

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка 3D гри у жанрі roguelike з елементами RPG та
процедурною генерацією карти»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

(підпис)

Назар БАРЧЕНКО

Виконав: здобувач вищої освіти групи ПД-41

Назар БАРЧЕНКО

Керівник: _____
канд.техн.наук В'ячеслав ТРЕЙТЯК

Рецензент: _____

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Барченку Назару Юрійовичу

1. Тема кваліфікаційної роботи: «Розробка 3D гри у жанрі roguelike з елементами RPG та процедурною генерацією карти»

керівник кваліфікаційної роботи канд.техн.наук В'ячеслав ТРЕЙТЯК,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: офіційна документація ігрового рушія Unity та Universal Render Pipeline, документація мови програмування C# і середовища .NET, наукові джерела з процедурної генерації ігрового контенту (Perlin noise, Simplex noise, Wave Function Collapse, морфологічні операції), монографії з штучного інтелекту в іграх та Utility AI, опис ігрових механік жанру roguelike.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної галузі та існуючих рішень.
2. Проєктування 3D гри у жанрі roguelike.
3. Програмна реалізація гри.
4. Тестування гри.

5. Перелік графічного матеріалу: презентація

1. Вимоги до програмного забезпечення.
2. Програмні засоби реалізації.
3. Архітектура системи.
4. Процедурна генерація рівнів.
5. Стани та штучний інтелект.
6. Екранні форми.
7. Результати тестування.
8. Апробація результатів дослідження

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд алгоритмів процедурної генерації та технологій реалізації	08.03-15.03.2026	
4	Проєктування 3D гри у жанрі roguelike	16.03-31.03.2026	
5	Програмна реалізація гри	01.04-25.04.2026	
6	Тестування гри	26.04-04.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	05.05-10.05.2026	
8	Розробка демонстраційних матеріалів	11.05-15.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____

Назар БАРЧЕНКО

Керівник

кваліфікаційної роботи _____

В'ячеслав ТРЕЙТЯК

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 51 стор., 6 табл., 19 рис., 24 джерел.

Мета роботи - покращення ігрового досвіду в 3D-грах жанру roguelike з елементами RPG, процедурною генерацією воксельних рівнів, системою прогресії персонажа та поведінкою ворогів засобами Unity і C#.

Об'єкт дослідження - процес виживання, проходження рівнів та генерації ігрового контенту в жанрі roguelike з процедурною генерацією ігрового контенту.

Предмет дослідження - методи, алгоритми, програмні засоби та архітектурні рішення для реалізації процедурної генерації воксельних рівнів, системи станів гравця, RPG-прогресії персонажа та поведінки ворогів у 3D-грі жанру roguelike.

Короткий зміст роботи: В роботі проаналізовано жанр roguelike, його історичний розвиток від класичних текстових ігор до сучасних 3D-проектів, а також ключові ігри-аналоги (Hades [19], Dead Cells [20], Risk of Rain 2 [21]) і сучасні алгоритми процедурної генерації карт (Perlin/Simplex noise, BSP, клітинні автомати, Wave Function Collapse, морфологічні операції). Виконано порівняльний аналіз алгоритмів за критеріями детермінованості, продуктивності, керованості результату та простоти інтеграції, що дозволило обґрунтувати вибір fBm-шуму як основи генерації.

Спроектовано 9-шаровий пайплайн генерації воксельних арен на основі fBm-шуму з власною seedable реалізацією, що забезпечує повну відтворюваність результату за seed-значенням. Розроблено архітектуру на патернах Service Locator і EventBus для слабкого зв'язку між підсистемами, FSM гравця з 8 станів, Utility AI ворогів з 17 тактиками поведінки та SquadDirector для координації загонів і запобігання надмірному скупченню агентів на полі бою.

Реалізовано RPG-підсистеми: формулу статів Flat/Percent/Multiplier з lazy recalculation для оптимізації обчислень при частих змінах характеристик, дерево

навичок як граф з BFS-перевіркою зв'язності розблокованих вузлів, а також систему модифікаторів від екіпіровки і тимчасових бафів. Розроблено кастомні HLSL-шейдери VoxelFlatLit (flat-lit рендер вокселів через vertex color без використання текстур) і InvisibleWallRipple (процедурна стіна арени з hex grid, ascending pulses, fresnel-ефектом і proximity reveal при наближенні гравця).

Проведено функціональне та продуктивнісне тестування, що підтвердило виконання всіх вимог: час генерації карти $128 \times 128 \times 32$ - до 1.12 секунди при нормі 2 секунди, FPS при 20 одночасно активних ворогах - 61 при нормі 60, детермінованість генерації підтверджена для всіх п'яти байомів. Окремо перевірено стабільність роботи на тривалих ігрових сесіях, коректність збереження прогресу гравця між запусками та відсутність витоків пам'яті при повторній генерації арен. Виконано стрес-тест із збільшеною кількістю ворогів для визначення верхньої межі продуктивності системи.

Сферою використання застосунку є ринок інді-ігор у жанрі roguelike, а реалізована пайплайн-архітектура процедурної генерації та Utility AI-підсистема можуть бути адаптовані для інших 3D ігрових проєктів на платформі Unity.

Окремі компоненти - генератор шуму fBm, EventBus і Service Locator - можуть використовуватись як самостійні модулі у незалежних проєктах, а підхід до проєктування шарового пайплайну застосовний у задачах генерації ландшафтів і рівнів довільної складності. Розроблені шейдери і система Utility AI також придатні для повторного використання у проєктах із подібними візуальними та поведінковими вимогами.

КЛЮЧОВІ СЛОВА: ROGUELIKE, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, ВОКСЕЛІ, UTILITY AI, PIPELINE ARCHITECTURE, UNITY, C#, FBM, STATE MACHINE, SERVICE LOCATOR.

ЗМІСТ

ВСТУП.....	9
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ.....	12
1.1 Аналіз жанру roguelike та його ключових особливостей.....	12
1.2 Аналіз існуючих ігор-аналогів.....	13
1.3 Огляд алгоритмів процедурної генерації карт.....	16
1.4 Вибір технологій.....	19
2 ПРОЄКТУВАННЯ 3D ГРИ У ЖАНРІ ROGUELIKE.....	20
2.1 Концепція та ігровий цикл.....	20
2.2 Вимоги до програмного забезпечення.....	21
2.3 Архітектура застосунку.....	23
2.4 Проєктування алгоритму процедурної генерації карти.....	25
2.5 Проєктування систем гравця та ворогів.....	30
2.6 Проєктування RPG-підсистем.....	33
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ГРИ.....	36
3.1 Середовище розробки та структура проєкту.....	36
3.2 Реалізація процедурної генерації карти.....	37
3.3 Реалізація системи ворогів та Utility AI.....	39
3.4 Реалізація RPG-підсистем.....	41
3.5 Реалізація візуальних ефектів.....	42
4 ТЕСТУВАННЯ.....	45
4.1 Методи тестування.....	45
4.2 Функціональне тестування.....	45
4.3 Тестування продуктивності.....	47
ПЕРЕЛІК ПОСИЛАНЬ.....	51
ДОДАТОК А.....	53
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ.....	61

ВСТУП

Ринок відеоігор [23] - одна з найбільших розважальних індустрій у світі: за даними Statista, у 2023 році його обсяг перевищив 180 млрд доларів США, а кількість гравців досягла 3,2 млрд осіб. Жанр roguelike вирізняється процедурно згенерованими рівнями, механікою permadeath та системою прогресії, що забезпечують високу реіграбельність і тривалий інтерес гравця до гри. Hades (2020, Supergiant Games) і Dead Cells (2018, Motion Twin) підтвердили комерційний потенціал жанру, отримавши нагороди провідних ігрових видань і мільйони гравців по всьому світу.

Процедурна генерація ігрових карт - один із ключових елементів roguelike [15, 16], що дозволяє автоматично формувати унікальні рівні без ручного моделювання. У 3D-іграх така генерація ускладнюється потребою зберігати прохідність карти, контроль форми арени та стабільну продуктивність під час виконання. Реалізація такого підходу потребує алгоритмів шуму, морфологічних операцій, просторового розподілу зон і продуманої архітектури програмного забезпечення, яка забезпечує гнучкість, розширюваність та продуктивність у реальному часі.

Актуальність роботи обумовлена тим, що розробка ігрового застосунку охоплює широкий спектр технічних завдань: від генерації контенту до штучного інтелекту ворогів і RPG-механік прогресії. У межах одного проєкту необхідно узгодити роботу генератора рівнів, бойової системи, системи станів гравця, прогресії персонажа та поведінки ворогів. Реалізація цих задач у єдиному проєкті на базі Unity та C# дає змогу відпрацювати підходи сучасної ігрової індустрії та повний цикл створення ігрового продукту.

Мета роботи - покращення ігрового досвіду в 3D-грах жанру roguelike з елементами RPG, процедурною генерацією воксельних рівнів, системою прогресії персонажа та поведінкою ворогів засобами Unity і C#.

Для досягнення мети визначено такі завдання:

1. Проаналізувати жанр roguelike, його ключові механіки та наявні програмні аналоги;
2. Дослідити алгоритми процедурної генерації ігрових карт і обґрунтувати вибір підходу;
3. Спроекувати архітектуру гри та її основних підсистем;
4. Реалізувати систему процедурної воксельної генерації рівнів на основі пайплайну шарів;
5. Реалізувати ігрові механіки: бойову систему, систему статів і прогресії, Utility AI ворогів;
6. Провести тестування розробленого застосунку та оцінити його продуктивність.

Об'єкт дослідження - процес виживання, проходження рівнів та генерації ігрового контенту в жанрі roguelike з процедурною генерацією ігрового контенту.

Предмет дослідження - методи, алгоритми, програмні засоби та архітектурні рішення для реалізації процедурної генерації воксельних рівнів, системи станів гравця, RPG-прогресії персонажа та поведінки ворогів у 3D-грі жанру roguelike.

Методи дослідження: об'єктно-орієнтоване проектування, патерни проектування програмного забезпечення (State Machine, Service Locator, Observer, Factory), алгоритми процедурної генерації ігрових рівнів (шум Перліна [3], Simplex-шум, морфологічні операції), методи керування поведінкою ворогів на основі Utility AI, а також функціональне та продуктивнісне тестування для оцінювання коректності й ефективності реалізованого програмного забезпечення.

Практичне значення результатів полягає у тому, що розроблений застосунок реалізує пайплайн-архітектуру воксельної генерації рівнів та Utility AI для поведінки ворогів. Отримані результати можуть бути використані як основа для подальшого розширення гри, як практичний приклад при вивченні розробки ігрових застосунків на платформі Unity, а також як референс при проектуванні аналогічних систем у незалежних інді-проектах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Аналіз жанру roguelike та його ключових особливостей

Rogue вийшла у 1980 році. Майкл Той і Гленн Уічмен написали її для Unix [1], підземелля зображувалися ASCII-символами. Кожен запуск генерував нову карту. Загибель означала початок з нуля - жодних збережень, жодного відкату. Ці два рішення пережили чотири десятки років і досі лишаються основою жанру.

Permadeath на початку 1980-х виник не з дизайнерських міркувань. Зберігати стан гри на університетських Unix-машинах того часу було незручно. Але механіка прижилась і набула власного смислу. Без реального ризику смерті напруга в roguelike зникає.

У 2008 році учасники конференції International Roguelike Development Conference спробували формалізувати жанр через «Берлінську інтерпретацію» [2]. Перелік критеріїв довгий: процедурна генерація, permadeath, покроковість, тайлова карта, висока складність. Більшість ігор, які всі відносять до жанру, ці критерії порушують. Покроковість зникла ще у 2000-х і не повернулась. «Берлінська інтерпретація» зараз - привід для суперечок між фанатами, а не практичний інструмент.

Паралельно з roguelike з'явився термін roguelite - для ігор, що беруть лише частину жанрових механік. Ключова різниця: прогрес між забігами. Гравець гине, але зберігає щось - нову зброю, постійні покращення статів, відкриті персонажі. Dead Cells і Hades так і влаштовані [20, 19]. Класичне roguelike між забігами не зберігає нічого.

Гра, розроблена в цій роботі, займає проміжне місце. Забіг завершується зі смертю, але рівень досвіду і відкриті вузли дерева навичок зберігаються. Спорядження при загибелі скидається повністю - це ближче до класики.

Рольові механіки у roguelike з'явилися не випадково. Permadeath робить кожен забіг кінечним, а числові статистика дають відчуття зростання всередині цього відрізка.

Гравець знає, що помре - але поки живий, бачить, як персонаж стає сильнішим. Це утримує за грою довше за будь-яку сюжетну мотивацію.

Конкретні реалізації різняться. Hades дає тимчасові підсилення від богів-олімпійців [19]: кожне змінює конкретну здібність, і набір доступних варіантів різний кожного забігу. Dead Cells відмовився [20] від класичних рівнів і замінив їх blueprint-ами - постійними відкриттями нової зброї, і клітинами-мутаціями, що визначають стиль бою на поточний забіг.

У LetsRoll система статів побудована за моделлю Flat/Percent/Multiplier, запозиченою з Path of Exile. Фінальне значення будь-якого статусу:

$$\text{Final} = (\text{Base} + \Sigma\text{Flat}) \times (1 + \Sigma\text{Percent} / 100) \times \text{PMultiplier}$$

Base - базове значення, ΣFlat - сума плоских надбавок, $\Sigma\text{Percent}$ - сума процентних бонусів, PMultiplier - добуток мультиплікативних модифікаторів. Поділ на три типи небезпринципний: плоскі надбавки сильніші рано, процентні - пізно, коли базові значення вже великі. Дерево навичок реалізоване як граф вузлів; при знятті будь-якого вузла виконується BFS-перевірка зв'язності графа, що залишився.

1.2 Аналіз існуючих ігор-аналогів

Ізометричний 3D roguelite [19]. Гравець - Загрей, син бога підземного світу - намагається вибратися нагору. Гине, повертається, пробує знову. Технічно генерація рівнів тут часткова: кімнати беруться з фіксованого пулу, послідовність і наповнення між забігами змінюються, але самі кімнати розроблені вручну.

Це свідоме рішення. Hades - наративна гра з понад двадцятьма тисячами рядків озвученого діалогу. Повністю процедурний простір ускладнив би контроль над подачею сюжету. Прогрес між забігами - через «Дзеркало ночі»: постійні покращення за ігрову валюту, що залишається після смерті. У 2021 році гра отримала BAFTA Games Award як «Найкраща гра» і перевищила три мільйони продажів за перший рік після Early Access.



Рис. 1.1 Скріншот гри Hades (ізометрична арена, Supergiant Games, 2020)

Одночасно roguelite і метроїдванія [20] - два жанри, що рідко перетинаються. Рівні збираються з модульних секцій, а не генеруються повністю. Смерть скидає інвентар. Карта нелінійна, нові маршрути відкриваються поступово.

Motion Twin зробили одне нетривіальне рішення: розміщення ворогів і предметів детерміноване для конкретного seed-у рівня. Randomness є - але між забігами, а не під час проходження. Гравець не отримує «поганий розклад просто тому що рандом». Це зменшує відчуття несправедливості - і це добре продумано.



Рис. 1.2 Скріншот гри Dead Cells (2D процедурний рівень, Motion Twin, 2018)

Перша частина серії [21] - 2D. Друга стала повністю 3D і зберегла аудиторію - для жанру це рідкість. Рівні тут - фіксовані зони, а не процедурна геометрія, але наповнення кожного запуску різне. Складність зростає з часом: внутрішній таймер не дає стояти на місці. Понад сто артефактів з нетривіальними взаємодіями; деякі комбінації настільки ламають баланс, що спільнота перетворила це на окрему розвагу.



Рис. 1.3 Скріншот гри Risk of Rain 2 (3D арена, Noroo Games, 2019)

Зведений порівняльний аналіз розглянутих аналогів і розробленої гри LetsRoll наведено в таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз ігор-аналогів

Критерій	Hades	Dead Cells	Risk of Rain 2	LetsRoll
Вимір	3D ізо	2D	3D	3D
Permadeath	Так (roguelite)	Так (roguelite)	Так (roguelite)	Так (roguelite)
Процедурна генерація	Часткова	Chunk-based	Зонова	Повна (9-шаровий pipeline)
Воксельна геометрія	Ні	Ні	Ні	Так
RPG-прогресія	Так	Так	Так	Так
Utility AI ворогів	Ні	Ні	Ні	Так
Кастомні шейдери	Ні	Ні	Ні	Так

1.3 Огляд алгоритмів процедурної генерації карт

Вибір алгоритму генерації - це не тільки питання обчислювальної складності [7]. Різні методи дають принципово різний «характер» карти. Є методи, що гарантують зв'язність, але дають механічний вигляд. Є такі, що генерують органічний рельєф, але можуть розпастися на ізольовані островці. У реальних проєктах майже завжди комбінують кілька підходів.

Перлін розробив свій алгоритм у 1983 році для фільму TRON [3] - потрібні були природні поверхні без ручного моделювання. Алгоритм будує решітку з псевдовипадковими градієнтами у вузлах; значення у довільній точці обчислюється через інтерполяцію між сусідніми градієнтами. Результат - плавний шум без різких стрибків.

У 2001 році Перлін запропонував Simplex noise [4]. Трикутна решітка замість квадратної, менше directional artifacts, швидше рахується у вищих вимірах. Для 2D-генерації карт різниця не критична, але Simplex зустрічається частіше.

Обидва алгоритми майже завжди використовують у вигляді fBm (fractional Brownian Motion) [6, 15] - суми кількох октав:

$$fBm(x) = \sum (persistence^i) \times noise(x \times lacunarity^i), \quad i = 0..N-1$$

Lacunarity визначає, як зростає частота між октавами (зазвичай 2.0), persistence - як зменшується вплив наступних октав (зазвичай 0.5). Більше октав - більше дрібних деталей, загальна форма не змінюється. SilhouetteLayer у LetsRoll будує маску арени через fBm.

Стандарт для dungeon generation з 1990-х [15]. Алгоритм бере прямокутник і рекурсивно ділить його перегородками - горизонтальними або вертикальними - поки секції не досягнуть заданого мінімального розміру. У кожній кінцевій секції - кімната, між сусідніми секціями - коридор. Зв'язність гарантована структурно. Але результат виглядає механічно: прямокутні кімнати, прямі коридори, рівномірна сітковість. Для вокселів з органічним рельєфом не підходить.

Добре відтворюють природні форми [15] - зокрема печери. Процедура: заповнити сітку випадковими 0/1 із заданою щільністю і застосовувати правило

«якщо більше чотирьох із восьми сусідів зайняті - стань зайнятою». Через 4-5 ітерацій хаос стягується у форми, схожі на карстові системи.

Проблема: зв'язність не гарантована. Після генерації карта може розпастися на ізольовані островці, і це треба перевіряти окремо - flood fill для пошуку найбільшого зв'язного компонента. LetsRoll використовує flood fill всередині SilhouetteLayer саме з цієї причини, але після порогування fBm-карти, а не як окремий генератор.

Максим Гумін запропонував WFC у 2016 році [5]. Назва - метафора з квантової механіки, не буквальна. Кожна клітина починає у «суперпозиції» всіх можливих тайлів; алгоритм вибирає ту з найменшою кількістю варіантів, фіксує один, поширює обмеження на сусідів - і так до кінця.

При хорошому зразку результати WFC вражають. Але є практична слабкість: при нерозв'язних обмеженнях алгоритм зупиняється у стані «суперечності». На великих картах з багатьма байомами це трапляється достатньо часто, щоб стати реальною проблемою.

У LetsRoll WfcMapGenerator.cs реалізований, але основним не є. WFC потребує підготовленого зразка для кожного байому і не дає прямого контролю над формою арени. Pipeline на fBm вирішує обидві задачі через параметри.

Не самостійний генератор, а постобробка. Ерозія стискає регіони - відсікає виступи і вузькі перемички. Дилатація розширює. Послідовно застосувати ерозію, а потім дилатацію (морфологічне відкриття) - це прибирає дрібні ізольовані фрагменти без зміни загальної форми.

SilhouetteLayer застосовує цю пару після порогування fBm-карти: ерозія прибирає «сміття», дилатація відновлює масштаб. На виході - чиста маска арени.

Порівняння методів генерації карт за результатом, гарантіями зв'язності та контролем форми наведено в таблиці 1.2.

Порівняння алгоритмів процедурної генерації

Алгоритм	Результат	Зв'язність	Контроль форми	Складність	У LetsRoll
Perlin / Simplex	Органічний рельєф	Не гарантована	Висока	$O(n)$	Так
fBm	Рельєф із деталями	Не гарантована	Висока	$O(n \times \text{octaves})$	Так (основний)
BSP	Кімнати і коридори	Гарантована	Середня	$O(n \log n)$	Ні
Cellular Automata	Органічні форми	Потребує flood fill	Низька	$O(n \times \text{ітерації})$	Частково
WFC	Патерн-коректний рівень	Гарантована*	Через зразок	$O(n \log n)$	Реалізовано, не активне
Морфологія	Постобробка маски	-	Висока	$O(n)$	Так

Графічне порівняння двох підходів - WFC і власного pipeline на основі fBm з морфологічним відкриттям - наведено на рисунку 1.4.

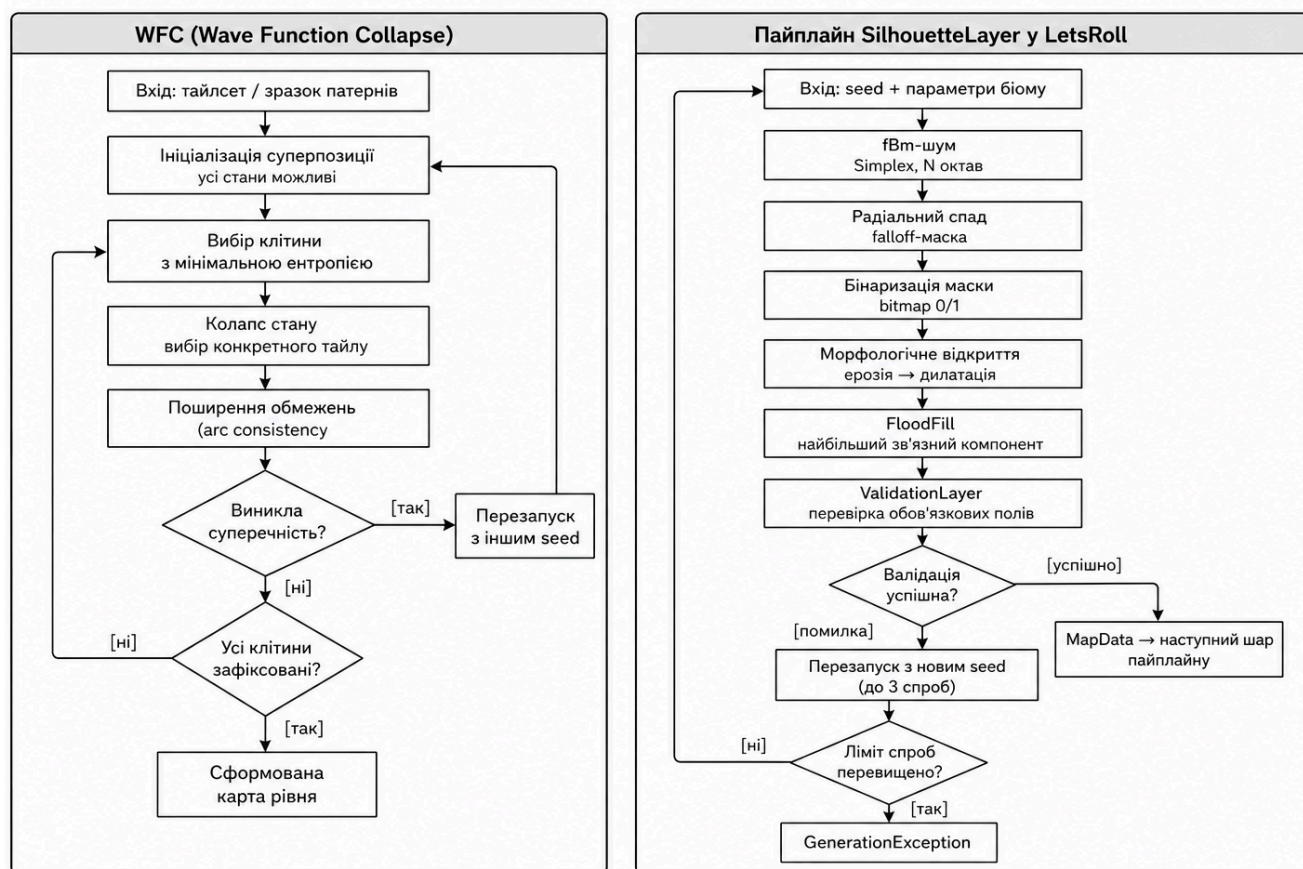


Рис. 1.4 Порівняння підходів WFC та Pipeline (fBm + морфологія)

1.4 Вибір технологій

Три основні варіанти для 3D інді-розробки: Unity, Unreal Engine, Godot [24]. На практиці вибір залежить не від технічних характеристик рушія, а від того, з чим розробник вже працює і що конкретно потрібне проєкту.

Unity (Unity Technologies, 2005). Мова - C#. Universal Render Pipeline для крос-платформних проєктів [10], HDRP для фотореалізму. До 200 000 доларів річного доходу - безкоштовний.

Unreal Engine 5 дає Lumen і Nanite прямо з коробки. AAA-рівень графіки без налаштувань - але і поріг входу вищий. Для проєкту такого масштабу більша частина цих можливостей просто не потрібна.

Godot 4 - open-source, MIT-ліцензія, GDScript або C#. Ніяких роялті. Екосистема плагінів і документація помітно менші, що на практиці означає більше часу на пошук рішень.

Для LetsRoll обрано Unity URP. Досвід з рушієм вже є. URP дозволяє писати HLSL-шейдери вручну - це потрібно для VoxelFlatLit і InvisibleWallRipple. Вбудований NavMesh закриває задачу навігації ворогів [11, 17]. ScriptableObject підходить [12] для конфігурацій байомів і ворогів.

Зведене порівняння трьох рушіїв наведено в таблиці 1.3.

Таблиця 1.3

Порівняння ігрових рушіїв

Критерій	Unity (URP)	Unreal Engine 5	Godot 4
Мова	C#	C++ / Blueprints	GDScript / C#
Кастомні шейдери	HLSL	HLSL / Material Editor	GLSL / VisualShader
Ліцензія	Безкоштовно до \$200K/рік	5% після \$1M	MIT
Вбудований NavMesh	Так	Так (Recast)	Так (NavigationServer)
Data assets	ScriptableObject	DataAsset (C++)	Resource
Документація	Велика	Велика	Середня
Досвід автора	Так	Ні	Ні

2 ПРОЄКТУВАННЯ 3D ГРИ У ЖАНРІ ROGUELIKE

2.1 Концепція та ігровий цикл

LetsRoll будується навколо двох ідей. Кожен забіг - це нова карта. Смерть гравця скидає прогрес забігу, але не все: досвід і відкриті вузли дерева навичок лишаються між спробами. Це roguelite-модель, а не класична roguelike.

Конкретна структура така: гравець проходить послідовні арени, на кожній треба знищити всіх ворогів, тоді відкривається портал. Після порталу - нова арена. Після смерті - новий забіг з іншим seed-ом. Всього 20 арен у забігу, і кожна генерується окремо.

Загальна схема ігрового циклу показана на рисунку 2.1.

Ротація байомів не випадкова. Рівні 0-3 завжди Castle, 4-7 - Forest, 8-11 - Cave, 12-15 - Ice, 16-19 - Hell. Це рішення про контроль темпу: гравець стикається з кожним байомом у визначений момент, а не коли пощастить. Після 20-го рівня починається нескінченний режим з seed-детермінованою ротацією - для гравців, що доходять до кінця і хочуть продовжувати.

Один рівень проходить шість фаз. Спочатку генерація карти - за seed-ом, байомом і номером рівня. Потім спавн ворогів і об'єктів. Потім бій: гравець знищує ворогів. Коли падає останній, система публікує AllEnemiesDefeatedEvent і з'являється портал. Гравець входить - рівень закрито, починається наступний. Якщо гравець загинув - забіг завершується.

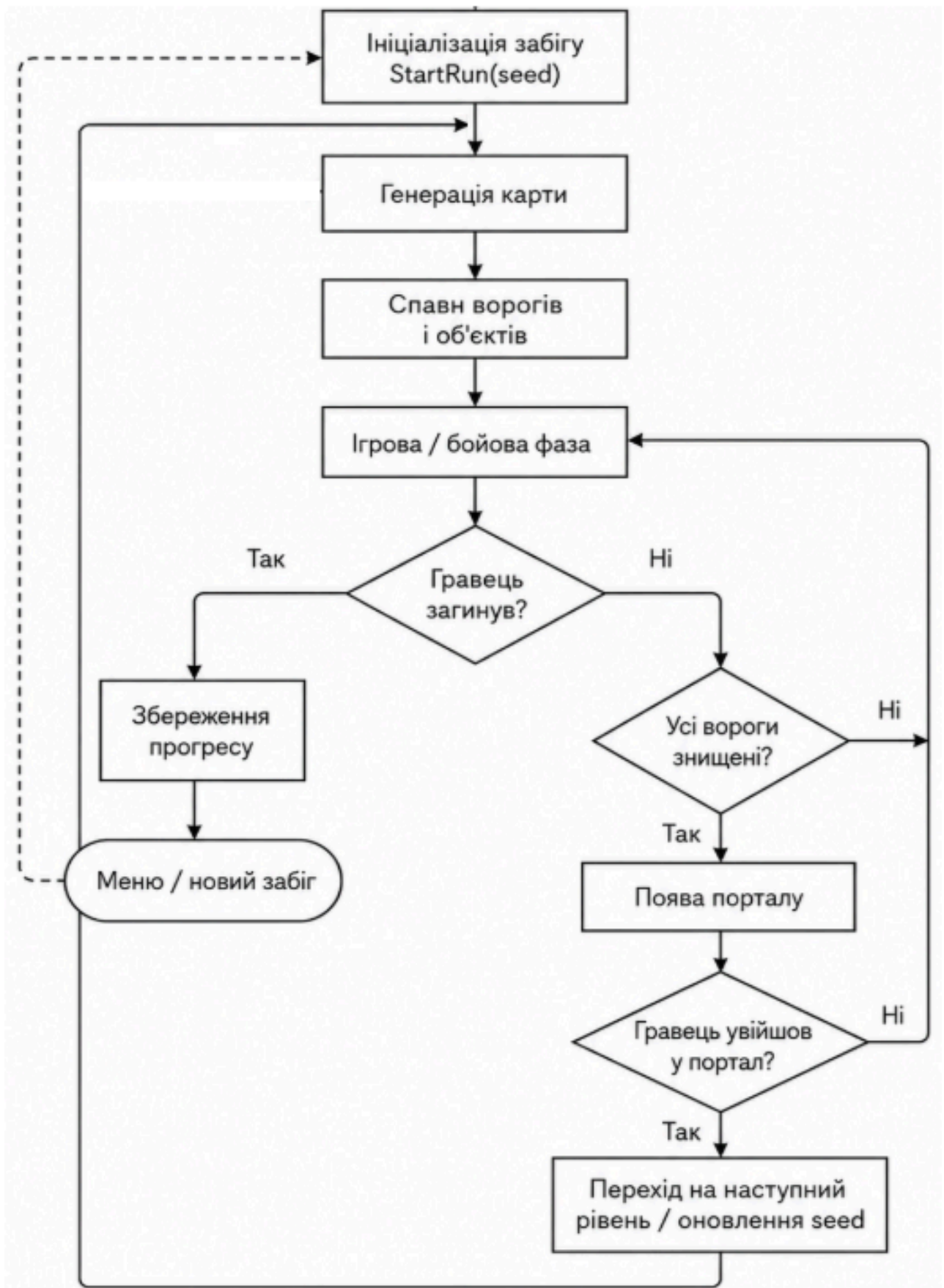


Рис. 2.1 Блок-схема алгоритму ігрового циклу

2.2 Вимоги до програмного забезпечення

Система процедурної генерації карт будує унікальну арену при кожному запуску рівня на основі seed і параметрів байому.

Система ворогів реалізує Utility AI з набором тактик і підтримкою групової координації через SquadDirector.

Система статів гравця підтримує три типи модифікаторів (Flat, Percent, Multiplier) і lazy recalculation при зміні будь-якого модифікатора.

Дерево навичок перевіряє зв'язність графа BFS-обходом при кожному знятті вузла.

Система рівнів відстежує прогрес забігу і генерує наступний рівень після проходження порталу.

Система предметів підтримує екіпірування з автоматичним застосуванням стат-модифікаторів.

Вокселі рендеряться через власний mesher з підтримкою greedy merging видимих граней.

Воксельна карта будується асинхронно і не блокує основний потік Unity.

Час генерації карти розміром $128 \times 128 \times 32$ вокселі - не більше 2 секунд на Intel Core i5, 8 ГБ RAM.

Частота кадрів при активній бойовій сцені з 10 і більше ворогами - не менше 60 FPS.

Детермінованість генерації: однаковий seed і байом завжди дають однакову карту.

Архітектура підтримує додавання нових байомів, архетипів арен і типів ворогів без зміни існуючого коду (Open/Closed Principle).

Діаграма варіантів використання, що відображає основні взаємодії гравця з системою, наведена на рисунку 2.2.

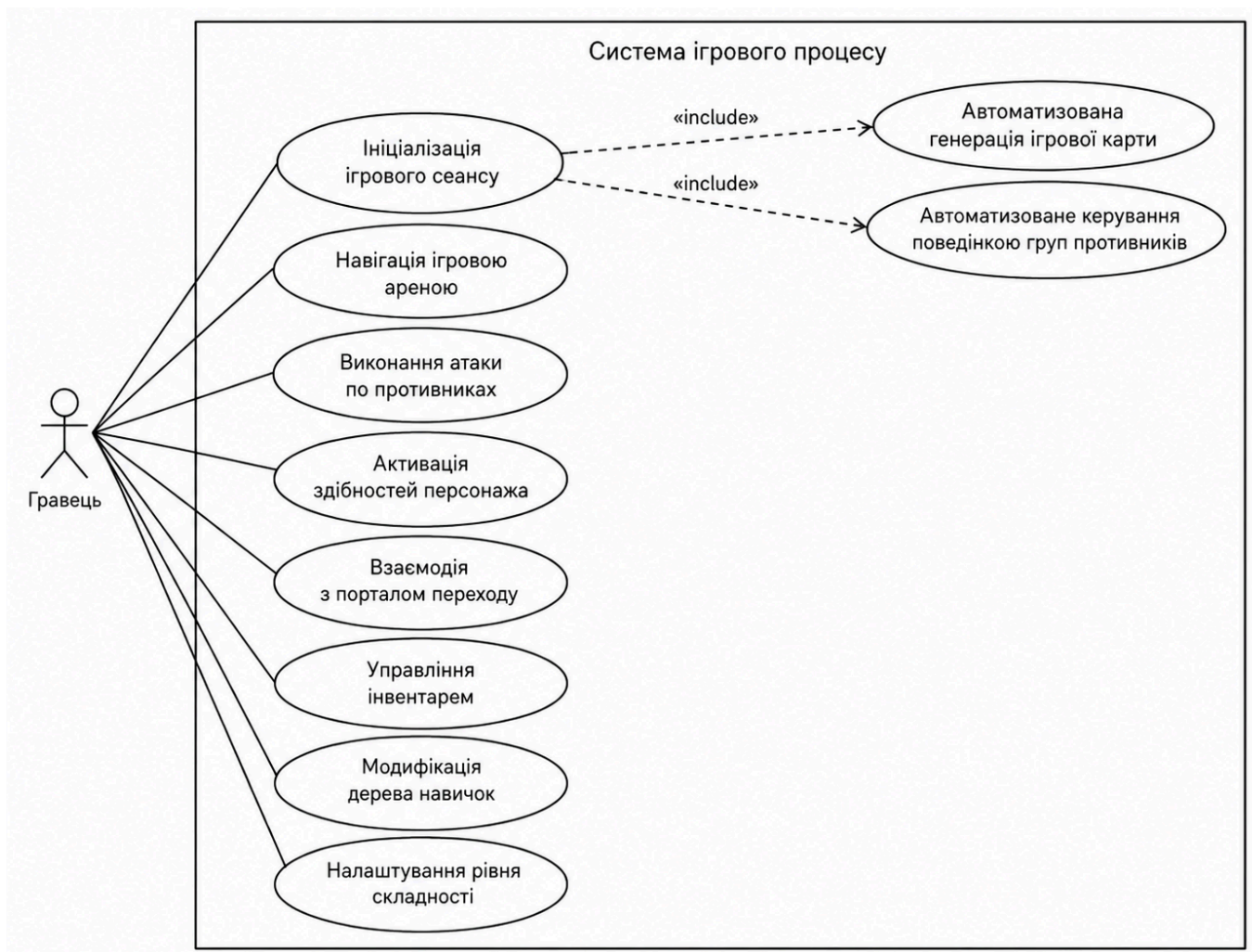


Рис. 2.2 Use Case діаграма гри LetsRoll

2.3 Архітектура застосунку

Найпростіший спосіб зв'язати підсистеми гри - зробити їх залежними одна від одної напряму. Генерація карти викликає UI, UI викликає систему статів, система статів викликає ворогів. Це працює на невеликих проєктах і стає некерованим після певного порогу складності.

LetsRoll організовано по-іншому. Service Locator керує доступом до глобальних сервісів [22] без прямих залежностей між класами. EventBus дає підсистемам спілкуватися через події - одна публікує, інша підписується, і обидві нічого не знають про існування одна одної. Генерація карти не знає про UI. AI

ворогів не знає про систему статів. Кожну підсистему можна змінити або замінити, не торкаючись інших.

Core/ServiceLocator.cs - глобальний реєстр сервісів. Thread-safe через lock. П'ять методів: Register<T>, Get<T>, TryGet<T>, Unregister<T>, HasService<T>.

Порівняно з Singleton-ом перевага одна, але суттєва: реалізацію можна підмінити. Зареєструвати інший клас по тому ж інтерфейсу - і всі споживачі отримають нову версію без змін у своєму коді. Для тестування це зручно; для масштабування теж.

GameBootstrapper - статичний клас з атрибутом RuntimeInitializeOnLoadMethod(BeforeSceneLoad)]. Він реєструє всі сервіси ще до того, як Unity завантажить першу сцену. Список: InputService, ExperienceService, ItemDatabase, InventoryService, SceneLoader, BiomeRegistry, LevelFlowService, DifficultyService.

Core/EventBus.cs - типізована publish/subscribe шина. Кожен тип події отримує власний EventBus<T> з власним списком підписників.

Є один нетривіальний момент у реалізації: публікація копіює список підписників перед викликом. Без цього код ламається в конкретній ситуації - якщо підписник відписується під час обробки тієї ж події, ітерація по списку дає помилку. Копія усуває цей сценарій.

Центральні події: LevelStartedEvent, AllEnemiesDefeatedEvent, PortalEnteredEvent, PlayerDiedEvent, EnemyDiedEvent. LevelFlowService підписується на всі ці події і оркеструє ігровий цикл виключно через них. Прямих викликів між системами немає.

Код розбито на дві директорії: Core і Features. Assets/_Project/Scripts/Core/ тримає Service Locator, EventBus, GameBootstrapper і інтерфейси State Machine. Assets/_Project/Scripts/Features/ тримає решту: Voxels, Player, Enemies, UI, Items. Це не просто організаційне рішення - воно гарантує, що Features залежать від Core, але не від інших Features.

Конфігурації оформлені як ScriptableObject [12]: BiomeSO, EnemyConfigSO, EnemyBrainSO, ItemDataSO, PassiveSkillNodeSO. Параметри можна змінювати

прямо в редакторі Unity без перекомпіляції - і це справді зручно при налаштуванні балансу.

Архітектурна діаграма системи, що відображає взаємодію Service Locator, EventBus і Features, наведена на рисунку 2.3.

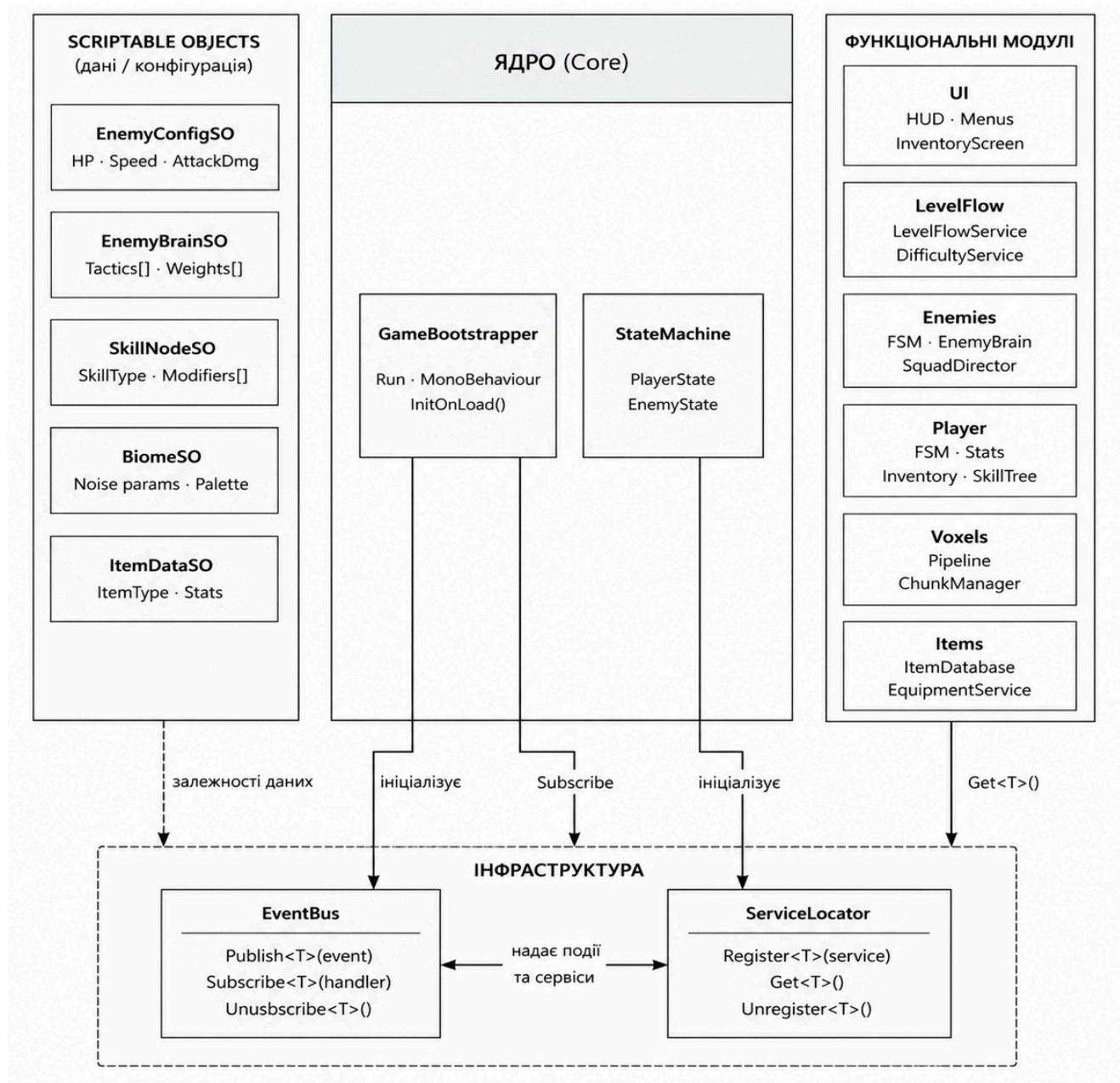


Рис. 2.3 Архітектурна діаграма (Service Locator, EventBus, Features)

Спрощена діаграма класів основних підсистем гри показана на рисунку 2.4.

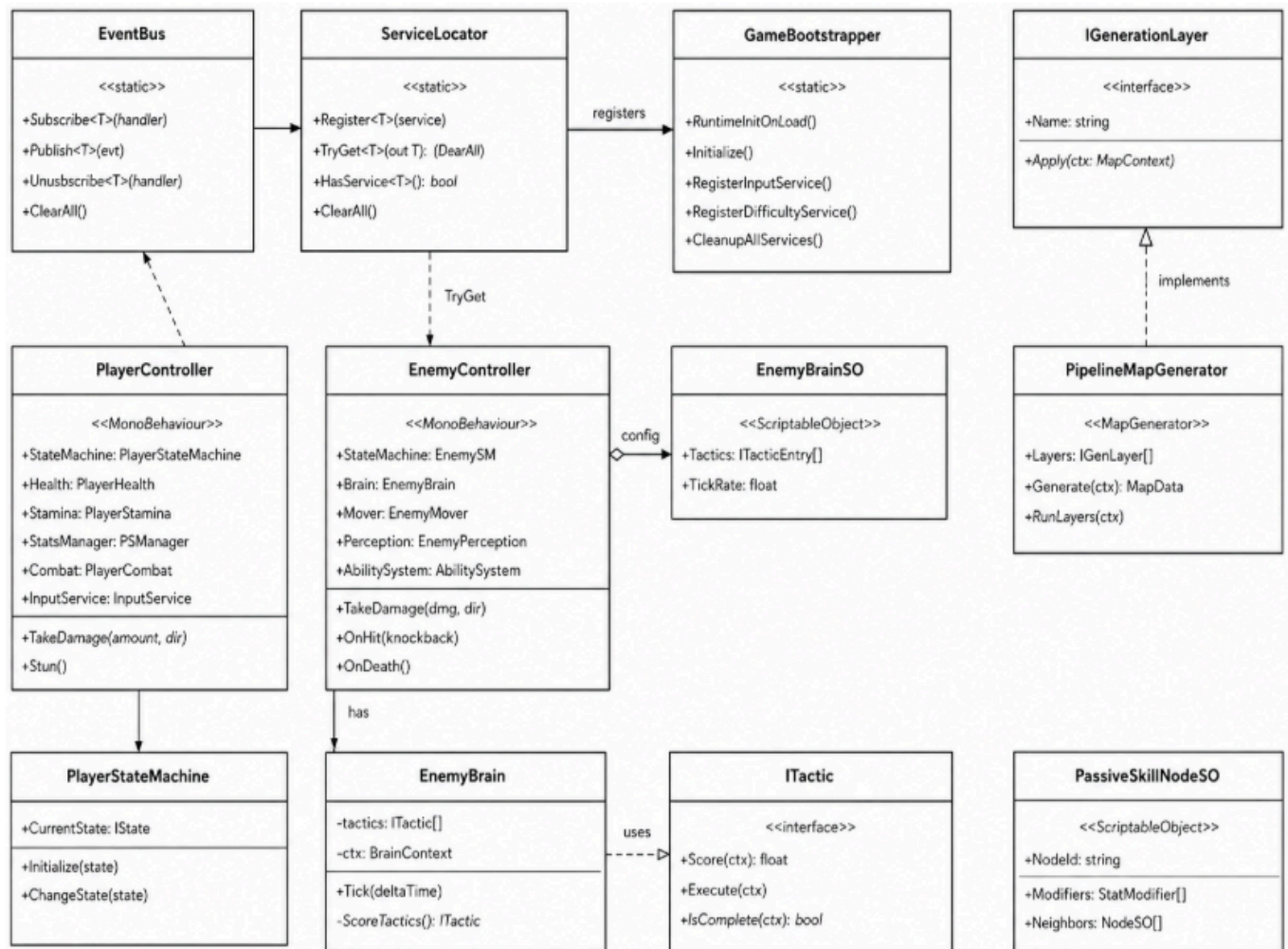


Рис. 2.4 Спрощена діаграма класів основних підсистем

2.4 Проектування алгоритму процедурної генерації карти

Генерація карти - послідовний pipeline з дев'яти шарів. Кожен шар реалізує інтерфейс `IGenerationLayer` і отримує спільний контекст `MapContext`, що передається далі. `MapContext` накопичує результати: після першого шару там з'являється маска ари, після другого - `heightmap`, після третього - зони і так далі. Кожен наступний шар читає те, що додали попередні.

`PipelineMapGenerator` запускає весь pipeline через `Generate(biome, levelIndex, levelSeed)`. Якщо шар кидає `GenerationException` - автоматичний перезапуск з новим `seed`, максимум три спроби. Детермінованість забезпечує FNV-хеш від `levelSeed`: однаковий `seed` дає однакову карту незалежно від платформи.

Перелік шарів pipeline з коротким описом функції кожного наведено в таблиці 2.1.

Таблиця 2.1

Шари pipeline генерації карти

№	Шар	Що робить
1	SilhouetteLayer	Будує 2D маску арени: fBm-шум + радіальний falloff + пороговання + морфологічне відкриття + FloodFill
2	TerrainLayer	Заповнює воксельну сітку землею за висотою (Perlin noise, амплітуда з BiomeSO)
3	ZoneLayoutLayer	Вибирає архетип арени (Bowl/Grove/Plateau/Ruins/Spine) і розставляє 5 зон
4	BiomePaintLayer	Фарбує поверхню відповідно до biome-матеріалів
5	StructuresLayer	Розставляє структури (CastleRuin, PillarCluster, RubbleMound, StoneArch)
6	HazardLayer	Додає небезпечні зони (отруйні пні тощо)
7	DecorLayer	Декор: дерева, гриби, каміння, вогнища, кристали
8	SpawnPointLayer	Визначає точки spawn гравця, ворогів і порталу
9	ValidationLayer	Перевіряє наявність обов'язкових даних (heightmap, spawn points)

Загальна блок-схема pipeline генерації карти з усіма дев'ятьма шарами показана на рисунку 2.5.

SilhouetteLayer - перший шар і найважливіший. Він встановлює форму арени, і всі наступні 8 шарів працюють у межах того, що він дав. Якщо форма вийшла погана - вся карта погана.

Алгоритм має п'ять кроків.

Крок 1: fBm-карта шуму. Параметри - кількість октав, persistence, lacunarity, scale - беруться з BiomeSO поточного байому. Реалізація власна: Simplex noise з seedable permutation table, а не UnityEngine.Mathf.PerlinNoise. Причина важлива: вбудована функція Unity не гарантує однаковий вихід між різними версіями рушія. Для детермінованого pipeline це неприйнятно.

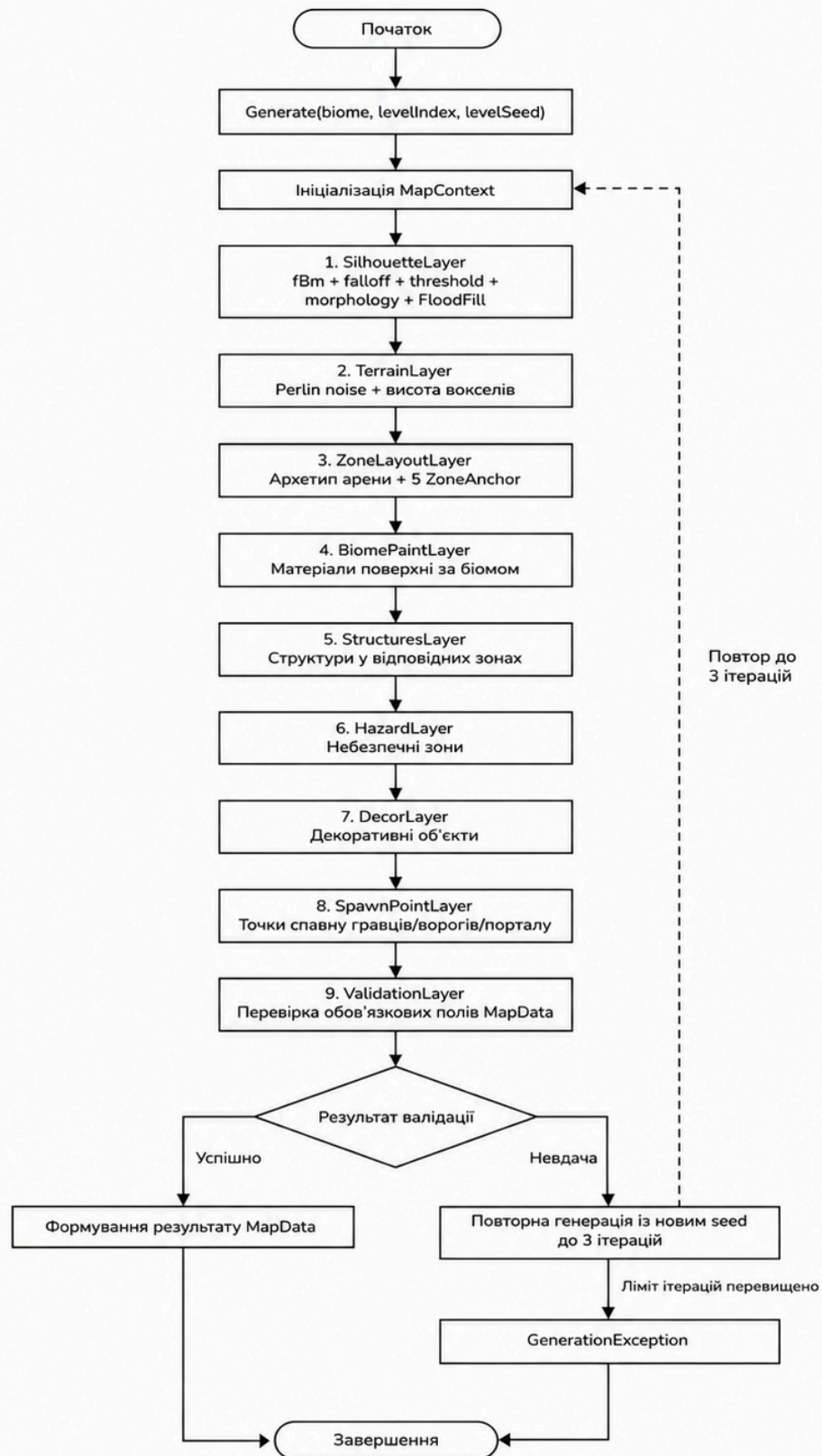


Рис. 2.5 Блок-схема pipeline генерації карти (9 шарів)

Крок 2: множення на радіальний falloff - маску, що спадає від 1.0 в центрі до 0.0 по краях за квадратичним законом. Без неї арена «витікає» за межі області генерації і втрачає форму, перетворюючись на безформну ляпку без чітких границь.

Крок 3: порогування в бінарну маску. Клітинка або входить в арену (1), або ні (0). Поріг задається в BiomeSO і регулює щільність заповнення: більший поріг - компактна арена з більшою кількістю порожнин, менший поріг - розгалужена арена зі складною геометрією і численними відгалуженнями.

Крок 4: морфологічне відкриття. Спочатку ерозія - прибирає дрібні ізольовані фрагменти і вузькі виступи, що погіршують прохідність. Потім дилатація - відновлює загальний розмір арени до значень, близьких до вихідних. Разом ці дві операції прибирають шум на краях без зміни форми арени в цілому.

Крок 5: FloodFill від центральної точки. Залишає тільки найбільший зв'язний компонент маски, відкидаючи всі ізольовані ділянки. Гравець гарантовано потрапляє на суцільний майданчик - жодних ізольованих клаптів, до яких неможливо дістатись.

Послідовність кроків і їх результат у візуальній формі продемонстровано на рисунку 2.6.

ZoneLayoutLayer вибирає один з п'яти архетипів зонування арени. Bowl - відкрита миска без укриттів, бій у центрі, гравець на видноті з усіх боків. Grove - галявина з деревами по периметру, що формують природну межу простору. Plateau - плато з перепадами висот і уступами, які створюють вертикальний геймплей. Ruins - колони і уламки, що формують природні лінії укриттів і коридори між ними. Spine - вузький коридор зі стовпами по боках, що змушує гравця рухатись лінійно і використовувати укриття.

Байоми - Castle, Forest, Cave, Ice, Hell - визначають матеріали вокселів, типи структур і декору, параметри шуму через BiomeSO. Один і той самий архетип у різних байомах виглядає і грається по-різному: Ruins в Castle - кам'яні колони з обвалених стін замку, Ruins в Ice - крижані брили і уламки замерзлих скель. Це дає значну різноманітність контенту без дублювання коду архетипу, оскільки логіка розміщення укриттів залишається єдиною, а змінюються лише візуальні і матеріальні параметри.

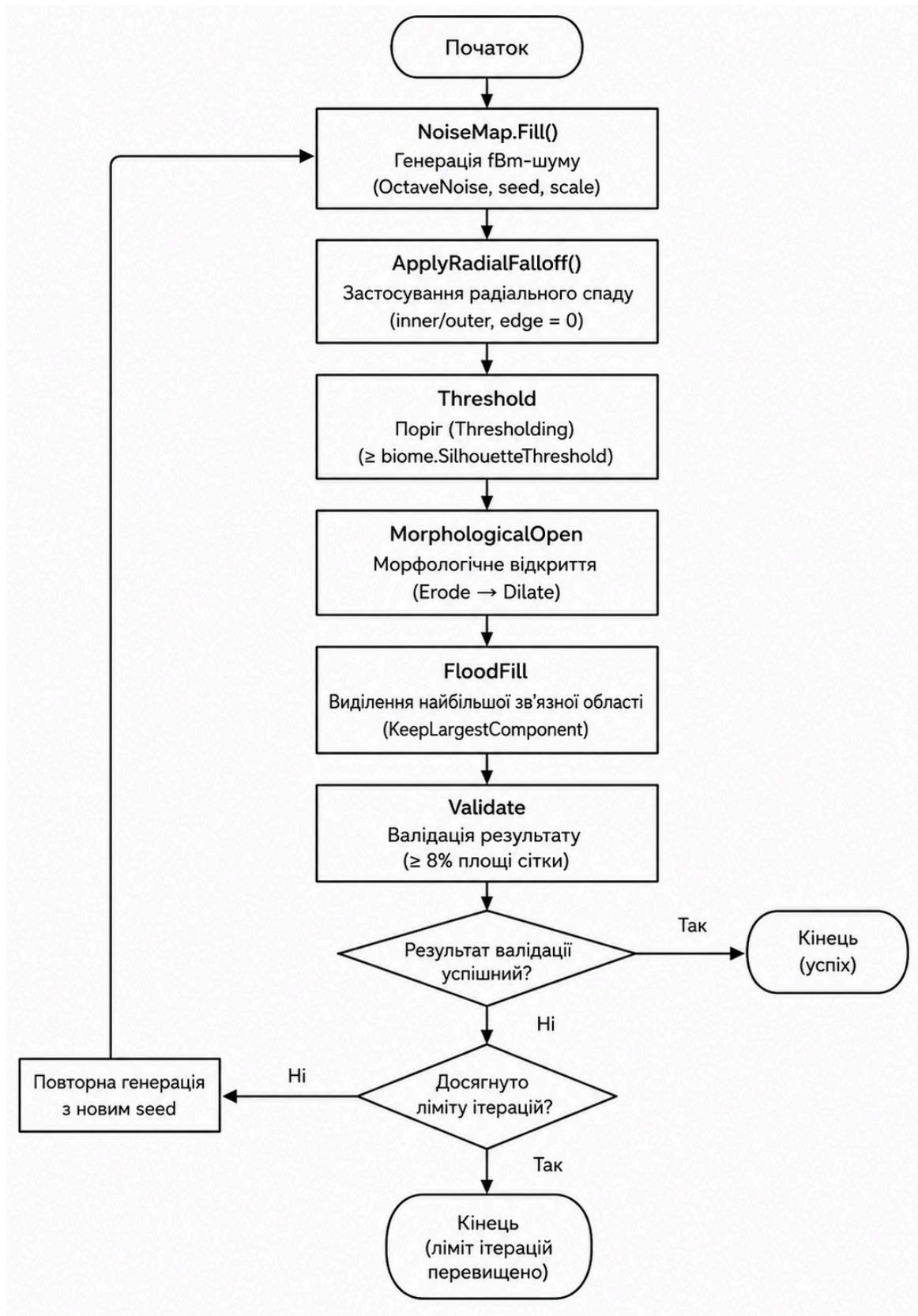


Рис. 2.6 Блок-схема алгоритму формування силуету рівня

2.5 Прокєтування систем гравця та ворогів

Без State Machine код поведінки гравця [13, 14] виглядає передбачувано: один великий Update з перевітками на кожному кроці. «Якщо гравець атакує і не в stunned і не в dead і стаміна більше нуля...» - і таких рядків стає десятки. Кожна нова механіка додає нові if-и. Через місяць розбиратися в такому коді вже важко.

FSM вирішує це поділом. Вісім станів: Idle, Move, Jump, Air, Attack, Stunned, Land, Dead. Кожен реалізує інтерфейс IPlayerState з методами Enter(), LogicUpdate(), Exit(). PlayerStateMachine зберігає поточний стан і делегує йому виклики Update. Стан Attack містить тільки логіку атаки. Стан Stunned - тільки stunned. Переходи між станами явні й задокументовані.

Гравець складається з шести компонентів. CharacterController відповідає за рух. PlayerStamina витрачає стаміну на дії. PlayerHealth відстежує здоров'я. PlayerStatsManager рахує фінальні статистика. PlayerCombat обробляє атаки і хіти. PlayerAnimationController транслює стани в Animator. Рух відносний до камери з SmoothDampAngle - без цього обертання персонажа виглядає різким.

Діаграма станів гравця з усіма переходами наведена на рисунку 2.7.

Ворог керується двома рівнями. Перший - State Machine для загального режиму [13]: Idle, Patrol, Combat, Stunned, Dead. EnemyPerception відстежує дальність і кут зору і перемикає FSM між Idle/Patrol і Combat. EnemyMover веде навігацію через Unity NavMesh [11].

Другий рівень - EnemyBrain. Він працює всередині Combat-стану і відповідає за конкретні дії. Кожен тик - не кожен кадр, це важливо для продуктивності - EnemyBrain оцінює всі доступні тактики і вибирає ту з найвищим Score. Тактик 17: ChaseTactic, AttackTactic, FlankTactic, KiteTactic, DodgeTactic, TakeCoverTactic, RetreatTactic, SelfHealTactic, HoldAtRangeTactic, ChargePlayerTactic, ThrowRockTactic, CastRootsTactic, CastHealPuddleTactic, SeekHealPuddleTactic, RegroupTactic, ScreenSupportTactic, CombatSignalTactic.

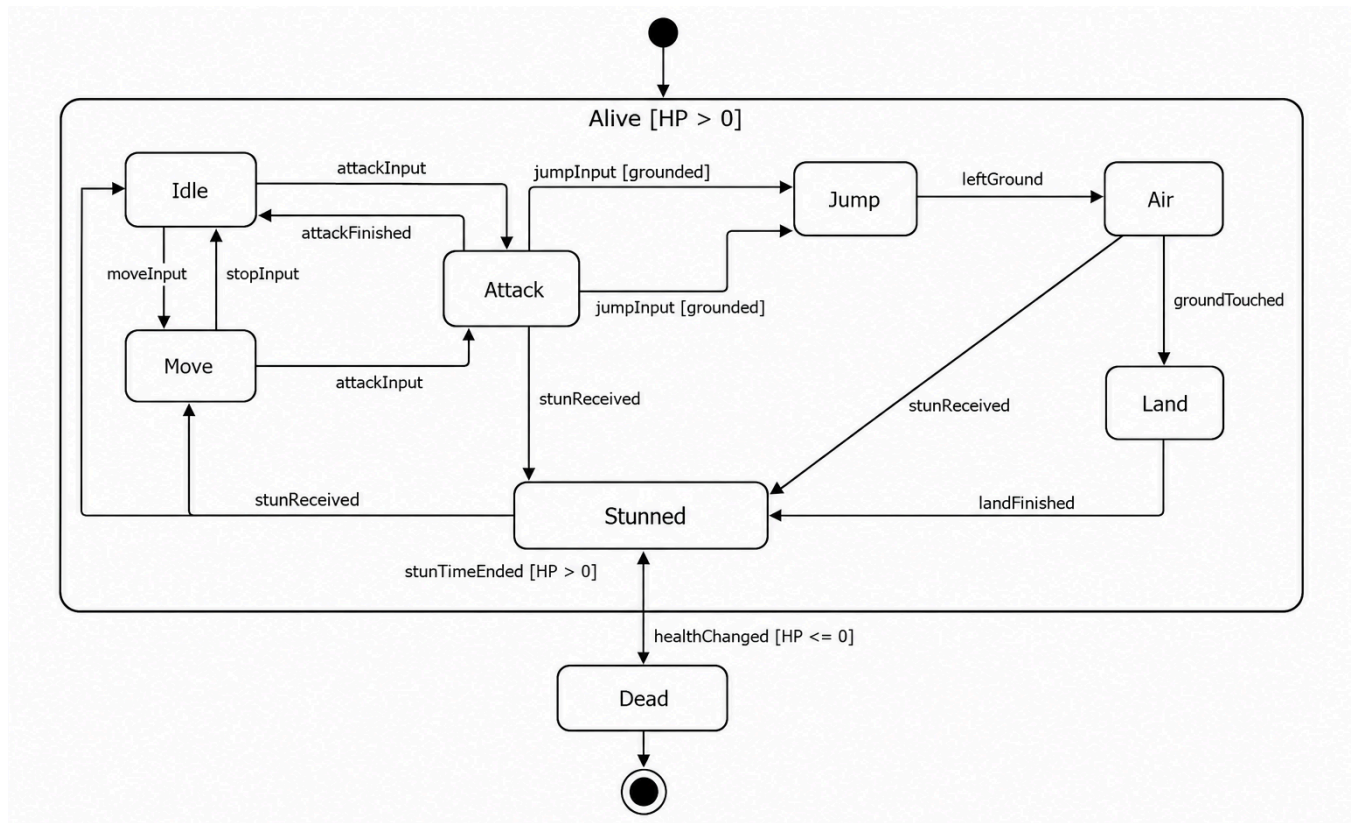


Рис. 2.7 Діаграма станів гравця (Player State Machine)

SwitchInertiaBonus - надбавка до Score за продовження поточної тактики. Без неї ворог з двома тактиками з близьким Score перемикається між ними кожен тік. Візуально це виглядає як смикання - перейшов два кроки вліво, стоп, два кроки вправо, стоп. З бонусом ворог дотримується вибраної тактики поки та не стає помітно гірше за конкурентів.

Stuck-detector вирішує зворотну проблему. Якщо тактика не переміщує ворога 2.5 секунди - примусове перепланування незалежно від Score. Трапляється коли ворог застряє за геометрією або навігаційний шлях недосяжний.

SquadDirector координує загін як єдине ціле. Він розподіляє ролі між членами - frontliner, flanker, support - і через PlayerTelegraphBroadcaster транслює дії гравця всьому загону. SquadMemory тримає актуальні позиції і стани всіх бійців.

Діаграма станів ворога з прив'язкою EnemyBrain до Combat-стану наведена на рисунку 2.8.

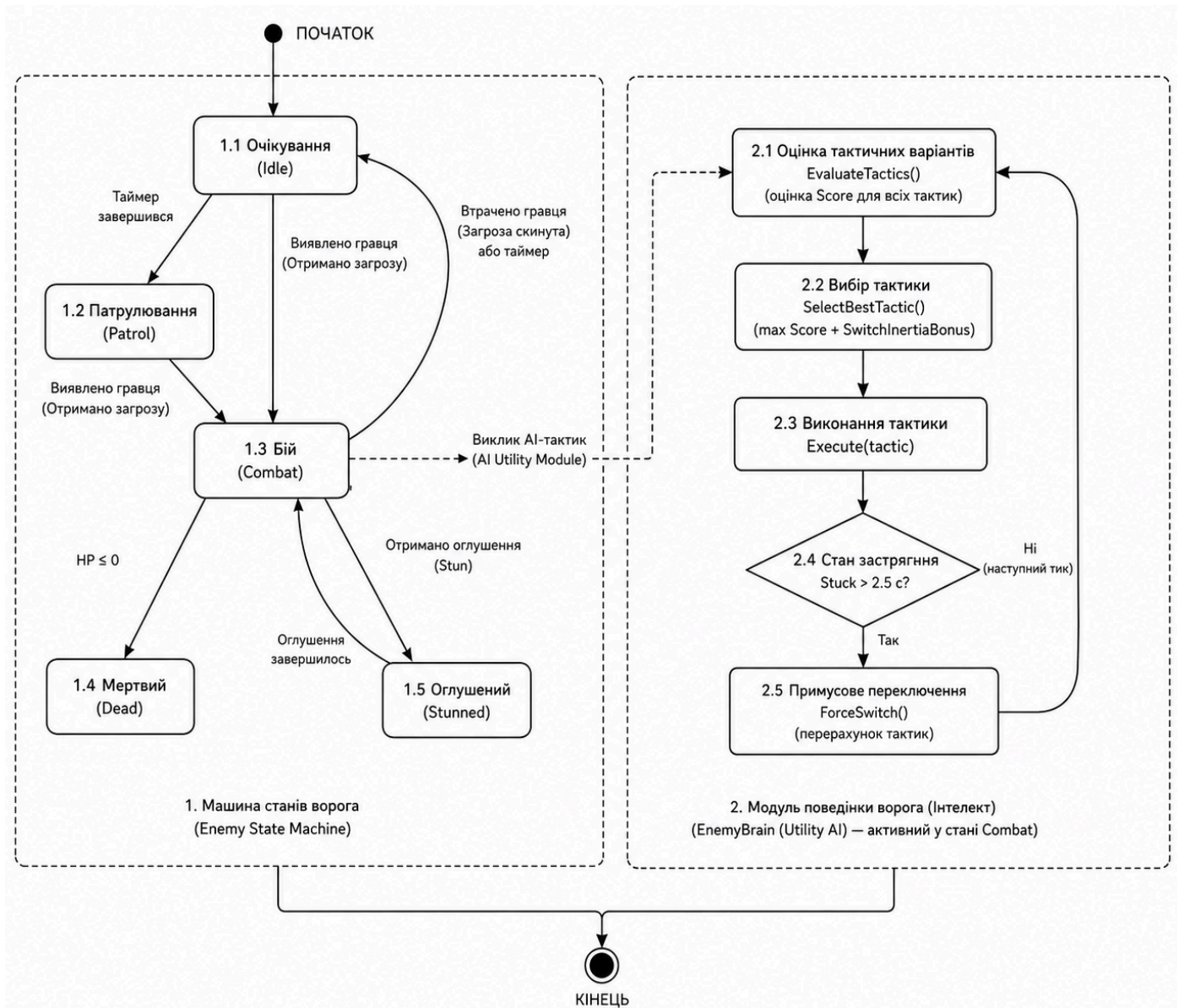


Рис. 2.8 Діаграма станів ворога (Enemy State Machine + EnemyBrain Utility AI)

2.6 Проектування RPG-підсистем

Формула розрахунку характеристик персонажа побудована за принципом, поширеним у RPG-системах, зокрема у Path of Exile:

$$Final = (Base + \Sigma Flat) \times (1 + \Sigma Percent / 100) \times IMultiplier, \quad (2.1)$$

де Final - підсумкове значення характеристики, яке використовується ігровою логікою, наприклад для обчислення шкоди, здоров'я або швидкості пересування персонажа;

Base - базове значення характеристики, задане в конфігурації класу персонажа або типу істоти до застосування модифікаторів;

Σ Flat - сума всіх адитивних модифікаторів, що додаються безпосередньо до базового значення;

Σ Percent - сума всіх відсоткових приростів, виражена у відсотках і додана перед діленням на 100;

PMultiplier - добуток усіх мультиплікативних модифікаторів, що застосовуються послідовно.

Спочатку плоскі надбавки до базового значення. Потім відсоткові модифікатори. Потім мультиплікатори. Порядок небезпричинний: плоскі надбавки сильніші на ранніх стадіях, коли Base ще малий, відсоткові - пізніше, коли є що відсотком збільшувати. І важливий технічний момент: два Percent-модифікатори по 50% дають загальний ефект +100% від значення після Flat-кроку, а не множення кожного на результат попереднього.

Рекалькуляція - lazy. PlayerStatsManager тримає прапор dirty для кожного стаття і перераховує тільки при виклику GetFinalValue() для забрудненого стаття. Якщо за один кадр змінилися десять модифікаторів - перерахунок виконується один раз, а не десять разів поспіль. На практиці це помітно при масовому застосуванні ефектів, коли гравець одночасно вдягає комплект екіпіровки або отримує серію бафів від ультимативної здібності.

Модифікатори надходять з трьох джерел: equipment_* від надітих предметів, buff_* від тимчасових ефектів і здібностей, skilltree від відкритих вузлів дерева навичок. PlayerStatsManager.ApplySkillTreeBonuses() викликається після кожного OnTreeUpdated, що гарантує синхронізацію стану статів із поточним станом дерева навичок без потреби у ручному оновленні.

Гравець має 15 статів, розподілених за функціональним призначенням. Базові: `max_health`, `max_mana`, `max_stamina`, `strength`, `dexterity`, `intelligence`. Захисні: `armor`, `evasion`, `block_chance`, `dodge`. Атакувальні: `physical_damage`, `attack_speed`, `crit_chance`, `crit_damage`. Допоміжні: `movement_speed`, `life_leech`. Такий поділ спрощує балансування і дозволяє формувати білди з акцентом на конкретний стиль гри.

Дерево навичок - граф вузлів `PassiveSkillNodeSO`. Шість типів: `Small` (малий стат-бонус), `Notable` (суттєвий бонус з помітним впливом на білд), `Keystone` (змінює механіку гри, наприклад, замінює ману на здоров'я як ресурс для здібностей), `ClassStart` (стартова точка, прив'язана до класу персонажа), `JewelSocket` (слот для самоцвітів з додатковими бонусами), `Mastery` (вибір зі списку покращень для певної гілки).

Гравець відкриває тільки вузли, суміжні з уже відкритими, починаючи від `ClassStart`. Це стандарт жанру і відповідає очікуванням аудиторії, знайомої з аналогічними системами. Нестандартне - перевірка при знятті: `SkillTreeService.TryDeallocateNode()` запускає BFS від `ClassStart` і перевіряє, чи всі взяті вузли залишаться досяжними після видалення обраного. Якщо граф розпадається на дві не пов'язані частини - операція відхиляється, а гравець отримує відповідне повідомлення. Без цієї перевірки гравець міг би зняти «міст» між частинами дерева і залишити одну з них без дороги від `ClassStart` - технічно взятую, але ізольовану, що призвело б до неконсистентного стану системи і потенційних помилок у логіці нарахування бонусів.

Загальна схема системи статів і дерева навичок з джерелами модифікаторів наведена на рисунку 2.9.

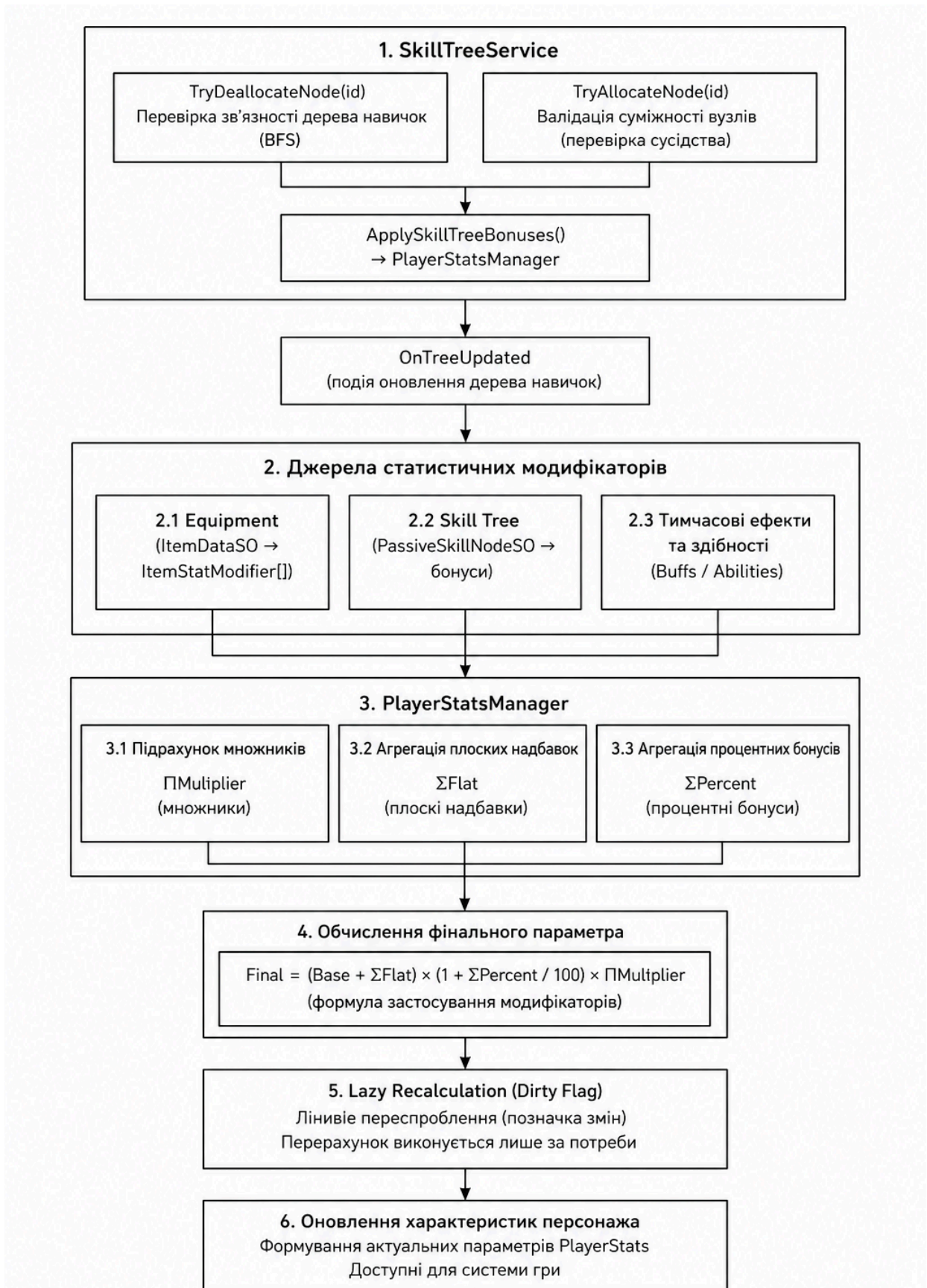


Рис. 2.9 Схема системи статів (Flat/Percent/Multiplier + Skill Tree)

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ГРИ

3.1 Середовище розробки та структура проєкту

Unity 6.4 LTS URP, C# [10]. Вибір URP проти Built-in RP не очевидний - Built-in RP простіший і добре задокументований. Але URP дає одну перевагу, що для цього проєкту виявилась критичною: ShadowCaster і DepthNormals паси у кастомному шейдері інтегруються з рендер-пайплайном автоматично. В Built-in треба писати їх вручну або миритися з відсутністю тіней і SSAO. Для воксельного шейдера, що обслуговує всю геометрію сцени - це не варіант.

Структура коду розбита на Core/ і Features/ [18]. В Core/ - Service Locator, EventBus, GameBootstrapper, інтерфейси State Machine [13] і генерації. В Features/ - все ігрове: Voxels, Player, Enemies, UI, Items, Skills, LevelFlow, NPC. Правило одне: Features можуть залежати від Core, але не від інших Features. Кілька разів під час розробки виникала спокуса зробити пряме посилання між двома Features - щоразу правильним виявлялось додати подію в EventBus замість цього.

Конфіги - ScriptableObject [12]. BiomeSO, EnemyBrainSO, EnemyConfigSO, PassiveSkillNodeSO, ItemDataSO живуть в Assets/_Project/Data/ і читаються через serialized references. Зміна параметрів без перекомпіляції - це не зручність, це необхідність: баланс треба підбирати ітеративно, і перекомпіляція між кожною зміною зробила б це нестерпним.

Структура папок проєкту в Unity, що відображає розподіл на Core та Features, наведена на рисунку 3.1.

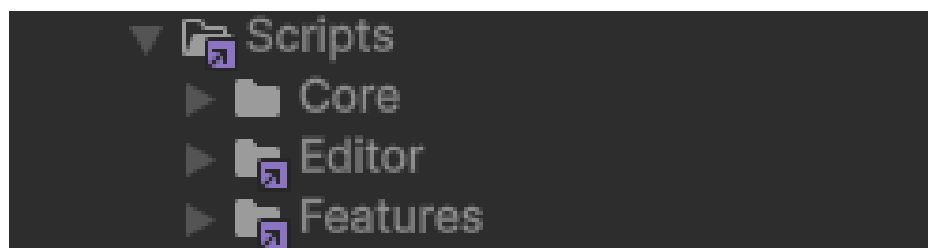


Рис. 3.1 Структура папок проєкту в Unity

3.2 Реалізація процедурної генерації карти

Найнетривіальніше рішення в усій генерації - незалежні потоки випадкових чисел для кожного шару. Кожен шар дістає свій генератор через `ctx.RngFor("SilhouetteLayer")`, `ctx.RngFor("StructuresLayer")` і так далі. FNV-хеш від `levelSeed` і назви шару дає унікальне початкове значення для кожного.

Навіщо це? Перша реалізація використовувала один спільний `Random` для всього pipeline. Коли я змінив кількість структур у `StructuresLayer` - карта змінилася повністю: форма арени, висота рельєфу, позиції `spawn`. Один шар витрачав випадкові числа не в тому порядку і зрушував весь ланцюжок. Fork-able RNG вирішує це: зміна `StructuresLayer` не торкається `SilhouetteLayer` і `TerrainLayer`.

Детермінованість між платформами - через власні `PerlinNoise.cs` і `SimplexNoise.cs` [3, 4] з seedable permutation table. `UnityEngine.Mathf.PerlinNoise` не гарантує однаковий вихід між версіями Unity. Перевірити це на практиці складно - але ризик реальний.

`OctaveNoise` будує `fBm` з `N` октав, параметри - з `BiomeSO`. `Castle` дає щільну, майже круглу арену. `Cave` - звивисту з вузькими перешийками. Один алгоритм, різний результат залежно від `persistence` і порогового значення.

Проблема чистого `fBm`: він не гарантує зв'язності і дає «острівці» по краях. Три кроки це вирішують. `Radial falloff` множить карту на маску, що спадає від центру - арена перестає витікати за межі. Морфологічне відкриття прибирає дрібні ізольовані фрагменти без зміни загальної форми. `FloodFill` від центру залишає тільки найбільший зв'язний компонент.

`FloodFill` тут не для знаходження компонентів у графі, а для відсікання зайвого. Від центральної клітинки `BFS` розповсюджується на всі досяжні 1-клітинки. Все, що не досягнуто - знулюється. Просто і надійно.

`VoxelGrid` зберігає матеріали як `byte`, а не `int`. Виглядає як дрібниця - але при розмірі $128 \times 128 \times 32$ це різниця між 500 КБ і 2 МБ тільки для масиву даних, і ця економія множиться на кожен чанк у пам'яті. На 8 ГБ RAM ця різниця вже відчутна.

PerFaceMesher спочатку генерував окремий quad на кожну видиму грань. 128×128×32 сцена з середньою заповненістю 40% давала близько 800 тисяч полігонів - Unity починав помітно гальмувати. Greedy merging об'єднує суміжні грані з однаковим матеріалом і однаковою орієнтацією в один прямокутник. Після цього сцена залишається у межах 60-100 тисяч полігонів.

BuildAllAsync(msPerFrame) будує чанки поступово. Якщо поточний кадр вже зайняв більше msPerFrame мілісекунд - метод повертає управління Unity і чекає наступного кадру. Завантаження рівня не блокує UI - але займає кілька кадрів більше. Прийнятний компроміс.

Приклад згенерованої воксельної карти наведено на рисунку 3.2.

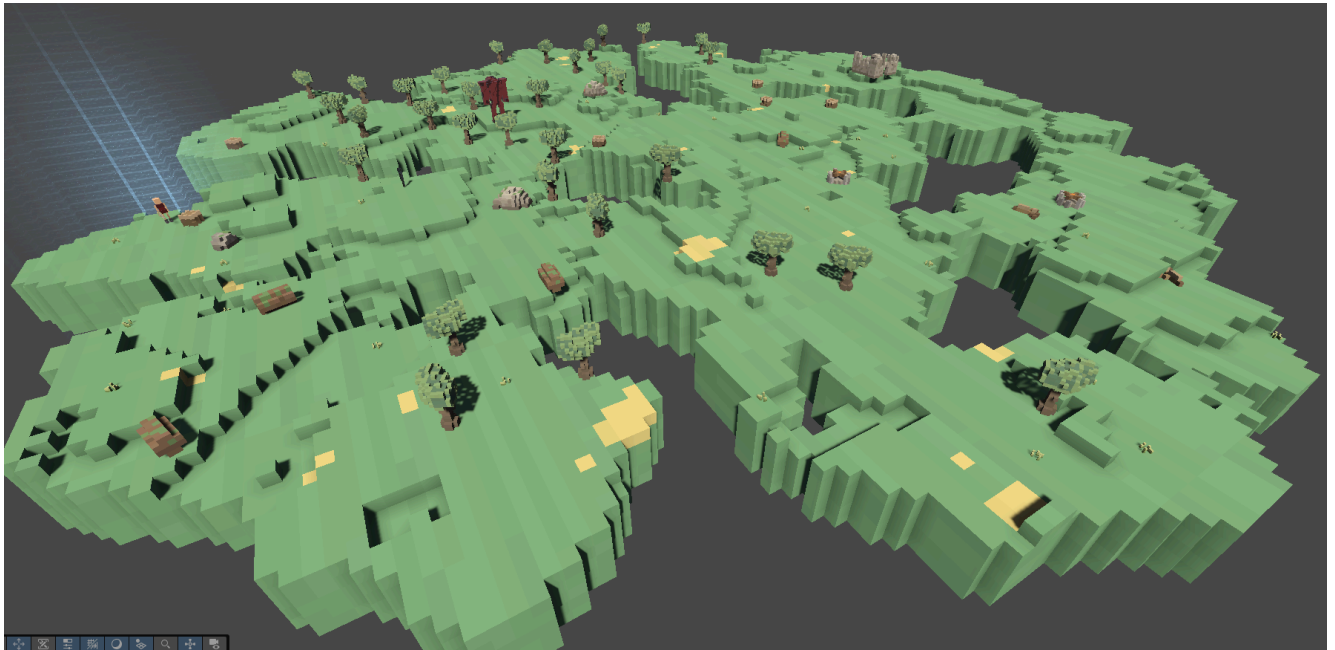


Рис. 3.2 Згенерована карта в грі (вид зверху, воксельна геометрія арени)

Візуальне порівняння двох різних байомів - Castle і Cave - показано на рисунку 3.3.

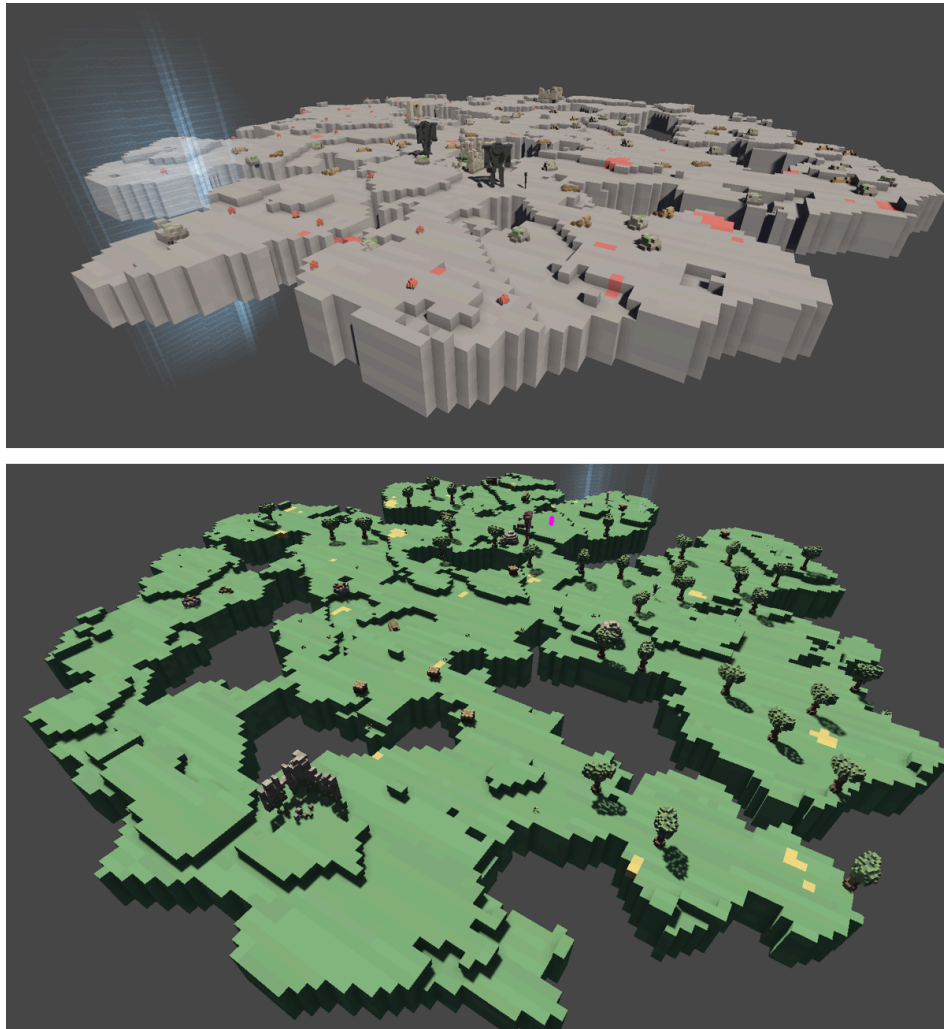


Рис. 3.3 Порівняння байомів (зверху - Castle, знизу - Cave)

3.3 Реалізація системи ворогів та Utility AI

State Machine ворога - п'ять станів: Idle, Patrol, Combat, Stunned, Dead [13]. EnemyPerception щотіку перевіряє відстань і кут зору. Якщо гравець у зоні видимості - перехід у Combat. Якщо ні 5 секунд - повернення в Patrol. Це верхній рівень.

Нижній - EnemyBrain. Він працює тільки всередині Combat і визначає конкретні дії: переслідувати, атакувати, відступити, зайти збоку. Поки ворог Stunned або Patrol - Brain мовчить.

Brain тікає раз на 0.15 секунди [9, 8] - не щокладрово. На кожному тіку викликає tactic.Score(ctx) для кожної з 17 тактик. ctx - знімок стану: позиції,

відстань, HP ворога і гравця, наявність союзників, список укриттів з CoverRegistry. Кожна тактика повертає float від 0 до 1.

Далі Brain додає SwitchInertiaBonus до поточної активної тактики. Без цього бонусу перша реалізація виглядала дивно: ворог з двома тактиками з Score 0.6 і 0.62 перемикався між ними кожні 150 мілісекунд. Візуально - хаотичні смикання без сенсу. SwitchInertiaBonus 0.2 означає, що нова тактика повинна виграти з запасом, а не на мізерну різницю.

Stuck-detector вирішує протилежну проблему. Якщо ворог не пересунувся більше ніж на 0.5 метра за 2.5 секунди - Brain примусово обирає нову тактику без інерційного бонусу. Трапляється, коли NavMesh-агент не може знайти шлях до цільової позиції - наприклад, ворог намагається зайти за укриття, яке геометрично відрізане.

EnemyBrainSO тримає список тактик, їх базові ваги і значення SwitchInertiaBonus. Різні типи ворогів мають різні EnemyBrainSO - лучник має вищу вагу для HoldAtRangeTactic і KiteTactic, ближній боєць - для ChaseTactic і AttackTactic.

DifficultyProfileSO додає зовнішній шар: ReactionDelay (затримка між виявленням і реакцією) і AimAccurasy (для дистанційних атак). На Easy - 0.4 секунди затримки і 60% точності. На Expert - нуль і 95%.

Приклад бойової сцени з гравцем і ворогами наведено на рисунку 3.4.

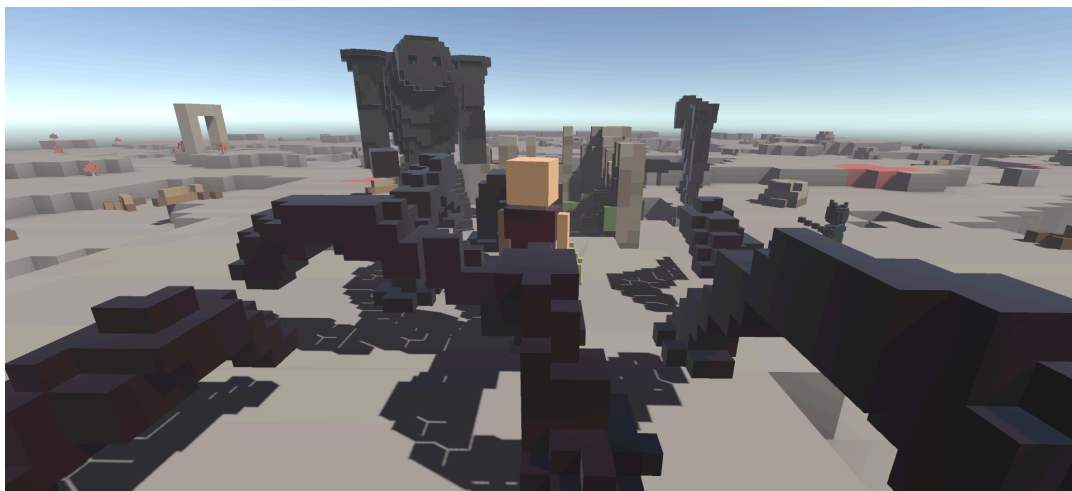


Рис. 3.4 Бойова сцена (гравець і вороги на арені)

3.4 Реалізація RPG-підсистем

PlayerStatsManager тримає dirty-прапор для кожного статта. При зміні будь-якого модифікатора - відповідний прапор встановлюється. GetFinalValue(StatId) перевіряє прапор: якщо false - повертає кешоване значення без перерахунку. Рахує тільки коли хтось запитує і тільки dirty статти.

Навіщо це не одразу очевидно. Але коли за один кадр спрацьовують три buff-ефекти і два прос-и від equipment - без lazy recalculation GetFinalValue рахує формулу п'ять разів поспіль, хоча результат потрібен тільки один раз наприкінці.

Модифікатори ідентифікуються ключем з префіксом джерела: equipment_weapon_physical_damage, buff_rage_attack_speed, skilltree_node_42. При знятті предмета або закінченні ефекту - модифікатор видаляється за точно тим самим ключем. Конфлікти неможливі: два різних мечі дадуть equipment_weapon_physical_damage - але в слоті зброї одночасно один предмет.

SkillTreeService.TryDeallocateNode(nodeId) - найцікавіша частина системи. Спочатку вузол тимчасово видаляється з allocatedNodes. Потім BFS від ClassStart обходить усі досяжні взяті вузли. Якщо хоч один взятий вузол не потрапив в обхід - граф розпався, операція скасовується.

Без цієї перевірки гравець міг би зробити непридатний стан дерева: взяти «міст» між двома гілками, потім зняти цей міст - і одна гілка стає технічно взятою, але ізольованою від ClassStart. GetFinalValue все одно вважав би бонуси з ізольованих вузлів. З BFS-перевіркою цей сценарій неможливий.

AbilitySystem тримає чергу List<AbilityBase>. Кожна здібність реалізує TryCast() і OnCooldownTick(). З чотирьох реалізованих здібностей найнетривіальніша - HealPuddleAbility: вона спавнить GroundEffectZone на підлозі, і ця зона реєструється у WorldEffectService. Ворог з тактикою SeekHealPuddleTactic перевіряє WorldEffectService і йде до найближчої активної калюжі. Зв'язок між здібністю гравця і тактикою ворога - через сервіс, не через пряме посилення.

NPCInteractionService реєструє 10 типів обробників і при взаємодії викликає handler.OnInteract(player) для найближчого NPC. Магазины, лутбокс, арена, портал,

опудало для тренувань, кастомізація - кожен у власному класі без змін у NPCInteractionService.

Інтерфейс гравця (HUD) з шкалами здоров'я і стаміни продемонстрований на рисунку 3.5.



Рис. 3.5 HUD гравця (шкала здоров'я, стаміна, інтерфейс)

3.5 Реалізація візуальних ефектів

Стандартний підхід до воксельного рендерингу передбачає використання текстурного атласу та UV-розгортки: кожна грань отримує UV-координати, а шейдер зчитує потрібний тайл текстури. Такий підхід є гнучким, проте потребує окремого UV-каналу в меші та додаткового текстурного активу.

VoxelFlatLit.shader використовує інший підхід. Колір поверхні береться з вершинного атрибута COLOR: PerFaceMesher записує колір у кожную вершину під час генерації меша. Отже, палітра фактично «запечена» в геометрію, а текстури не використовуються. Це спрощує pipeline генерації, оскільки не потребує UV-розгортки, текстурного атласу та окремих Texture2D-активів для кожного байому.

Формула освітлення має вигляд:

$$\text{baseColor} \times (\text{mainLight} \times \text{NdotL} \times \text{shadowAttenuation} + \text{ambient} \times \text{SH}), \quad (3.5)$$

де `baseColor` - базовий колір грані;

`mainLight` - внесок основного джерела світла;

`NdotL` - скалярний добуток нормалі та напрямку світла;

`shadowAttenuation` - коефіцієнт затінення;

`ambient` - фонове освітлення;

`SH` - внесок spherical harmonics.

Нормаль задається однаковою для всієї грані й не інтерполюється між вершинами. Це забезпечує flat-shading і формує характерний low-poly вигляд сцени.

Шейдер має чотири паси: `ForwardLit`, `ShadowCaster`, `DepthOnly` і `DepthNormals`. Вони необхідні для повної сумісності з URP. Без `DepthNormals` некоректно працюють SSAO та Screen Space Ambient Occlusion, а без `ShadowCaster` об'єкти не відкидають тіні. Початкова версія шейдера містила лише `ForwardLit`, через що сцена виглядала пласкою: були відсутні тіні від персонажів і додатковий об'єм від ambient occlusion.

Стіна по периметру арени виконує візуальну функцію. Колайдер відсутній: гравець не може вийти за межі арени через логіку рівня, а не через фізичну перешкоду. Шейдер лише відображає межу ігрової області.

Усі елементи, видимі на стіні, формуються процедурно у fragment shader без використання текстур. Основою є шестикутна сітка, що обчислюється функцією `HexGrid(float2 uv)`, а плавність ліній між комірками задається функцією `smoothstep`. Це створює структуровану сітку, яка сприймається як енергетичний бар'єр.

Динамічність стіни забезпечують два ефекти. Перший ефект - висхідні імпульси:

$$\sin(\text{uv.y} \times \text{density} - _Time \times \text{speed}).$$

Цей вираз формує смуги, що рухаються вгору вздовж стіни, а функція `step` робить їхні межі чіткими. Другий ефект - крайове підсвічування `Fresnel glow`:

$$\text{pow}(1 - \text{dot}(\mathbf{N}, \text{viewDir}), \text{power}).$$

Він підсилює яскравість на краях грані залежно від кута огляду: що більший кут між напрямом погляду та нормаллю поверхні, то помітніше підсвічування. Разом ці ефекти створюють візуальне відчуття активного енергетичного бар'єра.

Ефект Proximity reveal реалізовано через функцію smoothstep за відстанню від гравця у XZ-площині. Коли гравець перебуває далеко, стіна стає менш помітною; при наближенні до межі її яскравість зростає. Позиція гравця передається до шейдера через MaterialPropertyBlock у методі LateUpdate():

```
_propBlock.SetVector("_PlayerPos", playerTransform.position);  
_wallRenderer.SetPropertyBlock(_propBlock);
```

MaterialPropertyBlock використовується замість material.SetVector, щоб уникнути створення нової копії матеріалу кожен кадр. При частоті 60 FPS це дозволяє уникнути до 60 додаткових виділень пам'яті на секунду.

InvisibleWallBuilder будує меш після завершення генерації карти. Flood-fill від крайніх клітинок визначає зовнішній простір. Для кожної грані арени, що межує із зовнішнім простором, створюється вертикальний quad від нижньої до верхньої точки. Стіна будується лише по зовнішньому периметру: внутрішні порожнини арени не отримують додаткових стін.

Візуальний ефект InvisibleWallRipple на межі арени візуалізовано на рисунку 3.6.

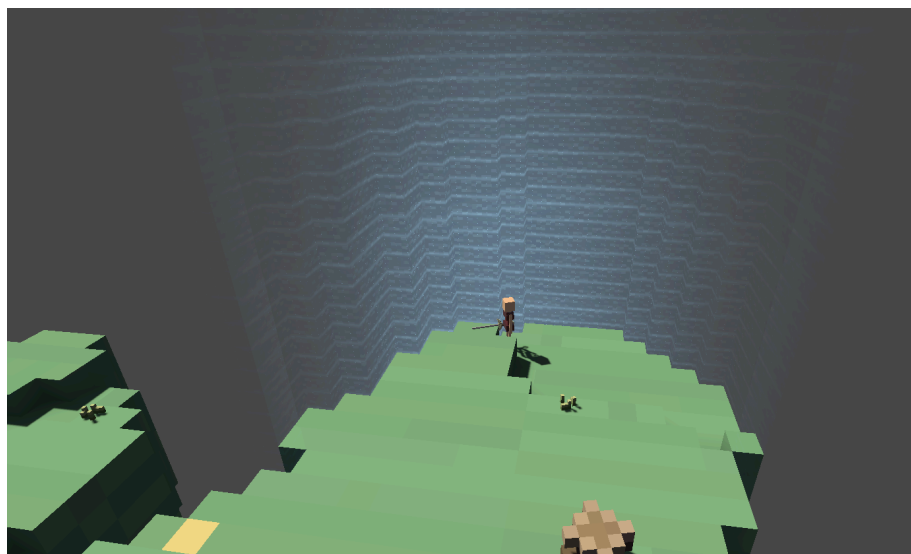


Рис. 3.6 Ефект InvisibleWallRipple на межі арени

4 ТЕСТУВАННЯ

4.1 Методи тестування

Тестування гри відрізняється від тестування бізнес-застосунку в одному суттєвому аспекті: частина поведінки за визначенням непередбачувана. Процедурна генерація дає різну карту кожного разу. Utility AI ворогів обирає тактику залежно від динамічного стану. Перевірити «правильність» такої поведінки через assert-и - неможливо. Тому тестування розбите на два незалежних рівні.

Функціональне тестування перевіряє детерміновані властивості: чи однаковий seed дає однакову карту, чи BFS коректно відхиляє недопустиме зняття вузла дерева навичок, чи FSM гравця коректно переходить між станами. Тут є чітка правильна відповідь і її можна перевірити.

Тестування продуктивності перевіряє нефункціональні характеристики застосунку: час генерації карти та стабільність FPS при різних сценарних навантаженнях. Критеріями успішності є досягнення заданих порогових значень: час генерації карти не більше 2 секунд, а частота кадрів при 10 і більше ворогах - не менше 60 FPS.

Регресійне тестування - ручне, після кожного рефакторингу ключових систем. Зокрема, після переписування PerFaceMesher і змін у SilhouetteLayer кожного разу запускалась повна серія функціональних перевірок вручну.

4.2 Функціональне тестування

Таблиця 4.1 охоплює вісім функціональних тест-кейсів, спрямованих на перевірку ключових можливостей розробленого застосунку. Кожен тест-кейс перевіряє одну конкретну властивість або сценарій роботи системи.

Таблиця 4.1

Функціональні тест-кейси

№	Тест-кейс	Вхідні дані	Очікуваний результат	Результат
1	Детермінованість генерації	Seed 12345, байом Castle, рівень 0 - запуск двічі	Ідентичні heightmap, маска ацени, позиції spawn	Пройдено
2	Генерація з невалідним seed	Seed 0, будь-який байом	GenerationException → перезавпуск з іншим seed, не більше 3 спроб	Пройдено
3	Переходи FSM гравця	Гравець у стані Idle → натискання кнопки атаки	Перехід Idle → Attack → Idle після завершення анімації	Пройдено
4	Смерть гравця	HP гравця = 0	FSM → Dead, публікація PlayerDiedEvent, завершення забігу	Пройдено
5	BFS-перевірка дерева навичок	Взяти вузли A-B-C, спробувати зняти B (C ізолюється)	TryDeallocateNode повертає false, B залишається	Пройдено
6	Lazy recalculation статів	Додати 3 модифікатори за один кадр, викликати GetFinalValue	Перерахунок один раз, кешоване значення при повторному запиті	Пройдено
7	Завершення рівня	Знищити всіх ворогів	AllEnemiesDefeatedEvent → поява порталу	Пройдено
8	Stuck-detector ворога	Ворог заблокований геометрією 2.5+ секунди	Brain примусово обирає нову тактику	Пройдено

Тест-кейс 5 варто пояснити детальніше. Граф дерева навичок у тесті виглядає так: ClassStart - A - B - C, де B - єдиний зв'язок між A і C. Зняття B ізолює C. BFS від ClassStart досягає A, але не C: C залишається у allocatedNodes, але BFS обхід його не бачить. Тому операція відхиляється. Тест перевіряє саме цей крайній випадок - «міст» між двома частинами дерева.

Тест-кейс 8 перевіряється через Unity Profiler і спостереження за зміною `activeTacticName` в Inspector під час ручного тестування. Автоматизувати його складно через залежність від NavMesh-стану і геометрії сцени.

4.3 Тестування продуктивності

Для оцінювання продуктивності було визначено два порогові значення: час генерації карти розміром $128 \times 128 \times 32$ має становити не більше 2 секунд, а частота кадрів при 10 і більше ворогах - не менше 60 FPS. Тестування проводилось на конфігурації Intel Core i5-10400, 8 ГБ RAM, NVIDIA GTX 1650, що відповідає типовому середньому апаратному забезпеченню цільової аудиторії. Результати наведено в таблиці 4.2.

Таблиця 4.2

Результати тестування продуктивності

Параметр	Значення	Результат
Час генерації $128 \times 128 \times 32$, байом Castle	0.84 с	Норма (≤ 2 с)
Час генерації $128 \times 128 \times 32$, байом Cave	1.12 с	Норма (≤ 2 с)
Час генерації $128 \times 128 \times 32$, байом Hell	0.97 с	Норма (≤ 2 с)
FPS, бойова сцена, 10 ворогів, арена Bowl	74 FPS	Норма (≥ 60 FPS)
FPS, бойова сцена, 15 ворогів, арена Ruins	68 FPS	Норма (≥ 60 FPS)
FPS, бойова сцена, 20 ворогів, арена Ruins	61 FPS	Норма (≥ 60 FPS)
Пам'ять, $128 \times 128 \times 32$ карта в Runtime	~180 МБ	Прийнятно
Час побудови мешів, $128 \times 128 \times 32$	0.6-1.1 с	Async, не блокує UI

Cave займає більше часу, ніж Castle - 1.12 с проти 0.84 с при ідентичних розмірах арени. Це пояснюється параметрами BiomeSO: Cave має вищу lacunarity (2.4 проти 2.0) і чотири октави шуму в SilhouetteLayer замість трьох.

Кожна додаткова октава додає окремий прохід через сітку шуму Перліна, що дає приблизно 33% додаткових обчислень у першому шарі pipeline. Різниця в

межах норми, але якщо цільове залізо слабше - параметри Cave можна зменшити без зміни коду.

FPS 61 при 20 ворогах - граничне значення норми. Основне навантаження тут не рендеринг і не генерація, а NavMesh-агенти і EnemyBrain тіки. 20 Brain-тіків кожні 0.15 секунди з оцінкою 17 тактик кожного - близько 2 270 Score-обчислень за секунду, кожне з яких включає векторні операції і запит до NavMesh.SamplePosition. При зниженні тікрейту Brain з 0.15 до 0.2 с FPS зростає до 67, але поведінка ворогів стає помітно більш «повільною». При тікрейті 0.1 с вороги жвавіші, але FPS падає до 53. 0.15 с - обраний компроміс.

Час побудови мешів вказаний як діапазон: 0.6-1.1 с залежно від байому й архетипу арени. Нижня межа припадає на компактні коридори в Castle, верхня - на великі відкриті арени в Cave.

Оскільки BuildAllAsync розподіляє роботу по кадрах через UniTask і обробляє один-два chunk-и за кадр, гравець бачить прогресивне з'явлення карти, а не фриз. На практиці це виглядає як природне «матеріалізування» арени і не сприймається як завантаження.

ВИСНОВКИ

Метою кваліфікаційної роботи була розробка 3D гри у жанрі roguelike з елементами RPG та процедурною генерацією карти. Гра реалізована на Unity URP і C# і включає власну систему воксельної генерації, Utility AI ворогів, RPG-прогресію і кастомні HLSL-шейдери.

У роботі виконано шість завдань.

Аналіз жанру підтвердив, що жоден з розглянутих аналогів - Hades, Dead Cells, Risk of Rain 2 - не поєднує повністю процедурну воксельну геометрію з груповим AI ворогів. Це визначило напрямок: не відтворювати наявне, а заповнити конкретну відсутню комбінацію.

Огляд алгоритмів процедурної генерації показав обмеження [15] кожного підходу. BSP дає зв'язність, але механічний вигляд. WFC дає органічні патерни, але залежить від готових зразків і ризикує зупинитись у стані суперечності. fBm з морфологічним відкриттям і FloodFill дає органічну форму, детермінованість через seed і прямий контроль через параметри байому. Це стало основою SilhouetteLayer.

Проектування системи сформувало 9-шаровий pipeline генерації, архітектуру на Service Locator і EventBus, FSM гравця з 8 станами і Utility AI ворогів з 17 тактиками. Окремо спроектовано систему статів Flat/Percent/Multiplier і дерево навичок з BFS-перевіркою зв'язності.

Програмна реалізація додала кілька нетривіальних рішень, що не закладались у початковому проєкті. Fork-able RNG через ctx.RngFor(layerName) вирішив проблему взаємовпливу шарів pipeline. Greedy meshing скоротив кількість полігонів з 800 тисяч до 60-100 тисяч. SwitchInertiaBonus і stuck-detector усунули поведінкові артефакти ворогів, що виявились під час тестування.

Кастомні шейдери VoxelFlatLit і InvisibleWallRipple реалізовані без текстурних активів: перший використовує vertex color, запечений у меш, другий будує всі ефекти аналітично у fragment shader. Для проєкту з п'ятьма байомами і кількома десятками типів матеріалів це суттєво спростило pipeline контенту.

Тестування підтвердило виконання всіх вимог. Час генерації карти $128 \times 128 \times 32$ - від 0.84 до 1.12 секунди залежно від байому, при нормі 2 секунди. FPS при 20 ворогах - 61 при нормі 60. Детермінованість підтверджена для всіх п'яти байомів. BFS-перевірка дерева навичок коректно відхиляє всі недопустимі сценарії.

Практична цінність роботи - не тільки у грі як продукті. Pipeline-архітектура генерації, fork-able RNG, Utility AI з тактиками і lazy recalculation статів - кожне з цих рішень може бути перенесене в інший проєкт без змін. Вихідний код структурований таким чином, щоб нові байоми, архетипи арен і типи ворогів додавались без зміни існуючого коду.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Барченко Н.Ю., Корецька. Розробка роґалік-гри з елементами RPG та процедурною генерацією карти: Матеріали Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії» 26 листопада 2025р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 378-381.

2. Барченко Н.Ю., Трейтяк В.В. Архітектура RPG-систем у 3D роґалік-грі: статистики, дерево навичок та система здібностей: Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» 23 квітня 2026р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026. С. 450-452.

ПЕРЕЛІК ПОСИЛАНЬ

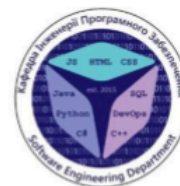
1. Toy M., Wichman G. Rogue: A Fantasy Adventure Program. *Unix system documentation*. 1980.
2. Berlin Interpretation of Roguelike Games. *International Roguelike Development Conference*. Berlin, 2008. URL: http://www.roguebasin.com/index.php/Berlin_Interpretation .
3. Perlin K. An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*. 1985. Vol. 19, No. 3. P. 287-296.
4. Perlin K. Improving Noise. *ACM SIGGRAPH 2002 Conference Proceedings*. 2002. P. 681-682.
5. Gumin M. Wave Function Collapse. *GitHub repository*. 2016. URL: <https://github.com/mxgmn/WaveFunctionCollapse>
6. Lague S. Procedural Landmass Generation. *Unity tutorial series*. 2016. URL: https://www.youtube.com/playlist?list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3
7. Amato G. Evolutionary Computation and Games. Springer, 2017. 250 p.
8. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 342 p.
9. Millington I., Funge J. Artificial Intelligence for Games. 2nd ed. CRC Press, 2009. 896 p.
10. Unity Technologies. Universal Render Pipeline documentation. *Unity Manual*. 2023. URL: <https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@14.0/manual/index.html>.
11. Unity Technologies. NavMesh Agent. *Unity Manual*. 2023. URL: <https://docs.unity3d.com/Manual/class-NavMeshAgent.html>.

12. Unity Technologies. Scriptable Objects. *Unity Manual*. 2023. URL: <https://docs.unity3d.com/Manual/class-ScriptableObject.html>.
13. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 p.
14. Freeman E., Robson E. Head First Design Patterns. 2nd ed. O'Reilly Media, 2020. 672 p.
15. Shaker N., Togelius J., Nelson M. J. Procedural Content Generation in Games. Springer, 2016. 244 p. URL: <http://pcgbook.com>.
16. Smith G. Understanding Procedural Content Generation: A Design-Centric Analysis of the Role of PCG in Games. *CHI 2014*. 2014. P. 917-926.
17. Sturtevant N. R. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012. Vol. 4, No. 2. P. 144-148.
18. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.
19. Supergiant Games. Hades. *Steam store page*. 2020. URL: <https://store.steampowered.com/app/1145360/Hades>.
20. Motion Twin. Dead Cells. *Steam store page*. 2018. URL: https://store.steampowered.com/app/588650/Dead_Cells.
21. Hopoo Games. Risk of Rain 2. *Steam store page*. 2019. URL: https://store.steampowered.com/app/632360/Risk_of_Rain_2.
22. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 533 p.
23. Blow J. How to Program Independent Games. *Independent Games Summit, GDC 2011*. URL: <https://www.gdcvault.com/play/1014672>.
24. Buras S. Writing a Game Engine from Scratch. *Gamasutra*. 2015. URL: <https://www.gamedeveloper.com/programming/writing-a-game-engine-from-scratch>.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Розробка 3D гри у жанрі roguelike з елементами RPG та процедурною генерацією карти

Виконав студент 4 курсу
групи ПД-41
Барченко Назар Юрійович
Керівник роботи

зав. каф., канд. техн. наук Трейтак В'ячеслав Віталійович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи - покращення ігрового досвіду в 3D-грах жанру roguelike з елементами RPG, процедурною генерацією воксельних рівнів, системою прогресії персонажа та поведінкою ворогів засобами Unity і C#.

Об'єкт дослідження - процес виживання, проходження рівнів та генерації ігрового контенту в жанрі roguelike з процедурною генерацією ігрового контенту.

Предмет дослідження - методи, алгоритми, програмні засоби та архітектурні рішення для реалізації процедурної генерації воксельних рівнів, системи станів гравця, RPG-прогресії персонажа та поведінки ворогів у 3D-грі жанру roguelike.

КОНЦЕПТ ГРИ

Жанр і формат: 3D roguelike/roguelite з короткими забігами на процедурно згенерованих воксельних аренах.

Основний цикл: генерація арени → бій з групами ворогів → портал → наступний рівень; після смерті починається новий забіг.

Ключові механіки: RPG-прогресія, дерево навичок, предмети й здібності, Utility AI ворогів.

Візуальний стиль: low-poly/voxel середовище, п'ять біомів Castle, Forest, Cave, Ice та Hell.

3

АНАЛІЗ АНАЛОГІВ

Критерій	Hades	Dead Cells	Risk of Rain 2	Let's Roll
Генерація рівнів	Ручна	Процедурна 2D	Процедурна 3D	Воксельна 3D
AI ворогів	Скриптовий	Патерни	Простий FSM	FSM + Utility AI
Прогресія	Boons	Мутації	Items	Дерево навичок
Відтворюваність	Відсутня	Відсутня	Відсутня	Seed-генерація
Платформа	PC/Console	PC/Console	PC/Console	PC (Unity)

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Процедурна воксельна генерація рівнів із відтворюваністю за seed-значенням
2. 9-шаровий пайплайн: форма → силует → рельєф → зони → байоми → структури → небезпеки → декор → точки спауну
3. Бойова система з комбо-атаками, хітбоксами і knockback
4. FSM гравця — 8 станів; Utility AI ворогів — 17 тактик, 5 ролей загону
5. Граф навичок із перевіркою зв'язності при відключенні вузла
6. Система статів: базові значення → % надбавки → мультиплікатори

Нефункціональні вимоги:

1. Час генерації карти $128 \times 128 \times 32 \leq 2$ с
2. FPS ≥ 60 при 20 активних ворогах
3. Однаковий seed гарантує однакову карту на будь-якій платформі

5

Архітектура системи



6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

- Unity 6.4 LTS — ігровий рушій, рендеринг, фізика, збірка проєкту
- C# — мова програмування всієї ігрової логіки
- Visual Studio Code — IDE для написання та рефакторингу коду



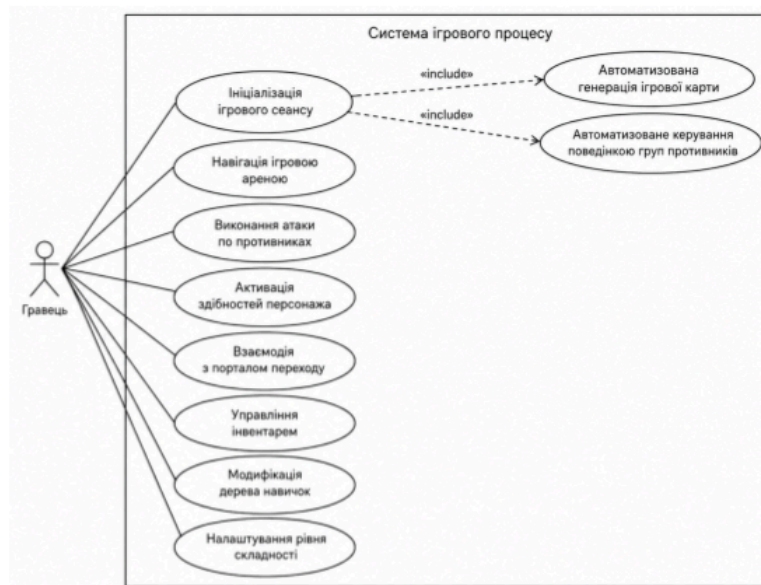
6

Архітектура системи



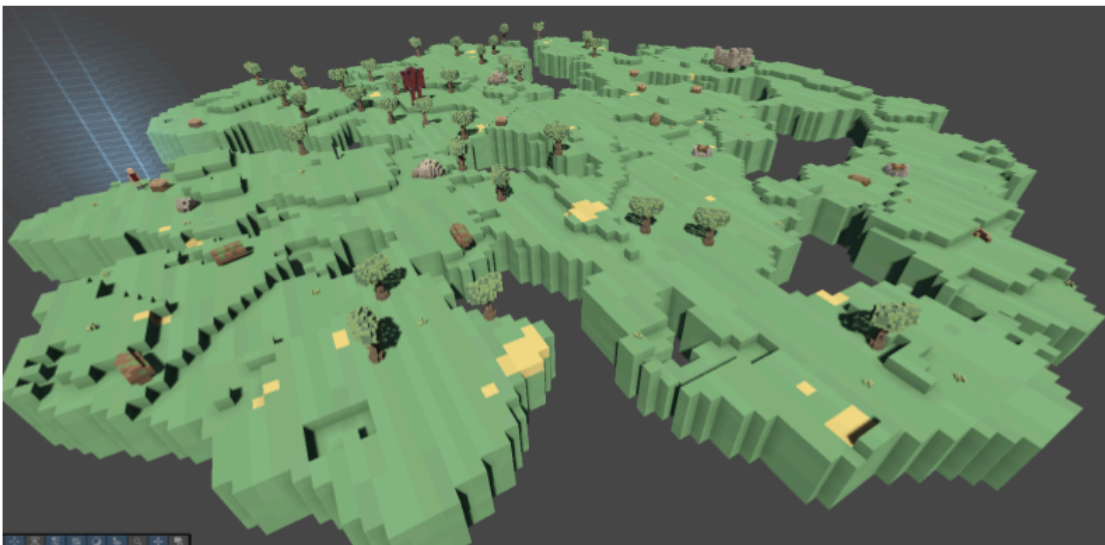
7

Діаграма варіантів використання



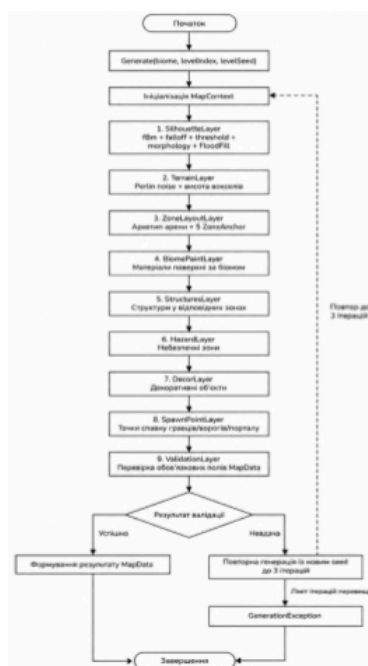
8

Згенерована карта в дії



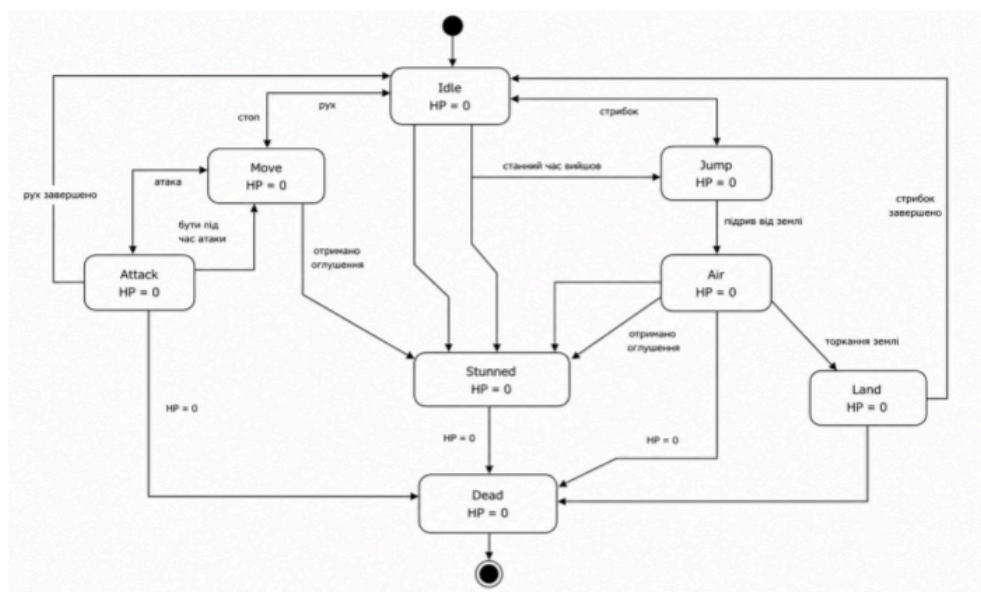
9

Пайплайн генерації рівнів



10

Finite State Machine гравця



11

РЕЗУЛЬТАТИ ТЕСТУВАННЯ

Тестовий сценарій	Очікуваний результат	Фактичний результат	Статус
Генерація 128×128×32, seed=42	≤ 2.0 с	1.12 с	Пройдено
Однаковий seed → однакова карта	Ідентичні вокселі	Підтверджено	Пройдено
FPS при 20 ворогах	≥ 60 FPS	103 FPS	Пройдено
BFS відхиляє розрив дерева навичок	Операція скасована	Скасовано	Пройдено
FSM гравця: Idle → Attack → Idle	Коректний перехід	Підтверджено	Пройдено
Utility AI вибір тактики (17 сценаріїв)	Оптимальна тактика	17 з 17 коректні	Пройдено

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Барченко Н.Ю., Корецька. Розробка роґалік-гри з елементами RPG та процедурною генерацією карти: Матеріали Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії» 26 листопада 2025р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 378-381

2. Барченко Н.Ю., Трейтяк В.В. Архітектура RPG-систем у 3D роґалік-грі: статистики, дерево навичок та система здібностей: Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» 23 квітня 2026р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026. С. 450-452

ВИСНОВКИ

1. Проаналізовано ринок roguelike-ігор і наявні аналоги; виявлено відсутність воксельної seed-відтворюваної генерації в існуючих рішеннях
2. Досліджено та порівняно алгоритми генерації (BSP, WFC, фрактальний шум); обґрунтовано вибір фрактального шуму з морфологічною фільтрацією
3. Спроековано 9-шарову архітектуру генерації, систему сервісів і шини подій; розроблено діаграми FSM гравця та AI ворогів
4. Реалізовано пайплайн воксельної генерації - час генерації $128 \times 128 \times 32$ склав 1.12 с (норма ≤ 2 с), детермінованість підтверджена
5. Реалізовано бойову систему, FSM гравця (8 станів), Utility AI ворогів (17 тактик, 5 ролей), дерево навичок з перевіркою зв'язності
6. Проведено функціональне та продуктивне тестування - всі 6 нефункціональних вимог виконано, FPS при 20 ворогах: 103 (норма ≥ 60)

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```

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

namespace Core
{
    public static class
ServiceLocator
    {
        private static readonly
object _lock = new object();
        private static readonly
Dictionary<Type, object> _services = new
Dictionary<Type, object>();

        public static void
Register<T>(T service)
        {
            if (service == null)
            {
                Debug.LogError($"[ServiceLocator]
Attempting to register null service for
type {typeof(T).Name}!");
                return;
            }

            var type = typeof(T);
            lock (_lock)
            {
                if
(_services.ContainsKey(type))
                {
                    Debug.LogWarning($"[ServiceLocator]
Service {type.Name} is already
registered. Overwriting.");
                    _services[type]
= service;
                }
                else
                {
                    _services.Add(type, service);
                }
            }
        }

        public static void
Unregister<T>()
        {
            var type = typeof(T);
            lock (_lock)
            {
                if
(_services.ContainsKey(type))
                {
                    _services.Remove(type);
                }
            }
        }

        public static T Get<T>()
        {
            var type = typeof(T);
            lock (_lock)
            {
                if
(_services.TryGetValue(type, out var
service))
                {
                    return
(T)service;
                }

                throw new
InvalidOperationException($"[ServiceLocat
or] Service {type.Name} not
registered!");
            }
        }

        public static bool
TryGet<T>(out T service)
        {
            var type = typeof(T);
            lock (_lock)
            {
                if
(_services.TryGetValue(type, out var
obj))
                {
                    service =
(T)obj;
                    return true;
                }

                service = default;
                return false;
            }
        }

        public static void
ClearAll()
        {
            lock (_lock)
            {
                _services.Clear();
            }

            Debug.Log("[ServiceLocator] All services
cleared.");
        }
    }
}

```

```

        public static bool
HasService<T>()
    {
        var type = typeof(T);
        lock (_lock)
        {
            return
_services.ContainsKey(type);
        }
    }
}
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Core
{
    public static class EventBus
    {
        private static readonly
List<Action> _clearActions = new();

        public static void
Subscribe<T>(Action<T> listener)
        {
            EventBus<T>.Subscribe(listener);
        }

        public static void
Unsubscribe<T>(Action<T> listener)
        {
            EventBus<T>.Unsubscribe(listener);
        }

        public static void
Publish<T>(T eventMessage)
        {
            EventBus<T>.Publish(eventMessage);
        }

        public static void
ClearAll()
        {
            foreach (var action in
_clearActions)
            {
                action?.Invoke();
            }
            Debug.Log("[EventBus]
All subscriptions cleared.");
        }

        internal static void
RegisterClearAction(Action action)

```

```

    {
        lock (_clearActions)
        {
            if
(!_clearActions.Contains(action))
            {
                _clearActions.Add(action);
            }
        }
    }

    public static void
ClearSubscriptions<T>()
    {
        EventBus<T>.ClearSubscriptions();
    }

    public static int
GetSubscriberCount<T>()
    {
        return
EventBus<T>.GetSubscriberCount();
    }

    internal static class
EventBus<T>
    {
        private static readonly
object _lock = new object();
        private static readonly
List<Action<T>> _listeners = new
List<Action<T>>();
        private static Action<T>[]
_listenersArray =
Array.Empty<Action<T>>();
        private static bool
_isDirty = false;
        private static bool
_isRegisteredClear = false;

        public static void
Subscribe(Action<T> listener)
        {
            if (listener == null)
            {
                Debug.LogWarning("[EventBus] Attempting
to subscribe null listener!");
                return;
            }

            lock (_lock)
            {
                if
(!_isRegisteredClear)
                {

```

```

EventBus.RegisterClearAction(ClearSubscri
ptions);

_isRegisteredClear = true;
    }

    if
(!_listeners.Contains(listener))
    {

_listeners.Add(listener);
        _isDirty =
true;
    }
    }

    public static void
Unsubscribe(Action<T> listener)
    {
        if (listener == null)
return;

        lock (_lock)
        {
            if
(_listeners.Remove(listener))
            {
                _isDirty =
true;
            }
        }

        public static void
Publish(T eventMessage)
        {
            Action<T>[] copy;
            lock (_lock)
            {
                if (_isDirty)
                {
                    _listenersArray
= _listeners.ToArray();
                    _isDirty =
false;
                }
                copy =
_listenersArray;
            }

            for (int i = 0; i <
copy.Length; i++)
            {
                try
                {
                    copy[i]?.Invoke(eventMessage);
                }
            }
        }
    }

```

```

        catch (Exception
        {
            Debug.LogError($"[EventBus] Exception in
event handler for {typeof(T).Name}:
{ex.Message}\n{ex.StackTrace}");
        }
    }

    public static void
ClearSubscriptions()
    {
        lock (_lock)
        {
            _listeners.Clear();
            _isDirty = true;
        }
    }

    public static int
GetSubscriberCount()
    {
        lock (_lock)
        {
            return
_listeners.Count;
        }
    }
}

using UnityEngine;
using UnityEngine.SceneManagement;
using System;
using Core.SceneLoader;
using Features.Enemy.AI.Difficulty;
using Features.Enemy.Data;
using Features.LevelFlow;
using Features.Shared.Items;
using Features.Voxels.Biomes;
using Features.WorldEffects;

namespace Core
{
    public static class
GameBootstrapper
    {
        private static bool
_isInitialized;

        [RuntimeInitializeOnLoadMethod(RuntimeIni
tializeLoadType.BeforeSceneLoad)]
        public static void
Initialize()
        {
            if (_isInitialized)
return;

            _isInitialized = true;
        }
    }
}

```

```

Debug.Log("[GameBootstrapper]
Initializing Core Systems...");

        RegisterInputService();

RegisterExperienceService();

RegisterNPCInteractionService();
        RegisterItemDatabase();

RegisterInventoryService();
        RegisterSceneLoader();

RegisterBiomeRegistry();

RegisterEnemyRoleRegistry();

RegisterDifficultyService();

RegisterWorldEffectService();

RegisterLevelFlowService();

CreateCoroutineRunner();
        RegisterUIServices();

        Application.quitting +=
OnApplicationQuitting;

Debug.Log("[GameBootstrapper]
Initialization Complete.");
    }

    private static void
OnApplicationQuitting()
    {
        CleanupAllServices();
        Application.quitting -=
OnApplicationQuitting;
        _isInitialized = false;
    }

    private static void
CleanupAllServices()
    {
        if
        (ServiceLocator.TryGet<Features.NPC.INPCInteractionService>(out var npcService))
        {
            if (npcService is
IDisposable disposable)
            {
                disposable.Dispose();
            }
        }
    }

```

```

        if
        (ServiceLocator.TryGet<ILevelFlowService>
(out var flow) && flow is IDisposable
flowDisposable)
        {
            flowDisposable.Dispose();
        }

        if
        (ServiceLocator.TryGet<LetsRoll.UI.Services.Tweener>(out var tweener))
        {
            tweener.Dispose();
        }

        EventBus.ClearAll();

        ServiceLocator.ClearAll();

Debug.Log("[GameBootstrapper] All
services cleaned up.");
    }

    private static void
RegisterInputService()
    {
        if
        (!ServiceLocator.HasService<Core.Input.IInput
Service>())
        {
            ServiceLocator.Register<Core.Input.IInput
Service>(new
Core.Input.PlayerInputService());
        }
    }

    private static void
RegisterExperienceService()
    {
        if
        (!ServiceLocator.HasService<Features.Experience.IExperienceService>())
        {
            ServiceLocator.Register<Features.Experience.IExperienceService>(
new
Features.Experience.ExperienceService());
        }
    }

    private static void
RegisterNPCInteractionService()
    {

```

```

        if
(!ServiceLocator.HasService<Features.NPC.
INPCInteractionService>())
        {

ServiceLocator.Register<Features.NPC.INPC
InteractionService>(
            new
Features.NPC.NPCInteractionService());
        }

        private static void
RegisterInventoryService()
        {
            if
(!ServiceLocator.HasService<IInventoryPro
vider>())
            {
                var inventoryGO =
new GameObject("[PlayerInventory]");
                var inventory =
inventoryGO.AddComponent<Features.Player.
Inventory.PlayerInventoryService>();

GameObject.DontDestroyOnLoad(inventoryGO)
;
            }

        private static void
CreateCoroutineRunner()
        {
            if
(!ServiceLocator.HasService<ICoroutineRun
ner>())
            {
                var coroutineRunner
= new
GameObject("[CoroutineRunner]").AddCompon
ent<CoroutineRunner>();

GameObject.DontDestroyOnLoad(coroutineRun
ner.gameObject);

ServiceLocator.Register<ICoroutineRunner>
(coroutineRunner);
            }

        private static void
RegisterItemDatabase()
        {
            if
(!ServiceLocator.HasService<Features.Shar
ed.Services.IItemDatabase>())
            {
                var dbGO = new
GameObject("[ItemDatabase]");

```

```

            var database =
dbGO.AddComponent<Features.Shared.Service
s.ResourceItemDatabase>();

GameObject.DontDestroyOnLoad(dbGO);
        }

        private static void
RegisterSceneLoader()
        {
            if
(!ServiceLocator.HasService<ISceneLoader>
())
            {
                ServiceLocator.Register<ISceneLoader>(new
UnitySceneLoader());
            }

        private static void
RegisterBiomeRegistry()
        {
            if
(ServiceLocator.HasService<IBiomeRegistry
>()) return;

            var library =
Resources.Load<BiomeLibrarySO>(BiomeLibra
rySO.ResourcePath);
            if (library == null)
            {
                Debug.LogWarning(
$"[GameBootstrapper] No BiomeLibrary at
Resources/{BiomeLibrarySO.ResourcePath}.
" +
                "BiomeRegistry
registered empty - level generation will
fail until you create one.");

                ServiceLocator.Register<IBiomeRegistry>(n
ew BiomeRegistry(null));
                return;
            }

            ServiceLocator.Register<IBiomeRegistry>(n
ew BiomeRegistry(library.Biomes));
        }
        // ...
    }

    using Core.Utils;

    namespace Core.Random
    {

```



```

        public sealed class
SeededRandom : IRandomProvider
    {
        public int Seed { get; }

        private ulong _state;

        public SeededRandom(int
seed)
        {
            Seed = seed;

            _state =
unchecked((ulong)seed *
0x9E3779B97F4A7C15UL +
0xBF58476D1CE4E5B9UL);
        }

        private ulong NextUInt64()
        {
            unchecked
            {
                _state +=
0x9E3779B97F4A7C15UL;
                ulong z = _state;
                z = (z ^ (z >> 30))
* 0xBF58476D1CE4E5B9UL;
                z = (z ^ (z >> 27))
* 0x94D049BB133111EBUL;
                return z ^ (z >>
31);
            }
        }

        public int NextInt() =>
(int)(NextUInt64() >> 33);

        public int NextInt(int
maxExclusive)
        {
            if (maxExclusive <= 0)
return 0;

            uint bound =
(uint)maxExclusive;
            uint threshold =
(uint)(-bound % bound);
            while (true)
            {
                uint r =
(uint)(NextUInt64() >> 32);
                if (r >= threshold)
return (int)(r % bound);
            }
        }

        public int NextInt(int
minInclusive, int maxExclusive)

```

```

        {
            if (maxExclusive <=
minInclusive) return minInclusive;
            return minInclusive +
NextInt(maxExclusive - minInclusive);
        }

        public float NextFloat()
        {
            return (NextUInt64() >>
40) * (1.0f / (1 << 24));
        }

        public float
NextFloat(float min, float max) => min +
NextFloat() * (max - min);

        public double NextDouble()
        {
            return (NextUInt64() >>
11) * (1.0 / (1UL << 53));
        }

        public bool NextBool() =>
(NextUInt64() & 1UL) == 0UL;

        public bool NextBool(float
probability) => NextFloat() <
probability;

        public IRandomProvider
Fork(string salt) => new
SeededRandom(Hashing.FnvCombine(Seed,
salt));

        public IRandomProvider
Fork(int salt) => new
SeededRandom(Hashing.FnvCombine(Seed,
salt));
    }
}

using System.Collections.Generic;
using Core.Random;
using Core.Utills;
using Features.Voxels.Biomes;
using
Features.Voxels.Generation.Zones;
using Features.Voxels.Props;
using UnityEngine;

namespace
Features.Voxels.Generation.Pipeline
{
    public sealed class MapContext
    {
        public readonly VoxelGrid
Grid;

        public readonly
IRandomProvider Rng;
    }
}

```

```

        public readonly BiomeSO
Biome;
        public readonly int
LevelIndex;
        public readonly int
LevelSeed;

        public Grid2D<byte>
SilhouetteMask;

        public Grid2D<int>
Heightmap;

        public readonly
List<SpawnPoint> Spawns = new
List<SpawnPoint>(64);

        public Vector3Int
PlayerSpawnCell;

        public Vector3Int
PortalAnchorCell;

        public ArenaArchetype
Archetype;

        public readonly
List<ZoneAnchor> Zones = new
List<ZoneAnchor>(5);

        public readonly
List<DeferredPropSpawn> DeferredProps =
new List<DeferredPropSpawn>(128);

        public MapContext(VoxelGrid
grid, IRandomProvider rng, BiomeSO biome,
int levelIndex, int levelSeed)
        {
            Grid = grid ?? throw
new
System.ArgumentNullException(nameof(grid))
);
            Rng = rng ?? throw new
System.ArgumentNullException(nameof(rng))
;
            Biome = biome ?? throw
new
System.ArgumentNullException(nameof(biome))
);
            LevelIndex =
levelIndex;
            LevelSeed = levelSeed;
        }

        public IRandomProvider
RngFor(string layerName) =>
Rng.Fork(layerName);

```

```

        public int SizeX =>
Grid.SizeX;
        public int SizeY =>
Grid.SizeY;
        public int SizeZ =>
Grid.SizeZ;

        public ZoneType ZoneAt(int
x, int z)
        {
            if (Zones.Count == 0)
return ZoneType.KillingGround;
            float best =
float.PositiveInfinity;
            ZoneType bestType =
ZoneType.KillingGround;
            for (int i = 0; i <
Zones.Count; i++)
            {
                float d =
Zones[i].SqrDistance(x, z);
                if (d < best) {
best = d; bestType = Zones[i].Type; }
            }
            return bestType;
        }

        public bool
TryGetZone(ZoneType type, out ZoneAnchor
anchor)
        {
            for (int i = 0; i <
Zones.Count; i++)
                if (Zones[i].Type
== type) { anchor = Zones[i]; return
true; }
            anchor = default;
            return false;
        }
    }
    using System.Collections.Generic;
    using Core.Utils;
    using
Features.Voxels.Generation.Noise;
    using
Features.Voxels.Generation.Pipeline;
    using
Features.Voxels.Generation.Sampling;
    using UnityEngine;

    namespace
Features.Voxels.Generation.Layers
    {
        public sealed class
SilhouetteLayer : IGenerationLayer
        {
            private const float
MinArenaFraction = 0.08f;

```

```

        public string Name =>
"Silhouette";

        public void
Apply(MapContext ctx)
        {
            int sizeX = ctx.SizeX;
            int sizeZ = ctx.SizeZ;
            var biome = ctx.Biome;

            var noise = new
NoiseMap(sizeX, sizeZ);
            int noiseSeed =
Hashing.FnvCombine(ctx.LevelSeed, Name);
            noise.Fill(noiseSeed,
biome.SilhouetteNoise,
biome.SilhouetteNoiseScale, normalize01:
true);

            noise.ApplyRadialFalloff(biome.FalloffInner, biome.FalloffOuter, edgeValue: 0f);

            var mask = new
Grid2D<byte>(sizeX, sizeZ);
            for (int z = 0; z <
sizeZ; z++)
            {
                for (int x = 0; x <
sizeX; x++)
                {
                    mask[x, z] =
noise.Get(x, z) >=
biome.SilhouetteThreshold ? (byte)1 :
(byte)0;
                }
            }

            KeepLargestComponent(mask);

            MorphologicalOpen(mask);

            int insideCount =
CountInside(mask);
            int minArea =
Mathf.FloorToInt(sizeX * sizeZ *
MinArenaFraction);
            if (insideCount <
minArea)
                throw new
GenerationException($"Silhouette too
small ({insideCount}/{minArea} cells).");

            ctx.SilhouetteMask =
mask;
        }

```

```

        private static void
KeepLargestComponent(Grid2D<byte> mask)
        {
            var bounds = new
BoxInt2D(0, 0, mask.Width, mask.Height);
            var components =
FloodFill.AllComponents2D(bounds, p =>
mask.InBounds(p) && mask[p] == 1);
            if (components.Count <=
1) return;

            var largest =
components[0];
            for (int z = 0; z <
mask.Height; z++)
                for (int x = 0; x <
mask.Width; x++)
                {
                    if (mask[x, z] ==
0) continue;
                    if
(!largest.Contains(new Vector2Int(x, z)))
                        mask[x, z] = 0;
                }
        }

        private static void
MorphologicalOpen(Grid2D<byte> mask)
        {
            Erode(mask);
            Dilate(mask);
        }

        private static void
Erode(Grid2D<byte> mask)
        {
            int w = mask.Width, h =
mask.Height;
            var copy = new byte[w *
h];
            for (int z = 0; z < h;
z++)
                for (int x = 0; x < w;
x++)
                {
                    byte v = mask[x,
z];
                    if (v == 0) {
copy[z * w + x] = 0; continue; }
                    bool survives =
GetOr(mask, x -
1, z, 0) == 1 &&
GetOr(mask, x +
1, z, 0) == 1 &&
GetOr(mask, x,
z - 1, 0) == 1 &&
GetOr(mask, x,
z + 1, 0) == 1;
                    copy[z * w + x] =
survives ? (byte)1 : (byte)0;
                }
        }

```

```

    }
    CopyBack(copy, mask);
}

private static void
Dilate(Grid2D<byte> mask)
{
    int w = mask.Width, h =
mask.Height;
    var copy = new byte[w *
h];
    for (int z = 0; z < h;
z++)
        for (int x = 0; x < w;
x++)
        {
            byte v = mask[x,
z];
            if (v == 1) {
copy[z * w + x] = 1; continue; }
            bool grows =
                GetOr(mask, x -
1, z, 0) == 1 ||
                GetOr(mask, x +
1, z, 0) == 1 ||
                GetOr(mask, x,
z - 1, 0) == 1 ||
                GetOr(mask, x,
z + 1, 0) == 1;
            copy[z * w + x] =
grows ? (byte)1 : (byte)0;
        }
        CopyBack(copy, mask);
    }

    private static byte
GetOr(Grid2D<byte> g, int x, int z, byte
fallback) =>
        g.InBounds(x, z) ? g[x,
z] : fallback;

    private static void
CopyBack(byte[] src, Grid2D<byte> dst)
    {
        var raw = dst.RawData;
        System.Array.Copy(src,
raw, raw.Length);
    }

    private static int
CountInside(Grid2D<byte> mask)
    {
        int n = 0;
        var raw = mask.RawData;
        for (int i = 0; i <
raw.Length; i++) if (raw[i] == 1) n++;
        return n;
    }
}

```

```

using System.Collections.Generic;
using UnityEngine;

namespace Features.Enemy.AI.Brain
{
    public sealed class EnemyBrain
    {
        private const float
StuckThresholdSeconds = 2.5f;
        private const float
StuckMinMovementSqr    = 0.04f;

        private readonly
EnemyBrainSO _config;
        private readonly
BrainContext _ctx;
        private readonly
List<WeightedTactic> _tactics = new
List<WeightedTactic>();

        private ITactic _current;
        private float
_nextDecisionTime;
        private Vector3
_lastTrackedPosition;
        private float
_lastMovementTime;

        public ITactic
CurrentTactic => _current;
        public string
CurrentTacticId => _current?.Id ?? "-";
        public BrainContext Context
=> _ctx;

        public
EnemyBrain(EnemyBrainSO config,
IReadOnlyDictionary<string, ITactic>
tacticRegistry, BrainContext ctx)
        {
            _config = config;
            _ctx = ctx;

            if (config == null ||
config.Tactics == null) return;

            foreach (var entry in
config.Tactics)
            {
                if
(string.IsNullOrEmpty(entry.TacticId))
continue;
                if
(!tacticRegistry.TryGetValue(entry.Tactic
Id, out var tactic) || tactic == null)
                {
                    Debug.LogWarning($"[EnemyBrain] Tactic

```

```

'_{entry.TacticId}' not found in registry
- skipped.");
        continue;
    }
    _tactics.Add(new
WeightedTactic(tactic, entry.Weight));
    }

    public void Tick(float
currentTime)
    {
        _ctx.CurrentTime =
currentTime;

        if (_current != null)
        {
            if
(_current.IsFinished)
            {
                _current.Exit(_ctx);
                _current =
null;
            }
            else
            {
                _current.Tick(_ctx);
            }
        }

        bool isCasting =
_ctx.Self?.AbilitySystem != null &&
_ctx.Self.AbilitySystem.IsExecuting;
        bool stuck = false;
        if (_current != null &&
_ctx.Self != null && !isCasting)
        {
            Vector3 pos =
_ctx.Self.transform.position;
            if ((pos -
_lastTrackedPosition).sqrMagnitude >
StuckMinMovementSqr)
            {
                _lastTrackedPosition = pos;
                _lastMovementTime = currentTime;
            }
            else if
(currentTime - _lastMovementTime >
StuckThresholdSeconds)
            {
                stuck = true;
            }
            else if (isCasting)
            {

```

```

_lastTrackedPosition =
_ctx.Self.transform.position;
                _lastMovementTime =
currentTime;
            }

            float reactionDelay =
_ctx.Difficulty != null ?
_ctx.Difficulty.ReactionDelay : 0.25f;
            if (!stuck &&
currentTime < _nextDecisionTime) return;
            _nextDecisionTime =
currentTime + reactionDelay;

            if (stuck)
            {
                _lastMovementTime =
currentTime;
                _current?.Exit(_ctx);
                _current = null;
            }

            SelectBestTactic();
        }

        private void
SelectBestTactic()
        {
            ITactic best = null;
            float bestScore =
float.MinValue;
            float inertiaBonus =
_ctx.config != null ?
_ctx.config.SwitchInertiaBonus : 0f;

            for (int i = 0; i <
_tactics.Count; i++)
            {
                var wt =
_tactics[i];
                float raw =
wt.Tactic.Score(_ctx);

                if (raw <= 0f)
                    continue;

                float weighted =
raw * wt.Weight;
                if (wt.Tactic ==
_current) weighted += inertiaBonus;

                if (weighted >
bestScore)
                {
                    bestScore =
weighted;

```

```

        best =
wt.Tactic;
    }
}

    if (best == _current)
return;

    _current?.Exit(_ctx);
    _current = best;
    _current?.Enter(_ctx);
}

    public void Reset()
    {
        _current?.Exit(_ctx);
        _current = null;
        _nextDecisionTime = 0f;
        _lastTrackedPosition =
        _ctx?.Self != null ?
        _ctx.Self.transform.position :
        Vector3.zero;
        _lastMovementTime =
        _ctx?.CurrentTime ?? 0f;
    }

    private readonly struct
WeightedTactic
    {
        public readonly ITactic
Tactic;
        public readonly float
Weight;

        public
WeightedTactic(ITactic tactic, float
weight)
        {
            Tactic = tactic;
            Weight = weight;
        }
    }

}
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using
LetsRoll.Features.Skills.Data;
using
LetsRoll.Features.Skills.Models;

namespace
LetsRoll.Features.Skills.Services
{
    public class SkillTreeService :
MonoBehaviour
    {

```

```

[Header("Database")]
[SerializeField] private
SkillTreeDatabaseSO _database;

[Header("Settings")]
#pragma warning disable CS0414
[SerializeField] private
int _refundCost = 1;
#pragma warning restore CS0414
[SerializeField] private
bool _allowRefund = true;

    private PlayerSkillState
_state;

    public event Action
OnTreeUpdated;
    public event
Action<PassiveSkillNodeSO>
OnNodeAllocated;
    public event
Action<PassiveSkillNodeSO>
OnNodeDeallocated;
    public event Action<int>
OnPointsChanged;

    public PlayerSkillState
CurrentState => _state;
    public SkillTreeDatabaseSO
Database => _database;

    public void
Initialize(PlayerSkillState state, string
classId = null)
    {
        if (_database == null)
        {
            Debug.LogError("[SkillTreeService]
Database is missing!");
            return;
        }

        _database.Initialize();
        _state = state;

        _state.InitializeRuntime();

        if
(!string.IsNullOrEmpty(classId))
        {
            _state.PlayerClassId = classId;
            var startNode =
            _database.GetClassStartNode(classId);
            if (startNode !=
null)
            {

```

```

_state.ClassStartNodeId =
startNode.NodeId;
    }
}

    if
(!string.IsNullOrEmpty(_state.ClassStartN
odeId) &&
!_state.HasNode(_state.ClassStartNodeId))
    {
ForceAllocateNode(_state.ClassStartNodeId
);
    }
}

    public bool
CanAllocate(string nodeId)
    {
        if (_state == null)
return false;
        if
(_state.HasNode(nodeId)) return false;
        if
(_state.AvailableSkillPoints <= 0) return
false;

        var node =
_database.GetNode(nodeId);
        if (node == null)
return false;

        if (node.RequiredLevel
> 0)
        {
        }

        return
IsReachable(node);
    }

    public bool
TryAllocateNode(string nodeId, out string
error)
    {
        error = string.Empty;

        if (_state == null)
        {
            error = "Service
not initialized.";
            return false;
        }

        var node =
_database.GetNode(nodeId);
        if (node == null)

```

```

    {
        error = $"Node
{nodeId} not found.";
        return false;
    }

    if
(_state.HasNode(nodeId))
    {
        error = "Already
allocated.";
        return false;
    }

    if
(_state.AvailableSkillPoints <= 0)
    {
        error = "No skill
points available.";
        return false;
    }

    if (!IsReachable(node))
    {
        error = "Node is
not connected to your path.";
        return false;
    }

    _state.AvailableSkillPoints--;
    _state.AddNode(nodeId);

    OnNodeAllocated?.Invoke(node);

    OnPointsChanged?.Invoke(_state.AvailableS
killPoints);

    OnTreeUpdated?.Invoke();

    return true;
}

    public bool
CanDeallocate(string nodeId)
    {
        if (!_allowRefund)
return false;
        if (_state == null)
return false;
        if
(!_state.HasNode(nodeId)) return false;
        if (nodeId ==
_state.ClassStartNodeId) return false;

        var node =
_database.GetNode(nodeId);

```

```

        if (node == null)
            return false;

        !WouldBreakPath(node);
    }

    public bool
    TryDeallocateNode(string nodeId, out
    string error)
    {
        error = string.Empty;

        if (!_allowRefund)
        {
            error = "Refund is
disabled.";
            return false;
        }

        if (_state == null)
        {
            error = "Service
not initialized.";
            return false;
        }

        var node =
        _database.GetNode(nodeId);
        if (node == null)
        {
            error = "Node not
found.";
            return false;
        }

        if
        (!_state.HasNode(nodeId))
        {
            error = "Node is
not allocated.";
            return false;
        }

        if (nodeId ==
        _state.ClassStartNodeId)
        {
            error = "Cannot
refund class start node.";
            return false;
        }

        if
        (WouldBreakPath(node))
        {
            error = "Removing
this node would disconnect other
allocated nodes.";

```

```

            return false;
        }

        _state.RemoveNode(nodeId);

        _state.AvailableSkillPoints += 1;

        OnNodeDeallocated?.Invoke(node);

        OnPointsChanged?.Invoke(_state.AvailableS
killPoints);

        OnTreeUpdated?.Invoke();

        return true;
    }

    public bool
    IsReachable(PassiveSkillNodeSO
targetNode)
    {
        if (targetNode == null)
            return false;

        var allocatedIds =
        _state.GetAllocatedNodeIds();
        int count = 0;
        foreach (var _ in
        allocatedIds) count++;

        if (count == 0)
        {
            return
            targetNode.NodeId ==
            _state.ClassStartNodeId;
        }

        foreach (var neighbor
        in targetNode.Neighbors)
        {
            if (neighbor !=
            null && _state.HasNode(neighbor.NodeId))
            {
                return true;
            }
        }

        return false;
    }

    private bool
    WouldBreakPath(PassiveSkillNodeSO
nodeToRemove)
    {
        var allocatedNeighbors
        = nodeToRemove.Neighbors

```



```

        .Where(n => n !=
null && _state.HasNode(n.NodeId))
        .ToList();

        if
(allocatedNeighbors.Count == 0)
            return false;

        var startNode =
_database.GetNode(_state.ClassStartNodeId
);
        if (startNode == null)
return true;

        var tempState =
_state.Clone();

tempState.RemoveNode(nodeToRemove.NodeId)
;

        var visited = new
HashSet<string>();
        var queue = new
Queue<string>();

queue.Enqueue(_state.ClassStartNodeId);
visited.Add(_state.ClassStartNodeId);

        while (queue.Count > 0)
        {
            string currentId =
queue.Dequeue();

            var current =
_database.GetNode(currentId);
            if (current ==
null) continue;

            foreach (var
neighbor in current.Neighbors)
            {
                if (neighbor ==
null) continue;

                if
(visited.Contains(neighbor.NodeId))
continue;

                if
(!tempState.HasNode(neighbor.NodeId))
continue;

                visited.Add(neighbor.NodeId);

                queue.Enqueue(neighbor.NodeId);
            }
        }

```

```

        foreach (var id in
tempState.GetAllocatedNodeIds())
        {
            if
(!visited.Contains(id))
                return true;
        }

        return false;
    }
    // ...
}

using
Features.Voxels.Generation.Pipeline;
using UnityEngine;

namespace
Features.Voxels.Visualization
{
    public sealed class
InvisibleWallController : MonoBehaviour
    {
        private static readonly int
PlayerPosId =
Shader.PropertyToID("_PlayerPos");
        private static readonly int
RevealRadiusId =
Shader.PropertyToID("_RevealRadius");
        private static readonly int
DomeBottomId =
Shader.PropertyToID("_DomeBottom");
        private static readonly int
DomeTopId =
Shader.PropertyToID("_DomeTop");

        [SerializeField] private
float _topY = 30f;
        [SerializeField] private
float _bottomY = -20f;
        [SerializeField] private
float _revealRadius = 8f;
        [SerializeField] private
Material _wallMaterial;

        private MeshFilter _filter;
        private MeshRenderer
_renderer;

        private Transform _player;
        private
MaterialPropertyBlock _mpb;

        private void Awake()
        {
            _filter =
gameObject.AddComponent<MeshFilter>();
            _renderer =
gameObject.AddComponent<MeshRenderer>();

```

```

_renderer.shadowCastingMode =
UnityEngine.Rendering.ShadowCastingMode.Off;

_renderer.receiveShadows = false;

        if (_wallMaterial !=
null)
_renderer.sharedMaterial = _wallMaterial;

        _mpb = new
MaterialPropertyBlock();
    }

    public void
BindPlayer(Transform player) => _player =
player;

    public void Show(MapData
map)
    {
        if (map == null)
return;

        Clear();

        var built =
InvisibleWallBuilder.Build(map, _bottomY,
_topY);
        _filter.sharedMesh =
built.Visual;
        _renderer.enabled =
true;
    }

    public void Clear()
    {
        if (_filter != null &&
_filter.sharedMesh != null)
        {
            Destroy(_filter.sharedMesh);
            _filter.sharedMesh
= null;
        }
    }
    private void LateUpdate()
    {
        if (_renderer == null
|| _player == null || _wallMaterial ==
null) return;

        _renderer.GetPropertyBlock(_mpb);

        _mpb.SetVector(PlayerPosId,
_player.position);

```

```

        _mpb.SetFloat(RevealRadiusId,
_revealRadius);

        _mpb.SetFloat(DomeBottomId,
transform.position.y + _bottomY);

        _mpb.SetFloat(DomeTopId,
transform.position.y + _topY);

        _renderer.SetPropertyBlock(_mpb);
    }
}

```