізка) по всіх трьох осях один і той же.

Аксонометрична проекція (від грец. ἄξων «вісь» + μετρέω «вимірюю») – спосіб зображення геометричних предметів на кресленні за допомогою паралельних проекцій.

Біом – в геймінгу означає конкретну локацію чи декілька локацій, які об’єднані певними ігровими аспектами (дизайн, стиль і т.д.).

Рандом – дія, виконана не за чітким планом, випадковість.

Спавн – поява НПС, предмету чи іншого явища грі, яка контролюється або не контролюється гравцем.

ВСТУП

Ігрова індустрія є однією з найдинамічніших та найскладніших галузей інформаційних технологій. Створення комп'ютерних ігор вимагає не лише технічних знань, але й глибокого розуміння жанрових особливостей, ігрових механік та потреб користувачів. Серед великої кількості жанрів, ігри типу Rogue-Like займають окреме місце завдяки своїй процедурній генерації рівнів, високій реіграбельності та добре реалізованими механіками прогресу.

Unity є однією з найпопулярніших платформ для розробки ігор завдяки своїй гнучкості, широкому набору інструментів та можливості створення багатоплатформених проектів. Застосування Unity для розробки ігор жанру Roguelike дозволяє реалізувати складні механіки, такі як генерація рівнів, управління персонажами, взаємодія об’єктів та оптимізація ігрового процесу.

Метою даної роботи є розробка прототипу комп’ютерної гри в жанрі Rogue-like за допомогою Unity, яка забезпечує захоплюючий ігровий досвід за рахунок використання процедурної генерації рівнів, механіки "жаб’ячого зору" та динамічного ігрового процесу.

Завдання дослідження:

1. Проаналізувати жанрові особливості ігор типу Rogue-like та виділити ключові елементи їх геймплею.
2. Дослідити можливості Unity для реалізації механік, характерних для жанру.
3. Розробити концепцію гри, архітектуру прототипу з урахуванням принципів процедурної генерації.
4. Реалізувати основні ігрові механіки в середовищі Unity.
5. Провести тестування прототипу та оцінити його відповідність заданим вимогам.

Об’єкт дослідження – процес створення комп'ютерної гри в жанрі Rogue-like із застосуванням сучасних інструментів розробки.

Предмет дослідження – методи та інструменти реалізації ігрових механік у жанрі Rogue-like на платформі Unity.

Розділ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Постановка та аналіз проблеми

Ігрова індустрія є невід'ємною частиною сучасного цифрового світу. За статистикою кожна 4 людина являється геймером, тому важливість розробки ігор та нового контенту для них є вкрай актуальною. Ігри не лише забезпечують розваги, але й стають важливим інструментом для навчання, соціалізації та творчого самовираження. Сучасні тенденції у геймінгу орієнтовані на інновації, залучення користувачів за допомогою реалістичної графіки, інтерактивності та динамічних ігрових сценаріїв.

Жанр Rogue-like є популярним напрямом у розробці комп'ютерних ігор завдяки своїй динамічності, реіграбельності та глибокому геймплею [4]. Проте створення якісної гри в цьому жанрі ставить перед розробниками низку проблем, які потребують ретельного опрацювання.

Основні аспекти проблеми:

1) Процедурна генерація рівнів: розробка алгоритмів, які забезпечують створення унікальних рівнів із заданими характеристиками (логічні структури, баланс складності тощо). Основна проблема – збереження ігрового балансу при використанні випадкових елементів [1].

2) Механіка геймплею: унікальні механіки, такі як "блінк" (телепортування), мають бути інтегровані так, щоб не тільки додавати складності, але й створювати цікавий та інтуїтивно зрозумілий ігровий досвід [2].

3) Оптимізація гри: ігри жанру Rogue-like часто включають великий обсяг динамічних об'єктів та складні алгоритми, що може призводити до зниження продуктивності, особливо на слабких пристроях. Оптимізація є критично важливою для забезпечення стабільності гри [3].

4) Динаміка та реіграбельність: гра повинна залишатися цікавою навіть після багатьох проходжень. Це вимагає збалансованого поєднання процедурної генерації, адаптивного дизайну ворогів і інтерактивності ігрового середовища [4].

5) Unity як платформа: Unity є потужним інструментом для створення ігор, робота з ним потребує глибокого знання його функціоналу, зокрема обробки анімацій, фізики та скриптової архітектури [5].

Аналіз проблеми: сучасні ігрові технології дозволяють значно спростити процес створення ігор, але залишаються відкритими питання забезпечення ігрової унікальності та продуктивності. Rogue-like ігри потребують нестандартних підходів до дизайну та програмування, щоб задовольнити високі очікування аудиторії.

Розв'язання цих проблем у межах роботи включатиме:

- розробку та інтеграцію інноваційних алгоритмів процедурної генерації;
- дизайн ігрових механік, які урізноманітнюють ігровий процес;
- проведення оптимізації гри.

Вирішення цих задач дозволить створити якісний продукт, що поєднує інноваційні технології та цікаву ігрову механіку.

Отже, **завданням кваліфікаційної роботи буде** розробити однокористувацьку гру в стилі Rogue-like, яка забезпечує випадково генеровані рівні, складність, що збільшується з кожним рівнем, та елементи стратегії і виживання. Гравець має пройти через серію процедурно згенерованих рівнів, борючись із ворогами, збираючи предмети та ресурси, розвиваючи свого персонажа і приймаючи рішення, що впливають на його подальший прогрес.

Основні вимоги до гри:

- *Процедурна генерація рівнів:*

1) Рівні мають генеруватися випадковим чином, що гарантує унікальність кожного проходження.

2) Процес генерації має включати випадкові об'єкти, ворогів, пастки або інші елементи.

- *Персонаж гравця:*

1) Гравець керує персонажем, який має певні параметри.

- *Бойова система:*

1) Бойова система повинна реалізована в реальному часі.

2) Вороги повинні мати різну складність.

- *Система смерті:*

1) При смерті персонажа гра повинна починатися спочатку.

Вхідні дані:

- Ініціалізація гри;
- Параметри персонажа;

Динамічні вхідні дані:

- Дії гравця;
- Взаємодія з об’єктами;
- Взаємодія з ворогами;
- Генерація рівнів;

Вихідні дані:

- Інтерфейс користувача;
- Результати взаємодії користувача з грою;

Збереження прогресу (якщо буде передбачено грою).

1.2. Огляд літератури та готових існуючих готових рішень

Проаналізувавши список літератури, який пропонує готові рішення для проблем, що можуть виникнути під час розробки гри, відзначили:

1) Unity Documentation – докладні матеріали для роботи з усіма компонентами Unity. Найголовніший посібник, який допоможе коректно використовувати інструменти та можливості, які дає Unity [6].

2) Game Programming Patterns – книга пояснює, як правильно організовувати код у ігрових проектах для підвищення його зрозумілості [7]. Автор розглядає типові проблеми, що виникають при розробці ігор, і пропонує рішення на основі перевірених патернів проектування.

3) The User Experience Team of One: A Research and Design Survival Guide – основа ідея книги полягає в тому, що гра має бути зручною для геймера, а не для

розробника [8]. Книга акцентує увагу на важливості UX у розробці ігор і надає рекомендації, як зробити гру зручною, захоплюючою та доступною для широкої аудиторії. Автор вивчає, як гравці взаємодіють із грою, що сприяє їхньому залученню, а що може викликати розчарування.

4) Procedural Content Generation for Unity Game Development – книга спеціально орієнтована на Unity і містить численні приклади та алгоритми для створення процедурно згенерованого контенту, від карт до текстур [9].

5) Procedural Generation in Game Design – книга є обов'язковою для аналізу разом із попередньою, оскільки може відповісти на питання, які можуть виникнути під час вивчення попередньої [10]. В цілому матеріал схожий, але попередня книга описує алгоритми та методи їх реалізації, в той час як ця книга дає розуміння того, як ці алгоритми мають працювати.

6) Rules of Play: Game Design Fundamentals – книга охоплює основи дизайну ігор, зокрема аспект балансу. Вона допомагає розуміти, як правильно налаштовувати ігрові механіки для досягнення хорошого балансу між різними елементами гри [11]. Сама книга не присвячена жанру Rogue-like, але дає загальне розуміння, як має працювати баланс в іграх.

7) Джеремі Гібсон Бонд – "Introduction to Game Design, Prototyping, and Development". Відмінний вступ до розробки ігор, що охоплює геймдизайн, програмування (C#) та роботу з Unity [12].

8) Джессі Шелл – "The Art of Game Design: A Book of Lenses". Одна з найкращих книг з геймдизайну, яка допоможе зрозуміти, як створювати цікаві й захоплюючі ігри [13].

9) Девід Еберлі – "3D Game Engine Design". Гарне введення в роботу з 3D-графікою, фізикою та архітектурою рушіїв [14].

10) Харрісон Ферроне – "Learning C# by Developing Games with Unity". Вчить C# у контексті розробки ігор в Unity, що робить його чудовим ресурсом для початківців [15].

Всі ці книги напряму не допомагають розробити гру, але дають розуміння того, як уникнути основних проблем під час розробки гри та після неї, як правильно реалізувати свої ідеї та досягти бажаного результату.

Для успішної реалізації проєкту, а саме гри в жанрі Rogue-lite, можна проаналізувати успіх ігор в цьому ж стилі. Для порівняння візьмемо найкращі проекти на ринку:

- Dead Cells [16].
- Darkest Dungeon [17].
- The Binding of Isaac [18].
- Hades [19].

На перший погляд ці проєкти абсолютні різні. Стилістика, механіка, поєднання ігрових жанрів. Але всі вони є кращими з кращих *рогаликів* (Rogue-like на сленговій мові).

Dead Cells [16] – комп'ютерна інді-гра в змішаному жанрі roguelike та метроідванії, розроблена та випущена французькою студією Motion Twin для платформ Windows, MacOS та Linux, ігрових консолей Nintendo Switch, PlayStation 4 та Xbox One у 2018 році (рис.1).

Рисунок 1– Логотип гри Dead Cells

Геймплей: гравець керує В’язнем, аморфною істотою, яка мандрує островом, повним мутованих монстрів. Коли гравець помирає, він втрачає всю зброю та покращення, отримані під час проходження, за винятком кількох постійних предметів. До зброї в першу чергу належать мечі, луки, щити та пастки, які можна встановити, щоб завдати шкоди ворогам, які наближаються до них. Під час бою Полонений може ухилятися по землі (пригинатися), щоб уникнути атак ворогів, або перестрибувати через атаки.

Переваги:

- *Інтерфейс.* Гра має максимально простий та зрозумілий інтерфейс, що дає гравцю можливість швидко та легко відслідковувати показники персонажа та його предмети.
- *Графіка.* Змішаний піксельний та light-fantasy стиль відразу кидаються в очі та дуже приваблюють гравця. Поєднання світлових ефектів з елементами середовища та мобами створюють ефектну картинку.
- *Механіки.* Поєднання різних елементів спорядження та навичками персонажа створюють великий простір для гравця.
- *Сюжет.* Сенс гри в не безцільному проходженню рівнів. Тут є найголовніше для хорошої гри – сюжет. Коротко про сюжет: «Ув'язнений прокидається в глибині в'язниці острова, страждаючи від амнезії. Солдат зустрічає Полоненого і каже, що вони більше не можуть померти. Ув'язнений намагається втекти з в'язниці, але його свідомість повертається на глибину, щойно його тіло знищується. Між наступними спробами втечі В’язень дізнається, що острів колись був могутнім королівством, яке занепало, коли чума, відома як «Недуга», перетворила більшість громадян королівства на мутованих монстрів.»

Недоліки:

- *Складність.* Гра є складною для входження новачків. Хоча варто відмітити, що Dead Cells не позиціонує себе як простий *рогалик.* Кожен рівень, кожен ворог, кожна пастка це виклик для нового гравця, який не знайомий з іграми в схожому жанрі.

Варто відмітити, що Dead Cells здобув свою популярність завдяки своїй складності, а не лише візуальним ефектам та увагою розробників до деталей.

Darkest Dungeon [17] – комп'ютерна рольова гра з roguelike-елементами, розроблена та випущена незалежною канадською студією Red Hook Studios (рис.2).

Рисунок 2 - Логотип гри Darkest Dungeon

Геймплей: гравець управляє групою шукачів пригод, які обстежують різні місця на території старовинного маєтку. Перед кожним походом гравець повинен вибрати чотирьох героїв з казарми і провести їх через генеровані випадковим чином (крім тих, що заскриптовані) підземелля, збираючи скарби, фамільні реліквії і знищуючи різних чудовиськ, що зустрічаються на шляху.

Перед гравцем стоїть завдання виконати умову завдання, і за бажанням, зберегти життя та психічне здоров'я героїв. Рівень психологічної напруги (стресу) зростає щоразу, коли герої отримують особливі удари від ворогів, потрапляють у пастки, взаємодіють із деякими дивовижами, відступають із поля бою чи невдало відпочивають у таборі; якщо у бійця здають нерви, він впадає в один із безлічі психозів, починаючи поводитися неадекватно, самовільно виконуючи будь-яку дію в бою, атакуючи себе, іншого героя, або наганяючи на всіх стрес, майже завжди позбавляючи гравця можливості віддати наказ. По ходу дослідження підземель персонажі набувають різних рис характеру - постійні особливості поведінки: як негативні (алкоголізм, клептоманія), так і позитивні, що роблять героя більш хоробрим і витривалим (німфоманія, смертоносність). Вороги в Darkest Dungeon

належать до певних класів, рас або типів. Деякі риси характеру героїв пов'язані з класами супротивника. Позитивні риси характеру можуть давати герою бонуси (наприклад, збільшення втрат, точності чи шансу завдання критичної шкоди), коли той стикається з ворогами певних класів. Тим часом негативні риси можуть накладати дебаффи (повністю дзеркальні бафам) при зіткненні з ворогами певних класів. Крім того, є також певні навички героя та прикраси, що дають бонуси проти деяких ворожих класів.

Переваги:

- *Графіка.* Традиційне темне фентезі, яке затягує гравця темними тонами та готичним стилем. Стилістика, яка найкраще описує цю гру та все, що в ній відбувається.
- *Механіки.* Велика реалізована кількість механік, кожна з яких є невід’ємною частиною ігрового процесу. Відчувається, що розробники провели велику роботу щоб це виглядало гармонічно та не відчувалося *дисбалансу*.
- *Ігровий процес.* Реалізація покрокового режиму гри є великою перевагою, оскільки недосвідчений гравець матиме більше часу для роздумів та оцінки ситуації на полі бою. Цей варіант ігрового процесу робить поріг для входження новачків меншим, що дає грі більше шансів привабити нову аудиторію.

Недоліки:

- *Складність.* Навіть враховуючи переваги ігрового процесу, гра є досить складною через кількість реалізованих механік (час горіння факелу, голод, темрява, страх та інше).

The Binding of Isaac [18] – це пригодницька гра 2011 року з rogue-like елементамии, створена незалежними розробниками Едмундом Макмілленом і Флоріаном Хімслом. Спочатку він був випущений для Microsoft Windows, потім перенесений на OS X і Linux. Гра стала результатом тижневої гри між Макмілленом і Хімслом, щоб розробити рогалик, натхненний The Legend of Zelda , що дозволило Макміллену продемонструвати свої почуття щодо як позитивних,

так і негативних аспектів релігії, які він відкрив під час конфліктів між члени його католицької та народженої знову християнської сім’ї під час дорослішання (рис.3).

Рисунок 4 - Логотип гри The Binding of Isaac

Геймплей: гра в підземеллях, що ведеться зверху вниз, представлена за допомогою двовимірних спрайтів, у яких гравець керує Ісааком або іншими персонажами, яких можна розблокувати, коли вони досліджують підземелля, розташовані в підвалі Ісаака. Персонажі відрізняються за швидкістю, кількістю здоров'я, кількістю завданої шкоди та іншими атрибутами. Механіка та презентація гри схожі на підземелля *Легенди про Зельду*, але включають випадкові, процедурно згенеровані рівні на манер гри Rogue-like. На кожному поверсі підземелля гравець повинен битися з монстрами в кімнаті, перш ніж перейти до наступної кімнати. Найчастіше це робиться сльозами персонажа, які є кулями в стилі *стрілялки*, але гравець також може використовувати обмежений запас бомб, щоб завдати шкоди ворогам і зачистити частину кімнати. Інші способи перемоги над ворогами стають можливими, коли персонаж отримує посилення (предмети), які автоматично носить персонаж-гравець, коли їх підбирає, що може змінити основні атрибути персонажа, такі як збільшення здоров’я або сила кожної сльози, або спричинити додаткові побічні ефекти, наприклад, дозволити стріляти

зарядженими пострілами після короткочасного утримання кнопки контролера. Бонуси включають пасивні предмети, які автоматично покращують атрибути персонажа, активні бонуси, які можна використати один раз, перш ніж вони будуть перезаряджені шляхом проходження додаткових кімнат у підземеллі, і одноразові бонуси, такі як таблетки або карти Таро, які надають одноразову вигоду при використанні, наприклад відновлення повного здоров’я або збільшення чи зменшення всіх атрибутів персонажа. Посилення накопичуються, щоб гравець міг складати дуже вигідні комбінації.

Після того, як кімната очищена від монстрів, вона залишатиметься вільною, дозволяючи гравцеві повторно простежити свій шлях через рівень, але коли вони перейдуть на наступний рівень, вони не зможуть повернутися. По дорозі гравець може збирати гроші, щоб купувати бонуси від власників магазинів, ключі для відкриття спеціальних кімнат зі скарбами, а також нову зброю та бонуси, щоб посилитись. Здоров'я гравця відстежується кількістю сердець; якщо персонаж втрачає всі свої серця, гра закінчується остаточною смертю і гравець повинен почати все спочатку зі щойно згенерованого підземелля. На кожному поверсі підземелля є бос, якого гравець повинен перемогти, перш ніж перейти на наступний рівень. На шостому з восьми поверхів гравець бореться з матір'ю Ісаака. Пізніші рівні значно складніші, а кульмінацією стає боротьба проти серця матері Ісаака на восьмому поверсі. Додатковий дев'ятий поверх - Шеол, містить боса Сатану. Перемога в грі з певними персонажами або за певних умов відкриває нові бонуси, які можуть з’явитися в підземеллі, або можливість використовувати одного з інших персонажів. У грі відстежуються різноманітні бонуси, які гравець знайшов з часом, які можна переглянути в меню гри.

Переваги:

- *Контент.* Неймовірно велика кількість контенту, що не дасть гравцю занудьгувати перші 100 годин проходження гри.
- *Механіки.* Кожен предмет, карта, чи пігулка вносять в *забіг* (сленгове слово, що означає повторне проходження рівня) створюють абсолютно унікальні

комбінації атаки, додаючи нові можливості для гравця. Для повного відкриття та вивчення існуючих бонусів знадобиться не одна сотня годин.

- *Сюжет.* Неймовірно цікава річ в цій грі. Відкривається поступово, що дає гравцю мотивацію проходити рівні за різних героїв на різних рівнях складності. Сюжет дуже частково натхненний однойменною біблійною історією: Ісаак, дитина, та його мати живуть у маленькому будиночку на пагорбі, щасливо тримаючись усамітнено, при цьому Ісаак малює малюнки та грається своїми іграшками, а його мати дивиться християнські трансляції по телебаченню. Одного разу мати Ісаака чує «голос згори», який, на її думку, є голосом Самого Бога. Голос заявляє, що її син Ісаак «зіпсований гріхом і потребує порятунку». Він просить її усунути від Ісаака все зло і його мати погоджується, забираючи його іграшки, малюнки і навіть одяг. Голос звертається до матері Ісаака вдруге, заявляючи, що Ісаак має бути відрізаний від усього зла у світі. Знову його мати погоджується і замикає Ісаака в його кімнаті. Востаннє голос звертається до матері Ісаака. Там стверджується, що вона вчинила добре, але все ще ставить під сумнів її відданість і каже їй пожертвувати сином. Вона погоджується, хапає м'ясний ніж з кухні та йде до кімнати Ісаака, готуючись принести його в жертву. Ісаак, дивлячись крізь велику щілину у дверях, починає панікувати. Він знаходить люк, захований під килимом, і стрибає туди, коли його мати вривається в двері його спальні. З цього самого моменту починається *забіг*.

Недоліки:

- *Одноманітність.* Хоч гра і має велику кількість контенту, все це зводиться до повторного проходження рівнів, що може негативно вплину на зацікавленість геймерів до повторного проходження гри.

Hades [19] – комп'ютерна гра в жанрах Rogue-like і action-RPG, розроблена та випущена американською студією Supergiant Games . Гра була випущена в ранній доступ у 2018 році як тимчасовий ексклюзив для запущеного тоді ж сервісу Epic Games Store. Повна версія для Windows, MacOS і Nintendo Switch була випущена в 2020 році (рис.4).

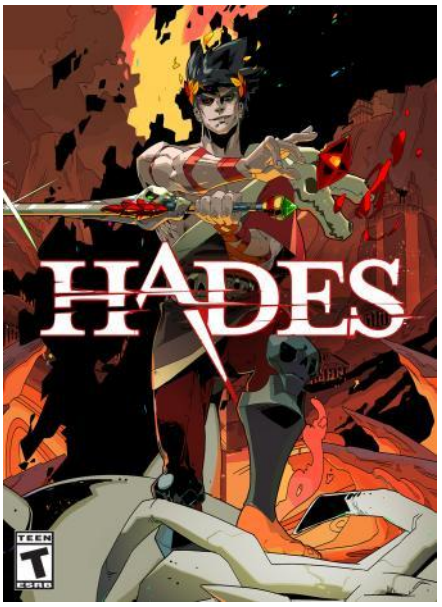

Рисунок 4 - Логотип гри Hades

Геймплей: Hades використовує двовимірну графіку в ізометричній проекції. Керований гравцем герой - бог Загрей, знаходиться у центрі екрану. У кожному проходженні гравець намагається пройти крізь довгу послідовність кімнат; при цьому порядок кімнат, вороги і нагороди, що з'являються в них, визначаються випадковим чином, так що жодне проходження не повторює попереднє. У грі представлені чотири області царства мертвих (біома), у кожній з яких гравець стикається з новими ворогами та небезпеками: Тартар , Асфодель , Елізій та Храм Стіксу. У битвах із ворогами Загрей використовує різну зброю, наприклад, меч, лук або кристал, який він може метати з великої відстані.

Після перемоги над усіма ворогами у кімнаті Загрей отримує відповідну нагороду та може перейти до наступної кімнати. Піктограми на дверях, які ведуть до наступних кімнат, вказують, яка в них нагорода чекає; персонаж може пройти тільки в одні двері. В нагороду Загрей може отримати дари олімпійців: посилення чи нові здібності, що тематично пов'язані з певним богом. Наприклад, Зевс, що наказує блискавками, може дарувати здатність вражати ворогів електричним розрядом.

У грі є безліч інших можливих нагород: так, «темрява» дозволяє назавжди збільшити характеристики героя між проходженнями, монети («оболи Харона») можна витратити на цінні предмети в магазині Харона, молоти Дедала покращують зброю персонажа тощо. Деякі предмети та здібності — дари олімпійців, оболи чи молоти — втрачаються після кожної чергової смерті Загрею, інші — як «темрява», ключі та самоцвіти — переходять у наступні проходження, і їх можна витратити на покращення характеристик чи розблокування нових видів зброї у палаці Аїда.

Переваги:

- *Графіка.* Гра має свою неповториму стилістику та атмосферу. Царство Аїду в якому опиняється ГГ може заворожити кожного геймера своїми пейзажами, а стилістика в поєднанні з грецькою архітектурою робить гру унікальною в своєму стилі.
- *Геймплей.* Максимально простий геймплей та легкість адаптації до динаміки гри робить поріг входження досить низьким, що дає змогу привабити більшу кількість аудиторії.
- *Варіативність.* Процедурна генерація рівнів та випадковий спавн роблять кожен забіг унікальним.

Недоліки:

- *Одноманітність.* Проблема одноманітності переслідує багато ігор в стилі Rogue-like. Hades не стала виключенням. Тому гра більше орієнтується на поціновувачів цього жанру, та людей, які люблять долати виклики, які їм підносять.

1.3. Дерево цілей

Дерево цілей – це інструмент для ієрархічної організації та визначення основних та підпорядкованих цілей для проекту або системи.

Основна мета кваліфікаційної роботи – розробка прототипу комп’ютерної гри в жанрі Rogue-like за допомогою Unity, яка забезпечує захоплюючий ігровий

досвід за рахунок використання процедурної генерації рівнів, механіки "жаб'ячого зору" та динамічного ігрового процесу.

Ця мета передбачає створення гри, яка:

1. Демонструє високий рівень технічної реалізації та стабільності.
2. Забезпечує реіграбельність завдяки процедурному створенню ігрових середовищ.
3. Пропонує унікальний ігровий досвід через використання нових механік і підходів.
4. Відповідає сучасним вимогам до продуктивності та якості на різних платформах.

На рисунку 5 представлено дерево цілей.

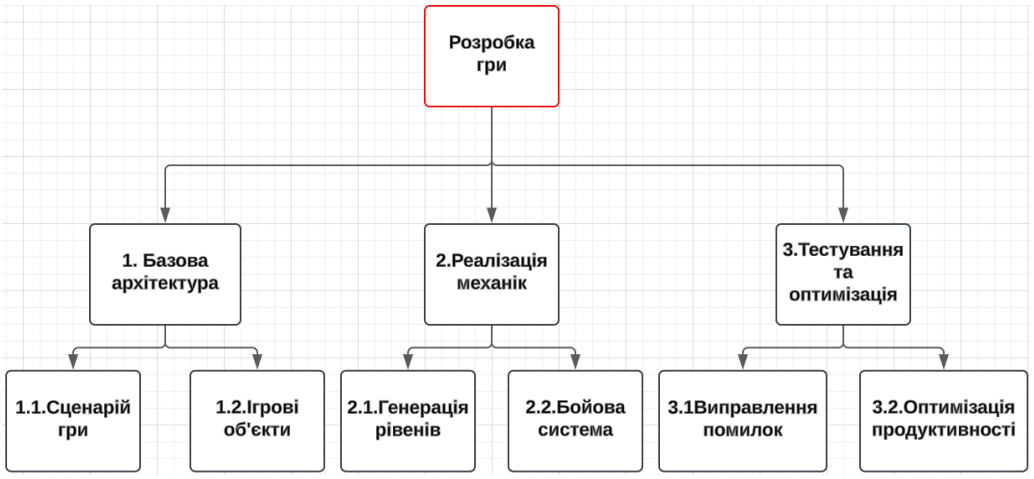

Рисунок 5 – Дерево цілей

Для кращого розуміння дерева цілей детально розглянемо його.

1. Базова архітектура – фундамент, на якому будується вся гра. Вона охоплює основні елементи структури проекту, взаємодію об'єктів, механізми гри та принципи організації коду.

1.1. Сценарій гри - пункт включає в себе не сюжет, сценарій гри охоплює всю гру. Від моменту запуску до моменту виходу. Сценарій – це головне меню, ігровий процес, завершення гри, рівні, предмети, локації, система бою, система інвентаря, система прогресу, це те, як має діяти гравець з моменту запуску гри.

1.2. Для реалізації сценарію потрібна не менш важлива річ – **ігрові об’єкти**. До ігрових об’єктів відносяться:

1) Гравець (параметри, управління);
2) НПС – неігровий персонаж (параметри, ШІ);
3) Предмети (зброя, засоби лікування і т.д.);
4) Рівні.

2. Реалізація механік.

Механіки в грі – це набір правил, принципів і систем, які визначають, як гравець взаємодіє з ігровим середовищем, персонажами та об'єктами. Вони формують основу ігрового процесу (геймплею).

2.1. Генерація рівнів. Особливість Rogue-like проектів є процедурна генерація рівні.

Процедурна генерація рівнів – це техніка автоматичного створення ігрових рівнів за допомогою алгоритмів, що дозволяє генерувати унікальні карти, кімнати або середовища кожного разу, коли гравець запускає гру.

Переваги процедурної генерації:

- унікальність кожного проходження;
- оптимізація часу розробки;
- масштабованість;
- варіативність складності.

2.2. Бойова система – це механіка гри, яка визначає, як гравець взаємодіє з ворогами або супротивниками, включаючи правила, способи атак, захисту, використання здібностей і тактичних дій. Це один із ключових елементів ігрового процесу, який впливає на динаміку гри.

Це основний етап, який формує кістяк гри. Для впровадження адекватної системи бою потрібно спершу розробити баланс, при якому гравець не буде надто сильним, та надто слабким, а також провести математичні обрахунки для розрахування характеристик ГГ, НПС, предметів та інших речей.

3. Тестування та оптимізація.

1) Модульне тестування:

- При проведенні тестування перевірити окремі механіки (генератор рівнів, бойова система, штучний інтелект ворогів).

2) Інтеграційне тестування:

- Провести аналіз взаємодії систем (розташування ворогів, згенерованих предметів);
- Провести аналіз колізій при розташуванні об’єктів та сутностей.

3.1 Виправлення помилок.

1) Коригування початкових параметрів:

- Відкоригувати параметри генерації рівнів;
- Відкоригувати параметри розташування сутностей;
- Відкоригувати параметри розташування інтерактивних об’єктів;
- Відкоригувати швидкість сутностей;
- Відкоригувати шкоду сутностей;
- Відкоригувати здоров’я сутностей;
- Відкоригуавти параметри патрулювання сутностей;
- Відкоригувати параметри агресії сутностей;

3.2 Оптимізація продуктивності.

1) Оптимізація коду;
2) Оптимізація рендерингу;
3) Управління пам’яттю;
4) Оптимізація анімацій;
5) Оптимізація UI.

2.1. Вимоги та особливості проєктування системи

Проєктування системи – це складний процес, що включає розробку структури, компонентів і функцій системи для досягнення певних цілей.

Вимоги до системи:

1) Функціональні вимоги

- Система ідентифікації гравців для збереження прогресу.

- Механіка досягнень та винагород.

2) Інтерфейс:

- Інтуїтивно зрозумілий і зручний дизайн для ПК.
- Можливість налаштування управління та візуальних елементів під уподобання гравців.

Особливості проєктування системи:

3) Розширюваність

- Гнучке проектування для можливості подальшого впровадження нових функцій.

4) Адаптивність

- Дизайн, оптимізований для гри на моніторах з різним розширенням та у різних форматах (повноекранний, віконний).

5) Висока продуктивність

- Враховуючи мінімалістичність гри, а саме інтерфейс та графіку, можна зробити висновок, що суттєвого навантаження на ПК не буде.

6) Моніторинг та підтримка

- Реалізація сервісу для відстеження та реагування на проблеми користувачів.

7) Регулярні оновлення

- Планування та впровадження регулярних оновлень гри, для утримання та підняття інтересу користувачів.

Розділ 2. ПРОЄКТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розроблюваний програмний продукт являє собою комп’ютерну гру у жанрі **Rogue-like**, створену на основі ігрового рушія **Unity**. Гра передбачає випадково згенеровані рівні, реальне бойове зіткнення, систему розвитку персонажа та елементи РПГ (рольової гри).

Головна особливість гри – **процедурна генерація рівнів**.

Для реалізації гри використано такі технології:

- **Ігровий рушій:** Unity
- **Мова програмування:** C#
- **Графіка:** 2D-спрайти, створені в Adobe Photoshop
- **Фізика та колізії:** Unity Physics
- **Обробка вхідних даних:** Unity Input System

2.1 Ескізний проєкт

Ескізний проєкт – це початковий етап проєктування, на якому визначаються загальні концепції, структура та основні характеристики програмного чи технічного продукту. Він є проміжним етапом між ідеєю та детальним проєктуванням.

Основні цілі ескізного проєкту:

- Визначення загальної архітектури та функціональності.
- Формування основних технічних вимог.
- Попередня оцінка складності та ресурсів для реалізації.
- Візуалізація основних компонентів (наприклад, блок-схеми, макети інтерфейсу).
- Обговорення з командою або замовником перед початком детального проєктування.

Ескізний проєкт для гри:

1. **Опис гри**

- Жанр, основні механіки, стиль гри.
- Головні особливості (процедурна генерація рівнів, бойова система).

2. **Архітектуру гри**

- Основні модулі гри (меню, рівні, механіка бою).
- Блок-схеми або UML-діаграми структури гри.

3. **Ескізи графіки та інтерфейсу**

- Чорнові малюнки або макети ігрового інтерфейсу.
- Основні елементи UI.

4. **Попередня технічна специфікація**

- Вибір ігрового рушія (Unity), мови програмування (C#).
- Список основних алгоритмів (генерація рівнів, поведінка ворогів).

5. **Попередня оцінка ресурсів**

- Необхідні фахівці (програмісти, художники, тестувальники).
- Оцінка часу та складності реалізації.

Для проєктування ескізу гри було обрано мову моделювання UML.

2.1.1 Контекстна діаграма

Контекстна діаграма - це початкова діаграма моделі, що характеризує зв’язки системи з навколишнім середовищем (дивитись рисунок 2.1).

Рисунок 2.1 – Контекстна діаграма системи

2.1.2 Діаграма станів

Діаграма станів — діаграма, що визначає зміну станів об'єкту у часі, одна з діаграм моделювання поведінки. На рисунку 2.12 зображено діаграму станів НПС (неігрового персонажа).

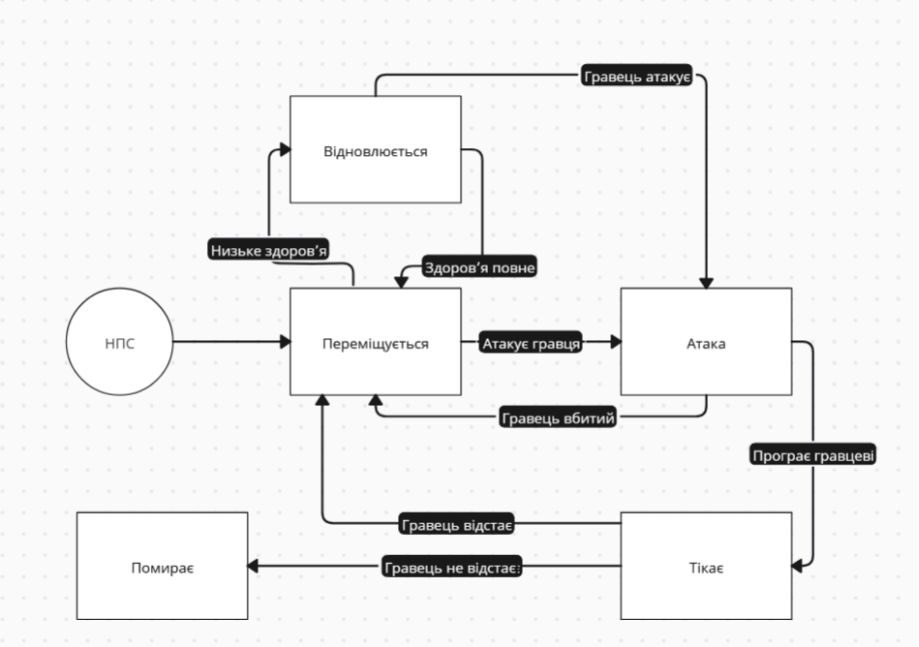

Рисунок 2.12 – Діаграма станів НПС

2.2 Технічний проект

Технічний проєкт – це детальна документація, що містить усі технічні аспекти розроблюваного програмного чи апаратного продукту. Він є наступним етапом після ескізного проєкту та включає структуру системи, технології, алгоритми, інтерфейси, взаємодію компонентів та інші технічні деталі.

Технічний проект є наступним етапом у процесі розробки програмного забезпечення, після ескізного проекту. На цьому етапі детально описуються всі технічні аспекти програми, включаючи її функціональність, структури даних, бази даних та алгоритми обробки даних.

2.2.1 Логічна модель

Логічна модель даних (ЛМД) – це представлення структури даних у вигляді взаємопов'язаних сутностей та атрибутів без прив’язки до конкретної СУБД чи фізичного рівня зберігання. Логічна модель даних сутностей та елементів представлена на рисунку 2.13.

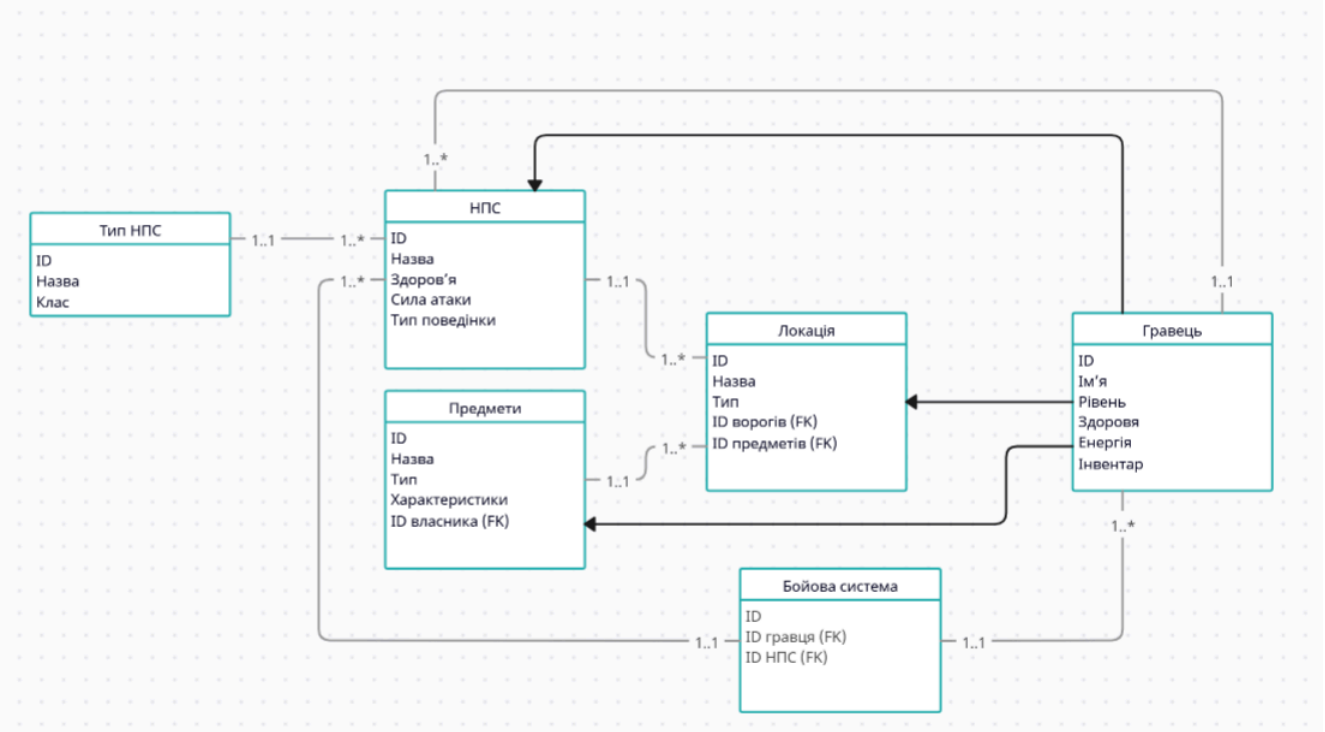

Рисунок 2.13 – Логічна модель даних

На цій моделі відображено логічні зв'язки між основними об'єктами гри: гравцем, локаціями, НПС (ворогами), та предметами.

Що відбувається у грі згідно з моделлю?

1. Гравець знаходиться в певній локації (зв'язок 1:1 між Гравець та Локація).

- Гравець може переміщуватися між різними локаціями.
- У кожній локації він може взаємодіяти з НПС та предметами.

2. Кожна локація містить ворогів (НПС) та предмети (зв'язки 1:*).

- У локації може бути кілька ворогів чи NPC з різним типом.
- Локація також може містити різні предмети.

3. НПС мають власні характеристики: здоров'я, силу атаки, тип.

- Гравець може вступати в бій із НПС або взаємодіяти з ними.

2.2.3 Діаграма класів

Діаграма класів – діаграма, що демонструє класи системи, їх атрибути, методи і взаємозв'язок між ними. Класи з атрибутами та методами представлені на рисунку 2.14.

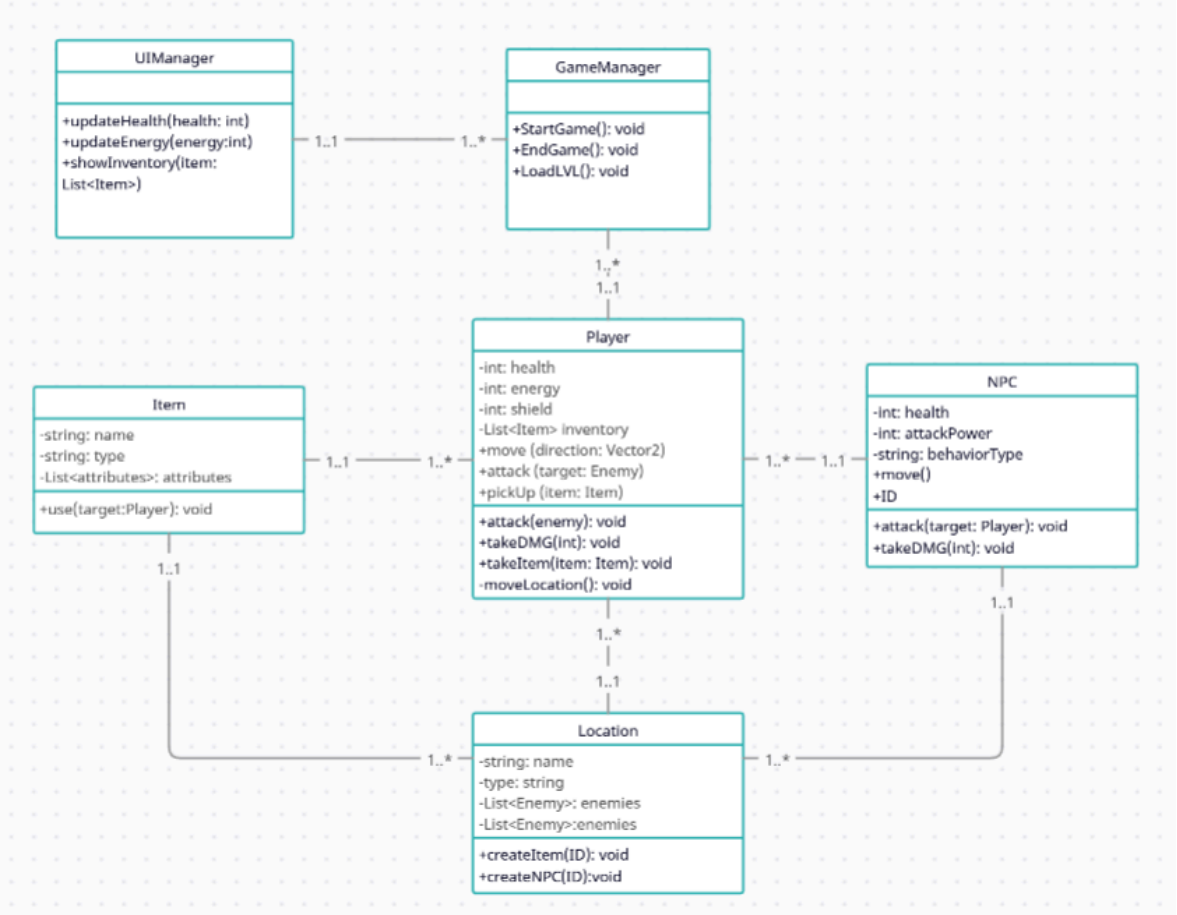

Рисунок 2.14 – Діаграма класів

Аналіз взаємодій між класами:

- Гравець може взаємодіяти з локаціями, NPC, брати предмети та атакувати ворогів.
- NPC можуть атакувати гравця і навпаки.
- GameManager керує загальним процесом гри.
- UIManager відображає оновлення для гравця.
- Локація містить ворогів та предмети, які можуть бути створені або взяті.

2.2.4 Діаграма діяльності

Діаграма діяльності - це блок-схема для представлення потоку від однієї діяльності до іншої. Діяльність можна описати як роботу системи. Основне призначення діаграм активності - відобразити динамічну поведінку системи. Її також називають об’єктно-орієнтованою блок-схемою. На рисунку 2.15 зображено блок-схему алгоритму проходження гри.

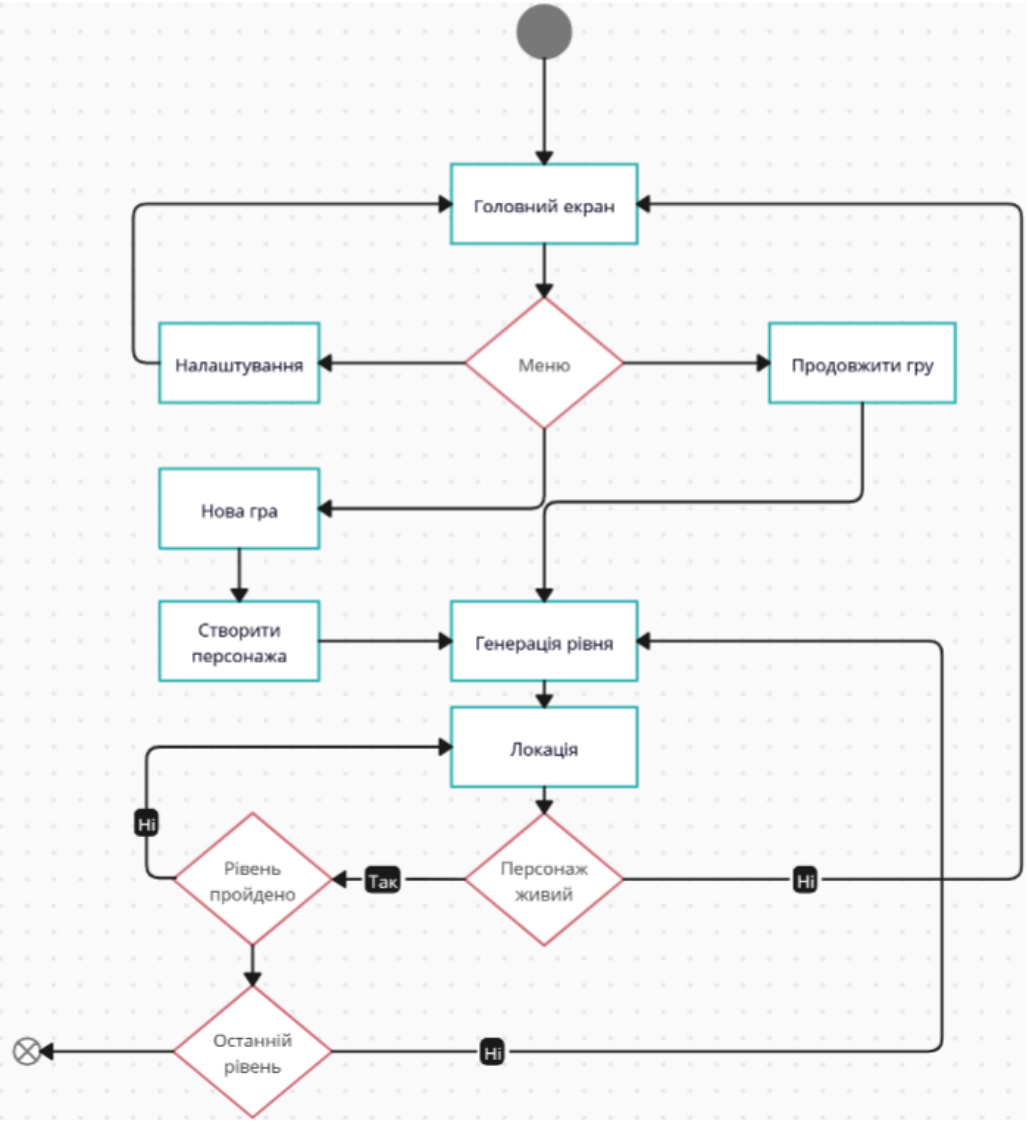

Рисунок 2.15 – Алгоритм проходження гри

2.3 Робочий проект

2.3.1 Вибір засобів розробки

В якості засобу розробки дипломного проекту було обрано ігровий рушій Unity (Рисунок 2.16.) та редактор коду Visual Studio (Рисунок 2.17.).

Unity [5] — багатоплатформовий інструмент для розроблення відеоігор і застосунків, і рушій, на якому вони працюють. Створені за допомогою Unity програми працюють на настільних комп'ютерних системах, мобільних пристроях та гральних консолях у дво- та тривимірній графіці, та на пристроях віртуальної чи доповненої реальності. Застосунки, створені за допомогою Unity, підтримують DirectX та OpenGL.

Microsoft Visual Studio [20] — серія продуктів фірми Майкрософт, які містять інтегроване середовище розробки програмного забезпечення та низку інших інструментальних засобів. Ці продукти дають змогу розробляти як консольні програми, так і програми з графічним інтерфейсом, включно з підтримкою технології Windows Forms, а також вебсайти, вебзастосунки, вебслужби як у рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight.

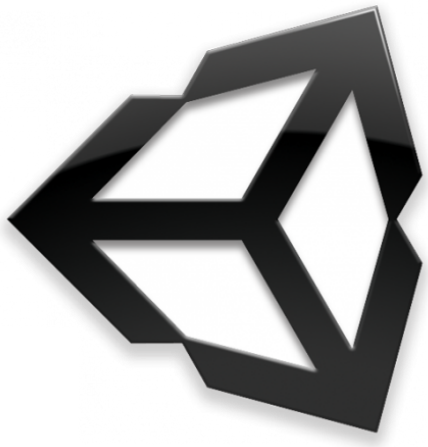

Рисунок 2.15 – Логотип Unity

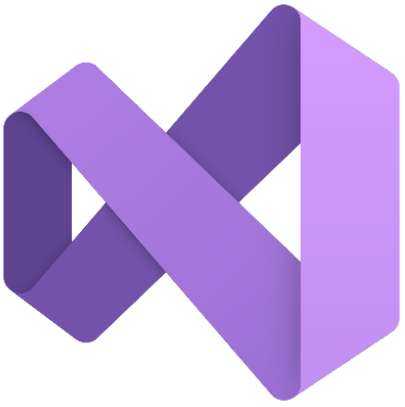

Рисунок 2.16 – Логотип Visual Studio

Unity: Ігровий рушій Unity є одним із найоптимальніших середовищ для розробки як двовимірних, так і тривимірних ігор. Його гнучкість та широкий

набір інструментів роблять його зручним вибором для створення проєктів різної складності.

Середовище розробки базується на мові програмування C#, яка забезпечує високу продуктивність і зручність роботи з об'єктно-орієнтованими структурами.. Для створення унікальних ігрових механік в Unity доступний великий набір вбудованих інструментів, зокрема система анімації, фізичний рушій, компоненти штучного інтелекту та розширені можливості для процедурної генерації контенту. Останній аспект є особливо важливим для ігор жанру rogue-lite, де рівні повинні змінюватися при кожному проходженні. Unity має зручний редактор, що включає засоби для тестування, профілювання та налагодження коду. Для прискорення розробки передбачені інструменти створення UI, редактор анімацій та візуальні скрипти, які дозволяють будувати логіку без необхідності написання великого обсягу коду. Таким чином, Unity є оптимальним вибором для розробки ігор жанру rogue-lite, оскільки поєднує широкий набір інструментів, високу продуктивність, зручність у використанні та широкий спектр можливостей.

Visual Studio: При виборі редактору коду, було обрано Visual Studio де є повна інтеграція з Unity через розширення Visual Studio Tools for Unity (VSTU). Плагін забезпечує автоматичне підключення до рушія, можливість налагодження коду в реальному часі та використання розширеного автодоповнення, що значно прискорює розробку.

Ще одним важливим фактором є розвинена система автодоповнення коду (IntelliSense). Visual Studio пропонує розширене розпізнавання структур коду, підказки щодо методів, класів і змінних, що допомагає уникати помилок і підвищує ефективність написання коду, також підтримує інструменти для профілювання, що дозволяють аналізувати продуктивність гри. Завдяки таким можливостям можна виявляти «вузькі місця» в коді, що можуть сповільнювати виконання скриптів, та оптимізувати їх для кращої продуктивності.

Розділ 3. РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМИ

Результатом розробки дипломного проєкту є гра для ПК платформи в стилі rogue-lite на двигуні Unity. Дана гра являє собою невеликий рогалик, де ваш персонаж май пройти підземелля, знайти вхід до фінального боса, та перемогти його виборовши своє право на свободу. Для досягнення даної цілі, було проаналізовано найуспішніші проєкти даного жанру, проведено велику роботу для розробки механік, спрайтів, текстур, анімацій та інших важливих аспектів гри.

3.1 Структура проєкту та сцен

В загальному в грі є 3 сцени, а саме:

- SampleScene – сцена головного меню;
- Game_Scene – сцена підземелля перед босом;
- BossScene – сцена підземелля з босом;

Нижче детально наведені структури кожної сцени:

1) SampleScene:

a) MainCamera;

b) Global Light 2D;

c) Canvas:

- MainMenu;
- SettingsMenu;

a) MenuManager;

b) MusicManger.

Примітка: *за всі елементи інтерфейсу та реалізацію їх функціоналу, також за спрайти, анімації та їх програвання відповідальний Голик Є.С., тому більш детально ознайомитись з елементами інтерфейсу користувача ви можете в 2 частині дипломної роботи.*

2) GameScene:

a) Grid:

- Tilemap;
- Collision.

b) MenuBarCanvas:

- MainMenuPanel

c) HpBarCanvas;

d) GameManager:

- ExitSpawn;
- DungeonGenerator;
- BoxGenerator;
- EntitySpawner;
- Player;
- MainCamera;

3) BossScene:

Ця сцена має аналогічну структуру до попередньої.

3.2 Впровадження механік гри

В результаті розробки було реалізовано:

Загальні механіки:

- Механіка генерації підземелля з детальними налаштуваннями:
  - Ширини та висоти;
  - Наповненість стінами;
  - Кількість ітерацій для згладжування стін;
  - Кількість кімнат;
  - Мінімальний та максимальний розмір кімнат;
  - Простір виділений для ширини стін та підлоги при генерації мапи;
- Механіка генерації НПС (неігрових персонажів) з детальними налаштуваннями спавну та кількості сутностей на локацію;
- Механіка генерації декораційних елементів з детальними налаштуваннями кількості та відстані спавну від гравця.
- Механіка для присвоєння та обробки ХП (від англ. HP – health point) гравця з методом TakeDamage для реагування на шкоду нанесену гравцю.
- Механіка для присвоєння та обробки ХП сутностей (від англ. Entity –

сутність) з методом TakeDamage для реагування на шкоду нанесену гравцю.

- Механіка слідування камери за гравцем;
- Механіка розбиття скринь з зіллями та їжею.

Механіки для сутностей:

Slime (слизень):

- Механіка патрулювання території обмеженої радіусом;
- Механіка отримання шкоди з методом Destroy та TakeDamage.

Goblin (гоблін):

- Механіка патрулювання території обмеженої радіусом;
- Механіка отримання шкоди з методом Destroy та TakeDamage;
- Механіка агресивної поведінки та атаки ближнього бою з методом TakeDamage, ChasePlayer та Attack.

Ghost (привид):

- Механіка патрулювання території обмеженої радіусом;
- Механіка отримання шкоди з методом Destroy та TakeDamage;
- Механіка агресивної поведінки та атаки дальнього бою з методом TakeDamage, ChasePlayer, Attack та Retreat.

Player (гравець):

- Механіка переміщення в просторі;
- Механіка телепортації гравця на певну відстань для ухилення від атак з методом Blink;
- Механіка отримання шкоди з методом TakeDamage та ReloadScene після смерті;
- Механіка відновлення ХП (здоров’я);
- Механіка атаки дального бою з методом PlayerShooting та SetCooldown;
- Скрипт для відтворення звуку ходіння тільки під час руху.

Boss (бос):

- Механіка переміщення в просторі;

- Механіка отримання шкоди та агресивної поведінки з методом Destroy, TakeDamage, CheckForEntities, CompleteLevel, isAggro та LoadNextScene;
- Механіку атак дального бою з різними шаблонами та методом SpawnBlade;

Механіки для префабів (від англ. Prefab – готовий шаблон) атак:

Player attack (атака гравця):

- Механіка для реєстрації влучань в сутності з методом OnTrigger;
- Механіка «життя» снаряду з методом onCollision для знищення проджектайлу (**проджектайл** (від англ. – projectile) - це об’єкт у грі, який рухається в просторі після "пострілу") після влучання.

Механіки атак для інших сутностей реалізовані схожим чином тому пропустимо їх.

3.3 Огляд механік гри

3.3.1 Механіка процедурної генерації рівнів

Генератор підземель використовує комбінацію алгоритмів для створення випадкових рівнів.

Основні фази генерації:

1) Ініціалізація та налаштування:
- Встановлюється розмір підземелля (dungeonWidth, dungenHeight);
- Налаштовуються параметри (fillPercent, smoothingiterations);
- Створюється двовимірний масив map, де 1 – стіна, 0 – підлога.

2) Генерація випадкової перечи з використанням RandomFillMap та SmoothMap:
- Спочатку заповнюється випадковим значенням згідно fillPercent;
- Потім застосовується кілька ітерацій згладжування (правило 4/5 сусідів).

3) Обробка карти (ProcessMap):
- Видаляються малі острови стін (менші за wallThresholdSize);
- Видаляються малі порожнечі (менші за roomThresholdSize).

4) Генерація кімнат (GenerateRooms):

- Створюється задана кількість прямокутних кімнат випадкового розміру;
- Кожна кімната записується у список survivingRooms.

5) З’єднання кімнат (GenerateClosestRooms):

- Знаходяться найближчі пари кімнат;
- Між ними створюються проходи (CreatePassage).

6) Візуалізація (RenderMap):

- Відображаються тайли підлоги і стін;
- Додається окремий шар для колізій.

7) Розміщення гравця (PlacePlayerInMainRoom):

- Гравець з’являється у центрі найближчої кімнати.

Ключові особливості:

1. Гібридний підхід: поєднує органічні печери (cellular automata) з прямокутними кімнатами.
2. Двошарова система: окремі Tilemap для підлоги, стін і колізій.
3. Оптимізація: використання CompositeCollider2D для ефективних колізій.
4. Гнучкість: багато параметрів для налаштування результату.

Результат:

Результатом роботи даної механіки є генерація підземелля, яке буде зображено на рисунку нижче.

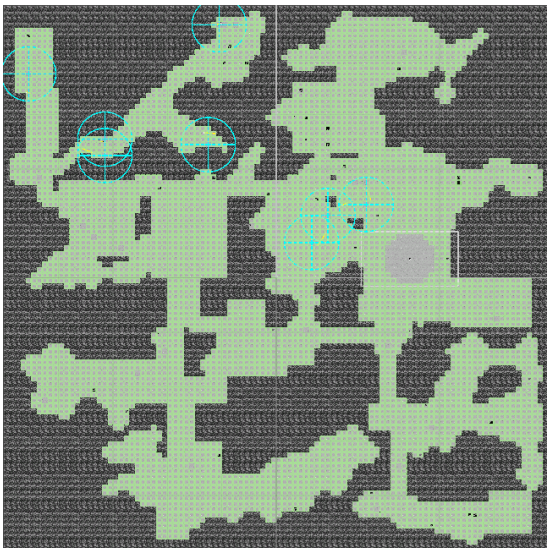

Рисунок 3.1 – Згенероване підземелля

3.3.2. Механіка розміщення сутностей

Ця механіка відповідає за генерацію ворогів у згенерованому DungeonGenerator підземеллі. Він використовує підлогу (floor tiles) як можливі точки появи, уникаючи зон близьких до гравця.

Основні функції:

1) Спавн ворогів з випадковим розподілом:

- Підтримує кілька типів ворогів (клас EnemyType);
- Враховує шанс появи, мінімальну та максимальну кількість, дистанцію від гравця.

2) Інтеграція з DungeonGenerator:

- Чекає подію OnDungeonGenerated перед початком спавну;
- Використовує FloorPosition для визначення точок появи.

3) Оптимізація та безпека:

- Ліміт спроб (maxSpawnAttemps) для уникнення навантаження та зависань;
- Автоматичне видалення метрвих ворогів методом Destroy.

4) Налаштування балансу:

- Інтенсивність (SpawnIntensity) для регулювання загальної кількості ворогів;

- maxTotalEnemies для обмеження спавну кількість ворогів.

Принцип роботи:

1) Підготовка точок спавну (PrepareSpawnPoints):

- Отримує всі клітинки підлоги з dungeonGeneraor.FloorPosition;

- Видаляє позиції ближче вказаних клітинок до гравця, щоб уникнути спавну перед гравцем.

2) Генерація ворогів (TrySpawnEnemy):

- Для кожного типу ворого обирається кількість між minCount та maxCount;

- Перевіряється шанс появи spawnChance;

- Створюється ворог у центрі клітинки GetCellCentreWorld.

3) Обмеження:

- Ліміт спроб для знаходження вільної точки для спавну;

- Ліміт кількості сутностей на мапі.

Результат:

Результатом роботи даної механіки є розміщення сутностей в згенерованому підземеллі. Нижче на рисунку зображено сутності, які розміщені на локації.

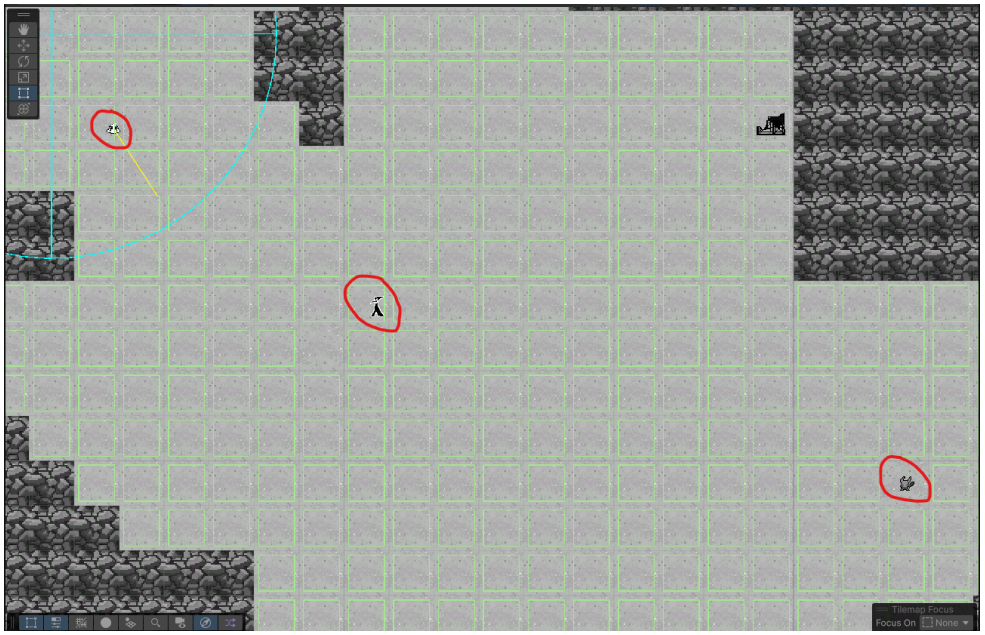

Рисунок 3.2 – Сутності розміщенні в підземеллі

3.3.3. Механіка розміщення декорацій

Цей скрипт створює випадкову кількість ящиків на підлозі згенерованого підземелля, уникаючи зон поблизу гравця.

Ключові Особливості

1) Гнучкі Налаштування:

- Декілька префабів ящиків (boxPrefabs) - можна додати різні варіанти (зброя, аптечки, скарби);
- Діапазон кількості (minBoxes, maxBoxes) - наприклад, від 10 до 30 ящиків на рівень;
- Шанс появи (boxSpawnChance) - регулює густину розміщення (наприклад, 70%);
- Захист від спавну біля гравця (minDistanceFromPlayer).

2) Інтеграція з DungeonGenerator:

- Автоматичний спавн після генерації рівня (через подію OnDungeonGenerated);
- Використовує список FloorPositions для точок появи.

3) Оптимізація:

- Перемішування точок (Shuffle) — запобігає скупченню ящиків в одній зоні;
- Контекстне меню (RespawnBoxes) — швидке тестування в редакторі Unity.

Принцип роботи:

1) Підготовка:

- Отримує всі клітинки підлоги з dungeonGenerator.FloorPositions;
- Перемішує їх у випадковому порядку.

2) Спавн ящиків:

- Для кожної клітинки перевіряє boxSpawnChance;
- Якщо точка далі minDistanceFromPlayer від гравця — створює випадковий ящик з boxPrefabs.

3) Ліміти:

- Процес зупиняється, коли досягнуто boxesToSpawn (випадкове число між minBoxes і maxBoxes).

Результат:

Результатом роботи даної механіки є розміщення префабів скринь в підземеллі. На рисунку 3.3 буде зображено розміщені скрині.

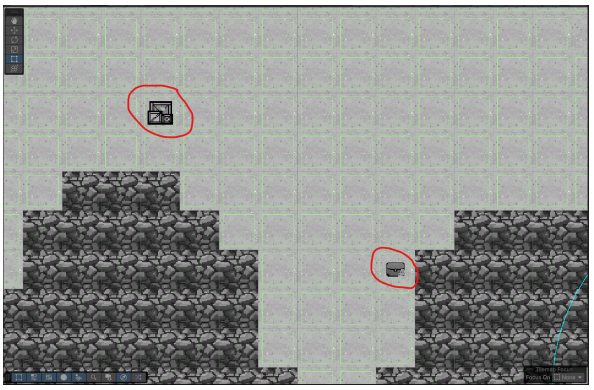

Рисунок 3.3 – Скрині в підземеллі

3.3.4. Базова механіка для бойової системи

1) Projectile.cs (Постріл/Снаряд):

Призначення: Літаючий об'єкт (куля), який зникає після зіткнення або через деякий час.

Ключові моменти:

- lifeTime: Автознищення через 2 секунди, якщо снаряд ні в кого не влучив.

Зіткнення з об'єктами:

- Реагує на об'єкти з тегами Box (ящики) та Entity (вороги).
- Після зіткнення зникає (Destroy).

2) AttackDamage.cs (Шкода від атаки):

Призначення: Нанесення шкода при зіткненні.

Ключові моменти:

- damage: Фіксований шкода.

Перевірка тега Entity: Уражає лише об'єкти з цим тегом.

Взаємодія з EnemyHealth: Викликає метод TakeDamage для зменшення HP.

В таблиці 1 наведено відмінності цих механік.

Таблиця 1 – Відмінності механік

Функція	Projectile	AttackDamage
Тип зіткнення	OnCollisionEnter2D	OnTriggerEnter2D
Об’єкти	Box, Entity	Entity
Логіка	Знищення	Шкода + знищення

3) EnemyHealth.cs (Здоров’я сутностей):

Основні функції:

Таблиця 2 – Основні функції EnemyHealth

Функціонал	Реалізація
Зберігання HP	Цілочисельне здоров'я (health = 100).
Отримання шкоди	Метод TakeDamage (int damage) зменшує HP і викликає подію OnDamageTaken.
Смерть	При health ≤ 0 викликається Die() із знищенням об'єкта (Destroy).
Події	Логи про отримання шкоди (Action<int>).
Безпека	Очищення подій при знищенні об'єкта (OnDestroy).

Принцип роботи:

1) Структура:

- Початкове здоров'я задається в інспекторі Unity;
- При отриманні шкоди (TakeDamage) відбувається зменшення здоров’я.

2) Подія OnDamageTaken:

- Дозволяє іншим скриптам реагувати на шкоду без прямих посилань.

3) Смерть:

- Виводить повідомлення в консоль;
- Знищує об’єкт.

Результат:

Результатом роботи даної механіки є реалізація бойової системи. Для перевірки працездатності скрипта було використано консоль. Для ознайомлення з логами консолі див. рис. 3.4.

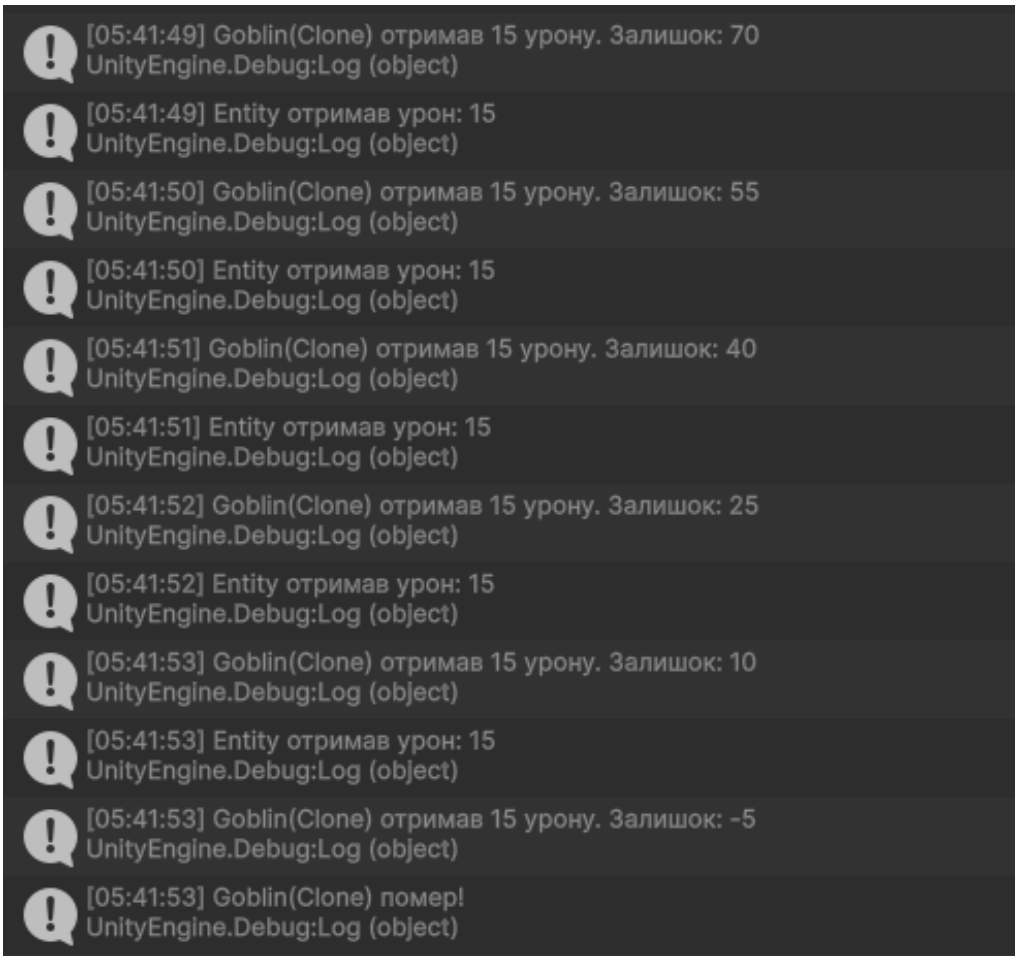

Рисунок 3.4 – Логи консолі для бойової системи

3.3.5. Механіка поведінки сутностей

Розглянемо механіку для 3 ключових агресивних сутностей.

1) Гоблін;
2) Привид;
3) Бос.

Та додатково для інтерактивних об’єктів.

3.3.5.1. Механіки поведінки гобліна

GoblinPatrol - Система патрулювання.

1) **Призначення**: Відповідає за базову поведінку ворога у спокійному стані.

2) **Ключові механіки**:

- Випадковий рух у межах заданого радіусу (patrolRadius);
- Використання корутин для плавного переміщення;

- Інтеграція з аніматором (перемикання між анімаціями ходьби/спокою);
- Перевірка перешкод за допомогою Raycast;
- Автоматичне припинення при активації агро-режиму.

3) **Оптимізаційні особливості**:

- Використання Rigidbody2D.MovePosition для плавного руху;
- Відслідковування початкової позиції (startPos) для обмеження зони патрулювання.

GoblinAggro - Система агресії.

1) **Призначення**: Відповідає за перехід у бойовий режим та переслідування гравця.

2) **Ключові механіки**:

- Двоетапна система виявлення (aggroRange > attackRange);
- Автоматичне переслідування гравця;
- Інтеграція з анімаціями;
- Метод IsAggro() для комунікації з іншими скриптами.

3) **Особливості**:

- Різні швидкості для патрулювання та переслідування;
- Чітка роздільна зона між станами "переслідування" і "атака".

GoblinAttack - Система ближньої атаки.

1) **Призначення**: Реалізує механіку атаки при зіткненні з гравцем.

2) **Ключові механіки**:

- Система кулдауну між атаками;
- Створення ефекту атаки (префаб);
- Автоматичне знищення ефекту атаки;
- Перевірка шару гравця для точної ідентифікації.

3) **Особливості реалізації**:

- Використання корутин для таймінгу атак;
- Проста, але ефективна система ближнього бою;
- Можливість легкого налаштування через інспектор.

GoblinAttackProjectile - Система снарядів.

1) **Призначення**: Відповідає за атаки гобліна.

2) **Ключові механіки**:

- Нанесення шкоди при контакті;
- Відтворення спец. ефекту попадання;
- Налаштовуваний час життя снаряда;
- Візуальна віддача при попаданні.

3) **Особливості**:

- Динамічне створення ефектів попадання;
- Відключення коллайдера після попадання;
- Гнучкі налаштування через інспектор.

Взаємодія між скриптами:

1) **GoblinAggro** виступає як "мозок", керуючи станами:

- Активація чи деактивація патрулювання;
- Перемикання між переслідуванням і атакою.

2) **GoblinPatrol** автоматично припиняється при активному агресивному стані.

3) **GoblinAttack** і **GoblinAttackProjectile**:

- Можуть використовуватись одночасно для різних типів атак;
- Інтегруються через загальну систему шкоди (PlayerHealth).

Результат:

Нижче на рисунку ви може побачити візуалізований результат взаємодії цих механік.

Рисунок 3.5 – Візауалізація механік Гобліна

3.3.5.2. Механіка поведінки привида.

1) **GhostAggro - Контролер агресії.**

Ядро поведінки, що керує трьома станами:

- Спокій (isAggro = false) – бездіяльність;
- Переслідування;
- Відступ (рух від гравця з retreatSpeed при наближенні).

Ключові особливості:

- Дві зони виявлення (aggroRange > safeDistance);
- Інтеграція з аніматором через UpdateAnimation();
- Метод IsAggro() для комунікації з іншими скриптами.

Оптимізація:

- Використання Rigidbody2D.linearVelocity замість Transform.position для плавного руху з фізикою.

2) **GhostPatrol - Система патрулювання.**

Реалізує поведінку у спокійному стані:

- Випадковий рух у межах patrolRadius;
- Очікування між точками (waitTime);
- Перевірка перешкод (obstacleLayers).

Специфіка реалізації:

- Використання корутин для послідовних дій;
- Автоматична пауза при активному агресивному стані (IsAggro());
- Синхронізація анімацій з рухом.

3) **GhostAttack - Система атаки**

Відповідає за дальні атаки:

- Затримка між атаками (attackCooldown);
- Створення снарядів (projectilePrefab);
- Фізичний рух снарядів (projectileSpeed).

Технічні деталі:

- Вимкнена гравітація для снарядів;
- Перевірка дистанції (attackRange < aggroRange).

4) **GhostAttackProjectile - Снаряди**

Обробка снарядів атаки:

- Нанесення шкоди (damage);
- Ефект попадання (hitSprite);
- Автознищення через lifetime.

5) **Взаємодія між скриптами.**

1. **GhostAggro** виступає як "мозок":

- Вмикає чи вимикає патрулювання;
- Керує переходами між станами.

2. **GhostPatrol**:

- Призупиняється при IsAggro() = true;
- Відновлюється автоматично.

3. **GhostAttack**:

- Активується тільки в агресивному режимі;
- Використовує дані про позицію гравця.

4. **GhostAttackProjectile**:

- Незалежний об'єкт після створення;
- Повідомляє PlayerHealth про шкоду.

Результат:

Нижче на рисунку ви може побачити візуалізований результат взаємодії цих механік.

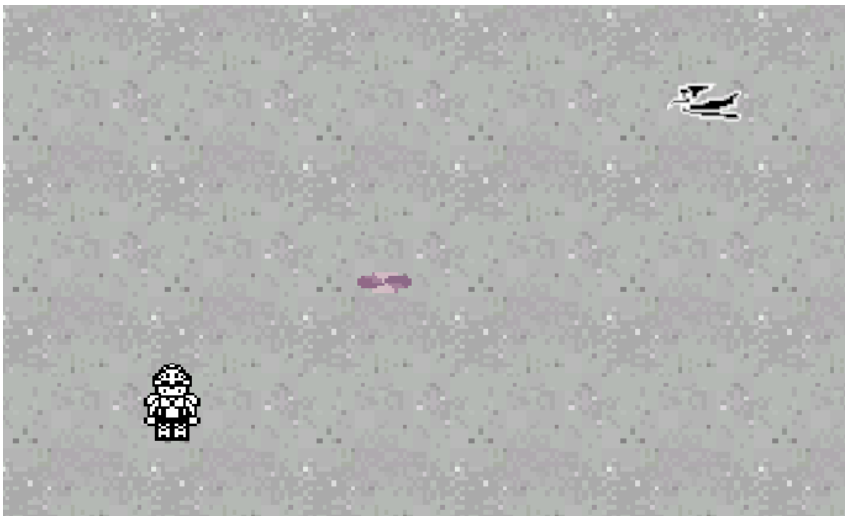

Рисунок 3.6 - Візауалізація механік Привида

3.3.5.3. Механіка поведінки боса.

1) **BossAggro - Контролер агресії.**

Ядро AI боса, що відповідає за:

- Переслідування гравця на фіксованій дистанції (stopDistance);
- Активацію агро-режиму через подію атаки гравця;
- Синхронізацію анімацій з рухом.

Ключові особливості:

- Гнучкий контроль швидкості;
- Чітка дистанція зупинки;
- Автоматичне вимкнення руху при досягненні цілі;

2) **BossBladeAttack - Система спеціальних атак.**

Механіка атаки з трьома типами патернів:

1. Спіральний (SpiralPattern) - послідовний викид префабів;
2. Круговий (CircularPattern) - одночасний викид по колу;
3. Хрестоподібний (CrossPattern) - 4 префаба атаки під прямим кутом.

Особливості реалізації:

- **Випадковий вибір патерну** (1-3)
- **Налаштовувані параметри**:

- bladeSpeed - швидкість польоту;
- bladesPerAttack - кількість префабів атаки;
- attackCooldown - затримка між атаками.

3) **BossAttackProjectile - Снаряди для атак.**

Технічні деталі:

- Оптимізоване знищення;
- Гнучкі налаштування через інспектор Unity;

Результат:

Нижче на рисунку ви може побачити візуалізований результат взаємодії цих механік.

3.3.5.4. Механіка предметів та інтерактивних об’єктів.

1) Box.cs – Випадкове випадіння предмету з ящиків.

Призначення: створює випадковий предмет при знищенні ящика гравцем.

Ключові особливості:

- 50% шанс випадання їжі (meatPrefab) або зілля (potionPrefab);
- Автоматичне знищення ящика після відкриття;

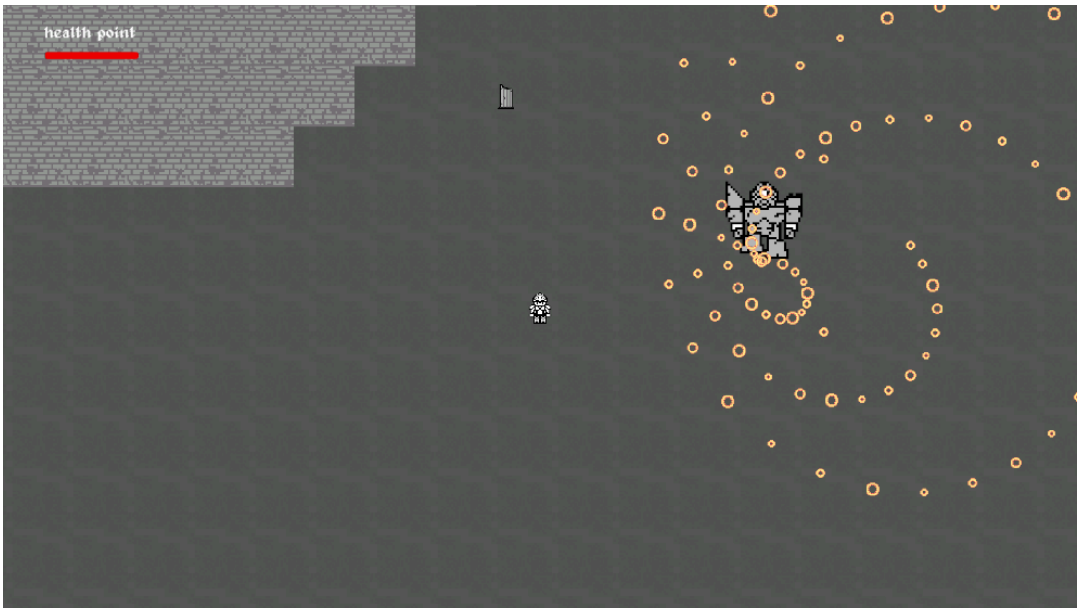

Рисунок 3.7 – Візуалізація механік Боса

2) FoodHeal.cs - Відновлення здоров'я.

Призначення: відновлює HP гравця при підборі їжі.

Ключові особливості:

- Перевірка повного здоров'я (!IsFullHealth());
- Конфігурований обсяг лікування (healAmount);
- Запобігання надмірному лікування (якщо здоров’я вже максимум).

3) SpeedPotion.cs - Тимчасовий буст швидкості.

Призначення: збільшує швидкість руху гравця.

Ключові особливості:

- Пряме збільшення швидкості через PlayerMovement;
- Можна налаштувати силу ефекту;
- Знищення після використання.

Нижче на рисунку 3.8 буде наведено діаграму, яка відображає принцип роботи механіки предметів та інтерактивних об’єктів.

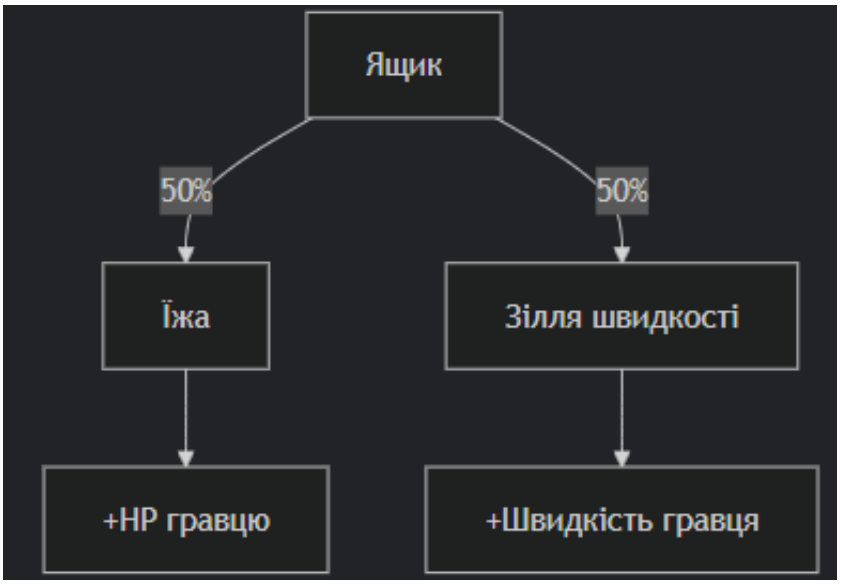

Рисунок 3.8 – Принцип роботи механіки

Також на рисунку 3.9 зображено візуалізований принцип роботи цієї механіки.

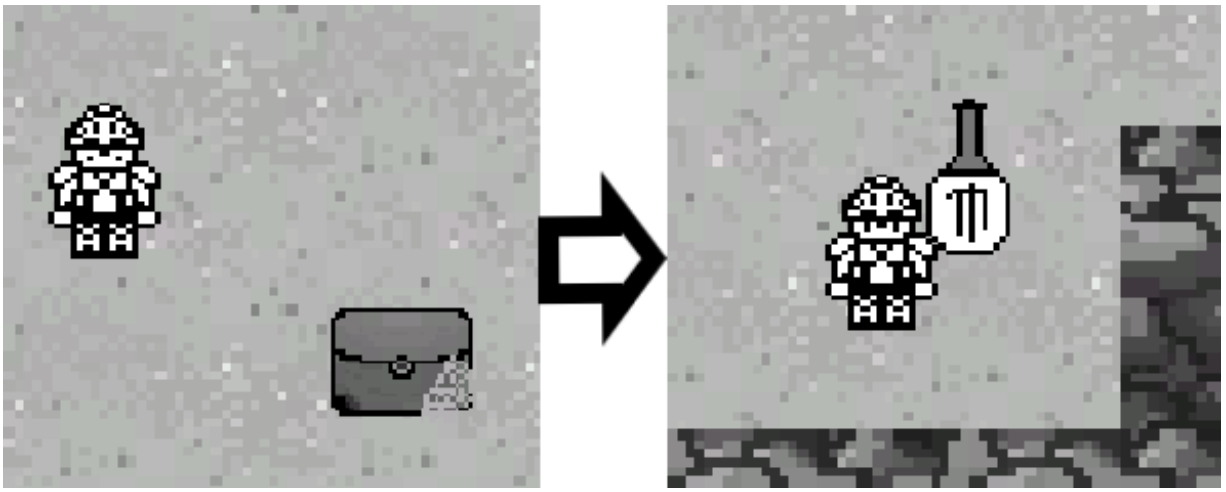

Рисунок 3.9 – Візуалізована механіка предметів

Примітка: для коректної роботи всіх механік потрібно правильно встановити компоненти в інспекторі, а саме Rigibody2D та BoxCollider для колізій та реєстрації влучань.

Розділ 4. ВИМОГИ, ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ СИСТЕМИ

4.1 Вимоги до гри та основні підходи до проєктування.

Розробка Rogue-like гри має ґрунтуватися на основних принципах проєктування, що забезпечують глибокий геймплей, реіграбельність, технічну стабільність та атмосферу. Ігровий процес має бути інтуїтивно зрозумілим, дозволяючи гравцям швидко освоїти основні механіки.

Ключові аспекти проектування включають:

1) **Процедурну генерацію контенту** як основу реіграбельності. Алгоритми створення рівнів мають забезпечувати унікальність кожного проходження.

2) **Глибоку бойову систему** з елементами стратегії. Гра має пропонувати різні стилі бою (ближній, дальній) з чіткими механіками.

3) **Атмосферний звуковий супровід** та візуальний стиль.

4.2. Основні характеристики, що враховуються при розробці гри.

1) **Геймплейні характеристики:**

- *Глибина механік* - система повинна поєднувати простоту освоєння з можливістю майстерного використання;
- *Реіграбельність* - забезпечується через процедурну генерацію, випадкові івенти та множинні стратегії проходження;
- *Баланс складності* - поступове зростання викликів відповідно до прогресу гравця.

2) **Технічні характеристики:**

- *Продуктивність* - стабільний FPS на цільових платформах (60+ для ПК, 30+ для мобільних);
- *Оптимізація* - ефективне використання ресурсів;
- *Стабільність* - мінімізація багів та інших технічних проблем.

4.2. Опис проєктованого продукту

Проєктований продукт: Комп’ютерна гра.

Назва продукту: The Game або HRA.

Опис продукту: HRA – це гра для ПК платформи в жанрі *rogue-lite* з елементами *RPG*. Гра стилізована під ретро-ігри часів третього покоління домашніх консолей [21]. Кольорова гама була обрана монохромна, в сірих кольорах. Музичний супровід також розроблений і стилізований.

Історія гри: «Події відбуваються в вигаданому фентезійному світі. Королівство жило мирно, поки не з’явився таємничий острів і не окутав землі туманом. Мирні жителі почали помирати від постійних набігів монстрів. Король посилає експедицію, та вона потрапляє у шторм і потерпає невдачі. Гравець бере на себе роль вцілілого солдату, якому треба розгадати загадку таємничого острову і врятувати королівство.»

Основні функції та можливості гри:

- Процедурна генерація рівнів

Однією з ключових особливостей проєкту є процедурна генерація рівнів. Під час кожного проходження гра створює унікальні підземелля з випадковими кімнатами, ворогами, що значно збільшує реіграбельність та робить кожну спробу відмінною від попередньої.

- Бойова система

Бойова система гри передбачає динамічні сутички з використанням зброї дальнього бою. Гравець може атакувати ворогів у напрямку курсора мишею, що дозволяє точно контролювати удари. Крім стандартних атак, персонаж володіє унікальними вміннями, наприклад, блискавичним ривком (блінком). Боси, які трапляються на певних етапах, мають складні патерни атак, що вимагають тактичного підходу та швидкої реакції.

- Середовище гри

Середовище гри побудоване таким чином, щоб стимулювати дослідження. Під час проходження гравець стикається з інтерактивними об’єктами. Взаємодія з такими елементами робить процес проходження більш захоплюючим. У кожній сесії гравець може знаходити нові шляхи.

- Звукове оформлення

Звукове оформлення також грає важливу роль у створенні атмосфери. Атмосферний саундтрек підсилює напруженість.

- Технічна реалізація

Технічна реалізація проєкту базується на ігровому рушії Unity, що забезпечує високу продуктивність та гнучкість у налаштуванні. Оптимізація графіки та фізики дозволяє досягти стабільної частоти кадрів.

4.3. План виробництва додатку

Етап 1: Аналіз вимог та проектування архітектури гри

Початковий етап передбачає проведення детального аналізу концепції гри та визначення її основних механік. На цьому етапі формуються такі основні документи:

- **Концептуальний документ гри** – містить загальний опис, жанр, основні елементи геймплею та візуальний стиль.
- **Технічна документація** – описує архітектуру гри, модульну структуру, вибір технологій (ігровий рушій Unity), формати збереження даних та методи оптимізації.
- **Документ з дизайну гри** – містить деталізований опис геймплею, сюжет, ігрові механіки, інтерфейс користувача, систему бойової взаємодії та генерацію рівнів.
- **Архітектура проєкту** – передбачає розробку компонентів гри: генерація рівнів, бойова система, анімації, AI ворогів, система інвентарю та розвиток персонажа.
- **План управління ризиками** – передбачає виявлення можливих технічних труднощів та розробку плану з їхнього подолання.

Етап 2: Розробка основних механік

На даному етапі реалізуються базові механіки гри, а саме:

- Рух та управління персонажем;
- Бойова система;

- Процедурна генерація рівнів;
- Система ворогів;

Етап 3: Розробка контенту та розширення функціоналу

Після тестування базового прототипу додаються нові елементи та покращується існуючий функціонал:

- Розробка додаткових механік – включає нові види ворогів;
- Покращення бойової системи – додавання механік для ухилення від атак ворогів;
- Механіка смерті та відродження – реалізація відродження після смерті;
- Звукове оформлення – додавання звуків атак, фонового саундтреку;
- Покращення анімацій – додаткові ефекти при атаці, смерті ворогів та використанні заклинань.
- Візуальна оптимізація – створення стилізованих текстур для рівнів, ворогів та предметів.

Етап 4: Тестування та налагодження гри

- Функціональне тестування – перевірка коректності основних механік (рух, атака, генерація).
- Стрес-тестування – перевірка на продуктивність при великій кількості ворогів або предметів.
- Тестування бойової системи – балансування урону, складності ворогів та виживання гравця.
- Альфа-тестування – залучення тестерів для виявлення недоліків та пропозицій щодо поліпшення.
- Виправлення багів – на основі фідбеку усуваються помилки та покращується стабільність гри.

Етап 5: Запуск бета-версії та збирання відгуків користувачів

Після успішного тестування проєкт збирається в готовий додаток гри для тестування обмеженим колом гравців для збору відгуків та подальшого виправлення помилок.

Етап 6: Випуск гри з бета-версії

Після успішного завершення всіх попередніх етапів і враховуючи отриманий відгук від гравців, випускається стабільна версія гри.

ВИСНОВКИ

У межах кваліфікаційної роботи було реалізовано розробку комп’ютерної гри в жанрі Rogue-like із використанням ігрового рушія Unity, зосереджуючись на аспектах геймдизайну, аудіовізуального оформлення та інтерфейсної частини. На основі проведеного жанрового аналізу було виділено ключові елементи, притаманні іграм цього типу, зокрема процедурну генерацію, високу складність, нелінійність проходження, перманентну смерть персонажа тощо.

Unity було обрано як ефективне середовище для реалізації зазначених механік завдяки широким можливостям з процедурної генерації рівнів, побудови інтерфейсу, інтеграції графічних та звукових елементів. Створено концепцію гри та прототип, що включає базові механіки: переміщення, бої з противниками, генерацію рівнів та інвентаризацію предметів.

Здійснено візуальне та звукове оформлення гри відповідно до обраної стилістики. Інтерфейс користувача реалізовано з урахуванням принципів зручності, інтуїтивної навігації та доступності.

Проведене тестування підтвердило працездатність прототипу, відповідність реалізованих механік жанровим вимогам, а також виявило напрями для подальшого вдосконалення, зокрема в частині балансу складності, розширення сценаріїв генерації та оптимізації продуктивності.

Отримані результати свідчать про практичну доцільність використання ігрового рушія Unity для створення ігор у жанрі Rogue-like і демонструють здатність автора до самостійного виконання повного циклу розробки від ідеї до прототипу.

Під час роботи над дипломним проектом були пройдені такі етапи:

− **Детальне дослідження предметної області, аналіз готових рішень та постановка задачі**

Було проведено аналіз ігрового ринку, зокрема жанру Rogue-like, щоб визначити потреби гравців та виявити актуальні тенденції. Розглянуто існуючі рішення, проаналізовано їхні сильні та слабкі сторони. На основі цього

сформульовано чітке завдання: розробити інноваційну гру жанру Rogue-like на основі ігрового двигуна Unity.

− **Проєктування**

Проєктування гри включало ескізний, технічний та робочий етапи. Для проєктування була обрана об’єктно-орієнтована методологія, що дозволила створити модульну та масштабовану архітектуру. На етапі ескізного проєктування були розроблені прототипи основних ігрових механік та дизайну користувацького інтерфейсу.

− **Розробка та тестування функціональних модулів гри**

Розробка гри здійснювалася на основі ігрового рушія Unity з використанням C#. Реалізовано ключові елементи ігрового процесу: управління персонажем, бойова система з використанням дистанційних атак, динамічне генерування рівнів та розміщення ворогів. Створено адаптивний інтерфейс, що дозволяє зручно відображати показники здоров’я та інші параметри.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жанр rogue-like: https://en.wikipedia.org/wiki/Roguelike
2. Механіки гри: https://en.wikipedia.org/wiki/Game_mechanics
3. Оптимізація гри: https://codefinity.com/blog/Optimization-Techniques-in-Game-Development
4. Реіграбельність гри: https://ksw.net.ua/shho-take-reigrabelnist-i-chomu-vona-vazhlyva/
5. Ігровий рушій Unity: https://en.wikipedia.org/wiki/Unity_(game_engine)
6. Документація Unity: https://docs.unity.com/
7. Nystrom, R. (2014). Game Programming Patterns. Genever Benning. – 345 с.
8. Unger, R. (2013). The User Experience Team of One: A Research and Design Survival Guide. Rosenfeld Media. – 288 с.
9. Kyburz, P. (2017). Procedural Content Generation for Unity Game Development. Packt Publishing. – 254 с.
10. Short, T., & Adams, T. (2017). Procedural Generation in Game Design. CRC Press. – 312 с.
11. Salen, K., & Zimmerman, E. (2004). Rules of Play: Game Design Fundamentals. MIT Press. – 688 с.
12. Gibson Bond, J. (2022). Introduction to Game Design, Prototyping, and Development (3rd ed.). Addison-Wesley. – 928 с.
13. Schell, J. (2019). The Art of Game Design: A Book of Lenses (3rd ed.). CRC Press. – 600 с.
14. Eberly, D. (2006). *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics* (2nd ed.). Morgan Kaufmann. – 1040 с.
15. Ferrone, H. (2023). Learning C# by Developing Games with Unity (7th ed.). Packt Publishing. – 450 с.
16. Стаття про гру DeadCells: https://en.wikipedia.org/wiki/Dead_Cells
17. Стаття про гру Darkest Dungeon: https://en.wikipedia.org/wiki/Darkest_Dungeon

18. Стаття про гру The Binding of Isaac:
https://en.wikipedia.org/wiki/The_Binding_of_Isaac
19. Стаття про гру Hades: https://en.wikipedia.org/wiki/Hades_(video_game)
20. Програмне забезпечення: https://en.wikipedia.org/wiki/Visual_Studio
21. Покоління консолей:
https://en.wikipedia.org/wiki/Third_generation_of_video_game_consoles

ДОДАТОК А. Код розробки

Лістинг А.1 – скрипт «DungeonGenerator»

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

public class DungeonGenerator : MonoBehaviour
{
    [Header("Tile Settings")]
    [SerializeField] private TileBase wallTile;
    [SerializeField] private TileBase sandTile;

    [Header("Collision Settings")]
    [SerializeField] private Tilemap collisionTilemap;
    [SerializeField] private TileBase collisionTile;

    [Header("Dungeon Settings")]
    [SerializeField] private int dungeonWidth = 100;
    [SerializeField] private int dungeonHeight = 100;
    [SerializeField][Range(0, 100)] private int fillPercent = 45;
    [SerializeField] private int smoothingIterations = 5;
    [SerializeField] private int wallThresholdSize = 50;
    [SerializeField] private int roomThresholdSize = 50;
    [SerializeField] private int passageWidth = 2;
    [SerializeField] private int roomCount = 5;
    [SerializeField] private int roomMinSize = 5;
    [SerializeField] private int roomMaxSize = 15;
```

```
[Header("References")]
[SerializeField] private Tilemap groundTilemap;
[SerializeField] private Tilemap wallsTilemap;
[SerializeField] private Transform player;

private int[,] map;
private List<Room> survivingRooms = new List<Room>();
private System.Random random;
private List<Vector3Int> floorPositions = new List<Vector3Int>();

public event System.Action OnDungeonGenerationComplete;
public List<Vector3Int> FloorPositions => floorPositions;
public Tilemap GroundTilemap => groundTilemap;
public Transform Player => player;
public event Action OnDungeonGenerated;

void Start()
{
    random = new System.Random(Time.time.GetHashCode());
    GenerateDungeon();
}

void GenerateDungeon()
{
    map = new int[dungeonWidth, dungeonHeight];

    RandomFillMap();

    for (int i = 0; i < smoothingIterations; i++)
```

```
    {
        SmoothMap();
    }

    ProcessMap();

    GenerateRooms();

    ConnectClosestRooms();

    RenderMap();

    PlacePlayerInMainRoom();
    OnDungeonGenerated?.Invoke();
    Debug.Log($"Generated dungeon with {floorPositions.Count} floor tiles");
}

void RandomFillMap()
{
    for (int x = 0; x < dungeonWidth; x++)
    {
        for (int y = 0; y < dungeonHeight; y++)
        {
            if (x == 0 || x == dungeonWidth - 1 || y == 0 || y == dungeonHeight - 1)
            {
                map[x, y] = 1;
            }
            else
            {
```

```
                map[x, y] = (random.Next(0, 100) < fillPercent) ? 1 : 0;
            }
        }
    }
}

void SmoothMap()
{
    for (int x = 0; x < dungeonWidth; x++)
    {
        for (int y = 0; y < dungeonHeight; y++)
        {
            int neighbourWallTiles = GetSurroundingWallCount(x, y);

            if (neighbourWallTiles > 4)
                map[x, y] = 1;
            else if (neighbourWallTiles < 4)
                map[x, y] = 0;
        }
    }
}

void ProcessMap()
{
    List<List<Coord>> wallRegions = GetRegions(1);
    foreach (List<Coord> wallRegion in wallRegions)
    {
        if (wallRegion.Count < wallThresholdSize)
        {
```

```
            foreach (Coord tile in wallRegion)
            {
                map[tile.tileX, tile.tileY] = 0;
            }
        }
    }

    List<List<Coord>> roomRegions = GetRegions(0);
    foreach (List<Coord> roomRegion in roomRegions)
    {
        if (roomRegion.Count < roomThresholdSize)
        {
            foreach (Coord tile in roomRegion)
            {
                map[tile.tileX, tile.tileY] = 1;
            }
        }
    }
}

void GenerateRooms()
{
    survivingRooms.Clear();

    for (int i = 0; i < roomCount; i++)
    {
        int roomWidth = random.Next(roomMinSize, roomMaxSize);
        int roomHeight = random.Next(roomMinSize, roomMaxSize);
```

```
            int roomX = random.Next(0, dungeonWidth - roomWidth - 1);
            int roomY = random.Next(0, dungeonHeight - roomHeight - 1);

            for (int x = roomX; x < roomX + roomWidth; x++)
            {
                for (int y = roomY; y < roomY + roomHeight; y++)
                {
                    if (x > 0 && x < dungeonWidth - 1 && y > 0 && y < dungeonHeight - 1)
                    {
                        map[x, y] = 0;
                    }
                }
            }

                survivingRooms.Add(new Room(new Coord(roomX, roomY), roomWidth,
roomHeight, map));
        }
    }

    void ConnectClosestRooms()
    {
        for (int i = 0; i < survivingRooms.Count - 1; i++)
        {
            float shortestDistance = float.MaxValue;
            Room closestRoomA = null;
            Room closestRoomB = null;

            for (int j = i + 1; j < survivingRooms.Count; j++)
            {
```

```
            float distance = Vector2.Distance(
                                                new   Vector2(survivingRooms[i].center.tileX,
survivingRooms[i].center.tileY),
                                                new   Vector2(survivingRooms[j].center.tileX,
survivingRooms[j].center.tileY));

            if (distance < shortestDistance)
            {
                shortestDistance = distance;
                closestRoomA = survivingRooms[i];
                closestRoomB = survivingRooms[j];
            }
        }

        if (closestRoomA != null && closestRoomB != null)
        {
            CreatePassage(closestRoomA, closestRoomB);
        }
    }
}

void CreatePassage(Room roomA, Room roomB)
{
    Coord start = roomA.center;
    Coord end = roomB.center;

    if (random.Next(0, 2) == 0)
    {
        CreateHorizontalPassage(start.tileX, end.tileX, start.tileY);
```

```
            CreateVerticalPassage(start.tileY, end.tileY, end.tileX);
        }
        else
        {
            CreateVerticalPassage(start.tileY, end.tileY, start.tileX);
            CreateHorizontalPassage(start.tileX, end.tileX, end.tileY);
        }
    }

    void CreateHorizontalPassage(int x1, int x2, int y)
    {
        for (int x = Mathf.Min(x1, x2); x <= Mathf.Max(x1, x2); x++)
        {
            for (int w = -passageWidth / 2; w <= passageWidth / 2; w++)
            {
                        if (x >= 0 && x < dungeonWidth && y + w >= 0 && y + w <
dungeonHeight)
                {
                    map[x, y + w] = 0;
                }
            }
        }
    }

    void CreateVerticalPassage(int y1, int y2, int x)
    {
        for (int y = Mathf.Min(y1, y2); y <= Mathf.Max(y1, y2); y++)
        {
            for (int w = -passageWidth / 2; w <= passageWidth / 2; w++)
```

```
        {
                if (x + w >= 0 && x + w < dungeonWidth && y >= 0 && y < dungeonHeight)
            {
                map[x + w, y] = 0;
            }
        }
    }
}

void RenderMap()
{
    groundTilemap.ClearAllTiles();
    wallsTilemap.ClearAllTiles();
    if (collisionTilemap != null) collisionTilemap.ClearAllTiles();

    floorPositions.Clear();

    for (int x = 0; x < dungeonWidth; x++)
    {
        for (int y = 0; y < dungeonHeight; y++)
        {
                Vector3Int tilePosition = new Vector3Int(x - dungeonWidth / 2, y - dungeonHeight / 2, 0);

            if (map[x, y] == 1)
            {
                wallsTilemap.SetTile(tilePosition, wallTile);
```

```
            if (collisionTilemap != null && collisionTile != null)
            {
                collisionTilemap.SetTile(tilePosition, collisionTile);
            }
        }
        else
        {
            groundTilemap.SetTile(tilePosition, sandTile);
            floorPositions.Add(tilePosition);
        }
    }
}

if (collisionTilemap != null)
{
                                        TilemapCollider2D collider =
collisionTilemap.GetComponent<TilemapCollider2D>();
                                    if (collider == null) collider =
collisionTilemap.gameObject.AddComponent<TilemapCollider2D>();

                                        CompositeCollider2D composite =
collisionTilemap.GetComponent<CompositeCollider2D>();
                                    if (composite == null) composite =
collisionTilemap.gameObject.AddComponent<CompositeCollider2D>();

    Rigidbody2D rb = collisionTilemap.GetComponent<Rigidbody2D>();
                                        if (rb == null) rb =
collisionTilemap.gameObject.AddComponent<Rigidbody2D>();
    rb.bodyType = RigidbodyType2D.Static;
```

```
    }
}

void PlacePlayerInMainRoom()
{
    if (player != null && survivingRooms.Count > 0)
    {
        Room largestRoom = survivingRooms[0];
        foreach (Room room in survivingRooms)
        {
            if (room.roomSize > largestRoom.roomSize)
            {
                largestRoom = room;
            }
        }

        Vector3Int tilePosition = new Vector3Int(
            largestRoom.center.tileX - dungeonWidth / 2,
            largestRoom.center.tileY - dungeonHeight / 2, 0);
        Vector3 worldPosition = groundTilemap.GetCellCenterWorld(tilePosition);

        player.position = worldPosition;
    }
}

int GetSurroundingWallCount(int gridX, int gridY)
{
    int wallCount = 0;
    for (int neighbourX = gridX - 1; neighbourX <= gridX + 1; neighbourX++)
```

```
        {
            for (int neighbourY = gridY - 1; neighbourY <= gridY + 1; neighbourY++)
            {
                if (neighbourX >= 0 && neighbourX < dungeonWidth && neighbourY >= 0
&& neighbourY < dungeonHeight)
                {
                    if (neighbourX != gridX || neighbourY != gridY)
                    {
                        wallCount += map[neighbourX, neighbourY];
                    }
                }
                else
                {
                    wallCount++;
                }
            }
        }
        return wallCount;
    }

    List<List<Coord>> GetRegions(int tileType)
    {
        List<List<Coord>> regions = new List<List<Coord>>();
        int[,] mapFlags = new int[dungeonWidth, dungeonHeight];

        for (int x = 0; x < dungeonWidth; x++)
        {
            for (int y = 0; y < dungeonHeight; y++)
            {
```

```
            if (mapFlags[x, y] == 0 && map[x, y] == tileType)
            {
                List<Coord> newRegion = GetRegionTiles(x, y);
                regions.Add(newRegion);

                foreach (Coord tile in newRegion)
                {
                    mapFlags[tile.tileX, tile.tileY] = 1;
                }
            }
        }
    }

    return regions;
}

List<Coord> GetRegionTiles(int startX, int startY)
{
    List<Coord> tiles = new List<Coord>();
    int[,] mapFlags = new int[dungeonWidth, dungeonHeight];
    int tileType = map[startX, startY];

    Queue<Coord> queue = new Queue<Coord>();
    queue.Enqueue(new Coord(startX, startY));
    mapFlags[startX, startY] = 1;

    while (queue.Count > 0)
    {
        Coord tile = queue.Dequeue();
```

```
            tiles.Add(tile);

            for (int x = tile.tileX - 1; x <= tile.tileX + 1; x++)
            {
                for (int y = tile.tileY - 1; y <= tile.tileY + 1; y++)
                {
                    if (x >= 0 && x < dungeonWidth && y >= 0 && y < dungeonHeight &&
(y == tile.tileY || x == tile.tileX))
                    {
                        if (mapFlags[x, y] == 0 && map[x, y] == tileType)
                        {
                            mapFlags[x, y] = 1;
                            queue.Enqueue(new Coord(x, y));
                        }
                    }
                }
            }
        }

        return tiles;
    }

    struct Coord
    {
        public int tileX;
        public int tileY;

        public Coord(int x, int y)
        {
```

```
            tileX = x;
            tileY = y;
        }
    }

    class Room
    {
        public List<Coord> edgeTiles;
        public List<Coord> tiles;
        public Coord center;
        public int roomSize;

        public Room(Coord start, int width, int height, int[,] map)
        {
            tiles = new List<Coord>();
            edgeTiles = new List<Coord>();

            for (int x = start.tileX; x < start.tileX + width; x++)
            {
                for (int y = start.tileY; y < start.tileY + height; y++)
                {
                                if (x >= 0 && x < map.GetLength(0) && y >= 0 && y <
map.GetLength(1))
                    {
                        tiles.Add(new Coord(x, y));

                        if (x == start.tileX || x == start.tileX + width - 1 ||
                            y == start.tileY || y == start.tileY + height - 1)
                        {
```

```
                    edgeTiles.Add(new Coord(x, y));
                }
            }
        }
    }

    roomSize = tiles.Count;
    center = new Coord(start.tileX + width / 2, start.tileY + height / 2);
  }
 }
}
```

Лістинг А.2 – скрипт «BoxGenerator»

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

public class BoxSpawner : MonoBehaviour
{
    [Header("Box Settings")]
    [SerializeField] private GameObject[] boxPrefabs;
    [SerializeField] private int minBoxes = 10;
    [SerializeField] private int maxBoxes = 30;
    [SerializeField][Range(0f, 1f)] private float boxSpawnChance = 0.7f;
    [SerializeField] private float minDistanceFromPlayer = 3f;
    [SerializeField] private bool spawnOnStart = true;

    private DungeonGenerator dungeonGenerator;
    private System.Random random;

    void Start()
    {
        dungeonGenerator = FindObjectOfType<DungeonGenerator>();
        if (dungeonGenerator == null)
        {
            Debug.LogError("DungeonGenerator not found in scene!");
            return;
        }

        random = new System.Random(Time.time.GetHashCode());

        dungeonGenerator.OnDungeonGenerated += HandleDungeonGenerated;

        if (spawnOnStart && dungeonGenerator.FloorPositions.Count > 0)
        {
            SpawnBoxes();
        }
    }

    void OnDestroy()
```

```
    {
        if (dungeonGenerator != null)
        {
            dungeonGenerator.OnDungeonGenerated -= HandleDungeonGenerated;
        }
    }

    private void HandleDungeonGenerated()
    {
        SpawnBoxes();
    }

    public void SpawnBoxes()
    {
        if (boxPrefabs == null || boxPrefabs.Length == 0)
        {
            Debug.LogWarning("No box prefabs assigned!");
            return;
        }

        List<Vector3Int> floorPositions = dungeonGenerator.FloorPositions;
        if (floorPositions == null || floorPositions.Count == 0)
        {
            Debug.LogWarning("No floor positions available!");
            return;
        }

        int boxesToSpawn = random.Next(minBoxes, maxBoxes + 1);
        int boxesSpawned = 0;

        List<Vector3Int> shuffledPositions = new List<Vector3Int>(floorPositions);
        Shuffle(shuffledPositions);

        foreach (Vector3Int floorPos in shuffledPositions)
        {
            if (boxesSpawned >= boxesToSpawn) break;

            if (random.NextDouble() < boxSpawnChance)
            {
                Vector3 worldPos =
dungeonGenerator.GroundTilemap.GetCellCenterWorld(floorPos);

                if (Vector3.Distance(worldPos, dungeonGenerator.Player.position) >
minDistanceFromPlayer)
                {
                    SpawnSingleBox(worldPos);
                    boxesSpawned++;
                }
            }
        }

        Debug.Log($"Spawned {boxesSpawned} boxes");
    }

    private void SpawnSingleBox(Vector3 position)
    {
        GameObject boxPrefab = boxPrefabs[random.Next(0, boxPrefabs.Length)];
        GameObject box = Instantiate(boxPrefab, position, Quaternion.identity,
transform);

        box.name = "Box_" + boxPrefab.name;
        box.tag = "Box";
    }
```

```
    private void Shuffle<T>(IList<T> list)
    {
        int n = list.Count;
        while (n > 1)
        {
            n--;
            int k = random.Next(n + 1);
            T value = list[k];
            list[k] = list[n];
            list[n] = value;
        }
    }

    [ContextMenu("Respawn Boxes")]
    public void RespawnBoxes()
    {
        foreach (Transform child in transform)
        {
            Destroy(child.gameObject);
        }

        SpawnBoxes();
    }
      }
```

Лістинг А.3 – скрипт «EntitySpawner»

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

[System.Serializable]
public class EnemyType
{
    public GameObject enemyPrefab;
    [Range(0.1f, 1f)] public float spawnChance = 0.5f;
    public int minCount = 1;
    public int maxCount = 3;
    public float minDistanceFromPlayer = 5f;
}

public class EntitySpawner : MonoBehaviour
{
    [Header("Enemy Settings")]
    [SerializeField] private List<EnemyType> enemyTypes = new List<EnemyType>();
    [SerializeField] private float spawnIntensity = 0.5f;
    [SerializeField] private int maxTotalEnemies = 20;
    [SerializeField] private bool spawnOnStart = true;
    [SerializeField] private int maxSpawnAttempts = 10;

    [Header("Debug")]
    [SerializeField] private bool showSpawnPoints = false;
    [SerializeField] private Color spawnPointColor = Color.red;

    private DungeonGenerator dungeonGenerator;
    private List<Vector3Int> availableSpawnPoints = new List<Vector3Int>();
    private List<GameObject> spawnedEnemies = new List<GameObject>();

    private void OnEnable()
    {
        dungeonGenerator = FindObjectOfType<DungeonGenerator>();
        if (dungeonGenerator == null)
        {
            Debug.LogError("DungeonGenerator not found!");
            return;
```

```
        }

        dungeonGenerator.OnDungeonGenerated += HandleDungeonGenerated;
    }

    private void OnDisable()
    {
        if (dungeonGenerator != null)
            dungeonGenerator.OnDungeonGenerated -= HandleDungeonGenerated;
    }

    void Start()
    {
        if (spawnOnStart && dungeonGenerator != null &&
dungeonGenerator.FloorPositions.Count > 0)
            SpawnEnemies();
    }

    private void HandleDungeonGenerated()
    {
        Debug.Log("Dungeon generation event received");
        SpawnEnemies();
    }

    public void SpawnEnemies()
    {
        ClearExistingEnemies();
        PrepareSpawnPoints();

        int totalEnemiesToSpawn = Mathf.RoundToInt(maxTotalEnemies * spawnIntensity);
        int enemiesSpawned = 0;

        foreach (var enemyType in enemyTypes)
        {
            if (enemiesSpawned >= totalEnemiesToSpawn) break;

            int enemiesToSpawn = Mathf.RoundToInt(
                Random.Range(enemyType.minCount, enemyType.maxCount + 1) * spawnIntensity
            );

            for (int i = 0; i < enemiesToSpawn; i++)
            {
                if (enemiesSpawned >= totalEnemiesToSpawn || availableSpawnPoints.Count
== 0)
                    break;

                if (Random.value <= enemyType.spawnChance)
                {
                    if (TrySpawnEnemy(enemyType))
                        enemiesSpawned++;
                }
            }
        }

        Debug.Log($"Spawned {enemiesSpawned} enemies");
    }

    private bool TrySpawnEnemy(EnemyType enemyType)
    {
        for (int attempt = 0; attempt < maxSpawnAttempts; attempt++)
        {
            int randomIndex = Random.Range(0, availableSpawnPoints.Count);
            Vector3Int spawnPos = availableSpawnPoints[randomIndex];
```

```
            Vector3 worldPos =
dungeonGenerator.GroundTilemap.GetCellCenterWorld(spawnPos);

            if (Vector3.Distance(worldPos, dungeonGenerator.Player.position) >=
enemyType.minDistanceFromPlayer)
            {
                GameObject enemy = Instantiate(
                    enemyType.enemyPrefab,
                    worldPos,
                    Quaternion.identity,
                    transform
                );

                spawnedEnemies.Add(enemy);
                availableSpawnPoints.RemoveAt(randomIndex);
                return true;
            }
        }
        return false;
    }

    private void PrepareSpawnPoints()
    {
        availableSpawnPoints.Clear();
        if (dungeonGenerator == null || dungeonGenerator.GroundTilemap == null) return;

        availableSpawnPoints.AddRange(dungeonGenerator.FloorPositions);

        Vector3Int playerCellPos =
dungeonGenerator.GroundTilemap.WorldToCell(dungeonGenerator.Player.position);
        availableSpawnPoints.RemoveAll(pos =>
            Vector3Int.Distance(pos, playerCellPos) < 5
        );
    }

    private void ClearExistingEnemies()
    {
        foreach (var enemy in spawnedEnemies)
        {
            if (enemy != null)
                Destroy(enemy);
        }
        spawnedEnemies.Clear();
    }

    [ContextMenu("Respawn Enemies")]
    public void RespawnEnemies()
    {
        SpawnEnemies();
    }

    void OnDrawGizmos()
    {
        if (!showSpawnPoints || availableSpawnPoints.Count == 0 || dungeonGenerator ==
null)
            return;

        foreach (var pos in availableSpawnPoints)
        {
            Gizmos.color = spawnPointColor;
            Gizmos.DrawWireCube(
                dungeonGenerator.GroundTilemap.GetCellCenterWorld(pos),
                dungeonGenerator.GroundTilemap.cellSize * 0.8f
            );
```

```
        }
    }
      }
```

Лістинг А.4 – скрипт «CameraFollow»

```
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    public Transform target;
    public Vector3 offset = new Vector3(0, 0, -10);
    public float smoothSpeed = 5f;

    void LateUpdate()
    {
        if (target != null)
        {
            Vector3 desiredPosition = target.position + offset;
            transform.position = Vector3.Lerp(transform.position, desiredPosition,
smoothSpeed * Time.deltaTime);
        }
    }
}
```

Лістинг А.5 – скрипт «ExitSpawn»

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

public class PortalGenerator : MonoBehaviour
{
    [Header("Door Settings")]
    [SerializeField] private GameObject exitDoorPrefab;
    [SerializeField] private float minDistanceFromPlayer = 5f;
    [SerializeField] private bool spawnOnStart = true;

    private DungeonGenerator dungeonGenerator;
    private System.Random random;
    private GameObject exitDoorInstance;

    void Start()
    {
        dungeonGenerator = FindObjectOfType<DungeonGenerator>();
        if (dungeonGenerator == null)
        {
            Debug.LogError("DungeonGenerator not found in scene!");
            return;
        }

        random = new System.Random(Time.time.GetHashCode());

        dungeonGenerator.OnDungeonGenerated -= HandleDungeonGenerated;
        dungeonGenerator.OnDungeonGenerated += HandleDungeonGenerated;

        if (spawnOnStart && dungeonGenerator.FloorPositions.Count > 0)
        {
            RespawnExitDoor();
        }
    }

    void OnDestroy()
    {
        if (dungeonGenerator != null)
        {
```

```
            dungeonGenerator.OnDungeonGenerated -= HandleDungeonGenerated;
        }
    }

    private void HandleDungeonGenerated()
    {
        RespawnExitDoor();
    }

    public void SpawnExitDoor()
    {
        if (exitDoorPrefab == null)
        {
            Debug.LogWarning("Exit door prefab is not assigned!");
            return;
        }

        if (exitDoorInstance != null)
        {
            Debug.LogWarning("Exit door already exists! Skipping spawn.");
            return;
        }

        List<Vector3Int> floorPositions = dungeonGenerator.FloorPositions;
        if (floorPositions == null || floorPositions.Count == 0)
        {
            Debug.LogWarning("No floor positions available!");
            return;
        }

        List<Vector3Int> shuffledPositions = new List<Vector3Int>(floorPositions);
        Shuffle(shuffledPositions);

        foreach (Vector3Int floorPos in shuffledPositions)
        {
            Vector3 worldPos =
dungeonGenerator.GroundTilemap.GetCellCenterWorld(floorPos);

            if (Vector3.Distance(worldPos, dungeonGenerator.Player.position) >
minDistanceFromPlayer)
            {
                SpawnSingleDoor(worldPos);
                Debug.Log("Exit door spawned at: " + worldPos);
                return;
            }
        }

        Debug.LogWarning("Failed to find a valid position for the exit door!");
    }

    private void SpawnSingleDoor(Vector3 position)
    {
        if (exitDoorInstance != null)
        {
            Destroy(exitDoorInstance);
        }

        exitDoorInstance = Instantiate(exitDoorPrefab, position, Quaternion.identity,
transform);
        exitDoorInstance.name = "ExitDoor";
    }

    private void Shuffle<T>(IList<T> list)
    {
```

```
        int n = list.Count;
        while (n > 1)
        {
            n--;
            int k = random.Next(n + 1);
            T value = list[k];
            list[k] = list[n];
            list[n] = value;
        }
    }

    [ContextMenu("Respawn Exit Door")]
    public void RespawnExitDoor()
    {
        if (exitDoorInstance != null)
        {
            Destroy(exitDoorInstance);
        }

        SpawnExitDoor();
    }
}
```

Лістинг А.6 – скрипт «PlayerMovement»

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class PlayerMovement : MonoBehaviour
{
    public float moveSpeed = 5f;
    private Rigidbody2D rb;
    private Animator animator;
    private Vector2 movement;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
    }

    void Update()
    {
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");

        animator.SetFloat("Speed", movement.sqrMagnitude);

        if (movement.x != 0)
            GetComponent<SpriteRenderer>().flipX = movement.x < 0;
    }

    void FixedUpdate()
    {
        rb.MovePosition(rb.position + movement.normalized * moveSpeed *
Time.fixedDeltaTime);
    }

    public void IncreaseSpeed(float amount)
    {
        moveSpeed += amount;
        Debug.Log("Швидкість гравця збільшена до: " + moveSpeed);
    }
}
```

Лістинг А.7 – скрипт «PlayerBlink»

```
using UnityEngine;
using System.Collections;

public class PlayerBlink : MonoBehaviour
{
    public float blinkDistance = 5f;
    public float blinkCooldown = 2f;
    public float blinkOutDuration = 0.2f;
    public LayerMask blinkObstacles;

    private float lastBlinkTime = -Mathf.Infinity;
    private Rigidbody2D rb;
    private Animator animator;
    private bool isBlinking = false;
    private SpriteRenderer spriteRenderer;
    private Collider2D playerCollider;

    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        spriteRenderer = GetComponent<SpriteRenderer>();
        playerCollider = GetComponent<Collider2D>();
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            TryStartBlink();
        }
    }

    private void TryStartBlink()
    {
        if (!isBlinking && Time.time >= lastBlinkTime + blinkCooldown)
        {
            StartCoroutine(Blink());
        }
    }

    private IEnumerator Blink()
    {
        isBlinking = true;


        yield return new WaitForSeconds(blinkOutDuration);

        SetBlinkVisibility(false);

        Vector2 blinkDirection = GetBlinkDirection();

        Vector2 blinkTarget = (Vector2)transform.position + blinkDirection *
blinkDistance;
        RaycastHit2D hit = Physics2D.Raycast(transform.position, blinkDirection,
blinkDistance, blinkObstacles);
        if (hit.collider != null)
        {
            blinkTarget = hit.point - blinkDirection * 0.5f;
        }
```

```
        rb.position = blinkTarget;

        SetBlinkVisibility(true);

        isBlinking = false;
        lastBlinkTime = Time.time;
    }

    private void SetBlinkVisibility(bool visible)
    {
        spriteRenderer.enabled = visible;
        playerCollider.enabled = visible;
    }

    private Vector2 GetBlinkDirection()
    {
        Vector2 inputDirection = new Vector2(
            Input.GetAxisRaw("Horizontal"),
            Input.GetAxisRaw("Vertical")
        ).normalized;

        if (inputDirection.magnitude > 0.1f)
            return inputDirection;

        return spriteRenderer.flipX ? Vector2.left : Vector2.right;
    }
}
```

Лістинг А.8 – скрипт «PlayerShooting»

```
using UnityEngine;

public class PlayerShooting : MonoBehaviour
{
    [Header("Options")]
    public GameObject projectilePrefab;
    public float projectileSpeed = 10f;
    public float attackCooldown = 0.5f;
    public bool autoFire = false;

    [Header("Effects")]
    public AudioClip shootSound;
    public GameObject shootEffect;

    private float nextAttackTime = 0f;
    private Camera mainCamera;

    void Start()
    {
        mainCamera = Camera.main;
        if (mainCamera == null)
        {
            Debug.LogError("Main camera not found!");
        }
    }

    void Update()
    {
        bool fireInput = autoFire ? Input.GetMouseButton(0) :
Input.GetMouseButtonDown(0);

        if (fireInput && Time.time >= nextAttackTime)
        {
            ShootProjectile();
```

```
            nextAttackTime = Time.time + attackCooldown;
        }
    }

    void ShootProjectile()
    {
        if (projectilePrefab == null)
        {
            Debug.LogError("Projectile Prefab is not assigned!");
            return;
        }

        Vector3 mousePosition = mainCamera.ScreenToWorldPoint(Input.mousePosition);
        Vector2 direction = (mousePosition - transform.position).normalized;

        GameObject projectile = Instantiate(projectilePrefab, transform.position,
Quaternion.identity);

        Rigidbody2D rb = projectile.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            rb.linearVelocity = direction * projectileSpeed;
        }
        else
        {
            Debug.LogError("Projectile does not have a Rigidbody2D component!");
        }

        Collider2D projectileCollider = projectile.GetComponent<Collider2D>();
        Collider2D playerCollider = GetComponent<Collider2D>();
        if (projectileCollider != null && playerCollider != null)
        {
            Physics2D.IgnoreCollision(projectileCollider, playerCollider);
        }

        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        projectile.transform.rotation = Quaternion.Euler(0, 0, angle);

        PlayShootEffects();
    }

    void PlayShootEffects()
    {
        if (shootSound != null)
        {
            AudioSource.PlayClipAtPoint(shootSound, transform.position);
        }

        if (shootEffect != null)
        {
            Instantiate(shootEffect, transform.position, Quaternion.identity);
        }
    }

    public void SetCooldown(float newCooldown)
    {
        attackCooldown = Mathf.Max(0, newCooldown);
    }
      }
```

Лістинг А.9 – скрипт «PlayerHealth»

```
using UnityEngine;
using System;
```

```
public class PlayerHealth : MonoBehaviour
{
    [Header("Налаштування")]
    [SerializeField] private int maxHealth = 100;
    [SerializeField] private int currentHealth;

    public event Action<int> OnHealthChanged;
    public int MaxHealth => maxHealth;
    public int CurrentHealth => currentHealth;

    private void Awake()
    {
        currentHealth = maxHealth;
        Debug.Log($"Початкове HP: {currentHealth}/{maxHealth}");
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        currentHealth = Mathf.Max(currentHealth, 0);
        Debug.Log($"Отримано {damage} шкоди. HP: {currentHealth}/{maxHealth}");

        OnHealthChanged?.Invoke(currentHealth);

        if (currentHealth <= 0) Die();
    }

    public void Heal(int healAmount)
    {
        int oldHealth = currentHealth;
        currentHealth = Mathf.Min(currentHealth + healAmount, maxHealth);
        Debug.Log($"Відновлено {currentHealth - oldHealth} HP. Тепер HP:
{currentHealth}/{maxHealth}");

        OnHealthChanged?.Invoke(currentHealth);
    }

    public bool IsFullHealth() => currentHealth >= maxHealth;

    private void Die()
    {
        Debug.Log("Гравець помер! HP: 0/" + maxHealth);
    }

    [ContextMenu("Test Take Damage")]
    private void TestDamage() => TakeDamage(10);

    [ContextMenu("Test Heal")]
    private void TestHeal() => Heal(15);
      }
```

Лістинг А.10 – скрипт «PlayerHeal»

```
using UnityEngine;
using System.Collections;

public class PlayerHeal : MonoBehaviour
{
    [Header("Heal Settings")]
    [SerializeField] private int healAmount = 20;
    [SerializeField] private float healCooldown = 2f;
    [SerializeField] private KeyCode healKey = KeyCode.H;
    private bool canHeal = true;
```

```
    [Header("Effects")]
    [SerializeField] private ParticleSystem healParticles;
    [SerializeField] private AudioClip healSound;
    private AudioSource audioSource;

    private PlayerHealth playerHealth;

    private void Awake()
    {
        playerHealth = GetComponent<PlayerHealth>();
        audioSource = GetComponent<AudioSource>();
        Debug.Log("Система лікування ініціалізована");
    }

    private void Update()
    {
        if (Input.GetKeyDown(healKey))
        {
            TryHeal();
        }
    }

    private void TryHeal()
    {
        if (!canHeal)
        {
            Debug.Log("Лікування на кд!");
            return;
        }

        if (playerHealth.IsFullHealth())
        {
            Debug.Log("У вас повне HP! Лікування не потрібне");
            return;
        }

        Debug.Log($"Спроба лікування (+{healAmount} HP)...");
        playerHealth.Heal(healAmount);
        PlayHealEffects();
        StartCoroutine(HealCooldown());
    }

    private void PlayHealEffects()
    {
        if (healParticles != null) healParticles.Play();
        if (healSound != null && audioSource != null) audioSource.PlayOneShot(healSound);
        Debug.Log("Ефекти лікування активовано");
    }

    private IEnumerator HealCooldown()
    {
        canHeal = false;
        Debug.Log($"КД лікування: {healCooldown} сек");
        yield return new WaitForSeconds(healCooldown);
        canHeal = true;
        Debug.Log("Лікування знову доступне!");
    }
      }
```

Лістинг А.11 – скрипт «SlimePatrol»

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

```
[RequireComponent(typeof(Rigidbody2D))]
public class SlimePatrol : MonoBehaviour
{
    [Header("Movement Settings")]
    public float moveSpeed = 1.5f;
    public float waitTime = 2f;
    public float patrolRadius = 5f;
    public LayerMask obstacleLayers;

    [Header("Debug")]
    [SerializeField] private bool drawGizmos = true;

    private Rigidbody2D rb;
    private Vector2 startPos;
    private Vector2 targetPos;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        startPos = transform.position;
        StartCoroutine(PatrolRoutine());
    }

    IEnumerator PatrolRoutine()
    {
        while (true)
        {
            yield return new WaitForSeconds(waitTime);

            int attempts = 0;
            do
            {
                targetPos = startPos + Random.insideUnitCircle * patrolRadius;
                attempts++;
                if (attempts > 10)
                {
                    targetPos = startPos;
                    break;
                }
            }
            while (Physics2D.Linecast(transform.position, targetPos, obstacleLayers));

            while (Vector2.Distance(transform.position, targetPos) > 0.1f)
            {
                Vector2 direction = (targetPos - (Vector2)transform.position).normalized;
                rb.MovePosition(rb.position + direction * moveSpeed *
Time.fixedDeltaTime);
                yield return new WaitForFixedUpdate();
            }
        }
    }

    private void OnDrawGizmos()
    {
        if (!drawGizmos) return;

        Gizmos.color = Color.cyan;
        Gizmos.DrawWireSphere(Application.isPlaying ? startPos :
(Vector2)transform.position, patrolRadius);

        if (Application.isPlaying)
        {
            Gizmos.color = Color.yellow;
```

```
            Gizmos.DrawLine(transform.position, targetPos);
        }
    }
      }
```

Лістинг А.12 – скрипт «PlayerHeal»

Лістинг А.13 – скрипт «GoblinPatrol»

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Rigidbody2D))]
public class GoblinPatrol : MonoBehaviour
{
    public float moveSpeed = 3f;
    public float waitTime = 2f;
    public float patrolRadius = 5f;
    public LayerMask obstacleLayers;

    private Animator animator;
    private Rigidbody2D rb;
    private Vector2 startPos;
    private Vector2 targetPos;
    private bool isMoving = false;

    private GoblinAggro aggroScript;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        startPos = transform.position;

        // Знаходимо скрипт агро
        aggroScript = GetComponent<GoblinAggro>();

        // Перевірка на наявність скрипту агро
        if (aggroScript == null)
        {
            Debug.LogError("GoblinAggro script not found on the same GameObject!");
        }
        else
        {
            StartCoroutine(Patrol());
        }
    }

    IEnumerator Patrol()
    {
        while (true)
        {
            if (aggroScript != null && aggroScript.IsAggro())
                yield break;

            yield return new WaitForSeconds(waitTime);

            targetPos = startPos + Random.insideUnitCircle * patrolRadius;

            RaycastHit2D hit = Physics2D.Linecast(transform.position, targetPos,
obstacleLayers);
            if (hit.collider != null)
            {
                continue;
```

```
            }

            isMoving = true;
            animator.SetBool("isWalking", true);

            while (Vector2.Distance(transform.position, targetPos) > 0.1f)
            {
                Vector2 dir = (targetPos - (Vector2)transform.position).normalized;
                rb.MovePosition(rb.position + dir * moveSpeed * Time.fixedDeltaTime);
                yield return new WaitForFixedUpdate();
            }

            // Зупинка руху
            isMoving = false;
            animator.SetBool("isWalking", false);
        }
    }
}
```

Лістинг А.14 – скрипт «GoblinAttackProjectile»

```
using UnityEngine;

public class GoblinAttackProjectile : MonoBehaviour
{
    [Header("Налаштування")]
    public int damage = 10;
    public LayerMask playerLayer;
    public Sprite hitSprite;
    public float spriteDisplayTime = 0.3f;

    private SpriteRenderer spriteRenderer;

    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    public void Setup(float lifeTime, float effectTime)
    {
        Destroy(gameObject, lifeTime);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (((1 << collision.gameObject.layer) & playerLayer) != 0)
        {
            if (hitSprite != null)
            {
                GameObject hitObject = new GameObject("HitSprite");
                SpriteRenderer hitRenderer = hitObject.AddComponent<SpriteRenderer>();
                hitRenderer.sprite = hitSprite;
                hitRenderer.sortingOrder = 10;

                hitObject.transform.position = transform.position;

                Destroy(hitObject, spriteDisplayTime);
            }

            PlayerHealth playerHealth = collision.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
            }
```

```
            GetComponent<Collider2D>().enabled = false;
            if (spriteRenderer != null)
            {
                spriteRenderer.enabled = false;
            }
        }
    }
      }
```

Лістинг А.15 – скрипт «GoblinAttack»

```
using UnityEngine;
using System.Collections;

public class GoblinAttack : MonoBehaviour
{
    public float attackRange = 1.5f;
    public float attackCooldown = 1f;
    public GameObject attackPrefab;
    public LayerMask playerLayer;
    private bool canAttack = true;

    public float attackLifetime = 0.5f;

    void Start()
    {
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (((1 << collision.gameObject.layer) & playerLayer) != 0)
        {
            if (canAttack)
            {
                StartCoroutine(Attack(collision.transform.position));
            }
        }
    }

    IEnumerator Attack(Vector2 playerPosition)
    {
        canAttack = false;

        GameObject attack = Instantiate(attackPrefab, playerPosition,
Quaternion.identity);

        Destroy(attack, attackLifetime);

        yield return new WaitForSeconds(attackCooldown);
        canAttack = true;
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
}
```

Лістинг А.16 – скрипт «GoblinAggro»

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class GoblinAggro : MonoBehaviour
{
    public float aggroRange = 5f;
    public float attackRange = 1f;
    public float chaseSpeed = 4f;

    private Transform player;
    private Rigidbody2D rb;
    private Animator animator;
    private bool isAggro = false;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").transform;

        if (player == null)
        {
            Debug.LogError("Player not found! Make sure player has 'Player' tag.");
        }
    }

    void Update()
    {
        if (player == null) return;

        float distanceToPlayer = Vector2.Distance(transform.position, player.position);

        if (distanceToPlayer <= aggroRange)
        {
            isAggro = true;

            if (distanceToPlayer <= attackRange)
            {
                Attack();
            }
            else
            {
                ChasePlayer();
            }
        }
        else
        {
            isAggro = false;
        }
    }

    void ChasePlayer()
    {
        Vector2 direction = (player.position - transform.position).normalized;
        rb.linearVelocity = direction * chaseSpeed;

        if (animator != null)
        {
            animator.SetBool("isWalking", true);
        }
    }

    void Attack()
    {
```

```
        if (animator != null)
        {
            animator.SetBool("isWalking", false);
        }
    }

    public bool IsAggro()
    {
        return isAggro;
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, aggroRange);

        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
      }
```

Лістинг А.17 – скрипт «GhostAttackProjectile»

```
using UnityEngine;

public class GhostAttackProjectile : MonoBehaviour
{
    [Header("Налаштування")]
    public int damage = 10;
    public float lifetime = 3f;
    public LayerMask playerLayer;
    public Sprite hitSprite;
    public float spriteDisplayTime = 0.3f;

    private SpriteRenderer spriteRenderer;
    private GhostAttackProjectile GhostAttack;
    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        Destroy(gameObject, lifetime);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (((1 << collision.gameObject.layer) & playerLayer) != 0)
        {
            if (hitSprite != null)
            {
                GameObject hitObject = new GameObject("HitSprite");
                SpriteRenderer hitRenderer = hitObject.AddComponent<SpriteRenderer>();
                hitRenderer.sprite = hitSprite;
                hitRenderer.sortingOrder = 10;
                hitObject.transform.position = transform.position;
                Destroy(hitObject, spriteDisplayTime);
            }

            PlayerHealth playerHealth = collision.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
            }

            Destroy(gameObject);
        }
```

```
    }
}
```

Лістинг А.18 – скрипт «GhostPatrol»

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Rigidbody2D))]
public class GhostPatrol : MonoBehaviour
{
    public float moveSpeed = 3.5f;
    public float waitTime = 1.5f;
    public float patrolRadius = 6f;
    public LayerMask obstacleLayers;

    private Animator animator;
    private Rigidbody2D rb;
    private Vector2 startPos;
    private Vector2 targetPos;
    private bool isMoving = false;

    private GhostAggro aggroScript;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        startPos = transform.position;

        aggroScript = GetComponent<GhostAggro>();
        if (aggroScript == null)
        {
            Debug.LogError("GhostAggro script not found on the same GameObject!");
        }
        else
        {
            StartCoroutine(Patrol());
        }
    }

    IEnumerator Patrol()
    {
        while (true)
        {
            if (aggroScript != null && aggroScript.IsAggro())
            {
                Debug.Log("Aggro active - pausing patrol");
                yield return new WaitUntil(() => !aggroScript.IsAggro());
                continue;
            }
            animator.SetBool("isWalking", true);

            yield return new WaitForSeconds(waitTime);

            targetPos = startPos + Random.insideUnitCircle * patrolRadius;

            RaycastHit2D hit = Physics2D.Linecast(transform.position, targetPos,
obstacleLayers);
            if (hit.collider != null)
                continue;

            isMoving = true;
            if (animator != null)
```

```
            {
                animator.SetBool("isWalking", true);

                Vector2 dir = (targetPos - (Vector2)transform.position).normalized;
            }

            while (Vector2.Distance(transform.position, targetPos) > 0.1f)
            {
                if (aggroScript != null && aggroScript.IsAggro())
                    break;

                Vector2 dir = (targetPos - (Vector2)transform.position).normalized;
                rb.MovePosition(rb.position + dir * moveSpeed * Time.fixedDeltaTime);
                yield return new WaitForFixedUpdate();
            }

            isMoving = false;
            if (animator != null)
            {
                animator.SetBool("isWalking", false);
            }
        }
    }
}
```

Лістинг А.19 – скрипт «GhostAttack»

```
using UnityEngine;

public class GhostAttack : MonoBehaviour
{
    public GameObject projectilePrefab;
    public float attackCooldown = 3f;
    public float projectileSpeed = 10f;
    public float attackRange = 5f;

    private float nextAttackTime = 0f;
    private Transform player;
    private GhostAggro aggroScript;

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        aggroScript = GetComponent<GhostAggro>();
    }

    void Update()
    {
        if (Time.time >= nextAttackTime && aggroScript.IsAggro())
        {
            float distanceToPlayer = Vector2.Distance(transform.position,
player.position);

            if (distanceToPlayer <= attackRange)
            {
                Attack();
                nextAttackTime = Time.time + attackCooldown;
            }
        }
    }

    void Attack()
```

```
    {
        if (projectilePrefab == null) return;

        Vector2 direction = (player.position - transform.position).normalized;

        Vector2 firePoint = (Vector2)transform.position + direction * 0.5f; // 0.5f -
відстань від центру привида

        GameObject projectile = Instantiate(projectilePrefab, firePoint,
Quaternion.identity);

        Rigidbody2D rb = projectile.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            rb.linearVelocity = direction * projectileSpeed;

            rb.gravityScale = 0f;
        }
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
}
```

Лістинг А.20 – скрипт «GhostAggro»

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class GhostAggro : MonoBehaviour
{
    public float aggroRange = 8f;
    public float safeDistance = 3f;
    public float moveSpeed = 4.5f;
    public float retreatSpeed = 3f;

    private Transform player;
    private Rigidbody2D rb;
    private Animator animator;
    private bool isAggro = false;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").transform;

        if (player == null)
        {
            Debug.LogError("Player not found! Make sure the player has the 'Player'
tag.");
        }
    }

    void Update()
    {
        if (player == null) return;

        float distanceToPlayer = Vector2.Distance(transform.position, player.position);

        if (distanceToPlayer <= aggroRange)
```

```
    {
        isAggro = true;

        if (distanceToPlayer <= safeDistance)
        {
            Retreat();
        }
        else
        {
            ChasePlayer();
        }

        UpdateAnimation();
    }
    else
    {
        isAggro = false;
        rb.linearVelocity = Vector2.zero;
        if (animator != null)
        {
            animator.SetBool("isWalking", false);
        }
    }
}

void UpdateAnimation()
{
    if (animator == null || rb.linearVelocity.magnitude < 0.1f) return;

    Vector2 direction = rb.linearVelocity.normalized;
    animator.SetBool("isWalking", true);
}

void ChasePlayer()
{
    Vector2 direction = (player.position - transform.position).normalized;
    rb.linearVelocity = direction * moveSpeed;

    if (animator != null)
    {
        animator.SetBool("isWalking", true);
    }
}

void Retreat()
{
    Vector2 direction = (transform.position - player.position).normalized;
    rb.linearVelocity = direction * retreatSpeed;

    if (animator != null)
    {
        animator.SetBool("isWalking", true);
    }
}

public bool IsAggro()
{
    return isAggro;
}

void OnDrawGizmosSelected()
{
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position, aggroRange);
```

```
        Gizmos.color = Color.cyan;
        Gizmos.DrawWireSphere(transform.position, safeDistance);
    }
}
```

Лістинг А.21 – скрипт «BossProjectile»

```
using UnityEngine;

public class BossAttackProjectile : MonoBehaviour
{
    [Header("Налаштування")]
    public int damage = 10;
    public LayerMask playerLayer;
    public GameObject hitEffectPrefab;
    public float lifeTime = 5f;
    public float hitEffectLifeTime = 0.3f;

    private SpriteRenderer spriteRenderer;

    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        Setup(lifeTime);
    }

    public void Setup(float lifeTime)
    {
        Destroy(gameObject, lifeTime);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (((1 << collision.gameObject.layer) & playerLayer) != 0)
        {
            if (hitEffectPrefab != null)
            {
                GameObject hitEffect = Instantiate(hitEffectPrefab, transform.position,
Quaternion.identity);
                Destroy(hitEffect, hitEffectLifeTime);
            }

            PlayerHealth playerHealth = collision.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
            }

            GetComponent<Collider2D>().enabled = false;
            if (spriteRenderer != null)
            {
                spriteRenderer.enabled = false;
            }

            Destroy(gameObject);
        }
    }
}
```

Лістинг А.22 – скрипт «BossAttack»

```
using UnityEngine;
```

```
using System.Collections;

public class BossBladeAttack : MonoBehaviour
{
    public GameObject bladePrefab;
    public float bladeSpeed = 2f;
    public float attackCooldown = 5f;
    public int bladesPerAttack = 20;

    private BossAggro bossAggro;
    private bool isAttacking = false;

    void Start()
    {
        bossAggro = GetComponent<BossAggro>();
    }

    void Update()
    {
        if (bossAggro.IsAggro() && !isAttacking)
        {
            StartCoroutine(BladeAttackRoutine());
        }
    }

    private IEnumerator BladeAttackRoutine()
    {
        isAttacking = true;

        int pattern = Random.Range(1, 4);

        switch (pattern)
        {
            case 1:
                StartCoroutine(SpiralPattern());
                break;
            case 2:
                StartCoroutine(CircularPattern());
                break;
            case 3:
                StartCoroutine(CrossPattern());
                break;
        }

        yield return new WaitForSeconds(attackCooldown);
        isAttacking = false;
    }

    private IEnumerator SpiralPattern()
    {
        for (int i = 0; i < bladesPerAttack; i++)
        {
            float angle = i * (360f / bladesPerAttack);
            SpawnBlade(angle);
            yield return new WaitForSeconds(0.1f);
        }
    }

    private IEnumerator CircularPattern()
    {
        for (int i = 0; i < bladesPerAttack; i++)
        {
            float angle = i * (360f / bladesPerAttack);
            SpawnBlade(angle);
```

```
        }
        yield return null;
    }

    private IEnumerator CrossPattern()
    {
        for (int i = 0; i < 4; i++)
        {
            SpawnBlade(i * 90f);
        }
        yield return null;
    }

    private void SpawnBlade(float angle)
    {
        Vector2 direction = new Vector2(Mathf.Cos(angle * Mathf.Deg2Rad), Mathf.Sin(angle
* Mathf.Deg2Rad));
        GameObject blade = Instantiate(bladePrefab, transform.position,
Quaternion.identity);
        Rigidbody2D rb = blade.GetComponent<Rigidbody2D>();
        rb.linearVelocity = direction * bladeSpeed;
    }
      }
```

Лістинг А.23 – скрипт «BossAggro»

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class BossAggro : MonoBehaviour
{
    public float speed = 2f;
    public float stopDistance = 3f;
    private GameObject player;
    private Rigidbody2D rb;
    private Animator animator;
    private bool isAggro = false;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player");

        if (player == null)
        {
            Debug.LogError("Player not found! Make sure the player has the 'Player'
tag.");
        }
    }

    void Update()
    {
        if (!isAggro || player == null) return;

        float distanceToPlayer = Vector2.Distance(transform.position,
player.transform.position);

        if (distanceToPlayer > stopDistance)
        {
            ChasePlayer();
        }
        else
        {
```

```
            StopChasing();
        }

        UpdateAnimation();
    }

    void ChasePlayer()
    {
        Vector2 direction = (player.transform.position - transform.position).normalized;
        rb.linearVelocity = direction * speed;
    }

    void StopChasing()
    {
        rb.linearVelocity = Vector2.zero;
    }

    void UpdateAnimation()
    {
        if (rb.linearVelocity.magnitude > 0f)
        {
            if (animator != null)
            {
                animator.SetBool("isMoving", true);
            }
        }
        else
        {
            if (animator != null)
            {
                animator.SetBool("isMoving", false);
            }
        }
    }

    public void TriggerAggro()
    {
        isAggro = true;
    }

    public bool IsAggro()
    {
        return isAggro;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("PlayerAttack"))
        {
            TriggerAggro();
        }
    }
}
```

Лістинг А.24 – скрипт «SpeedPotion»

```
using UnityEngine;

public class SpeedPotion : MonoBehaviour
{
    public float speedBoost = 0.2f;
```

```
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            PlayerMovement playerMovement = collision.GetComponent<PlayerMovement>();
            if (playerMovement != null)
            {
                playerMovement.IncreaseSpeed(speedBoost);
                Debug.Log("Move speed boosted: +" + speedBoost);
            }
            Destroy(gameObject);
        }
    }
}
```

Лістинг А.25 – скрипт «Meatheal»

```
using UnityEngine;

public class FoodHeal : MonoBehaviour
{
    public int healAmount = 10;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            PlayerHealth playerHealth = collision.GetComponent<PlayerHealth>();
            if (playerHealth != null && !playerHealth.IsFullHealth())
            {
                playerHealth.Heal(healAmount);
                Debug.Log("Restore " + healAmount + " HP!");
                Destroy(gameObject);
            }
        }
    }
}
```