ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

КВАЛІФІКАЦІЙНА РОБОТА

Тема: «Комп’ютерна гра «Procedural Depth» з використанням алгоритмів генерації рівнів локацій»

Ступінь вищої освіти – бакалавр
Спеціальність – 122 «Комп’ютерні науки»
Освітня програма «Комп’ютерні науки»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав: здобувач 4 курсу
групи КН-22
Роман ДАВИДЮК
Керівник: викладач кафедри інформаційного менеджменту,
математики та статистики
Олег МУШИНСЬКИЙ

Засвідчую, що кваліфікаційна
робота оформлена відповідно
до ДСТУ 3008:2015 та не містить запозичень з праць
інших авторів без відповідних
посилань.
Здобувач:_____
(підпис)

м. Київ – 2026 рік

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

ЗАТВЕРДЖУЮ:

завідувач кафедри комп'ютерних наук

_____ Сергій МІЧКІВСЬКИЙ

«__» _____ 20__ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Давидюк Роман Сергійович

Тема роботи	Комп'ютерна гра «Procedural Depth» з використанням алгоритмів генерації рівнів локацій
Номер та дата наказу про затвердження теми	№ 133-7 від 22 грудня 2025 року
Коротка постановка завдання	Розробка комп'ютерної гри «Procedural Depth» з використанням алгоритмів для процедурної генерації локацій
Посилання на джерела інформації (не більше п'яти найменувань, які рекомендує науковий керівник)	1. Лайтарук І., Гришанович Т. Огляд перспектив швидкодії алгоритмів процедурної генерації в комп'ютерних іграх. Міжнародна науково-практична конференція «Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем». 2025, 195–198. URL: https://apcssm.vnu.edu.ua/index.php/conf/article/view/164/190 2. I. Karth and A. M. Smith. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. IEEE Transactions on Games. 2022. DOI: 10.1109/TG.2021.3076368.
Вимоги до кваліфікаційної роботи	Кваліфікаційна робота має передбачати теоретичне, системотехнічне або експериментальне дослідження складного спеціалізованого завдання або практичної проблеми в галузі комп'ютерних наук, що характеризується комплексністю та невизначеністю умов і потребує застосування теорій і методів інформаційних технологій.

Дата видачі завдання 27 грудня 2025 р.

Керівник Олег МУШИНСЬКИЙ

Здобувач освітнього ступеня бакалавра Роман ДАВИДЮК

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання	Примітка
Підготовчий етап			
1	Вибір напряму дослідження	01.12.2025 р.	*виконано*
2	Формування теми та призначення керівника	15.12.2025 р.	*виконано*
3	Затвердження теми кваліфікаційної роботи	22.12.2025 р.	*виконано*
4	Затвердження завдання на кваліфікаційну роботу	26.12.2025 р.	*виконано*
Основний етап			
5	Розробка концепції кваліфікаційної роботи	12.01.2026 р.	*виконано*
6	Підбір та вивчення джерел інформації з напряму дослідження. Огляд існуючих аналогів	19.01.2026 р.	*виконано*
7	Затвердження розширеної постановки завдання. Підготовка та подання керівникові розділу 1 кваліфікаційної роботи	16.03.2026 р.	*виконано*
8	Проєктування. Підготовка та подання керівникові розділу 2 кваліфікаційної роботи	23.03.2026 р.	*виконано*
9	Підготовка доповіді для експертизи стану виконання кваліфікаційної роботи (проміжний контроль)	30.03-04.04.2026 р.	*виконано*
10	Реалізація. Підготовка та подання керівникові розділу 3 кваліфікаційної роботи	06.04.2026 р.	*виконано*
11	Підготовка та подання керівнику першого варіанту всієї кваліфікаційної роботи	13.04.2026 р.	*виконано*
12	Доопрацювання кваліфікаційної роботи з урахуванням зауважень керівника та представлення керівникові доопрацьованого варіанту кваліфікаційної роботи	20.04.2026 р.	*виконано*
Завершальний етап			
13	Представлення рукопису для перевірки на плагіат	27.04-03.05.2026 р.	*виконано*
14	Підготовка презентації та доповіді на передзахист	04.05-10.05.2026 р.	*виконано*
15	Передзахист кваліфікаційної роботи	11.05-15.05.2026 р.	*виконано*
16	Доопрацювання роботи за результатами передзахисту	18.05-05.06.2026 р.	*виконано*
17	Експертиза роботи керівником та зовнішнім експертом	08.06-14.06.2026 р.	*виконано*
18	Доопрацювання доповіді та презентації для захисту	08.06-14.06.2026 р.	*виконано*
19	Захист кваліфікаційної роботи	15.06-21.06.2026 р.	*виконано*

Керівник Олег МУШИНСЬКИЙ

Здобувач освітнього ступеня бакалавра Роман ДАВИДЮК

Давидюк Р. С. Розробка комп'ютерної гри «Procedural Depth» з використанням алгоритмів генерації рівнів локацій.

Пояснювальна записка кваліфікаційної роботи за спеціальністю 122 – Комп'ютерні науки (освітня програма – Комп'ютерні науки). – ВНЗ «Університет економіки та права «КРОК», Київ, 2026.

У даній роботі описано повний цикл розробки 2D комп'ютерної гри жанру roguelike на Unity з використанням процедурної генерації локацій. Досліджено предметну область, проаналізовано аналоги, сформульовано функціональні, нефункціональні та системні вимоги до програмного продукту. Запропоновано архітектуру гри, виконано моделювання поведінки та структури системи за допомогою UML-діаграм і моделі C4, описано алгоритм генерації рівня типу «кімнати + коридори», правила розподілу ворогів і ресурсів, а також принципи організації інтерфейсу користувача. У роботі наведено особливості реалізації ключових модулів, результати функціонального та модульного тестування, інструкцію користувача і підсумкові висновки щодо досягнення мети дослідження.

Ключові слова: комп'ютерна гра, roguelike, процедурна генерація, Unity, UML, Tilemap, тестування.
Табл. 6. Рис. 18. Бібліогр. 12.

Davydiuk R.S. The computer game «Procedural Depth» using algorithms for generating location levels.

Explanatory note of qualification work in specialty 122 – Computer Science, Bachelor's degree. – University of Economics and Law "KROK", Kyiv, 2026.

This paper describes the full development cycle of a 2D roguelike game in Unity using procedural generation of locations. The subject area is analysed, existing analogues are reviewed, and the functional, non-functional, and system requirements for the software product are formulated. The architecture of the game is proposed, the behaviour and structure of the system are modelled by means of UML diagrams and the C4 model, and the algorithm for generating room-and-corridor levels, the rules for distributing enemies and resources, and the organisation of the user interface are described. The paper also presents implementation details of the key modules, results of functional and unit testing, a short user guide, and final conclusions concerning the achievement of the research goal.

Keywords: computer game, roguelike, procedural generation, Unity, UML, Tilemap, testing.
Tables 6. Figures 18. Bibliography 12.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

AI (Artificial Intelligence)	штучний інтелект, що використовується у відеоіграх для моделювання поведінки NPC, балансування та інших механік.
NPC (Non-Player Character)	неігровий персонаж, що керується алгоритмами AI.
Roguelike	жанр ігор із процедурною генерацією рівнів, високою складністю та реіграбельністю.
PCG (Procedural Content Generation)	процедурна генерація контенту.
BSP (Binary Space Partitioning)	метод просторового поділу, який застосовують для генерації кімнатних структур.
WFC (WaveFunctionCollapse)	сімейство алгоритмів генерації контенту на основі обмежень і локальних шаблонів [2].
Perlin Noise	шумова функція, що використовується для генерації природних структур [1].
Pipeline	послідовність етапів створення ігрового продукту від прототипування до підтримки.
UI (User Interface)	інтерфейс користувача у грі.
Game Loop	основний цикл оновлення стану гри та рендерингу кадру.
Seed	початкове значення генератора випадкових чисел для відтворюваності процедурної генерації.
Room-based generation	підхід до генерації рівня, де карта складається з кімнат і коридорів.

ВСТУП

Актуальність теми. Розвиток комп’ютерних ігор є однією з найбільш динамічних сфер сучасних інформаційних технологій. Вимоги користувачів до різноманітності та якості ігрового контенту стимулюють розробників використовувати алгоритмічні підходи до створення ігрових рівнів та подій.

Особливої популярності набули ігри жанру roguelike, у яких процедурна генерація забезпечує унікальність кожного проходження, формує відчуття дослідження та підвищує реіграбельність. Для таких ігор важливо поєднати випадковість генерації з контрольованою структурою рівнів та їх прохідністю.

Гра «Procedural Depth» є 2D проєктом жанру roguelike, у якому ігрові локації формуються автоматично при запуску. У межах проєкту передбачається реалізація генерації структури рівня типу «кімнати + коридори» із подальшим розміщенням ігрових сутностей за заданими правилами.

Мета дослідження. Метою проєкту є розробка програмного забезпечення комп’ютерної гри «Procedural Depth» із процедурною генерацією локацій.

Для досягнення мети виконуються **завдання дослідження**:

– аналіз предметної області та існуючих аналогів;
– визначення функціональних і нефункціональних вимог;
– проєктування структури програмного продукту;
– реалізація алгоритму генерації рівнів у Unity;
– тестування програмного продукту на стабільність та продуктивність.

Об’єкт дослідження – процес створення 2D гри з процедурною генерацією локацій.

Предмет дослідження – комп’ютерна гра «Procedural Depth» та алгоритми процедурної генерації рівнів.

Методи дослідження. Дослідження базується на використанні аналітичних, порівняльних і експериментальних методів. Аналітичний метод застосовано для вивчення сучасних підходів до процедурної генерації в іграх, а також підходів генерації на основі обмежень. Порівняльний аналіз використано для оцінки конкурентів на ринку. Експериментальні методи застосовано під час розробки та тестування ігрових механік.

Практичне значення. Результати роботи можуть бути використані як основа для подальшого розвитку проєкту з додаванням нових типів кімнат та систем прогресії.

Структура та обсяг пояснювальної записки. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку джерел посилання. Загальний обсяг сторінок 50, робота містить 18 рисунків та 6 таблиць.

РОЗДІЛ 1
ПОСТАНОВКА ЗАВДАННЯ НА КОМП'ЮТЕРНУ ГРУ «PROCEDURAL DEPTH»

1.1 Опис предметної області

У сучасному світі комп'ютерні ігри відіграють важливу роль у цифрових розвагах. Ігрова індустрія є одним із секторів, що розвиваються найшвидше, оскільки поєднує програмування, дизайн, звук та тестування для створення інтерактивних продуктів.

Зростання конкуренції на ринку спонукає розробників підвищувати реіграбельність продуктів. Одним із підходів є процедурна генерація контенту, що дозволяє автоматично створювати великі обсяги рівнів і варіацій подій.

Для реального часу важливим фактором є швидкодія алгоритмів та їх ресурсна вимогливість, що підкреслено в оглядах швидкодії процедурної генерації [1].

Жанр roguelike передбачає динамічне формування рівнів у вигляді системи кімнат і коридорів. Для якісного ігрового процесу рівні мають бути логічно зв'язаними та прохідними. Узагальнену схему взаємопроникнення жанрів, з якої випливає місце roguelike, подано на рис. 1.1.

Під час розробки гри важливо врахувати pipeline: прототипування, проєктування, реалізація, тестування та підтримка. Відповідну схему життєвого циклу ігрового продукту наведено на рис. 1.2.

Для roguelike-ігор якість генерації визначається не лише випадковістю, а й структурною послідовністю. Якщо генератор не контролює зв'язність, рівень може містити недоступні кімнати або некоректні переходи. Тому в межах проєкту передбачається перевірка прохідності та обмеження на розташування ключових елементів.

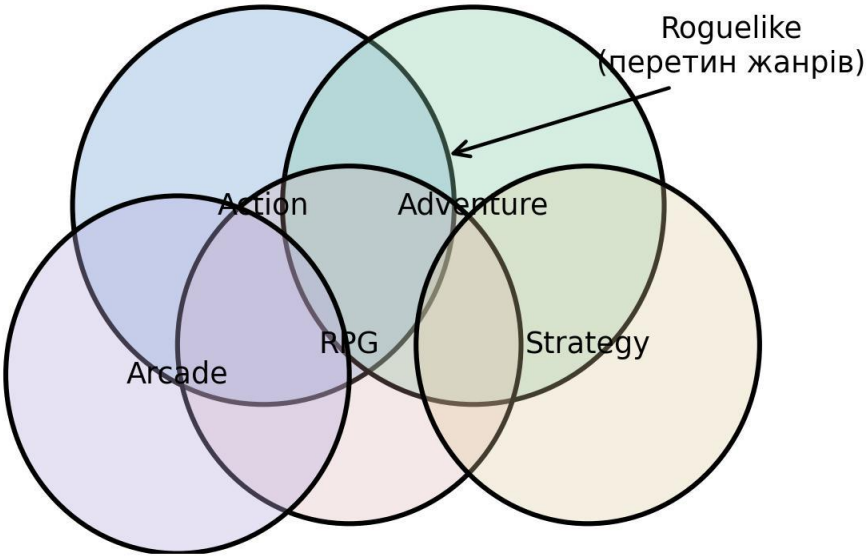

Рисунок 1.1 – Узагальнена діаграма перетину жанрів ігрової індустрії

Джерело: розроблено автором

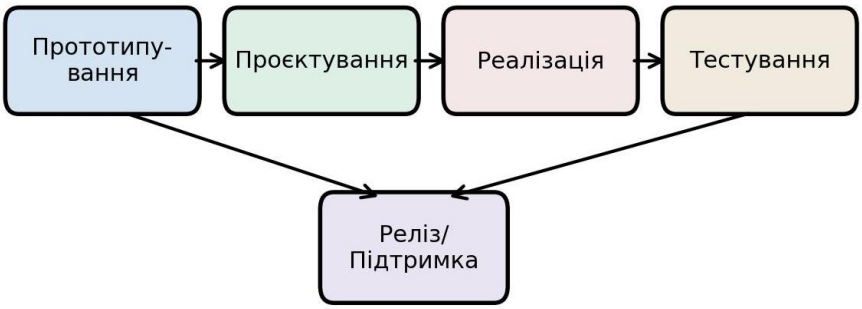

Рисунок 1.2 – Схема pipeline розробки ігрового продукту

Джерело: розроблено автором

Окремим аспектом є баланс: надмірна випадковість у розміщенні ворогів може створювати нерівномірну складність. Під час генерації доцільно застосовувати правила розподілу складності по зоні рівня.

У середовищі Unity реалізація процедурної генерації може бути виконана як окремий модуль, що працює до початку активного ігрового циклу або під час переходів між рівнями. Приклад структури рівня «кімнати + коридори» наведено на рис. 1.3, а порівняльну схему підходів процедурної генерації подано на рис. 1.4.

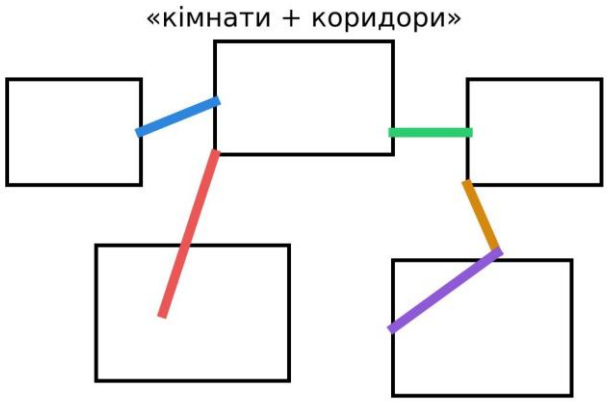

Рисунок 1.3 – Приклад структури рівня «кімнати + коридори»

Джерело: розроблено автором

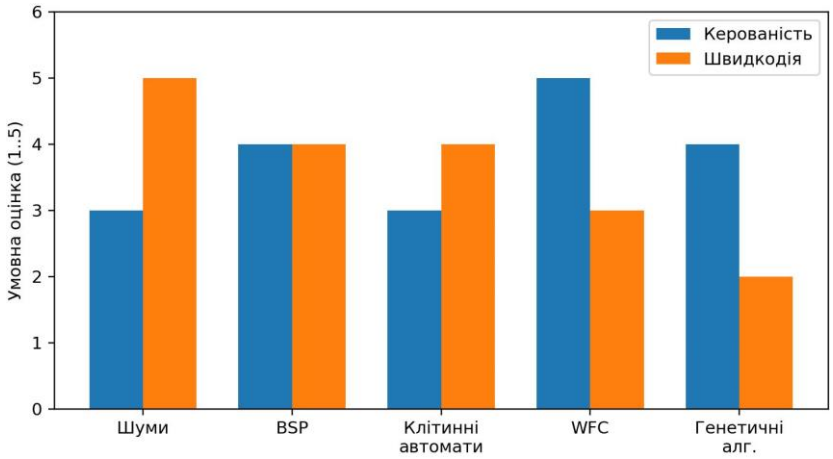

Рисунок 1.4 – Порівняльна схема підходів процедурної генерації (умовна оцінка)

Джерело: розроблено автором

1.2 Визначення потенційних конкурентних переваг

Жанр roguelike має значну кількість представників на ринку. Основними конкурентами проєкту «Procedural Depth» можна вважати «The Binding of Isaac», «Dead Cells» та «Hades». Для візуального порівняння приклади ігор наведено на рис. 1.5–1.7.

«The Binding of Isaac» відзначається великою кількістю предметів і комбінацій, що підвищує реіграбельність. Водночас структура кімнат у багатьох випадках є повторюваною.

«Dead Cells» робить акцент на динамічному геймплеї та прогресії. Перевагою є темп бою, недоліком – складність балансування та повторюваність окремих зон.

«Hades» інтегрує сюжетну складову та забезпечує різноманітність проходжень, але потребує значних ресурсів на контент і внутрішні механіки. Для детального аналізу конкурентних переваг проведено порівняння ключових характеристик у табл. 1.1.

Рисунок 1.5 – Приклад ігрового процесу «The Binding of Isaac: Rebirth»
Джерело: Push Square (скріншот гри)

Таке порівняння є важливим не лише з позиції візуального стилю, а й з точки зору організації геймплейного циклу. У конкурентних проєктах найбільшу цінність для користувача створює поєднання швидкого входження в сесію, зрозумілого темпу дослідження карти та відчуття поступового ускладнення, що мотивує до повторних проходжень.

Порівняння з «The Binding of Isaac» показує, що для забезпечення конкурентоспроможності «Procedural Depth» доцільно поєднати високу реіграбельність із більш керованою структурою рівня. На відміну від випадкового нагромадження кімнат і подій, у власному проєкті акцент

робиться на процедурній побудові локацій із контролем прохідності, темпу проходження та розподілу ресурсів.

Крім того, важливо забезпечити баланс між варіативністю та зрозумілістю ігрового простору. Гравець повинен відчувати новизну кожного проходження, але водночас швидко зчитувати логіку рівня, маршрути пересування, розташування ворогів і точки отримання винагород. Саме такий підхід формує практичну основу конкурентних переваг майбутнього програмного продукту.

Рисунок 1.6 – Приклад ігрового процесу «Dead Cells»

Джерело: Push Square (скріншот гри)

Рисунок 1.7 – Офіційний промо-арт гри «Hades»

Джерело: Supergiant Games (офіційні матеріали)

Таблиця 1.1 – Переваги «Procedural Depth» над конкурентами

Перевага	*Реалізація*
Структурована процедурна генерація	Генерація рівнів у вигляді кімнат і коридорів із контролем прохідності та логічної зв’язності.
Керований баланс складності	Правила розподілу ворогів і ресурсів залежно від віддалення від старту та параметрів складності.
Модульність та швидкість розширення	Архітектура дозволяє додавати нові типи кімнат, подій і ворогів без перепроєктування ядра.
Чесна модель прогресу	Відсутність pay-to-win механік; результат залежить від навичок гравця та тактики.

1.3 Постановка завдання на розробку програмного продукту

Завданням розробки є створення програмного забезпечення комп’ютерної гри «Procedural Depth» із процедурною генерацією локацій. Програмний продукт має забезпечувати гравцю можливість проходження послідовності згенерованих рівнів із взаємодією з ворогами та внутрішньоігровими об’єктами.

Для реалізації поставленої задачі необхідно виконати такі етапи: проєктування структури гри та модулів; підбір і підготовка ассетів (графіка, звук); моделювання геймплейних механік; реалізація програмного забезпечення у Unity; тестування та усунення дефектів.

Для вирішення задачі будуть реалізовані підсистеми: система керування гравцем; система генерації рівня; система спавну та базової поведінки ворогів; система колізій; UI для відображення станів (здоров’я, очки, рівень); система перезапуску і завершення сесії.

Функціональні вимоги: автоматична генерація унікального рівня при кожному запуску; логічна прохідність і зв’язність локацій; коректна взаємодія гравця з ворогами; обробка колізій; система відображення стану гри; можливість перезапуску та переходу між рівнями.

Нефункціональні вимоги: стабільна робота без критичних помилок; прийнятна швидкодія генерації; мінімізація просідань FPS у геймплеї; підтримуваність і модульність коду; можливість розширення контенту.

Системні вимоги: операційна система Windows; рушій Unity; мова програмування C#; підтримка базової цільової роздільної здатності 1920×1080.

Вхідними даними для програмного продукту є команди користувача з клавіатури, початкові параметри ігрової сесії, налаштування генерації рівня, а також службові дані про характеристики кімнат, ворогів, ресурсів і точок взаємодії.

Вихідними даними є згенерована структура локації, візуальне відображення стану гри на екрані, результати взаємодії гравця з ворогами та об’єктами середовища, а також підсумкові показники проходження, що передаються до елементів інтерфейсу користувача.

Висновки до розділу 1

Проведено дослідження предметної області, представленої 2D грою жанру roguelike та процесом її розробки. Визначено роль генератора локацій як ключового модуля продукту.

Аналіз конкурентів дозволив визначити потенційні конкурентні переваги «Procedural Depth» та сформувати перелік вимог до програмного продукту.

Сформульовано постановку завдання на розробку програмного продукту, визначено підсистеми та вимоги, що створює основу для подальших розділів.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Моделювання поведінки продукту

Поведінкова модель гри визначає, які дії може виконувати основний актор - гравець, а також як система реагує на ці дії в межах однієї ігрової сесії. У «Procedural Depth» основним актором є користувач, який ініціює запуск гри, керує персонажем, взаємодіє з рівнем та одержує зворотний зв’язок через інтерфейс користувача.

Ключові сценарії охоплюють запуск застосунку, генерацію нового рівня, вхід у цикл ігрової сесії, рух персонажа, зіткнення з ворогами, оновлення стану здоров’я та рахунку, а також перезапуск гри після завершення спроби. Таке розбиття дозволяє відокремити сценарії ініціалізації від сценаріїв оперативної взаємодії у геймплеї.

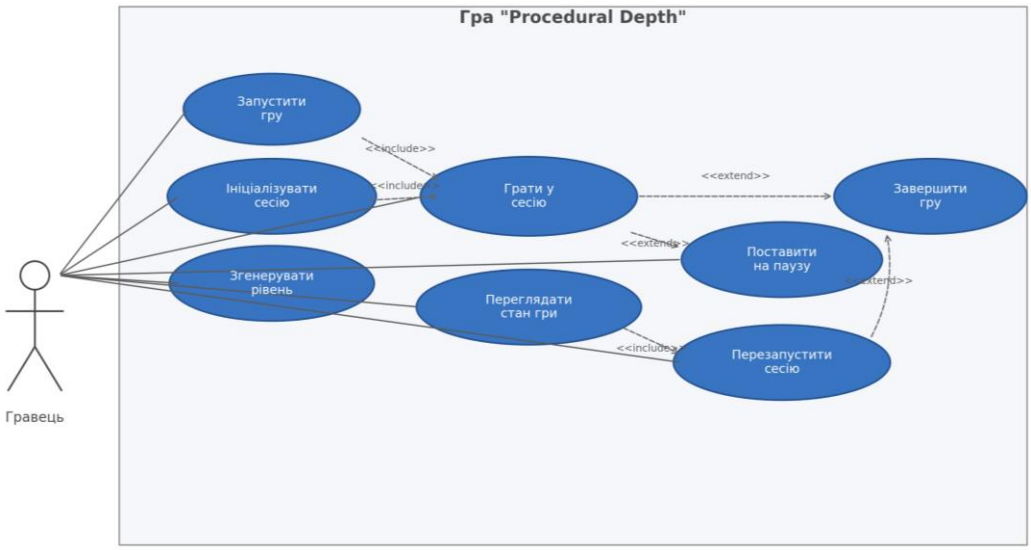

Рисунок 2.1 - Діаграма варіантів використання комп’ютерної гри «Procedural Depth»

Джерело: розроблено автором

На діаграмі варіантів використання показано, що гравець напряму взаємодіє з функціями запуску застосунку, генерації нового рівня, керування

персонажем, перегляду поточного стану гри та перезапуску сесії. Модель відображає функціональне ядро продукту і дає змогу перевірити повноту сформульованих у першому розділі вимог.

Важливим аспектом є те, що генерація рівня в моделі поведінки представлена як окремий варіант використання. Це підкреслює незалежність модуля генерації від логіки керування персонажем і спрощує подальше тестування: генератор можна перевіряти окремо на коректність структури карти, не запускаючи повну бойову сесію.

Для більш повного опису поведінки системи кожний варіант використання доцільно розглядати через передумови, основний сценарій та результат виконання. Передумовою сценарію «Запустити гру» є наявність встановленого застосунку та доступність конфігураційних ресурсів. Після запуску користувач переходить до стартового екрана, де система ініціалізує базові сервіси, завантажує параметри поточної збірки й готує сцену до початку сесії. Результатом цього сценарію є готовий до взаємодії інтерфейс та коректно підготовлений ігровий контекст.

Сценарій «Ініціалізувати сесію» охоплює службові дії, які не завжди помітні користувачеві, але є критичними для стабільної роботи гри. На цьому етапі зчитуються налаштування генератора, формується seed, перевіряється доступність потрібних префабів, а також створюються початкові об’єкти сцени. Саме тому в моделі поведінки ініціалізація виділена як окремий варіант використання, пов’язаний із запуском ігрової сесії, а не прихований усередині інших процесів.

Сценарій «Згенерувати рівень» відображає ключову функцію програмного продукту. Система отримує набір параметрів карти, допустимі межі розмірів кімнат, правила формування коридорів і конфігурації спавну. Далі формується набір кімнат-кандидатів, виконується перевірка їхньої сумісності, а після побудови геометрії карти система переходить до розміщення ворогів, ресурсів і точок взаємодії. Результатом сценарію є коректно згенерований і прохідний рівень, готовий до старту сесії.

Сценарій «Грати у сесію» є центральним у поведінковій моделі, оскільки саме в ньому поєднуються рух персонажа, бойова взаємодія, обробка зіткнень, оновлення стану здоров’я, підрахунок очок та відображення службової інформації через HUD. У межах цього сценарію гравець постійно отримує зворотний зв’язок від системи, а зміни в моделі стану негайно відображаються в інтерфейсі. Така організація дає змогу розмежувати керування ігровими подіями та їх візуальне подання.

Окремої уваги потребують сценарії «Поставити на паузу», «Переглядати стан гри», «Перезапустити сесію» та «Завершити гру». Пауза розглядається як розширення основної сесії, оскільки вона тимчасово припиняє активне оновлення логіки, але не руйнує поточний стан рівня. Перезапуск, навпаки, ініціює створення нової спроби, повторний виклик генератора та очищення тимчасових даних попередньої сесії. Завершення гри фіксує підсумковий результат і переводить користувача на службовий екран, з якого він може або вийти, або розпочати нове проходження.

З точки зору тестування така деталізація варіантів використання є корисною ще й тому, що дозволяє безпосередньо зіставити поведінкову модель із набором функціональних перевірок. Кожному основному сценарію відповідає окрема група тестів: перевірка запуску, перевірка коректності генерації, перевірка обробки керування, перевірка синхронізації HUD, а також перевірка завершення і повторного старту. У підсумку use case-модель слугує не лише засобом візуального опису, а й основою для побудови тестових сценаріїв третього розділу.

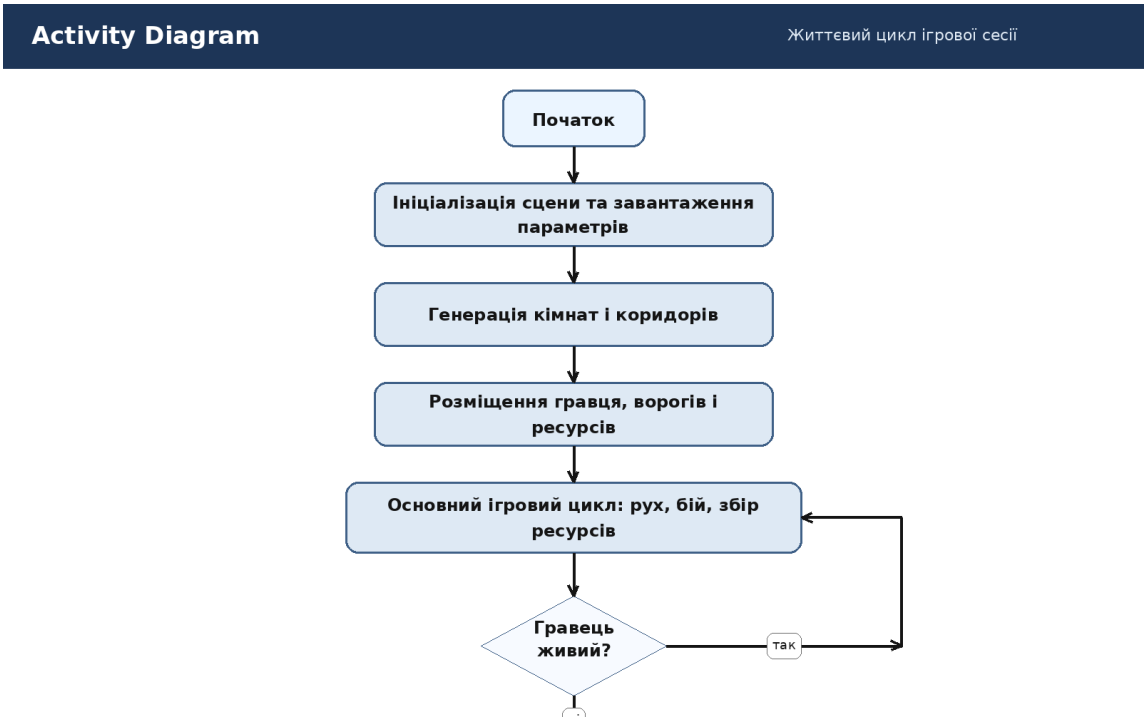

Рисунок 2.2 - Діаграма діяльності ігрової сесії

Джерело: розроблено автором

Діаграма діяльності деталізує послідовність кроків у межах однієї сесії: ініціалізація сцени, завантаження конфігурацій, генерація структури кімнат, розміщення ворогів і ресурсів, вхід до основного ігрового циклу, оновлення HUD, обробка завершення сесії та можливий перезапуск.

Така модель дозволяє чітко виділити точки контролю, у яких можливе виникнення помилок: перевірка коректності seed, валідація зв’язності кімнат, запобігання накладенню об’єктів, а також очищення тимчасових даних під час переходу між спробами. Усі ці точки надалі враховані у плані тестування третього розділу.

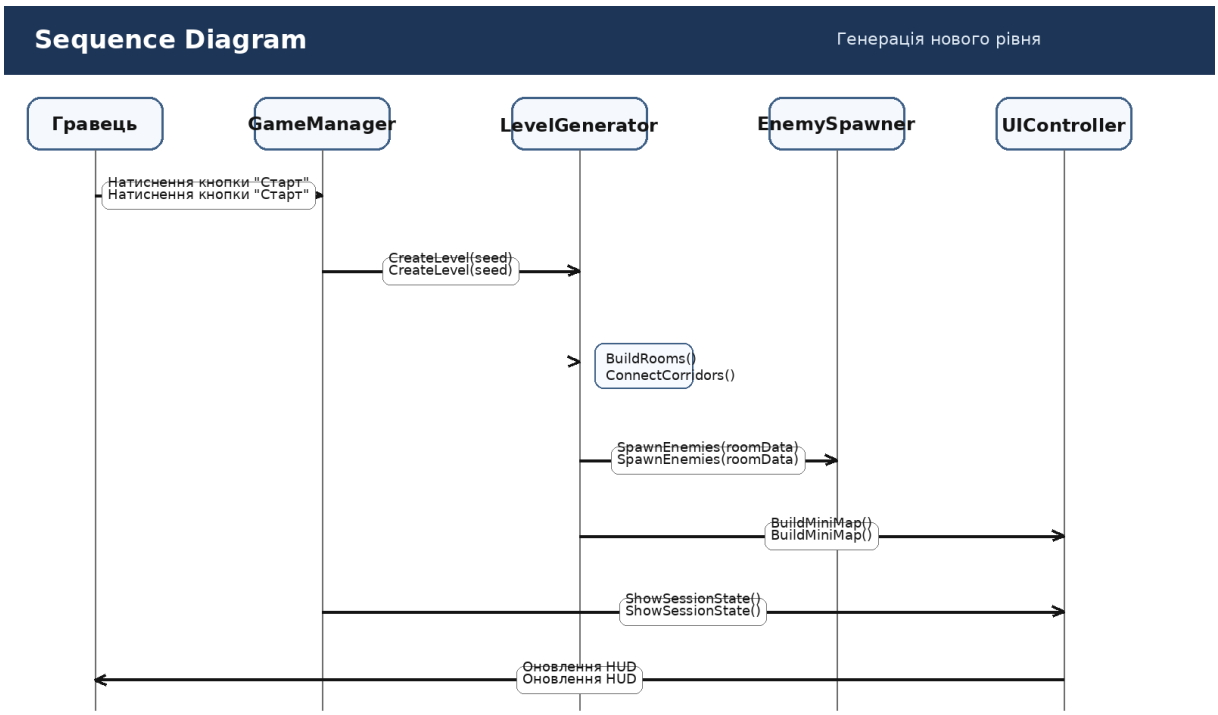

Рисунок 2.3 - Діаграма послідовності взаємодії основних менеджерів гри

Джерело: розроблено автором

Діаграма послідовності відображає обмін повідомленнями між об’єктами GameBootstrap, LevelGenerator, EnemySpawner, GameStateManager та UIController. Після запуску сцени ініціалізатор формує контекст сесії, передає запит генератору рівня, після чого модуль спавну використовує результати генерації для розміщення ворогів. Далі стан гри та параметри персонажа відображаються в UI.

Діаграма послідовності модель корисна тим, що дозволяє виявити залежності між окремими менеджерами. Зокрема, UI не повинен оновлюватися раніше, ніж сформовано стартовий стан сесії, а модуль ворогів не може працювати без валідної геометрії кімнат. Це зменшує ризик виникнення помилок ініціалізації та підказує, які інтерфейси необхідно формалізувати на рівні коду.

2.2 Моделювання архітектури та структури продукту

Архітектура програмного продукту побудована за багаторівневим принципом, який відповідає методичним вимогам до розробки програмного забезпечення [3]. У межах роботи виділено рівень зберігання і конфігурації даних, рівень доступу до цих даних, прикладний рівень ігрової логіки та рівень користувацького інтерфейсу.

Рівень зберігання даних реалізується через ресурсні ассети, префаби, таблиці налаштувань і серіалізовані об’єкти, що зберігають параметри кімнат, ворогів, шансів спавну та базових характеристик сесії. У Unity доцільно використовувати ScriptableObject як контейнер конфігурацій, що зменшує зв’язність між об’єктами сцени та полегшує повторне використання даних [8].

Рівень доступу до даних відповідає за завантаження конфігурацій, передачу їх генератору та іншим підсистемам, а також за локальне збереження окремих показників прогресу, наприклад рекорду або параметрів останньої сесії. Прикладний рівень містить модулі генерації, керування станами гри, обробки фізики та бойової логіки. Рівень UI формує відображення HUD, меню старту, екрана завершення та службових повідомлень [10].

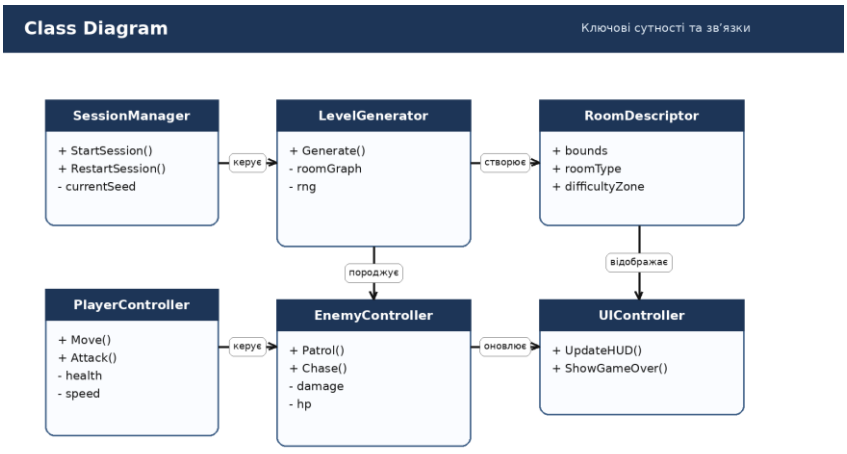

Рисунок 2.4 - Узагальнена діаграма класів програмного продукту

Джерело: розроблено автором

Діаграма класів демонструє, що центральними сутностями системи є GameBootstrap, LevelGenerator, RoomModel, PlayerController, EnemyController, GameStateManager та UIController. Клас RoomModel зберігає геометричні характеристики кімнати, точки входу та службові прапорці для алгоритму з'єднання. LevelGenerator створює колекцію таких моделей та передає її іншим модулям.

PlayerController і EnemyController інкапсулюють поведінку ігрових персонажів, а GameStateManager відповідає за переходи між станами «підготовка», «активна сесія», «перемога або поразка» та «перезапуск». UIController не містить бойової логіки, а лише підписується на події зміни стану та оновлює відповідні елементи HUD. Це зменшує ризик дублювання відповідальностей і робить архітектуру більш підтримуваною.

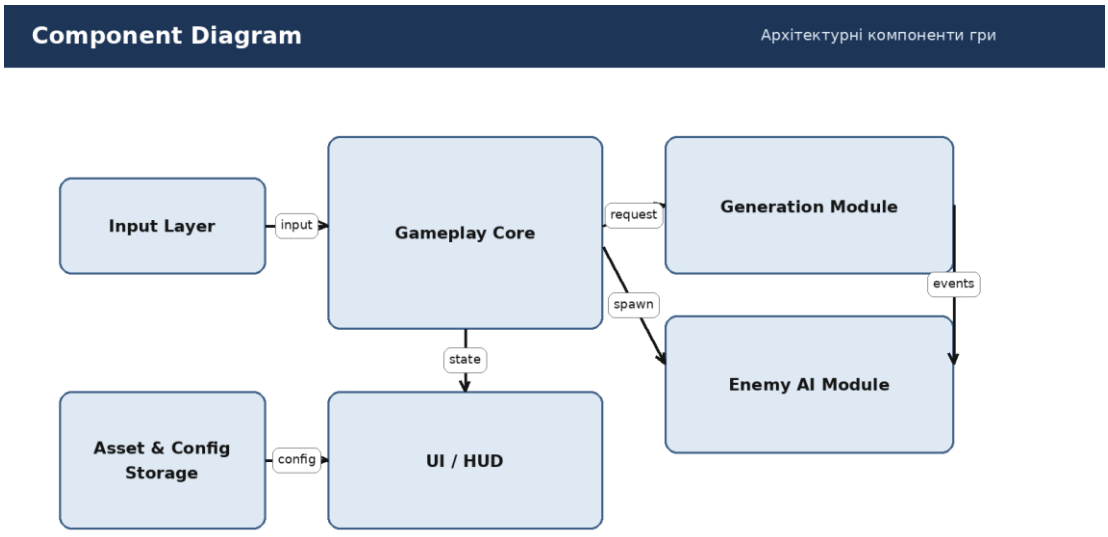

Рисунок 2.5 - Діаграма компонентів архітектури гри

Джерело: розроблено автором

Компонентна модель деталізує взаємодію між рівнями введення, ігровою логікою, модулем генерації, AI ворогів, UI та сховищем конфігурацій. Введення команд від користувача обробляється Input Layer,

далі події передаються до Gameplay Core, який викликає Generation Module та Enemy AI Module за потреби, а результати відображаються в UI / HUD.

Такий розподіл дозволяє локалізувати зміни. Наприклад, заміна алгоритму генерації не вимагає зміни контролера гравця або HUD, якщо зовнішній контракт модуля генерації залишається незмінним. Аналогічно, можна змінити модель поведінки ворогів без перепроєктування сценового UI.

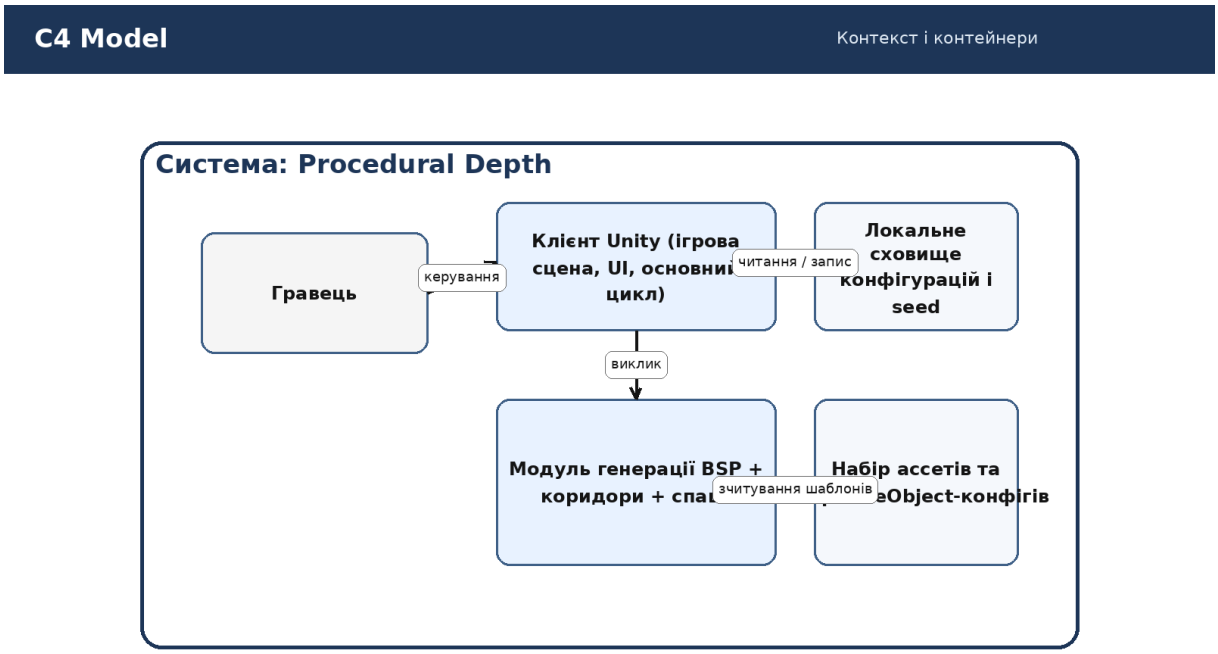

Рисунок 2.6 - Узагальнена C4-модель програмного продукту

Джерело: розроблено автором

C4-модель доповнює UML-діаграми та показує систему на різних рівнях деталізації. На контекстному рівні програмний продукт взаємодіє лише з гравцем і локальними конфігураційними ресурсами. На контейнерному рівні система складається з клієнтського застосунку Unity, набору ассетів конфігурації та локального шару збереження. На компонентному рівні виділяються менеджери запуску, генерації, геймплейної логіки, AI та UI.

Використання C4-моделі зручне для пояснення архітектури на захисті, оскільки вона зберігає баланс між технічною деталізацією та наочністю.

Якщо діаграма класів корисна для розробника, то C4-модель краще показує загальну будову системи комісії та дозволяє швидко пояснити, де саме знаходиться логіка генерації, де зберігаються дані та яким чином користувач взаємодіє з продуктом.

Таблиця 2.1 - Основні компоненти програмного продукту та їх відповідальність

Компонент	*Призначення*	*Вхідні дані*	*Вихідний результат*
GameBootstrap	Ініціалізація сцени, завантаження конфігурацій, запуск сесії	Параметри старту, seed, посилання на конфіги	Підготовлений контекст гри
LevelGenerator	Побудова структури кімнат і коридорів	Обмеження карти, seed, шаблони кімнат	Колекція кімнат і з’єднань
EnemySpawner	Розміщення ворогів відповідно до зони складності	Список кімнат, правила спавну, коефіцієнт складності	Активні об’єкти ворогів
GameStateManager	Керування переходами між станами сесії	Події бою, стан персонажа, завершення рівня	Поточний стан гри
UIController	Відображення HUD і службових екранів	Події зміни стану, параметри персонажа	Оновлені елементи інтерфейсу
ConfigStorage	Зберігання конфігурацій у ScriptableObject і префабах	Файли ресурсів, редакторні дані	Набір доступних налаштувань

З таблиці 2.1 видно, що кожен компонент має чітко визначений контракт. Такий підхід спрощує розробку й дає змогу тестувати компоненти

окремо. Наприклад, LevelGenerator можна перевіряти на зв’язність та коректність розмірів кімнат без необхідності запускати повноцінну анімацію чи бойову систему.

Таблиця 2.2 - Основні параметри генератора рівня

Параметр	*Позначення*	*Опис*	*Типове значення*
Ширина карти	mapWidth	Гранична ширина робочої області для генерації кімнат	80 клітин
Висота карти	mapHeight	Гранична висота робочої області	60 клітин
Мінімальний розмір кімнати	roomMin	Найменша допустима ширина або висота кімнати	6 клітин
Максимальний розмір кімнати	roomMax	Найбільша допустима ширина або висота кімнати	14 клітин
Кількість кімнат	roomsCount	Цільова кількість кімнат у сесії	8-12
Базова складність	Dbase	Початковий коефіцієнт складності рівня	1.0
Крок приросту складності	k	Приріст складності на кожну віддалену зону	0.15

Подібна параметризація дозволяє налаштовувати гру без зміни вихідного коду. Значення roomMin і roomMax використовуються при побудові допустимих прямокутників кімнат, а коефіцієнти Dbase та k впливають на інтенсивність розміщення ворогів і ресурсів. Зміна цих параметрів може бути винесена до зовнішніх ассетів і використовуватись для балансу складності.

Окремим архітектурним принципом у проєкті є відокремлення даних конфігурації від коду, який реалізує поведінку. Такий підхід особливо важливий для ігор, у яких баланс складності, правила спавну та візуальні параметри часто змінюються в процесі тестування. Якщо параметри розміщення кімнат, коефіцієнти складності та характеристики ворогів

зберігаються у ScriptableObject або споріднених конфігураційних сутностях, розробник може змінювати поведінку системи без переписування центральної логіки генератора.

З практичної точки зору архітектура гри має підтримувати кілька типових сценаріїв розширення. По-перше, це додавання нових типів кімнат, зокрема стартових, бойових, ресурсних або подієвих. По-друге, це підключення нових типів ворогів із власними параметрами руху, здоров’я та типом атаки. По-третє, це розширення підсистеми інтерфейсу, наприклад додавання екрана паузи, статистики проходження або налаштувань. Запропонований поділ на конфігураційний шар, ігрове ядро та UI-шар дозволяє реалізовувати такі зміни локально, без масштабного перепроєктування системи.

Ще одним важливим аспектом є організація обміну подіями між менеджерами. У добре спроєктованій системі GameStateManager не повинен напряму керувати вмістом кожного елемента HUD, так само як UIController не має містити правил бойової логіки. Натомість зміни стану, наприклад отримання ушкодження, завершення сесії або старт нового рівня, поширюються як події, на які підписуються відповідні модулі. Це спрощує налагодження, зменшує зв’язність між класами та знижує ймовірність помилок ініціалізації.

Архітектурна схема також враховує потребу в підтримуваності коду в довшій перспективі. Для кваліфікаційної роботи це означає, що будь-яка ключова логіка має бути не лише реалізована, а й описана таким чином, щоб сторонній розробник або член комісії міг простежити межі відповідальності кожного компонента. Саме тому поряд із діаграмою класів доцільно використовувати компонентну діаграму та C4-модель: вони показують одну й ту саму систему з різним рівнем узагальнення і зменшують ризик подвійного трактування архітектурних рішень.

На рівні сценової організації доцільно виділяти окремі вузли для bootstrap-логіки, генератора рівня, менеджера станів гри, контролера гравця

та контейнера UI. Така структура полегшує навігацію в проєкті Unity, оскільки дозволяє швидко знайти точку входу та візуально відокремити службові об’єкти від ігрових сутностей. Крім того, при повторному запуску сесії система може очищати лише тимчасові ігрові об’єкти, не зачіпаючи конфігураційні сервіси та контролери, що існують протягом життєвого циклу сцени.

Таким чином, архітектурне проєктування у межах «Procedural Depth» орієнтоване не тільки на демонстрацію UML-моделей, а й на практичну придатність до реалізації. Кожен компонент має зрозумілий контракт, конфігураційні дані відокремлені від поведінкової логіки, а розподіл обов’язків між менеджерами забезпечує можливість локального доопрацювання окремих підсистем без порушення стабільності всього продукту.

2.3 Опис алгоритмів та математичних моделей

Ключовим алгоритмом у проєкті є процедурна генерація рівня типу «кімнати + коридори». Для розв’язання задачі обрано комбінований підхід: спочатку формується набір допустимих кімнат у межах карти, далі кімнати впорядковуються за просторовими правилами, а потім поєднуються коридорами так, щоб граф рівня залишався зв’язаним. Такий підхід добре узгоджується з вимогами roguelike-ігор і може бути реалізований поверх Tilemap у Unity [7].

На відміну від повністю хаотичного розміщення блоків, комбінований підхід дозволяє контролювати прохідність карти. Кожна кімната перевіряється на допустимість розмірів і відсутність критичних перетинів із уже розміщеними кімнатами. Після цього між кімнатами формується мінімальний набір коридорів, а за потреби додаються альтернативні маршрути, що підвищують варіативність дослідження.

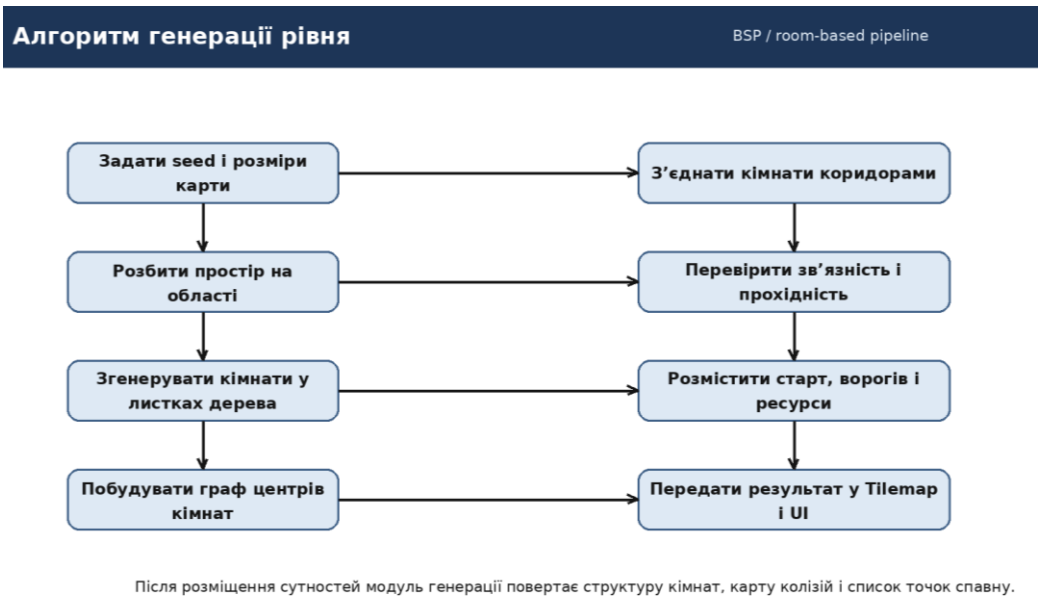

Рисунок 2.7 - Узагальнена схема алгоритму процедурної генерації рівня

Джерело: розроблено автором

Узагальнений алгоритм складається з шести етапів: ініціалізація seed; побудова набору кандидатів кімнат; валідація геометрії та відсікання конфліктів; з'єднання кімнат коридорами; розміщення ворогів, ресурсів і точок взаємодії; фінальна перевірка прохідності. Якщо перевірка не проходить, генерація повторюється з іншим seed або скоригованими параметрами.

Для оцінювання складності зони використовується проста модель лінійного приросту. Нехай zoneIndex - порядковий номер зони від стартової кімнати. Тоді коефіцієнт складності D обчислюється як D = Dbase + k * zoneIndex. Отримане значення використовується для вибору типу ворогів, їх кількості та ймовірності появи додаткових ресурсів. Така модель є достатньо простою, але забезпечує прогнозоване зростання навантаження на гравця.

Для вибору позицій спавну застосовується евристика, що враховує тип кімнати, вільний простір та мінімальну відстань до стартової точки входу. Якщо позначити відстань від об'єкта до старту через dist, то вагу позиції можна визначити як W = alpha * dist + beta * safetyScore. При цьому коефіцієнт safetyScore відображає, наскільки безпечною є стартова позиція

гравця з урахуванням оточення. У практичній реалізації замість складної оптимізації використовується ранжування кількох кандидатних точок.

Ще одним важливим елементом є модель колізій та взаємодії з середовищем. У Unity для 2D-проєктів доцільно використовувати Rigidbody 2D та Collider 2D, які забезпечують коректну фізичну взаємодію персонажа, ворогів і статичних перешкод [9]. Це дає змогу відокремити логіку переміщення від логіки перевірки зіткнень і зменшує кількість ручних перевірок координат.

Отже, алгоритмічна частина проєкту спирається не на одну процедуру, а на зв’язану систему правил: геометричне формування карти, зв’язування кімнат, контроль складності, вибір місць спавну та відображення стану гри. Саме така комбінація створює основу для якісного і відтворюваного геймплею, який зберігає новизну під час повторних проходжень.

Для формалізації алгоритмічної частини доцільно подати узагальнений псевдокод побудови рівня. Такий опис не дублює реалізацію мовою C#, а показує послідовність кроків у нейтральній формі, придатній для аналізу й обговорення на захисті.

Фрагмент лістингу 2.1 - Узагальнений псевдокод алгоритму генерації рівня

```
GenerateLevel(config, seed):
    rng <- InitRandom(seed)
    rooms <- []
    while rooms.Count < config.roomsCount:
        candidate <- CreateRoomCandidate(rng, config.roomMin,
config.roomMax)
        if IsInsideBounds(candidate, config.mapWidth,
config.mapHeight)
           and NotCriticalOverlap(candidate, rooms):
            rooms.Add(candidate)

    graph <- BuildNeighbourGraph(rooms)
    corridors <- ConnectRooms(graph)
    if not IsConnected(graph):
        return GenerateLevel(config, seed + 1)
```

```
    objects <- SpawnGameplayObjects(rooms, corridors, config)
    return LevelData(rooms, corridors, objects, seed)
```

Наведений псевдокод показує, що критичними точками алгоритму є перевірка геометричних обмежень, побудова графа суміжності та фінальна валідація зв'язності. Якщо будь-який із цих етапів завершується невдало, система не переходить до бойової сесії, а повторює генерацію з новим значенням seed. Такий підхід забезпечує стійкість алгоритму до випадків, коли набір кімнат-кандидатів виявляється структурно невдалим.

Після формування графа суміжності доречно виконувати окрему перевірку досяжності всіх кімнат зі стартової точки. У найпростішому випадку для цього можна використати пошук у ширину. Завдяки такій перевірці система гарантує, що жодна з кімнат не залишиться ізольованою і що гравець матиме змогу пройти рівень без штучно створених тупиків, які блокують прогрес.

Фрагмент лістингу 2.2 - Перевірка зв'язності графа рівня

```
CheckConnectivity(graph, startRoom):
    visited <- Set()
    queue <- Queue()
    queue.Enqueue(startRoom)
    visited.Add(startRoom)

    while queue is not empty:
        room <- queue.Dequeue()
        for each nextRoom in graph[room]:
            if nextRoom not in visited:
                visited.Add(nextRoom)
                queue.Enqueue(nextRoom)

    return visited.Count == graph.RoomsCount
```

З алгоритмічної точки зору найбільш витратною частиною побудови карти є перевірка конфліктів між кімнатами-кандидатами. Якщо виконувати

її прямим порівнянням кожної нової кімнати з усіма раніше створеними, складність у гіршому випадку наближається до $O(n^2)$, де n - кількість кімнат. Водночас перевірка зв’язності графа за допомогою пошуку у ширину має складність O(n + e), де e - кількість коридорів або ребер графа суміжності. Для цільових параметрів гри така складність є прийнятною і не створює помітних затримок під час старту сесії.

Окремої уваги заслуговує питання відтворюваності результатів генерації. Використання seed дає змогу повторно відтворити карту з однаковим набором кімнат і коридорів, що корисно як для налагодження, так і для тестування дефектів. Якщо під час ручної або автоматизованої перевірки виявлено некоректне розміщення ворогів чи ресурсів, конкретне значення seed можна зафіксувати та повторно використати під час пошуку причини помилки.

У підсумку алгоритмічна частина проєкту поєднує геометричну генерацію карти, перевірку топологічної зв’язності, модель керованого зростання складності та правила розміщення ігрових об’єктів. Саме таке поєднання є достатнім для створення стабільного roguelike-прототипу, у якому випадковість не руйнує логіку ігрового простору, а працює на підвищення реіграбельності.

Висновки до розділу 2

У другому розділі виконано проєктування програмного продукту «Procedural Depth». Побудовано поведінкові моделі, що відображають основні сценарії взаємодії гравця із системою, а також деталізують внутрішню послідовність ініціалізації та проходження ігрової сесії.

Описано архітектуру програмного продукту з поділом на рівні введення, логіки, генерації, UI та конфігураційних даних. Побудовано діаграми класів, компонентів і C4-модель, що дозволяють узгодити структуру майбутньої реалізації з методичними вимогами.

Також обґрунтовано алгоритмічні рішення: схему процедурної генерації рівня, модель зростання складності та правила розміщення ворогів і ресурсів. Отримані результати створюють достатню основу для переходу до етапу практичної реалізації та тестування гри.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Реалізація та конструювання програмного продукту

Програмний продукт реалізується у середовищі Unity з використанням мови програмування C#. Для побудови двовимірних рівнів доцільно використовувати Tilemap, оскільки цей інструмент забезпечує зручне розміщення плиток, інтеграцію з 2D-фізикою та ефективне оновлення сцени [7]. Для параметризації ворогів, кімнат та інших конфігурацій застосовуються ScriptableObject-ассети [8], а для реалізації фізичних взаємодій - Rigidbody 2D і Collider 2D [9].

Інтерфейс користувача реалізується на базі Canvas та стандартних UI-компонентів Unity [10]. Це дозволяє відображати показники здоров’я, очок, поточного рівня та службові повідомлення поверх сцени без дублювання логіки рендерингу. Для зберігання тимчасових колекцій кімнат, точок спавну та активних об’єктів використовуються стандартні узагальнені колекції C#, зокрема Dictionary<TKey, TValue> та List<T> [12].

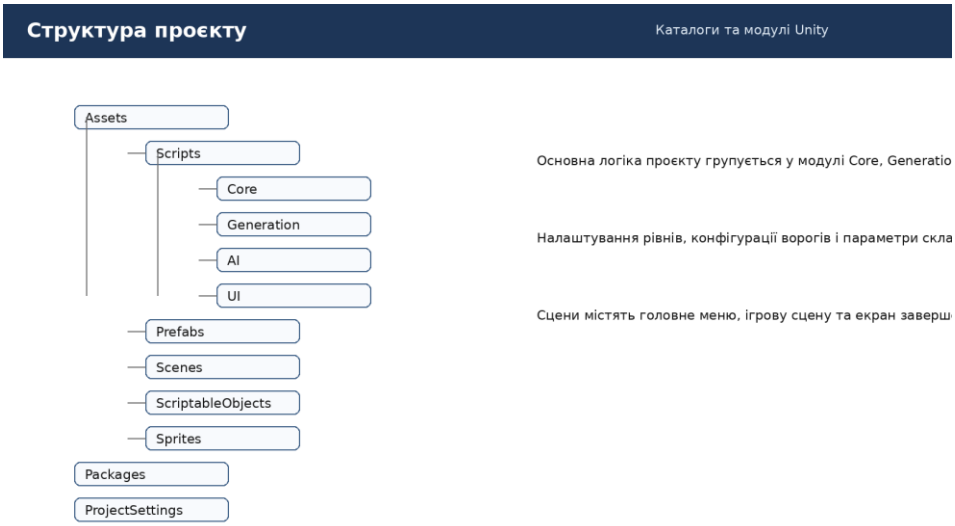

Рисунок 3.1 - Узагальнена структура директорій та основних скриптів проєкту

Джерело: розроблено автором

На практиці код організовано за принципом тематичних модулів. Окремо розміщуються директорії Core, Generation, Player, Enemies, UI та Configs. Це дозволяє логічно відокремити центральні менеджери, логіку генератора, класи керування персонажем, поведінку ворогів, елементи інтерфейсу та конфігураційні дані.

У межах сцени запуску об’єкт GameBootstrap створює або отримує посилання на необхідні менеджери, після чого ініціює побудову нового рівня. Після завершення генерації створюються об’єкти коридорів, ворогів і сервісних елементів, а потім запускається основний цикл оновлення стану гри. Узагальнений приклад такого виклику наведено у фрагменті лістингу 3.1.

Фрагмент лістингу 3.1 - Узагальнений виклик генерації та запуску ігрової сесії

```
public void StartSession(int seed)
{
    var context = configStorage.Load(seed);
    var level = levelGenerator.Build(context);
    tilemapRenderer.Draw(level);
    enemySpawner.Spawn(level, context.Difficulty);
    gameStateManager.EnterPlayingState(level);
    uiController.Refresh(gameStateManager.CurrentState);
}
```

У наведеному фрагменті видно, що ініціалізація сесії виконується послідовно: завантаження конфігурацій, побудова рівня, рендер структури Tilemap, спавн ворогів та перехід до активного стану гри. Така послідовність прямо відповідає послідовнісній діаграмі, побудованій у другому розділі.

Окремо реалізовано механізм вибору набору ворогів залежно від зони складності. Після отримання коефіцієнта D система визначає допустимий пул типів ворогів, кількість об’єктів у кімнаті та місця спавну. Це дозволяє уникати ситуацій, коли на старті гравець одразу потрапляє в надмірно складну бойову конфігурацію.

Фрагмент лістингу 3.2 - Визначення типу ворога за коефіцієнтом складності

```
private EnemyType ResolveEnemyType(int zoneIndex)
{
    float difficulty = baseDifficulty + difficultyStep * zoneIndex;
    if (difficulty < 1.3f) return EnemyType.Melee;
    if (difficulty < 1.8f) return EnemyType.Mixed;
    return EnemyType.Ranged;
}
```

Оскільки автономна 2D-гра не потребує повноцінної реляційної бази даних, рівень зберігання у проєкті представлено набором ассетів конфігурації та локальними структурами серіалізації. До таких даних належать шаблони кімнат, параметри ворогів, коефіцієнти складності, допустимі діапазони розмірів карти та налаштування HUD. За потреби локальне збереження може доповнюватися JSON-файлом для рекорду або статистики проходжень.

На практичному рівні важливим є не тільки написання окремих скриптів, а й узгодження життєвого циклу об’єктів сцени. Під час запуску гри службові компоненти створюють контекст сесії, після чого формується карта, на неї наносяться плитки, створюються вороги, ініціалізується гравець і лише після цього активуються системи введення. Такий порядок запуску виключає ситуації, коли персонаж або HUD звертаються до ще нестворених об’єктів рівня.

Для підтримки читабельності проєкту доцільно дотримуватися єдиних правил іменування скриптів, префабів та ассетів конфігурації. Наприклад, менеджери сценового рівня розміщуються в модулі Core, логіка процедурної побудови карти - в модулі Generation, а всі серіалізовані набори параметрів - у Configs. Така структура особливо корисна під час тестування та подальшого розширення функціоналу, оскільки дозволяє швидко визначити місце, у якому слід шукати причину дефекту або реалізовувати нову можливість.

Окремим практичним аспектом є організація шарів зіткнень і правил фізичної взаємодії. Для стін, ворогів, гравця та збираних об’єктів доцільно використовувати різні physics layers, щоб мінімізувати кількість зайвих

перевірок і чітко контролювати, які саме об’єкти можуть взаємодіяти між собою. Це спрощує відлагодження бойової системи та зменшує ризик випадкових зіткнень із декоративними елементами сцени.

Фрагмент лістингу 3.3 - Реакція менеджера стану на завершення сесії

```
public void HandlePlayerDeath()
{
    currentState = GameState.GameOver;
    inputBlocker.SetActive(true);
    uiController.ShowGameOverScreen(scoreModel.CurrentScore);
    sessionCleaner.ReleaseRuntimeObjects();
}

public void RestartSession()
{
    sessionCleaner.ReleaseRuntimeObjects();
    StartSession(seedProvider.NextSeed());
```

У наведеному фрагменті показано, що менеджер станів виконує дві різні за змістом дії: фіксує факт завершення поточної спроби та готує умови для нового старту. Очищення тимчасових об’єктів винесено в окремий сервіс, що зменшує навантаження на GameStateManager і дає змогу повторно використовувати логіку очищення під час зміни рівня або ручного перезапуску.

Крім того, під час реалізації важливо враховувати питання повторного використання префабів. Якщо вороги, снаряди або ефекти створюються щоразу заново, у довших сесіях це може призводити до непотрібних алокацій пам’яті. Тому навіть у межах навчального проєкту доцільно проєктувати систему так, щоб у майбутньому її можна було доповнити простим механізмом object pooling без змін у зовнішньому контракті більшості модулів.

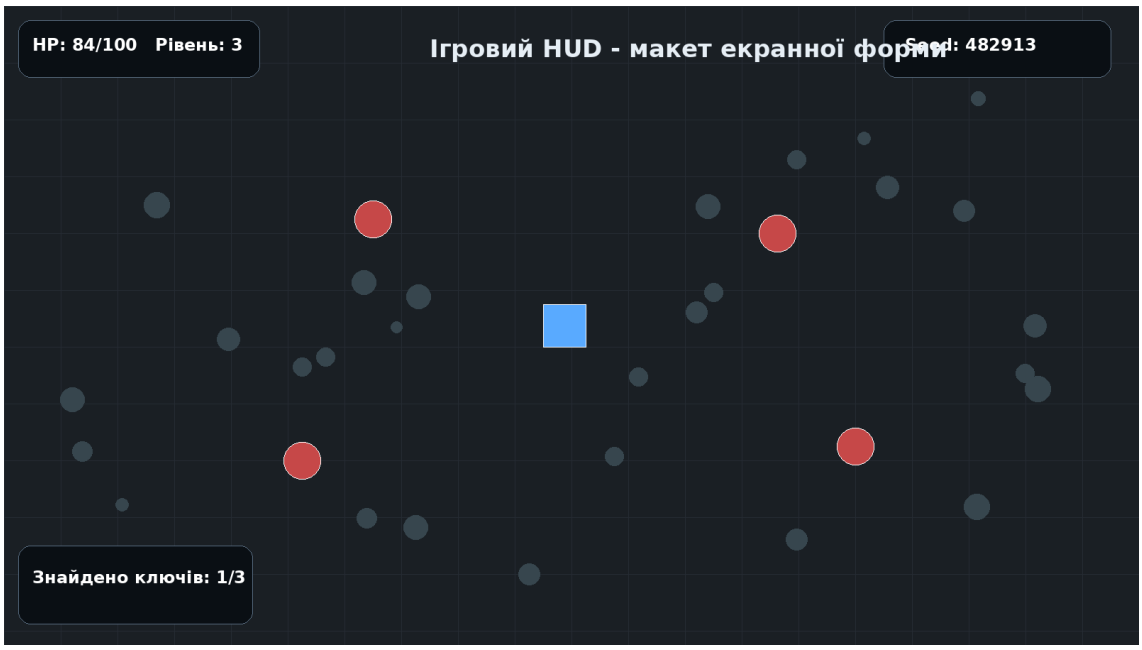

Рисунок 3.2 - Приклад HUD із показниками стану персонажа

Джерело: розроблено автором

HUD містить мінімально необхідні для гри елементи: показник здоров’я, лічильник очок, номер поточної сесії або рівня та службові повідомлення. Відокремлення інтерфейсу від ігрового ядра дозволяє змінювати спосіб відображення без впливу на бойову логіку або модуль генерації.

3.2 Тестування програмного продукту

Тестування програмного продукту виконується у двох напрямах: функціональному та модульному. Функціональне тестування перевіряє відповідність реалізованої поведінки заявленому функціоналу, а модульне тестування орієнтоване на перевірку окремих критичних компонентів. Для автоматизації модульних перевірок у середовищі Unity доцільно використовувати Unity Test Framework [11].

Під час тестування особливу увагу приділено генератору рівня, оскільки саме цей модуль формує основну цінність ігрового продукту. Крім того, окремо перевіряється система оновлення HUD, обробка завершення

сесії, логіка перезапуску та реакція на зіткнення персонажа з ворогами й перешкодами.

Таблиця 3.1 - Результати функціонального тестування програмного продукту

№	*Перевірка*	*Очікуваний результат*	*Фактичний результат*	*Статус*
1	Запуск гри	Відкривається стартове вікно і доступна кнопка початку сесії	Стартовий екран відображено коректно	Пройдено
2	Генерація нового рівня	Створюється зв’язна карта з кімнатами та коридорами	Карта згенерована, недоступних кімнат не виявлено	Пройдено
3	Рух персонажа	Персонаж реагує на введення без зависань	Керування стабільне	Пройдено
4	Колізії зі стінами	Персонаж не проходить крізь перешкоди	Зіткнення обробляються коректно	Пройдено
5	Взаємодія з ворогом	Зменшується запас здоров’я та оновлюється HUD	Стан здоров’я оновлюється вчасно	Пройдено
6	Завершення сесії	Після поразки відкривається екран завершення	Екран Game Over відображено	Пройдено
7	Перезапуск	Починається нова сесія з новим рівнем	Сесія перезапускається без артефактів	Пройдено
8	Відображення очок	Лічильник збільшується після досягнення події	Лічильник оновлюється коректно	Пройдено
9	Оновлення інтерфейсу	HUD синхронізовано зі станом GameStateManager	Розбіжностей не зафіксовано	Пройдено
10	Завантаження конфігів	Параметри кімнат і ворогів зчитуються із ассетів	Помилок десеріалізації не виявлено	Пройдено

Результати функціонального тестування показують, що базовий ігровий цикл працює коректна. Особливо важливо, що перезапуск не залишає за собою артефактів попередньої сесії, а HUD синхронізується зі станом гри без помітних затримок.

Таблиця 3.2 - Приклади модульного тестування критично важливих компонентів

Компонент	*Тестова умова*	*Критерій успішності*	*Результат*
RoomValidator	Перевірка допустимого перетину кімнат	Накладення кімнат понад поріг відхиляється	Успішно
GraphConnector	Перевірка зв’язності після побудови коридорів	Усі кімнати досяжні зі стартової	Успішно
DifficultyResolver	Перехід між зонами складності	Коефіцієнт D зростає лінійно	Успішно
EnemySpawner	Обмеження кількості ворогів у стартовій зоні	Ліміт стартового навантаження не перевищено	Успішно
GameStateManager	Перехід у стан завершення	Після нульового здоров’я активується Game Over	Успішно
UIController	Оновлення текстових полів і шкал	Значення збігаються зі станом моделі	Успішно

Модульні перевірки спрямовані насамперед на логіку, яка не залежить від графічного середовища. Такий підхід дозволяє швидко локалізувати дефекти в генераторі, правилах балансування та менеджері станів. Часткове покриття тестами критичних компонентів відповідає методичним вимогам до програмного продукту [11].

Окрім наведених у таблицях перевірок, методика тестування передбачає повторні серії запусків із різними значеннями seed та граничними параметрами генератора. Такі запуски дають змогу перевірити, чи не з’являються непрохідні конфігурації карти, накладання кімнат або некоректні

точки появи ворогів. Особливо важливими є сценарії з мінімальними та максимальними допустимими значеннями параметрів, оскільки саме в них найчастіше проявляються крайові дефекти.

Для функціонального тестування доцільно використовувати підхід, за якого кожний сценарій проходження пов’язується з конкретним очікуваним результатом. Наприклад, після запуску нової сесії гравець має з’явитися в стартовій кімнаті, HUD повинен містити початкові значення, а модуль ворогів не має активуватися раніше завершення генерації карти. Якщо система порушує хоча б одну з цих умов, відповідний сценарій не може вважатися успішно пройденим.

Під час перевірки перезапуску особливу увагу приділено очищенню тимчасових об’єктів. Для roguelike-гри це критично, оскільки залишки попередньої сесії, зокрема вороги, снаряди, події тригерів або значення службових лічильників, можуть спотворити результат нової спроби. Саме тому одним із критеріїв коректності є повна ізоляція кожної нової сесії від попередньої.

Модульне тестування, у свою чергу, доцільно концентрувати на тих частинах системи, де логіка не залежить від конкретної сцени та може бути перевірена окремо. До таких компонентів належать перевірка допустимості кімнат, побудова графа суміжності, розрахунок коефіцієнта складності, вибір типу ворога, а також синхронізація модельного стану з даними, що передаються в інтерфейс. Такий підхід скорочує час пошуку дефекту, оскільки дозволяє відразу локалізувати модуль, у якому порушено очікувану поведінку.

Важливою перевагою обраної схеми тестування є її придатність до подальшої автоматизації. Тести на коректність генерації, зв’язність графа та перехід менеджера станів можуть бути інтегровані у стандартний набір Unity Test Framework і виконуватися після кожної суттєвої зміни коду. Для кваліфікаційної роботи це демонструє не тільки працездатність прототипу, а й правильний підхід до забезпечення якості програмного продукту.

Отже, тестування в проєкті «Procedural Depth» розглядається як безперервний процес, що супроводжує і проєктування, і практичну реалізацію. Поєднання функціональних перевірок, модульних тестів і повторних запусків із різними параметрами генерації дозволяє вчасно виявляти як логічні, так і інтеграційні помилки, що безпосередньо впливають на стабільність ігрової сесії.

3.3 Використання програмного продукту

Використання програмного продукту не потребує спеціальної підготовки користувача. Після запуску застосунку відкривається стартовий екран, з якого гравець переходить до нової сесії. Під час старту автоматично формується структура рівня, створюються кімнати, коридори, точки появи ворогів і елементи інтерфейсу.

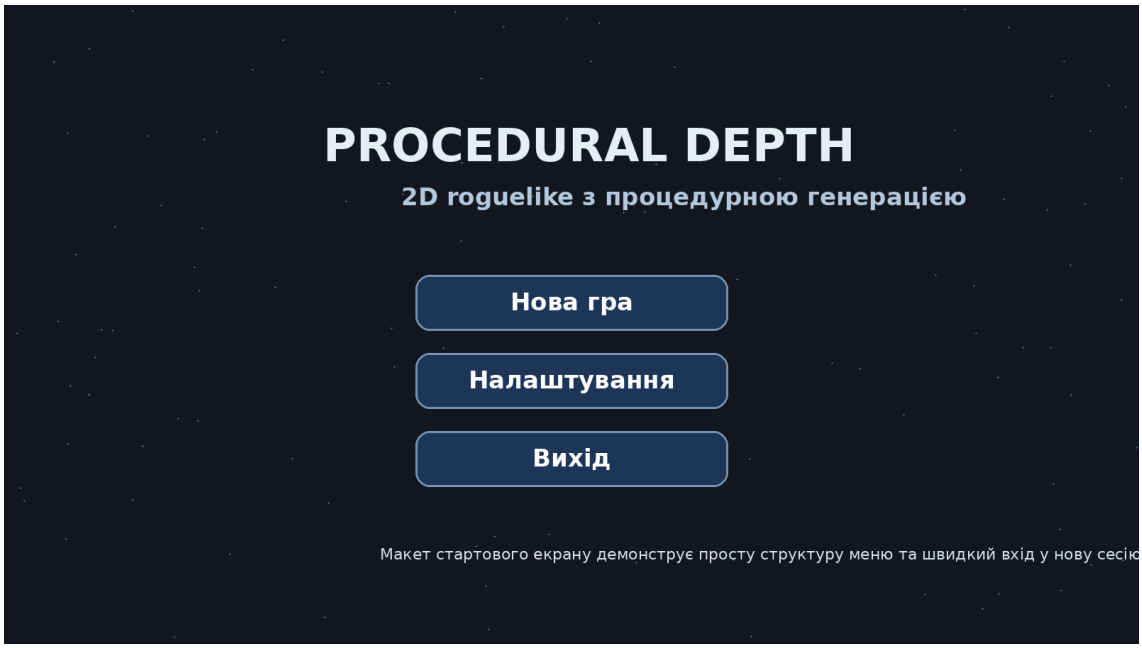

Рисунок 3.3 - Приклад стартового екрана гри

Джерело: розроблено автором

Після натискання кнопки початку гри користувач потрапляє до першої кімнати згенерованого рівня. Подальше проходження здійснюється за допомогою клавіатури. На екрані постійно відображаються показники

здоров’я, очок і поточного прогресу, що дозволяє гравцеві швидко оцінювати стан сесії.

Таблиця 3.3 - Основні команди користувача під час гри

Клавіша	*Призначення*	*Коментар*
W / A / S / D	Переміщення персонажа	Рух по кімнатах і коридорах
Space	Базова взаємодія / дія	Може використовуватися для підтвердження або дії залежно від сцени
Esc	Пауза або повернення до меню	Виклик службового меню
R	Перезапуск сесії	Швидкий старт нової спроби після завершення

У межах гри передбачено зрозумілий і короткий набір команд. Це відповідає вимозі швидкого входження в ігрову сесію та зменшує навантаження на користувача на початковому етапі. За необхідності таблиця команд може бути доповнена у вікні налаштувань або окремому розділі довідки.

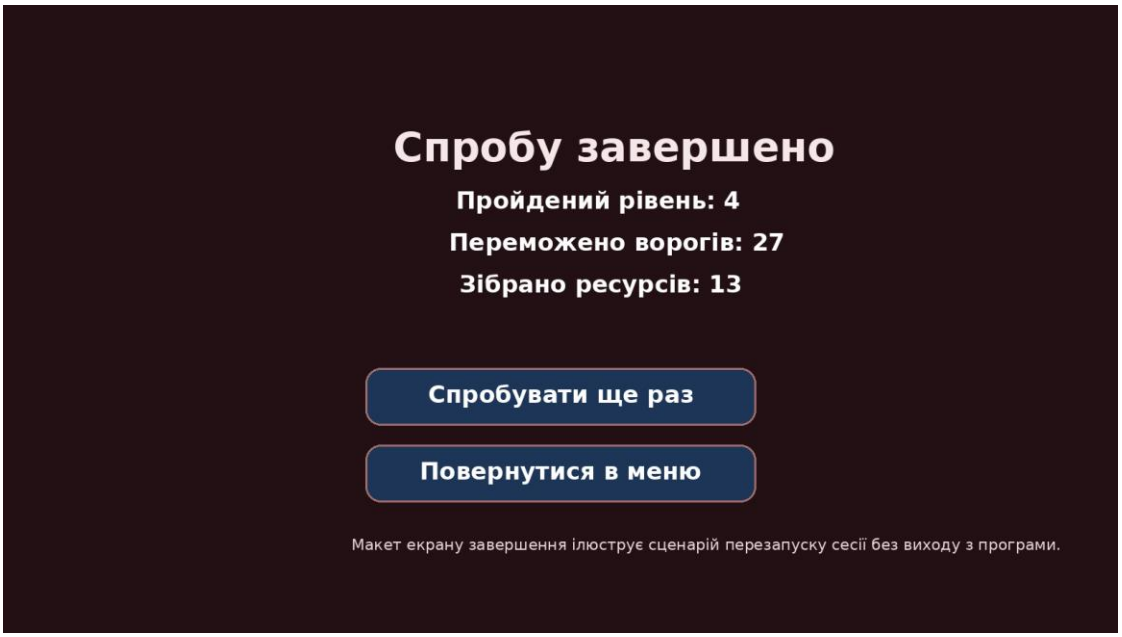

Рисунок 3.4 - Приклад екрана завершення сесії

Джерело: розроблено автором

Після завершення спроби користувач переходить на екран завершення, де може переглянути підсумкові показники та запустити нову сесію. Такий підхід підтримує циклічну природу roguelike-гри, де повторне проходження є важливою частиною загального досвіду користувача.

Отже, програмний продукт орієнтований на короткий сценарій взаємодії: запуск, автоматична генерація рівня, проходження з використанням базових елементів керування, завершення сесії та нова спроба. Простота сценарію використання робить гру придатною для подальшого розширення без зміни базових принципів взаємодії.

З практичної точки зору користувацький сценарій можна розширити короткою інструкцією з підготовки до запуску. Для роботи прототипу необхідно мати персональний комп’ютер з операційною системою Windows, встановлене середовище виконання, сумісне зі збіркою Unity, а також базові пристрої введення - клавіатуру і мишу або лише клавіатуру, якщо конкретна сцена не потребує наведення курсора. Після встановлення збірки користувач запускає виконуваний файл і переходить до стартового меню.

Під час першого запуску особливу роль відіграє зрозумілість інтерфейсу. Користувач має одразу бачити, як розпочати нову сесію, де переглянути поточний стан гри та яким чином повернутися до меню або повторити спробу після поразки. Саме тому в проєкті використовується мінімалістичний набір екранів і короткий набір команд, який не перевантажує гравця службовою інформацією на ранньому етапі взаємодії.

Для розробника або експерта, який перевіряє роботу, важливою є також можливість швидко відтворити окремі сценарії. Якщо в проєкті передбачено виведення seed, це дозволяє повторно завантажити конкретну конфігурацію рівня і перевірити поведінку ворогів, коректність колізій або роботу інтерфейсу саме в тому стані, у якому було зафіксовано дефект. Така практика корисна не тільки під час налагодження, а й у процесі демонстрації роботи комісії.

У подальшому програмний продукт може бути розширений системою налаштувань складності, збереженням рекордів, додатковими типами кімнат і більш розвиненим екраном статистики. Водночас базовий сценарій використання вже на поточному етапі є завершеним: користувач запускає гру, проходить автоматично згенерований рівень, отримує зворотний зв’язок через HUD, завершує сесію та за потреби переходить до нової спроби без повторної підготовки середовища.

Висновки до розділу 3

У третьому розділі описано практичну реалізацію програмного продукту «Procedural Depth» у середовищі Unity із застосуванням мови програмування C#, Tilemap, 2D-фізики, ScriptableObject та стандартних UI-компонентів.

Наведено приклади фрагментів лістингу, що ілюструють запуск ігрової сесії та правила визначення типів ворогів залежно від зони складності. Також описано, як у межах проєкту організовано шар конфігураційних даних і логіку оновлення інтерфейсу користувача.

Окремо представлено результати функціонального та модульного тестування, а також коротку інструкцію користувача. Це дозволяє вважати програмний продукт достатньо підготовленим до демонстрації, подальшого вдосконалення та використання в межах кваліфікаційної роботи.

ВИСНОВКИ

У кваліфікаційній роботі розглянуто задачу розробки комп’ютерної гри «Procedural Depth» із використанням алгоритмів генерації рівнів локацій. У вступі обґрунтовано актуальність теми, визначено мету, завдання, об’єкт і предмет дослідження, а також окреслено практичне значення результатів. У першому розділі досліджено предметну область, проаналізовано щонайменше три ігри-аналоги та сформульовано вимоги до програмного продукту.

У другому розділі виконано проєктування програмного продукту. Побудовано діаграму варіантів використання, діаграму діяльності, діаграму послідовності, діаграму класів, діаграму компонентів та C4-модель. Описано багаторівневу архітектуру застосунку, параметри генератора рівня, модель зростання складності та правила розміщення ворогів і ресурсів. Це дозволило формалізувати структуру системи до початку програмної реалізації.

У третьому розділі представлено особливості реалізації програмного продукту в середовищі Unity, наведено фрагменти лістингу, описано використання Tilemap, 2D-фізики, ScriptableObject, Canvas та стандартних засобів мови C#. Також виконано функціональне та модульне тестування критично важливих компонентів, сформовано коротку інструкцію користувача та узагальнено сценарій використання гри.

Поставлену в роботі мету - розробку програмного забезпечення 2D roguelike-гри з процедурною генерацією локацій - досягнуто на рівні проєктування та реалізації основного функціоналу. Усі основні завдання дослідження виконано: предметну область проаналізовано, архітектуру спроєктовано, алгоритмічні рішення описано, а програмний продукт представлено та перевірено. Посилання на GitHub-репозиторій проєкту: [вставити актуальне посилання автора перед поданням остаточної версії]. На момент підготовки цієї редакції окремі тези або статті з апробацією результатів дослідження не зазначалися.

Перспективами подальшого розвитку проєкту є ускладнення бойової системи, додавання нових типів кімнат і подій, розширення системи прогресії персонажа, а також поглиблення автоматизації тестування критично важливих модулів. Це дає змогу розглядати «Procedural Depth» не лише як навчальний прототип, а і як основу для подальшого ітеративного розвитку з більш повним контентним наповненням.

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАННЯ

1. Лайтарук І., Гришанович Т. Огляд перспектив швидкодії алгоритмів процедурної генерації в комп'ютерних іграх. Міжнародна науково-практична конференція «Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем». 2025. С. 195–198.

2. Karth I., Smith A. M. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. IEEE Transactions on Games. 2022. Vol. 14, No. 3. P. 364–376. DOI: 10.1109/TG.2021.3076368.

3. Shaker N., Togelius J., Nelson M. Procedural Content Generation in Games. Springer, 2016.

4. Push Square. The Binding of Isaac: Rebirth. Доступ: 05.03.2026.

5. Push Square. Dead Cells. Доступ: 05.03.2026.

6. Supergiant Games. Hades. Доступ: 05.03.2026.

7. Unity Technologies. Tilemaps. Unity Manual. URL: https://docs.unity3d.com/6000.4/Documentation/Manual/tilemaps/tilemaps-landing.html (дата звернення: 18.04.2026).

8. Unity Technologies. ScriptableObject. Unity Manual. URL: https://docs.unity3d.com/6000.4/Documentation/Manual/class-ScriptableObject.html (дата звернення: 18.04.2026).

9. Unity Technologies. Introduction to Rigidbody 2D. Unity Manual. URL: https://docs.unity3d.com/6000.3/Documentation/Manual/2d-physics/rigidbody/introduction-to-rigidbody-2d.html

10. Unity Technologies. Canvas. Unity UI Manual. URL: https://docs.unity3d.com/Packages/com.unity.ugui%402.6/manual/UICanvas.html

11. Unity Technologies. Unity Test Framework overview. URL: https://docs.unity3d.com/Packages/com.unity.test-framework%402.0/manual/index.html (дата звернення: 18.04.2026).

12. Microsoft. Dictionary<TKey,TValue> Class. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2 (дата звернення: 18.04.2026).