

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОТОТИПУ
ROGUELIKE ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ РІВНІВ
ТА ІНТЕЛЕКТУАЛЬНОЮ ПОВЕДІНКОЮ ВОРОГІВ
SOFTWARE IMPLEMENTATION OF A ROGUELIKE GAME
PROTOTYPE WITH PROCEDURAL LEVEL GENERATION
AND ALGORITHM-DRIVEN ENEMY BEHAVIOR

Виконав: студент 4 курсу, групи КНТ-113сп
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)
Інженерія програмного забезпечення

ПАНАСЮК О.О

(ПРИЗВИЩЕ та ініціали)

Керівник КОЛПАКОВА Т.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент КОЦУР М.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ПАНАСЮКА Олександра Олексійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація прототипу roguelike гри з процедурною генерацією рівнів та інтелектуальною поведінкою ворогів. Software Implementation of a Roguelike Game Prototype with Procedural Level Generation and Algorithm-driven Enemy Behavior

керівник проєкту (роботи) к.т.н., доцент, КОЛПАКОВА Тетяна Олексіївна,
(ПРИЗВИЩЕ, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “7” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Проєктування програмного забезпечення. 3. Основні рішення щодо реалізації компонентів системи. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	КОЛПАКОВА Т.О., доцент		
Нормоконтроль	КАЛІНІНА М. В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент (ка)

_____ Олександр ПАНАСЮК
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Тетяна КОЛПАКОВА
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра:
82 с., 5 табл., 22 рис., 2 дод., 15 джерел.

ROGUELIKE, ПРОЦЕДУРНА ГЕНЕРАЦІЯ РІВНІВ, ПОВЕДІНКОВІ
ПАТЕРНИ ВОРОГІВ, КОМП'ЮТЕРНА ГРА, UNITY.

Об'єкт дослідження – процес розробки двовимірної комп'ютерної гри жанру roguelike.

Предмет дослідження – методи реалізації процедурної генерації рівнів і поведінкових патернів ворогів у комп'ютерній грі.

Мета роботи – розробка прототипу двовимірної roguelike-гри з процедурною генерацією рівнів та алгоритмічною поведінкою ворогів на базі скінченного автомата станів.

Матеріали, методи та технічні засоби: мова програмування C#, ігровий рушій Unity, середовище розробки Microsoft Visual Studio, ОС Microsoft Windows.

Галузь використання – прототип може слугувати основою для розробки повноцінних ігрових проєктів жанру roguelike та у навчальних цілях.

Результати. Розроблено прототип двовимірної roguelike-гри з виглядом зверху. Реалізовано процедурну генерацію рівнів на основі BFS-подібного алгоритму з автоматичним призначенням ролей кімнатам та динамічною побудовою коридорів. Поведінку ворогів реалізовано через скінченний автомат станів з алгоритмами обходу перешкод і розштовхування ворогів. Також реалізовано систему бою з обробкою зіткнень та завданням шкоди.

Висновки. Розроблено функціональний прототип двовимірної комп'ютерної гри жанру roguelike мовою C# на рушії Unity, що реалізує процедурну генерацію рівнів та алгоритмічну поведінку ворогів і є базою для подальшого розвитку проєкту.

ABSTRACT

Explanatory note to the final qualifying work of the bachelor: 82 pages, 22 figures, 5 tables, 2 appendixes, 15 sources.

ROGUELIKE, PROCEDURAL LEVEL GENERATION, ENEMY BEHAVIOR PATTERNS, COMPUTER GAME, UNITY.

Object of research – the process of developing a two-dimensional roguelike computer game.

Subject of research – methods of implementing procedural level generation and enemy behavior patterns in a computer game.

Purpose of the work – development of a 2D roguelike game prototype featuring procedural level generation and algorithmic enemy behavior based on a finite state machine.

Materials, methods and tools: C# programming language, Unity game engine, Microsoft Visual Studio IDE, Microsoft Windows OS.

Field of application – the prototype can serve as a foundation for developing full-scale roguelike game projects and for educational purposes.

Results. A top-down 2D roguelike game prototype has been developed. Procedural level generation was implemented using a BFS-based algorithm with automatic room role assignment and dynamic corridor construction. Enemy behavior was implemented via a finite state machine (Idle, Patrol, Chase, Attack) with obstacle avoidance and enemy separation algorithms. A combat system with collision handling and damage dealing was also implemented.

Conclusions. A functional prototype of a 2D roguelike computer game has been developed in C# using the Unity engine, implementing procedural level generation and algorithmic enemy behavior, and serving as a foundation for further project development.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ	9
1 Аналіз предметної області	10
1.1 Визначення жанру Roguelike	10
1.2 Механіки жанру та їх роль в ігровому процесі	11
1.3 Аналіз існуючих ігор у жанрі Roguelike	12
2 Проєктування програмного забезпечення	18
2.1 Вимоги до системи	18
2.2 Вибір інструментів розробки	19
2.2.1 Порівняння ігрових рушіїв	19
2.2.2 Обґрунтування вибору Unity	22
2.2.3 Вибір середовища розробки	23
2.3 Архітектура програмного забезпечення	23
2.4 Проєктування системи процедурної генерації рівнів	23
2.5 Проєктування поведінкових патернів ворогів	23
2.6 Висновки за розділом 2	32
3 Основні рішення щодо реалізації компонентів системи	33
3.1 Структура проєкту	33
3.2 Реалізація системи процедурної генерації рівнів	35
3.2.1 Генерація сітки та BFS-розширення	35
3.2.2 Призначення ролей та побудова коридорів	36
3.3 Реалізація поведінки ворогів	37
3.4 Реалізація бойової системи	42
3.5 Графічний інтерфейс користувача	43
3.6 Висновки за розділом 3	43
4 Експлуатація, тестування та експериментальне дослідження програми	45
4.1 Призначення та умови виконання програми	45

4.2 Інструкція з експлуатації 46

4.3 Результати роботи програми 46

4.4 Тестування програми 48

4.5 Висновки за розділом 4.....49

Висновки.....51

Перелік джерел посилання 53

Додаток А Текст програми 55

Додаток Б Слайди презентації.....77

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

2D – двовимірний простір або двовимірна графіка;

BFS – Breadth-First Search, алгоритм обходу в ширину;

BSP – Binary Space Partitioning, метод бінарного поділу простору;

C# – мова програмування, що використовується для написання скриптів;

ECS – Entity Component System;

FSM – Finite State WMachine, скінченний автомат станів;

GML – GameMaker Language, мова програмування ігрового рушія;

HUD – Heads-Up Display, елементи інтерфейсу;

IDE – Integrated Development Environment;

UI – User Interface, користувацький інтерфейс;

VS Code – Visual Studio Code, редактор коду компанії Microsoft;

ЛКМ – ліва кнопка миші;

ОС – операційна система;

ПК – персональний комп'ютер;

ПЗ – програмне забезпечення;

ТЗ – технічне завдання.

ВСТУП

Ринок комп'ютерних ігор продовжує активно зростати, і жанр roguelike займає в ньому помітне місце. Ігри на зразок Hades, Dead Cells та Enter the Gungeon показали, що процедурна генерація рівнів у поєднанні з продуманою поведінкою ворогів можуть зробити гру цікавою навіть після десятків годин проходження – кожна спроба відчувається по-новому [1]-[3].

З технічної точки зору roguelike – це жанр, який вимагає одразу кількох нетривіальних рішень: як генерувати рівні так, щоб вони були прохідними і різноманітними, як змусити ворогів поводитись природньо без надмірної складності алгоритмів, як зв'язати все це в єдину ігрову систему. Саме тому розробка подібного прототипу є хорошою практичною задачею для програміста [4]-[5].

Метою даної роботи є створення прототипу двовимірної roguelike-гри з процедурною генерацією рівнів та поведінкою ворогів на основі скінченного автомата станів з використанням Unity та C# [4].

Актуальність теми пов'язана з попитом на розробників ігор і з практичним інтересом до алгоритмів процедурної генерації як способу створювати різноманітний ігровий контент без ручного проєктування кожного рівня.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Розробка ігрового програмного забезпечення вимагає чіткого розуміння жанрових особливостей та існуючих технічних підходів. Жанр roguelike має свою специфіку, яка безпосередньо визначає архітектурні рішення при розробці – зокрема, необхідність процедурної генерації рівнів та гнучкої поведінки ворогів.

У цьому розділі розглянуто ключові характеристики жанру, проаналізовано та порівняно сучасні ігри-представники з точки зору реалізації основних механік, а також визначено вимоги до розроблюваного прототипу.

1.1 Визначення жанру Roguelike

Roguelike – жанр комп'ютерних ігор, що сформувався на початку 1980-х років і з того часу виріс в окремий і впізнаваний напрямок ігрової індустрії. Його визначальні риси залишаються незмінними попри десятиліття еволюції: процедурно згенерований ігровий світ, незворотна загибель персонажа та висока варіативність ігрового процесу.

Ключовою особливістю жанру є процедурна генерація рівнів – підхід, за якого структура ігрового простору формується алгоритмічно при кожному новому запуску, а не створюється вручну розробником. Завдяки цьому кожна спроба проходження є унікальною: гравець ніколи не знає заздалегідь, яка кімната чекає його за наступним поворотом, де розміщені вороги і які предмети вдасться знайти. Саме ця непередбачуваність є головним джерелом реграбельності жанру [4].

Не менш важливою складовою є система незворотної загибелі. Вона означає, що смерть персонажа скасовує весь прогрес поточного забігу – гравець повертається до самого початку без можливості завантажити збереження. На перший погляд це може здатися надмірно суворим, однак саме ця механіка формує унікальну емоційну напругу: кожне рішення має вагу, а кожна помилка

– наслідки. Це докорінно відрізняє жанр від більшості інших, де гравець може безкарно повторювати спроби з останньої контрольної точки [6].

З часом жанр зазнав значної еволюції. Сучасні ігри нерідко пом'якшують класичні правила: частина з них дозволяє зберігати певний прогрес між забігами, інші відмовляються від покрокової механіки на користь динамічного бою в реальному часі. Такі ігри прийнято відносити до піджанру roguelite, хоча чіткої межі між двома поняттями досі не існує. Незважаючи на ці зміни, процедурна генерація та висока варіативність залишаються незмінними ознаками жанру й досі [6].

1.2 Механіки жанру та їх роль в ігровому процесі

Попри різноманіття сучасних ігор жанру, всі вони побудовані на спільному фундаменті з кількох взаємопов'язаних механік. Жодна з них окремо не визначає жанр – саме їх поєднання створює той характерний ігровий досвід, який гравці впізнають як roguelike [6].

Процедурна генерація є основою реграбельності. Кожен новий забіг формує унікальну карту: кімнати, коридори, розташування ворогів і предметів щоразу змінюються. Це унеможливлює механічне заучування рівня і змушує гравця щоразу адаптуватись до нових умов. Саме тому сотня годин у roguelike рідко відчувається як повторення – кожна спроба ставить нові задачі [4].

Система незворотної загибелі надає кожному рішенням реальну вагу. Коли помилка коштує всього прогресу, гравець починає інакше ставитись до кожного бою, кожного предмета і кожного ризику. Це створює емоційну напругу, яку важко відтворити іншими засобами. При цьому добре спроектований roguelike не карає гравця за невдачу несправедливо – смерть має бути наслідком рішення, а не випадковості [6].

Поведінка ворогів у жанрі – це не просто перешкода на шляху, а повноцінний ігровий виклик. Різноманітні патерни пересування та атак змушують гравця читати ситуацію і реагувати на неї, а не діяти за шаблоном.

Чим різноманітнішим є бестіарій, тим більше тактичних варіантів постає перед гравцем у кожній сутичці [5].

Окремо варто виділити ресурсний менеджмент. Здоров'я, боєприпаси, предмети – усе це обмежено і вимагає свідомого управління. Витратити ресурс зараз чи зберегти на потім, вступити в бій чи обійти ворога – такі рішення формують тактичну глибину, яка і відрізняє roguelike від більш лінійних жанрів [6].

1.3 Аналіз існуючих ігор у жанрі Roguelike

Для кращого розуміння жанру та визначення орієнтирів при розробці було проведено огляд кількох популярних ігор. Порівняння здійснювалось за чотирма критеріями: візуальна складова та інтерфейс, керування персонажем, інтелект ворогів та якість генерації рівнів. Вибір цих критеріїв зумовлений тим, що саме вони найбільш повно відображають ігровий досвід у жанрі roguelike і дозволяють виявити як сильні сторони існуючих рішень, так і аспекти, що потребують вдосконалення [1]-[3].

1.3.1 Hades

Hades – roguelike-гра з виглядом зверху, розроблена студією Supergiant Games. Гравець керує сином бога підземного царства та намагається вибратись із пекла, долаючи хвилі ворогів у послідовності процедурно згенерованих кімнат [1].

З точки зору візуальної складової та інтерфейсу гра є одним із найяскравіших прикладів у жанрі. Деталізована графіка, виконана у стилі грецької міфології, плавні анімації персонажів та виразні ефекти атак формують цілісну і впізнавану естетику. Інтерфейс при цьому залишається мінімалістичним – на екрані відображається лише необхідна інформація, що не

відволікає від ігрового процесу. Приклад інтерфейсу гри зображений на рисунку 1.1.



Рисунок 1.1 – Інтерфейс гри Hades [1]

Керування побудовано на поєднанні клавіш WASD та миші і відчувається надзвичайно відзивчиво. Кожна здібність персонажа має чітку механіку і власний ритм використання, що робить бойову систему глибокою попри зовнішню простоту. Гравець отримує широкі можливості для комбінування атак і адаптації стилю гри під поточну ситуацію.

Система ворогів у Hades відзначається різноманітністю як візуально, так і з точки зору поведінки. Кожен тип противника має унікальний набір атак і патернів пересування, що змушує гравця постійно змінювати тактику залежно від складу кімнати. Боси є особливо опрацьованими – їх поведінка ускладнюється з кожним новим забігом.

Генерація рівнів у грі базується на системі заздалегідь створених кімнат-шаблонів, які збираються у випадковій послідовності при кожному новому запуску. Гравець сам обирає напрямок руху між кімнатами, що додає елемент стратегічного планування [1], [4].

1.3.2 Dead Cells

Dead Cells – це гра в жанрі roguelike з елементами, притаманними дослідницьким екшен-іграм, створена студією Motion Twin. У ній гравець бере під контроль таємничого героя, який досліджує процедурно згенерований замок, протистоїть численним ворогам і з кожною новою спробою відкриває альтернативні шляхи проходження [2].

Візуальна складова гри побудована на піксельній графіці, яка попри свою стилістичну простоту виглядає надзвичайно виразно завдяки якісній анімації та динамічним ефектам атак. Кожна локація має власну палітру і атмосферу, що робить переміщення між зонами помітним і виразним. Інтерфейс мінімалістичний – на екрані присутнє лише те, що дійсно потрібно гравцю в поточний момент. Приклад інтерфейсу гри зображений на рисунку 1.2.

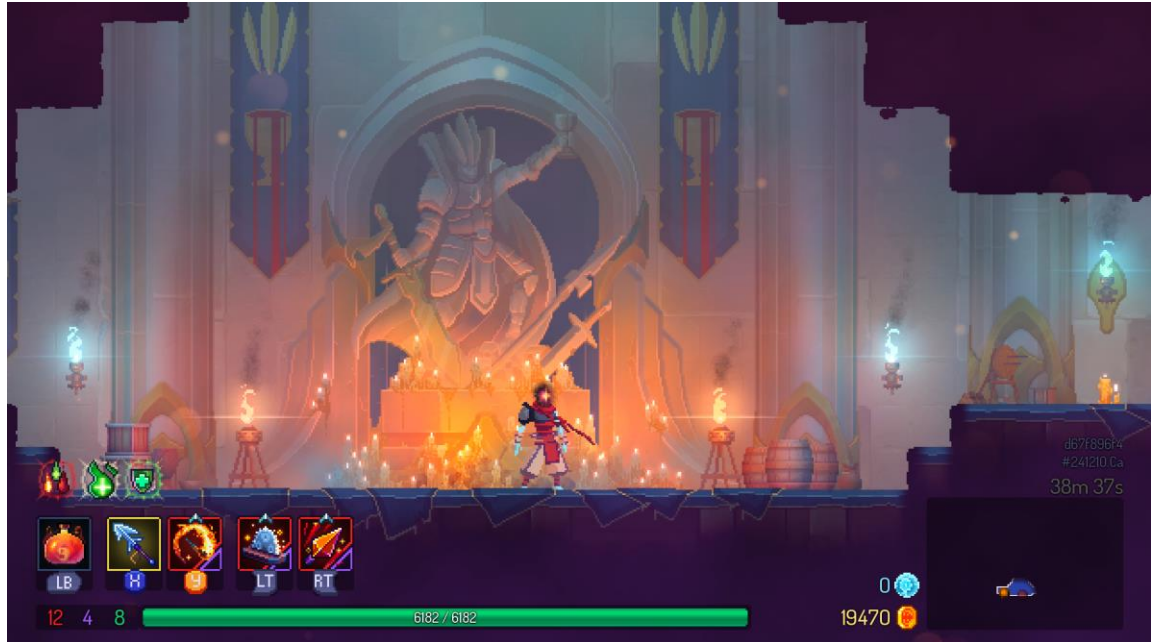


Рисунок 1.2 – Інтерфейс гри Dead Cells [2]

Керування є однією з найсильніших сторін гри. Система атак, ухилень і здібностей відпрацьована до дрібниць: кожна дія відчувається чіткою і миттєво

реагує на введення гравця. Dead Cells часто наводять як приклад досконалого ігрового відчуття від керування – так зване відчуття «game feel». Підтримка як клавіатури з мишею, так і геймпада робить гру доступною для широкої аудиторії.

Вороги у Dead Cells мають добре опрацьовану поведінку: кожен тип противника використовує унікальний набір атак із чіткими візуальними сигналами перед їх виконанням. Це робить бої читабельними – гравець завжди має можливість відреагувати на загрозу, якщо уважний. З ростом складності вороги стають агресивнішими, а їх комбінації в одній кімнаті – більш непередбачуваними.

Генерація рівнів базується на сітці фіксованих блоків-секцій, які збираються у випадковому порядку при кожному новому запуску. Між блоками динамічно прокладаються з'єднання, що забезпечує зв'язність рівня. Такий підхід поєднує передбачуваність окремих секцій із варіативністю загальної структури маршруту.

1.3.3 Enter the Gungeon

Enter the Gungeon – roguelike-гра з виглядом зверху, розроблена студією Dodge Roll. Гравець обирає одного з кількох персонажів і намагається пробитись крізь процедурно згенероване підземелля у пошуках легендарної зброї, здатної знищити минуле [3].

Візуальна складова гри виконана у піксельному стилі з високим рівнем деталізації. Анімації персонажів, ворогів та ефекти пострілів і вибухів ретельно опрацьовані та додають динаміки кожній сутичці. Інтерфейс лаконічний – базова інформація про здоров'я, броню та поточну зброю відображається ненав'язливо і не перевантажує екран.

Керування побудовано на поєднанні клавіш WASD для пересування, миші для прицілювання та стрільби і пробілу для ухилення перекатом. Механіка ухилення є центральною для ігрового процесу – своєчасний перека

дозволяє уникнути будь-якого пострілу, що робить бої читабельними і водночас вимогливими до реакції гравця. Така схема керування відповідає усталеним конвенціям жанру twin-stick shooter і є інтуїтивно зрозумілою для гравців, знайомих із подібними іграми. Водночас простота введення не знижує глибини бойової системи: ефективне використання перекаату потребує читання патернів атак ворогів та прийняття швидких тактичних рішень.

Система ворогів у Enter the Gungeon є однією з найбільш різноманітних у жанрі. Кожен тип противника має унікальний патерн стрільби у стилі bullet hell – складні візерунки куль, що вимагають від гравця постійного руху та уважного читання ситуації. Боси мають кілька фаз поведінки, що ускладнюються в міру зниження їхнього здоров'я. Приклад інтерфейсу гри під час поєдинку з босом зображений на рисунку 1.3.

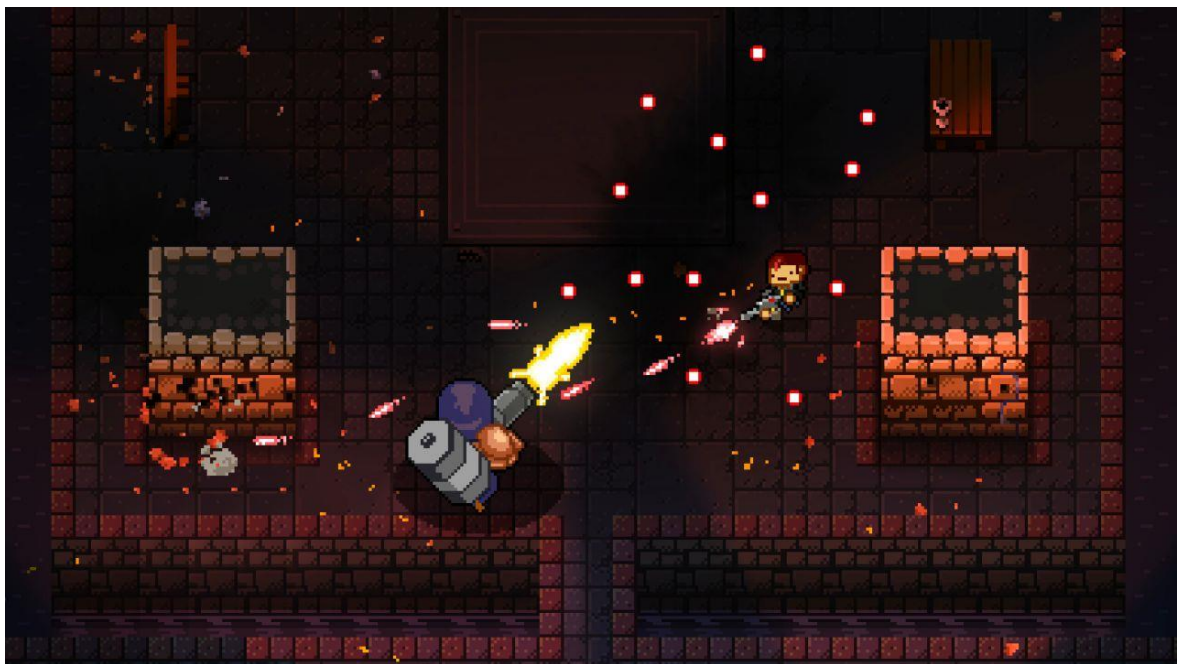


Рисунок 1.3 – Інтерфейс гри Enter the Gungeon [3]

Процедурна генерація рівнів базується на наборі заздалегідь створених кімнат-шаблонів, що збираються у випадковому порядку на кожному поверсі підземелля. Кожен поверх має власну тематику, унікальний набір ворогів та

характерне оформлення, що забезпечує відчуття прогресії та урізноманітнює кожен новий забіг [6].

1.3.4 Порівняння існуючих аналогів ігор у жанрі Roguelike

У п. 1.3.1–1.3.3 було проаналізовано існуючі аналоги ігор жанру Roguelike. Згідно з визначеними критеріями, у таблиці 1.1 наведено порівняння з оцінками кожної характеристики за шкалою від 1 до 5.

Таблиця 1.1 – Порівняльна таблиця

Назва гри	Графіка та інтерфейс	Зручність керування	Інтелект ворогів	Генерація рівнів
Hades	5	5	5	4
Dead Cells	5	5	4	4
Enter the Gungeon	5	5	5	5

1.3.5 Висновки за розділом 1

У розділі було виконано аналіз та порівняння трьох існуючих ігор жанру – Hades, Dead Cells та Enter the Gungeon. Аналіз показав, що всі три гри демонструють високий рівень реалізації бойової системи та керування, тоді як підходи до генерації рівнів і поведінки ворогів суттєво відрізняються між собою.

Отримані результати враховано при проєктуванні власного прототипу: зокрема, при визначенні підходу до процедурної генерації рівнів та розробці поведінкових патернів ворогів на основі скінченного автомата станів.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі описано проєктування програмного забезпечення. Розглянуто такі питання: вимоги до системи; архітектура розроблюваної системи; вибір та аналіз ігрового рушія; вибір інтегрованого середовища розробки (IDE); проєктування системи процедурної генерації рівнів та поведінкових патернів ворогів.

2.1 Вимоги до системи

Розроблюваний прототип комп'ютерної гри жанру roguelike повинен забезпечувати стабільну та коректну роботу на персональному комп'ютері під управлінням ОС Windows, мати зрозумілий інтерфейс і відзивчive керування.

До функціональних вимог належать:

- переміщення персонажа гравця по ігровому рівні за осями X та Y;
- процедурна генерація ігрових рівнів при кожному новому запуску;
- автоматичне призначення ролей кімнатам (стартова, звичайна, з ворогами, зі скарбами, босова);
- динамічна побудова коридорів між суміжними кімнатами;
- поведінка ворогів на основі скінченного автомата станів (Idle, Patrol, Chase, Attack);
- система бою з обробкою зіткнень та завданням шкоди;
- логіка програшу при загибелі персонажа;
- навігаційне меню для запуску та завершення гри.

До нефункціональних вимог належать:

- стабільна частота кадрів без суттєвих просідань під час генерації рівня та активного бою;
- модульна архітектура коду, що дозволяє легко розширювати функціонал;

– коректна обробка крайових ситуацій (застрягання ворогів, недосяжні точки патрулювання тощо).

2.2 Вибір інструментів розробки

2.2.1 Порівняння ігрових рушіїв

При розробці прототипу комп'ютерної гри жанру roguelike було проведено аналіз існуючих ігрових рушіїв. Розглядались три основні варіанти: Unity, Godot Engine та GameMaker Studio.

Unity – кросплатформний рушій для розробки 2D та 3D ігор, розроблений компанією Unity Technologies. Рушій підтримує мову програмування C# як основну мову написання скриптів, що робить його доступним для широкого кола розробників завдяки зрозумілому синтаксису та великій кількості навчальних матеріалів. Unity активно розвивається та регулярно отримує оновлення, що додають нові інструменти та покращують продуктивність [7].

Одною з ключових переваг Unity є великий Asset Store – офіційний магазин ресурсів, де розробники можуть знайти готові моделі, анімації, звуки, скрипти та цілі системи механік. Це суттєво прискорює розробку, дозволяючи зосередитись на реалізації унікальних елементів проєкту замість створення базових компонентів з нуля. Широка спільнота розробників і детальна офіційна документація забезпечують швидке вирішення більшості технічних питань.

Архітектура Unity побудована на системі компонентів: кожен ігровий об'єкт (GameObject) є контейнером, до якого прикріплюються компоненти (Component), що визначають його поведінку. Такий підхід забезпечує високу модульність і гнучкість коду – окремі системи легко переносити між проєктами і розширювати без зміни вже існуючої логіки. Рушій також надає потужні вбудовані інструменти: Physics2D для обробки зіткнень у двовимірному просторі, Tilemap для побудови рівнів із тайлів, Animator для створення та управління анімаціями персонажів і об'єктів, а також систему префабів для

динамічного створення об'єктів у runtime – що є особливо важливим при процедурній генерації рівнів [7]-[11].

Godot Engine – безкоштовний ігровий рушій з повністю відкритим вихідним кодом, що розробляється та підтримується спільнотою. Відсутність будь-яких ліцензійних відрахувань робить його привабливим вибором для інді-розробників та студентських проєктів. Рушій активно розвивається: кожна нова версія суттєво розширює функціонал та покращує стабільність роботи.

Godot підтримує дві основні мови програмування: власну мову GDScript, синтаксис якої схожий на Python і орієнтований на швидке написання ігрової логіки, а також C# – для розробників, яким важлива строга типізація та сумісність з іншими платформами. Підтримка C# є корисною, однак реалізована дещо обмеженіше порівняно з Unity: частина інструментів рушія доступна лише через GDScript, що може ускладнити розробку для тих, хто звик працювати виключно з C#.

Інтерфейс редактора Godot вважається одним із найпростіших серед сучасних рушіїв – він інтуїтивний, не перевантажений зайвими панелями і дозволяє швидко освоїтись навіть початківцям. Проте за обсягом офіційної документації та розміром спільноти Godot помітно поступається Unity: кількість готових рішень, туторіалів і сторонніх ресурсів значно менша, що може уповільнити розробку при виникненні нестандартних технічних задач. З огляду на це, для реалізації складних систем, таких як процедурна генерація рівнів та поведінкові патерни ворогів, Unity є більш обґрунтованим вибором завдяки зрілішій екосистемі та ширшій підтримці.

GameMaker Studio – ігровий рушій, орієнтований переважно на розробку двовимірних ігор. Він має довгу історію розвитку і свого часу був популярним інструментом серед інді-розробників завдяки низькому порогу входу та швидкому прототипуванню. На ньому створено кілька відомих ігор жанру roguelike, зокрема Undertale та Hotline Miami, що свідчить про його придатність для подібних проєктів на базовому рівні.

Основною мовою програмування в GameMaker Studio є власна мова GML (GameMaker Language), розроблена спеціально для цього рушія. З одного боку, GML є простою у освоєнні для початківців – вона має зрозумілий синтаксис і добре підходить для реалізації базової ігрової логіки. З іншого боку, знання GML практично не переносяться на інші платформи та рушії, що суттєво обмежує цінність набутих навичок з точки зору подальшого професійного розвитку. На відміну від C#, який використовується в Unity і є універсальною мовою програмування з широким застосуванням поза ігровою індустрією, GML залишається вузькоспеціалізованим інструментом.

Можливості масштабування проєктів у GameMaker Studio є помітно вужчими порівняно з Unity та Godot. Рушій не має повноцінної компонентної архітектури, що ускладнює організацію складних систем із великою кількістю взаємодіючих елементів.

Обґрунтування вибору ігрового рушія наведено у таблиці 2.1.

Таблиця 2.1 – Порівняння ігрових рушіїв

Критерій	Ігрові рушії		
	Unity	Godot Engine	GameMaker Studio
Підтримка C#	+	+/-	—
Зручність 2D-розробки	+	+	+
Обсяг документації	+	+/-	+/-
Розмір спільноти	+	+/-	—
Безкоштовність	+/-	+	—
Зручність для roguelike	+	+/-	—

Таким чином, для розробки прототипу обрано рушій Unity як найбільш зручний інструмент для 2D-розробки з підтримкою C# та широкими можливостями для реалізації процедурної генерації рівнів і системи поведінки ворогів.

2.2.2 Обґрунтування вибору Unity

Unity побудований на компонентній архітектурі: кожен ігровий об'єкт (GameObject) є контейнером, до якого прикріплюються компоненти з конкретною логікою. Це дозволяє будувати складні системи з незалежних модулів – наприклад, логіку руху ворога, його стан та обробку зіткнень можна реалізувати як окремі компоненти, не прив'язуючись до конкретної ієрархії класів. Така архітектура особливо важлива при реалізації скінченного автомата станів для поведінки ворогів, де кожен стан є самостійним класом із власною логікою [7], [9].

Для реалізації процедурної генерації рівнів Unity надає систему Tilemap – інструмент для побудови ігрових рівнів із тайлів безпосередньо на глобальній сцені. Це дозволяє динамічно малювати підлогу коридорів і видаляти тайли стін у місцях примикання кімнат у runtime, що є ключовою частиною алгоритму генерації рівнів даного прототипу. Система префабів, у свою чергу, дозволяє зберігати шаблони кімнат різних ролей і миттєво створювати їх екземпляри під час генерації рівня [7], [10].

Для реалізації бойової системи та поведінки ворогів Unity надає Physics2D з підтримкою Raycast – механізму випускання променів для перевірки прямої видимості між ворогом і гравцем, виявлення перешкод на шляху руху та реалізації алгоритму обходу стін. Animator забезпечує зручне управління анімаціями персонажів і ворогів залежно від їхнього поточного стану [7], [11].

Таким чином, Unity є найбільш обґрунтованим вибором для даного проєкту, оскільки його вбудовані інструменти безпосередньо відповідають

технічним вимогам реалізації як процедурної генерації рівнів, так і поведінкових патернів ворогів [7], [10], [11].

2.2.3 Вибір середовища розробки

При виборі середовища розробки для написання скриптів на мові C# розглядалось три варіанти: Microsoft Visual Studio, JetBrains Rider та Visual Studio Code.

Microsoft Visual Studio – це повнофункціональне інтегроване середовище розробки, створене компанією Microsoft, яке широко використовується для розробки програмного забезпечення різної складності, зокрема ігор. Середовище має офіційну інтеграцію з Unity (через інструменти Visual Studio Tools for Unity), що дозволяє безпосередньо взаємодіяти з проектом: відкривати скрипти, запускати налагодження та відстежувати виконання коду прямо під час роботи рушія [12].

Однією з ключових переваг Visual Studio є система інтелектуального автодоповнення коду IntelliSense, яка аналізує контекст і пропонує підказки, методи та змінні, що значно пришвидшує написання коду та зменшує кількість помилок. Вбудований відладчик підтримує встановлення точок зупинки, покрокове виконання програми, перегляд значень змінних у реальному часі та аналіз стеку викликів, що робить процес пошуку й усунення помилок значно ефективнішим.

Крім того, середовище забезпечує підсвічування синтаксису, рефакторинг коду, аналіз продуктивності та підтримку розширень, які дозволяють адаптувати IDE під конкретні потреби розробника. Visual Studio також має інтеграцію з системами контролю версій, такими як Git, що дає змогу керувати змінами в коді безпосередньо в середовищі розробки.

Безкоштовна версія Visual Studio Community надає практично повний набір функціональних можливостей і є достатньою для індивідуальних

розробників та невеликих команд, що робить її популярним вибором для навчальних і комерційних проєктів.

JetBrains Rider – це кросплатформне інтегроване середовище розробки від компанії JetBrains, орієнтоване на роботу з мовою C# та платформою .NET. IDE має повноцінну інтеграцію з Unity, що дозволяє ефективно працювати з ігровими проєктами: відкривати та редагувати скрипти, запускати відладку та аналізувати поведінку коду безпосередньо в контексті рушія.

Rider вирізняється потужною системою автодоповнення та статичного аналізу коду, яка базується на технологіях ReSharper. Це забезпечує глибоке розуміння структури проєкту, підказки щодо оптимізації, виявлення потенційних помилок ще на етапі написання коду та зручні інструменти рефакторингу. За рівнем інтелектуальності ці інструменти не поступаються можливостям Visual Studio, а в деяких аспектах, зокрема в аналізі коду та навігації по проєкту, можуть навіть перевершувати їх.

Середовище також підтримує роботу з системами контролю версій, такими як Git, має зручний інтерфейс і стабільно працює на різних операційних системах (Windows, macOS, Linux), що є важливою перевагою для кросплатформної розробки.

Водночас JetBrains Rider є комерційним продуктом і потребує придбання ліцензії або використання обмеженого пробного періоду. Це може ускладнювати його використання в навчальних проєктах або для індивідуальних розробників без доступу до безкоштовних освітніх ліцензій.

Visual Studio Code – це легкий і швидкий редактор коду, розроблений компанією Microsoft, який підтримує велику кількість розширень і гнучко налаштовується під різні задачі. Завдяки плагінам (зокрема для C#) його можна адаптувати для розробки на платформі .NET та частково інтегрувати з Unity.

Однак, попри широку функціональність і високу швидкість роботи, Visual Studio Code не є повноцінним інтегрованим середовищем розробки. У порівнянні з такими IDE, як Visual Studio або Rider, він поступається за глибиною інтеграції з Unity, можливостями відладки, а також зручністю роботи

з великими та складними проєктами. Багато функцій, які в IDE доступні «з коробки», у VS Code реалізуються через сторонні розширення, що може вимагати додаткового налаштування та не завжди забезпечує стабільний результат.

Обґрунтування вибору IDE наведено у таблиці 2.2.

Таблиця 2.2 – Порівняння середовищ розробки

Критерій	Visual Studio	JetBrains Rider	VS Code
Інтеграція з Unity	+	+	+/-
Підтримка C#	+	+	+/-
Вбудований відладчик	+	+	–
Безкоштовність	+	–	+
Зручність використання	+	+	+/-

Таким чином, для розробки прототипу комп'ютерної гри жанру roguelike обрано мову програмування C#, ігровий рушій Unity та середовище розробки Microsoft Visual Studio як оптимальний набір інструментів, що забезпечує зручність реалізації всіх ключових механік проєкту [7], [8], [12].

2.3 Архітектура програмного забезпечення

При проєктуванні архітектури прототипу комп'ютерної гри було розглянуто декілька підходів до організації програмного коду: монолітна архітектура, компонентно-орієнтована архітектура та архітектура на основі системи «Сутність–Компонент–Система» (Entity Component System, ECS) [9].

Монолітна архітектура передбачає зосередження логіки гри в єдиній або кількох великих класах. Такий підхід є простим для невеликих проєктів, однак суттєво ускладнює підтримку та розширення коду при збільшенні обсягу функціоналу.

Архітектура ECS розділяє дані (компоненти), логіку (системи) та ідентифікатори об'єктів (сутності). Вона забезпечує високу продуктивність завдяки cache-friendly розміщенню даних у пам'яті, проте має значний поріг входу та надмірну складність для невеликих прототипів.

Компонентно-орієнтована архітектура передбачає побудову ігрових об'єктів шляхом прикріплення до них незалежних компонентів-скриптів, кожен з яких відповідає за окремий аспект поведінки. Такий підхід є нативним для рушія Unity і забезпечує баланс між простотою реалізації та модульністю коду.

Порівняння архітектурних підходів наведено у таблиці 2.3.

Таблиця 2.3 – Порівняння архітектурних підходів

Критерій	Монолітна	Компонентна	ECS
Простота реалізації	+	+	–
Модульність	–	+	+
Масштабованість	–	+	+
Продуктивність	+/-	+/-	+
Відповідність Unity	–	+	+/-
Поріг входу	+	+	–

Отже, для розробки прототипу обрано компонентно-орієнтовану архітектуру як найбільш збалансований підхід, що органічно інтегрується з моделлю GameObject/Component рушія Unity.

Систему розділено на три незалежні підсистеми. Підсистема генерації рівнів відповідає за процедурну побудову ігрового простору під час завантаження сцени. Підсистема управління гравцем охоплює обробку введення, переміщення та логіку здоров'я. Підсистема поведінки ворогів реалізує автономну поведінку кожного ворога на основі скінченного автомата станів.

Взаємодія між підсистемами організована за принципом мінімальної зв'язаності: підсистема генерації рівнів розміщує ворогів шляхом інстанціювання префабів після завершення побудови рівня, тоді як кожен ворог самостійно відстежує позицію гравця. Така організація відповідає принципу єдиної відповідальності та забезпечує незалежне налагодження кожної з підсистем.

2.4 Проєктування системи процедурної генерації рівнів

Процедурна генерація рівнів – це підхід до створення ігрового контенту, при якому структура ігрового простору формується алгоритмічно під час запуску або завантаження гри, а не визначається вручну дизайнером. У жанрі roguelike цей підхід є ключовою механікою, оскільки забезпечує унікальність кожного проходження та підвищує реграбельність [4].

У сучасній розробці ігор застосовуються кілька поширених підходів до процедурної генерації рівнів. Перший – генерація на основі шаблонів кімнат (room template-based generation): дизайнер заздалегідь створює набір кімнат-шаблонів, а алгоритм обирає та з'єднує їх у випадковому порядку. Такий підхід забезпечує високу художню якість окремих кімнат при збереженні варіативності загальної структури рівня. Саме цей метод використовується у грі Hades: кожна кімната розроблена вручну, однак їх послідовність і набір ворогів визначаються алгоритмічно при кожному новому забігу, що дозволяє поєднати процедурність із ретельно вивіреним геймплейним балансом [1], [4].

Другий підхід – клітинні автомати та BSP-дерева (Binary Space Partitioning): ігровий простір рекурсивно поділяється на менші ділянки, у кожній з яких розміщується кімната, після чого між ними будуються коридори. Цей метод добре підходить для генерації щільних підземель з органічною структурою.

Третій підхід – сітковий метод із хвильовим розширенням: ігровий простір представляється у вигляді двовимірної сітки комірок, а розміщення

кімнат здійснюється шляхом поширення від початкової точки – наприклад, за допомогою алгоритму обходу у ширину (BFS). Це дозволяє гнучко контролювати форму рівня, щільність кімнат та гарантує зв'язність усього простору. Близький до цього підхід застосовується у грі Dead Cells: рівні генеруються на основі фіксованої сітки, де кімнати із заздалегідь заготовлених блоків розставляються процедурно, а коридори між ними прокладаються динамічно залежно від конфігурації сусідніх блоків [4], [13].

Порівняння підходів представлено у таблиці 2.4.

Таблиця 2.4 – Порівняння підходів до процедурної генерації рівнів

Критерій	Шаблони кімнат	BSP-дерево	Сітковий + BFS
Художній контроль	+	+/-	+/-
Варіативність структури	+/-	+	+
Гарантія зв'язності	+/-	+	+
Простота реалізації	+	-	+
Контроль ролей кімнат	+	+/-	+

Для реалізації прототипу обрано сітковий підхід із BFS-розширенням, оскільки він забезпечує гарантовану зв'язність рівня, дозволяє гнучко призначати ролі кімнатам залежно від їх відстані від старту та органічно поєднується з компонентною архітектурою рушія Unity. Детальна схема алгоритму генерації представлена на рисунку 2.1.

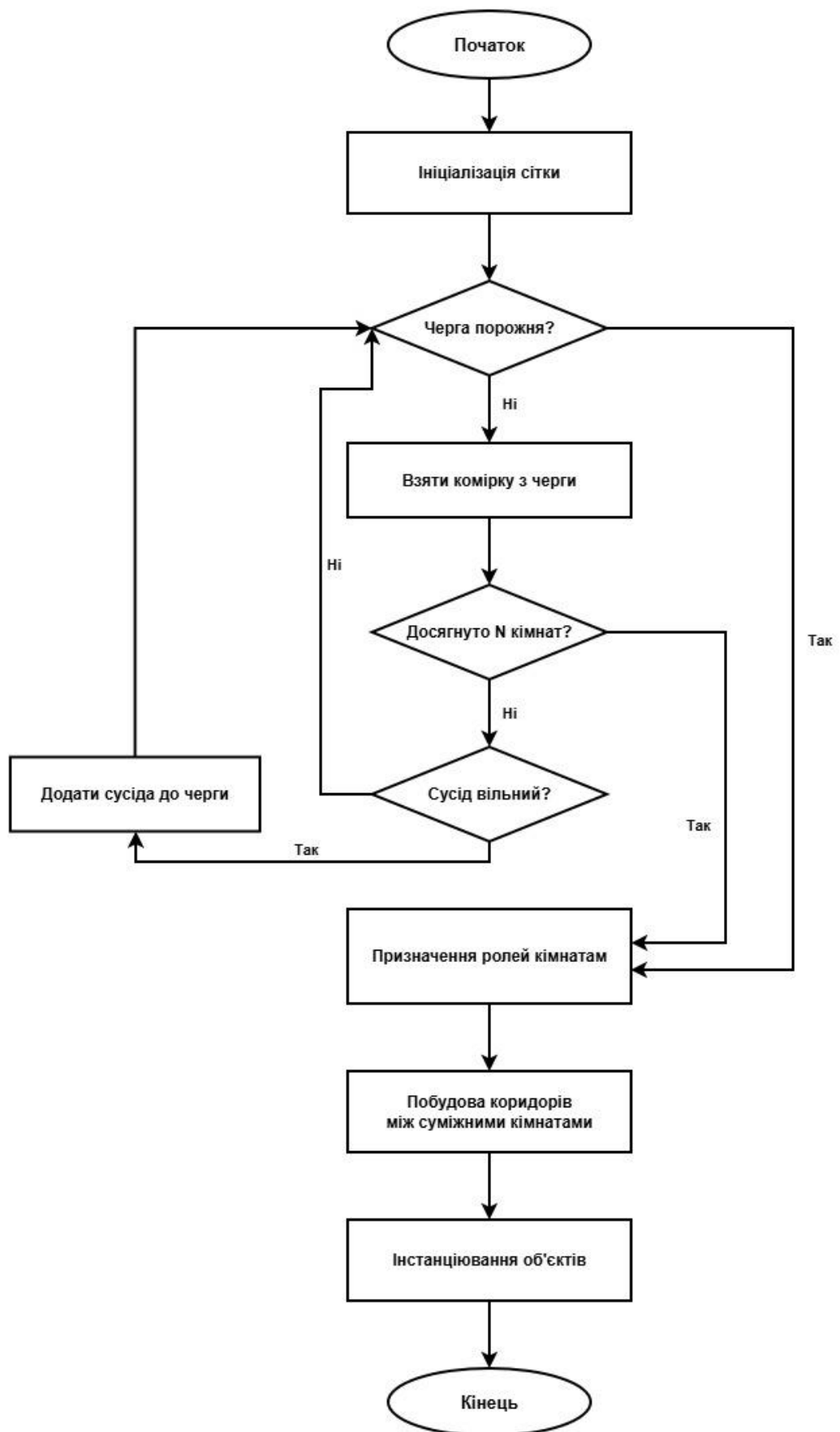


Рисунок 2.1 – Алгоритм генерації рівнів

2.5 Проєктування поведінкових патернів ворогів

Поведінка ворогів у комп'ютерних іграх є одним із ключових факторів, що визначають складність і відчуття занурення. У жанрі roguelike, де гравець постійно стикається з новими ситуаціями, важливо, щоб вороги поводитися передбачувано на рівні базових патернів, але при цьому реагували на дії гравця динамічно. Баланс між передбачуваністю та адаптивністю визначає, наскільки поведінка ворогів сприймається як осмислена, а не хаотична.

Одним із найпоширеніших підходів до реалізації поведінки ворогів є скінченний автомат станів (Finite State Machine, FSM). FSM – це модель обчислень, у якій об'єкт у кожен момент часу перебуває рівно в одному з кінцевого набору станів і переходить між ними залежно від зовнішніх умов. Завдяки чіткій структурі переходів між станами FSM добре піддається візуалізації та діагностиці на етапі розробки. Перевагами FSM є прозора логіка, простота налагодження та легкість розширення новими станами без зміни існуючих [14].

Альтернативними підходами є дерева поведінки (Behaviour Trees) та утилітарний ШІ (Utility AI). Дерева поведінки організовують логіку у вигляді ієрархічного дерева вузлів-умов і дій, що дозволяє описувати складнішу поведінку. Утилітарний ШІ присвоює числові оцінки можливим діям і обирає ту, що має найвищу оцінку в поточному контексті. Такий підхід є особливо виправданим у великих проєктах, де поведінка агентів залежить від значної кількості змінних середовища. Обидва підходи є потужнішими за FSM, але мають значно вищий поріг входу і надлишкову складність для невеликих прототипів [5], [14].

Для реалізації поведінки ворогів у даному прототипі обрано FSM з чотирма станами: Idle, Patrol, Chase та Attack. Діаграму переходів між станами представлено на рисунку 2.2.

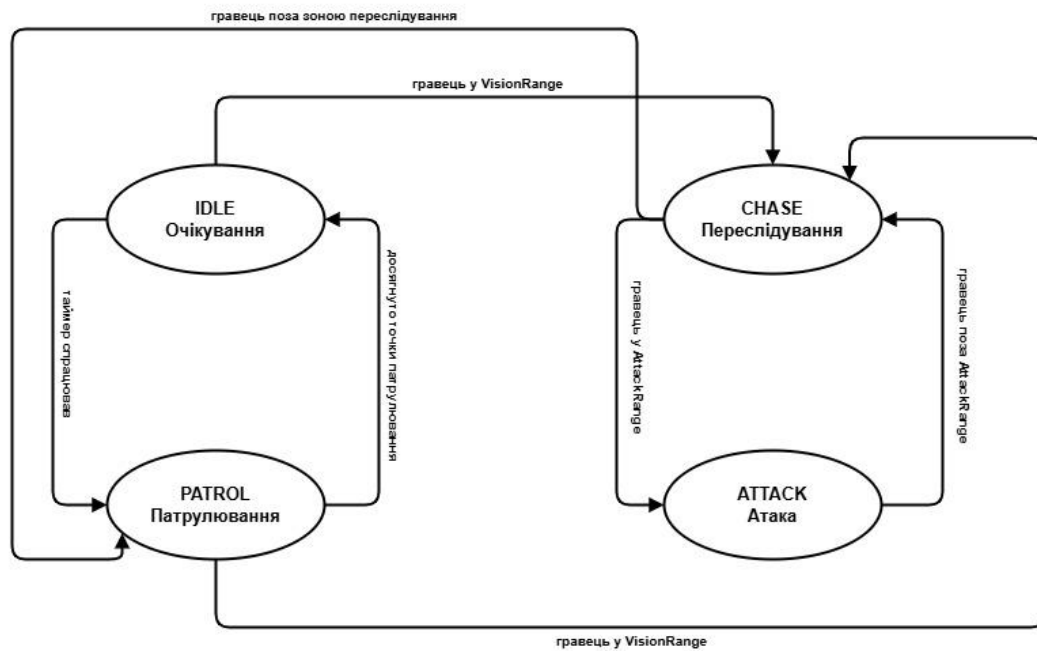


Рисунок 2.2 – Алгоритм поведінки ворогів

У стані Idle ворог перебуває нерухомо і очікує. Якщо гравець входить у радіус виявлення (VisionRange), автомат переходить у стан Chase. Якщо гравець не виявлений – через визначений інтервал часу ворог переходить у стан Patrol.

У стані Patrol ворог рухається до випадково обраної точки у своїй зоні. Перед кожним переміщенням виконується перевірка за допомогою Raycast, щоб уникнути руху крізь стіни. При виявленні гравця відбувається перехід у Chase. Після досягнення точки патрулювання ворог повертається в Idle.

У стані Chase ворог переслідує гравця. Для уникнення перешкод застосовується поворот вектора руху на основі Raycast, а для запобігання злипанню кількох ворогів реалізовано механізм розштовхування (Separation). Якщо гравець входить у радіус атаки (AttackRange), відбувається перехід у Attack. Якщо гравець виходить за межі зони переслідування – ворог повертається у Patrol.

У стані Attack ворог наносить шкоду гравцю з визначеним інтервалом між ударами. Після завершення атаки, якщо гравець все ще в радіусі атаки, цикл повторюється; інакше ворог повертається у Chase.

2.6 Висновки за розділом 2

У другому розділі визначено вимоги до прототипу, обрано інструменти розробки та спроектовано ключові підсистеми гри.

На основі аналізу ігрових рушіїв обрано Unity як найбільш зручний інструмент для 2D-розробки з підтримкою мови C# та компонентної архітектури. Як середовище розробки обрано Microsoft Visual Studio завдяки повній інтеграції з Unity та розвиненим інструментам налагодження.

Для організації коду обрано компонентно-орієнтовану архітектуру, що органічно відповідає моделі GameObject/Component рушія Unity. Систему розділено на три незалежні підсистеми: генерації рівнів, управління гравцем та поведінки ворогів – з мінімальною зв'язаністю між ними.

Систему процедурної генерації рівнів спроектовано на основі сіткового підходу з BFS-розширенням від центральної точки. Такий підхід гарантує зв'язність рівня та дозволяє гнучко призначати функціональні ролі кімнатам залежно від їх відстані від стартової позиції. Коридори між кімнатами будуються динамічно шляхом видалення граничних тайлів стін.

Поведінку ворогів спроектовано на основі скінченного автомата станів із чотирма станами: Idle, Patrol, Chase та Attack. Переходи між станами визначаються відстанню до гравця та результатами перевірки видимості через Raycast. Додатково реалізовано механізм розштовхування ворогів для запобігання їх злипанню під час переслідування.

Сукупність прийнятих проєктних рішень забезпечує модульність архітектури, можливість незалежного налагодження кожної підсистеми та достатній функціональний базис для реалізації прототипу гри.

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

У цьому розділі детально розглядаються практичні аспекти реалізації прототипу комп'ютерної гри. На основі архітектурних рішень, описаних у попередніх розділах, розкривається те, як саме кожна із задуманих механік була втілена у робочий код. Розглядається загальна структура проєкту та організація взаємодії між його складовими частинами.

Значну увагу приділено опису внутрішньої логіки ключових ігрових систем: алгоритмічним підходам, що лежать в їх основі, способам обробки ігрових подій та взаємодії об'єктів між собою. Усі аспекти реалізації розглядаються з точки зору технічної ефективності, зручності для гравця та узгодженості із загальною концепцією жанру roguelike.

Результати цього розділу дозволять повністю зрозуміти, як саме функціонує прототип зсередини, та стануть основою для оцінки його працездатності і відповідності поставленим вимогам.

3.1 Структура проєкту

Проєкт розроблено у середовищі Unity і організовано відповідно до стандартної структури Unity-проєкту. Усі ресурси зосереджено в директорії Assets, що поділяється на тематичні підпапки – кожна відповідає за окрему категорію ресурсів і забезпечує зручну навігацію по проєкту в міру його розширення.

Папка Scenes містить єдину ігрову сцену, в якій відбувається процедурна генерація рівня та весь ігровий процес. Використання однієї сцени спрощує управління станом гри: при кожному новому забігу сцена перезавантажується, що повністю скидає стан усіх об'єктів і запускає генерацію нового рівня з нуля.

Папка `Scripts` містить усі C#-скрипти проєкту, згруповані за підсистемами. Окремі підпапки відповідають за генерацію рівня, управління персонажем гравця, поведінку ворогів та бойову систему. Така організація спрощує орієнтування в кодовій базі та дозволяє вносити зміни в окрему підсистему, не зачіпаючи інші [7]-[8].

Папка `Prefabs` містить заздалегідь налаштовані ігрові об'єкти – префаби ворогів різних типів, снаряду гравця та елементів рівня. Префаби є ключовим інструментом Unity для динамічного створення об'єктів у runtime: під час генерації рівня система інстанціює відповідні префаби кімнат і розміщує їх на сцені згідно з побудованим графом.

Папка `Tilemaps` містить тайлові набори, що використовуються для побудови підлоги та стін кімнат і коридорів. Тайли підлоги коридорів малюються динамічно на глобальній тайлмапі під час генерації, тоді як тайли стін у місцях примикання коридорів видаляються програмно для створення проходів між кімнатами [10].

Папка `UI` містить елементи графічного інтерфейсу – зокрема компоненти HUD (Heads-Up Display) для відображення поточного рівня здоров'я гравця, а також екрани головного меню та екран програшу.

Структуру проєкту в редакторі Unity представлено на рисунку 3.1.

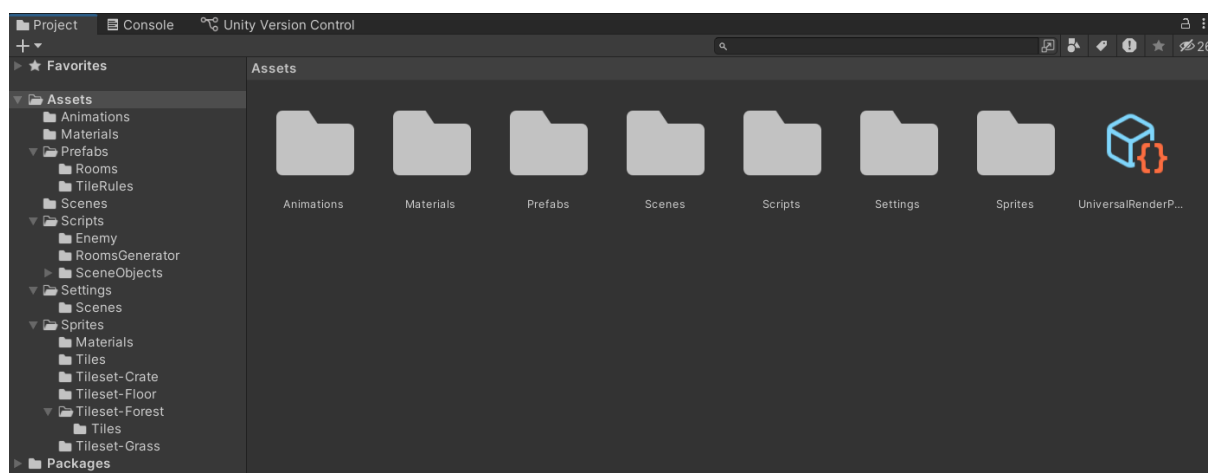


Рисунок 3.1 – Структура проєкту

Ігрова сцена організована за ієрархією об'єктів. На верхньому рівні знаходяться менеджер генерації рівня (RoomsGeneratorManager), об'єкт гравця (Player) та камера, що слідує за гравцем. Вороги і елементи рівня не присутні в ієрархії сцени статично – вони створюються динамічно під час генерації. Ієрархію об'єктів сцени представлено на рисунку 3.2.

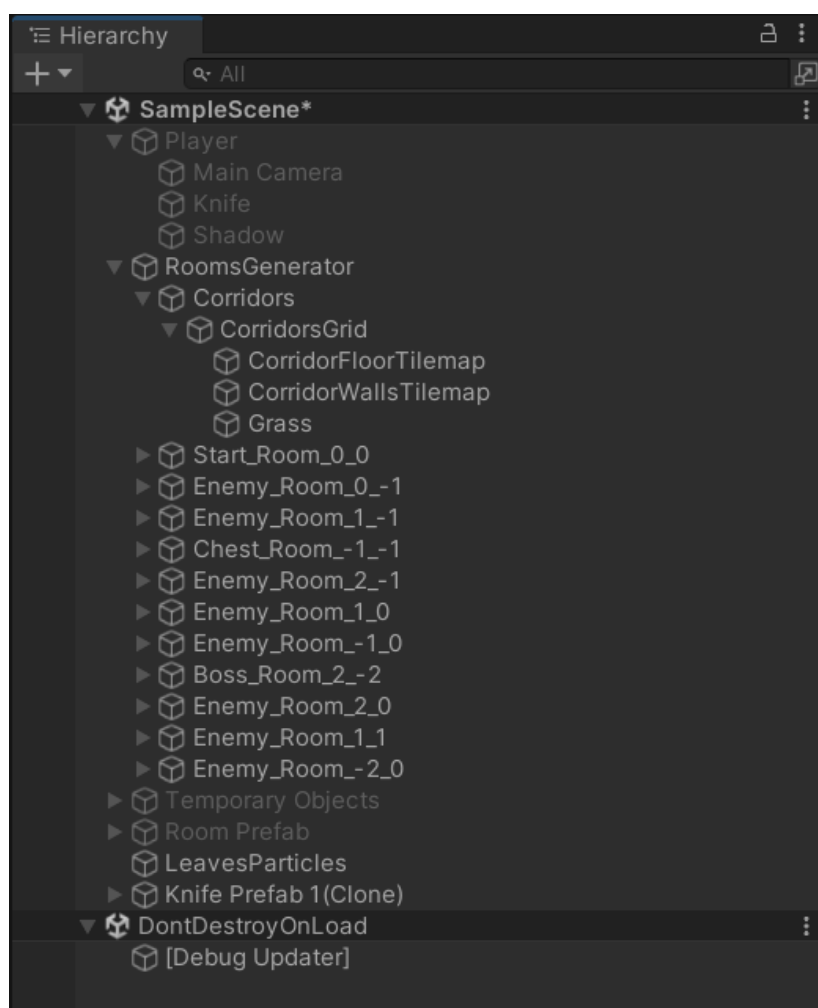


Рисунок 3.2 – Об'єкти на сцені

3.2 Реалізація системи процедурної генерації рівнів

3.2.1 Генерація сітки та BFS-розширення

Реалізацію системи процедурної генерації рівнів зосереджено у скрипті RoomsGeneratorManager.cs. Центральною структурою даних є HashSet зайнятих

комірок, що дозволяє за константний час перевіряти, чи не накладаються нові кімнати на вже існуючі.

Алгоритм починає роботу з центральної комірки з координатами (0, 0), яка одразу додається до черги розширення. На кожній ітерації з черги береться поточна комірка і для кожного з чотирьох можливих сусідів виконується перевірка: сусід не повинен бути вже зайнятим і має знаходитись у межах сітки. Кількість сусідів, що додаються на кожному кроці, визначається випадковим чином на основі ваг імовірностей, заданих у конфігурації (`normalNeighborProbabilities`). Якщо черга спустошується раніше, ніж досягнуто потрібної кількості кімнат, алгоритм знаходить вільну комірку біля вже існуючих і продовжує розширення – це запобігає ситуації, коли рівень виявляється меншим за очікуваний.

Після завершення побудови логічної сітки кожна комірка отримує фізичне розташування у сцені. Координати комірки множаться на параметр `cellSizeInTiles` (за замовчуванням 25 тайлів), що визначає відстань між центрами сусідніх кімнат. Для кожної комірки з відповідного набору обирається випадковий префаб кімнати, який інстанціюється у розрахованій позиції як дочірній об'єкт глобального контейнера `Rooms`.

3.2.2 Призначення ролей та побудова коридорів

Після формування сітки кожній комірці призначається функціональна роль. Стартова кімната – завжди комірка (0, 0). Для визначення позиції босової кімнати алгоритм обчислює відстань від стартової комірки до всіх інших за кількістю кроків у графі та обирає найвіддаленіший «лист» – комірку з єдиним сусідом. Це гарантує, що гравець буде змушений дослідити більшу частину рівня перед фінальним протистоянням. Кімнати з ворогами та скарбами обираються випадковим чином серед решти комірок відповідно до кількості, заданої у конфігурації. Усі інші комірки отримують роль звичайної кімнати.

Побудову коридорів виконує окремий компонент CorridorBuilder. Для кожної пари суміжних кімнат алгоритм визначає спільну межу та малює смугу підлоги і стін на виділеному тайлмепі коридорів. Паралельно метод RemoveOpeningInWalls «пробиває» отвори у тайлмепі стін самих кімнат у місцях з'єднання з коридором. Кожна кімната має компонент RoomController, який аналізує наявність сусідів з кожного боку: якщо сусід відсутній – відповідний прохід перекривається тайлами стіни [10], [13].

Кімната активується у момент входу гравця у її тригер. Якщо в кімнаті передбачені вороги, активується тайлмеп дверей, що блокує всі виходи до знищення останнього ворога. Загальний вигляд згенерованого рівня у грі представлено на рисунку 3.3.

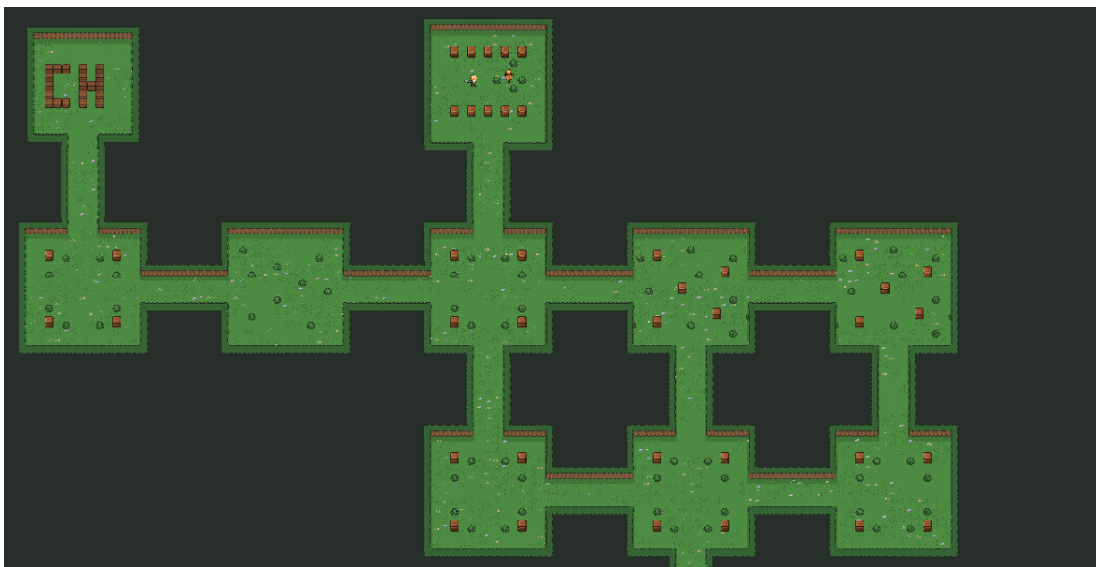


Рисунок 3.3 – Приклад згенерованого рівня

3.3 Реалізація поведінки ворогів

3.3.1 Стан спокою

Стан спокою (Idle) є базовим станом ворога. До нього ворог переходить після досягнення чергової точки патрулювання або після завершення переслідування гравця, якщо той вийшов за межі зони видимості. У цьому стані ворог повністю зупиняється і очікує протягом випадкового проміжку часу,

визначеного параметром `PauseBetweenPoints`. Діапазон цього параметру рандомізується індивідуально для кожного екземпляра ворога при його ініціалізації, що забезпечує природну асинхронність поведінки групи ворогів – вони не рухаються синхронно і не виглядають однаково.

Після закінчення паузи ворог автоматично переходить у стан `Patrol`. Стан `Idle` також слугує точкою входу одразу після появи ворога на сцені – перед початком патрулювання кожен ворог витримує початкову паузу, що додатково розсинхронізує їхню поведінку у групі.

3.3.2 Стан патрулювання

У стані патрулювання (`Patrol`) ворог самостійно переміщується по ігровому рівні у межах визначеної зони. На початку кожного циклу патрулювання скрипт обирає випадкову точку в межах радіусу `PatrolRadius` відносно початкової позиції ворога. Перед тим як почати рух до обраної точки, виконується перевірка доступності через `Raycast`: якщо між поточною позицією ворога та цільовою точкою виявлено перешкоду, точка відхиляється і обирається нова. Такий підхід забезпечує природність переміщення та виключає ситуації, коли ворог намагається рухатись крізь стіну або застряє біля неї, що могло б порушити ігровий процес і знизити якість взаємодії з середовищем.

Окрім суто геометричних перешкод у вигляді стін та меж рівня, у ролі перешкод можуть виступати й інші об'єкти ігрового середовища. Зокрема, це ящики, що руйнуються після кількох влучань і тим самим динамічно змінюють прохідність простору в процесі гри, а також декоративно-функціональні елементи. Сукупність таких об'єктів формує насичене та варіативне ігрове середовище, яке вимагає від системи патрулювання постійної адаптації до змінюваної геометрії рівня [11].

Приклад перешкод з власною колізією зображений на рисунку 3.4.



Рисунок 3.4 – Приклад ігрових перешкод

Додатково у стані Patrol реалізовано механізм виявлення застрягання. Якщо протягом певного часу (`stuckTimer`) ворог не змінив своєї позиції або змінив її несуттєво, система автоматично скидає поточну цільову точку і обирає нову. Це забезпечує стабільну поведінку навіть у випадках, коли геометрія рівня не дозволяє побудувати пряму траєкторію до обраної точки.

Як і параметр паузи, швидкість переміщення та радіус патрулювання рандомізуються індивідуально при ініціалізації кожного екземпляра ворога. Завдяки цьому вороги одного типу поведуться по-різному, що робить ігрове середовище більш живим та непередбачуваним для гравця.

Перехід зі стану Patrol або Idle до стану Chase відбувається у момент, коли відстань до гравця стає меншою за значення `VisionRange`, і між ворогом та гравцем немає перешкод. Ця перевірка виконується кожен кадр незалежно від поточного стану ворога.

3.3.3 Стан переслідування

Стан переслідування (Chase) активується у момент, коли гравець потрапляє у зону видимості ворога – відстань до нього стає меншою за VisionRange і між ними немає перешкод. Це найскладніший стан з точки зору реалізації, оскільки ворог має не лише рухатись у напрямку гравця, а й коректно обходити перешкоди та уникати злипання з іншими ворогами. Саме тому стан Chase вимагає поєднання кількох незалежних механізмів, що працюють узгоджено в межах одного ігрового кадру.

Для обходу перешкод застосовується механізм на основі Raycast: кожен кадр ворог випускає промінь у напрямку гравця. Якщо промінь виявляє стіну, вектор руху повертається на фіксований кут AvoidanceTurnAngle до отримання вільного напрямку. Цей процес повторюється ітеративно до знаходження прохідного напрямку, завдяки чому ворог плавно «ковзає» вздовж стіни доти, доки прямий шлях до гравця не відкриється. Перевагою такого підходу є його обчислювальна легкість порівняно з повноцінною навігаційною сіткою, що є критично важливим при одночасній активності кількох ворогів.

Для запобігання злипання кількох ворогів реалізовано механізм розштовхування (Separation). Кожен ворог у малому радіусі сканує простір навколо себе за допомогою маски EnemyMask та обчислює вектор відштовхування від кожного виявленого сусіда – чим ближче сусід, тим сильніший його вплив. Підсумковий вектор руху є зваженою сумою напрямку до гравця та вектора відштовхування від сусідів. Вагові коефіцієнти цих складових підібрані емпірично з урахуванням швидкості ворогів та середньої густоти їх скупчення на рівні. Практичним наслідком цього є те, що група ворогів намагається органічно оточити гравця з різних боків, а не збивається в одну точку, що суттєво підвищує складність і реалістичність бойових ситуацій.

Приклад переслідування групою ворогів наведено на рисунку 3.5.



Рисунок 3.5 – Приклад поведінки переслідування

Якщо гравець виходить за межі зони видимості або між ними з'являється перешкода, ворог повертається у стан Patrol, відновлюючи звичайний цикл патрулювання.

3.3.4 Стан атаки

Стан атаки (Attack) активується у момент, коли відстань між ворогом і гравцем стає меншою за значення AttackRange. Ворог зупиняється і починає завдавати шкоди гравцю з фіксованим інтервалом між ударами, що визначається параметром кулдауну атаки. Витримка між ударами запобігає миттєвому виснаженню здоров'я гравця при впритул і дає йому можливість відреагувати та відійти на безпечну відстань.

Якщо у процесі атаки гравець виходить за межі AttackRange – наприклад, відкочується або відбігає – ворог автоматично повертається у стан

Chase і відновлює переслідування. Це забезпечує плавні та логічні переходи між станами без розривів у поведінці.

При отриманні шкоди від гравця відтворюється візуальний індикатор влучення (TakeDamageIndicator), що забезпечує зворотний зв'язок і робить бойові взаємодії більш читабельними. Коли показник здоров'я ворога досягає нуля, вимикаються всі колайдери, зупиняється логіка скінченного автомата станів і через коротку затримку ігровий об'єкт видаляється зі сцени. Затримка перед видаленням дозволяє програти анімацію загибелі та не переривати її миттєвим зникненням об'єкта.

3.4 Реалізація бойової системи

Бойова система прототипу складається з двох взаємопов'язаних частин: атаки гравця через снаряди та реакції ворогів на отриману шкоду.

Атака гравця відбувається за натисканням відповідної клавіші. У момент натискання у позиції гравця інстанціюється префаб снаряду (Projectile), якому надається початковий імпульс у напрямку курсора. Снаряд має два умови знищення: зіткнення з ворогом або вичерпання часу існування (lifetime). Друге умова необхідне для того, щоб снаряди, що не влучили в ціль, не накопичувались у сцені. Між пострілами витримується кулдаун, що обмежує частоту атак.

При зіткненні снаряду з ворогом спрацьовує обробник колізій: ворог отримує задану кількість шкоди, знімаючи відповідне значення з його поточного запасу НР, після чого снаряд знищується. Оскільки вороги мають НР розраховане на кілька влучень, гравцю необхідно атакувати послідовно, що формує базову тактичну складову бою. При кожному влученні програється візуальний індикатор отриманої шкоди.

Здоров'я гравця зменшується при кожному успішному ударі ворога у стані Attack. Поточний рівень НР відображається у HUD у вигляді шкали здоров'я, що оновлюється в реальному часі.

Коли HP гравця досягає нуля, спрацьовує логіка програшу: ігровий час зупиняється (`Time.timeScale = 0`), що заморожує всі об'єкти сцени. У поточній версії прототипу перезапуск гри виконується вручну, оскільки екран програшу не реалізовано як окремий елемент інтерфейсу.

3.5 Графічний інтерфейс користувача

У поточній версії прототипу графічний інтерфейс зведено до мінімуму, оскільки основна мета розробки полягала у перевірці ігрових механік, а не в поліровці візуальної частини.

Єдиним елементом HUD є шкала здоров'я гравця, реалізована засобами Unity UI. Вона розташована у верхньому куті екрану та оновлюється в реальному часі при отриманні гравцем шкоди.

Навігаційне меню, екран програшу та інші елементи інтерфейсу у прототипі відсутні – їх реалізацію заплановано на етапі подальшого розвитку проекту.

3.6 Висновки за розділом 3

У третьому розділі описано практичну реалізацію усіх ключових підсистем прототипу комп'ютерної гри жанру roguelike.

Проект організовано відповідно до стандартної структури Unity-проекту з розподілом ресурсів за тематичними підпапками, що забезпечує зручність навігації та підтримки коду.

Систему процедурної генерації рівнів реалізовано у скрипті `RoomsGeneratorManager.cs` на основі сіткового підходу з BFS-розширенням. Результатом кожного запуску є унікальна зв'язна карта кімнат із автоматично призначеними ролями та динамічно побудованими коридорами.

Поведінку ворогів реалізовано у скрипті `MeleeEnemyController.cs` на основі скінченного автомата станів із чотирма станами. Механізми обходу

перешкод через Raycast та розштовхування ворогів через Separation забезпечують автономну та природну поведінку без використання навігаційних сіток.

Бойову систему побудовано на основі снарядів із обмеженим часом існування та обробці колізій. Здоров'я як гравця, так і ворогів розраховано на кілька влучень, що формує базову тактичну складову ігрового процесу.

Графічний інтерфейс у поточній версії обмежено шкалою здоров'я гравця, що відповідає цілям прототипу – верифікації ігрових механік без надмірних витрат на інтерфейсні рішення.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

У цьому розділі розглянуто практичні аспекти використання розробленого прототипу комп'ютерної гри жанру roguelike: його призначення, технічні умови запуску, інструкцію з керування та результати роботи програми в ігровому середовищі. Також проведено тестування ключових підсистем – процедурної генерації рівнів, поведінки ворогів та бойової системи – з метою перевірки їх відповідності сформульованим у першому розділі вимогам. Мета цього розділу – продемонструвати працездатність реалізованих механік та підтвердити ефективність прийнятих проєктних рішень на основі спостережень за реальним ігровим процесом.

4.1 Призначення та умови виконання програми

Основним призначенням розробленого прототипу є демонстрація ключових механік комп'ютерної гри жанру roguelike: процедурної генерації ігрових рівнів, автономної поведінки ворогів на основі скінченного автомата станів та базової бойової системи. Прототип не є завершеним комерційним продуктом – його мета полягає у перевірці працездатності реалізованих алгоритмів та обґрунтуванні технічних рішень, прийнятих у процесі проєктування [7], [12].

Розроблений прототип орієнтований на запуск на персональному комп'ютері під управлінням операційної системи Windows. Для коректного виконання програми комп'ютер має відповідати мінімальним технічним вимогам: процесор з тактовою частотою не нижче 2.0 ГГц, оперативна пам'ять обсягом не менше 4 ГБ та відеокарта з підтримкою DirectX 11. Зазначені вимоги є типовими для додатків, зібраних у середовищі Unity, і без труднощів задовольняються будь-яким сучасним персональним комп'ютером.

Програма поставляється у вигляді скомпільованого виконуваного файлу .exe та не потребує встановлення – достатньо запустити файл безпосередньо. Додаткове програмне забезпечення або сторонні залежності для роботи прототипу не потрібні.

4.2 Інструкція з експлуатації

Оскільки програма знаходиться на стадії прототипу, її запуск здійснюється безпосередньо у середовищі розробки Unity. Для цього необхідно відкрити проєкт у Unity Editor та натиснути кнопку Play у верхній панелі редактора. Після цього автоматично генерується новий ігровий рівень і персонаж гравця розміщується у стартовій кімнаті. Інтерфейс запуску у редакторі Unity представлено на рисунку.

Керування персонажем здійснюється за допомогою клавіатури та миші:

- W / A / S / D – переміщення персонажа;
- ЛКМ (ліва кнопка миші) – постріл снарядом у напрямку курсора.

Мета гравця – дослідити згенерований рівень, знищуючи ворогів у кімнатах, та дістатися до босової кімнати. При вході у кімнату з ворогами виходи блокуються до знищення всіх противників. Поточний рівень здоров'я відображається у шкалі HUD у верхній частині екрану.

У разі загибелі персонажа ігровий процес зупиняється. Для початку нової гри достатньо повторно натиснути Play в редакторі – рівень щоразу генерується заново.

4.3 Результати роботи програми

Після запуску прототипу у редакторі Unity система процедурної генерації автоматично будує унікальний ігровий рівень. Гравець починає у стартовій кімнаті, від якої відходять коридори до суміжних кімнат різних типів. Завдяки BFS-розширенню кожен запуск дає відмінну від попередньої

конфігурацію рівня – кількість кімнат, їх розташування та з'єднання щоразу генеруються заново.

При вході гравця у кімнату з ворогами виходи з неї блокуються і розпочинається бій. Вороги переходять зі стану патрулювання у переслідування, намагаючись оточити гравця з різних боків завдяки механізму розштовхування. Гравець відповідає пострілами снарядів, які летять у напрямку курсора та знищуються при влученні у ворога або після вичерпання часу існування.

Ігровий процес під час бою представлено на рисунку 4.1.



Рисунок 4.1 – Приклад ігрового процесу

Шкала здоров'я у верхній частині екрану відображає поточний стан гравця в реальному часі та зменшується при кожному влученні ворога. Після знищення всіх ворогів у кімнаті виходи розблоковуються і гравець може продовжити дослідження рівня.

Фінальною точкою рівня є босова кімната – найвіддаленіша від стартової позиції. Досягнення босової кімнати та її очищення є умовою завершення поточного забігу, що буде реалізовано згодом.

4.4 Тестування програми

Тестування прототипу проводилось у ручному режимі безпосередньо у середовищі Unity Editor шляхом багаторазового запуску програми та спостереження за поведінкою кожної підсистеми. Формальних автоматизованих тестів не застосовувалось, що є прийнятним для проєктів на стадії прототипування [15].

Тестування системи процедурної генерації рівнів полягало у багаторазовому перезапуску сцени з перевіркою унікальності та зв'язності кожного згенерованого рівня. У всіх перевірених випадках рівень виявлявся зв'язним – жодна кімната не була ізольована від решти. Стартова кімната завжди розміщувалась у центральній позиції, босова – у найвіддаленішій від старту точці. Коридори коректно з'єднували всі суміжні кімнати без накладань та прогалин у тайлмепі.

Тестування поведінки ворогів проводилось шляхом перевірки коректності переходів між станами FSM. Було підтверджено, що вороги переходять у стан Chase при наближенні гравця на відстань VisionRange та повертаються до патрулювання після його виходу із зони переслідування. Механізм розштовхування функціонував коректно – група ворогів не збивалась в одну точку, а намагалась оточити гравця з різних боків. У поодиноких випадках спостерігалось застрягання ворога у кутах кімнати при обході перешкод, що є відомим обмеженням реалізованого алгоритму уникнення на основі Raycast [11].

Приклад здійснювання тестування завдяки Unity Editor зображений на рисунку 4.2. На рисунку видно інспектор в стані відладки, що дає змогу бачити поточний стан ворога, його Reycast, а також всі змінні ворога, які впливають на швидкість, дальність зору, тощо.

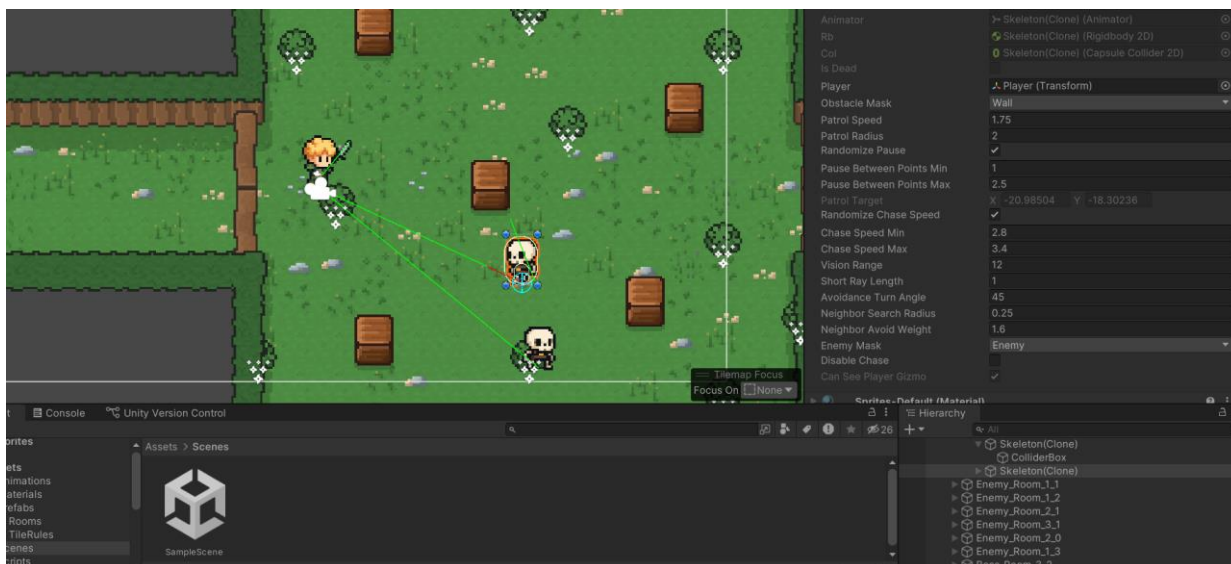


Рисунок 4.2 – Приклад тестування ворога в інспекторі

Тестування бойової системи підтвердило коректну роботу снарядів: при влученні у ворога снаряд знищувався і ворог отримував шкоду. Снаряди, що не влучили в ціль, знищувались після вичерпання lifetime. Шкала здоров'я гравця оновлювалась у реальному часі при кожному влученні ворога. Логіка програшу спрацьовувала коректно – при досягненні НР нуля ігровий процес зупинявся.

Загалом усі ключові механіки прототипу функціонують відповідно до вимог, сформульованих у першому розділі. Виявлені недоліки – зокрема застрягання ворогів у кутах – є прийнятними для стадії прототипу і можуть бути усунені у подальших ітераціях розробки.

4.5 Висновки за розділом 4

У даному розділі розглянуто практичні аспекти експлуатації та тестування розробленого прототипу комп'ютерної гри жанру roguelike.

Визначено призначення програми та технічні умови її виконання. Прототип запускається безпосередньо у середовищі Unity Editor і не потребує додаткового встановлення або налаштування. Керування реалізовано за допомогою стандартних засобів клавіатури та миші.

Продemonстровано результати роботи програми: система процедурної генерації щоразу формує унікальний зв'язний рівень, вороги коректно реагують на присутність гравця, а бойова система забезпечує базову тактичну взаємодію між гравцем і противниками.

За результатами ручного тестування підтверджено відповідність усіх ключових підсистем сформульованим вимогам. Виявлено одне обмеження поточної реалізації – можливе застрягання ворогів у кутах кімнат при обході перешкод, усунення якого заплановано у подальших ітераціях розробки.

ВИСНОВКИ

В ході виконання дипломної кваліфікаційної роботи бакалавра було проаналізовано та досліджено процес розробки прототипу комп'ютерної гри жанру roguelike з процедурною генерацією рівнів та системою поведінки ворогів на основі штучного інтелекту.

Визначено, що жанр roguelike є одним із найбільш технічно насичених жанрів комп'ютерних ігор, ключовою особливістю якого є процедурна генерація ігрового простору, що забезпечує унікальність кожного проходження та високу реграбельність. Проведено аналіз існуючих підходів до процедурної генерації рівнів та поведінки ворогів, розглянуто відповідні рішення у сучасних іграх жанру.

Сформульовано функціональні та нефункціональні вимоги до прототипу. На основі порівняльного аналізу обрано ігровий рушій Unity, мову програмування C# та середовище розробки Microsoft Visual Studio як оптимальний набір інструментів для реалізації поставлених задач.

Спроектовано архітектуру програмного забезпечення на основі компонентно-орієнтованого підходу з розподілом системи на три незалежні підсистеми: генерації рівнів, управління гравцем та поведінки ворогів.

Розроблено систему процедурної генерації рівнів на основі сіткового підходу з BFS-розширенням від центральної точки. Реалізовано автоматичне призначення функціональних ролей кімнатам та динамічну побудову коридорів між ними. Кожен запуск програми формує унікальну зв'язну карту рівня.

Реалізовано систему поведінки ворогів на основі скінченного автомата станів із чотирма станами: Idle, Patrol, Chase та Attack. Для забезпечення природної поведінки у стані переслідування застосовано механізм обходу перешкод на основі Raycast та механізм розштовхування ворогів Separation.

Розроблено бойову систему на основі снарядів з обробкою колізій та системою здоров'я як для гравця, так і для ворогів. Реалізовано відображення стану гравця через HUD зі шкалою здоров'я.

За результатами ручного тестування підтверджено відповідність усіх реалізованих підсистем сформульованим вимогам. Прототип демонструє працездатність обраних алгоритмів та підтверджує ефективність прийнятих проєктних рішень як основу для подальшого розвитку повноцінного ігрового продукту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Hades Official Page. Supergiant Games. URL: <https://www.supergiantgames.com/games/hades> (date of access: 20.04.2026).
2. Dead Cells Official Page. Motion Twin. URL: <https://www.motion-twin.com/dead-cells> (date of access: 22.04.2026).
3. Enter the Gungeon Official Page. Devolver Digital. URL: <https://www.devolverdigital.com/games/enter-the-gungeon> (date of access: 24.04.2026).
4. Shaker N., Togelius J., Nelson M. J. Procedural Content Generation in Games. Springer, 2016. 244 p. URL: <https://www.pcgbook.com> (date of access: 25.04.2026).
5. Millington I. AI for Games. 3rd ed. CRC Press, 2019. 896 p.
6. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. CRC Press, 2019. 600 p.
7. Unity Engine Official Documentation. Unity Technologies. URL: <https://docs.unity3d.com/Manual/index.html> (date of access: 27.04.2026).
8. Microsoft C# Language Documentation. Microsoft. URL: <https://learn.microsoft.com/en-us/dotnet/csharp> (date of access: 29.04.2026).
9. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 345 p. URL: <https://gameprogrammingpatterns.com> (date of access: 01.05.2026).
10. Unity Tilemap Documentation. Unity Technologies. URL: <https://docs.unity3d.com/Manual/class-Tilemap.html> (date of access: 03.05.2026).
11. Unity Physics 2D Documentation. Unity Technologies. URL: <https://docs.unity3d.com/Manual/Physics2DReference.html> (date of access: 05.05.2026).
12. Visual Studio Official Home Page. Microsoft. URL: <https://visualstudio.microsoft.com> (date of access: 07.05.2026).
13. Breadth-First Search. Wikipedia. URL: https://en.wikipedia.org/wiki/Breadth-first_search (date of access: 09.05.2026).

14. Finite-State Machine. Wikipedia. URL: https://en.wikipedia.org/wiki/Finite-state_machine (date of access: 12.05.2026).
15. Building for Playtesting. Unity Technologies. URL: <https://discussions.unity.com/t/building-for-playtesting/621453> (date of access: 15.05.2026).

ДОДАТОК А
Текст програми

A.1 Код гравця

PlayerController.cs

```
using UnityEngine;
```

```
[RequireComponent(typeof(Rigidbody2D))]
public class PlayerController : MonoBehaviour
{
```

```
    [Header("Movement")]
    public float moveSpeed = 5f;
```

```
    [Header("Health")]
    public int maxHealth = 8;
    private int currentHealth;
    private bool isDead = false;
```

```
    [Header("References")]
    public HeartUIManager heartUIManager;
```

```
    private Rigidbody2D rb;
    private Animator animator;
    private Vector2 input;
    private Transform graphics;
```

```
    public bool IsFacingRight { get; private set; } = true;
```

```
    void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        graphics = transform;
        currentHealth = maxHealth;

        heartUIManager?.SetHealth(currentHealth, maxHealth);
    }
```

```
    void Update()
    {
        if (isDead) return;

        HandleMovementInput();
        HandleSpriteFlip();
    }
```

```
    void FixedUpdate()
    {
        if (isDead) return;

        rb.velocity = input * moveSpeed;
    }
```

```

void HandleMovementInput()
{
    float moveX = Input.GetAxisRaw("Horizontal");
    float moveY = Input.GetAxisRaw("Vertical");
    input = new Vector2(moveX, moveY).normalized;

    animator.SetFloat("Speed", input.magnitude);
}

void HandleSpriteFlip()
{
    if (input.x != 0)
    {
        IsFacingRight = input.x > 0;

        Vector3 scale = graphics.localScale;
        scale.x = IsFacingRight ? 1f : -1f;
        graphics.localScale = scale;
    }
}

public void TakeDamage(int amount)
{
    if (isDead) return;

    currentHealth -= amount;
    currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);

    Debug.Log($"Player took damage: {amount}. Remaining HP: {currentHealth}");

    if (animator != null)
    {
        animator.SetTrigger("TakeDamage");
    }

    heartUIManager?.SetHealth(currentHealth, maxHealth);

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

void Die()
{
    isDead = true;
    rb.velocity = Vector2.zero;

    Debug.Log("Player Died");

    if (animator != null)
    {
        animator.SetTrigger("Die");
    }
}

```

```

    }
}

public void Heal(int amount)
{
    if (isDead) return;

    currentHealth += amount;
    currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);

    Debug.Log($"Player healed: {amount}. Current HP: {currentHealth}");

    heartUIManager?.SetHealth(currentHealth, maxHealth);
}
}

```

AttackController.cs

```

using UnityEngine;

public class AttackController : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private Transform weaponHolder;
    [SerializeField] private GameObject defaultWeaponPrefab;
    [SerializeField] private LayerMask obstacleLayer;

    [Header("Settings")]
    [SerializeField] private float weaponDistance = 1f;
    [SerializeField] private float followSpeed = 10f;
    [SerializeField] private float wallCheckDistance = 1f;

    private float defaultWeaponDistance;
    private GameObject currentWeaponGO;
    private WeaponBase currentWeapon;
    private Camera mainCamera;

    void Awake()
    {
        mainCamera = Camera.main;
        defaultWeaponDistance = weaponDistance;
    }

    void Start()
    {
        EquipWeapon(defaultWeaponPrefab);
    }

    void Update()
    {
        if (currentWeaponGO != null)
        {

```

```

FollowCursorWithDistance();
RotateWeaponToCursor();

if (Input.GetMouseButtonDown(0))
{
    currentWeapon?.Attack();
}
}

void FollowCursorWithDistance()
{
    Vector3 mouseWorldPos = mainCamera.ScreenToWorldPoint(Input.mousePosition);
    Vector3 holderPos = weaponHolder.position;

    Vector3 direction = mouseWorldPos - holderPos;
    direction.z = 0f;
    direction.Normalize();

    CheckWallProximity(holderPos, direction);

    Vector3 targetPosition = holderPos + direction * weaponDistance;
    currentWeaponGO.transform.position = Vector3.Lerp(currentWeaponGO.transform.position,
targetPosition, followSpeed * Time.deltaTime);
}

void RotateWeaponToCursor()
{
    Vector3 mousePos = mainCamera.ScreenToWorldPoint(Input.mousePosition);
    Vector2 direction = mousePos - currentWeaponGO.transform.position;
    float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;

    currentWeaponGO.transform.rotation = Quaternion.Euler(0f, 0f, angle);

    Vector3 localScale = currentWeaponGO.transform.localScale;
    localScale.y = Mathf.Abs(localScale.y) * (angle > 90f || angle < -90f ? -1f : 1f);
    currentWeaponGO.transform.localScale = localScale;
}

void CheckWallProximity(Vector2 origin, Vector2 direction)
{
    RaycastHit2D hit = Physics2D.Raycast(origin, direction, wallCheckDistance, obstacleLayer);

    if (hit.collider != null)
    {
        if (hit.collider.CompareTag("Wall") || hit.collider.CompareTag("Enemy"))
        {
            weaponDistance = 0f;
        }
        else
        {
            weaponDistance = defaultWeaponDistance;
        }
    }
}

```

```

    }
    else
    {
        weaponDistance = defaultWeaponDistance;
    }

    Debug.DrawRay(origin, direction * wallCheckDistance,
        hit.collider != null ? Color.red : Color.green);
}

public void EquipWeapon(GameObject weaponPrefab)
{
    if (currentWeaponGO != null)
    {
        Destroy(currentWeaponGO);
    }

    currentWeaponGO = Instantiate(weaponPrefab, weaponHolder.position, Quaternion.identity);
    currentWeapon = currentWeaponGO.GetComponent<WeaponBase>();

    if (currentWeapon == null)
    {
        Debug.LogWarning("Weapon prefab has no WeaponBase component.");
    }
}
}

```

PlayerSpawner.cs

using UnityEngine;

```

public class PlayerSpawner : MonoBehaviour
{
    [Tooltip("Player prefab to spawn at this location")]
    public GameObject playerPrefab;

    private void Start()
    {
        if (playerPrefab == null)
        {
            Debug.LogError("PlayerSpawner: Player prefab not assigned!");
            return;
        }

        Instantiate(playerPrefab, transform.position, Quaternion.identity);

        SpriteRenderer sr = GetComponent<SpriteRenderer>();
        if (sr != null)
        {
            Destroy(sr);
        }
    }
}

```

```
}
```

HeartUIManager.cs

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

public class HeartUIManager : MonoBehaviour
{
    public GameObject heartPrefab;
    public Sprite fullHeart;
    public Sprite halfHeart;
    public Sprite emptyHeart;

    private List<Image> hearts = new List<Image>();

    public void SetHealth(int currentHealth, int maxHealth)
    {
        int totalHearts = Mathf.CeilToInt(maxHealth / 2f);
        int heartIndex = 0;

        while (hearts.Count < totalHearts)
        {
            GameObject newHeart = Instantiate(heartPrefab, transform);
            Image heartImage = newHeart.GetComponent<Image>();
            hearts.Add(heartImage);
        }

        for (int i = 0; i < hearts.Count; i++)
        {
            Image img = hearts[i];

            if (heartIndex * 2 + 1 < currentHealth)
            {
                img.sprite = fullHeart;
            }
            else if (heartIndex * 2 < currentHealth)
            {
                img.sprite = halfHeart;
            }
            else
            {
                img.sprite = emptyHeart;
            }

            img.enabled = i < totalHearts;
            heartIndex++;
        }
    }
}
```

A.2 Код генератора рівнів

```

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

public class RoomsGeneratorManager : MonoBehaviour
{
    [Header("Room Categories")]
    public List<RoomCategory> roomCategories = new List<RoomCategory>();

    [Header("Generation Settings")]
    public Vector2Int cellSizeInTiles = new Vector2Int(30, 30);
    public Transform roomsParent;

    [Header("Corridor Settings")]
    public GameObject corridorsRoot;
    public TileBase corridorTile;
    public TileBase corridorWallTile;
    public int corridorWidth = 3;

    [Header("Growth Settings")]
    public Vector3 normalNeighborProbabilities = new Vector3(0.75f, 0.20f, 0.05f);
    public float extraBranchChance = 0.08f;

    private Tilemap corridorFloorTilemap;
    private Dictionary<Vector2Int, GameObject> placedRooms = new Dictionary<Vector2Int, GameObject>();
    private HashSet<Vector2Int> occupiedCells = new HashSet<Vector2Int>();
    private System.Random sysRand = new System.Random();

    private void Start()
    {
        if (corridorWidth < 1) corridorWidth = 1;
        if (corridorWidth % 2 == 0) corridorWidth += 1;

        if (corridorsRoot != null)
        {
            corridorFloorTilemap = corridorsRoot.GetComponentInChildren<Tilemap>().FirstOrDefault(t =>
t.name.ToLower().Contains("corridor") || t.name.ToLower().Contains("floor"));
        }

        GenerateRooms();
    }

    public void GenerateRooms()
    {
        if (roomsParent == null)
        {
            roomsParent = new GameObject("Rooms").transform;
        }
    }
}

```

```

roomsParent.parent = this.transform;
}

placedRooms.Clear();
occupiedCells.Clear();

int totalRooms = Mathf.Max(1, roomCategories.Sum(c => Mathf.Max(0, c.count)));

Queue<Vector2Int> frontier = new Queue<Vector2Int>();
Vector2Int startCell = Vector2Int.zero;
frontier.Enqueue(startCell);
occupiedCells.Add(startCell);

List<Vector2Int> finalCells = new List<Vector2Int> { startCell };

while (finalCells.Count < totalRooms && frontier.Count > 0)
{
    Vector2Int current = frontier.Dequeue();
    int spawnCount = DetermineSpawnCountForGeneration();
    List<Vector2Int> freeNeighbors = GetFreeNeighbors(current);

    Shuffle(freeNeighbors);

    int toSpawn = Mathf.Min(spawnCount, freeNeighbors.Count);
    for (int i = 0; i < toSpawn && finalCells.Count < totalRooms; i++)
    {
        Vector2Int chosen = freeNeighbors[i];
        if (!occupiedCells.Contains(chosen))
        {
            occupiedCells.Add(chosen);
            frontier.Enqueue(chosen);
            finalCells.Add(chosen);
        }
    }
}

// ... Adjacency and Role Assignment ...

Dictionary<Vector2Int, List<Vector2Int>> adjacency = new Dictionary<Vector2Int, List<Vector2Int>>();
foreach (var cell in finalCells) adjacency[cell] = new List<Vector2Int>();
foreach (var cell in finalCells)
{
    foreach (var d in new Vector2Int[] { Vector2Int.up, Vector2Int.down, Vector2Int.left, Vector2Int.right
})
    {
        Vector2Int n = cell + d;
        if (adjacency.ContainsKey(n)) adjacency[cell].Add(n);
    }
}

// ... Placement and Corridor Building ...
}

```

```

private int DetermineSpawnCountForGeneration()
{
    double r = sysRand.NextDouble();
    if (r < normalNeighborProbabilities.x) return 1;
    if (r < normalNeighborProbabilities.x + normalNeighborProbabilities.y) return 2;
    return 3;
}

private List<Vector2Int> GetFreeNeighbors(Vector2Int cell)
{
    List<Vector2Int> neighbors = new List<Vector2Int>();
    foreach (var d in new Vector2Int[] { Vector2Int.up, Vector2Int.down, Vector2Int.left, Vector2Int.right })
    {
        if (!occupiedCells.Contains(cell + d)) neighbors.Add(cell + d);
    }
    return neighbors;
}

private void Shuffle<T>(IList<T> list)
{
    for (int i = 0; i < list.Count; i++)
    {
        int j = sysRand.Next(i, list.Count);
        (list[i], list[j]) = (list[j], list[i]);
    }
}
}

```

RoomController.cs

```

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

[RequireComponent(typeof(Collider2D))]
public class RoomController : MonoBehaviour
{
    [Header("Tilemaps")]
    public Tilemap doorsTilemap;

    [Header("Room Settings")]
    public bool hasEnemies = true;
    public float openDelayAfterAllWaves = 0f;

    private EnemiesSpawner[] spawners;
    private HashSet<EnemiesSpawner> completedSpawners = new HashSet<EnemiesSpawner>();
    private bool isActive = false;

    private void Start()
    {
        if (doorsTilemap != null) doorsTilemap.gameObject.SetActive(false);
    }
}

```

```

spawnners = GetComponentInChildren<EnemiesSpawner>(true);

if (spawnners != null && spawnners.Length > 0)
{
    foreach (var s in spawnners)
    {
        if (s == null) continue;
        s.StopSpawning();
        s.OnAllWavesCompleted.AddListener(() => OnSpawnerAllWavesCompleted(s));
    }
}

Transform trigger = transform.Find("EnterTrigger");
if (trigger != null)
{
    RoomEnterTrigger triggerScript = trigger.GetComponent<RoomEnterTrigger>() ??
trigger.gameObject.AddComponent<RoomEnterTrigger>();
    triggerScript.room = this;
}
}

public void SetupDoors(List<Vector2Int> neighborDirections)
{
    if (doorsTilemap == null) return;

    bool hasTop = neighborDirections.Contains(Vector2Int.up);
    bool hasBottom = neighborDirections.Contains(Vector2Int.down);
    bool hasLeft = neighborDirections.Contains(Vector2Int.left);
    bool hasRight = neighborDirections.Contains(Vector2Int.right);

    BoundsInt cb = doorsTilemap.cellBounds;
    if (cb.size.x <= 0 || cb.size.y <= 0) return;

    int realMinX = int.MaxValue, realMaxX = int.MinValue;
    int realMinY = int.MaxValue, realMaxY = int.MinValue;
    for (int x = cb.xMin; x < cb.xMax; x++)
    {
        for (int y = cb.yMin; y < cb.yMax; y++)
        {
            if (doorsTilemap.GetTile(new Vector3Int(x, y, 0)) != null)
            {
                realMinX = Mathf.Min(realMinX, x);
                realMaxX = Mathf.Max(realMaxX, x);
                realMinY = Mathf.Min(realMinY, y);
                realMaxY = Mathf.Max(realMaxY, y);
            }
        }
    }

    int depth = 3;
    if (!hasTop) for (int d = 0; d < depth; d++) RemoveDoorRow(doorsTilemap, realMinX, realMaxX,
realMaxY - d);

```

```

        if (!hasBottom) for (int d = 0; d < depth; d++) RemoveDoorRow(doorsTilemap, realMinX, realMaxX,
realMinY + d);
        if (!hasLeft) for (int d = 0; d < depth; d++) RemoveDoorColumn(doorsTilemap, realMinY, realMaxY,
realMinX + d);
        if (!hasRight) for (int d = 0; d < depth; d++) RemoveDoorColumn(doorsTilemap, realMinY, realMaxY,
realMaxX - d);
    }

    private void RemoveDoorRow(Tilemap tilemap, int startX, int endX, int y)
    {
        for (int x = startX; x <= endX; x++) tilemap.SetTile(new Vector3Int(x, y, 0), null);
    }

    private void RemoveDoorColumn(Tilemap tilemap, int startY, int endY, int x)
    {
        for (int y = startY; y <= endY; y++) tilemap.SetTile(new Vector3Int(x, y, 0), null);
    }

    public void ActivateRoom()
    {
        if (isActive) return;
        isActive = true;

        if (hasEnemies && spawners != null && spawners.Length > 0)
        {
            if (doorsTilemap != null) doorsTilemap.gameObject.SetActive(true);
            completedSpawners.Clear();
            foreach (var s in spawners) s.StartSpawning();
        }
    }

    private void OnSpawnerAllWavesCompleted(EnemiesSpawner spawner)
    {
        completedSpawners.Add(spawner);
        if (completedSpawners.Count >= spawners.Length) StartCoroutine(OpenDoorsAfterDelay());
    }

    private IEnumerator OpenDoorsAfterDelay()
    {
        if (openDelayAfterAllWaves > 0f) yield return new WaitForSeconds(openDelayAfterAllWaves);
        if (doorsTilemap != null) doorsTilemap.gameObject.SetActive(false);
    }
}

```

RoomsGeneratorFacade.cs

```

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

namespace DungeonGen

```

```

{
    public class RoomsGeneratorFacade : MonoBehaviour
    {
        public RoomsConfigSO config;
        public Transform roomsParent;
        public GameObject corridorsRoot;
        public TileBase corridorTile;
        public TileBase corridorWallTile;

        public int seed = 0;
        public bool useRandomSeed = true;

        private System.Random sysRand;

        private void Start()
        {
            if (config == null) return;
            if (useRandomSeed || seed == 0) seed = Environment.TickCount;
            sysRand = new System.Random(seed);
            GenerateRooms();
        }

        public void GenerateRooms()
        {
            // Plan:
            // 1) Cell graph generation
            // 2) Assign roles
            // 3) Select prefabs
            // 4) Place rooms
            // 5) Setup doors
            // 6) Build corridors
            // 7) Merge floors
            // 8) Seed grass

            Debug.Log("[RoomsGeneratorFacade] Generated dungeon.");
        }

        private void SetupRoomDoors(Dictionary<Vector2Int, GameObject> placedRooms,
            Dictionary<Vector2Int, List<Vector2Int>> adjacency)
        {
            foreach (var kvp in placedRooms)
            {
                Vector2Int cellPos = kvp.Key;
                GameObject roomObj = kvp.Value;
                RoomController roomController = roomObj?.GetComponent<RoomController>();
                if (roomController == null) continue;

                List<Vector2Int> neighborDirections = new List<Vector2Int>();
                if (adjacency.ContainsKey(cellPos))
                {
                    foreach (var neighbor in adjacency[cellPos])
                        neighborDirections.Add(neighbor - cellPos);
                }
            }
        }
    }
}

```

```

        roomController.SetupDoors(neighborDirections);
    }
}
}
}

```

A.3 Код врага

```

using UnityEngine;

public abstract class EnemyBase : MonoBehaviour
{
    [Header("Stats")]
    public float maxHealth = 10f;
    protected float currentHealth;

    public virtual void Awake()
    {
        currentHealth = maxHealth;
    }

    public virtual void TakeDamage(float damage)
    {
        currentHealth -= damage;
        if (currentHealth <= 0)
        {
            Die();
        }
    }

    protected virtual void Die()
    {
        Destroy(gameObject);
    }
}

```

MeleeEnemyController.cs

```

using UnityEngine;

public class MeleeEnemyController : EnemyBase
{
    [Header("Movement")]
    public float PatrolSpeed = 2f;
    public float ChaseSpeed = 4f;
    public float PatrolRadius = 5f;
    public float PauseBetweenPoints = 2f;

    [Header("AI Vision")]
    public float VisionRange = 10f;
}

```

```

public LayerMask PlayerMask;
public LayerMask ObstacleMask;
public LayerMask EnemyMask;

[Header("Steering")]
public float NeighborSearchRadius = 1.5f;
public float NeighborAvoidWeight = 1.5f;
public float ShortRayLength = 1.5f;

public Transform Player { get; private set; }
public Animator Animator { get; private set; }
public Vector2 PatrolTarget { get; set; }

public IEnemyState CurrentState { get; private set; }
public IdleState IdleState { get; private set; }
public PatrolState PatrolState { get; private set; }
public ChaseState ChaseState { get; private set; }
public AttackState AttackState { get; private set; }

public override void Awake()
{
    base.Awake();
    Animator = GetComponent<Animator>();

    IdleState = new IdleState(this);
    PatrolState = new PatrolState(this);
    ChaseState = new ChaseState(this);
    AttackState = new AttackState(this);
}

private void Start()
{
    Player = GameObject.FindGameObjectWithTag("Player")?.transform;
    SwitchState(IdleState);
}

private void Update()
{
    CurrentState?.UpdateState();
}

public void SwitchState(IEnemyState newState)
{
    CurrentState?.ExitState();
    CurrentState = newState;
    CurrentState?.EnterState();
}

public void MoveTo(Vector2 target, float speed)
{
    Vector2 direction = (target - (Vector2)transform.position).normalized;
    MoveInDirection(direction, speed);
}

```

```

public void MoveInDirection(Vector2 direction, float speed)
{
    transform.Translate(direction * speed * Time.deltaTime);
    if (direction.magnitude > 0)
    {
        transform.localScale = new Vector3(Mathf.Sign(direction.x), 1f, 1f);
    }
    Animator.SetFloat("Speed", direction.magnitude * speed);
}

public bool CanSeePlayer()
{
    if (Player == null) return false;
    float distance = Vector2.Distance(transform.position, Player.position);
    if (distance > VisionRange) return false;

    RaycastHit2D hit = Physics2D.Raycast(transform.position, (Player.position -
transform.position).normalized, distance, ObstacleMask);
    return hit.collider == null;
}
}

```

EnemiesSpawner.cs

```

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

public class EnemiesSpawner : MonoBehaviour
{
    [System.Serializable]
    public struct EnemySpawn
    {
        public GameObject prefab;
        public int count;
    }

    [System.Serializable]
    public struct Wave
    {
        public string waveName;
        public EnemySpawn[] spawns;
        public float spawnInterval;
    }

    [Header("Configuration")]
    public Tilemap spawnTilemap;
    public List<TileBase> spawnMarkerTiles;
    public Wave[] waves;
}

```

```

public bool loopWaves = false;
public float globalSpawnInterval = 1f;
public int maxSimultaneousEnemies = 20;

[Header("Spawning Settings")]
public Vector3 spawnOffset = new Vector3(0.5f, 0.5f, 0);
public float spawnCheckRadius = 0.4f;
public int maxPositionAttempts = 20;

[Header("Events")]
public UnityEvent OnWaveStarted;
public UnityEvent OnWaveCompleted;
public UnityEvent OnAllWavesCompleted;

private List<Vector3Int> spawnCells = new List<Vector3Int>();
private bool isSpawning = false;
private Coroutine spawnRoutine;

public void Initialize()
{
    spawnCells.Clear();
    if (spawnTilemap == null) return;

    BoundsInt bounds = spawnTilemap.cellBounds;
    for (int x = bounds.xMin; x <= bounds.xMax; x++)
    {
        for (int y = bounds.yMin; y <= bounds.yMax; y++)
        {
            Vector3Int cell = new Vector3Int(x, y, 0);
            TileBase tile = spawnTilemap.GetTile(cell);
            if (tile == null) continue;

            foreach (var marker in spawnMarkerTiles)
            {
                if (tile == marker)
                {
                    spawnCells.Add(cell);
                    break;
                }
            }
        }
    }
}

public void StartSpawning()
{
    if (isSpawning) return;
    spawnRoutine = StartCoroutine(SpawnWavesCoroutine());
}

public void StopSpawning()
{
    if (spawnRoutine != null) StopCoroutine(spawnRoutine);
}

```

```

    isSpawning = false;
}

private IEnumerator SpawnWavesCoroutine()
{
    if (spawnCells.Count == 0) yield break;
    isSpawning = true;

    do
    {
        for (int w = 0; w < waves.Length; w++)
        {
            Wave wave = waves[w];
            OnWaveStarted?.Invoke();

            List<GameObject> toSpawn = new List<GameObject>();
            foreach (var sp in wave.spawns)
            {
                for (int i = 0; i < sp.count; i++) toSpawn.Add(sp.prefab);
            }

            foreach (var prefab in toSpawn)
            {
                while (transform.childCount >= maxSimultaneousEnemies) yield return null;

                Vector3 pos;
                if (!TryGetFreeSpawnPosition(out pos))
                    pos = GetWorldPosFromCell(spawnCells[Random.Range(0, spawnCells.Count)]);

                Instantiate(prefab, pos, Quaternion.identity, transform);
                yield return new WaitForSeconds(wave.spawnInterval > 0 ? wave.spawnInterval :
globalSpawnInterval);
            }

            while (transform.childCount > 0) yield return null;
            OnWaveCompleted?.Invoke();
        }
        OnAllWavesCompleted?.Invoke();
    } while (loopWaves && isSpawning);

    isSpawning = false;
}

private Vector3 GetWorldPosFromCell(Vector3Int cell)
{
    return spawnTilemap != null ? spawnTilemap.GetCellCenterWorld(cell) + spawnOffset : (Vector3)cell +
spawnOffset;
}

private bool TryGetFreeSpawnPosition(out Vector3 result)
{
    for (int i = 0; i < maxPositionAttempts; i++)
    {

```

```

        Vector3Int cell = spawnCells[Random.Range(0, spawnCells.Count)];
        Vector3 pos = GetWorldPosFromCell(cell);
        if (Physics2D.OverlapCircle(pos, spawnCheckRadius) == null)
        {
            result = pos;
            return true;
        }
    }
    result = GetWorldPosFromCell(spawnCells[Random.Range(0, spawnCells.Count)]);
    return false;
}
}

```

IEnemyState.cs

```

public interface IEnemyState
{
    void EnterState();
    void UpdateState();
    void ExitState();
}

```

IdleState.cs

```

using UnityEngine;

public class IdleState : IEnemyState
{
    private MeleeEnemyController enemy;
    private float timer;

    public IdleState(MeleeEnemyController enemy)
    {
        this.enemy = enemy;
    }

    public void EnterState()
    {
        timer = 0f;
        enemy.Animator.SetFloat("Speed", 0);
    }

    public void UpdateState()
    {
        timer += Time.deltaTime;
        if (timer >= enemy.PauseBetweenPoints)
        {
            enemy.SwitchState(enemy.PatrolState);
        }
    }
}

```

```

    public void ExitState() { }
}

```

PatrolState.cs

```
using UnityEngine;
```

```

public class PatrolState : IEnemyState
{
    private MeleeEnemyController enemy;

    public PatrolState(MeleeEnemyController enemy)
    {
        this.enemy = enemy;
    }

    public void EnterState()
    {
        SetNewPatrolPoint();
    }

    public void UpdateState()
    {
        if (enemy.CanSeePlayer())
        {
            enemy.SwitchState(enemy.ChaseState);
            return;
        }

        enemy.MoveTo(enemy.PatrolTarget, enemy.PatrolSpeed);

        if (Vector2.Distance(enemy.transform.position, enemy.PatrolTarget) < 0.2f)
        {
            enemy.SwitchState(enemy.IdleState);
        }
    }

    public void ExitState() { }

    private void SetNewPatrolPoint()
    {
        Vector2 origin = enemy.transform.position;
        enemy.PatrolTarget = origin + Random.insideUnitCircle * enemy.PatrolRadius;
    }
}

```

ChaseState.cs

```
using UnityEngine;
```

```
public class ChaseState : IEnemyState
```

```

{
    private MeleeEnemyController enemy;
    private float attackRange = 1.2f;

    public ChaseState(MeleeEnemyController enemy)
    {
        this.enemy = enemy;
    }

    public void EnterState() { }

    public void UpdateState()
    {
        if (enemy.Player == null) return;

        float distance = Vector2.Distance(enemy.transform.position, enemy.Player.position);
        if (distance <= attackRange)
        {
            enemy.SwitchState(enemy.AttackState);
            return;
        }

        if (enemy.CanSeePlayer())
        {
            enemy.MoveTo(enemy.Player.position, enemy.ChaseSpeed);
        }
        else
        {
            enemy.SwitchState(enemy.PatrolState);
        }
    }

    public void ExitState() { }
}

```

AttackState.cs

```

using UnityEngine;

public class AttackState : IEnemyState
{
    private MeleeEnemyController enemy;
    private float attackCooldown = 1f;
    private float timer = 0f;

    public AttackState(MeleeEnemyController enemy)
    {
        this.enemy = enemy;
    }

    public void EnterState()
    {

```

```

        timer = 0f;
    }

    public void UpdateState()
    {
        if (enemy.Player == null) return;

        float distance = Vector2.Distance(enemy.transform.position, enemy.Player.position);
        if (distance > 1.5f)
        {
            enemy.SwitchState(enemy.ChaseState);
            return;
        }

        timer += Time.deltaTime;
        if (timer >= attackCooldown)
        {
            PerformAttack();
            timer = 0f;
        }
    }

    public void ExitState() { }

    private void PerformAttack()
    {
        enemy.Animator.SetTrigger("Attack");
        PlayerController player = enemy.Player.GetComponent<PlayerController>();
        player?.TakeDamage(1);
    }
}

```

ДОДАТОК Б
Слайди презентації

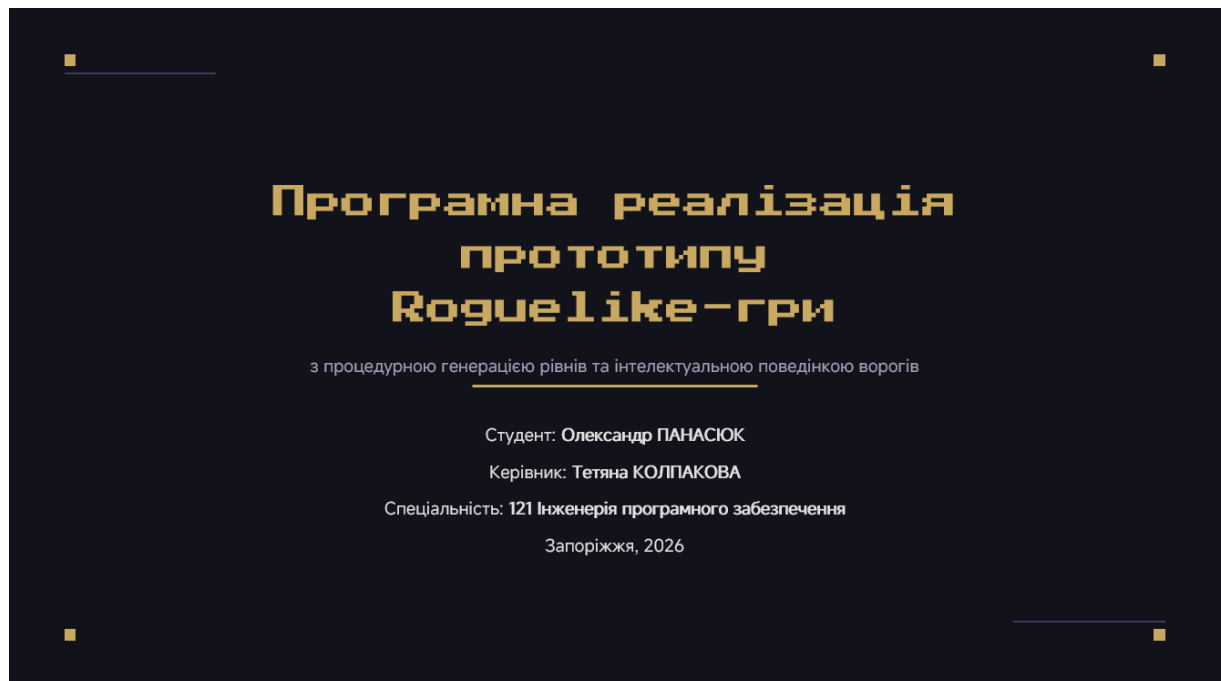


Рисунок В.1 – Слайд 1

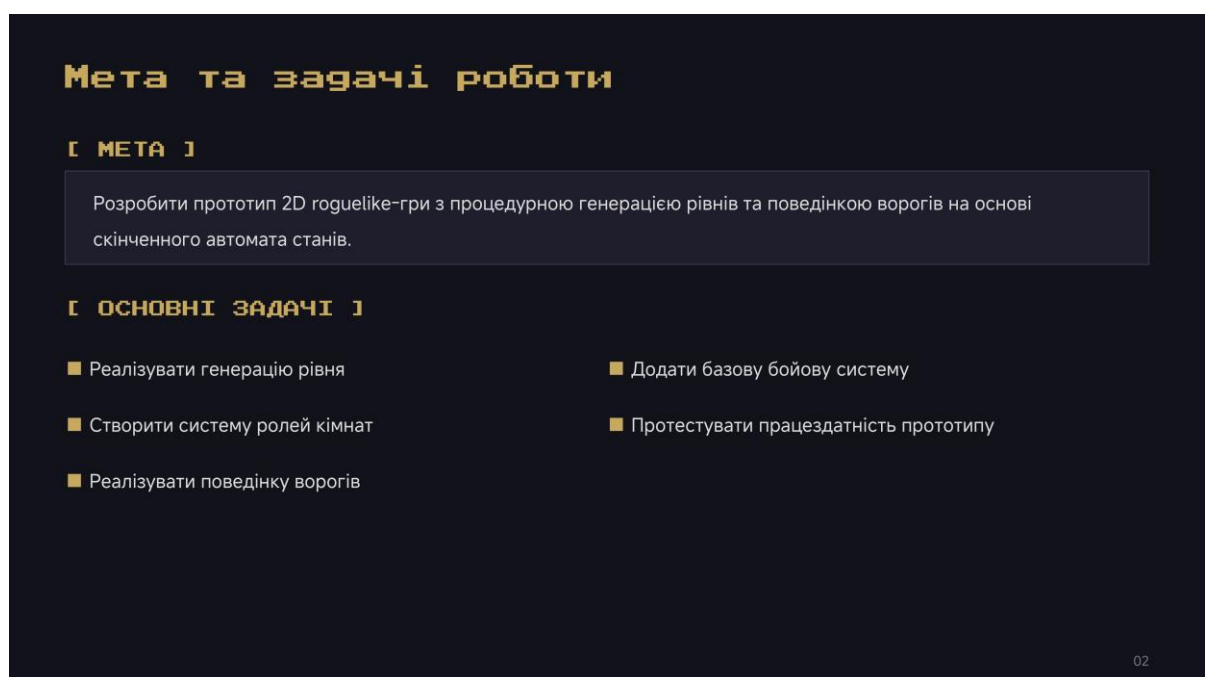


Рисунок В.2 – Слайд 2

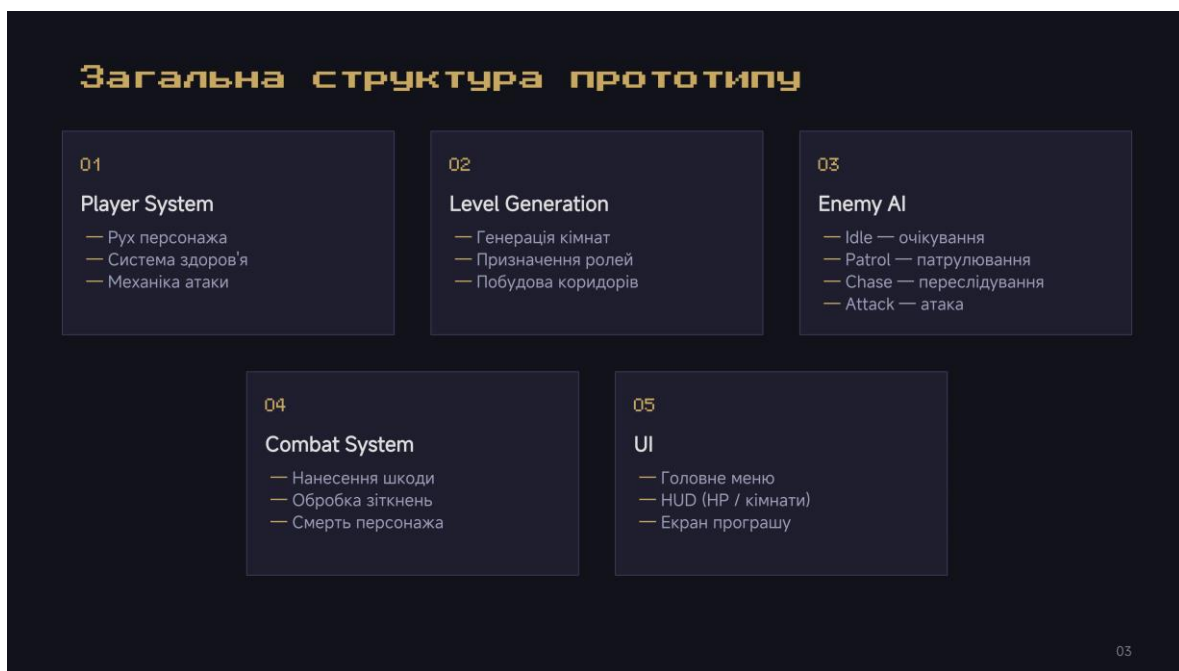


Рисунок В.3 – Слайд 3

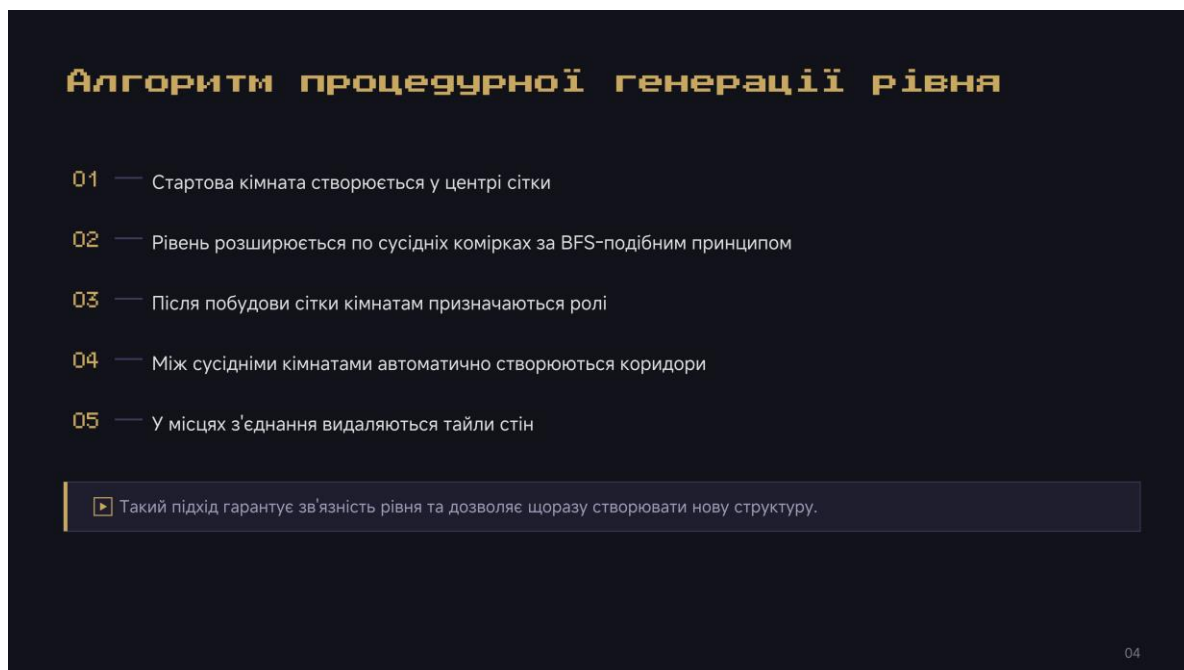
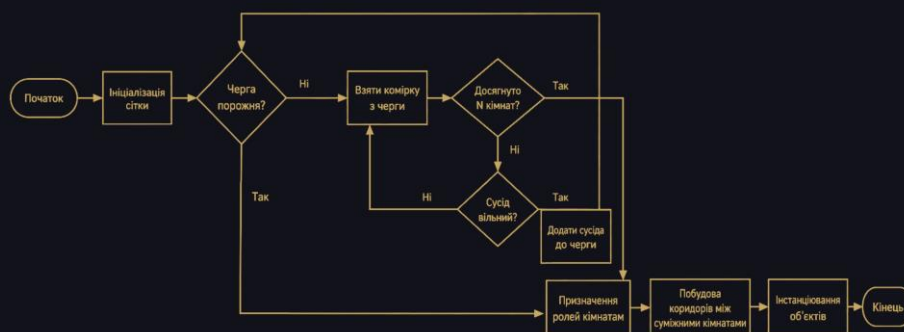


Рисунок В.4 – Слайд 4

Схематичне зображення алгоритму процедурної генерації рівня



05

Рисунок В.5 – Слайд 5

Ролі кімнат та логіка проходження

Start

Початок рівня

Normal

Звичайна кімната

Enemy

Кімната з ворогами

Chest

Кімната з нагородою

Boss

Фінальна кімната

▶ Гравець завжди починає зі стартової кімнати, натомість кімната з босом знаходиться в найвіддаленішій кімнаті.

06

Рисунок В.6 – Слайд 6



Рисунок В.7 – Слайд 7

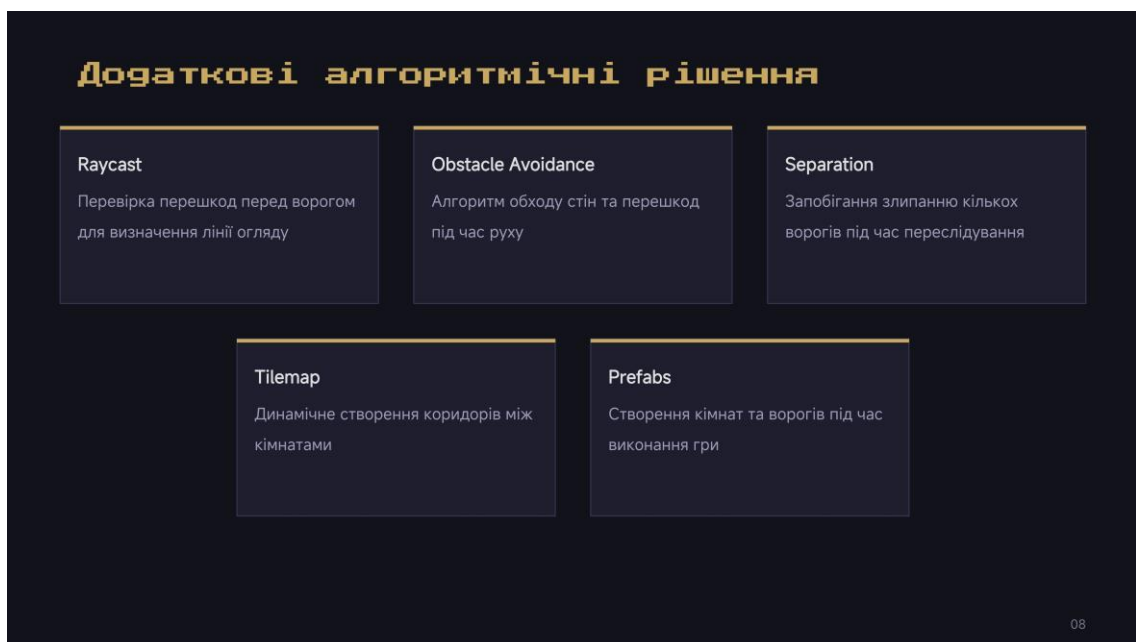


Рисунок В.8 – Слайд 8

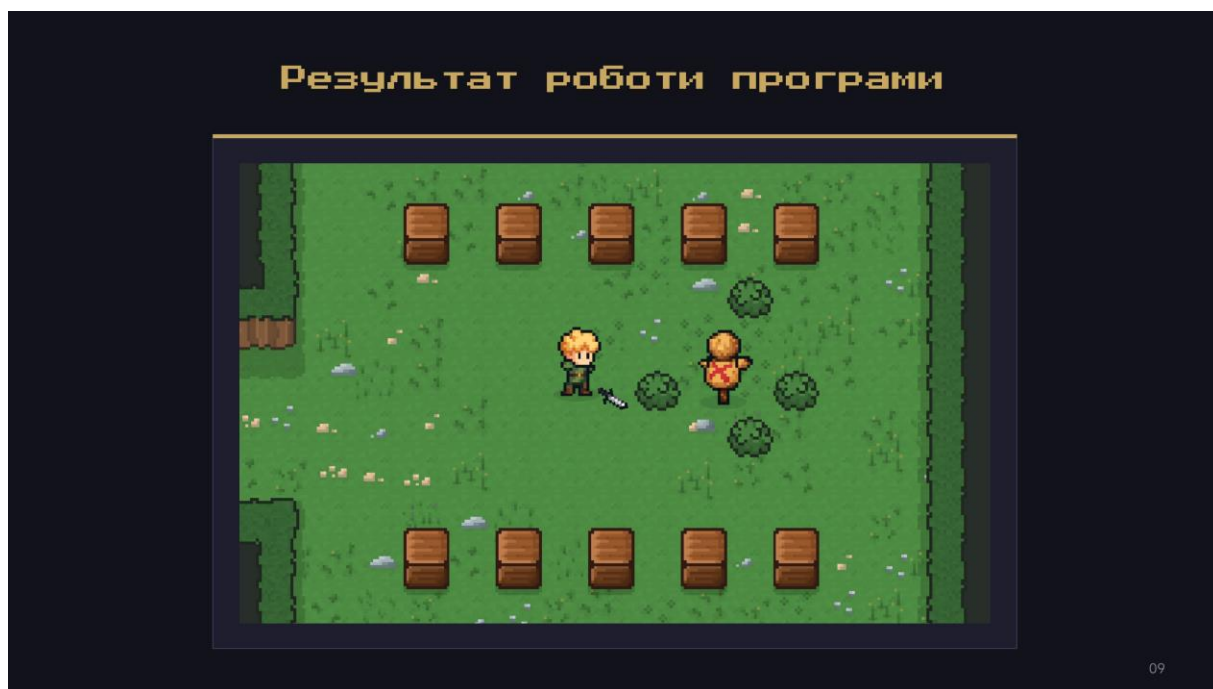


Рисунок В.9 – Слайд 9

Результати та висновки

[РЕАЛІЗОВАНО]

- Прототип 2D roguelike-гри
- Поведінку ворогів на FSM
- Процедурну генерацію рівнів
- Базову бойову систему
- Систему ролей кімнат, 16 сторінок:
Відсутні посилання на джерела, скріншот яких Ви покажете у своєму ПЗ.
Це планат!!! Дивіться приклад у шаблоні
- Меню, HUD та екран програшу

[ВИСНОВОК]

Розроблений прототип може бути використаний як основа для подальшого розвитку гри: додавання нових ворогів, предметів, рівнів та механік прогресії.

ДЯКУЮ ЗА УВАГУ!

10

Рисунок В.10 – Слайд 10