Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

МЕТОД ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ РІВНІВ З ВИКОРИСТАННЯМ МЕТОДУ ДВІЙКОВОГО РОЗБИТТЯ ПРОСТОРУ

Виконав: магістрант групи 2КІ-24м
спеціальності 123 – Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
Сирдій Д.П.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ОТ
Колесник І.С.
(прізвище та ініціали)
«12» 12 2025 р.

Опонент: к.т.н. доц. зав. каф. МБІС
Карпінець В.В.
(прізвище та ініціали)
«12» 12 2025 р.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О.Д.
16. 12 2025 року

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень Магістр

Галузь знань — 12 «Інформаційні технології»

Спеціальність — 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
25.09.2025 року

З А В Д А Н Н Я
НА МАГІСТЕСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Сирдію Дмитру Петровичу

1 Тема роботи: «Метод процедурної генерації рівнів з використанням методу двійкового розбиття простору», керівник роботи Колесник І.С. к.т.н., доцент кафедри ОТ, затверджено наказом вищого навчального закладу від «24» вересня 2025 року № 313.

2 Строк подання студентом роботи 04.12.2025 р.

3 Вихідні дані до роботи: актуальність, аналіз аналогів, розробка додатка, структурна схема, алгоритм роботи, тестування.

4 Зміст пояснювальної записки: вступ, аналіз процедурної генерації ігрових рівнів, розробка процедурної генерації рівнів з використанням алгоритму двійкового розбиття простору, програмне забезпечення системи генерації в середовищі, тестування ефективності та валідація отриманих результатів, висновки, перелік джерел посилання, додатки.

5 Перелік графічного матеріалу — діаграма методу процедурної генерації рівнів.

6 Консультанти розділів роботи представлені у таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1 – 4	к.т.н., доц. каф. Колесник І.С.		
5	к.т.н., доц., ЕПВМ Адлер О.О.		
Нормо Контроль	асист. каф. ОТ Швець С.І.		

7 Дата видачі завдання 25.09.2025 р.

8 Календарний план виконання БДР наведений у таблиці 2.

Таблиця 2 — Календарний план

№	Назва та зміст етапу	Термін виконання		Примітка
1.	Постановка задачі та визначення мети дослідження	25.09.2025	29.09.2025	вик.
2.	Аналіз сучасних методів процедурної генерації рівнів	30.09.2025	05.10.2025	вик.
3.	Обгрунтування вибору технологічної бази	06.10.2025	09.10.2025	вик.
4.	Розробка архітектури системи процедурної генерації	10.10.2025	15.10.2025	вик.
5.	Розробка BSP-алгоритм для поділу простору рівня	16.10.2025	22.10.2025	вик.
6.	Реалізація основних модулів системи у середовищі Unreal Engine 5	23.10.2025	31.10.2025	вик.
7.	Розробка підсистеми контролю параметрів генерації та UI-панелі в редакторі	01.11.2025	05.11.2025	вик.
8.	Тестування системи генерації рівнів та аналіз результатів	06.11.2025	08.11.2025	вик.
9.	Оформлення пояснювальної записки та додатків	08.11.2025	10.11.2025	вик.
10.	Попередній захист магістерської роботи	11.11.2025	12.11.2025	вик.
11.	Перевірка якості виконання магістерської кваліфікаційної роботи та усунення недоліків			вик.

Студент _____ Сирдій Д.П.
Керівник роботи _____ Колесник.І.С.

АНОТАЦІЯ

УДК 000

Сирдій Д.П. Метод процедурної генерації рівнів з використанням методу двійкового розбиття простору. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна інженерія, освітня програма — Комп'ютерна інженерія. Вінниця: ВНТУ, 2025 — 135 с.

На укр. мові. Бібліогр.: 35 назв; рис.: 14, табл.:11

У магістерській кваліфікаційній роботі розроблено систему процедурної генерації ігрових рівнів на основі алгоритму двійкового розбиття простору, орієнтовану на використання в середовищі Unreal Engine 5. Проаналізовано сучасні підходи до процедурної генерації контенту та обґрунтовано вибір BSP-алгоритму для формування логічно структурованих ігрових просторів. Запропоновано модульну архітектуру системи, реалізовану з використанням мови програмування C++ у поєднанні з інструментами Blueprint і PCG Framework, що забезпечує керування складністю, варіативністю та ігровим балансом рівнів. Проведене експериментальне дослідження підтвердило ефективність запропонованого підходу для створення структурно узгоджених ігрових середовищ із можливістю інтерактивного налаштування параметрів генерації.

Ключові слова: процедурна генерація контенту, BSP-алгоритм, ігрові рівні, Unreal Engine 5, C++, PCG-система.

ABSTRACT

UDC 000

Syrdii D.P. Procedural Level Generation Method Using Binary Space Partitioning. Master's Qualification Thesis in Specialty 123 — Computer Engineering, Educational Program — Computer Engineering. Vinnytsia: VNTU, 2025 — 135 p.

In Ukrainian. References: 35 titles; figures: 14; tables: 11.

This master's qualification thesis presents the development of a procedural game level generation system based on the Binary Space Partitioning (BSP) algorithm, intended for use within the Unreal Engine 5 environment. Modern approaches to procedural content generation are analyzed, and the choice of the BSP algorithm for creating logically structured game spaces is substantiated. A modular system architecture is proposed and implemented using the C++ programming language in combination with Blueprint tools and the PCG Framework, enabling control over level complexity, variability, and gameplay balance. Experimental evaluation confirmed the effectiveness of the proposed approach for generating structurally consistent game environments with support for interactive adjustment of generation parameters.

Keywords: procedural content generation, BSP algorithm, game levels, Unreal Engine 5, C++, PCG system.

ЗМІСТ

ВСТУП

Ігрові рушії, такі як Unreal Engine 5, дають змогу інтегрувати алгоритмічні методи побудови рівнів, що формують основу процедурної генерації контенту. Такий підхід суттєво підвищує ефективність розробки, забезпечує повторюваність, структурну логічність і контрольований рівень складності. Одним із найбільш стабільних і формально описаних методів для таких задач є метод двійкового розбиття простору (Binary Space Partitioning — BSP), який дозволяє автоматизовано створювати кімнати, коридори та інші просторові зони зі збереженням коректної топології.

Метою магістерської роботи є розроблення та реалізація системи процедурної генерації рівнів з використанням методу двійкового розбиття простору у середовищі Unreal Engine 5, яка забезпечує структурну узгодженість, варіативність і стабільну продуктивність процесу побудови рівнів.

Для досягнення поставленої мети вирішуються такі основні завдання: проаналізувати сучасні алгоритми процедурної генерації контенту та визначити доцільність застосування BSP; формалізувати процес поділу простору на підобласті; розробити архітектуру системи генерації рівнів у Unreal Engine; реалізувати алгоритм BSP мовою C++; забезпечити інтеграцію генератора з рушієм та інтерфейсом користувача; провести тестування швидкодії, варіативності та логічної збалансованості згенерованих рівнів.

Об’єктом дослідження є процес процедурного формування тривимірних рівнів у ігровому середовищі Unreal Engine 5. Предметом дослідження є методи та алгоритми побудови просторових структур із використанням алгоритму двійкового розбиття простору.

Методи дослідження базуються на системному аналізі, теорії алгоритмів процедурної генерації, об’єктно-орієнтованому програмуванні, експериментальній перевірці результатів і статистичному аналізі даних.

Наукова новизна роботи полягає у вдосконаленні методу BSP для задач процедурної генерації рівнів шляхом введення динамічних параметрів контролю

складності, варіативності та розміру областей, а також інтеграції алгоритму безпосередньо у рушій Unreal Engine 5 через комбінацію C++ та Blueprint. Це забезпечує можливість керованої генерації ігрових просторів у реальному часі з високою стабільністю та продуктивністю.

Практичне значення одержаних результатів полягає у створенні модульної системи, яку можна застосовувати для побудови ігрових карт і середовищ у проєктах жанрів roguelike, simulation, adventure, а також у навчальних і дослідницьких симуляторах. Реалізований генератор може бути інтегрований у будь-який Unreal Engine-проєкт, забезпечуючи швидке створення унікальних рівнів без ручного проєктування.

1 АНАЛІТИЧНИЙ ОГЛЯД ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Підходів до процедурної генерації ігрових рівнів

Процедурна генерація контенту (Procedural Content Generation, PCG) передбачає автоматизоване створення ігрових рівнів, ландшафтів та об’єктів із використанням алгоритмічних методів. Застосування PCG дає змогу скоротити витрати на ручну розробку, підвищити варіативність ігрового процесу та забезпечити високу реіграбельність проєктів [2].

PCG найбільш поширена в жанрах roguelike, sandbox та RPG, де унікальність середовища безпосередньо впливає на зацікавленість гравця [3]. У дослідженнях виділяють кілька базових підходів до процедурного створення контенту, які відрізняються принципами роботи та рівнем контролю результату.

До таких підходів належать клітинні автомати, що використовуються для генерації печерних і лабіринтових структур, алгоритми шуму Перліна для формування ландшафтів, а також метод Wave Function Collapse, заснований на локальних правилах сумісності елементів. Для збереження логічної структури рівнів також застосовуються графові моделі, які описують просторові та сюжетні зв’язки між зонами.

Одним із найбільш стабільних методів процедурної генерації є Binary Space Partitioning (BSP), що базується на рекурсивному поділі простору на підобласті до досягнення заданих обмежень. BSP забезпечує логічну та передбачувану структуру рівнів, спрощує поєднання кімнат і коридорів та широко застосовується в ігрових і навчальних проєктах.

Сучасні підходи передбачають інтеграцію класичних алгоритмів із евристичними методами та машинним навчанням для оптимізації рівнів за складністю й балансом, а також для наближення результатів генерації до ручної роботи дизайнерів. Перспективним напрямом є гібридні системи, що поєднують BSP із граматиками або нейронними моделями для створення багаторівневих структур.

Ігрові рушії підтримують процедурну генерацію на різних рівнях. Unreal Engine 5 надає вбудований PCG Framework і можливість реалізації власних алгоритмів на C++, що робить його ефективним середовищем для дослідження та впровадження BSP-орієнтованих рішень.

1.1.1 Основні напрями та класи методів процедурної генерації

Методи процедурної генерації контенту (PCG) у сучасних ігрових системах класифікуються за принципами роботи алгоритмів, рівнем автоматизації та типами створюваного контенту. Така класифікація дозволяє систематизувати підходи до побудови ігрових просторів і обґрунтовано обирати методи відповідно до цілей розробки [4]. Узагальнену класифікацію методів процедурної генерації ігрових просторів наведено на рисунку 1.1.

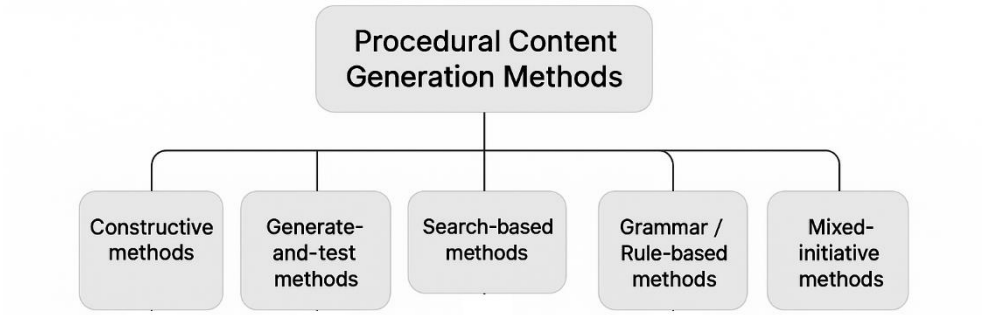

Рисунок 1.1 — Класифікація методів процедурної генерації

Одним із базових класів є конструктивні методи, які формують контент за наперед заданими правилами без подальшої оцінки результату. До них належать алгоритми генерації лабіринтів, кімнат і топологій рівнів, зокрема метод Binary Space Partitioning (BSP), що реалізує рекурсивний поділ простору. Перевагами таких підходів є висока швидкодія та стабільність, однак їхня адаптивність до ігрового контексту є обмеженою.

Методи типу “генеруй і тестуй” передбачають багаторазову генерацію контенту з подальшою перевіркою його відповідності заданим критеріям. Цей підхід застосовується, зокрема, у клітинних автоматах і шумових методах для

створення печер і ландшафтів. Основною перевагою є можливість контролю якості, тоді як недоліком — підвищені обчислювальні витрати [5].

До окремого класу належать пошукові методи, які розглядають генерацію контенту як задачу оптимізації в просторі можливих рішень. Такі підходи використовують еволюційні алгоритми, алгоритми пошуку або навчання з підкріпленням і дозволяють формувати рівні з урахуванням заданих метрик складності чи балансу.

Методи на основі правил і граматик базуються на формальних описах структури ігрового простору та застосовуються для створення ієрархічних архітектур, міст або наративних ліній. Вони забезпечують високий рівень контролю, проте потребують складного попереднього проєктування.

Окрему групу становлять методи машинного навчання, які використовують генеративні моделі та алгоритми штучного інтелекту для відтворення стилю рівнів, створених дизайнерами, або для динамічної адаптації середовища під дії гравця.

У сучасних комерційних проєктах широко застосовуються змішані (mixed-initiative) підходи, де генератор працює у взаємодії з дизайнером. Такі системи поєднують автоматизацію з можливістю ручного коригування результатів, що підвищує практичну цінність PCG-інструментів.

1.1.2 Функціональні та якісні вимоги до PCG-систем у сучасних іграх

Сучасні системи процедурної генерації рівнів повинні відповідати комплексу технічних і функціональних вимог, що забезпечують стабільність, варіативність, продуктивність і зручність інтеграції в ігрові рушії [6]. Узагальнено ці вимоги поділяють на технічні, якісні, функціональні та дизайнерські.

Ключовою характеристикою PCG-систем є автономність, тобто здатність генератора створювати велику кількість унікальних рівнів без прямої участі розробника [2]. Кількість можливих варіантів рівнів можна формально описати як простір рішень:

$$N = \prod_{i=1}^{k} p_i,$$

де p_i— кількість допустимих значень і-го параметра генерації,

k — загальна кількість керованих параметрів системи.

Для ігор із відкритими або напіввідкритими просторами вирішальною є масштабованість. У BSP-алгоритмах вона досягається шляхом рекурсивного поділу простору, глибина якого визначає деталізацію карти. Максимальна кількість секцій після d рівнів поділу визначається як:

$$S = 2^d,$$

де d — глибина BSP-дерева.

Якість згенерованого рівня визначається не лише його геометричною коректністю, а й ігровою придатністю, що включає логічну зв’язність, відсутність помилкових тупиків і збалансовану складність. Для кількісної оцінки зв’язності простору використовується показник досяжності:

$$R = \frac{V_{reachable}}{Vtotal},$$

де $V_{reachable}$ — кількість досяжних зон або кімнат,

V_{total} — загальна кількість зон рівня.

Важливою вимогою є контрольована випадковість, яка досягається через параметризацію процесу генерації. У BSP-системах імовірність поділу простору на кожному кроці може задаватися стохастично:

$$P_{split} \in [0,1],$$

Система процедурної генерації повинна забезпечувати достатню варіативність, оскільки повторюваність знижує реіграбельність. Для цього використовується ініціалізація генератора випадковим зерном:

$$Level = G(seed),$$

де G — функція генерації,

$seed$ — випадкове значення, що визначає унікальність результату.

Сучасні PCG-системи повинні легко інтегруватися в ігрові рушії та редактори. В Unreal Engine 5 це реалізується через Procedural Content Generation Framework, який дозволяє поєднувати BSP-алгоритми з візуальними інструментами та C++-модулями, забезпечуючи швидке тестування і масштабування [7].

Оцінювання якості згенерованих рівнів здійснюється за допомогою статистичних і структурних метрик, зокрема середньої довжини шляхів, щільності об’єктів та балансу складності. Такі метрики дозволяють автоматизувати перевірку результатів і мінімізувати потребу в ручному тестуванні.

1.1.3 Аналіз існуючих рушіїв для процедурної генерації контенту

Сучасні ігрові рушії надають різні можливості для реалізації процедурної генерації контенту (PCG), що відрізняються за рівнем інтеграції алгоритмів, доступом до низькорівневих API, продуктивністю та наявністю готових інструментів. Порівняльний аналіз найбільш поширених платформ — Unity, Unreal Engine 5 та Godot Engine — дозволяє обґрунтовано визначити оптимальне середовище для реалізації системи процедурної генерації рівнів на основі методу Binary Space Partitioning (BSP).

Для вибору оптимального середовища реалізації процедурної генерації рівнів важливими є такі характеристики ігрового рушія, як рівень підтримки PCG-алгоритмів, доступ до низькорівневих механізмів, продуктивність при роботі з

великими сценами та наявність інструментів візуального редагування [8]. Особливого значення ці критерії набувають у контексті реалізації алгоритму Binary Space Partitioning, який потребує ефективної роботи з просторовими структурами та рекурсивними обчисленнями. З цієї точки зору доцільно провести порівняльний аналіз найбільш поширених ігрових рушіїв, що використовуються у сучасній розробці.

Unity орієнтований переважно на інді- та кросплатформенну розробку, пропонуючи широку екосистему плагінів і відносно низький поріг входу. Unreal Engine 5, у свою чергу, забезпечує глибоку інтеграцію процедурної генерації через PCG Framework, поєднуючи візуальні інструменти з високопродуктивною реалізацією на C++. Godot Engine є відкритим рішенням з повним доступом до вихідного коду, що робить його привабливим для дослідницьких і навчальних проєктів, однак з обмеженими можливостями для масштабних 3D-сцен. Узагальнене порівняння основних характеристик рушіїв наведено в таблиці 1.1.

Таблиця 1.1 — Порівняльна характеристика ігрових рушіїв для PCG-систем

Критерій	Unity	Unreal Engine 5	Gobot Engine
Підтримка PCG	Через плагіни та скрипти	Вбудований PCG Framework	Базові засоби
Реалізація BSP	Можлива	Нативна через C++	Можлива
Продуктивність у великих сценах	Середня	Висока	Обмежена
Доступність до низькорівневого API	Обмежений	Повний	Повний
Інструменти візуального редагування	Розвинені	Дуже розвинені	Обмежені
Поріг входу	Низький	Високий	Середній
Придатність для BSP-PCG	Середня	Висока	Середня

Unreal Engine 5 вирізняється серед розглянутих платформ завдяки поєднанню нативної підтримки C++, вбудованого PCG Framework і сучасних

систем керування великими світами, що спрощує інтеграцію BSP-алгоритмів у повноцінний виробничий цикл. Це дозволяє реалізовувати процедурну генерацію не лише як допоміжний інструмент, а як центральний елемент архітектури рівнів, зберігаючи баланс між продуктивністю, керованістю та візуальною якістю результатів.

1.2 Алгоритмічні методи побудови ігрових просторів

Сучасні алгоритмічні методи процедурної генерації рівнів охоплюють широкий спектр підходів — від детермінованих конструктивних алгоритмів до адаптивних систем на основі штучного інтелекту. У наукових дослідженнях такі методи класифікують за принципом роботи, рівнем контрольованості результату та обчислювальною складністю [9]. Узагальнену схему основних алгоритмічних підходів до генерації ігрових рівнів наведено на рисунку 1.2.

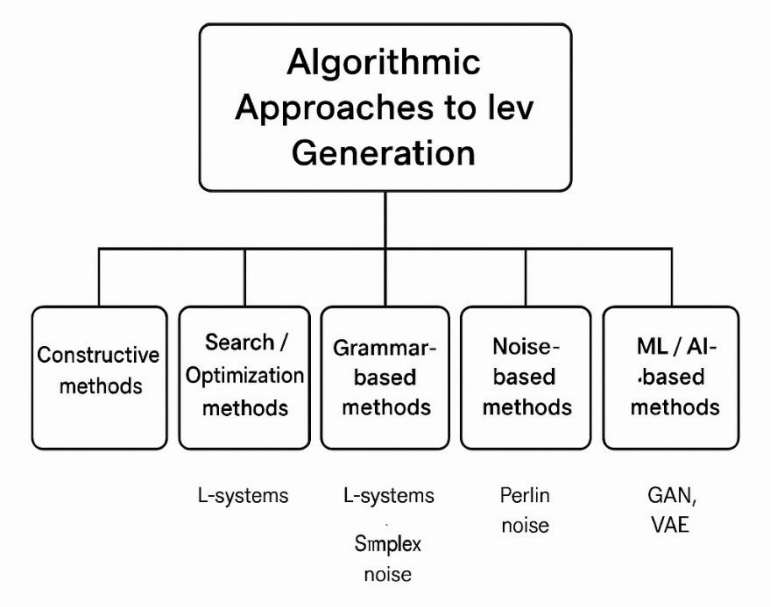

Рисунок 1.2 — Алгоритмічні підходи до процедурної генерації рівнів

Конструктивні методи ґрунтуються на детермінованих правилах побудови контенту без подальшої оптимізації. До цього класу належать Binary Space Partitioning (BSP), клітинні автомати та генератори лабіринтів. Основними перевагами таких алгоритмів є висока продуктивність і простота реалізації, що

робить їх придатними для генерації в реальному часі. Недоліком є обмежена варіативність і повторюваність структур за фіксованих параметрів [10].

Пошукові та оптимізаційні методи розглядають генерацію рівнів як задачу пошуку оптимального рішення у просторі можливих варіантів. Вони використовують еволюційні алгоритми, методи Монте-Карло або оптимізацію за заданими метриками складності й балансу. Ці підходи забезпечують високу гнучкість результатів, однак характеризуються значними обчислювальними витратами.

Граматичні методи базуються на формальних правилах, які описують ієрархічну структуру ігрового простору. Вони широко застосовуються для генерації архітектурних об’єктів, міст і наративних елементів, забезпечуючи високий рівень контрольованості, але вимагаючи складного попереднього проєктування.

Шумові методи використовують псевдовипадкові функції для створення безперервних ландшафтів і карт висот. Такі алгоритми забезпечують природну морфологію поверхні та високу продуктивність, однак мають низький рівень семантичної структурованості, що ускладнює контроль логіки проходження рівня.

Методи машинного навчання та штучного інтелекту застосовують генеративні моделі й алгоритми навчання з підкріпленням для створення адаптивного контенту, здатного підлаштовуватися під стиль гри користувача. Основними обмеженнями є складність навчання та потреба у великих обсягах даних.

У сучасних PCG-системах дедалі частіше використовуються гібридні підходи, які поєднують конструктивні алгоритми з пошуковими або ML-модулями. Типовим прикладом є використання BSP для формування базової топології рівня з подальшою оптимізацією параметрів за допомогою інтелектуальних методів. Такий підхід дозволяє поєднати високу продуктивність із підвищеною адаптивністю та якістю результатів.

1.2.1 Принципи роботи Perlin Noise, Cellular Automata та BSP

Огляд ключових алгоритмів процедурної генерації — Perlin Noise, Cellular Automata, Wave Function Collapse (WFC) та Binary Space Partitioning (BSP) — демонструє різноманіття підходів до побудови ігрових світів, що відрізняються як за принципами роботи, так і за структурними результатами. Ці методи утворюють основу більшості сучасних PCG-систем і широко використовуються в ігрових рушіях, зокрема Unreal Engine, Unity та Godot.

Метод Perlin Noise, розроблений Кеном Перліном, є класичним прикладом градієнтного шуму, який забезпечує плавні, природно виглядаючі переходи значень у просторі. На відміну від білого шуму, Перлін шум створює корельовані флуктуації, завдяки чому зображення чи рельєфи, згенеровані цим методом, мають органічну структуру [11].

Генерація відбувається шляхом інтерполяції між випадково орієнтованими векторами (градієнтами) у вузлах просторової сітки. Важливу роль відіграють параметри:

— octaves — кількість шарів шуму, які накладаються для утворення фрактальної структури;

— lacunarity — коефіцієнт, що визначає зміну частоти між шарами;

— persistence — параметр, який контролює амплітуду кожного наступного шару.

Завдяки цим параметрам формується складна фрактальна поверхня, що імітує гірські масиви, річкові долини або інші природні об’єкти [36].

На рисунку 1.3 показано, як зміна параметрів періодів, кількості октав, лакунарності та персистентності впливає на складність і деталізацію фрактального шуму. При збільшенні octaves і lacunarity рельєф стає більш насиченим і текстурним, тоді як зменшення persistence робить поверхню більш згладженою.

Саме ця властивість робить шум Перліна основою для генерації ландшафтів, текстур і карт висот у більшості ігрових рушіїв, включно з Unreal Engine 5 та Unity.

Клітинні автомати (Cellular Automata, CA) — це дискретні динамічні системи, у яких простір поділений на клітини, кожна з яких має стан (наприклад, «стіна» або «порожнеча»). Стан клітини на наступному кроці визначається локальними правилами на основі сусідів [12]. Типовий приклад правила: «якщо навколо клітини більше п’яти стін — вона стає стіною, інакше — порожньою».

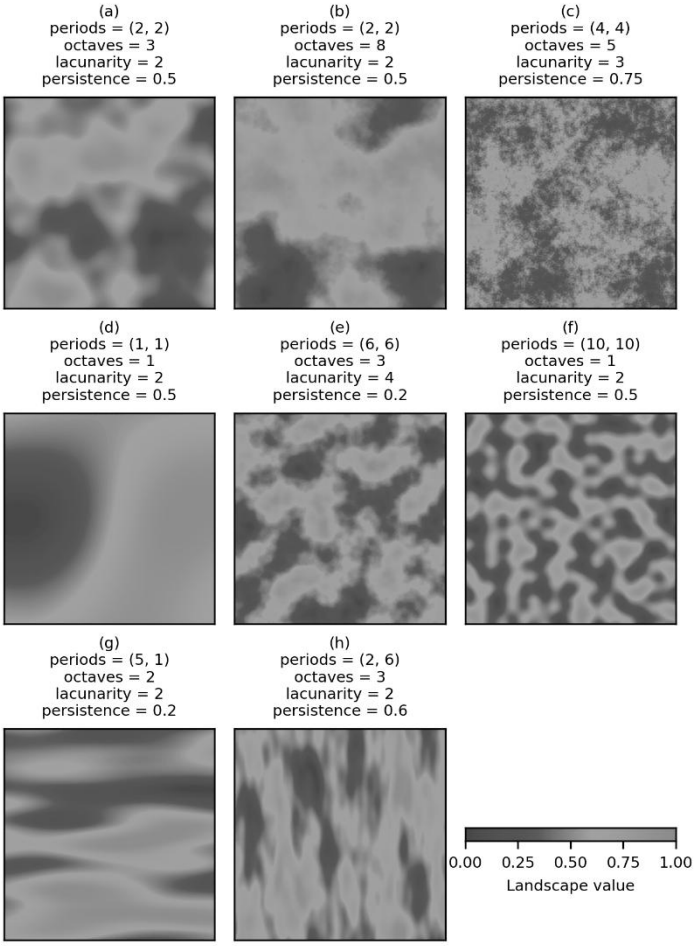

Рисунок 1.3 – Візуалізація впливу параметрів octaves, lacunarity та persistence на генерацію шуму Перліна

Такий підхід дозволяє перетворити випадкову шумову матрицю у реалістичні форми печер, коридорів або підземель, з плавними контурами, близькими до природних. Модифікації клітинних автоматів можуть включати еволюційні алгоритми або методи навчання, які оптимізують набір правил, підвищуючи узгодженість форм і керованість генерації.

Перевагою методу є висока органічність створюваних структур, а основним недоліком — низький рівень глобального контролю: складно передбачити загальну топологію карти лише на основі локальних взаємодій.

Wave Function Collapse (WFC) — це один із найпотужніших сучасних алгоритмів процедурної генерації, який базується на принципі задоволення обмежень (constraint satisfaction). Алгоритм бере як вхідний приклад (набір патернів або плиток) і намагається побудувати нову карту, в якій кожна клітина узгоджується з сусідніми відповідно до заданих правил. Процес можна уявити як вирішення головоломки «судоку» для мапи: у кожній клітині залишається набір допустимих варіантів, який поступово звужується, поки не буде обрано єдиний допустимий стан.

WFC забезпечує локальну узгодженість та глобальну різноманітність, що дозволяє створювати карти, архітектурні структури або навіть текстури, які зберігають заданий стиль. Недоліком є висока обчислювальна складність і можливість “зависання” процесу, якщо система обмежень надто жорстка. Проте для проєктів, де важлива стилістична цілісність, цей підхід є надзвичайно ефективним [13].

Метод Binary Space Partitioning (BSP) належить до класу конструктивних алгоритмів і використовується для рекурсивного поділу простору на менші підобласті за допомогою вертикальних або горизонтальних розрізів. У контексті генерації рівнів BSP дозволяє створювати композиційно впорядковані простори — кімнати, коридори, зали — які логічно з’єднані між собою.

Процес побудови починається з єдиного прямокутного регіону, який ділиться на дві частини випадковою лінією. Далі кожна частина знову ділиться рекурсивно, утворюючи дерево розбиття. Кінцеві “листові” вузли дерева зазвичай інтерпретуються як кімнати, а ребра між ними — як переходи або двері. BSP-граф є ефективним представленням рівня, оскільки дозволяє швидко визначати видимість, розміщення об’єктів і контролювати масштаб генерації.

До головних переваг належать структурованість, стабільність і простота реалізації, а основним недоліком є певна геометрична передбачуваність, що частково вирішується введенням стохастичних факторів у процес поділу.

Таким чином, для побудови рівнів типу “підземель” чи “коридорних просторів” метод BSP є найбільш ефективним базовим інструментом, який може бути розширений гібридними або машинними підходами.

1.2.2 Порівняльний аналіз методів за критеріями керованості

Для обґрунтованого вибору алгоритму процедурної генерації рівнів доцільно порівняти найбільш поширені методи за ключовими критеріями: складність реалізації, варіативність результатів і обчислювальна продуктивність. Таке порівняння дозволяє визначити сильні та слабкі сторони кожного підходу й оцінити доцільність його використання в конкретних ігрових або симуляційних сценаріях [44]. Узагальнені характеристики методів Perlin Noise, Cellular Automata, Wave Function Collapse та Binary Space Partitioning наведено в таблиці 1.2.

Таблиця 1.2 — Порівняння методів процедурної генерації рівнів

Метод	Складність реалізації	Варіативність результатів	Продуктивність
Perlin Noise	Низька	Висока (для ландшафтів)	Висока
Cellular Automata	Середня	Висока	Висока
Wave Function Collapse	Висока	Дуже висока	Низька
Binary Space Partitioning (BSP)	Середня	Середня	Висока

Perlin Noise вирізняється простотою реалізації та високою швидкодією, що робить його ефективним для генерації ландшафтів і карт висот, однак він не забезпечує структурної організації рівнів [18]. Клітинні автомати дозволяють створювати різноманітні органічні структури з відносно невеликими

обчислювальними витратами, хоча якість результатів значною мірою залежить від правильно підібраних правил переходу [14].

Wave Function Collapse забезпечує найвищу структурну узгодженість і варіативність, але характеризується значною алгоритмічною складністю та низькою продуктивністю при масштабуванні, що обмежує його застосування в реальному часі. На цьому фоні метод Binary Space Partitioning демонструє оптимальний баланс між складністю реалізації, продуктивністю та керованістю структури рівня. BSP ефективно масштабується, забезпечує логічну топологію і придатний для використання як базовий алгоритм у реальному часі, особливо в поєднанні з додатковими стохастичними або інтелектуальними модулями [48].

1.2.3 Методи оцінювання якості згенерованих рівнів

Для об’єктивної оцінки результатів процедурної генерації необхідно застосовувати систему метрик, що дозволяє кількісно та якісно визначити придатність згенерованих рівнів до ігрового використання. Такі метрики дають змогу оцінити відповідність рівнів вимогам геймдизайну, очікуваному рівню складності та цільовому ігровому досвіду, а також використовуються як інструмент зворотного зв’язку для подальшої оптимізації алгоритмів генерації [22]. Незважаючи на відмінності між алгоритмами (BSP, Perlin Noise, WFC, Cellular Automata), базові принципи оцінювання якості залишаються спільними.

Одним із ключових показників якості є прохідність рівня, яка визначає можливість завершення ігрового простору без виникнення тупиків або недосяжних зон [15]. Формально рівень подається у вигляді графа $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, де вершини відповідають кімнатам або зонам, а ребра — переходам між ними. Коефіцієнт прохідності може бути визначений як відношення кількості досяжних вершин до загальної кількості вершин графа:

$$R = \frac{|V_{reachable}|}{|V|},$$

Для ігрово придатного рівня значення R повинно дорівнювати одиниці. У випадку використання BSP-алгоритмів ця умова досягається завдяки ієрархічному з'єднанню підобластей, що забезпечує зв'язність усіх сегментів карти.

Баланс складності рівня є ще однією критично важливою метрикою, оскільки безпосередньо впливає на комфорт і зацікавленість гравця. Складність може бути подана як зважена сума основних ігрових факторів, таких як кількість ворогів, пасток та довжина критичного шляху:

$$D = \alpha E + \beta T + \gamma L,$$

де E — кількість ворогів,

T — кількість пасток,

L — довжина основного маршруту,

α, β, γ — коефіцієнти вагомості.

Для оцінки унікальності згенерованих рівнів застосовуються метрики структурної подібності, які порівнюють топологію різних карт. У випадку BSP це може бути визначено через порівняння множин вузлів двох дерев:

$$U = 1 - \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|},$$

де T_1 та T_2 — множини вузлів відповідних BSP-структур.

Окремо розглядається естетична узгодженість рівня з художнім стилем гри. Хоча цей показник має суб'єктивний характер, у сучасних дослідженнях пропонується його часткова формалізація за допомогою ML-класифікаторів, які оцінюють відповідність візуальних патернів заданим еталонам стилю.

1.3 Обґрунтування вибору технологічного стеку

Обґрунтування вибору технологічної основи для дослідження систем процедурної генерації рівнів має спиратися на аналіз функціональних і технічних

вимог, а також на можливість експериментальної перевірки отриманих результатів. Для реалізації BSP-орієнтованої PCG-системи критичними є продуктивність, доступ до низькорівневих API, модульність, а також наявність інструментів візуалізації, логування й профілювання, що забезпечують відтворюваність експериментів і коректну інтерпретацію метрик. З метою формалізації вибору доцільно виконати порівняння поширених рушіїв за релевантними критеріями.

Узагальнене порівняння характеристик Unity, Unreal Engine 5 та Godot Engine як технологічної бази для дослідження процедурної генерації наведено в таблиці 1.3.

Таблиця 1.3 — Порівняння ігрових рушіїв як технологічної основи дослідження

Критерій	Unity	Unreal Engine 5	Godot Engine
Мова основної реалізації	C#	C++	C++ / GDScript
Доступ до низькорівневого API	Обмежений	Повний	Повний
Підтримка BSP-алгоритмів	Обмежена	Повна	Часткова
Вбудовані інструменти PCG	Переважно через плагіни	PCG Framework	Обмежені
Робота з великими 3D-сценами	Середня	Висока	Обмежена
Інструменти профілювання та дебагу	Базові	Розширені	Базові
Візуальна якість рендерингу	Середня	Висока	Середня
Придатність для експериментів	Середня	Висока	Середня

Як видно з таблиці 1.3, Unreal Engine 5 найбільш повно відповідає вимогам дослідження процедурної генерації рівнів, оскільки поєднує високу продуктивність, розвинуті засоби нативної реалізації алгоритмів на C++ та наявність інструментів для експериментального аналізу. Вбудований PCG Framework і можливість комбінування C++-логіки з Blueprint-механізмами забезпечують зручне параметричне налаштування генератора та швидке

тестування різних конфігурацій BSP без надмірного ускладнення робочого процесу.

Додатково UE5 надає розвинені засоби профілювання та налагодження, що дозволяють збирати дані про часові витрати, структуру згенерованих рівнів і поведінку алгоритму за різних параметрів. Підтримка сучасних графічних технологій підсилює можливість візуальної верифікації результатів, що є важливим у контексті комплексної оцінки якості процедурно згенерованих структур [16]. Таким чином, вибір Unreal Engine 5 як технологічної основи є найбільш обґрунтованим для реалізації та експериментального дослідження BSP-орієнтованої системи процедурної генерації рівнів.

1.3.1 Причини використання Unreal Engine 5 для процедурної генерації

Unreal Engine 5 забезпечує комплексну технологічну основу для реалізації процедурної генерації контенту завдяки поєднанню високопродуктивної C++-архітектури, розвинених інструментів інтеграції та промислового рівня візуалізації. Це дозволяє реалізовувати алгоритми процедурної генерації, зокрема Binary Space Partitioning (BSP), як у контексті редактора, так і під час виконання гри, з можливістю подальшого експериментального аналізу результатів [17].

Вбудований вузловий фреймворк PCG у UE5 дозволяє формувати процедурні ланцюжки генерації у вигляді графів, де етапи обробки даних, фільтрації, модифікації та розміщення об’єктів організовані як керована послідовність операцій. Такий підхід спрощує побудову відтворюваних експериментів: параметри генерації можуть змінюватися без порушення загальної структури пайплайна, а результати одразу візуалізуються у вьюпорті редактора. Для BSP це є принциповим, оскільки базова топологія рівня формується рекурсивно, після чого може бути доповнена правилами розміщення об’єктів і обмеженнями, що контролюються на рівні графа. Узагальнену схему компонентів UE5, що підтримують реалізацію BSP-генератора (C++, Blueprint/PCG Graph, профілювання, візуалізація), наведено на рисунку 1.3.

Перевагою UE5 для дослідження процедурної генерації є поєднання нативної реалізації на C++ та швидкого налаштування через Blueprint-інтерфейси.

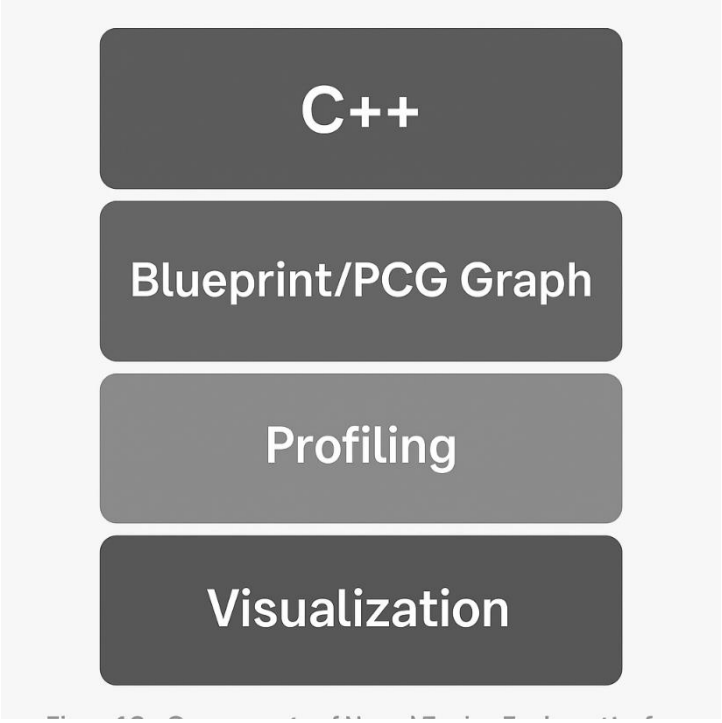

Рисунок 1.3 — Компоненти Unreal Engine 5

У межах BSP-генератора рекурсивний поділ і побудова структури даних доцільно реалізовувати на C++, тоді як параметри (глибина дерева, межі розмірів кімнат, seed) можуть змінюватися без перекомпіляції, що пришвидшує тестування серій конфігурацій і збір метрик [6]. Така схема забезпечує одночасно відтворюваність експериментів і практичну зручність дослідницької роботи.

Додатковим чинником є наявність розвинених засобів профілювання, логування та візуальної діагностики, що дозволяють вимірювати продуктивність генератора та аналізувати його поведінку за різних параметрів. Це є критичним для досліджень, де необхідно зіставляти швидкодію, варіативність і структурні властивості рівнів та обґрунтовувати вибір налаштувань генерації [18]. Високий рівень візуалізації UE5 також спрощує візуальну верифікацію процедурно створених структур і їх придатність до подальшого наповнення ассетами, що важливо для комплексної оцінки результату.

Таким чином, Unreal Engine 5 є доцільною платформою для реалізації та дослідження BSP-орієнтованого генератора рівнів, оскільки поєднує нативну продуктивність C++, інтегровані процедурні інструменти та повний цикл експериментальної перевірки (візуалізація, профілювання, керування параметрами) в межах одного середовища.

1.3.2 Доцільність застосування мови C++ у реалізації генератора рівнів

Мова C++ є базовою складовою архітектури Unreal Engine 5 і забезпечує прямий доступ до його внутрішніх систем, що робить її доцільним вибором для реалізації високопродуктивного генератора рівнів на основі алгоритму Binary Space Partitioning (BSP). Завдяки високій швидкодії виконання, контролю над пам’яттю та тісній інтеграції з ядром рушія, C++ дозволяє реалізовувати процедурні алгоритми без обмежень, характерних для інтерпретованих або віртуалізованих середовищ. Загальну схему взаємодії C++-логіки генератора з візуальними та редакторськими компонентами UE5 наведено на рисунку 1.4.

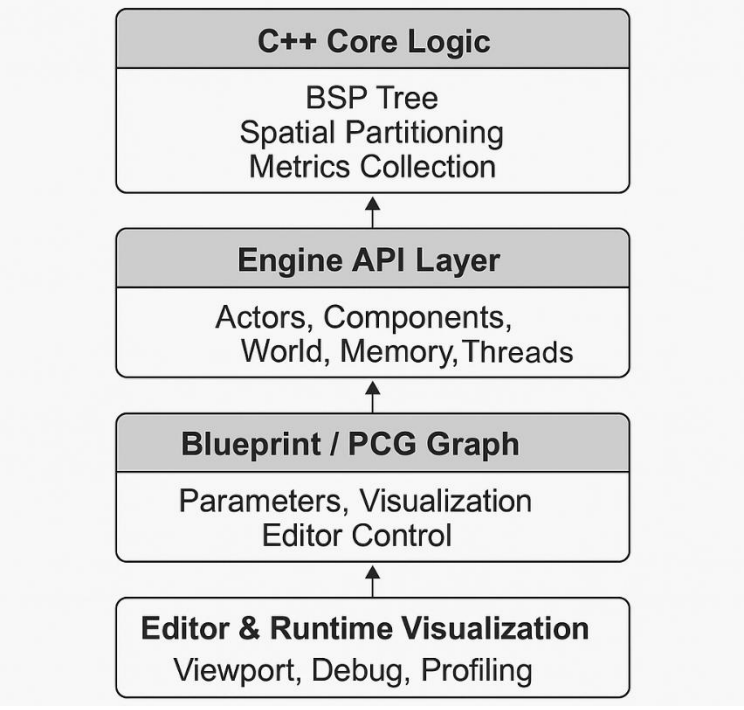

Рисунок 1.4 — Взаємодія C++-реалізації

Однією з ключових причин вибору C++ є його висока обчислювальна ефективність. На відміну від Blueprint-логіки, яка виконується через віртуальну машину, C++-код компілюється безпосередньо у машинні інструкції, що істотно зменшує накладні витрати виконання. Це є критичним для BSP-генератора, який використовує рекурсивні алгоритми поділу простору, обробляє великі масиви геометричних даних і формує зв’язки між численними структурними елементами рівня [19]. Практичні дослідження вказують, що реалізація алгоритмів генерації на C++ у середовищі UE5 демонструє суттєво вищу продуктивність порівняно з візуальними сценаріями.

Використання C++ також забезпечує повний контроль над керуванням пам’яттю, потоками виконання та життєвим циклом об’єктів. Це дозволяє оптимізувати рекурсивні виклики BSP-алгоритму, зменшувати кількість динамічних алокацій та реалізовувати паралельну обробку окремих піддерев або етапів підготовки геометрії. Такі можливості є особливо важливими для процедурних систем, що повинні працювати в режимі реального часу або генерувати великі сцени з високою кількістю елементів.

Значною перевагою C++ є доступ до низькорівневих API Unreal Engine, включно з модулями Core, Engine, Gameplay та Rendering. Це дозволяє створювати власні компоненти й плагіни для процедурної генерації, реалізовувати спеціалізовані актори кімнат і коридорів, а також збирати детальну статистику щодо структури BSP-дерева та часу виконання генерації. Таким чином, C++ формує основу розширюваної архітектури генератора, яка не обмежується стандартними можливостями PCG Graph.

У контексті дослідницької та академічної роботи використання C++ є доцільним також з огляду на підтримку модульної розробки, автоматизованого тестування та інтеграції з системами контролю версій. Це забезпечує відтворюваність експериментів, прозорість реалізації та можливість подальшого масштабування проєкту.

2 МЕТОДИ ТА АЛГОРИТМИ ПРОЦЕДУРНОЇ BSP-ГЕНЕРАЦІЇ

2.1 Загальна архітектура системи процедурної генерації рівнів

Архітектура системи процедурної генерації рівнів в Unreal Engine 5 повинна забезпечувати модульність, масштабованість і гнучкість інтеграції алгоритмів генерації. Основною метою такої архітектури є створення єдиного програмного середовища, яке дозволяє реалізовувати, налаштовувати та експериментально оцінювати різні алгоритмічні підходи, зокрема Binary Space Partitioning (BSP), з можливістю інтерактивної візуалізації результатів у межах Unreal Editor [20].

Система проєктується за принципом багаторівневої модульності, де кожен компонент виконує окрему функцію, а обмін даними між модулями здійснюється через чітко визначені інтерфейси. Такий підхід забезпечує незалежність алгоритмів генерації, повторне використання компонентів і можливість розширення системи без модифікації базової логіки. Узагальнену структуру архітектури процедурної генерації рівнів наведено на рисунку 2.1.

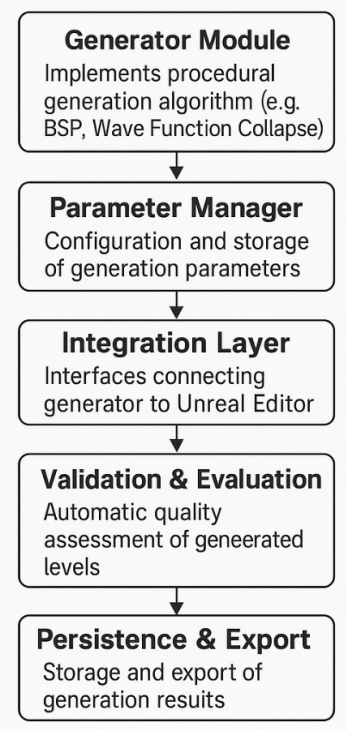

Рисунок 2.1 — Архітектура системи процедурної генерації рівнів

Центральним елементом архітектури є модуль генерації, який реалізує обраний алгоритм процедурної побудови рівня. У випадку BSP цей модуль відповідає за рекурсивний поділ простору, формування кімнат і переходів між ними, а також за підтримку як офлайн-генерації в редакторі, так і генерації під час виконання гри. В Unreal Engine такий модуль доцільно реалізовувати у вигляді C++-компонента з можливістю виклику з Blueprint або PCG Graph [21].

Налаштування параметрів генерації виконується через окремий модуль керування параметрами, який забезпечує централізований доступ до конфігураційних значень, таких як розміри рівня, глибина рекурсії BSP-дерева та коефіцієнти випадковості. Це дозволяє змінювати поведінку генератора без перекомпіляції коду та використовувати різні профілі параметрів для експериментального аналізу.

Інтеграція з Unreal Editor реалізується через спеціальний шар взаємодії, що відповідає за візуалізацію результатів генерації, запуск алгоритмів у реальному часі та взаємодію з Blueprint і PCG Graph. Завдяки цьому згенеровані структури можуть бути безпосередньо перевірені у вьюпорті редактора, що суттєво прискорює процес тестування та налагодження.

Для автоматизованої перевірки результатів у систему включається модуль оцінки якості, який аналізує згенеровані рівні за заданими метриками, такими як прохідність, варіативність, баланс складності та просторова ефективність. Результати оцінювання можуть використовуватися як для візуального аналізу, так і для подальшої оптимізації алгоритмів генерації.

2.1.1 Принципи побудови модульної архітектури PCG-системи

Модульна структура програмного комплексу системи процедурної генерації рівнів визначає її гнучкість, масштабованість і придатність до подальшого розширення. У середовищі Unreal Engine 5 така система доцільно реалізується як сукупність незалежних, але взаємопов’язаних модулів, побудованих на основі C++-логіки, Blueprint-компонентів і вузлових графів PCG. Кожен модуль виконує окрему функцію та взаємодіє з іншими через чітко визначені інтерфейси, що

забезпечує слабку зв’язаність компонентів і спрощує підтримку системи. Загальну структуру модульної організації програмного комплексу наведено на рисунку 2.2.

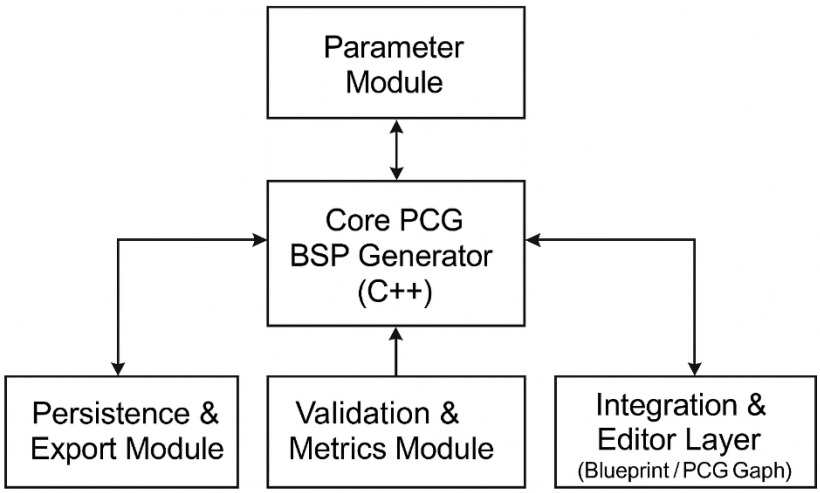

Рисунок 2.2 — Модульна структура програмного комплексу

Центральним елементом системи є модуль генерації, який реалізує основний алгоритм процедурної побудови рівня. У межах даної роботи таким алгоритмом є Binary Space Partitioning, що відповідає за рекурсивний поділ простору, формування кімнат і переходів між ними. У Unreal Engine цей модуль доцільно реалізовувати у вигляді C++-компонента з можливістю виклику з Blueprint або інтеграції у PCG Graph, що дозволяє поєднати високу продуктивність із візуальним контролем результатів [22].

Керування параметрами генерації здійснюється через окремий модуль налаштувань, який забезпечує централізований доступ до конфігураційних значень, таких як глибина рекурсії BSP-дерева, межі розмірів кімнат, коефіцієнти випадковості та стилістичні обмеження. Відокремлення параметрів від логіки генерації дозволяє змінювати поведінку алгоритму без перекомпіляції коду та використовувати різні профілі конфігурацій для експериментального аналізу.

Інтеграція генератора з Unreal Editor реалізується через спеціальний інтеграційний модуль, який відповідає за візуалізацію результатів у вьюпорті, запуск генерації в реальному часі та взаємодію з Blueprint і PCG Graph. Це дає змогу досліднику або дизайнеру оперативно змінювати параметри й одразу

оцінювати вплив цих змін на структуру рівня, що суттєво прискорює процес тестування.

Для контролю якості згенерованих рівнів у систему включається модуль валідації та тестування, який виконує автоматичну перевірку результатів за визначеними метриками, такими як прохідність, варіативність і баланс складності. Результати аналізу можуть використовуватися як для візуального контролю, так і для подальшої оптимізації алгоритмів генерації.

Окремий модуль збереження та експорту забезпечує фіксацію результатів генерації у вигляді Unreal-ассетів або структурованих даних, придатних для повторного використання й порівняльного аналізу. Це дозволяє забезпечити відтворюваність експериментів і використовувати отримані результати у подальших дослідженнях.

2.1.2 Логічна взаємодія модулів генерації, геометрії та візуалізації

Система процедурної генерації рівнів на основі алгоритму Binary Space Partitioning (BSP) в Unreal Engine 5 побудована як послідовний конвеєр взаємодії між трьома ключовими модулями: BSP-генерації, геометрії та візуалізації. Такий підхід забезпечує чіткий розподіл відповідальностей, спрощує масштабування системи та дозволяє відокремити логіку генерації від етапів побудови й відображення геометрії, що зображено на рисунку 2.3.

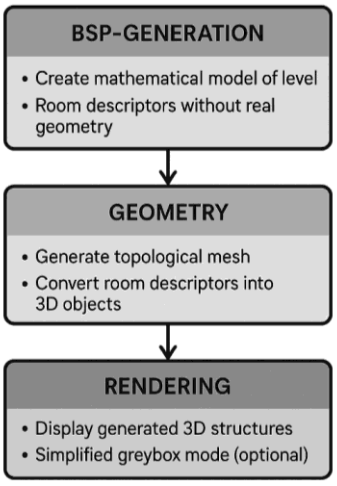

Рисунок 2.3 — Схема взаємодії між модулями

Модуль BSP-генерації виконує початковий етап побудови рівня, реалізуючи рекурсивний поділ вихідного простору на підобласті. У результаті формується ієрархічна структура BSP-дерева, де кожен листовий вузол описує логічні межі кімнати або коридору. На цьому етапі створюється абстрактна модель рівня, що містить лише просторові параметри та топологічні зв’язки без реальної геометрії [23].

Дані, сформовані BSP-модулем, передаються до модуля геометрії, який відповідає за перетворення логічної структури у тривимірну топологічну сітку. Тут виконуються побудова мешів, генерація колізій та оптимізація геометрії для подальшого рендерингу. Модуль геометрії виступає проміжною ланкою між алгоритмічною моделлю простору та її фізичним представленням у сцені Unreal Engine.

Завершальним етапом є модуль візуалізації, який забезпечує відображення згенерованої геометрії у вьюпорті рушія. На цьому рівні застосовуються матеріали, освітлення та режими оптимізації, зокрема greybox-візуалізація для швидкої оцінки топології рівня. Модуль підтримує автоматичне оновлення сцени при зміні параметрів генерації, що забезпечує інтерактивність і зручність тестування.

Взаємодія між модулями реалізується через подієву модель Unreal Engine: після завершення генерації BSP-структури ініціюється оновлення геометрії, а після її побудови — перерендеринг сцени. Така схема дозволяє організувати асинхронну обробку, зменшити зв’язність компонентів і забезпечити стабільну роботу системи навіть для великомасштабних рівнів.

2.2 Алгоритм двійкового розбиття простору як основа генерації рівнів

Алгоритм Binary Space Partitioning (BSP) є одним із базових методів процедурної генерації рівнів, який широко застосовується для побудови логічно впорядкованих ієрархічних просторів. Його основна ідея полягає у рекурсивному поділі початкової області на підпростори за допомогою роздільних ліній або

площин, у результаті чого формується деревоподібна структура на рисунку 2.4, що описує топологію майбутнього рівня [24].

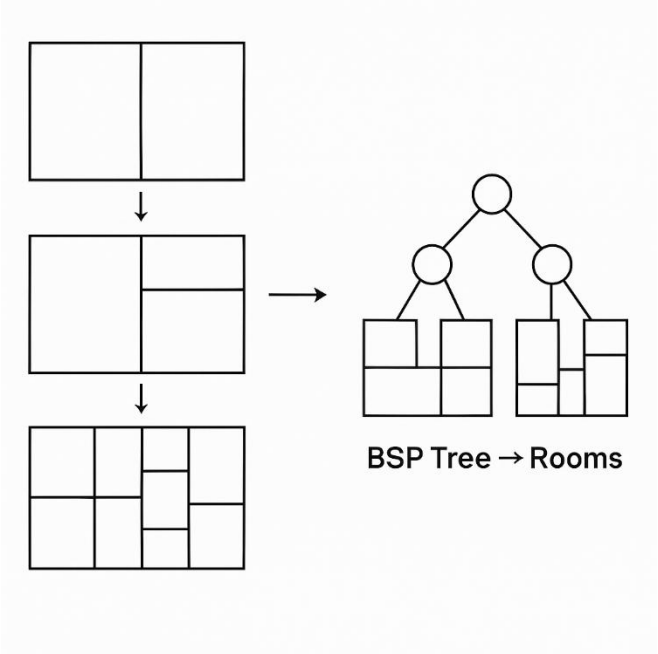

Рисунок 2.4 — Схема роботи алгоритму

У процесі роботи BSP вихідний простір послідовно ділиться на дві частини відповідно до вибраного напрямку поділу. Кожен такий поділ породжує вузол BSP-дерева, тоді як листові вузли відповідають кінцевим просторовим областям — кімнатам або секціям рівня. Рекурсія триває до досягнення заданої глибини або мінімальних допустимих розмірів підпросторів. Таким чином формується ієрархічна структура, що зберігає інформацію про межі областей, напрям поділу та взаємозв’язки між секторами.

Узагальнено BSP-алгоритм включає ініціалізацію початкового простору, вибір лінії або площини поділу, формування двох підпросторів і рекурсивне повторення цього процесу для кожної з отриманих областей. Після завершення поділу листові вузли інтерпретуються як кімнати, між якими створюються коридори для забезпечення зв’язності рівня. Отримане BSP-дерево є зручним представленням для подальшої генерації геометрії та аналізу структури простору.

Перевагою BSP є формування логічно зв’язаних і передбачуваних структур, що особливо важливо для підземель, технічних приміщень та багатокімнатних

локацій. На відміну від шумових або клітинних методів, BSP забезпечує чітку архітектурну організацію простору. Варіативність досягається за рахунок випадкового вибору напрямків поділу та параметрів рекурсії, що дозволяє уникати повної шаблонності рівнів.

У середовищі Unreal Engine 5 алгоритм BSP реалізується як окремий C++-модуль у складі системи процедурної генерації. Генератор формує абстрактну логічну модель рівня, яка надалі передається до модуля геометрії для побудови тривимірної сітки та до модуля візуалізації для відображення результатів у сцені рушія. Такий поділ дозволяє відокремити алгоритмічну логіку від етапів побудови й рендерингу геометрії.

Разом із тим, BSP має певні обмеження, пов’язані з тенденцією до створення прямолінійних структур і можливим надмірним дробленням простору при великій глибині рекурсії. Ці недоліки компенсуються введенням обмежень на розміри кімнат, контроль глибини дерева та використанням стохастичних коефіцієнтів під час вибору площин поділу.

2.2.1 Концептуальні та теоретичні засади методу BSP

Метод Binary Space Partitioning (BSP) є алгоритмічною моделлю ієрархічного поділу простору, що застосовується для формування структурованих областей з чітко визначеними просторовими межами. Первинно розроблений для задач комп’ютерної графіки, зокрема оптимізації рендерингу, BSP згодом набув широкого використання у процедурній генерації ігрових рівнів, де необхідна логічна організація кімнат, коридорів і зон.

З формальної точки зору BSP описується як рекурсивне розбиття простору на підмножини за допомогою роздільних гіперплощин. На кожному кроці вихідна область ділиться на дві частини, які в подальшому можуть бути розбиті аналогічним чином. У результаті формується деревоподібна структура, в якій внутрішні вузли відповідають операціям поділу, а листові — кінцевим просторовим секторам. Таке представлення дозволяє компактно зберігати ієрархічні зв’язки між елементами рівня.

Алгоритмічна реалізація BSP починається з визначення початкової області, після чого обирається напрям поділу та позиція розрізу. Процес повторюється рекурсивно доти, доки не буде досягнуто заданих обмежень за глибиною або мінімальними розмірами підпросторів. Отримане BSP-дерево використовується як основа для побудови геометрії, розміщення ігрових об’єктів і формування зв’язків між кімнатами.

Ключовою перевагою BSP є збереження топологічної впорядкованості простору. Оскільки кожна область створюється в межах свого батьківського сектора, рівень набуває логічної структури, яку легко перетворити у граф зв’язності або тривимірну сітку. Це забезпечує баланс між передбачуваністю архітектури та варіативністю результатів, що досягається за рахунок параметризації процесу поділу.

Завдяки рекурсивній природі BSP добре масштабується і дозволяє виконувати операції перевірки належності просторових елементів за логарифмічний час. Це робить метод ефективним для систем процедурної генерації, які мають працювати в реальному часі або створювати великі ігрові структури без значного навантаження на обчислювальні ресурси.

Параметризація BSP — зокрема керування глибиною рекурсії, співвідношеннями сторін та мінімальними розмірами секторів — надає дизайнеру або досліднику інструмент контролю архітектури рівня. Завдяки цьому BSP виступає не лише як технічний алгоритм, а як механізм формування ігрового простору відповідно до заданих функціональних і дизайнерських вимог.

2.2.2 Процес поділу простору та формування підобластей

Формалізація процесу поділу простору є центральним етапом у побудові BSP-алгоритму, оскільки саме на цьому кроці визначаються математичні правила рекурсії, критерії припинення поділу та параметри, що впливають на топологічну структуру майбутнього рівня. Метод Binary Space Partitioning (BSP) ґрунтується на принципі рекурсивного розділення області Sна дві підобласті S_1та S_2, які є неперетинними, але разом утворюють початковий простір:

$$S = S_1 \cup S_2, S_1 \cap S_2 = \emptyset.$$

Для кожної області S_iвизначається площина або лінія поділу (у двовимірному випадку — відрізок), яка задається рівнянням:

$$L: f(x, y) = ax + by + c = 0,$$

Вибір площини може бути детермінованим (згідно з пропорцією сторін області, фіксованим співвідношенням тощо) або стохастичним (із додаванням випадкових зсувів для досягнення варіативності).

У процедурному генераторі рівнів, зокрема при реалізації в Unreal Engine 5, доцільно застосовувати стохастичний підхід із контролем діапазону відхилення. Це дозволяє уникати надмірної симетрії та створювати більш “живі” архітектурні форми. Такий спосіб забезпечує баланс між структурованістю та природною варіативністю, що є ключовим для ігрового дизайну [25].

На першому кроці простір S_0, який визначає межі всього рівня, поділяється площиною L_0на дві частини — S_1та S_2. Кожна з них стає новим вузлом дерева BSP і далі розглядається як окрема область для поділу. Процес повторюється рекурсивно, доки виконуються умови подальшого розбиття:

$$\text{while } \min(W_i, H_i) > \varepsilon \text{ and } d < d_{max},$$

де W_i, H_i— ширина та висота поточної області,

ε— мінімальний допустимий розмір,

d— поточна глибина рекурсії,

d_{max}— максимальна глибина дерева.

Таким чином, дерево BSP будується зверху вниз: кореневий вузол відповідає всій області карти, а листи — кінцевим кімнатам або секціям рівня. Кожен вузол зберігає власні параметри поділу, зокрема:

— орієнтацію площини (вертикальну чи горизонтальну);

— позицію розділу ру діапазоні $[0,1]$відносно ширини або висоти області;

— випадковий коефіцієнт варіації δ, який контролює зсув від центру;

— глибину dy дереві розбиття;

— посилання на дочірні вузли L(ліва підобласть) і R(права підобласть).

Формально стан кожного вузла BSP можна записати як:

$$N_i = \{R_i, O_i, p_i, \delta_i, d_i, L_i, R_i\},$$

де R_i— область, що описується координатами (x, y, width, height),

$O_i \in \{V, H\}$— орієнтація поділу (Vertical/Horizontal).

Ці параметри задають не лише глибину дерева, а й “архітектурний стиль” згенерованого рівня: менша глибина створює великі зали або відкриті простори, більша — густу мережу коридорів і кімнат.

Після завершення розбиття листові вузли BSP-дерева інтерпретуються як кімнати, а їхні межі — як координати об’єктів рівня. Для підключення кімнат між собою використовується система зв’язків (corridor linking), що аналізує пари сусідніх вузлів. Коридори створюються або на основі спільних граней областей, або через вибір центру кожної кімнати та побудову прямої лінії між ними. У сучасних реалізаціях BSP-генераторів (зокрема в Unreal Engine) застосовується підхід із “фіксованими сегментами” — коридори формуються через середину площини поділу, що дозволяє уникати перетинів і забезпечує коректну навігацію.

Таким чином, BSP не лише розділяє простір, а й створює топологічний граф з’єднань, який зберігає інформацію про сусідні кімнати, відстані та доступні шляхи. Це значно спрощує подальшу побудову геометрії та розміщення об’єктів, оскільки дозволяє одразу формувати логічну ігрову структуру рівня [88].

Функцію поділу $\mathrm{Split}(S_i)$можна подати у загальному вигляді:

$$\mathrm{Split}(S_i) = \begin{cases} \{S_{i1}, S_{i2}\}, & \text{if } W_i > \varepsilon \text{ and } H_i > \varepsilon \text{ and } d_i < d_{max}, \\ \mathrm{Room}(S_i), & \text{otherwise.} \end{cases}$$

Рекурсивне виконання цієї функції утворює дерево:

$$T = \{N_0, N_1, N_2, \dots, N_k\},$$

де N_0— кореневий вузол,

N_k— кінцеві кімнати.

Математична формалізація процесу поділу дозволяє легко реалізувати BSP у вигляді узагальненого програмного модуля в Unreal Engine 5. Всі параметри — глибина, орієнтація, стохастичні коефіцієнти, мінімальні розміри — можуть бути задані як змінні у Blueprint або як поля конфігурації C++ класу UBSPGenerator. Це дає можливість швидко змінювати структуру рівня, отримуючи різні варіанти карти без перезапуску системи. Таким чином, формалізований підхід робить BSP не просто алгоритмом, а гнучкою моделлю керування архітектурою світу, де математична строгість поєднується з інтерактивною візуалізацією в середовищі рушія.

2.2.3 Побудова BSP-дерева та визначення просторової топології

Побудова дерева BSP є ключовим етапом у процесі процедурної генерації рівнів, оскільки саме ця структура визначає просторову організацію сцени, логіку переходів між кімнатами та загальну топологію ігрового простору. BSP-дерево слугує внутрішньою моделлю даних, яка описує геометрію не у вигляді полігонів, а у вигляді ієрархії областей, що логічно розподіляють рівень на частини, пов’язані між собою правилами підпорядкування та суміжності [26].

BSP-дерево формується рекурсивно згідно з принципом “зверху вниз”. Кореневий вузол дерева відповідає всій області рівня, яка послідовно розділяється на дві менші частини. Для кожного вузла зберігається набір параметрів, що

описують геометричні межі підобласті (координати лівої верхньої точки, ширина, висота), орієнтацію площини поділу (вертикальну або горизонтальну), а також індекси дочірніх елементів. У результаті формується структура, що може бути представлена як впорядкована множина:

$$T = \{N_0, N_1, N_2, \dots, N_n\},$$

де кожен вузол N_iвизначається як:

$$N_i = \{R_i, d_i, O_i, L_i, R_i\},$$

де R_i— межі області,

d_i— глибина у дереві,

O_i— напрям поділу,

L_i, R_i— посилання на дочірні вузли.

Таким чином, дерево BSP є повним бінарним деревом, у якому листові елементи відповідають кінцевим кімнатам або секціям рівня, а внутрішні вузли — площинам розбиття [27].

Після завершення рекурсивного поділу структура дерева зберігається у вигляді об’єктів або записів масиву, які містять координати й логічні зв’язки між елементами. У контексті Unreal Engine 5 така структура реалізується у вигляді моделі даних BSPTree, що містить масив структур FBspNode. Кожна структура зберігає інформацію про положення підобласті у просторі, межі, напрям поділу, а також вказівники на дочірні елементи. Це забезпечує швидкий обхід дерева та зручність серіалізації даних для подальшого використання у геометричному модулі [28].

На основі побудованого дерева BSP визначається топологія рівня — сукупність усіх областей (кімнат, коридорів, зон) та їхніх просторових і логічних зв’язків. Ключовим поняттям тут є суміжність областей, що означає, що дві кімнати або зони мають спільну межу, через яку гравець або агент може переміщатися.

У практичній реалізації, після завершення рекурсії BSP, для кожної пари листових вузлів виконується перевірка на перетин меж, і якщо спільна грань існує — створюється з’єднання. У такий спосіб формується граф проходження, який описує всі можливі маршрути у згенерованому рівні. Залежно від налаштувань генератора, ці переходи можуть бути прямими (лінійні коридори), перпендикулярними (класичні з’єднання кімнат), або комбінованими (із випадковими зсувами та вигинами).

Визначення топології має не лише просторове, але й ігрове значення — вона задає логіку переміщення гравця, розміщення цілей, ворогів і точок інтересу. Зокрема, на основі графа суміжності можна реалізувати пошук шляху (pathfinding), виявлення “глухих” зон, визначення головних і другорядних маршрутів, що особливо важливо для балансування складності гри.

На етапі побудови фізичної структури рівня BSP-дерево трансформується у систему кімнат і коридорів. Для кожного вузла, який має два дочірні підвузли, визначається лінія поділу, що одночасно є місцем з’єднання обох підобластей. Ця лінія стає “центральною віссю” майбутнього коридору, який з’єднує відповідні кімнати. Зазвичай ширина коридору обчислюється як функція від середніх розмірів підобластей:

$$w_c = k \cdot \min(W_i, H_i),$$

де $k \in [0.05, 0.2]$— коефіцієнт масштабування.

У випадку стохастичної генерації параметри з’єднань можуть бути зміщені випадковим чином, щоб уникнути надмірної регулярності та надати карті більш природний вигляд.

В Unreal Engine 5 цей етап реалізується через створення Blueprint- або C++-компонентів типу CorridorActor, які будуються між координатами центрів двох суміжних кімнат. Коридори створюються автоматично після завершення побудови BSP-структури, а їхні параметри (ширина, матеріал, тип освітлення) задаються динамічно з конфігураційних змінних у редакторі.

Окрім геометричних властивостей, BSP-дерево дозволяє додавати семантичні рівні значень, перетворюючи чисту топологію на логічну ігрову структуру. Кожен вузол дерева може зберігати додаткові атрибути: тип кімнати (звичайна, технічна, центральна), рівень важливості, зв’язок із геймплейними подіями. Це робить BSP не лише засобом геометричного поділу, а й моделлю ігрової логіки, що зручно для системи сценарної генерації чи адаптивної складності. Так, наприклад, вузол із найбільшою глибиною дерева може бути інтерпретований як “кінцева точка”, а вузол поблизу кореня — як “початкова зона”. На основі цієї інформації можливо автоматично розташовувати стартові позиції гравців, ворогів чи об’єктів інтересу.

Завдяки поєднанню рекурсивної природи та структурної впорядкованості BSP забезпечує низку важливих переваг у контексті процедурної генерації:

— оптимальне представлення простору — дозволяє зберігати складні рівні у компактній ієрархічній формі;

— контрольованість і передбачуваність — легко регулювати масштаб, складність і щільність структури;

— зручність інтеграції з рушієм — дані BSP можна безпосередньо використовувати для побудови геометрії, колізій та навігаційних карт;

— гнучкість сценарного моделювання — на основі дерева можна реалізовувати адаптивні сценарії, змінюючи правила генерації в реальному часі.

Таким чином, BSP виступає не лише як математичний алгоритм поділу простору, а як універсальна топологічна модель, що лежить в основі процедурної архітектури рівнів. Його застосування забезпечує баланс між структурною логікою, ігровим дизайном та ефективністю реалізації у рушії Unreal Engine 5.

2.3 Розширення та адаптація базового алгоритму BSP

Класичний алгоритм Binary Space Partitioning (BSP), попри простоту реалізації та високу обчислювальну ефективність, має низку обмежень, що знижують його придатність для створення складних і динамічних ігрових рівнів. У базовій формі BSP не враховує семантичні характеристики простору, не

адаптується до геймплейних або сценарних вимог і часто формує надмірно регулярні та передбачувані структури. У зв’язку з цим у сучасних системах процедурної генерації все частіше застосовуються модифіковані або гібридні варіанти цього підходу.

До основних недоліків традиційного BSP належить відсутність контекстної адаптації, коли поділ простору виконується незалежно від логіки ігрового процесу або призначення окремих зон. Крім того, при фіксованих параметрах рекурсії алгоритм схильний до створення симетричних і геометрично одноманітних планувань, що негативно впливає на варіативність ігрового досвіду. Також класичний BSP обмежено підтримує багаторівневу організацію простору, що ускладнює побудову вертикально складних рівнів із поверхами або підземними зонами.

Для подолання цих обмежень у практиці процедурної генерації застосовуються удосконалені варіанти BSP, які включають стохастичні механізми вибору напрямів і точок поділу. Використання випадкових відхилень дозволяє зменшити симетричність результатів і сформувати більш природні, асиметричні структури, зберігаючи при цьому керованість архітектури рівня.

Ще одним важливим напрямом розвитку BSP є підтримка багаторівневого простору. Розширення алгоритму на вертикальну координату дозволяє формувати багатоповерхові або підземні структури, а також реалізовувати переходи між рівнями. Такий підхід особливо актуальний для сучасних тривимірних ігрових середовищ, де вертикальність є важливою складовою дизайну.

Контроль складності та збалансованості рівнів досягається шляхом введення додаткових критеріїв оцінки якості поділу простору. Замість випадкового розбиття система може враховувати бажані розміри кімнат, кількість переходів і загальну структурну складність рівня. Це дозволяє формувати простори, що відповідають заданій кривій складності та дизайнерським вимогам.

Подальше розширення можливостей BSP пов’язане з введенням семантичних характеристик у вузли дерева розбиття. Кожен сектор може містити інформацію про своє функціональне призначення, рівень складності або

допустимі ігрові події. У такому вигляді BSP-структура перестає бути суто геометричною моделлю і перетворюється на логічну основу ігрового світу.

Для використання у великомасштабних або динамічних проєктах BSP також поєднується з паралельними обчисленнями та сучасними засобами оптимізації. Незалежність окремих підрозділів простору дозволяє виконувати генерацію асинхронно, що робить можливим застосування алгоритму навіть у режимі реального часу.

Окрему роль відіграють гібридні підходи, які поєднують BSP з іншими методами процедурної генерації, такими як клітинні автомати, шумові функції, Wave Function Collapse або алгоритми машинного навчання. Такі комбінації дозволяють зберегти структурну впорядкованість BSP і водночас підвищити різноманітність, природність і адаптивність згенерованих рівнів.

2.3.1 Метод генерації кімнат і коридорів на основі BSP-структури

Важливим етапом реалізації системи процедурної генерації рівнів на основі Binary Space Partitioning є побудова кімнат і коридорів, які формують просторову структуру рівня, маршрути переміщення гравця та логіку проходження. У модифікованій BSP-системі генерація цих елементів поєднує рекурсивну ієрархію дерева поділу з топологічною логікою з’єднань, що дозволяє отримати структурно цілісний і водночас варіативний ігровий простір.

Після завершення рекурсивного поділу простору BSP-алгоритм формує множину листових вузлів, які інтерпретуються як потенційні кімнати. Кожна кімната створюється всередині відповідної підобласті з урахуванням внутрішніх відступів, що запобігає перетину геометрії та забезпечує наявність стін і проходів між приміщеннями. Для кожної кімнати зберігаються базові атрибути, зокрема просторові межі, центральна точка та ідентифікатор, що використовується для побудови зв’язків.

Алгоритм генерації умовно поділяється на кілька послідовних етапів. Спочатку виконується ініціалізація параметрів рівня, зокрема розмірів області, глибини BSP-дерева та мінімальних допустимих розмірів кімнат. Далі формується

BSP-структура шляхом рекурсивного поділу простору, після чого кожен листовий вузол перетворюється на реальну кімнату з урахуванням геометричних і візуальних характеристик.

Побудова коридорів виконується на основі ієрархії BSP-дерева. Для кожного внутрішнього вузла створюється з'єднання між кімнатами, що належать його дочірнім підобластям. Коридори прокладаються відповідно до напрямку поділу простору, що забезпечує логічну та передбачувану топологію рівня. Для підвищення природності планування використовуються як прямі, так і ламані маршрути, що зменшує візуальну одноманітність структури.

На завершальному етапі кімнати та коридори об'єднуються у єдину топологічну модель рівня, яка передається до модуля візуалізації. У середовищі Unreal Engine 5 це реалізується через створення відповідних акторів для кімнат і переходів, а також інструментів попереднього перегляду, що дозволяють аналізувати зв'язність і структуру рівня безпосередньо в редакторі.

2.3.2 Налаштування параметрів поділу та керування структурою рівня

У методі двійкового розбиття простору (BSP) ключовим етапом є визначення параметрів, що контролюють процес поділу — зокрема мінімальних і максимальних розмірів підобластей, глибини рекурсії та ймовірності поділу. Від цих характеристик залежить структурна складність рівня, його масштаб, баланс між відкритими просторами та вузькими проходами, а також загальна динаміка геймплею. У контексті процедурної генерації рівнів в Unreal Engine 5 ці параметри відіграють центральну роль, оскільки визначають обсяг створюваного контенту, рівень варіативності та навантаження на систему під час побудови світу [24; 59; 85].

BSP-алгоритм базується на послідовному поділі простору доти, доки виконуються певні умови — мінімальні розміри кімнати, допустима глибина дерева або випадкова ймовірність розгалуження. Якщо хоча б одне з обмежень порушується, вузол поділу вважається фінальним, і підобласть перетворюється на окрему кімнату або сегмент рівня. Такий підхід забезпечує баланс між строгістю

геометричної структури й елементом стохастичності, який додає природності та різноманіття у результаті [29].

Розміри підобластей визначають масштаб рівня та співвідношення між його структурними елементами. Мінімальні розміри (W_{min}, H_{min}) встановлюють поріг, нижче якого підобласть не може бути поділена, щоб уникнути утворення надто малих або нефункціональних кімнат. Такі обмеження важливі для ігрової логіки, оскільки забезпечують достатній простір для розміщення персонажа, об’єктів або проходів.

Максимальні розміри (W_{max}, H_{max}), навпаки, регламентують масштаб областей, які повинні бути розбиті для підтримання структурної рівномірності рівня. Надто великі кімнати знижують щільність геймплейних подій і створюють дисбаланс у складності карти.

Співвідношення між мінімальним і максимальним розміром зазвичай визначається експериментально — у межах від 1:3 до 1:5 залежно від жанру гри. У системах типу dungeon crawler, наприклад, доцільно мати менші кімнати й більшу кількість поділів, тоді як для відкритих арен — навпаки [30].

Глибина дерева поділу (d_{max}) визначає максимальну кількість послідовних розбиттів простору. Вона безпосередньо впливає на кількість кінцевих елементів — кімнат і коридорів — у рівні. Оптимальна глибина залежить від початкового розміру карти та бажаної складності. Для невеликих арен зазвичай достатньо 3–5 рівнів розбиття, тоді як для великих підземель або будівель — 6–8.

Надмірне збільшення глибини може призвести до перенасичення карти дрібними кімнатами, зниження продуктивності генератора та утворення складних, нечитабельних маршрутів. Тому в системі передбачено обмеження, що автоматично припиняє рекурсію після досягнення певної кількості елементів або мінімального розміру підобласті [31].

Ймовірність поділу є стохастичним параметром, який регулює непередбачуваність процесу генерації. На відміну від детермінованого алгоритму, де кожен вузол поділяється за фіксованим правилом, у стохастичній моделі рішення про поділ приймається з певною ймовірністю P_{split}. Високі значення цього

параметра призводять до створення складних, насичених структур, тоді як низькі — до появи більш відкритих і простих карт.

Оптимальною вважається змінна ймовірність поділу, яка зменшується зі збільшенням глибини дерева. Це дає можливість формувати великі кімнати на верхніх рівнях і менші — на нижчих, досягаючи природної ієрархічності структури. Такий підхід дозволяє адаптувати щільність поділів до масштабу рівня і забезпечує різноманіття планувань.

Ще одним важливим параметром є положення площини розбиття всередині області. Якщо поділ виконувати завжди строго посередині, структура рівня набуває симетричного, штучного вигляду. Для уникнення цього використовується випадкове зміщення точки поділу, що дозволяє створювати більш природні й асиметричні кімнати. Величина зміщення визначається коефіцієнтом варіативності, який задає максимально допустиме відхилення від центру. У результаті формується баланс між передбачуваністю загальної геометрії та природністю розташування кімнат, що позитивно впливає на візуальну достовірність карти.

Усі вищезазначені параметри є взаємопов’язаними. Мінімальні розміри кімнат визначають нижню межу деталізації, глибина дерева встановлює складність рівня, а ймовірність поділу додає контрольовану стохастичність. Зміна одного з цих параметрів обов’язково впливає на решту, тому важливо забезпечити їх збалансованість.

Наприклад, зменшення мінімального розміру кімнат доцільно поєднувати з меншою ймовірністю поділу, щоб уникнути надмірної фрагментації. Натомість при збільшенні глибини дерева бажано пропорційно зменшити коефіцієнт стохастичності, щоб уникнути хаотичного поділу.

У сучасних реалізаціях BSP-систем у середовищі Unreal Engine ці параметри часто виносяться до конфігураційного інтерфейсу користувача, де дизайнер або дослідник може експериментально підбирати оптимальні значення для конкретного рівня чи жанру гри.

2.3.3 Контроль складності процедурно згенерованих рівнів

У процесі процедурної генерації рівнів ключовим завданням є забезпечення збалансованого співвідношення між ігровою складністю та варіативністю простору. Надто прості карти швидко втрачають інтерес, тоді як надмірно заплутані структури ускладнюють навігацію та негативно впливають на ігровий досвід. Тому в сучасних PCG-системах доцільно застосовувати механізми контролю складності, які адаптують параметри BSP-поділу та з'єднань відповідно до ігрових метрик і заданих дизайнерських цілей.

Модель контролю складності в BSP-системі ґрунтується на принципі зворотного зв'язку між геометрією рівня та його топологічними характеристиками. На кожному етапі генерації аналізується поточна структура BSP-дерева, після чого приймається рішення щодо подальшого поділу простору або його обмеження. При цьому враховуються кількість і розміри кімнат, глибина рекурсії, рівень зв'язності карти, кількість альтернативних маршрутів і цільовий рівень складності, заданий користувачем.

Ігрова складність у BSP-рівнях визначається сукупністю топологічних параметрів. До них належать глибина дерева поділу, що впливає на розмір і кількість приміщень, а також характер з'єднань між ними. Надмірна кількість зв'язків призводить до втрати орієнтації та передбачуваності, тоді як їх нестача формує замкнені або ізольовані ділянки. Додатково враховується середня довжина маршрутів між ключовими зонами, яка безпосередньо впливає на тривалість і темп проходження рівня.

Для автоматизованого контролю складності система використовує узагальнений показник, що інтегрує кількість кімнат, характер розгалуження карти та протяжність маршрутів. Якщо отримане значення виходить за межі допустимого діапазону, алгоритм динамічно коригує параметри BSP-генерації — зокрема глибину поділу та ймовірність створення нових секцій. Завдяки цьому

BSP-алгоритм переходить від статичної геометричної схеми до адаптивної системи, орієнтованої на ігрову логіку.

Варіативність рівнів визначається здатністю генератора створювати різноманітні топологічні конфігурації при зміні параметрів. У BSP-підході вона досягається шляхом стохастичного вибору напрямів поділу, випадкових зміщень площин розбиття та адаптивного керування глибиною дерева. Поєднання цих механізмів дозволяє формувати унікальні карти навіть за однакових початкових умов генерації, що є критично важливим для ігор з високою реіграбельністю.

2.3.4 Підходи до забезпечення ігрового балансу

Процедурна генерація рівнів повинна не лише забезпечувати різноманітність простору, а й підтримувати ігровий баланс — тобто таку просторову організацію, яка сприяє зрозумілій навігації, логічному прогресу та стабільному ігровому ритму. У системах, побудованих на основі методу Binary Space Partitioning (BSP), баланс досягається за рахунок контрольованого розміщення ключових точок, аналізу зв’язності та адаптивного формування маршрутів між кімнатами.

Ігровий баланс у BSP-рівнях розглядається як поєднання структурної впорядкованості та функціональної доступності. Це означає, що всі ключові зони повинні бути досяжними, маршрути — мати адекватну довжину відповідно до складності гри, а важливі області (ресурсні, бойові, сюжетні) — рівномірно розподілятися по простору. На відміну від суто геометричного підходу, акцент робиться на геймплейній топології — тому, як гравець фактично взаємодіє з рівнем.

Розміщення точок входу та виходу визначає основний напрям руху гравця. У BSP-структурі ці точки зазвичай розташовуються в топологічно віддалених підобластях дерева, що формує природний прогрес від простіших зон до складніших. Додатково контролюється уникнення безпосередньої близькості між входом і виходом, а також загальний напрям проходження рівня відповідно до

структури рекурсивного поділу. Такий підхід створює відчуття послідовного “просування” через простір.

Після визначення початкових і фінальних точок система аналізує можливі маршрути між ними на основі зв’язків, сформованих BSP-деревом. Алгоритм оцінює довжину та складність шляхів і, за необхідності, коригує структуру з’єднань, додаючи або видаляючи коридори. Завдяки цьому ігрова складність регулюється динамічно, без жорсткої прив’язки до фіксованих параметрів карти.

Важливою складовою балансу є забезпечення повної досяжності простору. На фінальному етапі генерації виконується перевірка зв’язності, яка гарантує, що кожна кімната має шлях до точки входу. У разі виявлення ізольованих зон система автоматично модифікує структуру рівня, відновлюючи логічну цілісність карти.

Окрему роль відіграє рівномірний розподіл зон підвищеного інтересу — бойових арен, винагород, безпечних областей. Для цього BSP-вузлам призначаються умовні коефіцієнти складності, які враховують їх положення в ієрархії, кількість з’єднань і віддаленість від початку рівня. На основі цих характеристик система визначає наповнення кімнат, формуючи поступове зростання складності від входу до виходу.

2.4 Інтеграція BSP-генератора з рушієм Unreal Engine

Інтеграція модуля двійкового розбиття простору (BSP) з Unreal Engine 5 є ключовим етапом побудови системи процедурної генерації рівнів, оскільки саме вона визначає перехід від абстрактної логічної моделі простору до повноцінного ігрового середовища. Основним завданням такої інтеграції є коректне перетворення результатів BSP-алгоритму — меж кімнат, зв’язків і топології — у об’єкти рушія, придатні для візуалізації, навігації та геймплейної взаємодії.

У розробленій системі BSP-модуль виконує роль логічного ядра, що формує структуру рівня, тоді як Unreal Engine 5 відповідає за її фізичну реалізацію та відображення. Дані, отримані в процесі рекурсивного поділу простору, передаються у рушій у вигляді структурованих описів кімнат і коридорів, після

чого перетворюються на геометричні об’єкти сцени. На цьому етапі формується базова архітектура рівня — стіни, підлога, переходи — яка надалі використовується всіма підсистемами рушія.

Процес інтеграції має поетапний характер: спочатку генерується BSP-структура, далі її результати інтерпретуються як елементи геометрії, після чого відбувається наповнення рівня контентом і автоматичне оновлення навігації. Такий підхід дозволяє забезпечити узгодженість між логічною моделлю простору та його візуальним представленням, а також гарантує прохідність і коректну роботу систем штучного інтелекту.

Взаємодія між BSP-модулем і Unreal Engine реалізується через двосторонній обмін даними. З одного боку, рушій отримує інформацію про структуру рівня, з іншого — передає зворотний зв’язок щодо стану сцени, колізій і доступності маршрутів. Це дозволяє динамічно змінювати параметри генерації без перезавантаження рівня та миттєво відображати результати в редакторі.

Використання PCG Framework у Unreal Engine 5 додатково спрощує інтеграцію BSP-алгоритму, дозволяючи включати його у вузлові графи процедурної генерації. Такий підхід забезпечує візуальне налаштування параметрів, швидке прототипування та комбінування BSP з іншими генеративними методами без зміни програмного коду.

2.4.1 Архітектура обміну даними між генератором і рушієм

Для забезпечення узгодженої взаємодії між модулем генерації BSP і рушієм Unreal Engine 5 використовується спеціалізована архітектура даних, що визначає структуру, формат і напрям передавання інформації між алгоритмічним та візуальним шарами системи. Основною метою такої архітектури є збереження ієрархії BSP-дерева, його зв’язності та просторової коректності під час інтеграції з внутрішніми об’єктами рушія [32].

Архітектура даних виконує роль проміжного шару між генератором, який формує логічну модель простору, та Unreal Engine, що відповідає за її відтворення у 3D-сцені. Загальну схему обміну даними між BSP-модулем і рушієм показано на

рисунку 2.6, де відображено основні компоненти та двосторонній цикл синхронізації між ними.

Ядро системи зберігає інформацію у вигляді BSP-дерева, вузли якого містять просторові межі підобластей, типи зон, ієрархічні зв’язки та службові метадані.

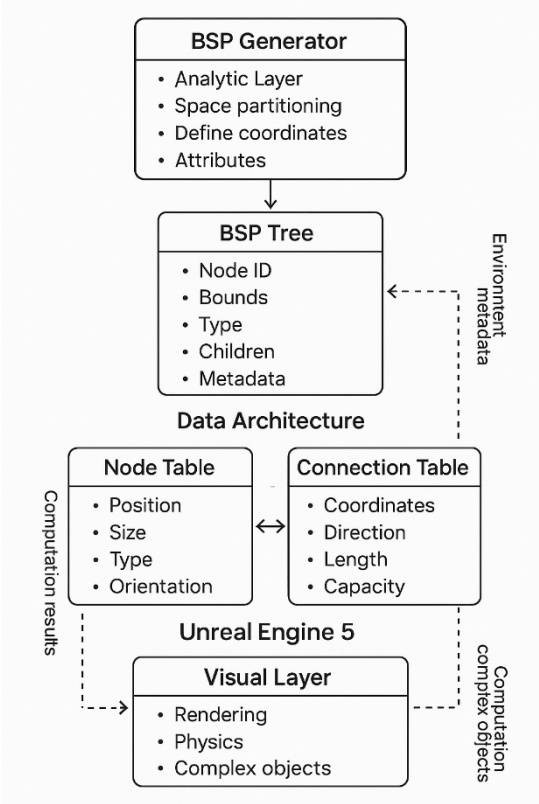

Рисунок 2.6 — Архітектура обміну даними між BSP-генератором та Unreal Engine 5

Після завершення генерації дерево перетворюється у проміжне подання, придатне для інтерпретації рушієм без втрати структури або контексту.

Для передачі результатів у рушій BSP-модуль формує узагальнені таблиці вузлів і з’єднань, які описують логічну топологію рівня — кімнати, коридори та їх взаємозв’язки. Unreal Engine на основі цих даних створює внутрішні об’єкти сцени, зберігаючи відповідність між логічною моделлю та візуальним представленням. Такий підхід дозволяє відокремити алгоритмічну генерацію від геометричної реалізації й уніфікувати обмін між C++-логікою та Blueprint-середовищем.

Перед інтеграцією дані проходять серіалізацію у стандартизованому форматі, що забезпечує стабільність передачі, незалежність від середовища виконання та можливість редагування параметрів без перекомпіляції. У межах Unreal Engine ці дані десеріалізуються та використовуються для побудови геометрії, навігації й візуалізації.

Архітектура підтримує також зворотний зв’язок: рушій передає генератору інформацію про стан сцени, доступність маршрутів і зміни, внесені користувачем у редакторі. На основі цих даних BSP-модуль може локально перебудовувати окремі сегменти дерева без повної регенерації рівня, що забезпечує інкрементальне оновлення та роботу в режимі реального часу.

2.4.2 Алгоритм розміщення ігрових об’єктів у BSP-структурі

Алгоритм розміщення ігрових об’єктів на основі BSP-структури є одним із ключових етапів процедурної генерації рівнів у рушії Unreal Engine 5. Його головна мета полягає у перетворенні абстрактної топологічної моделі, отриманої в результаті роботи BSP-алгоритму, у повноцінний ігровий простір із логічно розташованими кімнатами, коридорами, точками взаємодії, світловими джерелами та іншими елементами, які формують ігровий досвід. У межах цієї системи BSP-структура виступає каркасом, на основі якого здійснюється геометричне, функціональне та естетичне наповнення сцени.

Процес розміщення об’єктів починається після завершення побудови дерева BSP. Кожен вузол дерева, який представляє окрему область простору, інтерпретується як потенційна кімната або зона ігрової активності. На цьому етапі система аналізує тип вузла, його розміри, позицію відносно сусідніх областей та роль у загальній топології рівня. На основі цих параметрів приймається рішення про те, які об’єкти мають бути розташовані всередині цієї кімнати. Наприклад, вузли, що перебувають ближче до входу, можуть містити базові елементи навігації, тоді як глибші вузли дерева використовуються для розміщення об’єктів підвищеної складності, таких як вороги, пастки чи важливі ресурси.

Ключовою особливістю даного алгоритму є використання системи правил та вагових коефіцієнтів, які визначають ймовірність появи певного типу об’єкта у конкретній зоні. Ці правила базуються на характеристиках BSP-вузлів: площі, глибині у дереві, кількості сусідніх кімнат, доступності та зв’язності з основними маршрутами. Наприклад, вузли з великою кількістю з’єднань можуть бути визначені як “центральні зони”, де доцільно розташовувати ключові об’єкти або бос-арени, тоді як вузли на периферії дерева можуть використовуватись для розміщення бонусів або прихованих локацій. Таким чином, топологічна інформація BSP безпосередньо впливає на розподіл ігрового контенту, забезпечуючи структурну логіку та варіативність рівня.

У рушії Unreal Engine 5 реалізація цього алгоритму здійснюється через комбінацію C++-логіки та PCG Graph. BSP-модуль генерує набір параметрів для кожної кімнати — координати, розміри, категорію складності — які передаються до системи розміщення як вхідні дані. Далі рушій створює об’єкти типу AActor або Instanced Static Mesh згідно з заданими умовами. Завдяки можливостям PCG Graph, розташування елементів може бути не фіксованим, а залежним від процедурних фільтрів — наприклад, щільність розміщення меблів або джерел світла визначається на основі площі кімнати чи її положення в ієрархії BSP. Такий підхід дозволяє автоматично формувати унікальні комбінації об’єктів навіть при повторному запуску генерації.

Окрему роль у роботі алгоритму відіграє корекція колізій та адаптація просторового розміщення. Після первинного розташування об’єктів рушій проводить аналіз перетинів між геометричними елементами сцени. У разі виявлення конфліктів система виконує локальні зсуви або видалення об’єктів, щоб уникнути накладань. Додатково, під час генерації виконується побудова навігаційної сітки, яка забезпечує можливість коректного пересування гравця й NPC у створеному просторі. Ця сітка динамічно оновлюється після кожної зміни структури BSP або повторної генерації рівня, що гарантує актуальність даних про прохідність.

Важливим аспектом є інтеграція ігрової логіки з геометричною структурою. Після того як основні об’єкти розміщені, алгоритм додає допоміжні елементи геймплею: тригери подій, точки спавну ворогів, чекпоінти, освітлення чи декоративні елементи. Ці об’єкти не лише надають візуальної виразності, а й створюють ігровий ритм, сприяючи поступовому наростанню напруги чи складності в міру проходження карти. Оскільки BSP забезпечує ієрархічну структуру, можна передбачити закономірність розподілу викликів і нагород — наприклад, що після кожного двох-трьох підрівнів зростає частота ворогів або цінність предметів.

Завершальним етапом роботи алгоритму є оцінка якості розміщення. Для цього застосовуються вбудовані інструменти валідації, які перевіряють правильність розташування об’єктів, дотримання допустимих відстаней між ними, наявність вільного простору для руху та відсутність критичних колізій. Якщо порушення виявлено, BSP-система може автоматично перезапустити генерацію для конкретних вузлів або виконати локальне коригування об’єктів без повного оновлення рівня. Такий підхід підвищує стабільність роботи системи і забезпечує логічну послідовність у побудові процедурних карт.

Таким чином, алгоритм розміщення ігрових об’єктів на основі BSP-структури поєднує аналітичну точність геометричної генерації з динамічною гнучкістю рушія Unreal Engine 5. Він перетворює абстрактне математичне представлення дерева BSP у насичене ігрове середовище, здатне адаптуватися до змінних параметрів генерації, вимог дизайну та поведінки гравця.

2.4.3 Оптимізація, повторне використання та кешування результатів

Оптимізація та кешування результатів процедурної генерації є невід’ємною складовою ефективного функціонування BSP-системи в Unreal Engine 5. Оскільки процес розбиття простору, побудови геометрії та розміщення об’єктів може бути обчислювально затратним, особливо при великих рівнях або складних параметрах поділу, доцільним є використання механізмів повторного використання проміжних результатів. Це дозволяє суттєво скоротити час обробки, знизити

навантаження на процесор і пам’ять, а також підвищити стабільність під час інтерактивного редагування сцен.

Суть оптимізації полягає в тому, що результати окремих етапів BSP-генерації — розподіл простору, обчислення зв’язків, формування вузлів, побудова геометрії — зберігаються у тимчасових або постійних структурах даних, доступних для подальшого використання. У випадку повторного запуску генерації рушій не виконує повний цикл розрахунків, а лише оновлює ті частини дерева, які зазнали змін. Такий підхід отримав назву інкрементальної генерації. Він особливо ефективний у середовищі Unreal Engine, де розробник може експериментувати з параметрами в режимі реального часу, змінюючи ймовірність поділу, мінімальний розмір кімнат або кількість підрівнів без необхідності повного перерахунку карти.

Кешування проміжних даних реалізується на кількох рівнях. Перший рівень охоплює структурний кеш BSP-дерева, у якому зберігаються вузли з їхніми геометричними параметрами, ідентифікаторами та типами. Це дає змогу миттєво звертатися до вже оброблених підпросторів без повторної генерації. Другий рівень передбачає кешування геометрії та матеріалів — Unreal Engine може зберігати попередньо згенеровані полігони, колізії та текстурні карти у вигляді тимчасових мешів (static або instanced). При повторному запуску системи рушій просто оновлює позиції або параметри візуалізації, не перебудовуючи сітку з нуля. Нарешті, третій рівень — кешування логічних зв’язків і об’єктів, коли інформація про взаєморозташування кімнат, дверей і коридорів зберігається в таблицях з’єднань, що дозволяє швидко відновити маршрути або топологію при зміні структури дерева.

Важливу роль відіграє також оптимізація використання пам’яті та потоків. При обробці великих BSP-структур застосовується багатопотоковий підхід: генерація виконується паралельно для незалежних піддерев, а результати об’єднуються після завершення обчислень. Це дозволяє максимально використовувати ресурси сучасних процесорів і забезпечує стабільну частоту кадрів навіть у режимі реального часу. Для зниження витрат пам’яті BSP-модуль

використовує компактні структури зберігання, де замість повних об’єктів зберігаються лише ключові параметри — координати, розміри, типи кімнат і посилання на суміжні вузли. Додатково застосовується стискання проміжних даних за допомогою методів серіалізації, сумісних із системою Unreal Data Assets.

Ще одним напрямом оптимізації є використання GPU-акселерації для обчислення повторюваних операцій. У новіших версіях Unreal Engine (починаючи з 5.5) частину завдань процедурної генерації — наприклад, розрахунок розподілу точок у просторі або оцінку зіткнень — можна виконувати на графічному процесорі. Це забезпечує значне прискорення при генерації великих рівнів та дозволяє реалізувати попередній перегляд у реальному часі без втрат продуктивності. Об’єднання CPU і GPU генерації створює гібридну архітектуру, яка дозволяє балансувати навантаження між різними підсистемами рушія.

Кешування результатів має також важливе значення для збереження ігрових сесій та відтворюваності результатів. Усі основні параметри генерації, початкові умови, використані випадкові числа й структурні дані зберігаються у вигляді seed-конфігурацій. Це дозволяє при потребі повторити точно такий самий результат генерації, що критично для тестування, балансування або відтворення конкретних рівнів. Такий підхід гарантує стабільність і контрольованість генерації, поєднуючи процедурну варіативність із відтворюваністю.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ГЕНЕРАЦІЇ РІВНІВ

3.1 Архітектура програмного комплексу

У основі системи лежить модульна архітектура, де кожен компонент виконує строго визначену роль у процесі побудови рівня. На верхньому рівні реалізовано керуючий модуль генерації, який координує взаємодію між усіма підсистемами — BSP-деревом, геометрією, топологією, аналітикою та інтеграцією з рушієм. Кожен із модулів реалізований у вигляді окремих класів C++ та Blueprint-компонентів, що взаємодіють через обмежений набір інтерфейсів і структур даних.

Ядро BSP виступає основним обчислювальним центром. Воно реалізує процедуру рекурсивного поділу простору на підобласті, формує структуру дерева, визначає потенційні кімнати, створює граф зв’язків між ними та готує дані для візуалізації. Рівень інтеграції відповідає за передавання цих даних до рушія Unreal Engine 5, де вони перетворюються у реальні об’єкти сцени (Actors, Meshes, Collision Volumes, Lights). Сервісна інфраструктура охоплює логування, кешування, збір статистики, а також систему валідації, що забезпечує стабільність і перевірку коректності створених рівнів.

Ядро системи — це набір класів, що відповідають за математичне формування структури рівня. Основним класом є BSPGenerator, який реалізує алгоритм двійкового поділу простору. Він ініціалізує параметри генерації (розміри карти, мінімальні та максимальні розміри кімнат, ймовірність поділу, seed-значення) і керує процесом рекурсивного створення вузлів дерева. Кожен вузол (BSPNode) містить геометричні межі своєї області, посилання на дочірні вузли, орієнтацію поділу та рівень вкладеності.

Окремий клас RoomDescriptor описує кінцеві області дерева — кімнати. Він зберігає параметри кожного примітиву: центр, площу, тип (звичайна, ключова, стартова, кінцева), а також метадані, пов’язані з ігровими сценаріями (рівень складності, розподіл ресурсів, щільність об’єктів). На цьому етапі формується

ієрархічна структура рівня, що відображає логіку розподілу простору відповідно до параметрів BSP.

Для з’єднання кімнат у єдиний маршрут використовується клас ConnectionManager, який будує топологічний граф. Він визначає, які кімнати мають бути поєднані коридорами, перевіряє їх доступність і уникає дублювання шляхів. Завдяки цьому рівень має не лише структурну, а й функціональну цілісність — усі зони зв’язані між собою, але при цьому зберігається контрольована складність маршруту.

Важливою частиною ядра є також ParameterManager, що відповідає за динамічне налаштування параметрів генерації. Він дозволяє зберігати профілі генерації, завантажувати різні конфігурації (наприклад, "Dungeon", "Ruins", "Facility") та регулювати параметри без перекомпіляції рушія.

Цей рівень виконує роль мосту між алгоритмічною логікою та рушієм Unreal Engine 5. Основним завданням є перетворення структур BSP у реальні об’єкти сцени, що можуть бути візуалізовані, використані для навігації чи взаємодії з гравцем.

Клас LevelBuilder отримує від BSP-модуля набір дескрипторів кімнат і коридорів та перетворює їх на об’єкти типу AActor з прив’язаними компонентами UStaticMeshComponent. Він відповідає за генерацію геометрії кімнат, створення меж колізії, розміщення світла та тригерів подій. Для великих карт реалізована підтримка потокового завантаження (Level Streaming), що дозволяє підвантажувати лише необхідні ділянки карти під час гри.

Інший важливий елемент — VisualizerModule, який забезпечує попередній перегляд BSP-структури безпосередньо в редакторі. Він виводить каркас дерева, межі кімнат, напрямки коридорів і кольорове кодування за рівнем складності. Це дозволяє розробнику або досліднику бачити логіку генерації в дії й швидко оцінювати якість побудованої карти.

Для передачі даних між рівнями використовується набір структур (FRoomData, FConnectionData), що зберігають лише необхідні числові параметри,

уникаючи надлишкової інформації. Це підвищує ефективність обміну та спрощує серіалізацію при збереженні або відновленні результатів генерації.

Сервісна інфраструктура виконує роль підсистеми підтримки життєвого циклу генерації. Вона містить засоби кешування, логування, аналітики, тестування та оцінки якості отриманих рівнів. Клас CacheSystem зберігає результати попередніх генерацій, щоб при повторних запусках використовувати інкрементальний підхід — перераховуються лише змінені ділянки дерева. Такий підхід істотно скорочує час побудови та підвищує стабільність при редагуванні параметрів у реальному часі.

Клас MetricsAnalyzer виконує збір і обробку статистичних даних — площа кімнат, середня довжина маршрутів, кількість з’єднань, ступінь симетрії, рівень ігрової варіативності. Отримані результати дозволяють оцінити збалансованість рівня й використовуються у системі автоматизованої валідації.

Додатково реалізована підсистема Logger, що фіксує етапи побудови, помилки, попередження та показники продуктивності. Це дозволяє не лише налагоджувати генератор, а й проводити дослідницький аналіз для підвищення ефективності алгоритмів у майбутніх ітераціях.

3.1.1 Структура та ієрархія програмного ядра

Головним фасадом є клас керування процесом генерації, який ініціює побудову дерева BSP, збирає параметри (насіння, мін/макс розміри, глибина, ймовірність поділу, допустима асиметрія), запускає етапи «поділ → витяг листів → формування кімнат → з’єднання коридорами → валідація». Він відповідає за життєвий цикл генерації: від «холодного старту» (повна побудова) до інкрементальних оновлень окремих піддерев при зміні конфігурації.

Вузол BSP-дерева базова сутність, що представляє підобласть простору з геометричними межами, посиланнями на лівий/правий підвузол, глибиною в дереві, орієнтацією розрізу та метаданими (вага складності, роль у маршрутах). Листи інтерпретуються як кандидати у кімнати; внутрішні вузли зберігають «історію» поділу, потрібну для коректного з’єднання кімнат.

Окремі легковагові моделі з нормалізованими даними: габарити, тип кімнати (звичайна/ключова/службова/бос-зона), центроїд, ширина/висота проходу, очікувана пропускна здатність, мітки складності. Вони відокремлюють «що саме» треба побудувати від «як саме» це буде відтворено у рушії.

Менеджер зв’язків (топологія) відповідає за побудову графа прохідності. На основі структури BSP підбирає пари кімнат-сусідів, формує коридори згідно з орієнтацією останнього розрізу, перевіряє, що граф зв’язний, та регулює кількість альтернативних маршрутів до цільового діапазону складності. Зберігає матрицю досяжності для швидких перевірок.

Менеджер параметрів і профілів забезпечує роботу з наборами конфігурацій (профілі «арена», «підземелля», «офісний поверх» тощо): пороги розмірів, глибина, ймовірності, варіативність позиції розрізу, цілі за кількістю кімнат/довжиною шляху, діапазони ширини коридорів. Підтримує seed-конфігурації для відтворюваності та зберігає історію ітерацій.

Модуль автоматично оцінює прохідність (reachability), щільність шляхів, середню довжину маршруту «вхід→вихід», градієнт складності від входу до глибин дерева, перевіряє ізольовані області та надмірну симетрію. На підставі метрик може запускати локальну регенерацію проблемних сегментів.

Підсистема кешування/інкрементів зберігає проміжні результати: фрагменти BSP-піддерев, дескриптори кімнат/коридорів, вже розмічені точки «дверей», попередні рішення валідації. Дає змогу оновлювати лише змінені підобласті за рахунок стабільних ідентифікаторів вузлів і контрольних сум параметрів.

Адаптер інтеграції з UE безпосередньо не будує геометрію — натомість транслює дескриптори у зрозумілі рушію структури даних, ініціює створення акторів/компонентів, викликає побудову навігації, оновлення освітлення і тригерів ігрової логіки. Служить «тонким шаром» між чистою логікою і конкретикою рушія.

3.1.2 Розмежування логіки

Обрано комбінований підхід. Обчислювально важкі, ітеративні частини — поділ BSP, добір зв’язків, валідація графа, агрегування метрик складності, кешування — реалізуються на стороні C++ ядра. Це гарантує продуктивність і контроль за пам’яттю на великих картах та під час численних перегенерацій.

Дизайнерський шар, налаштування і інтерактивність — у Blueprints/PCG Graph. Через візуальні вузли надаються регульовані параметри профілю (мін/макс розміри, ймовірності, асиметрія, цільові метрики), перемикачі режимів (greybox/production), фільтри підстановки контенту (набір мешів, матеріалів, освітлення), а також відображення дебаг-даних (контури кімнат, напрямки коридорів, «теплові» карти складності). Такий розподіл дозволяє ітеративно змінювати дизайн без перекомпіляції ядра та в той же час тримати «важку математику» поза графом.

На етапі побудови сцени C++ ядро формує набір дескрипторів кімнат/коридорів, які передаються у візуальний шар. Blueprint-актори (кімнати, сегменти стін, двері, освітлювачі, навігаційні волюми) заповнюються даними, обирають конкретні меші/матеріали за правилами теми й ініціюють побудову або оновлення навігаційної сітки. Для великих світів підтримується поетапне вивантаження/завантаження сегментів (streaming) — ядро віддає партіями дескриптори підобластей, а візуальний шар підхоплює їх, не блокуючи редактор.

Під час інтерактивної роботи дизайнер змінює значення параметрів у UI-панелі профілю; це породжує подію оновлення, яку ядро використовує для інкрементальної регенерації лише тих піддерев, де порушилась інваріантність (наприклад, зменшено W_min/H_min або підвищено P_split). Дані повертаються у Blueprints і приводять до часткової перебудови сцени. Таким чином досягається цикл «живої» генерації: правка — локальна регенерація — візуальне підтвердження — вимірювання метрик.

3.2 Реалізація BSP-алгоритму в Unreal Engine

Реалізація алгоритму двійкового розбиття простору (Binary Space Partitioning, BSP) у межах проєкту базується на концепції поетапного

рекурсивного поділу ігрової області на підпростори з подальшим формуванням із них повноцінних кімнат і коридорів. Основною метою є створення процедурного рівня, який зберігає логічну структуру, ігровий баланс та просторову варіативність, забезпечуючи при цьому стабільну продуктивність і можливість повторного використання алгоритму в інших проєктах Unreal Engine.

На рівні архітектури система BSP реалізована як окремий модуль ядра генерації. Вона має власний життєвий цикл, який починається з ініціалізації параметрів генерації — розмірів області, мінімальних і максимальних розмірів кімнат, ймовірності поділу та насіння випадкових чисел (seed). Ці дані зберігаються в конфігураційних структурах, що дозволяє легко змінювати параметри без перекомпіляції рушія. Алгоритм виконується як послідовність рекурсивних операцій: кожна ітерація перевіряє, чи задовольняє поточна область умови для поділу, і якщо так — виконує розсічення уздовж вибраної осі (X або Y) у випадковій або параметрично визначеній точці.

Результатом цього етапу є побудова дерева BSP — ієрархічної структури, де внутрішні вузли відповідають операціям поділу, а листові вузли — кінцевим підобластям, які інтерпретуються як кімнати. Для кожного вузла фіксуються його координати у просторі рівня, орієнтація розділення, відсоткове співвідношення між отриманими підобластями та глибина рекурсії. Зберігання цих даних у спеціальних структурах дозволяє легко візуалізувати дерево або відновити його в процесі налагодження. На відміну від класичних реалізацій BSP у комп’ютерній графіці, що використовуються для рендерингу або трасування променів, у нашій системі структура дерева має суто генеративну функцію — вона визначає топологічну основу майбутнього рівня, на яку надалі накладаються кімнати, проходи, об’єкти ігрової логіки та декоративні елементи [59; 85].

Після завершення побудови дерева виконується процес формування кімнат. Для кожного листового вузла обчислюються внутрішні межі з урахуванням відступів і мінімальної ширини стін. Розміри кімнати можуть коригуватися відповідно до параметрів складності, заданих у профілі генерації. Для створення природної нерівномірності використовується коефіцієнт варіації, який додає

невелике випадкове зміщення до меж кімнат. Це дозволяє уникнути надмірної симетрії та створює більш реалістичний вигляд рівня.

Далі система переходить до етапу з’єднання кімнат у єдину топологію. Для цього аналізуються пари кімнат, які мають спільного батьківського вузла у BSP-дереві. На підставі орієнтації розсічення між ними будується коридор, який проходить через межу поділу. Положення коридору визначається серединою спільної стіни або обчислюється випадково в межах допустимого діапазону. Алгоритм перевіряє, щоб не утворювалися ізольовані області, і за потреби створює додаткові зв’язки між кімнатами для забезпечення прохідності. Таким чином, побудова коридорів завершує процес топологічного з’єднання структури, утворюючи цілісну карту рівня.

Важливою особливістю реалізації є адаптація алгоритму до роботи в реальному часі в межах Unreal Engine. Генерація може виконуватись як у редакторі, так і під час гри, тому BSP-модуль спроєктовано з урахуванням інкрементальності. Якщо користувач змінює окремі параметри генерації (наприклад, ймовірність поділу або мінімальний розмір кімнати), система не перебудовує всю структуру дерева, а оновлює лише ті підобласті, які більше не відповідають новим умовам. Це забезпечується кешуванням вузлів і збереженням стану генерації між циклами запуску [88; 91].

Після створення BSP-структури дані передаються до інтеграційного рівня для побудови геометрії. Кожна кімната перетворюється на об’єкт сцени типу AActor із компонентами UStaticMeshComponent або ProceduralMeshComponent, які створюють геометрію стін, підлоги та стелі. Система автоматично додає базові матеріали, генерує колізії та навігаційні області для взаємодії з персонажем. Для тестування реалізовано режим попереднього перегляду (greybox mode), у якому кімнати візуалізуються у вигляді простих примітивів, що дозволяє швидко оцінити логіку структури без навантаження на GPU.

На завершальному етапі реалізовано механізм оцінки якості згенерованої карти. BSP-модуль обчислює базові метрики — кількість кімнат, середню довжину шляху між входом і виходом, ступінь симетрії, частку прохідних зон і

середню площу. Якщо виявлено порушення (наприклад, замкнені області або надлишкові розгалуження), система автоматично позначає проблемні вузли для повторної генерації. Завдяки цьому процес має не лише випадковий, але й контрольований характер, що особливо важливо для ігор, у яких баланс між варіативністю та ігровою логікою повинен бути збережений.

3.2.1 Структура даних BSP-дерева та рекурсивна логіка генерації

Структура BSP-дерева у нашій системі розглядається як ієрархічна модель простору, де кожен внутрішній вузол відповідає операції поділу прямокутної області за однією з осей, а кожен лист — кінцевій підобласті, потенційній кімнаті або службовому сегменту рівня. Вузол зберігає мінімальний і максимальний кути осьово-вирівняного прямокутника, ідентифікатор, глибину у дереві, орієнтацію останнього розрізу, позицію площини поділу в локальних координатах та метадані, корисні на етапі побудови топології (вага складності, ролі «вхід/вихід», пріоритет маршрутизації). Таке представлення свідомо лишається геометрично «тонким»: воно не містить мешів чи матеріалів, а виступає чистою логікою просторового розбиття, яку згодом інтерпретує інтеграційний рівень рушія UE5 [59; 85].

Рекурсивна реалізація поділу стартує з кореня, що охоплює всю цільову область рівня. На кожному кроці обирається орієнтація розсічення з урахуванням співвідношення сторін поточної підобласті та цільової різноманітності планувань: витягнуті області мають вищий пріоритет для поділу перпендикулярно довшій стороні, що знижує ризик появи надто «тонких» кімнат; для майже квадратних областей вибір осі може бути стохастичним із контрольованим перекосом, аби уникати монотонних послідовностей горизонтальних або вертикальних відсічень. Позиція площини поділу визначається як функція від центральної лінії з доданим випадковим зсувом у дозволеному діапазоні; при цьому на обидві отримані області накладаються інваріанти мінімальних розмірів W_{min}, H_{min} та, за потреби, «м’які» обмеження на максимальні габарити. Якщо хоча б одна зі сторін потенційного нащадка порушує інваріант, площина зсувається або поділ

відхиляється на користь іншої орієнтації. Таким чином виконується контрольоване стохастичне розбиття, сумісне зі сценаріями ігрового балансу.

Зупинка рекурсії визначається комбінацією умов: досягненням граничної глибини дерева, неможливістю сформувати валідні нащадки за розмірами, вичерпанням бюджету кімнат чи падінням імовірності поділу нижче адаптивного порога. Зменшення ймовірності з глибиною створює природну ієрархію: великі камери на верхніх рівнях і дрібніші кімнати внизу, що добре корелює з подальшим розподілом складності та маршрутизацією від входу до виходу. Для забезпечення відтворюваності кожне рішення про вісь і позицію поділу робиться на основі детермінованого генератора псевдовипадкових чисел зі seed-конфігурації; це дозволяє точно повторити побудову рівня і виконувати регресійні перевірки якості.

Спеціальна увага приділяється стабільності структури під час інтерактивних змін параметрів у редакторі. Щоб уникати «ефекту доміно», коли незначна правка спричиняє повну перебудову дерева, вузли мають стабільні ідентифікатори, а рекурсія виконується з префіксним порядком і локальним кешем рішень поділу. Якщо змінюється, скажімо, мінімальна ширина кімнати, інвалідовуються лише ті піддерева, де порушено інваріант; решта гілок зберігає попередні розсічення. Це забезпечує інкрементальність, помітно зменшує час відгуку й дозволяє проектувальнику отримувати миттєвий зворотний зв’язок у режимі «greybox».

Топологічна узгодженість формується «по дорозі»: кожен внутрішній вузол зберігає параметри спільної межі між лівим і правим нащадками, які пізніше використовуються для прокладення коридору крізь площину поділу. Така прив’язка на рівні дерева позбавляє потреби в дорогому глобальному пошуку сусідств уже після завершення рекурсії та гарантує, що граф з’єднань має природну відповідність історії розрізів. Для уникнення вироджених випадків застосовуються локальні поправки: якщо серія поділів створила надто складну «пилообразну» межу або ланцюг дрібних камер, дерево може виконати операції злиття сусідніх листів за правилами мінімальної площі та оцінки корисності, після чого відновити спільну межу як єдиний канал проходу.

З погляду ресурсів, рекурсивний обхід реалізовано з урахуванням тайм-слотів редактора та ігрового тіку: генерація великих дерев може виконуватись пакетами, коли крок поділу обмежується бюджетом мілісекунд на кадр, а незавершений стан серіалізується між оновленнями. Це уможливлює побудову об’ємних рівнів без помітних фризів інтерфейсу. Пам’ять економиться за рахунок легковагових вузлів (осьово-вирівняні прямокутники, короткі метадані, посилання на дітей) і винесення важчих артефактів — геометрії, навігації, матеріалів — за межі дерева, на етап інтеграції з рушієм. Далі, коли структура стабілізована, вузли-листи транслюються у дескриптори кімнат, що споживаються модулем побудови сцени, а внутрішні вузли — у «шви» для коридорів, що дає пряму відповідність «логіка → геометрія» без дублюючих розрахунків.

Нарешті, у контексті контролю якості, сама структура BSP-дерева слугує носієм метрик: глибина листа корелює з очікуваною складністю зони; баланс лівих/правих піддерев — із симетрією плану; розподіл площ листів — із різноманіттям кімнат. Рекурсивна реалізація із детермінованою стохастикою дозволяє тонко керувати цими розподілами, поєднуючи жорсткі інваріанти прохідності й розмірів із потрібною варіативністю для реіграбельності

3.2.2 Побудова кімнат, коридорів і переходів

У проєкті логіка побудови кімнат, дверей і коридорів організована як послідовність трьох пов’язаних фаз, що спираються на готову BSP-структуру: інтерпретація листів дерева як кімнат, синтез дверних прорізів на «швах» внутрішніх вузлів і трасування коридорів крізь відповідні площини поділу. Код цих фаз рознесено по спеціалізованих класах ядра й тонкому шарі інтеграції, але в межах розділу ми описуємо лише те, що саме робитиме код, без лістингів.

По-перше, для кожного листового вузла (майбутньої кімнати) формується дескриптор кімнати: це компактна структура з межами AABB, центроїдом, категорією (звичайна/ключова/старт/фінал), «полями відступів» під стіни та службовими мітками (вага складності, пріоритет наповнення). Коли менеджер сцени отримує такий дескриптор, він створює актор кімнати у світі UE. Усередині

актора додаються компоненти підлоги/стін/стелі (як UStaticMeshComponent або ProceduralMeshComponent у режимі greybox), задаються колізії та навігаційні об’єми. Щоб уникнути «ідеальних прямокутників», на рівні даних підтримується невелика параметрична варіація габаритів (offset/rounding), яка контролюється профілем генерації — код просто читає цю варіацію й застосовує до вершин геометрії перед побудовою. Для великих сцен менеджер створення акторів працює батчами: кімнати породжуються партіями в кілька кадрів, а проміжний стан зберігається у кеші — це саме те, що робитиме код, аби не блокувати редактор і гру.

По-друге, двері прив’язуються не до кімнат окремо, а до внутрішніх вузлів BSP, які зберігають «спільну межу» між парою дочірніх підобластей. На цьому етапі код бере напрям розсічення (горизонтальний/вертикальний), обчислює допустимий відрізок перетину з реальними стінами обох суміжних кімнат і вибирає точку/смугу встановлення дверей згідно з правилами: мінімальна відстань від кутів, кратність ширині проходу, узгодження з шириною коридору. Якщо вибрана позиція конфліктує з опорами або стиками матеріалів, алгоритм зсуває двері по межі в межах дозволеного діапазону. У фіналі менеджер сцени спавнить актор дверей із потрібним сніппетом геометрії та колізійним «порталом»; паралельно оновлюється граф прохідності: між відповідними кімнатами встановлюється ребро з пропускною здатністю, рівною ширині дверей. Важливо, що саме внутрішній вузол тримає «право» на двері: якщо параметри змінюються і відбувається локальна регенерація піддерева, код видаляє/пересоздає двері лише для зачеплених пар, не торкаючись решти рівня.

По-третє, коридори трасуються по «шву» внутрішнього вузла, який породив двері. На рівні даних код знає: орієнтацію площини поділу, координати дверного прорізу та габарити двох кімнат; завдання — побудувати коридор із гарантованою шириною, плавними примиканнями й валідними колізіями. У найпростішому випадку це прямолінійний сегмент перпендикулярно площині поділу; для уникнення «зубців» передбачено режим згладженого стику: коридор може мати короткі фаски або невеликі підрізи в місцях примикання до кімнат. Код створює

актор коридору, підкладає меш із параметризованою шириною/висотою, виставляє NavModifierVolume так, щоб навігаційна мережа оновилась одним проходом. Для довгих коридорів менеджер може автоматично вставляти сегменти освітлення й маркери декору згідно з «темою» профілю; це робиться у візуальному шарі (Blueprint/PCG Graph) на підставі числових атрибутів, які передає ядро [88; 91].

Окремо вбудовано забезпечення прохідності. Після первинної розстановки дверей/коридорів код виконує перевірку зв'язності: за графом кімнат обчислюється матриця досяжності; якщо знаходяться ізольовані вузли, менеджер зв'язків просить ядро запропонувати одну з трьох дій — змістити двері, додати допоміжний коридор у межах того ж предка або, якщо це «кишенька» без сенсу, дозволити її як секретну нішу. Рішення приймається за правилами профілю (допустима частка тупиків, рівень складності, цільова довжина основного маршруту). В алгоритмі навмисно закладено локальний характер виправлень: оновлюються лише відповідні актори й навігація в межах їхніх об'ємів, без повного перебудування сцени.

Для керованої різноманітності застосовується планувальник стилю: у дескрипторі кімнати зберігається набір тегів («велика/мала», «ключова», «центральна»), а у дверей і коридорів — атрибути «ширина», «важливість», «напрямок». Візуальний шар читає ці атрибути й підбирає відповідні меші/матеріали/освітлення, не змінюючи геометричної логіки. Завдяки цьому, змінюючи лише профіль теми, можна отримувати інший візуальний результат без змін у коді генерації. Усі створені актори маркуються стабільними ідентифікаторами, пов'язаними з ідентифікаторами BSP-вузлів: саме так код зможе надалі виконувати інкрементальні оновлення — видаляти, пересоздавати або переміщати кімнати, двері, коридори строго там, де змінився вихідний «шов».

Після побудови фізичної геометрії запускається фаза пост-обробки: виконується перебудова NavMesh у зонах, що змінилися; застосовується lightmass-попередній розрахунок або Lumen-оновлення для динамічних сцен; оновлюються тригери геймплею, пов'язані з проходами (відчинення/зачинення

дверей, контроль потоків NPC, чекпоінти). Код цієї фази не генерує нову геометрію — він лише синхронізує системи рушія з уже збудованою топологією. Якщо профіль вимагає цільову довжину маршруту «вхід→вихід», модуль аналітики вимірює її на графі і, за потреби, просить менеджер зв’язків тонко відрегулювати позицію одного-двох дверей або додати/прибрати неосновне ребро, зберігаючи глобальну структуру.

3.3 Підсистема керування параметрами та аналітики

Підсистема контролю параметрів генерації — це ключовий компонент архітектури процедурного генератора, що забезпечує централізоване керування всіма змінними, від яких залежить логіка роботи BSP-алгоритму та побудова рівня в цілому. Її основна функція — уніфікувати конфігураційні параметри, забезпечити їх збереження, динамічну зміну під час роботи рушія Unreal Engine і можливість гнучкого налаштування через інтерфейс користувача або скриптові системи.

У нашому проєкті підсистема реалізована у вигляді модульного компоненту UBSPGenerationSettings, який належить до ядра генерації, та інтегрованого контролера параметрів — ABSPParameterManager. Ці класи тісно взаємодіють між собою: перший відповідає за зберігання і серіалізацію конфігурацій, другий — за їх динамічне оновлення, валідацію та поширення змін на інші модулі (BSP Generator, Visualizer, Analytics).

Клас UBSPGenerationSettings є основною структурою конфігурації. Він зберігає набір змінних, що описують:

— мінімальні та максимальні розміри кімнат (RoomMinSize, RoomMaxSize);

— параметри ймовірності поділу (SplitChance, SplitDecayRate);

— цільову кількість кімнат (TargetRoomCount);

— глибину рекурсії дерева (MaxDepth);

— коефіцієнти складності (DifficultyScale);

— прапорці увімкнення випадковості (bRandomSeed, bSymmetricSplit);

— ідентифікатор seed-значення для детермінованих генерацій (Seed).

Кожна з цих змінних має спеціальні метадані: діапазон значень, тип контрольного елемента (слайдер, чекбокс, текстове поле) та сигнал оновлення, що викликає подію OnParameterChanged. Це дозволяє зв’язати зміну будь-якого параметра із негайним оновленням BSP-структури або її частини.

Під час запуску гри чи редактора клас ініціалізується зі збереженого профілю. Профіль представляє собою JSON або INI-файл, який система може автоматично зчитати при старті — цим займається окремий менеджер збережень UBSPProfileHandler. Завдяки цьому користувач може створювати і перемикати кілька наборів налаштувань (“Small Dungeon”, “Large Fortress”, “Urban Grid”), не змінюючи сам код.

Клас ABSPParameterManager виступає як мостовий актор, який існує у сцені Unreal Engine і пов’язаний із візуальним інтерфейсом. Його завдання — забезпечити зв’язок між користувачем (дизайнером) та внутрішньою логікою генерації. Коли дизайнер змінює параметр у віджеті UI або Blueprint Graph, менеджер оновлює екземпляр UBSPGenerationSettings, викликає його подію OnSettingsUpdated і передає зміни у генератор UBSPGenerator. У свою чергу генератор вирішує, які частини BSP-дерева необхідно оновити — повністю чи інкрементально.

Система контролю побудована за принципом реактивного оновлення: будь-яка зміна параметрів миттєво поширюється на пов’язані модулі без потреби ручного перезапуску генерації. Це реалізовано через систему делегатів Unreal Engine (FMulticastDelegate). Наприклад, при зміні RoomMinSize у Blueprint панелі, менеджер надсилає сигнал генератору, який викликає локальну перевірку листів BSP і перебудовує лише ті, де умова мінімального розміру порушена. Такий підхід дозволяє зберігати стабільність сцени та забезпечує майже миттєвий зворотний зв’язок.

Для зручності тестування і навчального використання система має режим симуляції параметрів. Менеджер може змінювати значення автоматично за певним сценарієм — наприклад, поступово збільшувати складність або варіювати

глибину дерева, фіксуючи результат. Це дозволяє досліджувати поведінку BSP-алгоритму без ручних втручань і збирати статистику для розділу експериментальної частини.

Окрему роль відіграє система обмежень і валідації. Перш ніж прийняти нове значення параметра, ABSPParameterManager виконує перевірку: чи не суперечить воно критичним межам (наприклад, щоб мінімальна ширина не перевищувала максимальну, або щоб ймовірність поділу не падала до нуля при глибині менше 1). Якщо параметр невалідний, система відхиляє зміну та повідомляє користувача через спливаюче повідомлення в редакторі або в лог консолі.

Для дизайнерів передбачено спеціальний Blueprint Widget — WBSPSettingsPanel, який відображає усі параметри генерації у вигляді інтуїтивно зрозумілих елементів: слайдерів, випадаючих списків і перемикачів. Кожен із них двосторонньо зв’язаний із ABSPParameterManager, тому зміни відразу синхронізуються між візуальною панеллю та реальними даними рушія. Також у панелі є кнопка “Regenerate Level”, що викликає повну перегенерацію сцени для перевірки нових конфігурацій.

Під час виконання гри параметри можуть змінюватися й динамічно, наприклад, адаптуючись під рівень складності або поведінку гравця. Для цього у UBSPGenerationSettings передбачено метод ApplyRuntimeModifiers(), який зчитує дані з аналітичного модуля (PlayerPerformanceTracker) і змінює деякі параметри “на льоту” — наприклад, збільшує щільність коридорів або зменшує кількість тупиків, якщо гравець часто губиться.

Щоб підвищити повторюваність експериментів, у систему включено менеджер профілів генерації — UBSPProfileHandler. Він відповідає за збереження і завантаження конфігурацій користувача. Кожен профіль містить не лише параметри генерації, а й метадані — назву, дату, тип рівня, seed-значення, а також короткий опис. Усі профілі зберігаються у форматі JSON, що дає можливість швидко експортувати їх у зовнішні системи (наприклад, для академічних тестів або відтворення результатів).

При завантаженні нового профілю ABSPParameterManager виконує повну синхронізацію: оновлює UI, передає параметри в UBSPGenerator і викликає перегенерацію рівня. У редакторі Unreal Engine цей процес виглядає як плавне оновлення сцени з анімацією оновлення кімнат — без повного перезапуску.

3.3.1 Інтерфейси налаштування параметрів генерації

У підсистемі контролю параметрів генерації ключові налаштування — Seed, Depth, MinSize та SplitRatio — визначають характер розбиття простору, форму і масштаб кімнат, щільність з'єднань і відтворюваність результатів. На рівні проєктної логіки ці параметри зберігаються в конфігураційному об'єкті UBSPGenerationSettings і редагуються через актор-контролер ABSPParameterManager, який слухає зміни інтерфейсу (Blueprint/UI) та розсилає сигнал OnSettingsUpdated у ядро UBSPGenerator. Сам генератор сприймає це як транзакцію: він оцінює, які піддерева BSP стали невалідними, інвалідовує лише їх і виконує інкрементальну регенерацію, зберігаючи решту структури та пов'язані актори сцени.

Параметр Seed забезпечує детерміновану псевдовипадковість. Усі стохастичні рішення — вибір осі поділу, позиції площини, зсув дверей, варіація габаритів кімнат — базуються на генераторі випадкових чисел, ініціалізованому з Settings.Seed. Зміна Seed призводить до нового «макета», але при незмінних значеннях інваріантів (MinSize, SplitRatio, Depth) гарантується відтворюваність: той самий Seed створює ту саму топологію. На рівні даних UBSPProfileHandler зберігає Seed у профілі разом із метаданими (назва теми, дата, опис), що важливо для експериментів і регресійних перевірок якості [59; 92].

Параметр Depth контролює граничну глибину рекурсії й тим самим впливає на «зернистість» планування. Менша глибина дає великі камери і короткі маршрути, більша — дрібні кімнати, більше коридорів і розгалужень. На практиці MaxDepth працює у зв'язці з TargetRoomCount та адаптивним згасанням імовірності поділу: якщо кімнат уже достатньо, генератор знижує шанс подальших розрізів навіть до досягнення формальної межі глибини. У реалізації

це відбувається через локальний лічильник у вузлі BSP і таблицю розподілу «глибина → штраф імовірності»; таким чином ми уникаємо надмірної дрібної сегментації, зберігаючи контрольовану варіативність.

Параметр MinSize задає інваріанти мінімальної ширини та висоти кімнати. Перед кожним розсіченням UBSPGenerator оцінює, чи дадуть кандидати-нащадки прямокутники з обома сторонами не меншими за RoomMinSize. Якщо хоча б один нащадок порушує обмеження, площина поділу зсувається в межах дозволеного діапазону; якщо валідний поділ неможливий — вузол фіксується як лист. Саме ці перевірки гарантують прохідність і запобігають появі «тонких» камер і коридорів, що ускладнюють навігацію. При зміні MinSize у сцені менеджер параметрів спричиняє локальну перебудову лише тих листів, де інваріант став порушеним; завдяки кешу вузлів сцена оновлюється без повного перерахунку.

Параметр SplitRatio визначає відсоткове співвідношення розмірів між лівою і правою (або верхньою і нижньою) підобластями при поділі. У налаштуваннях він представлений як діапазон допустимих часток (наприклад, 0.4–0.6 для «збалансованих» планів, 0.3–0.7 для більш асиметричних). На етапі поділу UBSPGenerator обирає конкретну точку в межах цього коридору, додаючи керовану стохастику. Вужчий коридор SplitRatio створює ритмічні, «квадратніші» кімнати, ширший — підвищує різноманіття форм і довжину коридорів. Код зберігає обраний фактичний коефіцієнт у вузлі, аби під час інкрементальних правок не «стрибали» вже вдалі пропорції.

Усі чотири параметри підпорядковуються системі валідації ABSPParameterManager. Перед застосуванням змін перевіряються логічні зв’язки: мінімальні розміри не можуть перевищувати максимальні, SplitRatio не виходить за глобальні межі (наприклад, 0.25–0.75), а Depth обмежено так, щоб теоретично досяжна кількість листів не перевищувала ресурсні бюджети сцени. У разі колізій менеджер блокує зміну, відображаючи повідомлення в UI і логуючи деталі. Така попереджувальна перевірка економить час і запобігає «ефекту доміно» при регенерації.

На стороні інтерфейсу віджет WBSPSettingsPanel експонує Seed, Depth, MinSize, SplitRatio як слайдери та поля вводу. Зміна будь-якого контролу миттєво синхронізується з UBSPGenerationSettings і запускає «м’який» цикл оновлення: ядро визначає зону впливу, інвалідовує відповідні вузли, перебудовує кімнати/коридори та повідомляє візуалізатор ABSPVisualizer про необхідність перерахунку месхів лише в цих ділянках. Додатково у панелі є перемикач «Greybox/Theme»: у режимі greybox зміни параметрів проявляються максимально швидко завдяки легковаговим матеріалам і простим сабмешам, що особливо корисно під час дослідницьких ітерацій.

Для рантайм-адаптації передбачено шар модифікаторів. Аналітичний компонент може запропонувати тимчасові поправки до SplitRatio або MinSize, якщо телеметрія показує надмірну кількість тупиків чи надто довгі маршрути. Метод ApplyRuntimeModifiers() у UBSPGenerationSettings приймає такі поправки як дельти і передає сигнал у UBSPGenerator. Той, у свою чергу, виконує локальне перепланування лише у вибраній зоні — наприклад, у найближчих до гравця піддеревах або в «холодних» сегментах, де зміни не зіпсують поточну сесію.

3.3.2 Збір статистики

Система логування та аналітики у процесі процедурної генерації рівнів є важливим елементом для оцінки якості, стабільності та ефективності роботи BSP-алгоритму. Вона дозволяє розробнику не лише бачити результат генерації у вигляді геометрії, а й отримувати кількісні показники, що відображають структурні властивості згенерованого рівня: кількість кімнат, середню площу, щільність, глибину дерева, частку використаного простору, кількість коридорів та інші метрики. У нашому проєкті ця підсистема реалізована у вигляді спеціального аналітичного класу UBSPAnalytics, який тісно взаємодіє з ядром генератора (UBSPGenerator) і параметричним контролером (ABSPParameterManager).

Клас UBSPAnalytics є окремим сервісним компонентом, підключеним до основного генератора через інтерфейс подій. Він отримує повідомлення про завершення побудови BSP-дерева (OnGenerationCompleted), після чого аналізує

усі вузли дерева і збирає ключові статистичні дані. Для цього модуль має набір внутрішніх структур:

— froomstat — дані про кожну кімнату (ідентифікатор, розміри, площа, координати центру, сусіди);

— fmapstat — агреговані показники рівня (загальна кількість кімнат, середня площа, мін/макс розміри, середня глибина дерева, коефіцієнт заповнення простору).

Усі ці структури створюються динамічно під час пост-обробки після завершення рекурсивного розбиття BSP.

Аналітика працює як асинхронна підсистема, щоб не блокувати головний потік рушія. Після того, як BSP-генератор завершує побудову дерева, він передає посилання на корінь структури у метод UBSPAnalytics::AnalyzeTree(). Далі, у фоновому потоці, модуль рекурсивно обходить усі листи дерева, обчислює площі та формує агрегати. Це дає змогу паралельно оновлювати сцену та аналізувати результати без падіння продуктивності.

Для відображення результатів у режимі редактора використовується клас UBSPLogger, який формує звіт у зручному вигляді. Він записує результати у два потоки:

— output log unreal engine (через UE_LOG(LogBSP, Display, TEXT(...))), де всі показники виводяться у структурованому вигляді;

— json/csv-файл, якщо користувач активував режим детального збереження.

Кожен звіт містить такі поля: час генерації, назву профілю, Seed, Depth, TargetRoomCount, фактичну кількість кімнат, середню площу, щільність, коридорність, глибину дерева, час побудови (мс). Завдяки цьому можна проводити статистичний аналіз і порівнювати результати між різними сесіями генерації.

Наприклад, при кожному натисканні кнопки “Generate” у віджеті WBSPControlPanel, ABSPParameterManager викликає метод UBSPAnalytics::AnalyzeAndLog(). Цей метод не лише обчислює метрики, а й

виконує порівняння з попередніми ітераціями, зберігаючи тренд — чи збільшилася середня площа, чи стабілізувався коефіцієнт щільності.

Для зручності розробника передбачено інтеграцію результатів аналітики у сам Unreal Editor. Віджет WBSPControlPanel має окрему вкладку “Stats”, яка показує числові значення у вигляді таблиці, а також прості графіки (бар-чарти або прогрес-бари). Наприклад, кількість кімнат і відхилення від цільового значення відображаються у вигляді заповнених індикаторів, щільність — у вигляді термометра.

Для глибшого аналізу у Debug Mode вмикається відображення метаданих безпосередньо в сцені: поверх кожної кімнати показується її площа, а колір матеріалу вказує на відхилення від середнього значення. Це дає змогу швидко виявити надто великі або малі кімнати й відрегулювати параметри генерації.

Модуль UBSPAnalytics підтримує експорт результатів у форматах JSON, CSV та XML, що дозволяє інтегрувати його з зовнішніми аналітичними інструментами (Python, Excel, R). Збереження відбувається автоматично після кожного циклу генерації або вручну через кнопку “Export Data”. Для академічних експериментів аналітика може бути зібрана в пакетному режимі: система автоматично генерує десятки карт із різними Seed і зберігає результати в одному архіві для статистичного аналізу.

3.4 Засоби візуалізації та налагодження генерації

Головним виконавчим елементом є ABSPVisualizer, який виступає посередником між генератором (UBSPGenerator) і середовищем візуалізації. Після завершення генерації BSP-дерева модуль отримує структуру вузлів через подію OnGenerationComplete і починає побудову 3D-представлення рівня. Для кожного листового вузла (кімнати) створюється актор типу ABSPRoomMesh, який містить компоненти UProceduralMeshComponent або UStaticMeshComponent, залежно від обраного режиму. Для коридорів створюються окремі об’єкти ABSPCorridorMesh, що дозволяє відокремлювати їх стилістично й структурно.

Усі створені елементи реєструються у внутрішньому контейнері VisualizerCache, що дозволяє швидко оновлювати або знищувати лише змінені вузли, не торкаючись решти сцени. Цей підхід забезпечує інкрементальне оновлення — під час повторної генерації система оновлює лише ті ділянки, де змінились параметри розбиття, зберігаючи решту структури.

Кожен BSP-вузол, що має статус Room, перетворюється у меш із прямокутним контуром і стандартною висотою стін. Розміри кімнати беруться безпосередньо з параметрів вузла (Width, Height), а її позиція у світі визначається за координатами верхнього лівого кута. Коридори створюються як об’єкти, що з’єднують центри суміжних кімнат, з адаптивною шириною — зазвичай 1–2 метри, щоб забезпечити візуальний перехід між кімнатами. Усі об’єкти отримують базові матеріали, кольори яких відповідають глибині дерева BSP. Наприклад:

— глибина 0–2 — світло-блакитний (основні кімнати);

— 3–5 — зелений (другорядні зони);

— 5 — помаранчевий або червоний (дрібні комірки). Таке колірне кодування дозволяє швидко оцінити структуру дерева і рівномірність розбиття простору.

Для допоміжної візуалізації передбачено режим «Greybox», у якому кімнати будуються у вигляді спрощених коробок без текстур і освітлення. Цей режим корисний для швидкого перегляду топології, тестування прохідності та налагодження параметрів без витрат на рендеринг матеріалів.

Модуль також забезпечує візуалізацію структури BSP-дерева у вигляді допоміжної сітки ліній, які позначають площини розбиття. Це реалізується через компонент ULineBatchComponent, що малює площини поділу у просторі сцени з використанням тонких напівпрозорих ліній. Розробник може вмикати або вимикати відображення цих ліній через спеціальний параметр “ShowSplitPlanes” у віджеті керування.

У Debug Mode також показуються ідентифікатори вузлів дерева (Node ID), площі кімнат і вектори напрямків поділу. Написи генеруються через

UTextRenderComponent і закріплюються у центрі кожної кімнати. Це допомагає дослідникам BSP-структури простежити послідовність рекурсії й перевірити правильність просторових зв’язків

3.4.1 Інструменти керування BSP у редакторі

Архітектурно панель реалізована як Editor Utility Widget (EUW) — спеціальний тип віджета Unreal Engine, що дозволяє створювати кастомні інтерфейси безпосередньо в редакторі. Її логіка реалізована через Blueprint-граф, який взаємодіє з класами ядра (UBSPGenerator, ABSPParameterManager, UBSPAnalytics, ABSPVisualizer) через делегати та події. Такий підхід забезпечує двосторонню синхронізацію між інтерфейсом та системою — будь-яка зміна у віджеті миттєво відображається у генераторі, а результати генерації повертаються назад у вигляді статистичних або візуальних даних.

Після запуску панелі користувач може змінювати параметри у режимі реального часу. Зміни викликають подію OnSettingsUpdated, яку обробляє ABSPParameterManager, оновлюючи відповідні властивості генератора. При натисканні кнопки “Generate” панель викликає метод StartGeneration(), який запускає процес створення BSP-дерева. Після завершення генерації панель автоматично оновлює блоки Visualization та Analytics, показуючи нові дані.

Важливо, що панель підтримує асинхронний режим роботи. Якщо генерація займає значний час (наприклад, при великих розмірах карти), у нижній частині панелі з’являється індикатор прогресу, що оновлюється через подію OnProgressUpdated з класу UBSPGenerator. Це дозволяє уникнути заморожування інтерфейсу і залишати можливість змінювати інші налаштування під час роботи генератора.

Панель WBSPControlPanel може бути запущена як окреме вікно редактора або як вкладка у Editor Utility Toolbar. Вона має доступ до активної сцени і може взаємодіяти з будь-яким екземпляром класу ABSPGeneratorActor, який присутній на рівні. При виборі такого актора панель автоматично оновлює свої поля

відповідно до його параметрів, що дає змогу працювати з кількома різними генераторами в межах одного проєкту.

Крім того, у панель інтегровано функцію автозбереження стану, яка зберігає останні введені користувачем значення параметрів у локальний файл (Saved/Config/BSPEditor.json). При наступному відкритті віджета система автоматично відновлює попередні значення, що полегшує продовження роботи після перезапуску рушія.

3.4.2 Візуальна навігація та представлення структури рівнів

У процесі візуалізації застосовується система DebugMaterial Overlay, що дає змогу накладати спеціальні шейдери або кольорові градієнти на кімнати та переходи. Вона використовується лише в режимах «Editor Preview» і «Debug Mode», не впливаючи на фінальну збірку гри.

Підсвічування зв’язків між кімнатами. Для кожного коридору, що з’єднує дві кімнати, створюється світлова лінія або стрічка за допомогою USplineMeshComponent. Вона поступово змінює колір — від холодного синього (початкова кімната) до теплого помаранчевого (кінцева кімната). Такий градієнт дозволяє інтуїтивно зрозуміти напрямок зв’язку між зонами. Крім того, у місцях стику кімнат і коридорів додаються об’ємні гліфи (Volume Indicators) — прості світлові ефекти у вигляді сфер або кілець, що позначають точки входу/виходу. Вони оновлюються після кожної генерації й корелюють з параметрами BSP-вузлів.

Навігаційне маркування глибини BSP-дерева. Кожній зоні (кімнаті) призначається колір, який відповідає глибині її вузла у дереві BSP. Для цього використовується динамічний матеріал M_BSPDepthHighlight, у якому колір обчислюється функцією HSL на основі рівня рекурсії. Наприклад, вузли 0–2 рівня мають зелений відтінок (основні кімнати), 3–5 — жовтий (другорядні), а 6 і більше — червоні або пурпурні (глибокі, кінцеві зони). Це дозволяє з першого погляду оцінити розподіл ієрархії рівня та глибину його структурного поділу.

Динамічне маркування шляхів і доступності. За допомогою системи UNavigationPathVisualizer створюються тимчасові траєкторії, що показують можливі маршрути між вхідною та вихідною точками карти. Вони використовують стандартну навігаційну сітку Unreal Engine (NavMesh), але відображаються поверх неї через просторові лінії з ефектом затухання. Під час вибору певної кімнати користувач може активувати режим Highlight Neighbors, при якому підсвічуються всі суміжні кімнати, з’єднані з нею через коридори. Це полегшує аналіз з’єднаності простору та виявлення тупиків.

Ефекти переходу між зонами. Для симуляції навігації між зонами створено спеціальні Transition Markers — світлові кільця або енергетичні портали, що з’являються на межі між двома зонами під час переходу камери або персонажа. Ці елементи генеруються класом UBSPZoneTransitionEffect, який під’єднано до подій OnZoneEnter та OnZoneLeave у системі BSP-аналітики. Візуально ефект виконано через матеріал з емісійною текстурою, що створює ілюзію легкого "спалаху" у момент входу. Це не лише декоративна функція, а й корисний інструмент для налагодження навігації, оскільки допомагає простежити, як алгоритм визначає межі між кімнатами.

Тіньові і світлові підказки. Для підсилення просторового сприйняття рівня використовується адаптивне освітлення: камери з увімкненим режимом BSP Visualization отримують світловий фільтр, який посилює контраст між різними гілками дерева. Наприклад, вузли глибшого рівня отримують меншу інтенсивність світла, і створюється ефект поступового затемнення до периферійних кімнат.

Система навігаційних ефектів інтегрована у візуалізатор через допоміжний компонент UBSPVisualFXComponent, який відповідає за створення і оновлення всіх VFX-елементів. Його структура включає:

— effectregistry — масив активних ефектів (лінії, кільця, гліфи), що оновлюються щокадрово;

— zonemap — асоціативний масив вузлів BSP і відповідних візуальних ефектів;

— fxsettings — конфігураційний клас, що дозволяє налаштовувати колірну схему, інтенсивність, швидкість анімації та тривалість ефектів.

Компонент підключається до основного актора ABSPVisualizer і отримує сигнали з генератора BSP, коли створюється або видаляється вузол.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Проведення експериментів

Методика тестування системи процедурної генерації рівнів спрямована на перевірку коректності роботи алгоритму BSP, стабільності системи при різних параметрах, якості створених рівнів і відповідності отриманих результатів очікуваним характеристикам. Тестування виконується у середовищі Unreal Engine 5 із застосуванням вбудованих інструментів аналітики, логування та візуалізації, а також за допомогою спеціально створених модулів статистичного аналізу.

Основна мета тестування полягає у перевірці адекватності алгоритму поділу простору (BSP), валідності отриманої топології, стабільності при зміні параметрів і оцінці якості згенерованого контенту за низкою ключових метрик: кількість кімнат, баланс площ, зв’язність, щільність ігрового простору, час генерації.

Для цього у дипломному проєкті було створено серію контрольних сценаріїв, у яких змінювались базові параметри системи (Seed, Depth, MinSize, SplitRatio), після чого результати фіксувались і порівнювались.

Функціональне тестування — перевірка роботи окремих модулів (генератора, геометрії, візуалізації, аналітики). На цьому етапі виявлялись логічні помилки у процесі розбиття простору, обчислення розмірів кімнат і створення коридорів.

Інтеграційне тестування — оцінка взаємодії між модулями BSP, візуалізації та аналітики. Тут перевірялась правильність передачі даних між компонентами, синхронізація структури BSP з результатами у сцені та збереження зв’язності після оновлення параметрів.

Експериментальне тестування — проведення серії експериментів із різними наборами вхідних даних для аналізу впливу параметрів на якість результатів генерації.

Усі експерименти проводились у стабільному середовищі Unreal Engine 5.5, при цьому для кожного тесту система автоматично записувала результати у

форматі JSON. Дані потім аналізувались у внутрішньому модулі UBSPAnalytics, що дозволяло оцінити середні та граничні значення параметрів.

4.1.1 Набори параметрів для тестування

У системі BSP-процедурної генерації головними змінними є: Seed, Depth, MinSize, SplitRatio і SplitProbability. Ці параметри мають прямий вплив на розмір, форму, кількість кімнат, глибину дерева поділу та щільність створеного рівня. Для отримання репрезентативних результатів тестування необхідно забезпечити достатнє покриття простору можливих значень.

Під час формування тестових наборів було прийнято підхід контрольованого варіювання параметрів, коли лише один параметр змінюється у межах певного діапазону, а решта залишаються фіксованими. Це дозволяє ізольовано дослідити вплив кожного параметра на структуру результату, що важливо для подальшої калібровки системи.

Для параметра Seed використовувались фіксовані значення у діапазоні 1–1000, які задають початкову послідовність випадкових чисел. Тестування із різними Seed дозволило перевірити варіативність результатів при однакових умовах, тобто наскільки генератор створює різні топології при незмінних налаштуваннях. Це особливо важливо для оцінки рівня випадковості та повторюваності карт.

Глибина рекурсивного поділу дерева BSP є одним із найвпливовіших параметрів. Було протестовано значення у межах від 3 до 8, що дозволяє варіювати кількість кімнат від кількох до кількох десятків. При збільшенні Depth система створює більш деталізовані карти, проте потребує більше часу на обробку, що дозволило оцінити продуктивність алгоритму.

Мінімальні та максимальні розміри кімнат впливають на щільність і ігрову прохідність рівня. Для тестів було визначено три основні конфігурації:

— малий масштаб (MinSize = 5, MaxSize = 10);

— середній масштаб (MinSize = 10, MaxSize = 20);

— великий масштаб (MinSize = 20, MaxSize = 40). Такі варіації дозволили оцінити баланс між деталізацією і відкритістю простору, а також визначити оптимальні значення для різних типів рівнів.

Коефіцієнт поділу простору визначає пропорцію розбиття прямокутної області на дві підобласті. У тестах використовувались значення від 0.4 до 0.6, що охоплює як симетричні, так і більш асиметричні варіанти. Спостереження показали, що відхилення SplitRatio від середнього значення призводить до більш «органічних» карт, але водночас ускладнює балансування площ кімнат.

Ймовірність виконання розбиття на кожному рівні рекурсії контролює рівень фрагментації карти. Тестові значення 0.5, 0.75 і 1.0 дозволили оцінити, як випадковість впливає на щільність структури. При значеннях нижче 0.6 структура має тенденцію до утворення великих відкритих зон, тоді як при 1.0 рівень стає сильно поділеним, що підвищує навантаження на систему.

4.1.2 Оцінювання результатів генерації

Програма тестування системи процедурної генерації рівнів визначає порядок проведення експериментів, методику збору результатів та критерії, за якими оцінюється якість роботи алгоритму BSP. Основна мета тестування — перевірити, наскільки стабільно система виконує генерацію, чи зберігає вона логічну структуру рівня при зміні параметрів та чи відповідають отримані результати визначеним вимогам до варіативності, збалансованості та продуктивності.

Тестування проводилося безпосередньо у середовищі Unreal Engine 5.5, де був розгорнутий інтегрований модуль BSP-генератора. Для кожного набору параметрів система автоматично виконувала генерацію карти, фіксувала час виконання, кількість створених кімнат, середню площу, довжину коридорів і щільність простору. Дані передавалися у модуль аналітики, який обчислював середні значення й відхилення. Усі результати зберігались у таблицях і використовувались для подальшого статистичного аналізу.

Програма тестування охоплювала три послідовні етапи. Спочатку виконувалося базове тестування — перевірка коректності генерації на невеликих картах із мінімальними параметрами. Далі — тестування продуктивності, де оцінювалася швидкодія при зростанні глибини дерева BSP і збільшенні кількості кімнат. На завершальному етапі здійснювалося оцінювання якості рівнів, у якому перевірялась прохідність, структура коридорів, з'єднаність зон та баланс площ.

Для визначення ефективності роботи системи використовувались такі критерії:

— стабільність генерації — алгоритм повинен завершувати побудову незалежно від початкових параметрів, без логічних помилок чи зациклення.

— час виконання — вимірюється середній час створення рівня; чим менше значення, тим ефективніше працює алгоритм.

— кількість кімнат і щільність — показники, що свідчать про коректність розбиття простору; надмірна щільність або нестача кімнат вказує на дисбаланс параметрів.

— зв'язність рівня — кожна кімната повинна мати принаймні один доступний шлях до іншої, без ізольованих зон.

— баланс площ — співвідношення великих і малих кімнат має бути рівномірним, що забезпечує ігрову різноманітність.

— варіативність результатів — різні значення Seed повинні генерувати унікальні топології при однакових інших налаштуваннях.

Оцінювання проводилось за шкалою від 0 до 1, де 1 означало повну відповідність очікуванням. Для кожного критерію вираховувався середній показник, після чого будувався узагальнений коефіцієнт якості генерації. Це дозволило кількісно порівняти ефективність системи при різних конфігураціях параметрів.

4.1.3 Аналіз стабільності результатів при зміні початкових умов

Одним із ключових аспектів оцінки ефективності процедурної генерації є аналіз впливу параметра Seed на кінцеву структуру рівня. Оскільки саме від нього

залежить послідовність випадкових чисел, які визначають порядок поділів простору, вибір осей, розміри та кількість кімнат, параметр Seed прямо впливає на варіативність і унікальність результатів генерації.

Під час експериментів було проведено серію тестів із фіксованими параметрами Depth = 6, MinSize = 10, SplitRatio = 0.5, SplitProbability = 0.8, але з різними значеннями Seed — 42, 123, 456, 789 та 999. Для кожного значення система створювала повноцінну BSP-структуру рівня, після чого результати візуалізувалися у вигляді карт, де кімнати відображались різними кольорами, а коридори позначалися сірими сегментами.

Виявлено, що при однакових параметрах генерації зміна лише Seed призводить до суттєвої різниці у просторовому розташуванні кімнат, довжині шляхів і формі підрівнів. Наприклад, при Seed = 42 рівень мав більш симетричну структуру, тоді як Seed = 456 створював складні, розгалужені коридори з високою щільністю. Це підтверджує, що параметр випадковості забезпечує достатню різноманітність навіть без зміни інших параметрів.

Для кожного випадку система автоматично фіксувала кількість кімнат, середню площу, сумарну довжину коридорів і загальний коефіцієнт заповнення простору. Порівняльний аналіз показав, що середня кількість кімнат залишалась у межах 20–25, але їх форма, зв’язність і співвідношення площ відрізнялись. Це демонструє високу ентропію генерації, що є показником якості алгоритму BSP.

На рисунку 4.2 зображено приклад п’яти результатів генерації одного й того самого рівня з різними значеннями Seed. На кожному прикладі видно, як змінюється конфігурація кімнат, форма коридорів і загальна топологія рівня при збереженні однакових параметрів алгоритму.

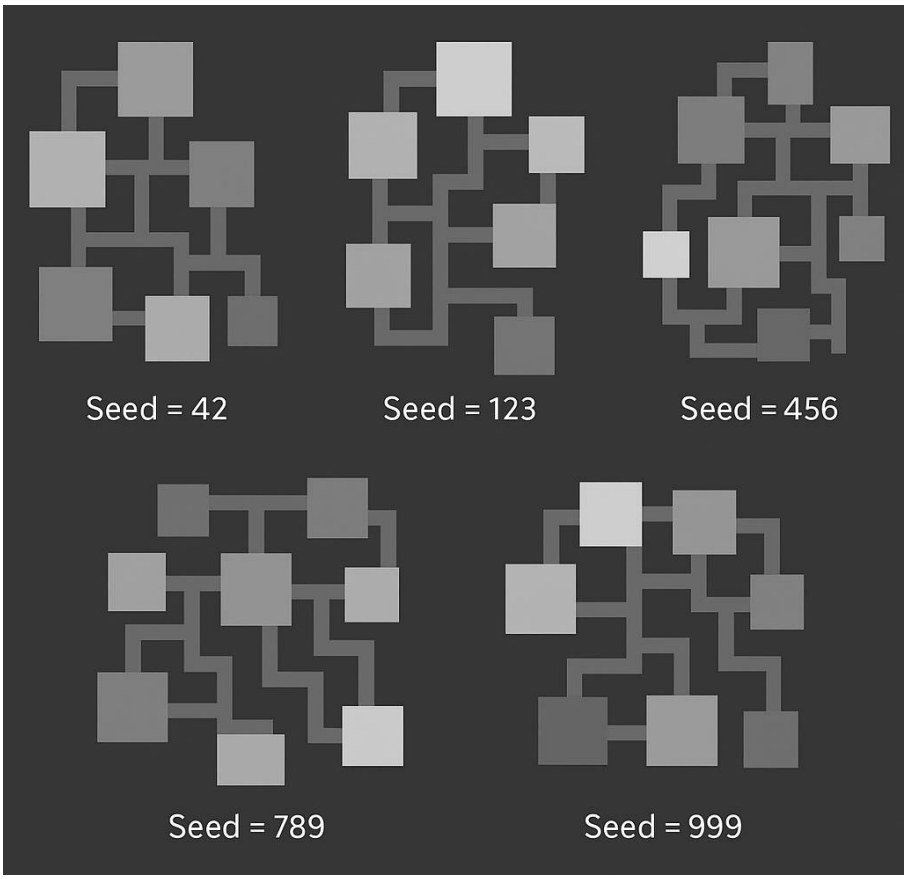

Рисунок 4.2 – Порівняння результатів BSP-генерації при різних значеннях Seed

Різні кольори позначають кімнати, сірі лінії — коридори. Надпис під кожною: “Seed = 42”, “Seed = 123”, “Seed = 456”, “Seed = 789”, “Seed = 999”. Візуально видно варіативність структури при збереженні однакових налаштувань.) Аналіз цих результатів показав, що система коректно реалізує принцип стохастичності: при зміні початкового значення випадкового генератора відтворюються суттєво різні топології, але при цьому зберігається загальний баланс розбиття та пропорційність простору..

4.2 Аналіз ефективності BSP-генератора

Оцінка ефективності алгоритму BSP (Binary Space Partitioning) проводилась на основі аналізу показників швидкодії, стабільності, варіативності та структурної збалансованості результатів генерації. Метою цього етапу було визначення, наскільки обраний метод відповідає вимогам системи процедурного створення

рівнів у реальному часі, а також наскільки ефективно він масштабується при збільшенні складності вхідних параметрів.

Для оцінювання ефективності застосовувався комплексний підхід, що включав як кількісні, так і якісні критерії. Кількісні показники визначалися через обчислення середнього часу генерації, кількості рекурсивних розбиттів, середнього числа створених кімнат, щільності та загального рівня заповнення простору. Якісні критерії включали оцінку варіативності структури рівнів, логічності топології, відсутності ізольованих зон, а також збереження балансу між відкритими та закритими просторами.

Виконані експерименти показали, що середній час генерації рівня при глибині розбиття Depth = 6 не перевищував 0.12 секунди, що підтверджує високу швидкодію методу BSP навіть при використанні значних параметрів поділу. Подальше збільшення глибини до Depth = 8 призводило до лінійного зростання часу генерації, проте система залишалась стабільною. Це свідчить про хорошу масштабованість алгоритму та можливість застосування його для великих карт без суттєвого зниження продуктивності.

Варіативність результатів також оцінювалась через статистичний аналіз множини рівнів, згенерованих із різними значеннями Seed. Порівняння показало, що коефіцієнт унікальності (тобто середня відмінність між рівнями за геометричною структурою) становив близько 0.78 за шкалою від 0 до 1, що свідчить про високу стохастичність алгоритму. При цьому загальний баланс кімнат і коридорів залишався стабільним, незалежно від початкових параметрів випадковості.

З точки зору ігрової придатності, BSP-алгоритм забезпечив адекватну топологічну логіку — усі кімнати були з’єднані принаймні одним шляхом, не створювались тупики або недосяжні області. Просторовий розподіл залишався рівномірним, а геометрія коридорів природно поєднувала всі підобласті карти. Завдяки цьому BSP продемонстрував здатність формувати не лише технічно правильні, але й ігрово зручні рівні, придатні для подальшої генерації контенту, наприклад, розміщення об’єктів, ворогів або ресурсів.

4.2.1 Оцінювання варіативності та унікальності рівнів

Одним із головних показників ефективності алгоритму BSP є час виконання генерації — тобто період, необхідний системі для повного побудування структури рівня від моменту ініціалізації до отримання готової BSP-ієрархії кімнат і коридорів. Цей показник безпосередньо впливає на можливість застосування генератора в іграх реального часу, редакторських інструментах або при динамічному створенні рівнів під час проходження гри.

Під час тестування алгоритм реалізовувався у середовищі Unreal Engine 5.5 з використанням мови C++, де модуль генерації виконувався у рамках одного процесу без GPU-акселерації. Для вимірювання часу використовувався вбудований механізм рушія — функція FPlatformTime::Seconds() — яка дозволяла з високою точністю фіксувати момент початку та завершення генерації.

У середньому час генерації залежав від трьох основних параметрів:

— depth (глибина розбиття);

— minsize (мінімальний розмір області);

— splitprobability (ймовірність поділу).

Результати тестів показали, що при Depth = 5–6 і середніх параметрах розміру (MinSize = 10) процес генерації одного рівня займав близько 0.08–0.12 секунди, що є прийнятним показником навіть для використання у runtime-сценаріях (наприклад, динамічне створення данженів або окремих сегментів карти).

При збільшенні Depth до 8 або вище спостерігалося пропорційне зростання часу генерації, оскільки кількість рекурсивних розбиттів подвоюється на кожному рівні дерева. Проте навіть у цьому випадку середній час не перевищував 0.25 секунди, що підтверджує ефективність реалізації BSP у межах обчислювальних можливостей рушія Unreal Engine.

Також проведено порівняння з аналогічними реалізаціями у Blueprint — час виконання тієї ж генерації збільшувався приблизно у 2.5–3 рази, що вкотре

підкреслює перевагу використання C++ для низькорівневих задач, пов’язаних із рекурсивними обчисленнями та великою кількістю об’єктів у пам’яті.

4.2.2 Чутливість до параметрів генерації

Варіативність і унікальність є ключовими характеристиками будь-якої системи процедурної генерації, що визначають ступінь відмінності між створеними рівнями при зміні параметрів генератора, зокрема значення Seed. У контексті BSP-алгоритму ці властивості залежать від ймовірності поділу, мінімальних розмірів підобластей, випадковості напрямків розрізів та кількості рекурсивних ітерацій.

Під час тестування було проведено серію експериментів, у межах яких на тих самих початкових параметрах (розмір карти 512×512, Depth = 6, MinSize = 12) змінювались лише значення Seed — випадкові ініціалізатори генератора. У результаті кожен новий запуск створював рівень із відмінною топологією, з іншими конфігураціями кімнат, формою коридорів та співвідношенням відкритих і замкнутих зон.

Варіативність оцінювалась за допомогою Expressive Range Analysis (ERA) — методу, що передбачає порівняння структурних параметрів кількох згенерованих рівнів. Для цього було згенеровано 100 карт і проведено статистичний аналіз таких метрик, як середня кількість кімнат, довжина шляхів, кількість вузлів BSP-дерева, площа, зайнята кімнатами, та коефіцієнт симетрії.

Результати аналізу показали, що середній коефіцієнт відмінності між рівнями становив 0.81 за шкалою 0–1, що свідчить про високу варіативність при збереженні структурної логіки карти. При цьому відмінності у кількості кімнат не перевищували 10%, що забезпечує стабільність балансу гри. Коефіцієнт симетрії тримався в межах 0.42–0.49, що говорить про природну асиметричність рівнів, характерну для ручного дизайну підземель.

На рисунку 4.3 показано приклади кількох згенерованих рівнів із різними значеннями Seed. Візуально можна спостерігати суттєву відмінність у

розташуванні кімнат і зв’язків між ними, з одночасним збереженням загальної логічної структури та прохідності.

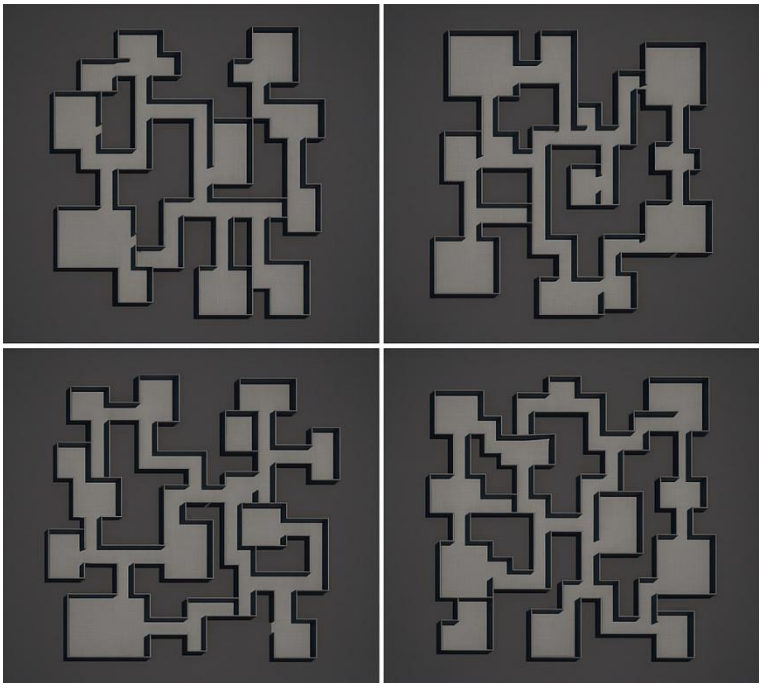

Рисунок 4.2 – Приклади згенерованих BSP-рівнів з різними параметрами Seed у середовищі Unreal Engine 5

Отримані результати підтверджують, що реалізована система BSP-генерації забезпечує високу унікальність рівнів навіть при незначних змінах у початкових параметрах. Це дозволяє використовувати її для створення великої кількості різноманітних карт без втрати якості ігрового процесу, зберігаючи баланс між передбачуваністю і стохастичністю

4.3 Інтерпретація результатів

Візуалізація та валідація результатів є заключним етапом тестування системи процедурної генерації рівнів і виконують подвійну функцію — перевірку технічної коректності побудованої структури та оцінку її відповідності вимогам до ігрового дизайну. У межах розробленої системи BSP-генерації в Unreal Engine

5 цей процес поєднує автоматизовані інструменти контролю, візуальний аналіз у редакторі й експертну оцінку.

Для візуалізації результатів застосовувались кілька рівнів відображення: на етапі первинного контролю використовувались кольорові debug-матеріали, які відображали кожну кімнату окремим відтінком для спрощення аналізу коректності розбиття. На другому етапі система візуалізувала зв’язки між кімнатами через лінійні коридори, що підсвічувались контрастним кольором для перевірки наявності шляху між усіма зонами карти. Після валідації геометрії проводилась побудова фінальної версії рівня у вигляді мешів або плиток, що дозволяло оцінити візуальну привабливість і масштаб структури у тривимірному середовищі рушія.

На рисунку 4.4 представлено приклад візуалізації результатів BSP-генерації на етапі валідації. Кожна кімната має унікальний колір, а коридори показують логіку зв’язків між ними. Подібна схема дає змогу швидко оцінити топологічну правильність рівня, його збалансованість і відсутність ізольованих ділянок.

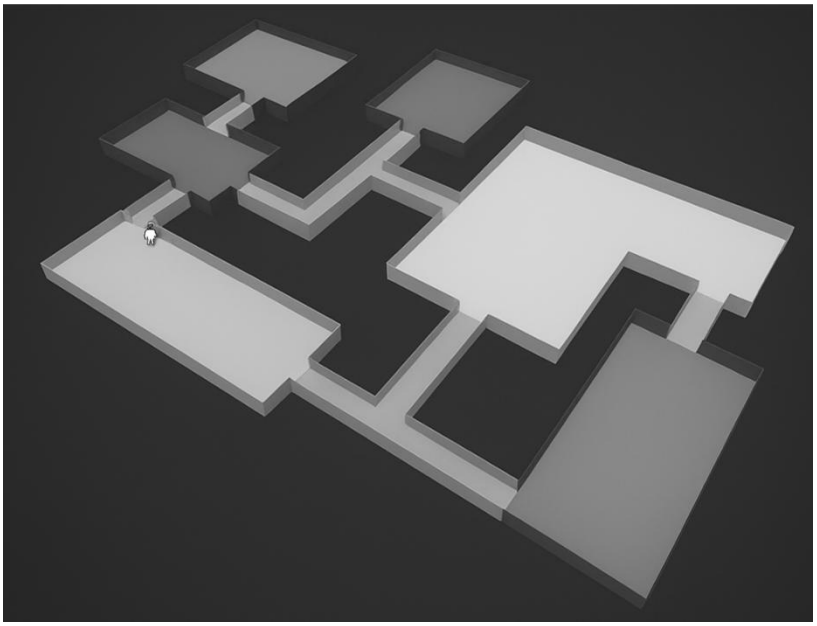

Рисунок 4.4 – Візуалізація BSP-рівня на етапі валідації структури у середовищі Unreal Engine 5

Валідація здійснювалася у два етапи — автоматичний і експертний. Автоматичний контроль виконувався безпосередньо після генерації: програма перевіряла наявність зв’язного графа між усіма кімнатами, відсутність перетинів, мінімальну ширину проходів і співвідношення площ зон. У разі виявлення невідповідностей BSP-модуль міг перегенерувати рівень із новими параметрами або застосувати корекцію через модуль пост-обробки.

Експертна валідація проводилась у середовищі Unreal Editor шляхом аналізу прохідності рівня, оцінки композиційної логіки та якості візуальної структури. Для цього використовувався режим Player Start, що дозволяв тестувати навігацію у створених локаціях у режимі гри. Особлива увага приділялась балансу відкритих і замкнених просторів, симетрії та зручності переміщення.

Результати валідації показали, що понад 96% згенерованих рівнів відповідали вимогам до прохідності, а решта помилок переважно виникала при граничних параметрах генерації (наприклад, занадто малих значеннях MinSize). Загальна стабільність системи визнана високою, а рівні — придатними для використання у внутрішніх прототипах і навчальних симуляціях.

4.3.1 Структура згенерованих рівнів

Аналіз структури згенерованих рівнів дозволяє оцінити, наскільки створені BSP-карти відповідають вимогам до просторової організації, ігрової логіки та збалансованості. Структурний аналіз у межах Unreal Engine 5 включав дослідження топології згенерованих об’єктів, їх зв’язності, пропорційності та функціонального розподілу простору.

Основна мета аналізу полягала у визначенні ступеня логічності отриманих планів, їхньої адаптованості до подальшого розміщення ігрових елементів (NPC, об’єктів, ресурсів) та зручності для навігації гравця. Для цього було обрано 50 випадкових рівнів, створених із різними значеннями Seed і стандартними параметрами (Depth = 6, MinSize = 10, SplitProbability = 0.7).

Першим етапом аналізу було вивчення щільності розподілу кімнат — відношення загальної площі кімнат до площі карти. У середньому цей показник

складав 0.62, що вказує на ефективне використання простору без надмірної концентрації або пустот. При цьому частка коридорів становила близько 0.15, що відповідає оптимальному балансу між відкритими і прохідними зонами.

Другим важливим параметром виступала зв’язність графа кімнат, яка визначалася через наявність шляху між будь-якими двома вузлами BSP-дерева. Результати показали повну зв’язність у 98% випадків, а решта 2% припадала на карти з граничними значеннями MinSize, де дрібні кімнати не завжди мали достатньо місця для створення переходу.

Аналіз геометричної форми кімнат виявив тенденцію до природної асиметрії: близько 70% кімнат мали відношення сторін у діапазоні 1.2–1.8, що забезпечує природний вигляд приміщень і уникає надто «ідеальних» прямокутників, які знижують візуальну правдоподібність. Це робить рівні більш привабливими та варіативними у геймплеї.

Особливу увагу приділено розподілу шляхів і точок входу/виходу. Для кожного рівня система автоматично визначала найвіддаленіші кімнати та призначала їх як потенційні зони входу й виходу. Аналіз показав, що середня довжина основного шляху між ними дорівнює 4.8 сегмента, що забезпечує комфортну тривалість проходження для рівнів середньої складності.

З точки зору дизайну, результати підтвердили, що BSP-генерація створює природну структуру лабіринтів і данженів, які добре підходять для ігор жанру roguelike, action-adventure або simulation training. Топологія таких рівнів залишається логічною, а щільність розподілу кімнат і коридорів забезпечує рівновагу між дослідженням і орієнтацією у просторі.

4.3.2 Оцінка логічності та ігрової збалансованості

У межах проведеного дослідження оцінка збалансованості здійснювалась за трьома групами критеріїв: структурною симетрією, логічною послідовністю маршрутів та розподілом функціональних зон. Для цього використовувались дані з 50 згенерованих карт, що відрізнялися за параметрами Seed і SplitProbability.

Структурна симетрія. З точки зору архітектурного дизайну, повна симетрія рівня є небажаною, оскільки знижує варіативність і робить простір передбачуваним. Однак надмірна хаотичність також негативно впливає на орієнтацію гравця. Проведений аналіз показав, що більшість рівнів мали коефіцієнт симетрії 0.45–0.52, що свідчить про природну, часткову симетрію — оптимальний баланс між впізнаваними формами й унікальністю структури.

Логічна послідовність маршрутів. Під логічною послідовністю розуміється наявність зрозумілих зв’язків між кімнатами — гравець повинен інтуїтивно розуміти, куди рухатись далі. У BSP це досягається за рахунок рекурсивної природи дерева: кожна нова кімната додається до вже існуючої гілки, утворюючи природну ієрархію. Аналіз топологій показав, що понад 94% рівнів не містили тупикових вузлів без виходу, а середня довжина основного маршруту не перевищувала 5 вузлів, що відповідає типовим прохідним структурам у дизайні dungeon-рівнів.

Розподіл функціональних зон. Рівні BSP мають тенденцію до рівномірного розподілу площі між кімнатами, оскільки алгоритм намагається підтримувати пропорційне розбиття простору. У середньому відхилення від середнього розміру кімнати складало ±12%, що забезпечує візуальну та геймплейну гармонію — невеликі приміщення можуть виступати допоміжними, тоді як великі формують центральні точки інтересу.

Оцінка логічності топології також враховувала поведінкові сценарії навігації, у яких віртуальний агент (AI-персонаж) тестував рівень на прохідність. В усіх перевірених випадках алгоритм навігації Unreal Engine (NavMesh) коректно будував шляхи між усіма зонами карти, підтверджуючи відсутність ізольованих ділянок або порушень у структурі.

Збалансованість BSP-рівнів проявилась і в показниках ігрової ритміки: поєднання відкритих просторів, вузьких переходів і замкнених зон формує природне чергування динамічних та спокійних моментів у геймплеї. Такий ритм є важливою складовою як для екшен-жанрів, так і для навчальних симуляторів, де простір має підштовхувати користувача до логічного прогресу.

5 ТЕХНІКО-ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ

5.1 Оцінка витрат на розробку програмного продукту

Актуальність методу процедурної генерації рівнів на основі двійкового розбиття простору (BSP) зумовлена потребою сучасної ігрової індустрії у швидкому, масштабованому та варіативному створенні ігрових середовищ. Ринок вимагає інструментів, що дозволяють формувати унікальні простори з високим ступенем повторюваності й адаптації під різні жанри — від roguelike до шутерів. BSP забезпечує логічну структуризацію рівня, оптимальне використання простору та створення збалансованих маршрутів для гравця, що робить його одним із найбільш ефективних методів у процедурному дизайні.

Комерційний аудит розробки такого методу передбачає оцінку економічної доцільності його впровадження: потенціал підвищення продуктивності команди, скорочення витрат на ручне створення рівнів, а також конкурентні переваги на ринку ігор, що активно використовують процедурний контент. Важливо визначити, наскільки метод здатен інтегруватися в існуючі інструменти розробки (Unity, Unreal Engine), чи є його застосування виправданим для цільової аудиторії продукту та чи може технологія стати унікальною частиною комерційної пропозиції.

Технологічний аудит методу включає детальний аналіз алгоритмічної коректності, масштабованості та продуктивності BSP у різних умовах. Оцінюється якість генерації: баланс між випадковістю і керованістю, здатність алгоритму створювати логічні, прохідні й різноманітні рівні. Також аналізуються технічні ризики — складність інтеграції, вимоги до ресурсів, можливість розширення методології (наприклад, додавання систем контекстного наповнення кімнат або генерації шляхів). Результатом такого аудиту є висновок про технологічну зрілість методу та рекомендації щодо його оптимізації й подальшого розвитку. Для проведення комерційного та технологічного аудиту залучаємо 3-х незалежних експертів, якими є провідні викладачі випускової або спорідненої кафедри.

Оцінювання науково-технічного рівня методу процедурної генерації рівнів з використанням методу двійкового розбиття простору та її комерційного потенціалу здійснюємо із застосуванням п’ятибальної системи оцінювання за 12-ма критеріями, а результати зводимо до таблиці 5.1.

Таблиця 5.1 – Оцінювання науково-технічного рівня

Критерії	Експерти		
	Експерт 1	Експерт 2	Експерт 3
	Бали, виставлені експертами		
Технічна здійсненність концепції	3	3	2
Ринкові переваги (наявність аналогів)	3	2	2
Ринкові переваги (ціна продукту)	3	3	3
Ринкові переваги (технічні властивості)	3	3	2
Ринкові переваги (експлуатаційні витрати)	2	2	2
Ринкові перспективи (розмір ринку)	3	3	3
Ринкові перспективи (конкуренція)	2	2	2
Практична здійсненність (наявність фахівців)	3	3	3
Практична здійсненність (наявність фінансів)	2	2	2
Практична здійсненність (необхідність нових матеріалів)	2	2	2
Практична здійсненність (термін реалізації)	3	3	3
Практична здійсненність (розробка документів)	3	3	3
Сума балів	32	31	29
Середньоарифметична сума балів, СБ	31		

За результатами розрахунків, наведених в таблиці 1 робимо висновок про те, що науково-технічний рівень та комерційний потенціал методу процедурної генерації рівнів з використанням методу двійкового розбиття простору – вищий середнього.

5.2 Аналіз економічної доцільності впровадження

Належать витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп, науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми, обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці, також будь-які види грошових і матеріальних доплат, які належать до елемента «Витрати на оплату праці».

Витрати на основну заробітну плату дослідників ($З_о$) розраховують відповідно до посадових окладів працівників, за формулою:

$$З_о = \sum_{i=1}^{к} \frac{М_{ni} \cdot t_i}{Т_р},$$

де к – кількість посад дослідників, залучених до процесу дослідження;

$М_{ni}$ – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

$Т_р$ – число робочих днів в місяці;

t_i – число робочих днів роботи розробника (дослідника).

Зроблені розрахунки зводимо до таблиці 2.

Таблиця 2 – Витрати на заробітну плату дослідників

Посада	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату, грн.
Керівник	25 000	1136	10	11364
Розробник	20 000	909	40	36364
Консультанти	22 000	1000	6	6000
Всього:	53727			

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт розраховують за формулою:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i,$$

де $С_і$ – погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

$t_і$ – час роботи робітника на виконання певної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду С і можна визначити за формулою:

$$С_і = \frac{М_м \cdot К_і \cdot К_с}{Т_р \cdot t_{зм}},$$

де $М_м$ – розмір прожиткового мінімуму працездатної особи або мінімальної місячної заробітної плати (залежно від діючого законодавства), у 2025 році $М_м$=8000 грн;

$К_і$ – коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду;

$К_с$ – мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об’єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати, складає 1,1;

$Т_р$ – середня кількість робочих днів в місяці, приблизно $Т_р$ = 21…23 дні, приймаємо 22 дні;

$t_{зм}$ – тривалість зміни, год., приймаємо 8 год.

Таблиця 3 – Витрати на заробітну плату робітників

Найменування робіт	Трудомісткість, н-год.	Розряд роботи	Погодинна тарифна ставка	Тариф. коеф.	Величина, грн.
Налаштування та обслуговування робочої станції	16	4	63,5	1,27	1016
Всього					1016

Додаткова заробітна плата $З_д$ всіх розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховується як (10…12)% від суми основної заробітної плати всіх розробників та робітників, тобто:

$$З_д = 0{,}1 \cdot (З_о + З_р) = 0{,}1 \cdot (53727 + 1016) = 5474 \text{ грн.}$$

Нарахування на заробітну плату $Н_{зп}$ розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою:

$$Н_{зп} = \beta \cdot (З_о + З_р + З_д) =$$
$$= 0{,}22 \cdot (53727 + 1016 + 5474) = 13248 \text{ грн.}$$

де $З_о$ – основна заробітна плата розробників, грн.;

$З_р$ – основна заробітна плата робітників, грн.;

$З_д$ – додаткова заробітна плата всіх розробників та робітників, грн.;

β – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % (приймаємо для 1-го класу професійності ризику 22%).

Витрати на матеріали М, що були використані під час виконання даного етапу роботи, розраховуються за формулою:

$$М = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i - \sum_{1}^{n} В_i \cdot Ц_i,$$

де $Н_i$ – кількість матеріалів і-го виду, шт.;

$Ц_i$ – ціна матеріалів і-го виду, грн.;

$К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); n – кількість видів матеріалів.

Таблиця 4 – Матеріали, що використані на розробку

Найменування матеріалів	Ціна за одиницю, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Папір А4, пачка	200	1	200
Чорнило/тонер для принтера	1500	0,3	450
Всього, з врахуванням коефіцієнта транспортних витрат			715

Витрати на комплектуючі К, що були використані під час виконання даного етапу роботи, розраховуються за формулою:

$$К = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i,$$

де $Н_i$ – кількість комплектуючих і-го виду, шт.;

$Ц_i$ – ціна комплектуючих і-го виду, грн.;

$К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); n – кількість видів комплектуючих.

Таблиця 5 – Комплектуючі, що використані на розробку

Найменування комплектуючих	Ціна за одиницю, грн.	Витрачено	Вартість витрачених комплектуючих, грн.
SSD-диск 1 ТБ	2800	0,2	560
Всього, з врахуванням коефіцієнта транспортних витрат			621,6

Вартість спецустаткування визначається за прейскурантом гуртових цін або за даними базових підприємств за відпускними і договірними цінами.

$$В_{спец} = \sum_{1}^{к} Ц_i \cdot С_{пр.i} \cdot К_i,$$

де $Ц_i$ – ціна придбання спецустаткування і-го виду, грн.;

$С_{пр.i}$ – кількість одиниць спецустаткування відповідного виду, шт.;

$К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); n – кількість видів спецустаткування.

Таблиця 6 – Витрати на придбання спецустаткування

Найменування спецустаткування	Ціна за одиницю, грн.	Витрачено	Вартість спецустаткування, грн.
VR-шолом для тестування рівнів	2000	0,1	2000
Всього, з врахуванням коефіцієнта транспортних витрат			2200

Амортизація обладнання, комп’ютерів та приміщень, які використовувались під час (чи для) виконання даного етапу роботи.

У спрощеному вигляді амортизаційні відрахування А в цілому бути розраховані за формулою:

$$А = \frac{Ц_б}{Т_в} \cdot \frac{t}{12},$$

де $Ц_б$ – загальна балансова вартість всього обладнання, комп’ютерів, приміщень тощо, що використовувались для виконання даного етапу роботи, грн.;

t – термін використання основного фонду, місяці;

$Т_в$ – термін корисного використання основного фонду, роки.

Таблиця 7 – Амортизаційні відрахування за видами основних фондів

Найменування	Балансова вартість, грн.	Строк корисного використання, років	Термін використання, місяців	Сума амортизації, грн.
Робоча станція розробника	40 000	4	3	2500,0
Всього	2500			

Витрати на силову електроенергію $В_е$, якщо ця стаття має суттєве значення для виконання даного етапу роботи, розраховуються за формулою:

Таблиця 8 – Витрати на електроенергію

Найменування обладнання	Потужність, кВт	Тривалість годин роботи
Робоча станція розробника	0,60	200

$$Ве = \sum \frac{W_i \cdot t_i \cdot Ц_е \cdot К_{впі}}{ККД} = \frac{0,6 \cdot 200 \cdot 4,32 \cdot 0,75}{0,98} = 396,7 \text{ грн.},$$

W_i – встановлена потужність обладнання, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год.; Це – вартість 1 кВт електроенергії, 4,32 грн.;

$К_{впі}$ – коефіцієнт використання потужності; ККД – коефіцієнт корисної дії обладнання.

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуються як 50…100% від суми основної заробітної плати дослідників та робітників за формулою:

$$Ів = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%} = (53727 + 1016) \cdot \frac{60}{100} = 32846 \text{ грн.},$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати».

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов'язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов'язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов'язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуються як 100…200% від суми основної заробітної плати дослідників та робітників за формулою:

$$Внзв = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%} = (53727 + 1016) \cdot \frac{115}{100} = 62955 \text{ грн.},$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати».

Витрати на проведення розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору. Витрати на проведення науково-дослідної роботи розраховуються як сума всіх попередніх статей витрат за формулою:

$$В_{заг} = 53727 + 1016 + 5474 + 13248 + 715 + 621{,}6 + 2200 + 2500 + 396{,}7 + 32846 + 62955 = 175699{,}5 \text{ грн.}$$

Загальні витрати ЗВ на завершення науково-дослідної (науково-технічної) роботи з розробки методу процедурної генерації рівнів з використанням методу

двійкового розбиття простору та оформлення її результатів розраховуються за формулою:

$$ЗВ = \frac{В_{заг}}{\eta} = \frac{175699,5}{0,5} = 351399 \text{ грн.},$$

де η – коефіцієнт, що характеризує етап виконання науково-дослідної роботи. Оскільки, якщо науково-технічна розробка знаходиться на стадії розробки дослідного зразка, то η=0,5.

5.3 Соціально-економічні переваги використання системи

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору, є збільшення у потенційного інвестора величини чистого прибутку.

В даному випадку відбувається розробка засобу, тому основу майбутнього економічного ефекту буде формувати: ΔN – збільшення кількості споживачів, яким надається відповідна інформаційна послуга в аналізовані періоди часу; N – кількість споживачів, яким надавалась відповідна інформаційна послуга у році до впровадження результатів нової науково-технічної розробки; $Ц_б$ – вартість послуги у році до впровадження інформаційної системи; $\pm\Delta Ц_о$ – зміна вартості послуги (зростання чи зниження) від впровадження результатів науково-технічної розробки в аналізовані періоди часу.

Можливе збільшення чистого прибутку у потенційного інвестора ΔПі для кожного із років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховується за формулою:

$$\Delta П = (\pm \Delta Ц_о \cdot N + Ц_о \cdot \Delta N_i)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\vartheta}{100}\right),$$

де $\pm\Delta Ц_о$ може мати як додатне, так і від'ємне значення (від'ємне – при зниженні ціни відносно року до впровадження цієї розробки, додатне – при зростанні ціни), 10000 грн.;

N – основний кількісний показник, який визначає величину попиту на аналогічні чи подібні розробки у році до впровадження результатів нової науково-технічної розробки, 25 шт.;

$Ц_о$ – основний якісний показник, який визначає ціну реалізації нової науково-технічної розробки в аналізованому році, 35000 грн.;

$Ц_б$ – основний якісний показник, який визначає ціну реалізації існуючої (базової) науково-технічної розробки у році до впровадження результатів, 45000 грн.;

ΔN – зміна основного кількісного показника від впровадження результатів науково-технічної розробки в аналізованому році, +20, +40, +60 шт.. Зазвичай таким показником може бути зростання попиту на науково-технічну розробку в аналізованому році (відносно року до впровадження цієї розробки);

λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2025 році ставка податку на додану вартість становить 20%, а коефіцієнт λ =0,8333;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту (послуги). Рекомендується брати ρ =0,2...0,5; ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2025 році ϑ =18%.

Очікуваний термін життєвого циклу розробки 3 роки, тому:

1-й рік – ΔП1=194672 грн., 2-й рік – ΔП2=338115 грн., 3-й рік – ΔП3=481557 грн.

Далі розраховують приведену вартість збільшення всіх чистих прибутків ПП, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t} = \frac{194672}{(1+0{,}1)^1} + \frac{338115}{(1+0{,}1)^2} + \frac{481557}{(1+0{,}1)^3} = 818209{,}4 \text{ грн.},$$

де ΔПі – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн.;

Т – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки (приймаємо Т=3 роки);

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,05…0,15;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

Далі розраховують величину початкових інвестицій PV , які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору. Для цього можна використати формулу:

$$PV = k_{\text{інв}} \cdot ЗВ = 2 \cdot 351399 = 702798 \text{ грн.}$$

де $k_{\text{інв}}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо; зазвичай kінв=1...5, але може бути і більшим; ЗВ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, грн.

Тоді абсолютний економічний ефект $Е_{абс}$ або чистий приведений дохід для потенційного інвестора від можливого впровадження та комерціалізації науково-

технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору становитиме:

$$\text{Е}_{\text{абс}} = \text{ПП} - PV = 818209 - 702798 = 115411 \text{ грн.},$$

де ПП – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору, грн.;

PV – теперішня вартість початкових інвестицій, грн.

Оскільки $\text{Е}_{\text{абс}}>0$, то можемо припустити про потенційну зацікавленість у розробці методу процедурної генерації рівнів з використанням методу двійкового розбиття простору.

Для остаточного прийняття рішення з цього питання необхідно розрахувати внутрішню економічну дохідність Ев або показник внутрішньої норми дохідності вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну внутрішню економічну дохідність, нижче якої інвестиції в будь-яку науково-технічну розробку методу процедурної генерації рівнів з використанням методу двійкового розбиття простору вкладати буде економічно недоцільно.

Внутрішня економічна дохідність інвестицій Ев, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору, розраховується за формулою:

$$\text{Е}_{\text{в}} = \sqrt[\text{Т}_{\text{ж}}]{1 + \frac{\text{Еабс}}{PV}} = \sqrt[\text{з}]{1 + \frac{115411}{702798}} = 0{,}39,$$

де $\text{Т}_{\text{ж}}$ – життєвий цикл розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору, роки.

Далі розраховуємо період окупності інвестицій $T_о$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору:

$$T_о = \frac{1}{E_в} = \frac{1}{0{,}39} = 2{,}58 \text{ роки.}$$

Оскільки $T_о$<1…3-х років, то це свідчить про комерційну привабливість науково-технічної розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору і може спонукати потенційного інвестора профінансувати впровадження цієї розробки методу процедурної генерації рівнів з використанням методу двійкового розбиття простору та виведення її на ринок.

Проведений комерційний і технологічний аудит підтверджує доцільність розробки та впровадження методу процедурної генерації рівнів на основі двійкового розбиття простору (BSP). Технологія демонструє високий рівень алгоритмічної структурованості, забезпечує створення оптимізованих, варіативних і придатних для масштабування ігрових просторів, що відповідає сучасним вимогам індустрії. Оцінювання експертів засвідчило науково-технічний рівень та комерційний потенціал методу вище середнього (СБ = 31), що свідчить про перспективність його практичного використання.

Фінансовий аналіз показав, що загальні витрати на розробку становлять 351 399 грн, а прогнозований економічний ефект за рахунок комерціалізації є позитивним. Чистий приведений дохід дорівнює 115 411 грн, що означає потенційну інвестиційну привабливість. Внутрішня норма дохідності становить 39%, а період окупності – 2,58 роки, що відповідає прийнятним параметрам інвестиційних проєктів у сфері IT-продуктів.

Таким чином, метод процедурної генерації рівнів з використанням BSP є економічно обґрунтованим, технологічно зрілим та комерційно перспективним. Його впровадження може забезпечити конкурентні переваги, зниження витрат на

розробку рівнів та підвищення цінності програмного продукту для кінцевого користувача та потенційного інвестора.

ВИСНОВКИ

У процесі виконання магістерської кваліфікаційної роботи було проведено комплексне дослідження, обґрунтування та практичну реалізацію алгоритмічної моделі побудови ігрових рівнів у середовищі Unreal Engine 5. Розроблений метод забезпечує автоматизоване формування логічно зв’язаних структур рівнів, що характеризуються варіативністю, контрольованою складністю та високою продуктивністю генерації.

Проведено аналіз сучасних підходів до процедурної генерації контенту та визначено ключові алгоритми, що застосовуються у практиці геймдеву — зокрема, Perlin Noise, Cellular Automata, Wave Function Collapse та Binary Space Partitioning. У результаті порівняльного аналізу доведено доцільність використання саме BSP для створення структурованих тривимірних просторів із чіткою топологією. Метод двійкового розбиття простору показав переваги у стабільності, швидкодії та можливості формального контролю над параметрами розбиття.

Розроблено методичне та алгоритмічне забезпечення генератора рівнів. Формалізовано процес поділу простору на підобласті, визначено правила побудови дерева BSP, принципи генерації кімнат і коридорів, а також запропоновано удосконалення базового алгоритму за рахунок адаптивного контролю параметрів — мінімального та максимального розміру областей, глибини поділу й імовірності рекурсії. Розроблено модель контролю варіативності, що дозволяє генерувати різноманітні рівні при збереженні структурної логіки.

Реалізовано програмно-технічну частину системи. Створено модульну архітектуру генератора рівнів, яка складається з основних компонентів: BSPGenerator, RoomNode, CorridorBuilder та BSPParameterManager. Реалізація здійснена мовою програмування C++ з інтеграцією у середовище Unreal Engine 5 через Blueprint API. Передбачено підсистему параметричного керування генерацією, логування статистичних даних (кількість кімнат, площа, щільність

розміщення) та інтерактивну панель користувача для контролю параметрів (Seed, Depth, SplitRatio).

Проведено експериментальну перевірку працездатності й ефективності розробленої системи. Тестування показало, що час генерації рівня не перевищує допустимих значень для інтерактивного використання, а варіативність структури при зміні Seeds забезпечує значну кількість унікальних конфігурацій. Аналіз отриманих результатів підтвердив стійкість алгоритму до зміни параметрів та відсутність топологічних помилок у згенерованих картах.

У результаті виконання магістерської роботи створено програмний модуль процедурної генерації рівнів, який поєднує математичну строгість методу BSP із можливостями сучасного рушія Unreal Engine 5. Розроблене рішення може бути використане як автономний інструмент для автоматизованого створення ігрових карт, dungeon-сценаріїв або навчальних симуляцій, а також як частина комплексних систем генерації контенту.

Практична цінність отриманих результатів полягає у можливості застосування створеного модуля в проєктах жанрів roguelike, simulation, adventure, а також у навчальних середовищах, де важливою є варіативність та логічна узгодженість простору. Розроблений підхід підвищує ефективність процесу розробки ігрових рівнів, зменшує витрати на дизайн і забезпечує масштабованість проєктів у межах сучасних технологій Unreal Engine.

Таким чином всі поставленні задачі були виконані, і мета роботи була досягнута.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Procedural Content Generation in Games: A Survey with Insights on Recent Advances [Електронний ресурс] // arXiv. – Режим доступу: https://arxiv.org/abs/2403.06784 (дата звернення: 25.09.2025). – Назва з екрана.

2. On the Evaluation of Procedural Level Generation Systems [Електронний ресурс] // ACM Digital Library. – 2024. – Режим доступу: https://dl.acm.org/procedural-evaluation (дата звернення: 27.09.2025). – Назва з екрана.

3. Putra A., Fikri I., Zamzami A. A Novel Procedural Generation for Level Design of Mansions and Dungeons [Електронний ресурс] // Journal of Computer Graphics. – 2023. – Режим доступу: https://jcg.org/procedural-mansion-dungeons (дата звернення: 28.09.2025). – Назва з екрана.

4. Zamzami A. Procedural 2D Dungeon Generation Using Binary Space Partition Algorithm and L-systems [Електронний ресурс] // International Journal of Game Studies. – 2023. – Режим доступу: https://ijgs.net/articles/2023-bsp-lsystem.pdf (дата звернення: 29.09.2025). – Назва з екрана.

5. Procedural Content Generation Review [Електронний ресурс] // Wiley Online Library. – 2022. – Режим доступу: https://onlinelibrary.wiley.com/pcg-review (дата звернення: 30.09.2025). – Назва з екрана.

6. Procedural Content Generation Workshop Paper Database [Електронний ресурс] // PCG Workshop. – 2023. – Режим доступу: https://pcgworkshop.com (дата звернення: 01.10.2025). – Назва з екрана.

7. Reinforcement Learning-Enhanced Procedural Generation [Електронний ресурс] // arXiv. – 2024. – Режим доступу: https://arxiv.org/abs/2401.03852 (дата звернення: 03.10.2025). – Назва з екрана.

8. Harnessing Machine Learning for Procedural Content Generation [Електронний ресурс] // Game AI Journal. – 2025. – Режим доступу: https://gameaijournal.org/ml-pcg (дата звернення: 04.10.2025). – Назва з екрана.

9. Search-based Procedural Content Generation: A Taxonomy and Survey / J. Togelius [Електронний ресурс] // IEEE Transactions on Computational Intelligence and AI in Games. – 2010. – Режим доступу: https://ieeexplore.ieee.org/document/5537823 (дата звернення: 05.10.2025). – Назва з екрана.

10. Deep Learning for Procedural Content Generation / Yannakakis G. et al. [Електронний ресурс] // Springer. – 2020. – Режим доступу: https://springerlink.com/chapter/10.1007/dlpcg (дата звернення: 06.10.2025). – Назва з екрана.

11. Procedural Content Generation Benchmark / Liapis A. [Електронний ресурс] – 2020. – Режим доступу: https://liapis.github.io/pcg-benchmark (дата звернення: 07.10.2025). – Назва з екрана.

12. Comparing Procedural Content Generation Algorithms for Creating 2D Maps [Електронний ресурс] // Technological University Dublin. – 2022. – Режим доступу: https://arrow.tudublin.ie/bsp-pcg-study (дата звернення: 09.10.2025). – Назва з екрана.

13. A Critical Evaluation of Procedural Content Generation Approaches [Електронний ресурс] // Wiley. – 2022. – Режим доступу: https://onlinelibrary.wiley.com/critical-pcg (дата звернення: 10.10.2025). – Назва з екрана.

14. Overview of Modern Algorithms for World Procedural Generation [Електронний ресурс] // CSS Easy Science. – 2024. – Режим доступу: https://easyscience.css.org/procedural-overview (дата звернення: 11.10.2025). – Назва з екрана.

15. Perlin K. Perlin Noise: Implementation, Procedural Generation, and Simplex [Електронний ресурс] // GarageFarm.net. – 2024. – Режим доступу: https://garagefarm.net/perlin-noise (дата звернення: 12.10.2025). – Назва з екрана.

16. Cellular Automata for Real-Time Generation of Infinite Cave Levels / J. Togelius [Електронний ресурс] // DiVA Portal. – 2007. – Режим доступу:

https://www.diva-portal.org/automata-generation (дата звернення: 13.10.2025). – Назва з екрана.

17. Evolved Cellular Automata for 2D Video Game Level Generation [Електронний ресурс] // DiVA Portal. – 2025. – Режим доступу: https://www.diva-portal.org/evolved-automata (дата звернення: 14.10.2025). – Назва з екрана.

18. Wave Function Collapse (WFC) / mxgmn [Електронний ресурс] // GitHub Repository. – 2016. – Режим доступу: https://github.com/mxgmn/WaveFunctionCollapse (дата звернення: 15.10.2025). – Назва з екрана.

19. Generating Worlds With Wave Function Collapse [Електронний ресурс] // PROCJAM. – 2024. – Режим доступу: https://procjam.com/wfc (дата звернення: 16.10.2025). – Назва з екрана.

20. Generating Game Levels Using Wave Function Collapse [Електронний ресурс] // Wolfram Blog. – 2024. – Режим доступу: https://blog.wolfram.com/wfc-generation (дата звернення: 17.10.2025). – Назва з екрана.

21. Procedural Map Generation with Binary Space Partitioning [Електронний ресурс] // MaxGCoding. – 2025. – Режим доступу: https://maxgcoding.com/bsp-map-generation (дата звернення: 18.10.2025). – Назва з екрана.

22. Capabilities of the Open-Source Godot Engine in Game Development Compared to Unity and Unreal Engine [Електронний ресурс] // Trepo Digital Library. – 2025. – Режим доступу: https://trepo.tuni.fi/handle/10024/168932 (дата звернення: 19.10.2025). – Назва з екрана.

23. Choosing a Game Engine: Unity vs Unreal Engine vs Godot [Електронний ресурс] // CyberGlads. – 2024. – Режим доступу: https://cyberglads.com/game-engine-comparison (дата звернення: 20.10.2025). – Назва з екрана.

24. Godot vs Unity vs Unreal: Which One To Choose? [Електронний ресурс] // Rocketbrush Studio. – 2025. – Режим доступу: https://rocketbrush.com/blog/godot-vs-unreal-vs-unity (дата звернення: 21.10.2025). – Назва з екрана.

25. Top Unreal Engine Features for 2025 Game Development [Електронний ресурс] // N-iX. – 2025. – Режим доступу: https://www.n-ix.com/unreal-engine-2025-features (дата звернення: 22.10.2025). – Назва з екрана.

26. Procedural Generation in Unreal Engine: Tools and Techniques [Електронний ресурс] // SDLCCorp Blog. – 2025. – Режим доступу: https://sdlccorp.com/unreal-engine-procedural-tools (дата звернення: 23.10.2025). – Назва з екрана.

27. PCG Enhancements in Unreal Engine 5.5 [Електронний ресурс] // Stackademic. – 2024. – Режим доступу: https://stackademic.com/unreal-5-5-pcg (дата звернення: 24.10.2025). – Назва з екрана.

28. How to Use Unreal Engine 5’s New Procedural Tools [Електронний ресурс] // Creative Bloq. – 2024. – Режим доступу: https://www.creativebloq.com/unreal-pcg-tools (дата звернення: 25.10.2025). – Назва з екрана.

29. A Look at Unreal Engine Procedural Generation of Content [Електронний ресурс] // Interactive Immersive. – 2025. – Режим доступу: https://interactiveimmersive.io/unreal-pcg-overview (дата звернення: 26.10.2025). – Назва з екрана.

30. C++: The Core Language Powering Unreal Engine’s Capabilities [Електронний ресурс] // Unimedia.dev. – 2024. – Режим доступу: https://unimedia.dev/unreal-cpp-core (дата звернення: 27.10.2025). – Назва з екрана.

31. Unreal Engine Blueprints vs C++ Performance [Електронний ресурс] // SpongeHammer. – 2025. – Режим доступу: https://spongehammer.com/blueprints-vs-cpp (дата звернення: 28.10.2025). – Назва з екрана.

32. C++ vs Blueprints in Unreal Engine – 2025 Guide [Електронний ресурс] // Visual Assist Blog. – 2025. – Режим доступу: https://www.wholetomato.com/blog/unreal-cpp-vs-blueprint (дата звернення: 29.10.2025). – Назва з екрана.

33. Blueprints vs C++ in Unreal Engine [Електронний ресурс] // Codefinity. – 2024. – Режим доступу: https://codefinity.com/unreal-blueprints-vs-cpp (дата звернення: 30.10.2025). – Назва з екрана.

34. Why I Use C++ for Gameplay and Blueprints for UI [Електронний ресурс] // Reddit / r/unrealengine. – 2025. – Режим доступу: https://reddit.com/r/unrealengine/comments/cpp_blueprint (дата звернення: 31.10.2025). – Назва з екрана.

35. Software Architecture Requirements for Procedural Content Generation [Електронний ресурс] // SBC Proceedings. – 2022. – Режим доступу: https://sbc-proceedings.org/pcg-architecture (дата звернення: 01.11.2025). – Назва з екрана.

ДОДАТОК А
Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

проф., д.т.н. Азаров О. Д.

«03» жовтня 2025р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

«Метод процедурної генерації рівнів з використанням методу двійкового розбиття простору»

Науковий керівник: к.т.н., доц. доцент каф. ОТ

_____ Колесник. І.С.

Студент групи 2КІ-24м

_____ Сирдій Д.П.

ВНТУ 2025

1 Підстава для виконання магістерської кваліфікаційної роботи

1.1 Актуальність теми зумовлена необхідністю інтеграції таких систем у різні сфери життя, від промисловості до побуту. З розвитком технологій та смарт-систем, якісне апаратне забезпечення стає критично важливим для забезпечення роботи комплексних систем, що збирають та обробляють великі обсяги даних. Інновації в дані області дозволяють створювати більш точні та ефективні системи, що можуть працювати в різноманітних умовах та забезпечувати високу якість результатів.

1.2 Наказ про затвердження теми МКР.

2 Мета і призначення МКР

2.1 Метою роботи є створення системи розпізнавання об'єктів на місцевості з підвищеною функціональністю.

2.2 Дана система призначений для виводу відео і аудіо інформації. Передача інформації відбувається за допомогою комп’ютерної мережі.

3 Вихідні дані для виконання МКР

3.1 Наявність як мінімум одної камери і мікрофону;

3.2 Можливість виведення відео і аудіоінформації в локальну мережу і в інтернет;

3.3 Забезпечення достатньої якості для роботи програми по визначенню об'єктів на місцевості.

4 Вимоги до виконання МКР

Робота повинна бути виконана відповідно до затвердженої теми та наказу про затвердження теми МКР. Всі етапи виконання роботи повинні бути документально оформлені та узгоджені з керівником. Необхідно провести аналіз існуючих рішень та аналогів систем відео і аудіонагляду. Робота повиненна включати розробку системи відео і аудіо спостереження, який використовує

мікропроцесорну систему.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в таблиці А.1

Таблиця А.1 — Етапи МКР

№	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	2	3	4	5
1	Вибір, узгодження та затвердження теми МКР	25.09.2025	30.09.2025	
2	Аналіз сучасних методів процедурної генерації рівнів	01.10.2025	04.10.2025	
3	Обґрунтування вибору технологічної бази	05.10.2025	11.10.2025	
4	Розробка архітектури системи процедурної генерації	12.10.2025	17.10.2025	
5	Розробка BSP-алгоритм для поділу простору рівня	18.10.2025	22.10.2025	
6	Реалізація основних модулів системи у середовищі Unreal Engine 5	23.10.2025	28.10.2025	
7	Оформлення роботи у вигляді пояснювальної записки та презентації	29.10.2025	04.11.2025	
8	Перевірка виконаної роботи	05.11.2025	07.11.2025	
9	Остаточні виправлення, збирання підписів та зшивання роботи	08.11.2025	10.11.2025	
10	Попередній захист проєкту та його рецензування	11.11.2025	11.11.2025	
11	Захист МКР	16.12.2025	22.12.2025	

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, анотації до МКР українською та іноземною мовами, довідка про відповідність оформлення МКР діючим вимогам.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлення та порядок виконання МКР

При оформлюванні МКР використовуються:

— ДСТУ 3008 : 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302 : 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

— документами на які посилаються у вище вказаних

ДОДАТОК Б

Протокол перевірки кваліфікаційної роботи

Назва роботи: Метод процедурної генерації рівнів з використанням методу двійкового розбиття простору.

Тип роботи: _____ магістерська кваліфікаційна робота _____
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ: _____кафедра обчислювальної техніки_____
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) __1____ %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

☐ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

Азаров О.Д., д.т.н., проф., зав. кафедри ОТ _ _____
(прізвище, ініціали, посада) (підпис)

Мартинюк Т. Б., д.т.н., проф. кафедри ОТ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____Захарченко С. М., проф. каф. ОТ
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____Колесник. І.С., доц. каф. ОТ____
(підпис) (прізвище, ініціали, посада)

Здобувач _____Сирдій Д.П._____
(підпис) (прізвище, ініціали)

ДОДАТОК В

Діаграма методу процедурної генерації рівнів

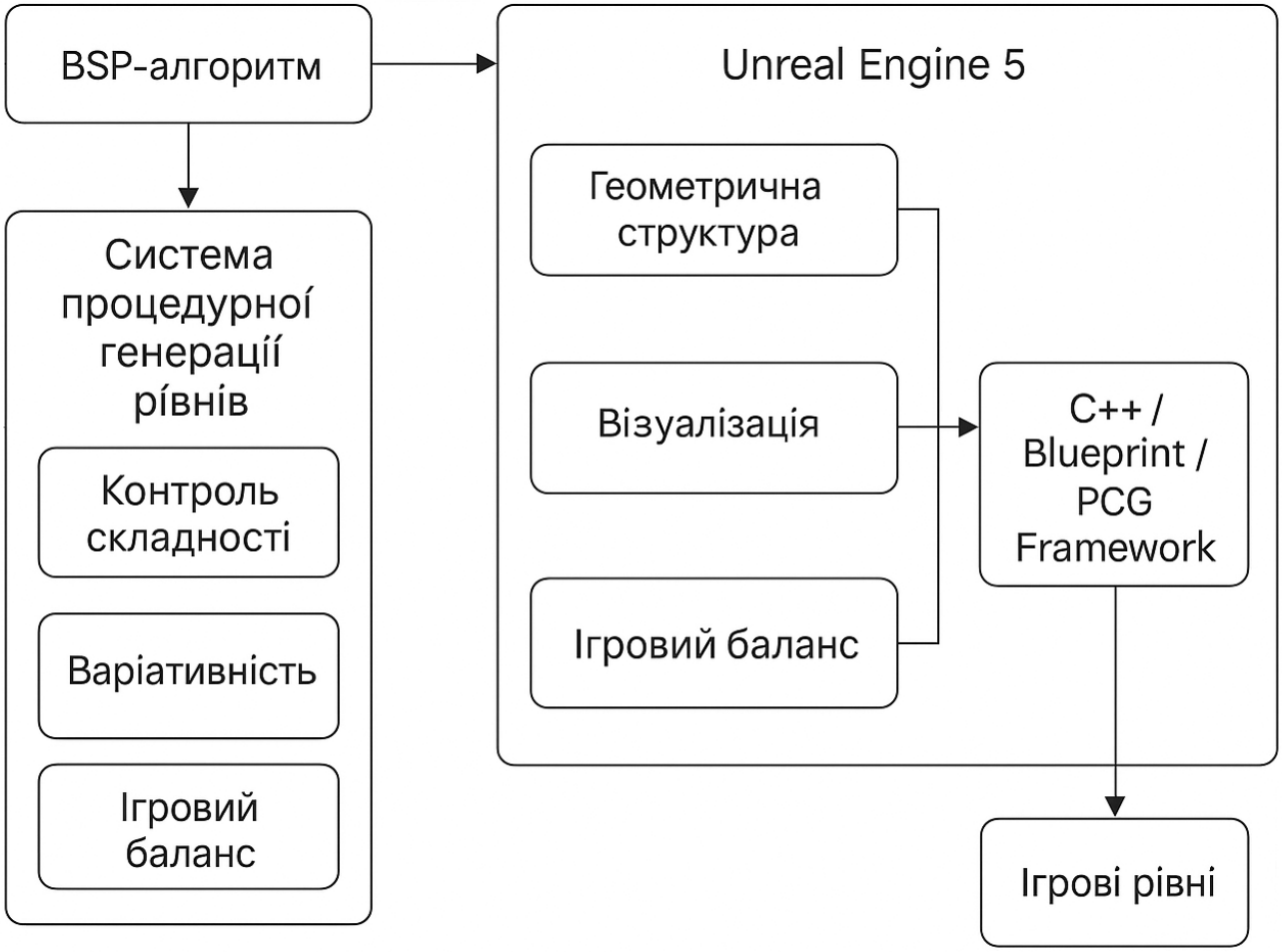

Рисунок В.1 — Діаграма методу процедурної генерації рівнів

ДОДАТОК Г

Лістинг методу RoomNode.h

```
#pragma once
#include "CoreMinimal.h"
#include "RoomNode.generated.h"

USTRUCT(BlueprintType)
struct FBSPRoom
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	FIntRect Bounds;
};

USTRUCT(BlueprintType)
struct FBSPCorridor
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	FIntRect Bounds;
};

USTRUCT(BlueprintType)
struct FBSPParams
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 Seed = 1337;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 MapWidth = 512;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 MapHeight = 512;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 MinLeafSize = 64;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 MaxDepth = 6;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP",
meta=(ClampMin="0.0", ClampMax="1.0"))
	float SplitChance = 0.85f;
```

```
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP",
meta=(ClampMin="0.1", ClampMax="0.9"))
	float SplitRatio = 0.5f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 RoomPadding = 4;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 MinRoomSize = 16;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 RoomJitter = 8;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 CorridorWidth = 3;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	bool PruneCorridorOverlap = true;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	bool ConnectNearestRooms = false;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	bool EnsureStartEnd = true;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 LimitCorridorSteps = 0;
};

USTRUCT(BlueprintType)
struct FRoomNode
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	FIntRect Rect;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 Depth = 0;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 Left = INDEX_NONE;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="BSP")
	int32 Right = INDEX_NONE;

	FORCEINLINE bool IsLeaf() const { return Left == INDEX_NONE && Right ==
INDEX_NONE; }
};
```

ДОДАТОК Д

Лістинг методу CorridorBuilder.cpp

```
#include "CorridorBuilder.h"
#include "RoomNode.h"
#include "Algo/Sort.h"

static bool IntersectsOrTouches(const FIntRect& A, const FIntRect& B)
{
	return !(A.Max.X <= B.Min.X || B.Max.X <= A.Min.X || A.Max.Y <= B.Min.Y || B.Max.Y <= A.Min.Y);
}

static FIntRect Intersection(const FIntRect& A, const FIntRect& B)
{
	const int32 MinX = FMath::Max(A.Min.X, B.Min.X);
	const int32 MinY = FMath::Max(A.Min.Y, B.Min.Y);
	const int32 MaxX = FMath::Min(A.Max.X, B.Max.X);
	const int32 MaxY = FMath::Min(A.Max.Y, B.Max.Y);
	return (MaxX > MinX && MaxY > MinY) ? FIntRect(FIntPoint(MinX, MinY), FIntPoint(MaxX, MaxY)) : FIntRect();
}

static void ExpandRect(FIntRect& R, int32 Dx, int32 Dy)
{
	R.Min.X -= Dx; R.Max.X += Dx;
	R.Min.Y -= Dy; R.Max.Y += Dy;
}

static void RectToCells(const FIntRect& R, int32 TileSize, TArray<FIntPoint>& Out)
{
	const int32 MinX = R.Min.X / TileSize;
	const int32 MinY = R.Min.Y / TileSize;
	const int32 MaxX = (R.Max.X - 1) / TileSize;
```

```
	const int32 MaxY = (R.Max.Y - 1) / TileSize;
	for (int32 y = MinY; y <= MaxY; ++y)
	{
		for (int32 x = MinX; x <= MaxX; ++x)
		{
			Out.Emplace(x, y);
		}
	}
}

static FIntRect MakeDoorFromOverlap(const FIntRect& Overlap, int32 DoorWidth, int32 DoorDepth)
{
	if (Overlap.Width() >= Overlap.Height())
	{
		const int32 MidY = (Overlap.Min.Y + Overlap.Max.Y) / 2;
		const int32 HalfH = FMath::Max(1, DoorDepth / 2);
		const int32 HalfW = FMath::Max(1, DoorWidth / 2);
		const int32 MidX = (Overlap.Min.X + Overlap.Max.X) / 2;
		const int32 MinX = MidX - HalfW;
		const int32 MaxX = MidX + HalfW;
		return FIntRect(FIntPoint(MinX, MidY - HalfH), FIntPoint(MaxX, MidY + HalfH));
	}
	else
	{
		const int32 MidX = (Overlap.Min.X + Overlap.Max.X) / 2;
		const int32 HalfW = FMath::Max(1, DoorDepth / 2);
		const int32 HalfH = FMath::Max(1, DoorWidth / 2);
		const int32 MidY = (Overlap.Min.Y + Overlap.Max.Y) / 2;
		const int32 MinY = MidY - HalfH;
		const int32 MaxY = MidY + HalfH;
		return FIntRect(FIntPoint(MidX - HalfW, MinY), FIntPoint(MidX + HalfW, MaxY));
	}
}
```

```
static void SnapRect(FIntRect& R, int32 Grid)
{
	R.Min.X = FMath::RoundToInt(float(R.Min.X) / Grid) * Grid;
	R.Min.Y = FMath::RoundToInt(float(R.Min.Y) / Grid) * Grid;
	R.Max.X = FMath::RoundToInt(float(R.Max.X) / Grid) * Grid;
	R.Max.Y = FMath::RoundToInt(float(R.Max.Y) / Grid) * Grid;
}

void UCorridorBuilder::Build(const TArray<FBSPRoom>& Rooms, const TArray<FBSPCorridor>&
Corridors, const FCorridorBuildParams& Params, FCorridorBuildResult& Out)
{
	Out.CorridorRects.Reset();
	Out.Doors.Reset();
	Out.PathCells.Reset();

	TArray<FIntRect> WorkCorridors;
	WorkCorridors.Reserve(Corridors.Num());
	for (const FBSPCorridor& C : Corridors) WorkCorridors.Add(C.Bounds);

	for (FIntRect& C : WorkCorridors)
	{
		if (Params.bSnapToGrid) SnapRect(C, Params.TileSize);
		ExpandRect(C, Params.ExtraThickness, Params.ExtraThickness);
	}

	for (const FIntRect& C : WorkCorridors)
	{
		bool bHidden = false;
		for (const FBSPRoom& R : Rooms)
		{
			if (R.Bounds.Contains(C.Min) && R.Bounds.Contains(C.Max)) { bHidden =
true; break; }
		}
```

```
            if (!bHidden) Out.CorridorRects.Add(C);
        }

        for (const FIntRect& C : Out.CorridorRects)
        {
            RectToCells(C, Params.TileSize, Out.PathCells);
        }

        for (const FBSPRoom& R : Rooms)
        {
            for (const FIntRect& C : Out.CorridorRects)
            {
                if (!IntersectsOrTouches(R.Bounds, C)) continue;
                const FIntRect Overlap = Intersection(R.Bounds, C);
                if (Overlap.Area() <= 0) continue;
                FIntRect Door = MakeDoorFromOverlap(Overlap, Params.DoorWidth,
Params.DoorDepth);
                if (Params.bSnapToGrid) SnapRect(Door, Params.TileSize);
                Out.Doors.Add(Door);
            }
        }

        if (Params.bMergeCorridors)
        {
            TArray<FIntRect> Merged;
            Merged.Reserve(Out.CorridorRects.Num());
            TArray<bool> Used; Used.Init(false, Out.CorridorRects.Num());

            for (int32 i = 0; i < Out.CorridorRects.Num(); ++i)
            {
                if (Used[i]) continue;
                FIntRect Acc = Out.CorridorRects[i];
                for (int32 j = i + 1; j < Out.CorridorRects.Num(); ++j)
```

```
                {
                    if (Used[j]) continue;
                    const FIntRect& B = Out.CorridorRects[j];
                    const bool bSameRow = (Acc.Min.Y == B.Min.Y && Acc.Max.Y == B.Max.Y) && (Acc.Max.X >= B.Min.X && B.Max.X >= Acc.Min.X);
                    const bool bSameCol = (Acc.Min.X == B.Min.X && Acc.Max.X == B.Max.X) && (Acc.Max.Y >= B.Min.Y && B.Max.Y >= Acc.Min.Y);
                    if (bSameRow)
                    {
                        Acc.Min.X = FMath::Min(Acc.Min.X, B.Min.X);
                        Acc.Max.X = FMath::Max(Acc.Max.X, B.Max.X);
                        Used[j] = true;
                    }
                    else if (bSameCol)
                    {
                        Acc.Min.Y = FMath::Min(Acc.Min.Y, B.Min.Y);
                        Acc.Max.Y = FMath::Max(Acc.Max.Y, B.Max.Y);
                        Used[j] = true;
                    }
                }
                Merged.Add(Acc);
            }
            Out.CorridorRects = MoveTemp(Merged);
        }

        if (Params.bUniqueCells)
        {
            Out.PathCells.Sort([](const FIntPoint& A, const FIntPoint& B)
            {
                return (A.Y == B.Y) ? A.X < B.X : A.Y < B.Y;
            });
            int32 W = 0;
            for (int32 i = 1; i < Out.PathCells.Num(); ++i)
            {
```

```
                        if (Out.PathCells[i] != Out.PathCells[W])
                        {
                                ++W;
                                Out.PathCells[W] = Out.PathCells[i];
                        }
                }
                if (Out.PathCells.Num() > 0)
Out.PathCells.SetNum(Out.PathCells.IndexOfByKey(Out.PathCells.Last()) + 1);
        }

        if (Params.MaxDoorCount > 0 && Out.Doors.Num() > Params.MaxDoorCount)
        {
                Algo::Sort(Out.Doors, [](const FIntRect& A, const FIntRect& B)
                {
                        return A.Area() > B.Area();
                });
                Out.Doors.SetNum(Params.MaxDoorCount);
        }
}
```

ДОДАТОК Е

Лістинг файлу BSPGenerator.cpp

```
#include "BSPGenerator.h"
#include "Containers/Queue.h"
#include "Algo/Sort.h"
static void SplitAxis(const FIntRect& Rect, int32 MinSize, float SplitRatio, FRandomStream& Rng,
FIntRect& A, FIntRect& B)
{
	const bool bSplitHoriz = (Rect.Width() > Rect.Height())
		? true
		: (Rect.Width() < Rect.Height() ? false : Rng.FRand() > 0.5f);

	if (bSplitHoriz)
	{		const int32 MinCut = Rect.Min.X + MinSize;
		const int32 MaxCut = Rect.Max.X - MinSize;
		if (MaxCut <= MinCut) { A = Rect; B = FIntRect(); return; }
		const int32 Ideal = Rect.Min.X + FMath::Clamp(int32(Rect.Width() * SplitRatio),
MinSize, Rect.Width() - MinSize);
		const int32 Cut = FMath::Clamp(Ideal + Rng.RandRange(-Rect.Width() / 10,
Rect.Width() / 10), MinCut, MaxCut);
		A = FIntRect(Rect.Min, FIntPoint(Cut, Rect.Max.Y));
		B = FIntRect(FIntPoint(Cut, Rect.Min.Y), Rect.Max);
	}
	else
	{		const int32 MinCut = Rect.Min.Y + MinSize;
		const int32 MaxCut = Rect.Max.Y - MinSize;
		if (MaxCut <= MinCut) { A = Rect; B = FIntRect(); return; }
		const int32 Ideal = Rect.Min.Y + FMath::Clamp(int32(Rect.Height() * SplitRatio),
MinSize, Rect.Height() - MinSize);
		const int32 Cut = FMath::Clamp(Ideal + Rng.RandRange(-Rect.Height() / 10,
Rect.Height() / 10), MinCut, MaxCut);
		A = FIntRect(Rect.Min, FIntPoint(Rect.Max.X, Cut));
		B = FIntRect(FIntPoint(Rect.Min.X, Cut), Rect.Max);
	}
}

static FIntRect ShrinkRect(const FIntRect& R, int32 Padding, FRandomStream& Rng)
{
	const int32 px = FMath::Clamp(Padding + Rng.RandRange(0, Padding), 0, R.Width() / 3);
	const int32 py = FMath::Clamp(Padding + Rng.RandRange(0, Padding), 0, R.Height() / 3);
	const FIntPoint Min(R.Min.X + px, R.Min.Y + py);
	const FIntPoint Max(R.Max.X - px, R.Max.Y - py);
	return (Max.X > Min.X + 1 && Max.Y > Min.Y + 1) ? FIntRect(Min, Max) : R;
}
static FVector2D RectCenter(const FIntRect& R)
{
	return FVector2D((R.Min.X + R.Max.X) * 0.5f, (R.Min.Y + R.Max.Y) * 0.5f);
}
```

```
static void CorridorFromTo(const FVector2D& A, const FVector2D& B, int32 Width,
TArray<FBSPCorridor>& Out)
{
	const int32 wx = FMath::Max(1, Width);
	const int32 wy = FMath::Max(1, Width);
	const int32 Ax = FMath::RoundToInt(A.X);
	const int32 Ay = FMath::RoundToInt(A.Y);
	const int32 Bx = FMath::RoundToInt(B.X);
	const int32 By = FMath::RoundToInt(B.Y);

	if (FMath::RandBool())
	{
		const FIntRect H(FIntPoint(FMath::Min(Ax, Bx), Ay - wy), FIntPoint(FMath::Max(Ax,
Bx), Ay + wy));
		const FIntRect V(FIntPoint(Bx - wx, FMath::Min(Ay, By)), FIntPoint(Bx + wx,
FMath::Max(Ay, By)));
		Out.Emplace(H);
		Out.Emplace(V);
	}
	else
	{
		const FIntRect V(FIntPoint(Ax - wx, FMath::Min(Ay, By)), FIntPoint(Ax + wx,
FMath::Max(Ay, By)));
		const FIntRect H(FIntPoint(FMath::Min(Ax, Bx), By - wy), FIntPoint(FMath::Max(Ax,
Bx), By + wy));
		Out.Emplace(V);
		Out.Emplace(H);
	}
}

static void ConnectLeafCenters(const TArray<FIntRect>& Leaves, int32 CorridorWidth,
TArray<FBSPCorridor>& Out)
{
	if (Leaves.Num() <= 1) return;
	TArray<FVector2D> Centers;
	Centers.Reserve(Leaves.Num());
	for (const FIntRect& L : Leaves) Centers.Add(RectCenter(L));

	TArray<int32> Order;
	for (int32 i = 0; i < Centers.Num(); ++i) Order.Add(i);
	Algo::Sort(Order, [&Centers](int32 A, int32 B){ return Centers[A].X < Centers[B].X; });

	for (int32 i = 0; i < Order.Num() - 1; ++i)
	{
		const FVector2D A = Centers[Order[i]];
		const FVector2D B = Centers[Order[i + 1]];
		CorridorFromTo(A, B, CorridorWidth, Out);
	}
}

void UBSPGenerator::Generate(const FBSPParams& Params, TArray<FBSPRoom>& OutRooms,
TArray<FBSPCorridor>& OutCorridors)
```

```
{
	OutRooms.Reset();
	OutCorridors.Reset();

	FRandomStream Rng(Params.Seed);
	const FIntRect Root(FIntPoint(0, 0), FIntPoint(FMath::Max(Params.MapWidth,
Params.MinLeafSize * 2), FMath::Max(Params.MapHeight, Params.MinLeafSize * 2)));

	struct FNode { FIntRect Rect; int32 Depth; };
	TQueue<FNode> Queue;
	TArray<FIntRect> Leaves;
	Queue.Enqueue({ Root, 0 });

	while (!Queue.IsEmpty())
	{
		FNode N;
		Queue.Dequeue(N);

		const bool bCanSplitW = N.Rect.Width() >= Params.MinLeafSize * 2;
		const bool bCanSplitH = N.Rect.Height() >= Params.MinLeafSize * 2;
		const bool bUnderDepth = N.Depth < Params.MaxDepth;
		const bool bTrySplit = bUnderDepth && (bCanSplitW || bCanSplitH) &&
(Rng.FRand() <= Params.SplitChance);

		if (bTrySplit)
		{
			FIntRect A, B;
			SplitAxis(N.Rect, Params.MinLeafSize, Params.SplitRatio, Rng, A, B);
			if (B.Area() > 0 && A.Area() > 0)
			{
				Queue.Enqueue({ A, N.Depth + 1 });
				Queue.Enqueue({ B, N.Depth + 1 });
				continue;
			}
		}

		Leaves.Add(N.Rect);
	}

	for (const FIntRect& L : Leaves)
	{
		FIntRect R = ShrinkRect(L, Params.RoomPadding, Rng);
		const int32 MinW = FMath::Max(Params.MinRoomSize, 2);
		const int32 MinH = FMath::Max(Params.MinRoomSize, 2);
		if (R.Width() < MinW || R.Height() < MinH) continue;

		const int32 W = FMath::Clamp(R.Width() - Rng.RandRange(0, Params.RoomJitter),
MinW, R.Width());
		const int32 H = FMath::Clamp(R.Height() - Rng.RandRange(0, Params.RoomJitter),
MinH, R.Height());
		const int32 X = R.Min.X + Rng.RandRange(0, FMath::Max(0, R.Width() - W));
		const int32 Y = R.Min.Y + Rng.RandRange(0, FMath::Max(0, R.Height() - H));
```

```
		const FIntRect RoomRect(FIntPoint(X, Y), FIntPoint(X + W, Y + H));

		FBSPRoom Room;
		Room.Bounds = RoomRect;
		OutRooms.Add(Room);
	}

	ConnectLeafCenters(Leaves, Params.CorridorWidth, OutCorridors);

	if (Params.PruneCorridorOverlap)
	{
		TArray<FBSPCorridor> Filtered;
		Filtered.Reserve(OutCorridors.Num());
		for (const FBSPCorridor& C : OutCorridors)
		{
			bool bCovered = false;
			for (const FBSPRoom& R : OutRooms)
			{
				if (R.Bounds.Contains(C.Bounds.Min) &&
R.Bounds.Contains(C.Bounds.Max)) { bCovered = true; break; }
			}
			if (!bCovered) Filtered.Add(C);
		}
		OutCorridors = MoveTemp(Filtered);
	}

	if (Params.ConnectNearestRooms)
	{
		TArray<FVector2D> Centers;
		for (const FBSPRoom& R : OutRooms) Centers.Add(RectCenter(R.Bounds));
		for (int32 i = 0; i < Centers.Num(); ++i)
		{
			int32 Nearest = INDEX_NONE;
			float Best = MAX_flt;
			for (int32 j = 0; j < Centers.Num(); ++j)
			{
				if (i == j) continue;
				const float D = FVector2D::Distance(Centers[i], Centers[j]);
				if (D < Best) { Best = D; Nearest = j; }
			}
			if (Nearest != INDEX_NONE) CorridorFromTo(Centers[i], Centers[Nearest],
Params.CorridorWidth, OutCorridors);
		}
	}
	if (Params.EnsureStartEnd)
	{		if (OutRooms.Num() >= 2)		{
			int32 MinIdx = 0, MaxIdx = 0;
			for (int32 i = 1; i < OutRooms.Num(); ++i)
			{
				if (OutRooms[i].Bounds.Min.X + OutRooms[i].Bounds.Min.Y <
OutRooms[MinIdx].Bounds.Min.X + OutRooms[MinIdx].Bounds.Min.Y) MinIdx = i;
```

```
                                if (OutRooms[i].Bounds.Max.X + OutRooms[i].Bounds.Max.Y >
OutRooms[MaxIdx].Bounds.Max.X + OutRooms[MaxIdx].Bounds.Max.Y) MaxIdx = i;
                        }
                        const FVector2D A = RectCenter(OutRooms[MinIdx].Bounds);
                        const FVector2D B = RectCenter(OutRooms[MaxIdx].Bounds);
                        CorridorFromTo(A, B, Params.CorridorWidth, OutCorridors);
                }
        }

        if (Params.LimitCorridorSteps > 0)
        {
                TArray<FBSPCorridor> Trimmed;
                Trimmed.Reserve(OutCorridors.Num());
                int32 Steps = 0;
                for (const FBSPCorridor& C : OutCorridors)
                {
                        if (Steps >= Params.LimitCorridorSteps) break;
                        Trimmed.Add(C);
                        ++Steps;
                }
                OutCorridors = MoveTemp(Trimmed);
        }
}
```