МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

ЛАЙТАРУК ІВАН ФЕДОРОВИЧ

ДОСЛІДЖЕННЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СВІТУ НА ОСНОВІ РОЗРОБКИ КОМП'ЮТЕРНОЇ ГРИ З АНАЛІЗОМ ПОВЕДІНКИ ГРАВЦЯ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
Гришанович Тетяна Олександрівна,
кандидат фізико-математичних наук,
доцент кафедри комп'ютерних наук
та кібербезпеки

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №__
засідання кафедри комп'ютерних
наук та кібербезпеки
від _____ 2025 р.
Завідувач кафедри

(_____) Гришанович Т. О.

ЛУЦЬК – 2025

ЗМІСТ

ВСТУП

Актуальність теми. Сьогодні гравців комп'ютерних ігор важко здивувати одноманітністю віртуального світу, тому розробники часто вдаються до використання алгоритмів процедурної генерації. Процедурна генерація світу передбачає використання стохастичних алгоритмів для створення ігрового середовища, яке не повторюється навіть при однакових вхідних даних. Це доцільний та актуальний підхід, якщо необхідність у великій кількості унікального контенту більша, ніж необхідність у добре продуманому, але створеному від руки світі. Якщо правильно розробити модель генерації, то проблем із непередбачуваністю та несумісністю під час проходження гри можна уникнути.

Існує велика кількість методів та підходів для реалізації випадкових світів – від алгоритмів шумів до різних класів еволюційних алгоритмів. Більшість з них походять від алгоритмів для комп'ютерної графіки – шуми для генерації карт висот, біомів, плавних змін у відкритому світі, діаграма Вороного для розподілення світу на регіони, BSP дерево для генерації кімнат і коридорів. Інші методи були розроблені на основі біологічних чи математичних моделей та легко інтегруються в контекст процедурної генерації – клітинні автомати для генерації печер чи поселень, переписування графу (graph grammar) для 2D та 3D моделювання, генетичні алгоритми для найкращого вибору елементів генерації залежно від вхідних параметрів. У реальних проєктах використовують комбінації та модифікації цих методів, вибудовуючи складну та комплексну модель генерації світу.

Використання процедурної генерації світу разом із методами аналізу даних (поведінки гравця) є маловивченим розділом, оскільки традиційно процедурна генерація розвивалася для створення статичних та повторюваних шаблонів. Щоб аналізувати поведінку гравця потрібні репрезентативні дані, що особливо критично для синглплеєрних ігор, де немає великого пулу гравців для навчання моделей. Однак результати таких досліджень можуть дозволити

розробити моделі, які глибоко інтегрують процедурну генерацію з динамічними моделями гравця, та генерувати максимально адаптивний до вподобань або відраз гравця світ.

Метою роботи є розробка моделі процедурної генерації світу на основі генетичного алгоритму та BSP дерева для комп'ютерної гри з адаптацією до проаналізованої за допомогою методу нечіткої кластеризації поведінки гравця та проведення аналізу перспектив її покращення.

Для досягнення мети необхідно виконати наступні **завдання**:

- проаналізувати існуючі методи процедурної генерації контенту в ігрових середовищах, включно з алгоритмами BSP дерева та генетичними алгоритмами;
- дослідити методи нечіткої кластеризації даних про поведінку гравця, зокрема Fuzzy C-Means Clustering, а також врахувати психотипи гравців і принцип «Exploration–Exploitation» для інтерпретації поведінки;
- обрати інструментальні засоби та розробити базовий прототип гри;
- реалізувати формування структури будівлі з окремими кімнатами на основі BSP дерева;
- розробити генетичний алгоритм визначення характеристик кімнат на основі аналізу поведінки гравця;
- виконати аналіз поведінки гравця під час проходження гри за допомогою методу нечіткої кластеризації та підібрати характеристики для формування інтерпретованих кластерів;
- інтегрувати результати кластеризації поведінки гравця у генетичний алгоритм для оптимізації характеристик кімнат;
- оцінити перспективи вдосконалення розробленої моделі процедурної генерації світу та подальші напрямки досліджень.

Об'єкт дослідження — алгоритми процедурної генерації світу та методи аналізу даних поведінки гравця.

Предмет дослідження: генерація будівлі для гри за допомогою BSP

дерева та генетичного алгоритму з використанням нечіткої кластеризації Fuzzy C-Means для аналізу поведінки гравця.

Публікації:

1. Laitaruk I. F., Hryshanovych T. O. Overview of modern algorithms for world procedural generation in computer games. Proceedings of the 7th Workshop for Young Scientists in Computer Science & Software Engineering (CS&SE@SW 2024). 2024. P. 152–163. URL: https://cssesw.easyscience.education/cssesw2024/CSSESW2024/paper21.pdf.
2. Лайтарук І. Ф., Куротич А. О., Булатецька Л. В. Операції оновлення ієрархічних структур у моделі вкладених множин. *Прикладні проблеми комп'ютерних наук, безпеки та математики*. 2025. № 4. С. 40–49. URL: https://apcssm.vnu.edu.ua/index.php/Journalone/article/view/132/103.
3. Лайтарук І., Гришанович Т. Огляд перспектив швидкодії алгоритмів процедурної генерації в комп'ютерних іграх. *ІІ Міжнародна науково-практична конференція «Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем»*. 2025. С. 195–198. URL: https://apcssm.vnu.edu.ua/index.php/conf/article/view/164/190.
4. Лайтарук І., Гришанович Т., Онищук О. Порівняння навчання з підкріпленням в ігрових рушіях unreal engine 5 та unity. *Прикладні проблеми комп'ютерних наук, безпеки та математики*. 2025. № 5. С. 4–15. URL: https://apcssm.vnu.edu.ua/index.php/Journalone/article/view/140.

РОЗДІЛ 1.
АЛГОРИТМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СВІТУ. НЕЧІТКА КЛАСТЕРИЗАЦІЯ ДАНИХ ПОВЕДІНКИ ГРАВЦЯ

1.1. Процедурна генерація контенту в ігрових середовищах

Процедурна генерація контенту (procedural content generation, PCG) [1-3] виникла як результат автоматизації процесу створення великої кількості ігрового наповнення, що дало змогу розробникам зосереджуватися не на рутинній ручній роботі, а на технологічній складовій свого продукту. Це дозволяє забезпечити високу реіграбельність для гравця – отримання унікального досвіду під час кожного проходження гри. Окрім того, перевагою PCG є і компактність зберігання даних світу – об'ємні наповнені карти заміняються зберіганням моделей генерації та сіду. Сід (seed) – це вхідний параметр (зазвичай просто ціле число) для будь-якої стохастичної моделі, який забезпечує відтворюваність однакових вихідних даних шляхом генерування одних і тих же псевдовипадкових чисел.

Алгоритми процедурної генерації мають широкий спектр створюваного контенту. Вони включають алгоритми генерації:

- геометрії світу – ландшафти, будівлі, печери (шум Перліна, BSP дерева, клітинні автомати, діаграма Вороного);
- сюжетів та діалогів (ланцюги Маркова, генеративна граматика);
- аудіо та музики (процедурні синтезатори, алгоритмічна композиція).

Такі результати показують, що алгоритми PCG у вихідних даних схожі до моделей штучного інтелекту. Різниця полягає у тому, що процедурна генерація базується заданих алгоритмах (хоч і стохастичних) і правилах, де результат залежить від вхідних параметрів, а генерація за допомогою ШІ використовує моделі, які вчилися на великій кількості даних, де результат є виведенням шаблонів з вже відомих прикладів, а не тільки за фіксованими правилами.

Природа алгоритмів процедурної генерації описує характер отриманих

результатів їхнього використання, але деякі методи є настільки загальними, що важко передбачити, яким алгоритмом були згенеровані ті чи інші дані. Однак різноманіття цих методів дозволяє контролювати ті частини генерації, які потрібно чітко визначити, задати схожі шаблони чи давати алгоритму повну волю для генерації максимально непередбачуваних даних. Тому варто розглянути класичні підходи до процедурної генерації.

Переписування графу (graph rewriting) [4-5] є типом генеративної граматики та дозволяє керувати елементами генерації за допомогою представлення даних у формі графу. Частини графу можуть перетворюватися в інші за чітко заданими правилами. Існує декілька підходів переписування графу, однак на практиці найчастіше використовують алгебраїчний підхід із двома методами логіки заміни підграфів – подвійне виштовхування (double pushout) та одинарне виштовхування (single pushout). Їхня різниця полягає у тому, що подвійне виштовхування використовує інтерфейс (контекст) при перевірці на валідність заміни підграфу. Це дозволяє уникнути конфліктів заміни частин графу, однак одинарне виштовхування описується кращою гнучкістю в кінцевих згенерованих даних.

Градієнтні шуми [6-7] розбивають всю площу генерації на комірки за ґраткою і замість випадкових значень у вузлах цієї ґратки використовують випадковий градієнт, а шум у кожній точці розраховується як інтерполяція скалярних добутків між цими градієнтами та векторами до точки. Існує багато алгоритмів цього класу та модифікацій до них – шум Перліна, симплекс-шум, дробовий броунівський шум та інші. Найчастіше такі методи використовуються для генерації рельєфів, хмар, текстур вогню чи мармуру, оскільки забезпечують доволі природний результат – градієнти допомагають згладити значення шуму в конкретній точці та локалізувати його вплив на сусідні дані.

Діаграма Вороного [8] часто лежить в основі багатьох алгоритмів процедурної генерації будівель, карти біомів чи текстур. Суть полягає в розподіленні простору за випадково заданими точками центрів – усі точки всередині кімнати найближчі до своєї точки розбиття. Для реалізації цього

методу існує два варіанти: попарний перебір усіх точок або більш оптимізований – алгоритм Форчуна [9].

У цій роботі детальніше розглядається декілька конкретних алгоритмів процедурної генерації, а саме алгоритм на основі BSP дерева та генетичний алгоритм.

1.2. Алгоритм на основі BSP дерева

BSP дерево (binary space partitioning) [10-13] – це структура даних, яка представляє бінарне розбиття евклідового простору за допомогою рекурсивного поділу кожної частини на дві необов'язково однакові за розміром. Воно є основоположним у тривимірній комп'ютерній графіці, оскільки долає основні проблеми рендерингу, які виникають при алгоритмі художника чи рейкастингу, та характеризується візуалізацією об'єктів від найближчого до найдальшого й відсутністю перемальовування (перекривання) пікселів, що забезпечує швидкість та ефективність алгоритму.

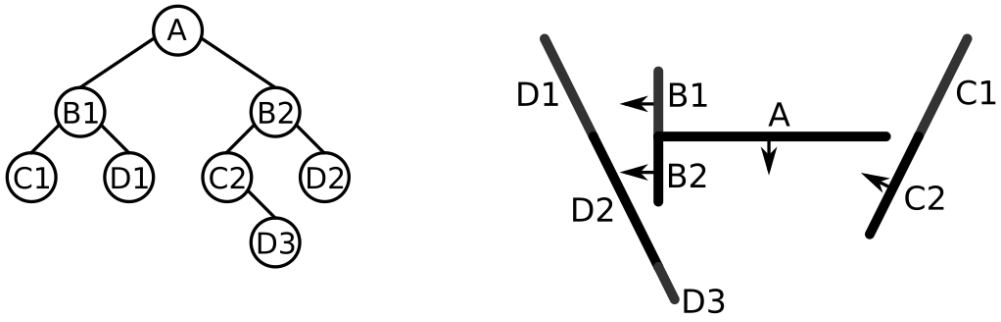

Рис. 1.2.1 – Результат побудови BSP дерева в двовимірному просторі, де полігони – лінії (A, B, C, D). Зліва – побудоване BSP дерево, справа – розподілений простір.

Отже, кроки роботи алгоритму побудови BSP дерева (рис. 1.2.1) це:

1. Вибір полігона P з масиву полігонів.
2. Створення вузла N в дереві та додання полігона P в масив цього вузла.
3. Для кожного іншого полігона в масиві:

3.1. Якщо полігон знаходиться в передній площині відносно полігона P, то перемістити цей полігон в масив вузлів спереду P.

3.2. Якщо полігон знаходиться в задній площині відносно полігона P, то перемістити цей полігон в масив вузлів позаду P.

3.3. Якщо полігон пересікається площиною відносно полігону P, розділити цей полігон і віднести нові полігони до відповідних масивів спереду та позаду P.

3.4. Якщо полігон лежить в площині полігона P, додати його в масив полігонів вузла N.

4. Застосувати ці кроки для полігонів спереду P.
5. Застосувати ці кроки для полігонів позаду P.

При заданій перспективі та обході BSP дерева в ширину визначається порядок рендерингу полігонів. Конкретний алгоритм рендерингу тут не розглядається.

У контексті процедурної генерації BSP дерево використовується для генерування кімнат і коридорів. Дерево генерується в результаті рекурсивного випадкового розбиття простору вертикальними чи горизонтальними лініями. Після чого, кожен листок дерева являє собою простір для генерації геометрії кімнати. Коридори утворюються шляхом злиття двох сусідніх кімнат – лівого та правого вузла. Такий алгоритм створює каркас для згенерованого світу та забезпечує єдиний можливий шлях між усіма кімнатами без циклів.

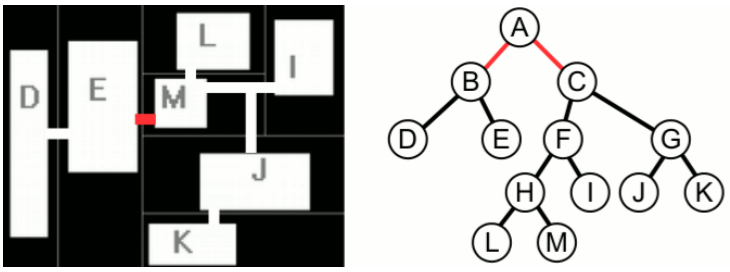

Рис. 1.2.2 – Результат генерації кімнат і коридорів на основі BSP дерева. Остання ітерація злиття просторів під вузлами B та C. Ліворуч – результат згенерованих кімнат і коридорів. Праворуч – згенероване BSP дерево

Алгоритм побудови кімнат та коридорів на основі BSP дерева:

1. Задати весь простір генерації як кореневий вузол дерева.
2. Розділити простір вертикальною або горизонтальною лінією, якщо його площа більша за мінімально допустиму. Задати дочірній лівий та дочірній правий вузли дерева.
3. Рекурсивно повторити крок 2 для дочірнього лівого вузла дерева.
4. Рекурсивно повторити крок 2 для дочірнього правого вузла дерева.
5. Для усіх листків дерева створити кімнату в просторі листка залежно від необхідної геометрії (наприклад, прямокутна кімната – задання верхньої лівої та нижньої правої точок в діапазоні простору листка).
6. Розпочинаючи від листків, для усіх пар дочірніх вузлів дерева створити коридор між лівим та правим вузлом (етап злиття простору).

Бінарне розбиття простору в основному використовується для створення кімнат, що не перекриваються, однак слід зазначити, що ієрархія BSP дерева може бути використана і для інших аспектів генерації. Рисунок 1.2.2 демонструє, як зв'язок кімнат може бути визначеним за допомогою BSP дерева – використання коридорів для з'єднання кімнат, що належать одному батьківському вузлу для забезпечення відсутності перекриття коридорів. Окрім того, не листкові вузли розподілу можуть бути використані для визначення груп кімнат, що відповідають одній темі. Таким способом ці групи кімнат можуть мати єдиний вхід з рештою розподіленого простору карти.

1.3. Генетичний алгоритм

Генетичний алгоритм [14-17] має еволюційний характер та був розроблений на основі того, як відбувається природний відбір. Нащадки отримують свої якості у формі генетичної інформації від батьків і найкращі представники мають більше шансів на виживання та продовження продукування нових нащадків. Отже, після декількох десятків чи сотень поколінь можна помітити зростаючу тенденцію пристосованості все новіших представників

виду.

Зведення механізмів генетичного алгоритму явно вказує на те, що такий підхід застосовується для вирішення задач оптимізації та моделювання, де кожне наступне покоління представляє все більш оптимальний розв'язок. Цей алгоритм часто є узагальним і використовує свої модифікації залежно від необхідності в різних галузях: оптимізації запитів у базах даних, налаштуванні і навчанні штучної нейронної мережі, ігрових стратегіях, задачах біоінформатики.

Незважаючи на те, що кожен етап алгоритму може різнитися в реалізації залежно від потреб, класичний генетичний алгоритм визначає такі етапи (рис. 1.3.1.):

1. Створення початкової популяції всіх можливих рішень (осіб).
2. Обчислення функції допасованості кожної особи.
3. Відбір (селекція) найкращих осіб на основі значень функції допасованості. Відкидання гірших представників популяції.
4. Схрещування осіб.
5. Мутація осіб.
6. Ітераційне повторювання кроків 2-5 до виконання критерія зупинки алгоритму.
7. Вибір найкращого рішення на основі значення функції допасованості.

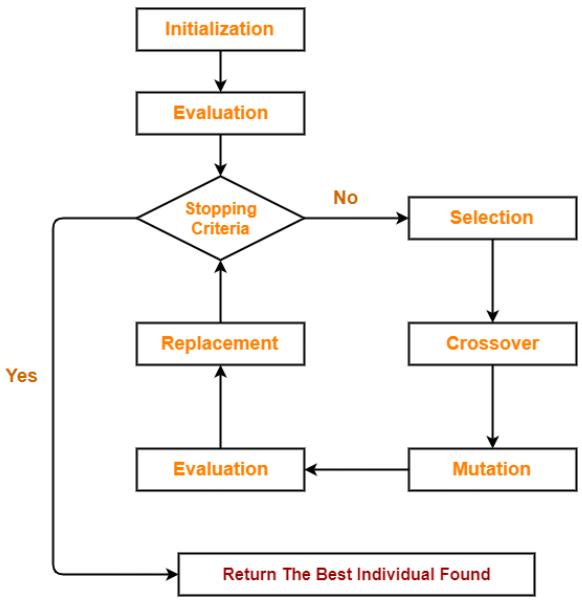

Рис. 1.3.1 – Блок-схема роботи класичного генетичного алгоритму

Генетичний алгоритм містить специфічний понятійно-категоріальний апарат, тому доцільно розглянути кожен етап детальніше.

Створення початкової популяції передбачає задання стартових осіб, які утворені випадковим чином. Кожна особа задається набором хромосом – інформацією про індивідума, зазвичай закодованою в послідовність нулів і одиниць. До початкової популяції немає вимог щодо «якісних» представників. Особи повинні обов'язково відповідати формату рішень, над яким проводиться оптимізація, та якомога різноманітнішими, однак повторюваність індивідів не є перепоною – генетичний алгоритм швидко продукує нові неоднакові особи. Мінімальна кількість індивідів в початковій популяції має бути достатня для репрезентації різноманіття генів, необхідного для вирішення задачі, і зазвичай описується як $N \approx 10 \ldots 20 * L$, де N – це кількість особин в початковій популяції, L – довжина хромосоми. Це дає шанс, що всі важливі комбінації генів з'являться хоча б один раз.

Функція допасованості (fitness function), яку часто називають функцією пристосованості або функцією оцінки, потрібна для розрахунку міри якості індивіда. Вона характеризує те, що насправді оптимізується (мінімізується чи максимізується), тому є цільовою функцією задачі. Для кожної можливої хромосоми функція повинна повертати числове значення, яке явно відображає якість рішення. Її можна визначити як суму ваг кожного гена в хромосомі або як середнє цих ваг, якщо значення не має залежати від довжини хромосоми L. У класичній теорії генетичних алгоритмі функція пристосованості формулюється так, що її значення максимальне для найкращих рішень і мінімальне для найгірших, однак часто в реальних задачах ми хочемо мінімізувати похибку, час чи відстань. У таких випадках доцільно обернути початкову функцію, яку треба мінімізувати $g(x)$ так, що $f_{fitness}(x) = \frac{1}{1+g(x)}$ або задати велике число C, яке більше за максимальне можливе значення $g(x)$ і рахувати функцію допасованості як $f_{fitness}(x) = C - g(x)$.

Селекція визначає індивідів, які перейдуть до наступного покоління. Є різні стратегії відбору індивідів – стратегія елітизму, стратегія рулетки та стратегія *k*-турніру. При стратегії елітизму декілька найкращих індивідів переходять у наступне покоління без змін, що дозволяє весь час залишати найкращі рішення. Стратегія рулетки передбачає, що індивід I буде обраний до наступного покоління з ймовірністю $p_I = \frac{f_I}{\sum_j f_j}$, де f_I – значення функції пристосованості індивіда I, $\sum_j f_j$ – сума значень функцій пристосованості всіх індивідів поточного покоління (рис. 1.3.2). Такий підхід передбачає, що індивіди можуть повторюватися, що дозволить якіснішим рішенням розповсюджуватися швидше. При стратегії *k*-турніру обирається k індивідів та залишається лише декілька з найкращими значеннями функції допасованості. Інколи йому надають перевагу над стратегією рулетки задля уникнення стохастичного шуму.

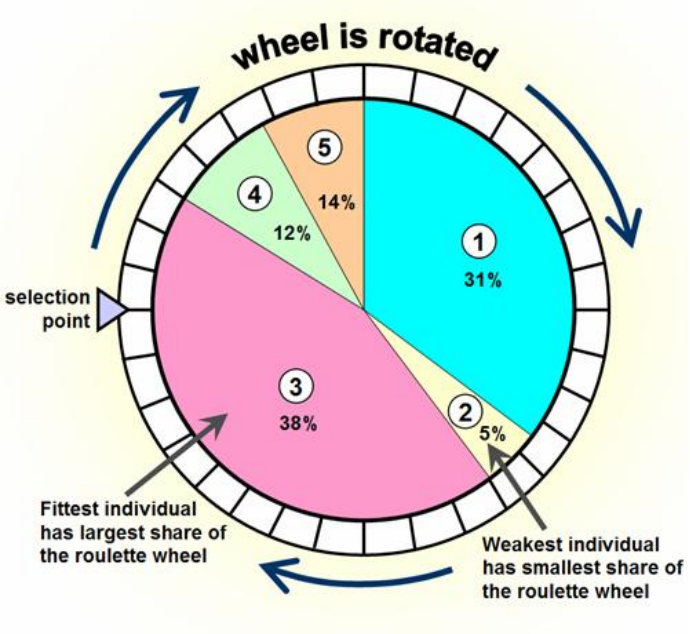

Рис. 1.3.2 – Візуалізація етапу селекції за стратегією рулетки – площа вибору кожного індивіда є їхньою відносною допасованістю. Їхні шанси пройти селекцію пропорційні площі сектора кола

Етап схрещування (кросоверу) відбувається для продукування нового покоління від батьківських індивідів (рис. 1.3.3). Здебільшого схрещування (та й загалом генетичні операції) відбувається на закодованих хромосомах, однак не є

обов’язковим критерієм. Це дозволяє відокремити логіку алгоритму від логіки конкретної задачі. Існує багато алгоритмів реалізації схрещування, серед яких одноточкове схрещування, двоточкове схрещування та рівномірне схрещування. Одноточкове схрещування встановлює одну точку між генами двох хромосом та обмінюється всіма генами після встановленої точки. При двоточковому схрещування вибираються дві точки в двох хромосомах – це дозволяє краще зберігати структуру, ніж одноточкове. Рівномірне схрещування передбачає незалежний вибір гену з пари батьківських генів з певною ймовірністю p. Це дозволяє добре перемішувати гени для продукування різноманітніших структур, однак може зруйнувати гени, які інтерпретуються в конкретні риси тільки в конкретній послідовності.

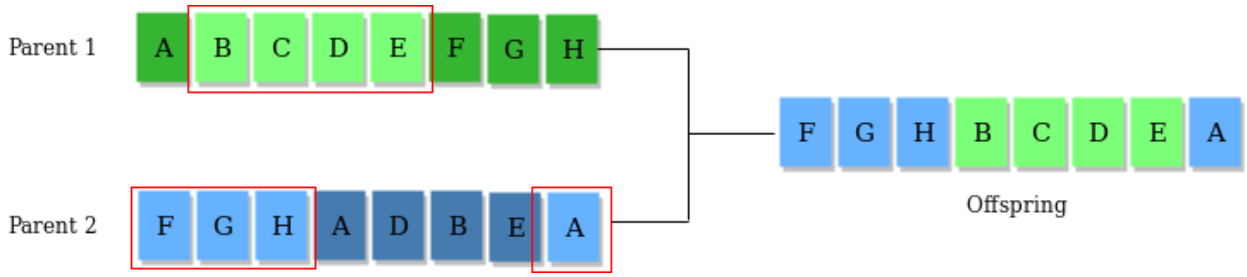

Рис. 1.3.3 – Етап схрещування ГА. Гени для схрещування обрані випадково

Мутація є генетичною операцією, яка відбувається над індивідом окремо (рис. 1.3.4). Існує три види мутації: зростання, відрізання та заміна. Зростання (grow) додає випадковий ген до хромосоми, який не досяг максимальної кількості генів. Відрізання (cut) видаляє випадковий ген з хромосоми. Заміна (alter) характеризується переписуванням параметрів певного гена на випадкові. Частка того, скільки індивідів пройдуть мутацію і скільки схрещування є параметром генетичного алгоритму і може фіксуватися жорстко (наприклад, 70% схрещуються і 30% мутують).

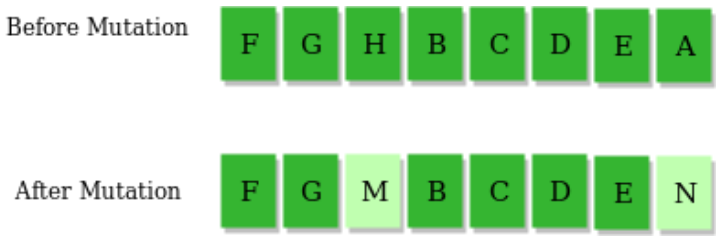

Рис. 1.3.4 – Етап мутації ГА. Випадкові гени замінилася на інші

Критерій зупинки генетичного алгоритму часто описується декількома аспектами: максимальною кількістю поколінь, досяганням певного цільового значення функції допасованості чи відсутності покращення при певній кількості ітерацій. Зазвичай використовується комбінація критеріїв:

$$\text{зупити, якщо}(G_{\text{поточне}} \geq G_{max}) \vee (f_{best} \geq f_{target}) \vee (\text{немає покращення } k \text{ поколінь})$$

Зупинка генетичного алгоритму може відбутися у випадку виродження популяції, що називається передчасною збіжність (premature convergence), однак зазвичай мутація забезпечує уникнути цього.

У генетичному алгоритмі варто розділяти два окремих втілення рішення – генотип і фенотип. Генотип індивіда описується даними (зазвичай закодованими) безпосередньо в його хромосомі і тільки з ними працює алгоритм. Це може бути бінарний вектор (0 і 1), масив дійсних чи цілих чисел або будь-яка інша структура, яка однозначно описує рішення. Фенотип – це реальне втілення рішення, яке декодоване та готове до використання в задачі (наприклад, найоптимальніший маршрут між містами, найкращий тижневий розклад занять чи найбільш адаптований згенерований будинок з кімнатами і коридорами).

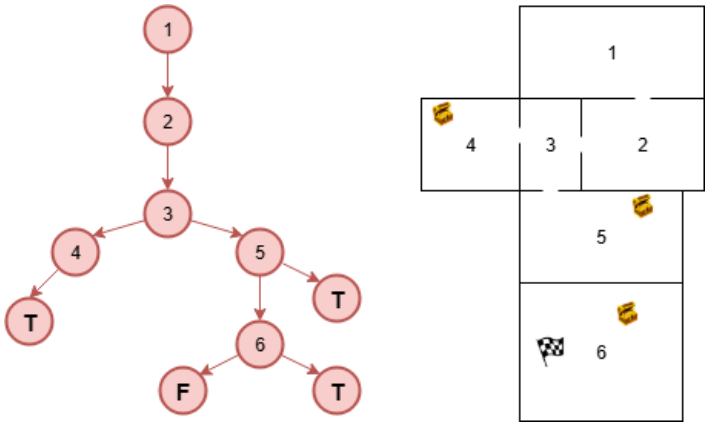

Рис. 1.3.5 – Зліва – деревовидна структура представлення генотипа системи кімнат. Справа – прототип реальної будівлі з виглядом зверху (фенотип)

Оскільки генетичний алгоритм має доволі загальний характер, він знайшов своє застосування і в процедурній генерації. Ним можна підбирати параметри генератора, щоб отримати карти певної складності, оптимізовувати розташування кімнат і скарбів для збалансованого геймплею чи провести еволюцію ландшафту для певного стилю гри. Часто задіюються принципи переписування графів та деревоподібні структури для задання генотипу рішень для процедурної генерації [3] (рис. 1.3.5).

1.4. Метод нечіткої кластеризації Fuzzy C-Means Clustering

Кластерний аналіз дозволяє виділити у великій вибірці окремі групи за допомогою статистичних методів та надати їм значення (інтерпретації). Зазвичай кластеризація розподіляє дані на чіткі підмножини, тобто так, що кожен об’єкт належить тільки одного і ні до якого іншого кластеру. Однак на основі нечітких моделей аналізу даних були розроблені модифікації до класичних алгоритмів кластеризації, що дозволяють описати кожен об’єкт не просто приналежністю до одного кластеру а до всіх певною мірою.

Метод Fuzzy C-Means (далі FCM) [18-19] є модифікацією до класичного методу *k*-середніх (*k*-means) та описує кожне спостереження «м’яко» – вектором

мір приналежності до кожного з c кластерів (рис. 1.4.1). Так само як і в методі k-середніх, кількість кластерів є вхідним параметром у FCM та визначається людиною.

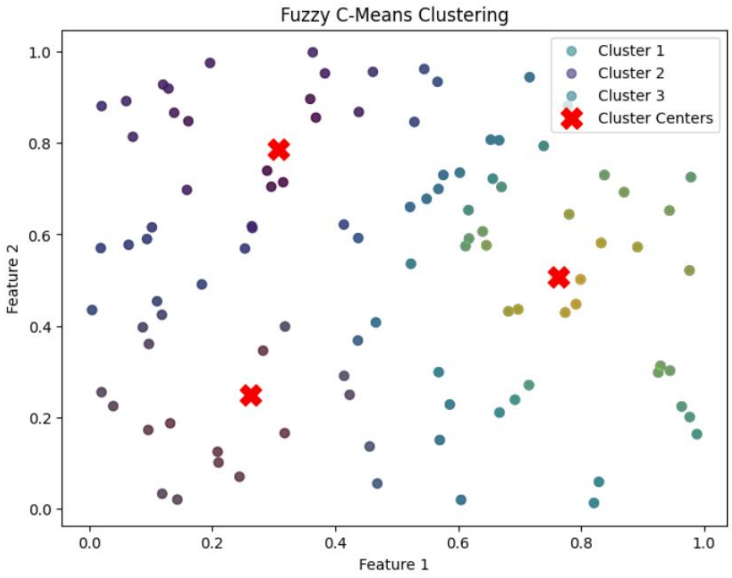

Рис. 1.4.1 – Приклад результатів FCM. Колір кожного спостереження нечіткий – змішаний настільки, наскільки відноситься до кожного кластеру

Отже, алгоритм FCM передбачає таку послідовність кроків:

1. Визначення кількості кластерів розбиття c.
2. Випадкове задання кожному спостереженню вектору міри належності до кластеру γ_i, де i – це кластер.
3. Обрахування центрів кластерів: $V_{ij} = \frac{\sum_{k=1}^{n} \gamma_{ik}^{m} * x_{kj}}{\sum_{k=1}^{n} \gamma_{ik}^{m}}$, де V_{ij} – це j-та координати i-того центру, n – кількість спостережень, x_{kj} – це j-та координата k-того спостереження, γ_{ik}^{m} – міра належності k-того спостереження до i-того кластеру, m – ступінь нечіткості вихідних кластерів, де $m \in (1; \infty)$. За замовчуванням $m = 2$.
4. Обрахування дистанції від кожного спостереження до центру кожного кластеру за будь-якою метрикою. Наприклад, за метрикою Евкліда: $D_{ki} = \sqrt{\sum_{j=1}^{t}(x_{kj} - V_{ij})^2}$, де D_{ki} – відстань від k-того спостереження до

i-того центру, t – загальна кількість координат даних (вимірність), x_{kj} – j-та координата k-того спостереження, V_{ij} – це j-та координати i-того центру.

5. Оновлення мір належності для кожного спостереження:

   $\gamma_{ki} = (\sum_{j=1}^{c}\{\frac{D_{ki}^2}{D_{kj}^2}\}^{(\frac{1}{m-1})})^{-1}$, де γ_{ki} – міра належності k-того спостереження до i-того кластеру, c – кількість кластерів, D_{ki} – відстань від k-того спостереження до i-того центру, D_{kj} – відстань від k-того спостереження до j-того центру, m – ступінь нечіткості вихідних кластерів.

6. Повторення кроків 3-5 допоки значення γ не стануть константними або різниця між мірами належності останньої ітерації і попередньої менша ніж допустиме значення.

Ступінь нечіткості m напряму керує тим, наскільки «розмитими» будуть результати кластеризації. При наближенні m до одиниці, результати стають жорсткими, тобто спостереження будуть належати тільки до одного кластеру (наприклад, якщо кластерів 4, то міра належності може описуватися як $\gamma = \{0,0,1,0\}$). І навпаки – якщо m прямує до нескінченності, то все більше даних будуть мати рівномірний ступінь належності до всіх кластерів ($\gamma = \{0.25,0.25,0.25,0.25\}$). Існують модифікації до FCM, при якому значення m адаптується та змінюється під час роботи алгоритму.

Нечітка кластеризація використовується там, де об’єкти чи явища складно віднести тільки до однієї групи. Часто такі підходи використовуються в медичній діагностиці, коли пацієнт може частково належати до кількох діагностичних категорій.

1.5. Психотипи Бартла

Розробка комп’ютерних ігор включає в себе не тільки створення графіки, програмування і проєктування ігрових механік, а й детальний аналіз поведінки

гравців. Таким питанням задався британський геймдизайнер Річард Бартл [20-22] при розробці першої текстової багатокористувацької гри MUD (Multi-User Dungeon). Аналізуючи дії, реакції та зосередженість гравців на різних елементах гри він узагальнив це в такі психотипи:

1. Досягальники (Achievers) – ті, кому приносить задоволення прогрес та нагороди. Вони зосереджуються на зборі всіх трофеїв, виконують всі квести задля максимізації своєї особистої статистики.
2. Дослідники (Explorers) – ті, хто шукає новизну в ігровому світі. Вони відвідують всі локації, тестують механіки та зацікавлені в ігровій історії.
3. Кілери (Killers) – ті, хто прагне домінувати та впливати на інших. Зазвичай це про гравців, які шукають суперництва, PvP (player versus player) битв чи PvE (player versus environment) верховенства.
4. Соціальники (Socializers) – ті, хто хоче спілкуватися та знаходити нові зв'язки. Такі люди створюють спільноти в грі та допомагають іншим.

Модель Бартла стала фундаментальною при роботі з геймдизайном, особливо в MMO (massively multiplayer online) та іграх із відкритим світом. Вона дозволяє балансувати контент так, щоб утримувати зацікавленість гравців різних типів і забезпечувати їм індивідуально привабливий ігровий досвід.

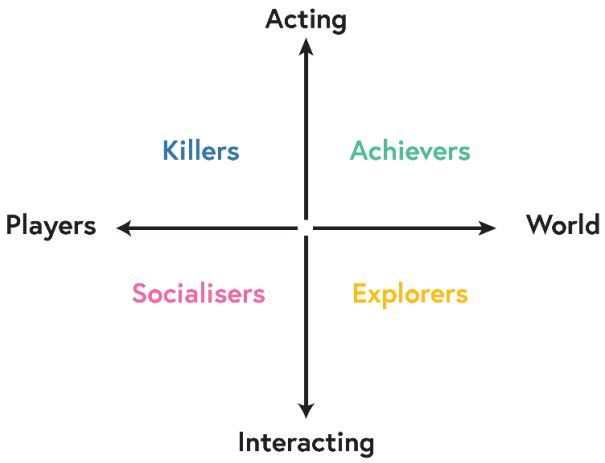

Рис. 1.5.1 – Візуалізація Моделі Бартла. Acting (дія) – вплив на світ або інших, Interaction (взаємодія) – комунікація чи дослідження замість прямого впливу, Players (гравці) – фокус на інших гравцях, World (світ) – фокус на ігровому світі

З часом, психотипи Бартла стали більш універсальними, оскільки розробники почали спиратися на них не тільки при розробці ММО. Така класифікація є корисною у будь-яких ігрових та навіть неігрових системах, де є взаємодія користувача зі світом чи іншими учасниками. Наприклад, досягальники є в RPG чи гонках (максимізація особистого рівня, рекордні бали), дослідники – в стратегіях чи платформерах (знаходження секретів, експерименти з механіками), кілери – в шутерах, файтингах чи навіть економічних симуляторах, соціальники – здебільшого у кооперативних іграх, але й в сингліплеєрних іграх теж можна виділити такий тип гравців (симулятори життя, спілкування з неігровими персонажами, спілкування у VR-просторі).

1.6. Дилема «Exploration–Exploitation»

У штучному інтелекті, а найчастіше у навчанні з підкріпленням, рутинною проблемою є так названа дилема «exploration–exploitation» [23] (розвідка та експлуатація). Її суть полягає у знаходженні найоптимальнішого розв'язку задачі у випадках, коли система має приймати рішення між дослідженням нових можливостей (exploration) та використанням уже відомих вигідних стратегій (exploitation). Розвідка означає спроби знайти нові рішення, навіть якщо вони можуть тимчасово знизити ефективність. Це дозволяє у майбутньому відкрити стратегії з кращими результатами. Експлуатація, навпаки, зосереджується на використанні вже здобутого знання для отримання максимальної користі. Якщо система бачить, що певний тип контенту відповідає гравцеві (наприклад, досліднику подобається знаходити секретні кімнати), вона починає створювати більше таких сценаріїв. Це підвищує задоволення гравця і дозволяє системі швидко оптимізуватися під конкретні вподобання. Проте занадто сильна експлуатація веде до одноманітності й втрати інтересу. Задля мотивації алгоритму досліджувати нові варіанти зазвичай вводяться штучні винагороди за етап розвідки, балансуючи роботу такого підходу.

Принцип «exploration–exploitation» активно використовується в різних сферах: у рекомендаційних системах (наприклад, Netflix чи YouTube) — щоб збалансувати показ уже улюбленого контенту користувача з новими, потенційно цікавими матеріалами; у маркетингу та A/B-тестуванні — для дослідження нових стратегій просування, не втрачаючи прибутку від перевірених методів; у робототехніці та підкріплювальному навчанні — для балансування між відомими діями, що приносять винагороду, і новими, які можуть дати кращий результат у майбутньому.

1.7. Інформаційна ентропія Шеннона

Інформаційна ентропія Шеннона [24] — це міра невизначеності або непередбачуваності випадкової величини. Вона показує, скільки інформації в середньому міститься в одному повідомленні джерела, якщо відомі ймовірності всіх можливих подій. Чим рівномірніше розподілені ймовірності, тим більша ентропія, адже тоді важче передбачити результат. Наприклад, ентропія розподілу відвідувачів магазину буде вищою, якщо всі товари купують приблизно однаково часто, і нижчою, якщо майже всі покупці беруть лише один конкретний товар. Математичне формулювання таке: для випадкової величини X з множиною можливих подій $\{x_1, x_2, \dots, x_n\}$ та ймовірностями $p(x_i)$ ентропія визначається як:

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i).$$

Ентропія Шеннона широко застосовується в теорії інформації, машинному навчанні, криптографії, аналізі даних та біоінформатиці. У кластеризації вона використовується для вимірювання «розмитості» належності об’єкта до різних кластерів. У випадку аналізу поведінки гравця ентропія може допомогти оцінити, наскільки визначений його психотип: якщо одні кластери домінують — ентропія низька; якщо всі ймовірності схожі — ентропія висока. Це дозволяє балансувати між експлуатацією відомих патернів і дослідженням нових.

1.8. Аналіз останніх досліджень у процедурній генерації світу та аналізі поведінки гравця

Дослідження процедурної генерації здебільшого спрямовані на розробку таких алгоритмів, які дають достатньо контролю над результатом генерації і водночас забезпечують різноманітність та непередбачуваність. Проблема кластеризації даних поведінки гравців полягає в правильній обробці багатовимірності спостережень та коректній інтерпретації вихідних кластерів, оскільки це на пряму впливає на результативність проведеного дослідження для розробників ігор, які прагнуть використати плоди праці для покращення продукту.

Одним з останніх оглядів алгоритмів процедурної генерації є робота «Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration» [25], яка враховує активний розвиток великих мовних моделей LLM. Автори класифікували методи на 5 великих груп (рис. 1.6.1):

1. Пошукові (search-based) – еволюційні алгоритми, шуми, методи Монте-Карло.
2. Методи машинного навчання – навчання з підкріпленням (reinforcement learning), автоенкодери (VAE), марковські моделі.
3. Інші методи – фрактали, генеративна граматика та інші графові моделі.
4. Великі мовні моделі (LLM) – генерація наративів, діалогів, квестів, рівнів.
5. Комбіновані методи – інтеграція декількох підходів (наприклад, LLM та генетичні алгоритми)

Окрім цього, автори описують майбутні можливі напрями розвитку процедурної генерації – удосконалення 3D PCG, використання LLM як асистентів для дизайнерів, поєднання LLM з іншими алгоритмами для створення гібридних систем, розробка інструментів для виправлення та вдосконалення контенту, а не тільки для повної його генерації.

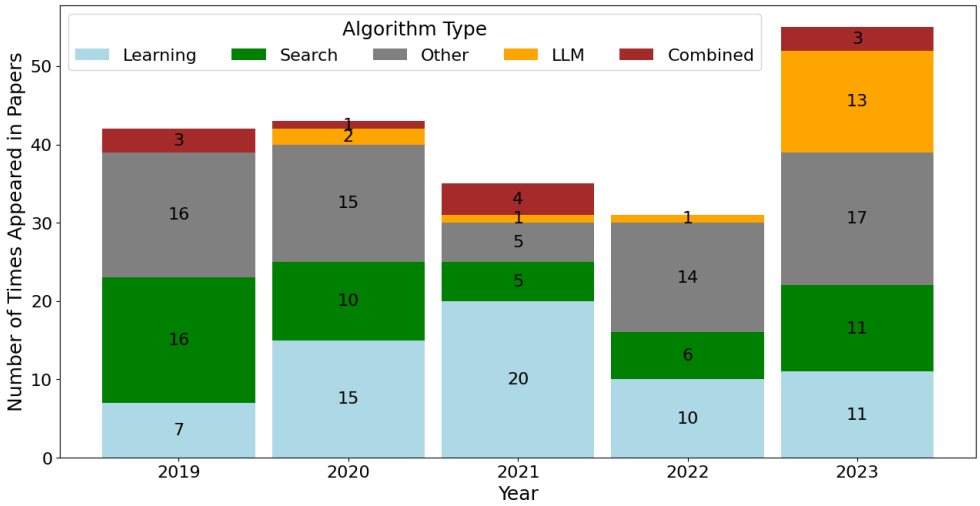

Рис. 1.6.1 – Статистика використання різних типів алгоритмів, пов'язаних з PCG, в дослідницьких роботах за 2019-2023 роки

У роботі Пола Меррела «Example-Based Procedural Modeling Using Graph Grammars» [26] автор розробив новий підхід до генерації правил переписування графів на основі початкових шаблонів (рис. 1.6.2). Вхідним параметром для формування правил є приклад – патерн, в малих масштабах кожна частина якого може бути подібна до вихідного графу, але при загальному вигляді абсолютно відрізняється. Такий шаблон розбивається на примітиви та визначає ієрархію підграфів, таким чином, що новий варіант обов'язково може бути спрощений до того, який знаходиться вище в ієрархії. Окрім цього, Пол Меррел визначив як описувати кордони графу (межі трасування), які використовуються в переміщені підграфів.

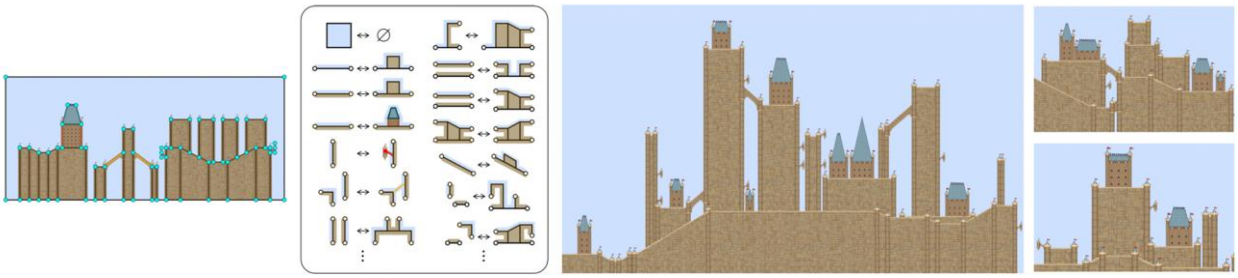

Рис. 1.6.2 – Алгоритм Пола Меррела. Зліва направо – вхідні приклади-шаблони, формування правил переписування графу, вихідний результат

У роботі німецьких та австрійських дослідників «ShapeGenetics: Using Genetic Algorithms for Procedural Modeling» [27] представлено спосіб використання генетичних алгоритмів для керування вихідними результатами процедурного 3D моделювання. Вони використовують підходи переписування графів – граф-генератор, для задання логіки генерації, та пропонують ефективний спосіб кодування хромосом і організовують їх в ієрархічне представлення геному. Генетичні операції мутації та схрещування відбуваються на цих представленнях – деревах характеристик моделі генерації. Окрім цього, вони порівнюють свій спосіб із іншими методами контрольованої процедурної генерації – зворотний стрибок Монте-Карло марковських ланцюгів (Reversible Jump Markov Chain Monte Carlo, RJMCMC) та стохастично впорядкований послідовний метод Монте-Карло (Stochastically-Ordered Sequential Monte Carlo, SOSMC). Автори показують як їхній генетичний алгоритм демонструє швидшу конвергенцію ніж SOSMC і генерує кращі моделі ніж RJMCMC у довгостроковій перспективі.

У роботі «Guns, Swords and Data: Clustering of Player Behavior in Computer Games in the Wild» [28] автори описали, як кластеризували більш ніж 260 000 спостережень поведінки гравців з двох великих комерційних комп’ютерних ігор – Tera і Battlefield 2: Bad Company 2 (BF2BC2). Їхня мета була знайти поведінкові шаблони, які можуть допомогти розробникам балансувати ігровий процес, виявляти проблемні зони, визначати групи ризику гравців, які можуть покинути гру, чи ідентифікація цінних гравців. Для такого аналізу були використаний класичний метод *k*-середніх для загального розподілу поведінки та Simplex Volume Maximization (SiVM) для виявлення екстремальних типів гравців. За результатами кластеризації були виявлені 6 кластерів для Tera та 7 для BF2BC2. Оскільки Tera є грою жанру MMO та спирається більше на PvE контент, то кластери добре відображають різні ігрові стилі (бойовий, соціальний, економічний, ремісничий), а у BF2BC2 кластери розподіляють гравців за рольовою поведінкою та ефективністю (снайпери, ветерани, новачки-жертви,

спеціалісти з техніки).

У роботі «A Modified Genetic Algorithm Based FCM Clustering Algorithm for Magnetic Resonance Image Segmentation» [29] індійські вчені описують модифікований метод нечіткої кластеризації Fuzzy C-Means на основі генетичного алгоритму для сегментації зображень МРТ. Запропонований алгоритм MfGA покращує ініціалізацію популяції за допомогою класових рівнів, які обчислюються через зважене середнє інтенсивностей пікселів, що зменшує ймовірність хибної кластеризації. Їхній алгоритм сегметації включає таку послідовність кроків: ГА знаходить оптимальні початкові центри кластерів, отримані значення передаються в FCM як стартові точки, FCM уточнює сегментацію через покращені початкові центри. Запропонований алгоритм показав кращі результати за точністю і часом роботи у порівнянні з класичним FCM.

Проаналізувавши широкий спектр досліджень процедурної генерації, аналізу поведінки гравця та кластеризації даних, можна зробити висновок, що сучасні підходи у розробці ігор поєднують багато парадигм автоматизованого створення ігрового контенту та інтелектуальних методів обробки даних. Така інтеграція дозволяє забезпечити різноманіття та адаптивність ігрових середовищ, що відкриває можливості для створення динамічних і персоналізованих світів.

РОЗДІЛ 2.
РОЗРОБКА МОДЕЛІ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СВІТУ З АНАЛІЗОМ ПОВЕДІНКИ ГРАВЦЯ ДЛЯ КОМП'ЮТЕРНОЇ ГРИ

2.1. Постановка задачі

Однією із проблем сучасної процедурної генерації віртуального світу є її адаптація в широкому сенсі – важко передбачити результати генерації за вхідними параметрами, отже і важко її адаптувати. З іншого боку алгоритми, які забезпечують хорошу контрольованість результатів, не задовольняють гравців, оскільки генерація часто повторювана та не унікальна. Зважаючи на це, формується проблема балансування передбачуваності, різноманітності та індивідуалізації в процедурно згенерованому адаптивному світі.

У цій роботі розглядається спосіб процедурної генерації будівлі розподіленої на кімнати для комп'ютерної гри, яка передбачає циклічне проходження рівня від початку до кінця, збір даних та адаптацію генерації на наступній ітерації. Для цього використовуються алгоритм на основі BSP дерева з модифікаціями для задання типу та формування багатокутних (непрямокутних) кімнат, генетичний алгоритм для характеристик кожної кімнати (її наповнення ворогами, діалогами, предметами) та нечітка кластеризація поведінки гравця, як вхідний параметр для генетичного алгоритму. Середовищем розробки є сучасний рушій для розробки комп'ютерних ігор Unreal Engine 5.4 з використанням фреймворку для процедурної генерації Procedural Content Generation Framework, який дає широкий спектр інструментів для написання логіки генерації та оптимізації в реальному часі. Після розробки, охарактеризовано можливі покращення та оптимізації алгоритму в різних його аспектах.

Очікуваний результат – це розроблений прототип гри із готовим алгоритмом, який генеруватиме різні варіанти будинку та наповнення кімнат, і буде адаптовуватися до поведінки гравця так, щоб йому було цікавіше

перепроходити рівні. У перспективі ця модель процедурної генерації може бути використана і в реальних ігрових проєктах та досліджена краще.

2.2. Методологія дослідження

Для досягнення поставленої мети необхідно насамперед проаналізувати сучасні алгоритми процедурної генерації, які покладають базу для розуміння функціонування цього аспекту розробки відеоігор. Вони включають методи переписування графу, генетичний алгоритм, BSP дерево, діаграма Вороного та градієнтні шуми. Для аналізу поведінки гравця необхідно розглянути особливості нечіткої кластеризації та деталі роботи алгоритму Fuzzy C-Means, а також психотипи Бартла для оцінки можливих кластерів у розроблювальній грі.

Наступним кроком дослідження є розробка базового прототипу, основний функціонал якого описаний на діаграмі прецедентів у додатку А. Необхідно реалізувати геймплей від першої особи із базовими можливостями руху, паузи, використання предметів для встановлення в слот персонажа чи одноразово, діалогу з неігровими персонажами (NPC) та їхній пропуск, а також ворогів, які атакують, і механіку перемоги над ними. Такий обсяг контенту дозволить проводити дослідження з аналізу впливу поведінки гравця на процедурну генерацію та відстежувати зміни в роботі алгоритму.

Після цього можна приступати до реалізації моделі PCG, яка має включати в себе адаптивність до поведінки гравця. У додатку Б на діаграмі послідовності визначений основний ігровий цикл між гравцем і моделлю PCG:

1. Гравець розпочинає гру.
2. PCG модель генерує випадкове BSP дерево за заданою кількістю кімнат – каркас розподілу будинку на кімнати.
3. PCG модель випадково визначає типи кімнат з BSP дерева – початкова, проміжна чи кінцева.
4. PCG модель нечітко кластеризує збережені дані гравця за допомогою Fuzzy C-Means методу.

5. PCG модель генерує характеристики кожної з кімнат генетичним алгоритмом на основі проаналізованих даних.
6. Відкриття мапи будинку з повідомленням очікування генерації.
7. PCG модель генерує всі об’єкти будинку в окремому потоці.
8. Гравець проходить гру і закінчує ігрову сесію.
9. PCG модель зберігає дані з цього проходження.
10. Гравець опиняється в головному меню гри.

Визначення каркасу має відбуватися на основі випадкового BSP дерева із заданою кількістю кімнат. Алгоритм на основі BSP дерева модифіковано так, щоб забезпечити розподілення кімнат за типом (початкова і кінцева кімната повинні знаходитися подалі одна від одної) та багатогранність кімнат будинку. Кожен вузол BSP дерева представляє центр розбиття простору, а не просто територію, яка була розділена вертикальною чи горизонтальною лінією на попередньому кроці.

Генетичний алгоритм необхідно проводити спираючись на результати нечіткої кластеризації. Правила меж адаптивності визначити експериментально на основі проходження гри різними гравцями. Характеристики кімнат, на яких буде проводитися генетичний алгоритм включать: скриня з предметами, NPC, ворог, пастка, світло, стовп, колекційний предмет, предмет дослідження.

Для аналізу поведінки гравця в контексті цього дослідження необхідно використати такі показники, що явно покажуть результати, які можна інтерпретувати на основі класифікації Бартла. Для цього виділено такі показники в межах одного проходження:

- відсоток відвіданої території будинку;
- відсоток відкритих скринь;
- відсоток зібраних колекційних предметів;
- відсоток діалогів з NPC;
- відсоток пропуску діалогів з NPC;
- відсоток вбитих ворогів;
- відсоток ворогів, з якими розпочалась битва, але гравець їх уникнув;

- відсоток статичних предметів, з якими взаємодіяв гравець.

Завершальним етапом роботи буде проаналізувати результати генерації – різноманітність контенту, його відповідність до поведінки гравця, швидкодія алгоритму та процесу генерації, коли гравець очікує. Окрім цього, необхідно оцінити подальші перспективи розвитку методу, можливі покращення та інші сфери його застосування.

2.3. Обґрунтування вибору інструментальних засобів

Оскільки дослідження потребує розробки прототипу комп'ютерної гри, буде доцільним використати готовий рушій для розробки відеоігор. Вибір розробників часто залежить від специфіки роботи, складності та крос-платформеності вихідного продукту, над яким працюють. Сьогодні найпопулярнішими ігровими двигунами є Unity, Godot та Unreal Engine 5 (UE5). Варто розглянути детальніше їх придатність до процедурної генерації, підтримки 3D графіки та доступності ресурсів для навчання.

Unity [30-31] є одним із найпопулярніших рушіїв через свою простоту в базовому освоєнні. Він добре підходить для 2D та мобільних ігор з простим казуальним геймплеєм. Сам рушій написаний на C++, однак для написання всіх логік гри використовується C#. Однак, вбудованих засобів для реальної 3D-процедурної генерації середовищ мало, тому доведеться писати більшість алгоритмів вручну. Окрім того, інструментів для якісної роботи з 3D графікою відносно мало.

Godot [32-33] особливий тим, що має відкритий вихідний код, безкоштовний та його популярність серед нових розробників наразі зростає. Він підтримує скриптування ігрових механік на GDScript, C# та C++. Є можливість для роботи з простою процедурною генерацією, однак екосистема слабша, ніж в Unity чи UE5. Графіка та оптимізація поступаються рівню високобюджетних ігор.

Unreal Engine 5 [34] характеризується потужною графікою (Nanite, Lumen),

яка забезпечує реалістичність навіть при процедурній генерації в реальному часі. Розробка всередині рушія відбувається за допомогою візуальної мови скриптування – Bluerpints, або C++ для складніших алгоритмів, які вимагають більш глибокого управління ресурсами. Спільнота UE величезна: існує велика кількість туторіалів, плагінів та документації, до якої легко доступитися онлайн. Окрім того, для UE версії 5.4 існує плагін, який легко підключити всередині рушія.

Отже, попри те, що Unity і Godot можуть бути простішими в освоєнні для початківців, UE5 є найбільш оптимальним для дослідження процедурної генерації світу та його адаптації до поведінки гравця на основі алгоритмів нечіткої кластеризації.

Плагін Procedural Content Generation (PCG) [35-36] дозволяє автоматизувати створення контенту як і в реальному часі, так і перед застосуванням даних у реальних ігрових ситуаціях. Написання логіки генерації відбувається за допомогою графової системи вузлів схожої на Blueprints. Усі операції відбуваються на основі задання точок в просторі (за сплайном – всередині, за межами чи по його контуру) та їхнього перетворення (операції масштабування, переміщення, обертання, проєкції на площину чи пряму) чи фільтрації (за висотою, кутом, тегами, щільністю). Для функціонування вхідною точкою плагіна є компонент PCGComponent для будь-якого актора та заданий PCG Graph у параметрах цього компонента. Такий спосіб робить плагін легким та зручним у використанні. Найчастіше його використовують для генерації ландшафту (дерев, трави, каміння, висот), міст чи будівель (розташування уздовж вулиць, інтер'єри/екстер'єри), підземель чи лабіринтів (кімнат і коридорів) та інших геймплейних характеристик (ворогів, предметів, пасток).

Для реалізації програмної C++ часини було використано два середовища розробки (IDE): JetBrains Rider та Visual Studio.

JetBrains Rider версії 2023.3.3 [37] – це кросплатформенне середовище розробки від компанії JetBrains, орієнтоване на C#, C++ та .NET-проєкти. Воно має розширені можливості для роботи з Unreal Engine, включаючи підтримку

C++-проектів, інтеграцію з CMake, налагодження, підсвічування коду та рефакторинг. Порівняно з Visual Studio, Rider забезпечує швидший аналіз коду, зручнішу навігацію по проєкту та сучасний інтерфейс, тому в даному дослідженні Rider було використано як основне середовище для написання і налагодження коду процедурної генерації.

Visual Studio версії Community 2022 [38] – це потужне середовище розробки від Microsoft, яке є стандартним для Unreal Engine. Воно має вбудовану інтеграцію з системами контролю версій, включно з Git та GitHub, тому в даній роботі Visual Studio використовувалося переважно для взаємодії з GitHub: керування репозиторієм, комітами, гілками та синхронізацією коду з віддаленим сервером.

Робочою станцією для розробки є ноутбук HP Pavilion Gaming Laptop 15 з такими характеристиками: процесор AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx (2.30 GHz), відеоадаптери AMD Radeon(TM) RX Vega 10 Graphics та дискретна NVIDIA GeForce GTX 1650, 16 ГБ ОЗП, 6 ГБ графічна плата, операційна система Windows 11 Pro (24H2). Для проєкту виділено 30 ГБ сховища на диску.

2.4. Реалізація PCG моделі

2.4.1. Структура проєкту та реалізація базового функціоналу гри

Оскільки гра створюється як варіант-прототип для роботи з процедурною генерацією, деталізація ігрових механік, історії та атмосфери відходить на другий план. Було вирішено, що гра буде представником хоррор-жанру, де гравець блукає у темряві.

Обрано розбити файли проєкту для легшого орієнтування та структурованості на такі елементи: Base, Building, AI, DataAssets, Effects, Inventory, Menu, NPC, Story, UI.

Модуль Base містить основний функціонал для роботи гри – базовий клас основного гравця (AHorrorGameCharacter), його контролер

(AHorrorGameController), ігрового екземпляру (AHorrorGameInstance), класи для роботи з динамічними даними гри та окремо гравця (AHorrorGameState та AHorrorPlayerState), а також клас головного збереження (UMainSaveGame). Окремо в папці Components знаходяться компоненти, які реалізують основні механіки гри – здоров'я, виснаженість, взаємодія з ефектами, інвентарем, діалогами.

Уся логіка для роботи з процедурною генерацією описана в модулі Building. Він включає клас аналізу даних (UBehaviourAnalysis), збереження (UBehaviourSaveGame), роботу з BSP деревом (UBSPTree) та генетичним алгоритмом (UTraitGeneticAlgorithm, URoomTraits, UGeneticAlgorithmSave).

Модуль DataAssets містить базові C++ класи створених дата асетів – UPDABSPTree та UPDAModularRoom. У Blueprint створені дочірні класи та визначено дані, з якими вони працюють. На рівні C++ описані функції ініціалізації, роботи та отримання даних.

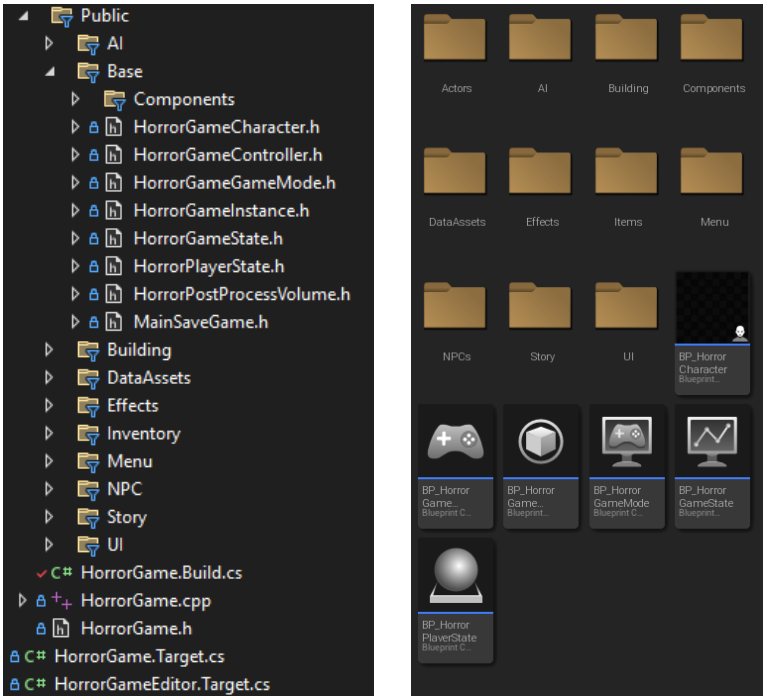

Рис. 2.4.1 – Структура проєкту. Ліворуч – в середовищі розробки Vusial Studio. Праворуч – в едітору рушія UE

Гра містить функціонал ефектів на гравця та предмети від неігрових

персонажів чи пасток. У модулі Effects знаходиться C++ клас UEffect та Blueprint класи кожного конкретного ефекту (рис. 2.4.2):

- Pixelization (пікселізація): перетворює зображення гравця на пікселі, може накопичуватися до 5-ти разів.
- Feedback Loop (проблема роботи предмета): затримує електричний предмет у циклі власної енергії, спричиняючи повторювані або посилені ефекти роботи, однак також ризикуючи самопошкодженням або нестабільністю.
- Electrolysis (електроліз): електричні предмети можуть викликати удар струмом та уповільнити швидкість. Якщо удар накопичується 5 разів за певний час, то з’являється Shock Wave (ударна хвиля), яка приголомшує гравця.
- Overcharged (перезарядження): збільшує потужність електричного предмета в руках, але з ризиком перегорання або пошкодження.
- Photon Flux (світло фотона): предмет у руках починає випромінювати світло.
- Pixel Shield (піксельний щит): поглинає одну зовнішню шкоду від ворога чи пастки. Може накопичуватися до 3 разів.
- Sense of Danger (почуття небезпеки): дає гравцю шанс збільшити швидкість персонажа протягом певного часу.

У модулі Inventory описані класи для роботи з предметами інвентаря. Серед них базовий клас UItem, який описує загальний функціонал предмету, дочірній UBodyItem, який може бути розміщений в один зі слотів (мозок, очі, легені, серце, хребет), AItemActor – описує фізичний вигляд та функціонал предмету в світі, він прив’язаний до UItem, дочірній AElectricItemActor – додає функціонал обробки ефектів, які впливають на електричні предмети. Серед основних реалізованих предметів є:

- Flashlight (ліхтарик): головний предмет для освітлення зовнішнього простору. Повільно розряджається. Для відновлення заряду необхідно знайти батарейки (Energized Stars).

- Electric Heart (електричне серце): встановлюється в слот серця. Додає здоров'я ігровому персонажу.
- Night Vision Glasses (окуляри нічного бачення): встановлюється в слот очей. Дає ефект нічного бачення в темряві.
- Electric Lungs (електричні легені): покращує витривалість (збільшує показник stamina), але накладає 2 накопичення ефекту пікселізації.
- Waterproof Shell (водонепроникна оболонка): запобігає накладанню ефектів Feedback Loop та Electrolysis.

Рис. 2.4.2 – Розроблені блупрінти ефектів в едіторі двигуна

Модуль Menu містить класи для роботи карти меню. Головний з них – клас AMenuGameMode.

Модуль NPC описує класи роботи неігрових персонажів, які гравець може зустріти в будинку. Стандартно це класи його фізичного представлення (ANPCCharacter) та ШІ контролер (ANPCController).

Модуль Story створений для задання сюжету гри, а саме діалогів з неігровими персонажами. Основний класом тут є UStoryline, в якому описуються послідовні кроки роботи кожного діалогу.

Модуль UI описує всі віджети та методи взаємодії з графічним інтерфейсом гри (рис. 2.4.3). Клас AHorrorHUD – це проміжний клас для розмежування логіки самої гри і логіки інтерфейсу. Саме через нього здійснюється взаємодія: дані гри → $AHorrorHUD$ → інтерфейс.

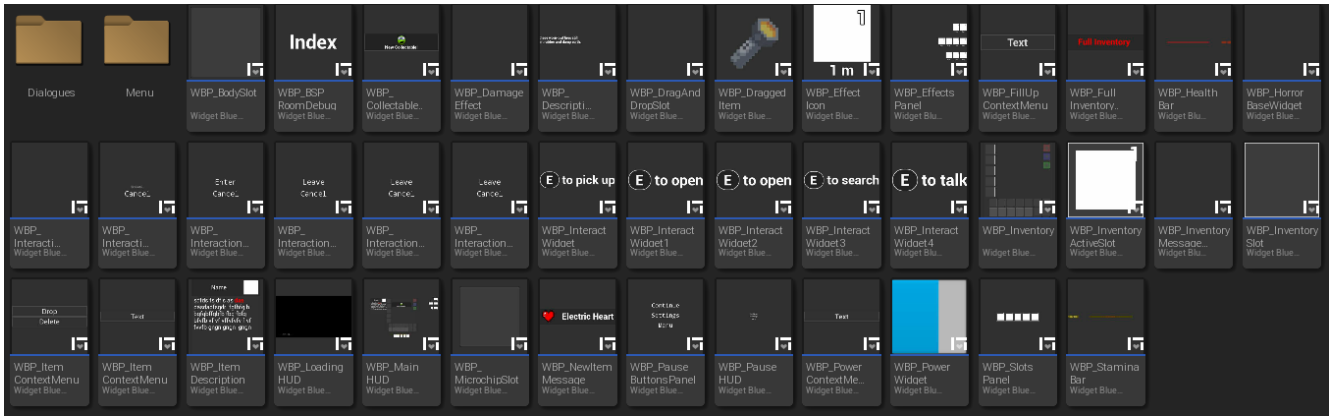

Рис. 2.4.3 – Розроблені UI віджети в едіторі рушія

У модулі AI знаходяться класи AEnemyCharacter та AEnemyController, які описують батьківських функціонал для всіх AI ворогів гри. Вони включають дочірніх ворогів Cactus, Slime, Turtle та Chest, які мають свої особливі атаки та ефекти накладання на гравця. Логіка їхнього функціонування написана в BehaviourTree – спеціальних асетах для формування дерева поведінки штучного інтелекту агента.

2.4.2. Реалізація базового функціоналу PCG моделі

Для початку роботи із PCG плагіном, необхідно його підключити. Для цього у вкладці проєкту Edit > Plugins обрано Procedural Content Generation Plugin (рис. 2.4.4) та додано модуль PCG у проєкт.Build.cs файлі (рис. 2.4.5).

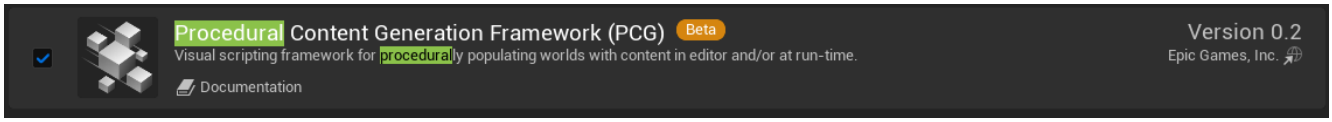

Рис. 2.4.4 – Підключення плагіну PCG

Після перезапуску проєкту можна розпочинати роботу з плагіном. Перш за все, створений клас, який є вхідною точкою роботи з процедурною генерацією – APCGRoom, та його нащадок BP_PCGRoom. Йому надано компоненти

UPCGComponent та USplineComponent, який буде використовуватись усередині PCG компонента для виявлення меж генерації. Окрім цього, створено параметри WallWidth, WallHeight, WallThickness для базового визначення розмірів кімнати, а також Subdivision – для задання кількості розподілень (кімнат) всього простору, Seed – для репродукції однакових «випадкових» результатів.

```
PublicDependencyModuleNames.AddRange(collection: new string[] {
    "Core", "CoreUObject", "Engine", "InputCore", "HeadMountedDisplay", "EnhancedInput",
        "UMG", "Slate", "SlateCore", "PCG", "AIModule", "Json", "JsonUtilities" });
```

Рис. 2.4.5 – Додання модуля PCG в залежності проєкту

У компоненті PCGComponent заданий граф, у якому розпочинається робота всієї логіки генерації – PCG_RoomMain. Він використовує інформації зі сплайну актора, для виявлення меж генерації. Тобто в межах сплайну будуть задані точки, які слугуватимуть вхідними даними для усіх інших трансформацій. Усе що генерується в графах PCG, виконується в окремому потоці у реальному часі та безпосередньо в едіторі розробки – без запуску гри. Це дозволяє швидко та ефективно проводити відлагодження процесу розробки PCG.

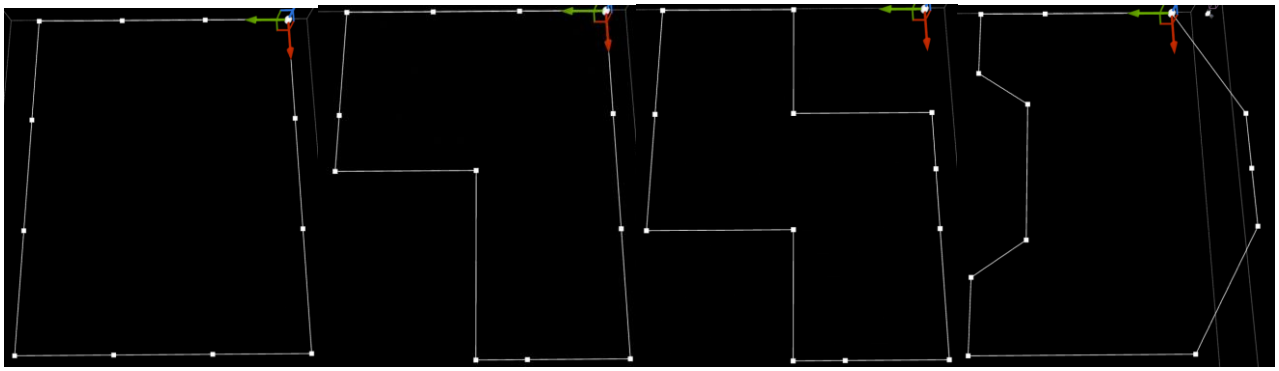

Рис. 2.4.6 – Форми сплайну за коефіцієнтом TransformationCoefficient. Зліва направо – 0; 0.25; 0.5 ; 1.

Оскільки PCG граф може діставати дані із самого актора, де знаходиться PCG компонент, важливим аспектом є використання дата асетів – контейнерів для зручного управління збереженням та накопиченням даних. У акторі

BP_PCGRoom задано два дата асети: DA_SplineData та PDA_BSPTreeMain. У DA_SplineData задані опорні точки трансформації сплайну для інтерполяційної зміни між різними формами будинку генерації. Таким чином, вийшло задати 4 форми сплайну (рис. 2.4.6) залежно від значення TransformationCoefficient(0; 0.25; 0.5; 1). У PDA_BSPTreeMain – уся інформація для формування BSP дерева та характеристик його кімнат. Окремо у графі PCG_RoomMain заданий дата асет PDA_ModularRoomCommon для задання даних каркасних елементів генерації (рис. 2.4.7) – стін, стовпів, підлоги, стелі, дверей залежно від типу кімнати, а також їхніх параметрів для правильного формування точок генерації.

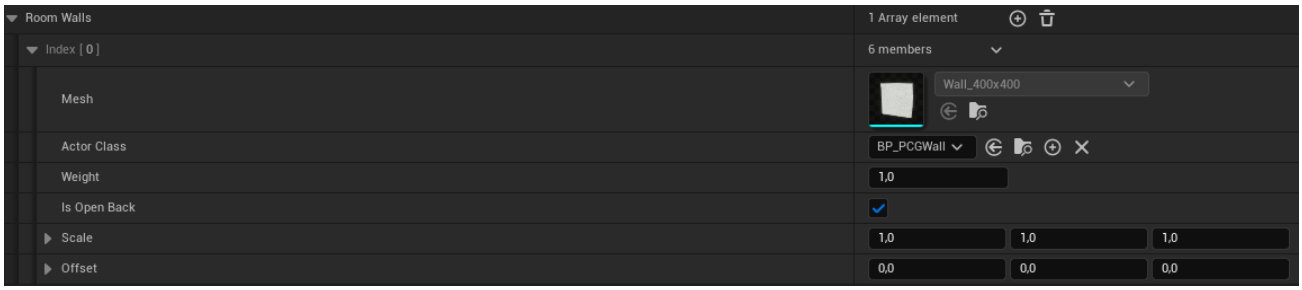

Рис. 2.4.7 – Задання стіни в PDA_ModularRoom. Задано меш, клас актора (йому надається пріорітетність над мешем), вага (ймовірність генерації), масштабування та відступ по X, Y, Z та параметр IsOpenBack (якщо false, то спавниться ще один меш, який закриває спину)

Логіка у графі PCG_RoomMain сформована таким чином, що всі деталі розподілені по окремим підграфах, що дозволяє перевикористовувати певний код та організувати правильні шари абстрагування логіки. Першим етапом у графі є діставання даних вхідного сплайну та, за допомогою ноди Spline Sampler (рис. 2.4.8), формування точок всередині замкнутої кривої лінії (інтер'єр сплайну). Після трансформації цих точок можна отримати вторинні точки побудови будинку. Наприклад, підграф PCG BSPSubdivisions повертає точки центрів кімнат за BSP деревом, PCG Room Walls – точки стін за точками сплайну, PCG Scale Mesh to Size – проходиться по кожній вхідній точці та масштабує за заданими параметрами ширини, висоти та товщини. Детальніше

графи з PCG_RoomMain наведені в додатку В.

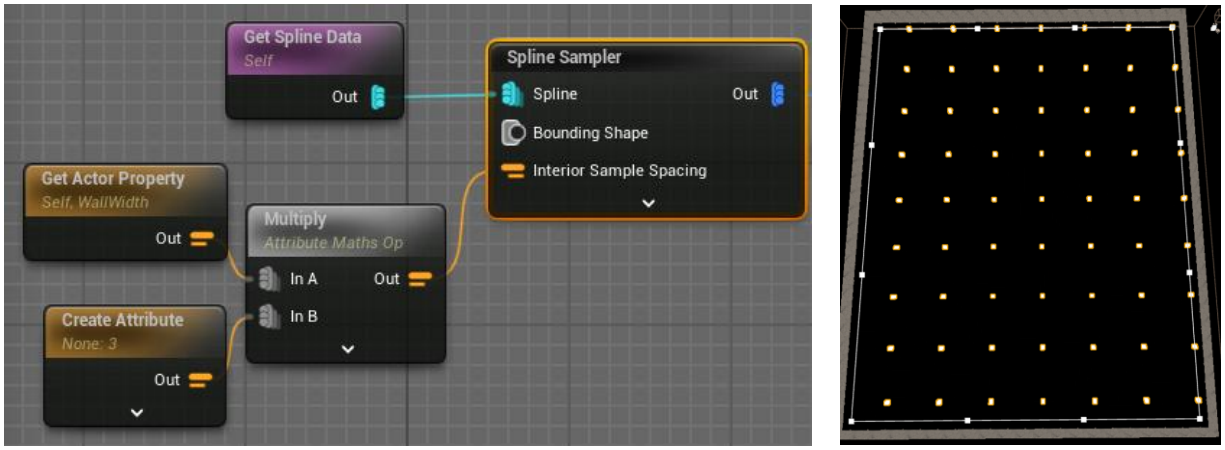

Рис. 2.4.8 – Ліворуч – використання Spline Simpler для формування точок генерації, interior sample spacing – відступ між точками (три довжини стіни). Праворуч – дебаг відображення точок по інтер'єру сплайну

2.4.3. Реалізація формування каркасу на основі BSP дерева

Етап формування каркасу передбачає знаходження найоптимальнішого «випадкового» варіанту розподілу всього простору будинку на кімнати. Класичний алгоритм на основі BSP дерева передбачає розподіл порожнього простору і формування кімнат за листками дерева, таким чином, що кімнати не мають спільних стін – звичайні прямокутні зони з'єднані коридорами. Такий підхід зрозумілий та ефективний, однак не підходить для розподілу обмеженого простору: наприклад, замкненого простору будинку за сплайном. Розроблена модифікація дозволяє використати BSP дерево і для такого випадку.

У класичному підході розбиття вузла (простору) BSP дерева на лівий і правий відбувається за допомогою прямої лінії, а самі вузли представляють простір, однак ніяк не інтерпретують його. Основна суть модифікації полягає у тому, що вузли BSP дерева – це центри кімнат, а розбиття відбувається не прямими лініями, а віднесенням кожної найменшої одиниці будинку (комірки) до найближчого вузла BSP дерева. Такий спосіб явно є й алгоритмом формування «випадкової» діаграми Вороного на основі BSP дерева із відносно

рівномірно розподіленими комірками.

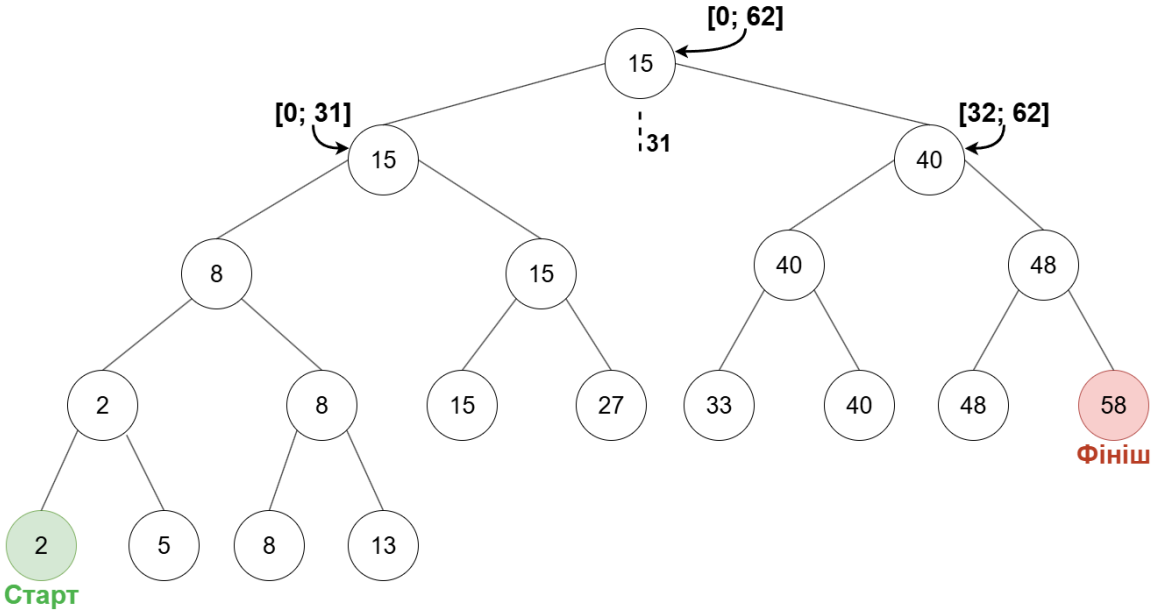

Рис. 2.4.9 – Приклад етапу формування BSP дерева для $Seed = 1$. Показано визначення центрів у трьох початкових вузлах. Також визначені початкова і кінцева кімнати

Етап злиття кімнат відбувається шляхом генерування дверей між кімнатами, а не прямих коридорів. У результаті вийде повністю розбитий будинок на кімнати зі спільними і не прямими стінами. Однак, виникає проблема сумісності злиття кімнат – у такій модифікації немає гарантії, що ліва і права вузли BSP дерева будуть мати спільну стіну. Це може статися через дві причини:

1. Кімнати за BSP вузлами на рівень нижче можуть «відрізати» поточний лівий і правий вузол.
2. Центри кімнат обралися так, що знаходяться на різних кінцях будинку. Тоді їх можуть перекривати не тільки вузли BSP дерева на рівень нижче.

Цю проблему було вирішено за допомогою введення черги злиття – вузли, які не можуть бути злиті зараз, переносяться в чергу та приєднуються після завершення основного злиття, а алгоритм продовжує роботу, ігноруючи не прикріплені кімнати.

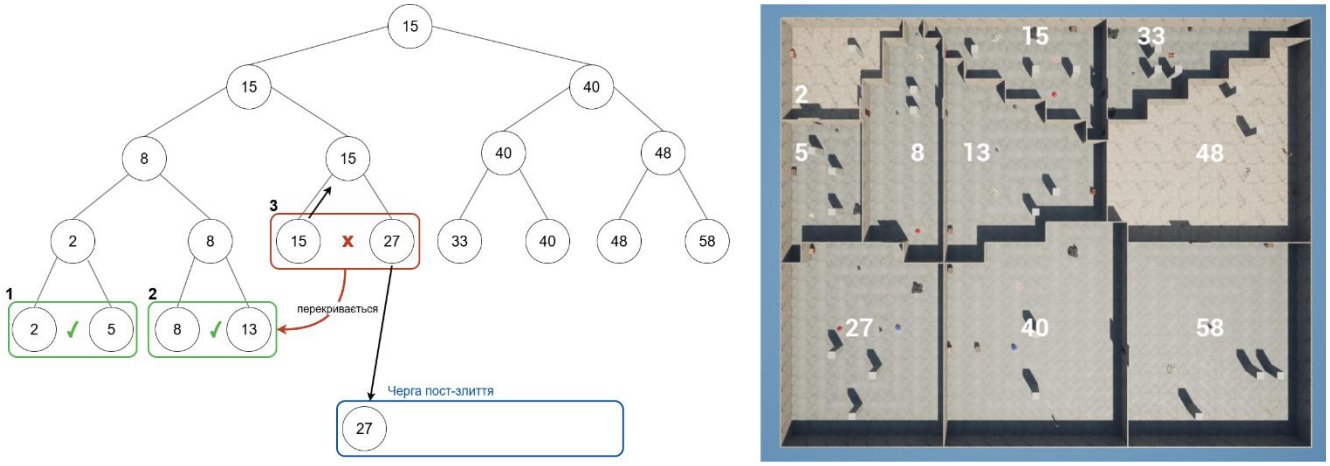

Рис. 2.4.10 – Приклад етапу злиття BSP дерева для Seed = 1. Показано перші три кроки злиття: кімнати із центрами 2 і 5, 8 і 13 злилися успішно, а 15 і 27 перекриваються злиттям 8 і 13, тому 15 переходить до вверх по дереву, а 27 переноситься в чергу пост-злиття

Якщо кількість кімнат (листків BSP дерева задана), то класичний підхід також не визначає конкретніше де відрізати простір, що робить розподілені кімнат нерівномірними (наприклад, одна дуже велика, інші малі знаходяться в купі). Також він не передбачає надання кімнатам сенсу – не визначено, яка кімната початкова, кінцева чи проміжна. Модифікований алгоритм намагається досягти рівномірності шляхом рекурсивного розподілення центрів кімнат за відсортованими по відстані точками можливих центрів (від найближчого до найдальшого) так, що центр лівого дочірнього вузла обирається з лівої половини відсортованих центрів, а правого – з правої (рис. 2.4.9). Початкова і кінцева кімнати обираються з найбільш лівих і найбільш правих листків сформованого дерева.

Отже, етап формування BSP дерева (розподілення на кімнати):

1. Задання кількості кімнат n і можливих центрів кімнат $r_{centers}$. Обов'язково $length(r_{centers}) \leq n$.
2. Сортування $r_{centers}$ за відстанню до певної опорної точки (наприклад, відносно першої точки центру) в порядку зростання.

3. Випадковий вибір кореня дерева з $[0; length(r_{centers}) - 1]$.
4. Рекурсивне призначення індексів центрів кімнат:
   4.1. Вхід в рекурсію розпочинається призначенням лівому вузлу індексу з $\left[0; \frac{(length(r_{centers})-1)}{2}\right]$, правому – з $\left[\frac{(length(r_{centers})-1)}{2} + 1; length(r_{centers}) - 1\right]$. Призначення $leftMin = 0$, $rightMax = ength(r_{centers}) - 1$, $sep = \frac{(length(r_{centers})-1)}{2}$.
   4.2. Виконання рекурсивного призначення центрів кімнат для лівого вузла з $[leftMin; sep]$.
   4.3. Виконання рекурсивного призначення центрів кімнат для лівого вузла з $[sep + 1; rightMax]$.
   4.4. Умова виходу з рекурсії – кількість вузлів дерева n.
5. Вибір початкової кімнати з $[0; n * 0.2]$ перших листків дерева, кінцевої – $[n * 0.8; n - 1]$ останніх листків, інші кімнати призначити проміжними.

Після цього відбувається розподіл комірок по своїх кімнатах та формування точок стін кімнат. Тоді можна приступати до етапу злиття BSP дерева (рис. 2.4.10) (знаходження проходів між кімнатами):

1. Ітераційний процес для проходження всіх вузлів BSP дерева з найглибшого рівня найлівішого листка – $n_{BSP} - 2, right = n_{BSP} - 1$, де n_{BSP} – кількість вузлів BSP дерева, які залишилися для проходу.
2. Злиття кімнат $left$ та $right$ кімнат – знаходження спільних стін та обрання випадкової для генерації проходу. Якщо спільних стін не знайдено, то той вузол, який є батьком для $left$ і $right$ вважати злитим, а інший – перенести в чергу пост-злиття $n_{excessive}$.
3. Призначення $n_{BSP} = n_{BSP} - 2$. Якщо $n_{BSP} \geq 2$, перехід до кроку 1. Інакше – зупинка ітераційного процесу.
4. Ітераційний процес проходження черги $n_{excessive}$. Злиття $n_{excessive}[0]$ і основного дерева. Якщо злиття успішне – видалення $n_{excessive}[0]$, інакше – перенесення $n_{excessive}[0]$ в кінець черги.

5. Продовження ітераційного процесу кроку 4 допоки $length(n_{excessive}) \neq 0$.

Реалізація етапу формування BSP дерева написана на C++ у класі UBSPTree (рис. 2.4.11-2.4.12). Функція InitializeRandom перевіряє всі умови алгоритму та ініціалізує дерево. PrescribeRoomIndex є точкою входу в рекурсивний процес формування BSP дерева.

```
void UBSPTree::InitializeRandom(int32 RoomsCount, int32 AvailableZonesCount)
{
    if(!Tree.IsEmpty() && !bAllowReinitializeFromRandom)
    {
        return;
    }
    if(RoomsCount > AvailableZonesCount)
    {
        return;
    }

    TArray<int32> AvailableZoneIndexes;
    for(int32 i = 0; i < AvailableZonesCount; ++i)
    {
        AvailableZoneIndexes.Add(i);
    }

    const int32 NodesCount = 2 * RoomsCount - 1;
    Tree.Empty(NodesCount);
    Tree.SetNum(NodesCount);

    if(Seed >= 0)
    {
        Stream.Initialize(Seed);
    }
    else
    {
        Stream.Initialize(NAME_None);
    }

    const int32 ZoneIndexToPrescribe = AvailableZoneIndexes[Stream.RandRange( Min: 0,  Max: AvailableZoneIndexes.Num() - 1)];
    PrescribeRoomIndex( TreeIndex: 0, ZoneIndexToPrescribe,  [&] AvailableZoneIndexes,  LeftIndexMin: 0,  RightIndexMax: AvailableZoneIndexes.Num() - 1);
}
```

Рис. 2.4.11 – Функція ініціалізації випадкового BSP дерева

```cpp
void UBSPTree::PrescribeRoomIndex(int32 TreeIndex, int32 ZoneIndexToPrescribe, TArray<int32>& AvailableZoneIndexes, int32 LeftIndexMin, int32 RightIndexMax)
{
    if(!Tree.IsValidIndex(TreeIndex))
    {
        return;
    }

    Tree[TreeIndex].ZoneIndex = ZoneIndexToPrescribe;

    const int32 LeftTreeIndex = 2 * TreeIndex + 1;
    const int32 RightTreeIndex = 2 * TreeIndex + 2;

    if(!Tree.IsValidIndex(LeftTreeIndex))
    {
        return;
    }
    if(!Tree.IsValidIndex(RightTreeIndex))
    {
        return;
    }

    if(!AvailableZoneIndexes.IsEmpty())
    {
        const int32 SeparatingIndex = (LeftIndexMin + RightIndexMax) / 2;
        const bool bIsNewIndexLeft = ZoneIndexToPrescribe >= SeparatingIndex;

        const int32 TreeIndexToPrescribeNewZoneIndex = bIsNewIndexLeft ? LeftTreeIndex : RightTreeIndex;
        const int32 AnotherTreeIndex = bIsNewIndexLeft ? RightTreeIndex : LeftTreeIndex;

        int32 NewRandomIndex = -1;
        while (!AvailableZoneIndexes.Contains(NewRandomIndex))
        {
            NewRandomIndex = bIsNewIndexLeft ?
            Stream.RandRange(LeftIndexMin,  Max: SeparatingIndex) :
            Stream.RandRange( Min: SeparatingIndex + 1, RightIndexMax);
        }
        AvailableZoneIndexes.Remove(NewRandomIndex);
```

```cpp
        if(bIsNewIndexLeft)
        {
            PrescribeRoomIndex(TreeIndexToPrescribeNewZoneIndex, ZoneIndexToPrescribe: NewRandomIndex, [&] AvailableZoneIndexes, LeftIndexMin, RightIndexMax: SeparatingIndex);
            PrescribeRoomIndex(AnotherTreeIndex, ZoneIndexToPrescribe, [&] AvailableZoneIndexes, LeftIndexMin: SeparatingIndex + 1, RightIndexMax);
        }
        else
        {
            PrescribeRoomIndex(TreeIndexToPrescribeNewZoneIndex, ZoneIndexToPrescribe: NewRandomIndex, [&] AvailableZoneIndexes, LeftIndexMin: SeparatingIndex + 1, RightIndexMax);
            PrescribeRoomIndex(AnotherTreeIndex, ZoneIndexToPrescribe, [&] AvailableZoneIndexes, LeftIndexMin, RightIndexMax: SeparatingIndex);
        }
    }
}
```

Рис. 2.4.12 – Рекурсивна функція призначення вузлам BSP дерева центру кімнати

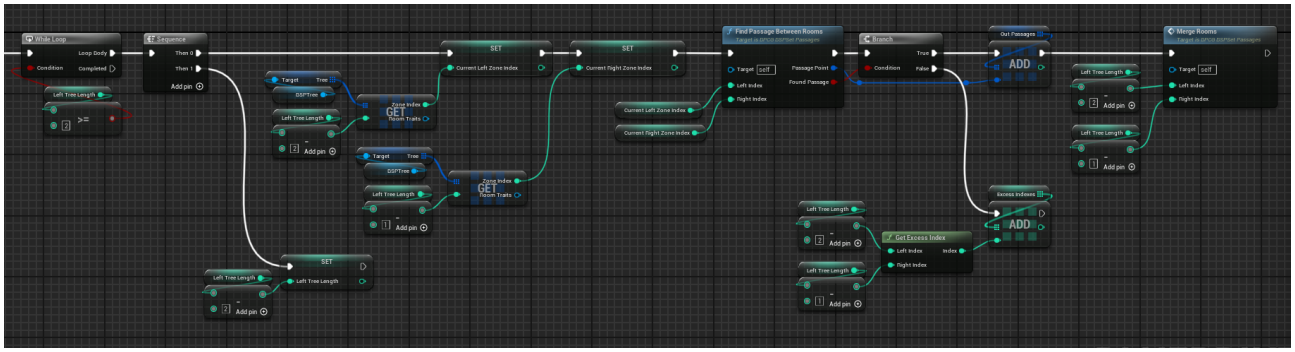

Рис. 2.4.13 – Проходження всіх вузлів BSP дерева, внесення надлишкових у чергу пост-злиття

Реалізація етапу злиття BSP дерева проведена на Blueprints як кастомна нода для PCG графу (рис. 2.4.13-2.4.14). Це зроблено задля забезпечення доступу

до даних про визначені точки стін, оскільки ця інформація використовується всередині функції FindPassageBetweenRooms.

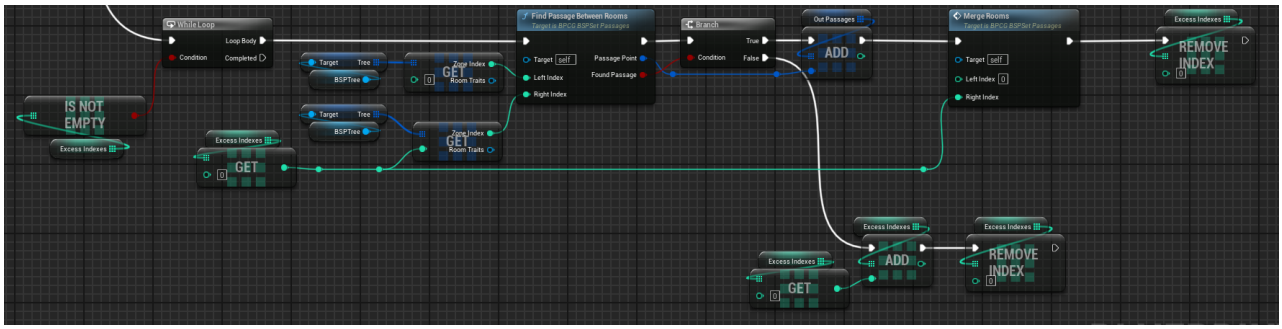

Рис. 2.4.14 – Оброблення черги пост-злиття визначення проходів між кімнатами

Розроблена модифікація дозволила використати BSP дерево в дещо іншому контексті. Вона має перевагу у керуванні логікою розташування типів кімнат і створює більш природні зв’язки між ними. Порівняння методів зображено на таблиці 1.

Таблиця 1. Порівняння класичного і модифікованого алгоритмів на основі BSP дерев

Характеристика	**Класичний BSP-алгоритм**	**Модифікований BSP-алгоритм**
Принцип поділу	Рекурсивне ділення області прямими (по вертикалі або горизонталі) на дві частини, поки не досягнуто мінімального розміру кімнат	Використання попередньо відсортованих індексів центрів кімнат (за дистанцією). Поділ здійснюється через побудову дерева на основі індексів центрів кімнат
Форма кімнат	Прямокутні області, отримані в результаті поділу BSP	Кімнати прив’язуються до обраних центрів. Всі клітинки відносяться до найближчого центру (подібно до діаграми Вороного)
Розташування кімнат	Визначається суто геометричними розрізами, випадковість обмежується напрямком і відсотком поділу	Випадковість закладена у виборі індексів у BSP-дереві, але зберігається відносна рівномірність завдяки розділенню на діапазони

Продовження таблиці 1

Етап злиття	Кімнати зазвичай з'єднуються коридорами між прямокутними областями. Стіни не обов'язково спільні.	Виконується знизу вгору. Якщо кімнати мають спільну стіну — формується прохід у ній. Якщо спільної стіни нема, то кімнати потрапляють у чергу пост-злиття для подальшого з'єднання
Типи кімнат	Зазвичай обираються випадково або за геометричними критеріями. Опирається на вже сформований розподіл	Визначаються найлівіший і найправіший листок у дереві, що гарантує максимальну відстань між ними. Опирається тільки на дерево, без його фізичного представлення
Структура рівня	Набір прямокутних кімнат, з'єднаних коридорами	Сітка кімнат зі спільними стінами; структура більш природна, схожа на граф приміщень із суміжністю
Контроль складності	Менш гнучкий: залежить від глибини поділу	Гнучкіший: можна керувати складністю через вибір індексів і розташування початкової/кінцевої кімнати

2.4.4. Реалізація кластеризації даних гравця

Першим етапом для кластеризації даних є їхній збір. Це реалізується за допомогою структури FBehaviourVector з визначеними атрибутами:

- CellsOverlappedPercentage: відсоток відвіданої території будинку;
- TreasuresCollectedPercentage: відсоток відкритих скринь;
- GlobalsCollectedPercentage: відсоток зібраних колекційних предметів;
- DialoguesPercentage: відсоток діалогів з NPC;
- DialoguesSkippedPercentage: відсоток пропуску діалогів з NPC;
- EnemiesKilledPercentage: відсоток вбитих ворогів;

- EnemiesAvoidedPercentage: відсоток ворогів, з якими розпочалась битва, але гравець їх уникнув;
- ItemsInteractedPercentage: відсоток статичних предметів, з якими взаємодіяв гравець.

Ці характеристики були відібрані таким чином, щоб їх можна було інтерпретувати як 4 кластери за класифікацією Бартла: досягальник, соціальник, кілер, дослідник.

```
void UBehaviourSaveGame::WriteJson(FString JsonFilePath)
{
    FString JsonString;

    JsonString.Append("[\n");
    for(int32 i = 0; i < BehaviourData.Num(); ++i)
    {
        if(const TSharedPtr<FJsonObject> JsonObject = FJsonObjectConverter::UStructToJsonObject(BehaviourData[i]))
        {
            FString StructInstanceString;
            FJsonSerializer::Serialize( Object: JsonObject.ToSharedRef(),
                Writer: TJsonWriterFactory<>::Create(&StructInstanceString, InitialIndent: 0));

            JsonString.Append(StructInstanceString);
            if(i < BehaviourData.Num() - 1)
            {
                JsonString.Append(",\n");
            }
            else
            {
                JsonString.Append("\n]");
            }
        }
    }

    const FString FilePath = FPaths::ProjectPersistentDownloadDir() + TEXT("/") + JsonFilePath;
    FFileHelper::SaveStringToFile(JsonString, *FilePath);
}
```

Рис. 2.4.15 – Функція збереження даних поведінки гравця в JSON форматі

Збереження даних відбувається за допомогою класу UBehaviourSaveGame, унаслідуваного від USaveGame, який містить масив векторів поведінки гравця за кожне його проходження. Для відлагодження параметру фазифікації також у цьому класі була реалізована функція WriteJson для збереження даних у JSON форматі (рис. 2.4.15).

Після чого використано R та RStudio для візуалізації збережених даних. Серед них 49 різнопланових спостережень проходження гри. Декілька спроб

дозволило визначити, що для достатньої розмитості вихідних кластерів параметр фазифікації оптимально встановити на рівні $m = 1.2$ (рис. 2.4.16). Так вийде показати і дані, які явно відносяться до певного типу поведінки, і дані, які поєднують характеристики різних психотипів. Використаний код на R зображений у додатку Г.

Безпосередня реалізація алгоритму (рис. 2.4.17) відбувалася на C++ у класі UBehaviourAnalysis. Етапи кластеризація розбито на окремі функції: InitializeFCMMatrix, CalculateCentroids, CalculateDistancesToCentroids, CalculateNewFCMMatrix – після чого організовані в одну основну функцію FuzzyCMeans. Увесь код реалізованого методу можна переглянути в додатку Д.

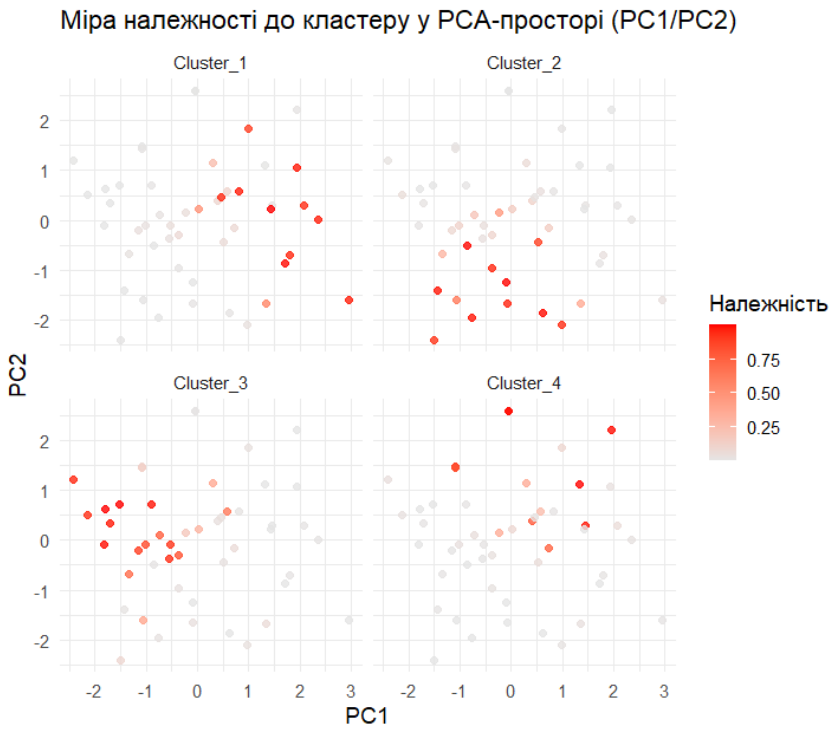

Рис. 2.4.16 – Візуалізація розмитих кластерів у R
за двома найважливішими ознаками

```
void UBehaviourAnalysis::FuzzyCMeans(TArray<TArray<float>>& FCMMatrix, TArray<TArray<float>>& Centroids,
    int32 ClustersCount)
{
    TArray<TArray<float>> NewFCMMatrix;
    InitializeFCMMatrix([&]NewFCMMatrix, ClustersCount);

    do
    {
        FCMMatrix = NewFCMMatrix;
        TArray<TArray<float>> DistancesToCentroids;

        CalculateCentroids([&]Centroids, FCMMatrix);
        CalculateDistancesToCentroids([&]DistancesToCentroids, Centroids);
        CalculateNewFCMMatrix([&]NewFCMMatrix, DistancesToCentroids);
    }
    while (GetTolerance(Old: FCMMatrix, NewFCMMatrix) > Tolerance);
}
```

Рис. 2.4.17 – Реалізація основної функції виконання алгоритму Fuzzy C-Means

Метод Fuzzy C-Means, як і класичні методи чіткої кластеризації, не дає визначення знайденим кластерам – не надає їм ніякого сенсу. Зазвичай потрібна людина, щоб правильно і оптимально інтерпретувати кластери. Однак, зважаючи на те, що вже визначено природу кластерів, які хочеться отримати, реалізовано простий алгоритм інтерпретації кожного кластеру, залежно від їхніх найкращих показників. Для цього оцінюється вектор середніх значень приналежності до кожного кластеру.

Отже, щоб інтерпретувати кластери в системі Бартла, розроблено такий алгоритм:

1. Формування вектора $MeanObj$ – середні значення всіх об’єктів приналежності до кожного кластеру.
2. Для кожного центроїда визначити кластер Бартла:
    2.1. Відсортувати ознаки центроїда за спаданням.
    2.2. Ітераційний процес по всіх відсортованих ознаках:
        2.2.1. Залежно від того, за який кластер відповідає атрибут, до такого додати вагу за формулою (n – кількість атрибутів, k – ітерація, позиція у відсортованому масиві) :

- Для досягальника: відсоток відкритих скринь – $\frac{n-k}{2}$, відсоток зібраних

колекційних предметів – $\frac{n-k}{2}$;

- Для соціальника: відсоток діалогів з NPC – $\frac{n-k}{2}$, відсоток пропусків діалогів – $\frac{k}{2}$;
- Для кілера: відсоток вбитих ворогів – $\frac{n-k}{2}$, відсоток уникнутих ворогів – $\frac{k}{2}$;
- Для дослідника: відсоток пройденої території – $\frac{n-k}{3}$, відсоток відкритих скринь – $\frac{n-k}{3}$, відсоток взаємодії із предметами – $\frac{n-k}{3}$.

2.3. Обрати незайнятий кластер із найбільшою вагою.

3. Приписати мірам належності вектора $MeanObj$ мітки кластерів Бартла.

Такий алгоритм розрахований на те, що центри із атрибутами найбільшими за значеннями визначають приналежність до кластеру найбільшою мірою. Окрім цього, алгоритм балансує неоднорідність атрибутів за допомогою введених дільників кількості відповідальних ознак за той чи інший кластер Бартла (наприклад, для досягальника вага ділиться на 2, бо відповідальні за цей кластер атрибути: TreasuresCollectedPercentage, GlobalsCollectedPercentage).

```
void UBehaviourAnalysis::InterpretClusterResult(TArray<TArray<float>>& Centroids, TArray<TArray<float>>& FCMMatrix)
{
    TArray<float> MeanObj;
    for(int32 i = 0; i < FCMMatrix[0].Num(); ++i)
    {
        float Membership = 0.f;
        for(int32 j = 0; j < FCMMatrix.Num(); ++j)
        {
            Membership += FCMMatrix[j][i];
        }
        MeanObj.Add( Item: Membership / FCMMatrix.Num());
    }

    TArray<BehaviourType> Interpretation;
    GetCentroidsInterpretation(Centroids, [&]Interpretation);

    for(int32 i = 0; i < Interpretation.Num(); ++i)
    {
        FuzzyClusteringResult[Interpretation[i]] = MeanObj[i];
    }
}
```

Рис. 2.4.18 – Функція отримання інтерпретації кластерів

Реалізовано інтерпретацію кластерів на C++ за допомогою базових

структур та операторів (рис. 2.4.18). Середнє значення обраховується з останньої матриці належності FCMMatrix. Функція GetCentroidsInterpretation бере центроїди кластерів, які є векторними представленнями середніх значень ознак для кожного кластеру. Таким чином, кожен кластер інтерпретується як найбільш близький до одного з типів гравців за Бартлом. Детальніше функцію GetCentroidsInterpretation можна переглянути в додатку Е.

2.4.5. Реалізація генетичного алгоритму характеристик кімнат

Після реалізованого каркасу будинку необхідно згенерувати наповнення кожної кімнати, звідки збираються описані дані для кластеризації. Це відбувається за допомогою генетичного алгоритму атрибутів кімнати. У переліку (enumeration) було визначено, що вона може мати такі характеристики:

- ETraitEnum::Treasure – скарби;
- ETraitEnum::Dialogue – неігровий персонаж з діалогом;
- ETraitEnum::Enemy – ворог;
- ETraitEnum::Trap – пастка;
- ETraitEnum::Light – джерело світла;
- ETraitEnum::Pillar – стовп;
- ETraitEnum::GlobalCollectable – колекційний предмет;
- ETraitEnum::Explore – предмет дослідження;
- ETraitEnum::ExploreBorder – предмет дослідження, який може бути розташований тільки уздовжд стін кімнати;
- ETraitEnum::Flashlight – батарейки для ліхтаря.

У класі URoomTraits заданий масив характеристик Traits. Для кожного екземпляру характеристики задана її вага (Weight, за замовчування – 1). Екземпляри такого класу є хромосомами генетичного алгоритму. У цьому класі визначені також основні операції схрещування, мутації, а також отримання значення функції допасованості.

Функція Crossover приймає як аргумент другий операнд цієї операції та

формує нового представника популяції. Кількість характеристик визначається як середнє арифметичне характеристик лівої і правої хромосоми. Самі характеристики обираються випадковим чином з усіх характеристик батьків (навіть, якщо вони повторюються) (рис. 2.4.19).

```
URoomTraits* URoomTraits::Crossover_Implementation(URoomTraits* Other)
{
    URoomTraits* Child = NewObject<URoomTraits>( Outer: this, Class: GetClass());

    const int32 TraitsCount = FMath::RoundToInt32( F: (Traits.Num() + Other->Traits.Num()) / 2.f);

    TArray<FTTNode> MergedTraits;
    MergedTraits.Append(Traits);
    MergedTraits.Append(Other->Traits);
    Algo::RandomShuffle( [&] MergedTraits);

    for(int32 i = 0; i < TraitsCount; ++i)
    {
        Child->Traits.Add(MergedTraits[i]);
    }
    Child->RoomType = RoomType;

    return Child;
}
```

Рис. 2.4.19 – Функція операції схрещування індивідів характеристик кімнати

Функція мутації Mutate (рис. 2.4.20) приймає ймовірність зростання та обрізання (ймовірність зміни визначається з $AlterChance = 1.0 - GrowChance - CutChance$), а також мінімальну а максимальну кількість ітерацій проведення операції. Окремо визначені функції Grow, Cut та Alter. Grow – додає до масиву Traits нову характеристику з випадковою вагою [0; 1], Cut – видаляє характеристику, якщо в масиві більше однієї, Alter – змінює наявну характеристику і її вагу на випадкову.

```cpp
void URoomTraits::Mutate_Implementation(float GrowChance, float CutChance,
    int32 MinIterationCount, int32 MaxIterationCount)
{
    const int32 IterationCount = FMath::RandRange(MinIterationCount, MaxIterationCount);

    for(int32 i = 0; i < IterationCount; ++i)
    {
        const float RandValue = FMath::RandRange(InMin: 0.f, InMax: 1.f);

        if(RandValue >= 0.f && RandValue < GrowChance)
        {
            Grow();
        }
        else if(RandValue >= GrowChance && RandValue < GrowChance + CutChance)
        {
            Cut();
        }
        else
        {
            Alter();
        }
    }
}
```

Рис. 2.4.20 – Функція операції мутації індивідів характеристик кімнати

Функція GetFitnessValue (рис. 2.4.21) повертає значення функції допасованості конкретного індивіда. Її формула виглядає так:

$$FitnessValue = \frac{\sum_{i=1}^{N} w_i \left(\frac{1}{M_i} \sum_{j=1}^{M_i} FCR(b_{ij})\right)}{\sum_{i=1}^{N} w_i},$$

де N – кількість характеристик індивіда, M_i – кількість поведінкових типів (4), пов’язаних з i-тою характеристикою, w_i – базова вага характеристики, $FCR(b_{ij})$ – міра належності до кластеру b_{ij} з FuzzyClusteringResult, $B_i = \{b_{i1}, b_{i2}, b_{i3}, b_{i4}\}$ – вектор мір належності характеристик до кластерів Бартла. Такий підхід розроблено для того, щоб забрати залежність пристосованості індивіда від кількості його характеристик, оскільки так, завдання би зводилося до вибору індивідів із найбільшою їхньою кількістю, та врахувати і вагу самої характеристики, і пов’язані з нею міри належності до кластерів.

```cpp
float URoomTraits::GetFitnessValue() const
{
    float FitnessValue = 0.f;
    float WeightSum = 0.f;

    for(const FTTNode& Node : Traits)
    {
        float BehaviourWeight = 0.f;
        int32 BehaviourCount = 0;

        for(FTraitsBehaviourTypes Types : TraitsBehaviourTypes)
        {
            if(Types.Trait == Node)
            {
                for(const BehaviourType Behaviour : Types.Behaviours)
                {
                    BehaviourWeight += FuzzyClusteringResult[Behaviour];
                    ++BehaviourCount;
                }
            }
        }
        BehaviourWeight /= BehaviourCount;

        FitnessValue += Node.Weight * BehaviourWeight;
        WeightSum += Node.Weight;

    }

    FitnessValue /= WeightSum;

    return FitnessValue;
}
```

Рис. 2.4.21 – Реалізація функції допасованості GetFitnessValue

Оскільки генетичний алгоритм потребує великої кількості індивідів для своєї роботи, запустити його з перших проходжень гри було би не раціонально. Тому спочатку характеристики кімнат формуються за замовчування через Blueprint нащадок класу URoomTraits. Там задана структура DefaultTraitsSpawnChance (рис. 2.4.22), яка увідповіднює кожну характеристику і її вагу з ймовірністю додання до індивіда. Характеристики, які використовуються для кімнат зберігаються та формують популяцію для подальших ітерацій генетичного алгоритму.

Рис. 2.4.22 – Задання структури DefaultTraitsSpawnChance. Характеристики Dialogue та Enemy мають ймовірність генерації 50%

Безпосередньо сам генетичний алгоритм виконується окремо для початкових, кінцевих і проміжних кімнат. У кінці обираються найкращі індивіди за функцією допасованості: одна початкова, одна кінцева і решту проміжних кімнат. Для класу UTraitGeneticAlgorithm створено BP нащадок, якому можна задавати параметри алгоритму: тип кімнат, клас індивідів кімнат, частка індивідів схрещення, ймовірності мутації зростання та обрізання. Селекція індивідів, які використовують для генетичних операцій відбувається методом рулетки.

Особливість етапу селекції полягає у використанні принципу «Exploration–Exploitation». Залежно від інформаційної ентропії метод рулетки враховує або пряме значення функції допасованості, або обернене (рис. 2.4.23). Таким чином, на початку гри генетичний алгоритм цілеспрямовано створює кімнати, не відповідні до поточних уподобань гравця. Це стимулює дослідження, вихід за межі зони комфорту – гравець змушений взаємодіяти з іншими типами контенту, що збагачує поведінковий профіль і дозволяє уникнути умовної локальності кластеризації. Після достатнього збору даних, коли розподіл нечітких кластерів виглядає більш чітко і стабільно, відбувається перехід у фазу адаптації до гравця – використовуючи коефіцієнти кластеризації як ваги, обираються найбільш релевантні варіанти кімнат.

```
void UTraitGeneticAlgorithm::RouletteWheelSelection(TArray<URoomTraits*>& OutPopulation)
{
    TArray<float> Probabilities;
    float FitnessSum = 0.f;

    for(const URoomTraits* RoomTraits : TraitsPopulation)
    {
        FitnessSum += bIsExploration ? (1.f / RoomTraits->GetFitnessValue()) : RoomTraits->GetFitnessValue();
    }

    float CumulativeProbability = 0.f;
    for(const URoomTraits* RoomTraits : TraitsPopulation)
    {
        Probabilities.Add(CumulativeProbability);
        CumulativeProbability += (bIsExploration ?
            (1.f / RoomTraits->GetFitnessValue()): RoomTraits->GetFitnessValue()) / FitnessSum;
    }
    Probabilities.Add(CumulativeProbability);

    for(int32 i = 0; i < TraitsPopulation.Num(); ++i)
    {
        const float RandomValue = FMath::RandRange(InMin: 0.f, InMax: 1.f);
        for(int32 j = 0; j < Probabilities.Num() - 1; ++j)
        {
            if(RandomValue >= Probabilities[j] && RandomValue < Probabilities[j + 1])
            {
                OutPopulation.Add(Item: TraitsPopulation[j]->GetCopy(this));
            }
        }
    }
}
```

Рис. 2.4.23 – Функція селекції RouletteWheelSelection методом рулетки

Пороговий рівень ентропії для генетичного алгоритму встановлений на рівні 1.9. Якщо він більший за це значення відбувається етап Exploration, інакше – наступає Exploitation (рис. 2.4.24). Решту коду реалізованого генетичного алгоритму можна переглянути в додатку Є.

```cpp
bool UTraitGeneticAlgorithm::GetExplorationExploitationType(UBehaviourAnalysis* Behaviour)
{
    if(Behaviour == nullptr)
    {
        return true;
    }

    float Entropy = 0.f;

    TArray<BehaviourType> BehaviourTypes;
    Behaviour->FuzzyClusteringResult.GetKeys([&]BehaviourTypes);

    TArray<float> BehaviourCoefficients;
    for(const BehaviourType BType : BehaviourTypes)
    {
        BehaviourCoefficients.Add(Behaviour->FuzzyClusteringResult[BType]);
    }

    for(const float Coefficient : BehaviourCoefficients)
    {
        Entropy += Coefficient * FMath::Log2(Coefficient);
    }
    Entropy *= -1.f;

    return Entropy > EntropyExploitationLevel;
}
```

Рис. 2.4.24 – Визначення етапу «Exploration–Exploitation» за рівнем інформаційної ентропії

Інтерпретація цих характеристик у фізичний формат відбувається за допомогою дата асету PDA_BSPTreeMain, де задана структура FTraitSpawnChance, у якій вказана характеристика і масив можливих акторів генерації з їхньою вагою спавну (рис. 2.4.25).

Рис. 2.4.25 – Інтерпретація характеристики Enemy – актори спавняться з однаковим шансом, оскільки мають однакову вагу

Отже, генетичний алгоритм характеристик кімнат має наступні кроки:

1. Перевірка поточної популяції. Якщо вона вже має потрібну кількість найкращих особин, то зберегти їх одразу у вихідних характеристиках і завершити алгоритм.
2. Підвантаження збережених індивідів певного типу кімнат.

3. Селекція індивідів для генетичних операцій:

3.1. Визначення етапу «Exploration–Exploitation».

3.2. Якщо відбувається етап Exploration рахувати значення функції допасованості індивіда як $f_{fitness} = \sum_{i=1}^{N} \frac{1}{f_i}$, де N – кількість характеристик в індивіді, f_i – значення функції допасованості i-того індивіда. Якщо відбувається етап Exploitation – $f_{fitness} = \sum_{i=1}^{N} f_i$.

3.3. Методом рулетки обрати індивідів за значенням їхніх функцій $f_{fitness}$.

4. Випадковий поділ обраних індивідів на тих, які пройдуть схрещування та тих, які пройдуть мутацію.

5. Схрещування. Якщо кількість індивідів непарна, остання особина схрещується з першою. Нові нащадки додаються у загальну популяцію.

6. Мутація. Мутовані особини замінюють старі у популяції.

7. Вибір найкращих індивідів методом елітизму.

8. Збереження обраних індивідів.

Можна помітити, що індивіди не кодуються для генетичних операцій, як це робиться в класичному варіанті ГА. Тут особини представлені безпосередньо масивом характеристик. Такий підхід хоч і позбавлений певної уніфікації, абстракції та теоретичного аналізу, але добре працює із природою цих даних – характеристиками кімнат, що забезпечує виграш у практичності та робить алгоритм ближчим до «об’єктно-орієнтованого» генетичного алгоритму.

2.4.6. Результати дослідження процедурної генерації світу

Розроблена PCG модель дозволила дослідити процедурну генерацію світу в контексті адаптації її до поведінкових патернів гравця. Обраховані кластери визначають характеристики кімнат наступного проходження більшою мірою, однак якщо є атрибути, які важливіші (мають в рази більшу вагу) за всі інші, вони ймовірно будуть відібрані, незалежно від обмежень зі сторони результатів кластеризації. На рисунка 2.4.26-2.4.29 зображені результати генерації для чотирьох різних випадків генетичного алгоритму, але однакового BSP дерева.

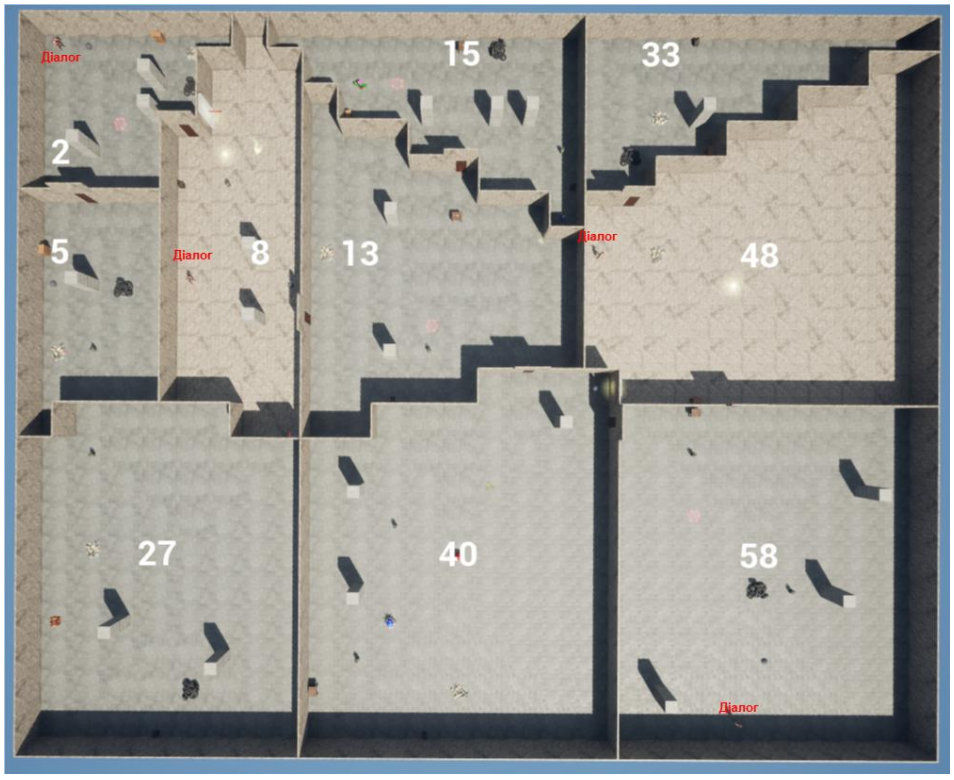

Рис. 2.4.26 – Генерація при розподілі кластерів 0.7 для «соціальника» і 0.1 – інші

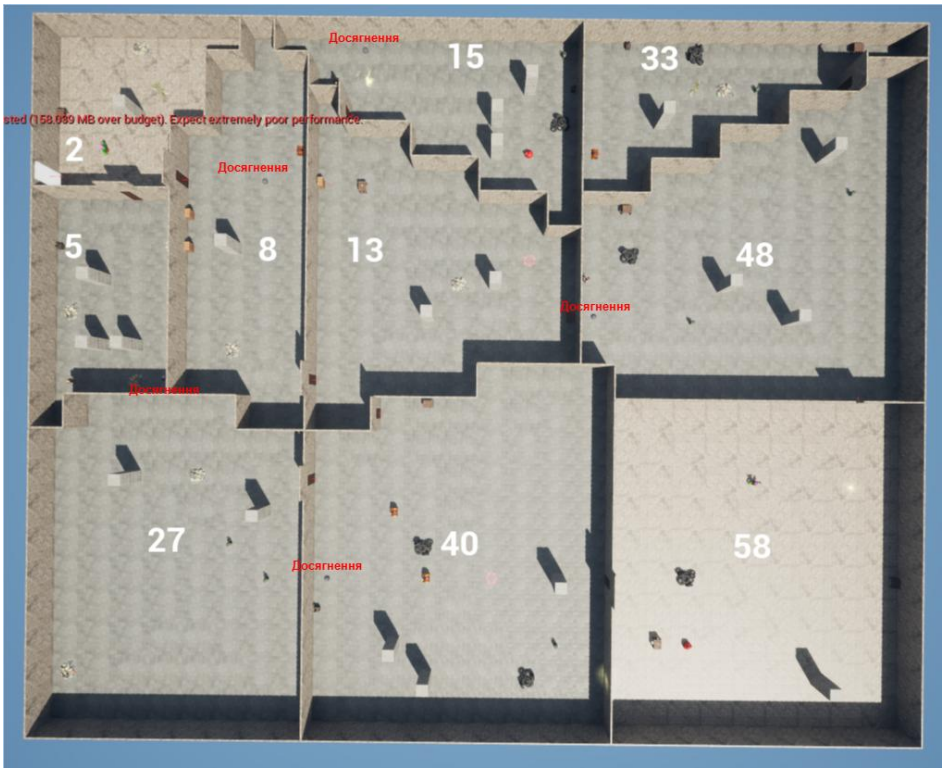

Рис. 2.4.27 – Генерація при розподілі кластерів 0.7 для «досягальника» і 0.1 – інші

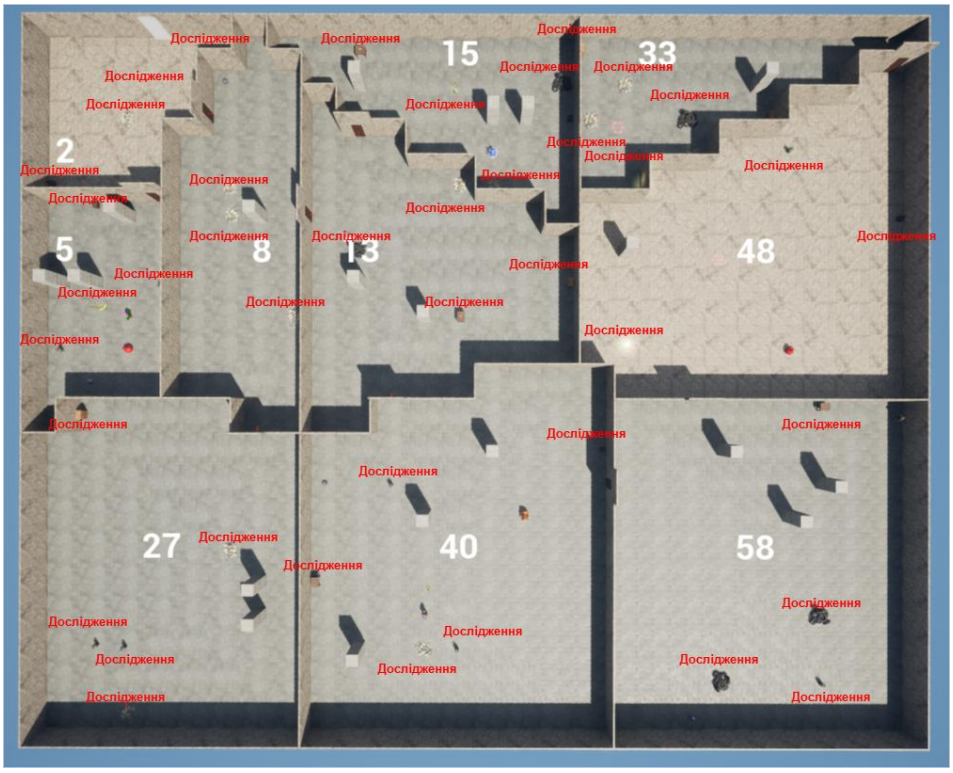

Рис. 2.4.28 – Генерація при розподілі кластерів 0.7 для «дослідника» і 0.1 – інші

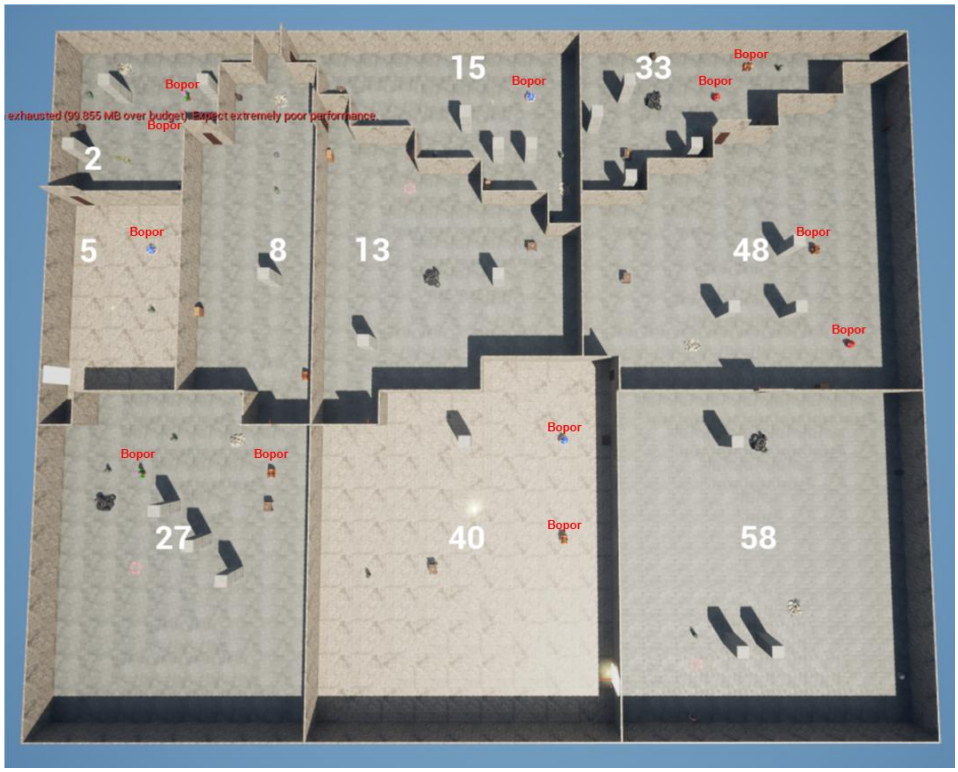

Рис. 2.4.29 – Генерація при розподілі кластерів 0.7 для «кілера» і 0.1 – інші

Нерівномірність розподілу генерації між різними варіантами кластеризації полягає у різних особистих вагах характеристик. Ваги атрибутів Dialogue та GlobalCollectable природньо менші ніж Enemy чи Explore/ExploreBorder.

Модифікація до алгоритму на основі BSP дерева показала, як можна використати старий підхід і для трохи інших ситуацій. Таким чином, за допогою простої черги пост-злиття вдалося уникнути проблем із замкненими кімнатами.

Використання інформаційної ентропії Шеннона допомогла визначити етапи формування даних так, щоб вони були різноплановими. Це важливо оскільки може статися ситуація, коли гравець із самого початку буде поводитися однотипно, що призведе до визначення кластерів неправдиво – занадто локально. Наприклад, дані із високими показниками відсотка зібраних скарбів і пройденої території були у всіх проходженнях. Тоді до кластеру «дослідника» попадуть дані з найвищими значеннями, а до інших – просто високими. Якщо ж після цього гравець зіграє абсолютно інакшим шаблоном, кластеризація може визначити це занадто жорстко. Такий підхід балансує цю ситуацію.

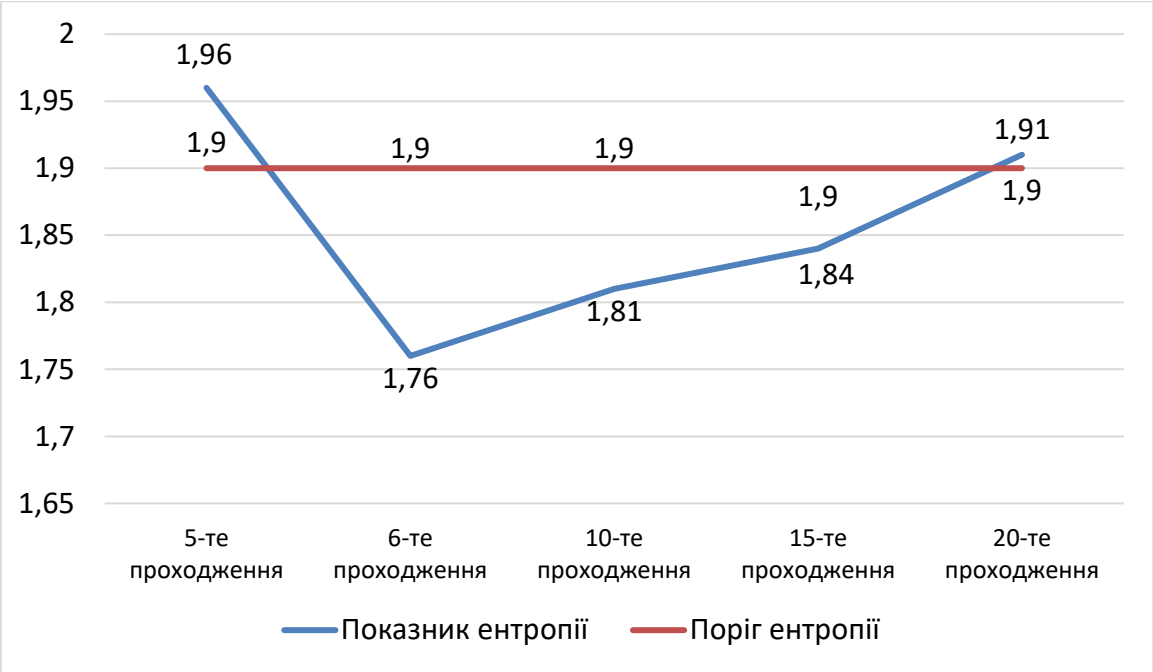

Рис. 2.4.30 – Зміна показника ентропії протягом 20 проходжень гри

На рис. 2.4.30 зображено як змінювалося значення ентропії впродовж 20 проходжень гри з встановленим порогом етапу «Exploration» 1,9. Після перемикання з режиму «Exploration» на «Exploitation» у 6-ому проходженні, ентропія помітно зменшилася, що логічно, оскільки тепер кластери формуються за вподобаннями гравця. Не швидкими темпами вона збільшується, тому на 21-ому проходженні гра знову перемкнеться на дослідження профілю поведінки гравця.

ВИСНОВКИ

Під час дослідження процедурної генерації світу було проаналізовано широкий спектр сучасних алгоритмів, методів та підходів, які є основоположними у PCG сьогодні, а також розроблена нова модель процедурної генерації.

У загальних рисах розглянуті алгоритми генерації на основі генеративної граматики, градієнтних шумів та діаграми Вороного. Вони показали основні напрямки та сфери роботи процедурної генерації – створення самоподібних структур, розподілу простору, біомів, рельєфних пейзажів. Детальніше розглянуто робота алгоритму на основі BSP дерева та генетичного алгоритму. Показано, як BSP дерево ставало все більш універсальним інструментом – перше застосування у комп'ютерній графіці трансформувалося в алгоритм генерації 2D кімнат і коридорів. Щодо генетичного алгоритму, розглянути його основні засади, конкретні етапи роботи та сфери застосування. Окрім цього, проаналізовано роботу класичного алгоритму нечіткої кластеризації Fuzzy C-Means, класифікацію Бартла (поведінки гравців), принцип навчання з підкріпленням «Exploration–Exploitation» та інформаційну ентропію Шеннона.

На основі такої широкої бази була розроблена модель процедурної генерації розподіленого будинку та різними характеристиками кожної кімнати. Реалізація відбувалася на ігровому двигуні Unreal Engine версії 5.4 з додатковим плагіном PCG Plugin. Використано мову програмування C++ та графічний інструмент скриптування Blueprints. Таким чином, була розроблена модифікація до класичного алгоритму на основі BSP дерева, яка передбачає трактування вузлів дерева не просто, як розподілений простір, а центр цього простору. Це призвело до того, що етап злиття став не таким прямолінійним як раніше, тому довелося ввести чергу пост-злиття. Окрім цього, випадкова генерація BSP дерева організована так, щоб задати початкову і кінцеву кімнати проходження гри, без фізичного представлення будинку. Вибір наповнення кімнат проводилося за допомогою генетичного алгоритму на основі нечітко кластеризованих даних

гравця. Використання принципу «Exploration–Exploitation» та інформаційної ентропії Шеннона допомогла збалансувати цей процес, що й проаналізовано в результатах дослідження.

Розроблена модель неодмінно показує високу адаптивність та ефективність своєї роботи, однак є певні точки покращення її охоплення та роботи:

1. Модель не визначає автоматично кількість кімнат і форму будинку (TransformationCoefficient), однак це можна зробити, спираючись на проаналізовану поведінку гравця.
2. Розподіл атрибутів кластеризованих даних нерівномірний – різна кількість характеристик відповідає за той чи інший тип поведінки. Це не критична проблема, бо балансується заданою формулою функції допасованості, однак підхід може бути покращений.
3. Можна спробувати реалізувати залежність частки індивідів, які пройдуть схрещування та мутацію, від ентропії поведінки гравця. Якщо гравець дуже передбачуваний, відсоток мутацій збільшується, щоб виходити за межі та дивувати.

Окрім цього, модель відкрита для експериментів з іншими методами кластеризації, інтеграції з новими підходами та запровадження використання навчання з підкріпленням для формування більш складного розподілу простору.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Guzdial M., Snodgrass S., Summerville A. J. Procedural Content Generation via Machine Learning. Cham : Springer International Publishing, 2022. URL: https://doi.org/10.1007/978-3-031-16719-5.
2. Harris J. Exploring Roguelike Games. CRC Press, 2020. 556 p.
3. Laitaruk I., Hryshanovych T. Overview of modern algorithms for world procedural generation in computer games. *CEUR Workshop Proceedings*. 2024. P. 152–163. URL: https://cssesw.easyscience.education/cssesw2024/CSSESW2024/paper21.pdf.
4. Rozenberg G. Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Pub Co Inc, 1997. 553 p.
5. Graph Rewriting for Procedural Level Generation. *BorisTheBrave.Com*. URL: https://www.boristhebrave.com/2021/04/02/graph-rewriting/.
6. Procedural Noise/Categories. *PhysBAM*. URL: https://physbam.stanford.edu/cs448x/old/Procedural_Noise(2f)Categories.html.
7. Gonzalez Vivo P., Lowe J. The Book of Shaders. *The Book of Shaders*. URL: https://thebookofshaders.com/11/.
8. Dobrin A. A review of properties and variations of voronoi diagrams. URL: https://www.whitman.edu/Documents/Academics/Mathematics/dobrinat.pdf.
9. Vassilev T. S. Generalizations of the Voronoi Diagram. *American Journal of Computational and Applied Mathematics*. 2013. P. 91–96. URL: https://www.researchgate.net/publication/270161065_Generalizations_of_the_Voronoi_Diagram.
10. Naylor B. F. Constructing Good Partitioning Trees. *AT&T Bell Laboratories*. 2001. URL: https://www.researchgate.net/publication/2492209_Constructing_G

ood_Partitioning_Trees.

11. Uribe G. Dungeon Generation using Binary Space Trees. *Medium*. URL: https://medium.com/@guribemontero/dungeon-generation-using-binary-space-trees-47d4a668e2d0.
12. Dungeon generation using BSP trees. *eskerda.com | your daily cup of tea*. URL: https://eskerda.com/bsp-dungeon-generation/.
13. Personal website of Antonios Liapis / N. Shaker et al. *Personal website of Antonios Liapis*. URL: https://antoniosliapis.com/articles/pcgbook_dungeons.php.
14. Genetic Algorithm in AI | Operators | Working | Gate Vidyalay. *Gate Vidyalay | A temple of learning for GATE, NET, PSU's*. URL: https://www.gatevidyalay.com/genetic-algorithm-in-ai-operators-working/ (date of access: 25.08.2025).
15. GeeksforGeeks. Genetic Algorithms. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/dsa/genetic-algorithms/ (date of access: 25.08.2025).
16. GA Roulette wheel selection. *EDC: Design, development and performance*. URL: http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php (date of access: 25.08.2025).
17. Кононюк А.Ю. Нейронні мережі і генетичні алгоритми. Київ : Корнійчук, 2008. 446 с. URL: https://pdf.lib.vntu.edu.ua/books/2016/Kononyk_2008_470.pdf.
18. GeeksforGeeks. Fuzzy Clustering - ML. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/machine-learning/ml-fuzzy-clustering/.
19. Mahesh Huddar. Fuzzy C Means Clustering Algorithm Solved Example | Clustering Algorithm in ML & DL by Mahesh Huddar, 2023. *YouTube*. URL: https://www.youtube.com/watch?v=X7co6-U4BJY.
20. Механіки утримання в казуальних іграх: як залучати користувачів із

різними психотипами • VOKI Games. *Voki Games*. URL: https://vokigames.com/ua/mehaniky-utrymannya-v-kazualnyh-igrah-yak-zaluchaty-korystuvachiv-iz-riznymy-psyhotypamy/.

21. Gamify's Official Blog. *Play meets purpose. Gamified experiences that drive impact*. URL: https://www.gamify.com/gamification-blog/the-bartle-test-of-gamer-psychology-gamer-types.
22. Stuart K. Richard Bartle: we invented multiplayer games as a political gesture. *the Guardian*. URL: https://www.theguardian.com/technology/2014/nov/17/richard-bartle-multiplayer-games-political-gesture.
23. Rocca J. The exploration-exploitation trade-off: intuitions and strategies. *Medium*. URL: https://medium.com/data-science/the-exploration-exploitation-dilemma-f5622fbe1e82.
24. Shannon Entropy. *ScienceDirect*. URL: https://www.sciencedirect.com/topics/engineering/shannon-entropy.
25. Farrokhi Maleki M., Zhao R. Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration. *arXiv.org e-Print archive*. URL: https://arxiv.org/html/2410.15644v1.
26. Merrell P. Example-Based Procedural Modeling Using Graph Grammars. *ACM Transactions on Graphics (TOG)*. 2023. P. 1–16. URL: https://doi.org/10.1145/3592119.
27. Haubenwallner K., Seidel H.-P., Steinberger M. ShapeGenetics: Using Genetic Algorithms for Procedural Modeling. *Computer Graphics Forum 2017 The Eurographics Association and John Wiley & Sons Ltd*. 2017. URL: https://www.researchgate.net/publication/317749627_ShapeGenetics_Using_Genetic_Algorithms_for_Procedural_Modeling.
28. Guns, Swords and Data: Clustering of Player Behavior in Computer Games in the Wild / A. Drachen et al. *IEEE Computational Intelligence in Games*. 2012.
URL: https://www.researchgate.net/publication/250305990_Guns_Swords

_and_Data_Clustering_of_Player_Behavior_in_Computer_Games_in_the_Wild_Pre-print.

29. Das S., De S. A Modified Genetic Algorithm Based FCM Clustering Algorithm for Magnetic Resonance Image Segmentation. *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*. 2017. P. 435–443. URL: https://www.researchgate.net/publication/315330523_A_Modified_Genetic_Algorithm_Based_FCM_Clustering_Algorithm_for_Magnetic_Resonance_Image_Segmentation.
30. Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. *Unity*. URL: https://unity.com/.
31. Yadav A. Exploring Procedural Generation in Unity: Tools and Techniques. *Medium*. URL: https://medium.com/@yadavaman/exploring-procedural-generation-in-unity-tools-and-techniques-e0f16e88e2f2.
32. Godot Engine - Free and open source 2D and 3D game engine. *Godot Engine*. URL: https://godotengine.org/.
33. Procedural geometry. *Godot Engine documentation*. URL: https://docs.godotengine.org/en/4.4/tutorials/3d/procedural_geometry/index.html.
34. Unreal Engine 5.4 Documentation. *Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-4-documentation?application_version=5.4.
35. Procedural Content Generation Overview. *Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-overview.
36. Procedural Content Generation Framework. *Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-framework-in-unreal-engine.
37. Rider – the ultimate IDE for Unreal Engine. *JetBrains: Developer Tools for Professionals and Teams*. URL: https://www.jetbrains.com/lp/rider-unreal/.

38. Visual Studio 2022 IDE - AI for coding debugging and testing. *Visual Studio*. URL: https://visualstudio.microsoft.com/vs/.

ДОДАТКИ

Додаток А

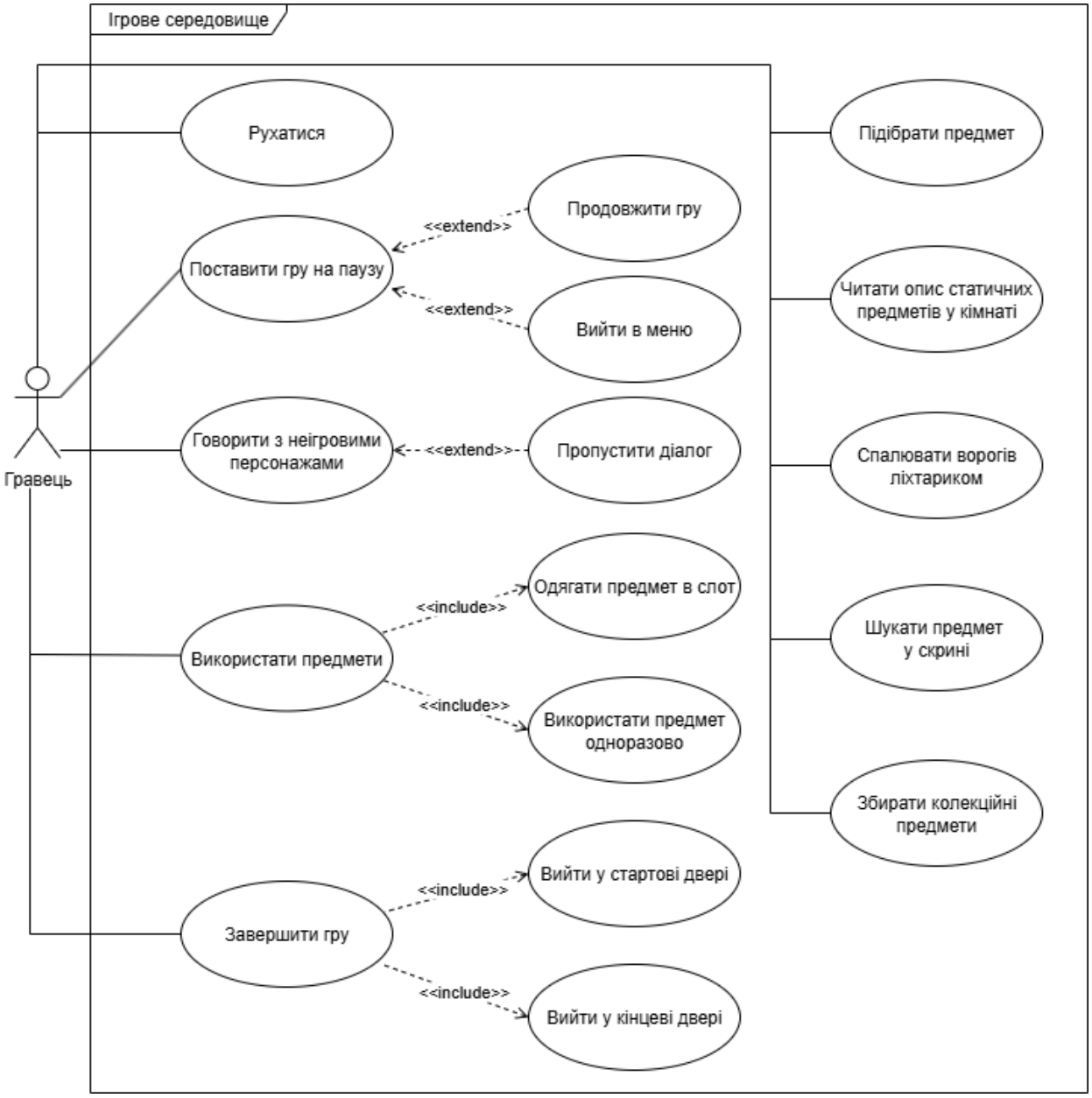

Рис. А.1 – Діаграма прецедентів розроблювальної гри

Додаток Б

Рис. Б.1 – Діаграма послідовності процесу процедурної генерації світу

Додаток В

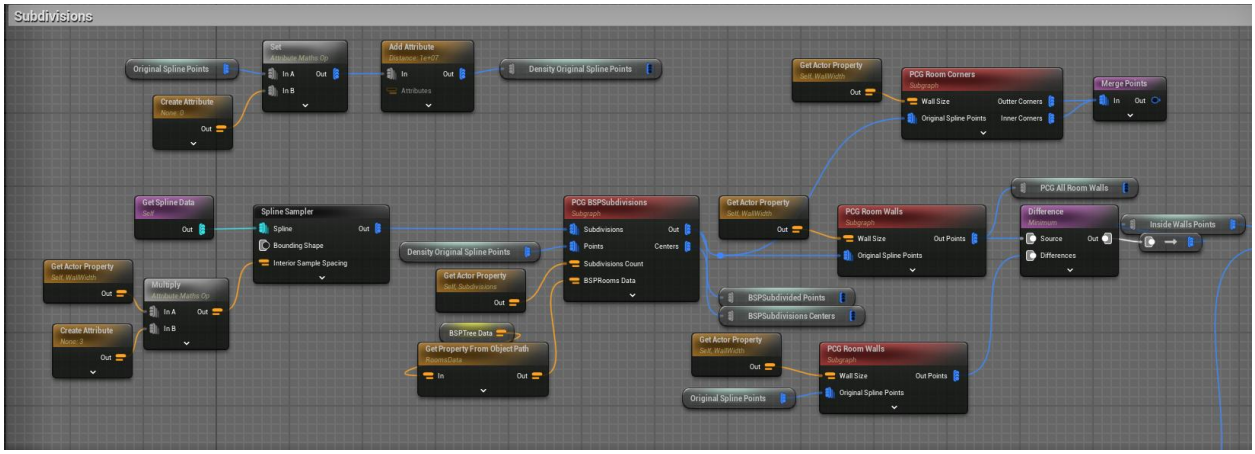

Рис. В.1 – Граф PCG_RoomMain з логікою побудови кімнат

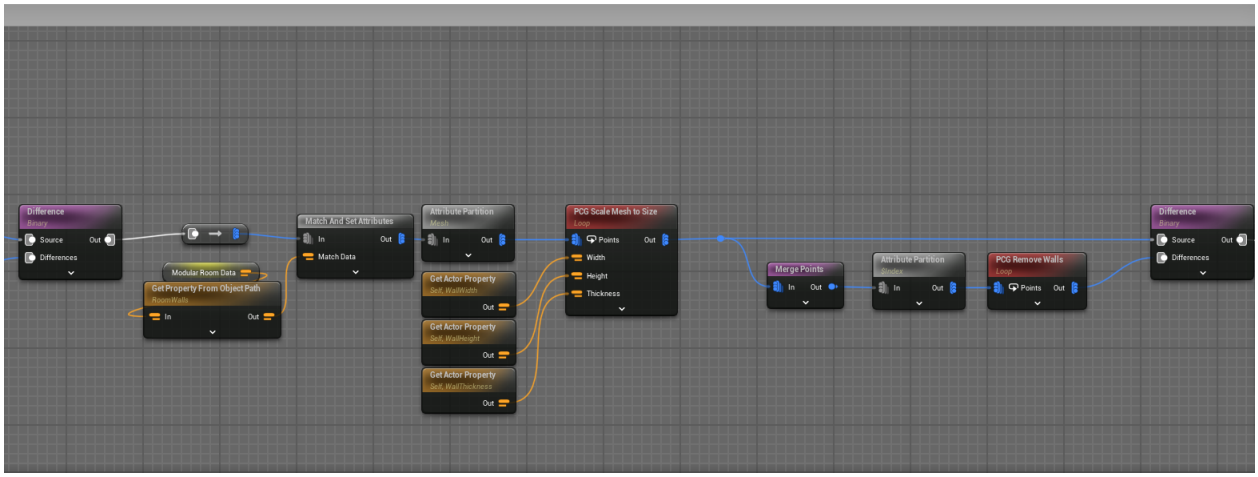

Рис. В.2 – Граф PCG_RoomMain з логікою побудови кімнат. Продовження 1

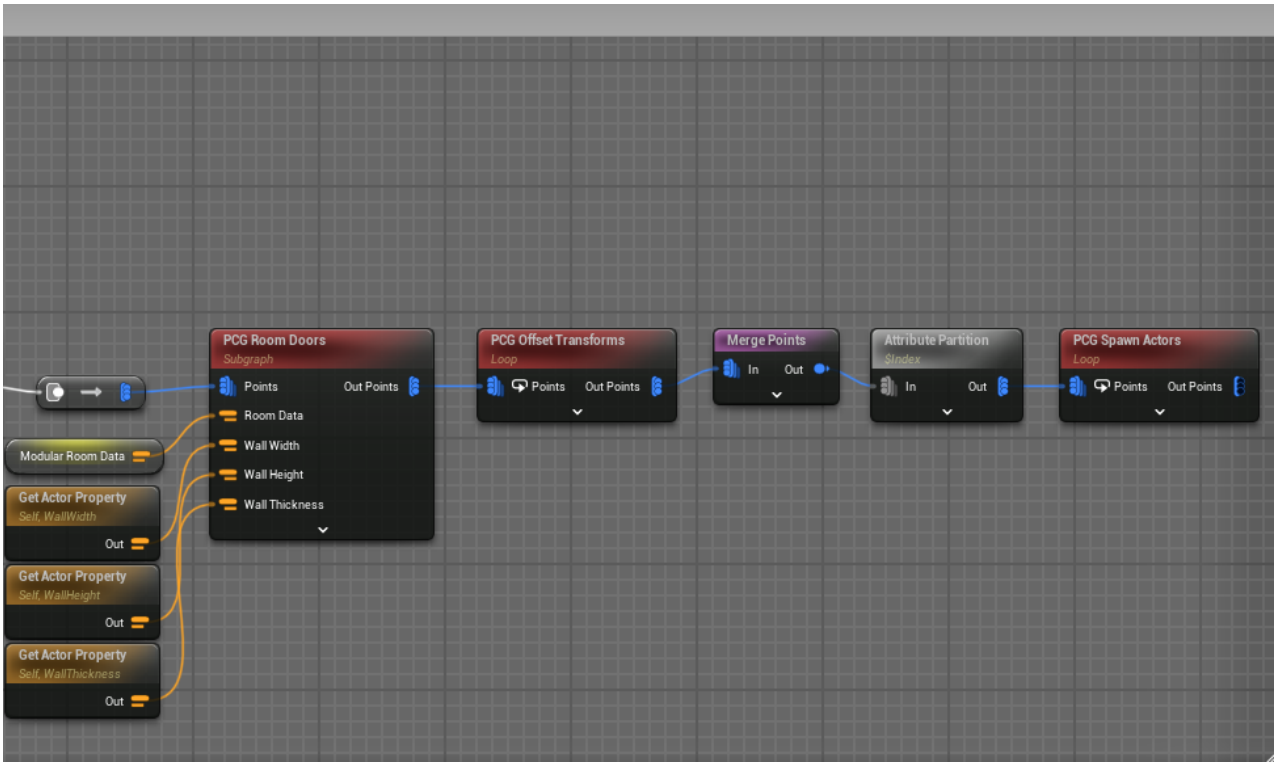

Рис. В.3 – Граф PCG_RoomMain з логікою побудови кімнат. Продовження 2

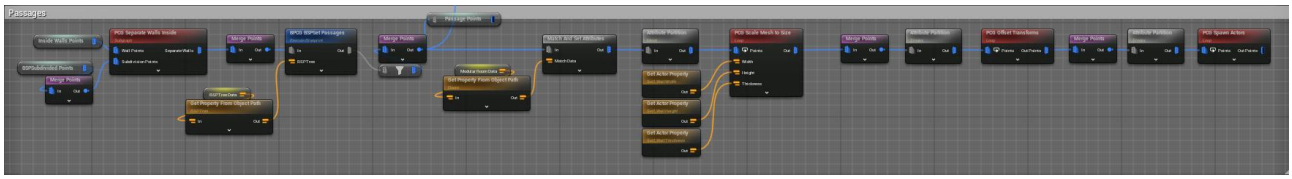

Рис. В.4 – Граф PCG_RoomMain з логікою пошуку проходів між кімнатами

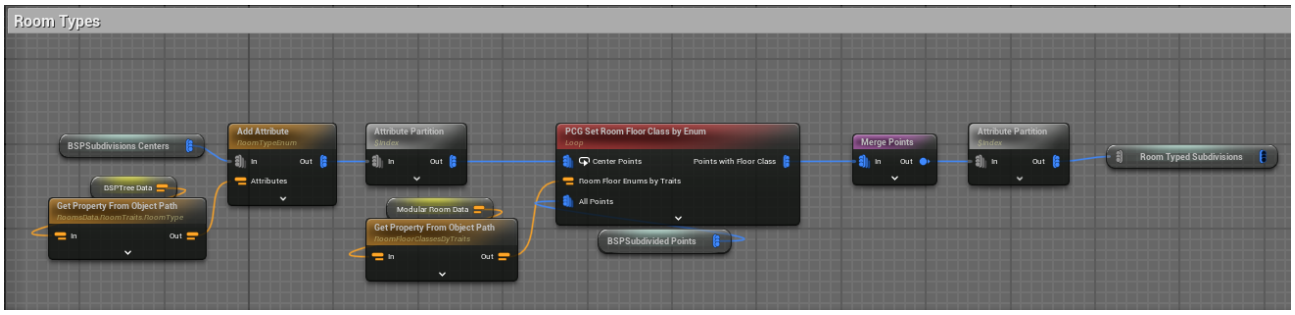

Рис. В.5 – Граф PCG_RoomMain з логікою отримання типів кімнат

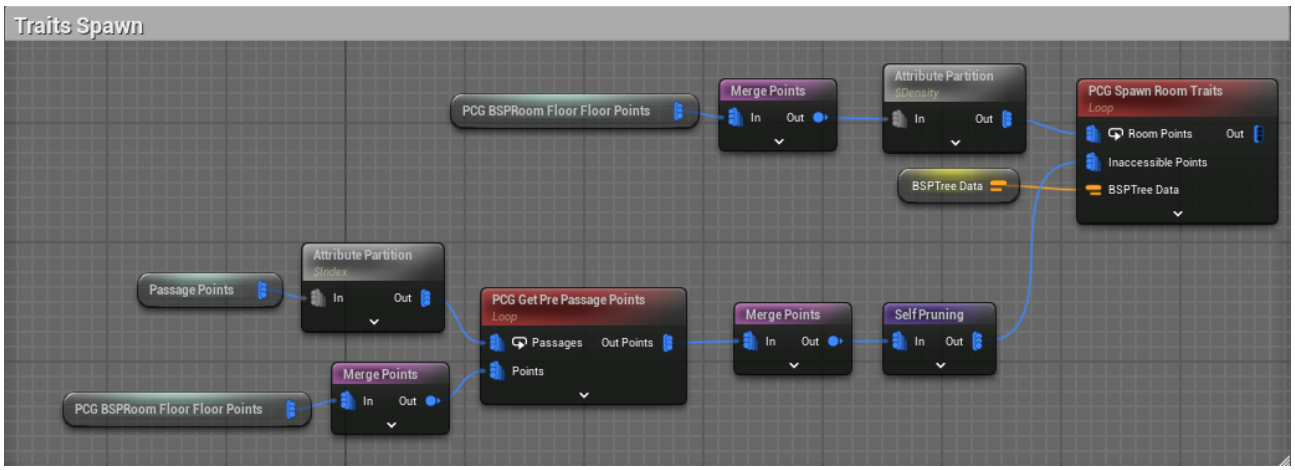

Рис. В.6 – Граф PCG_RoomMain з логікою генерації характеристик кімнат

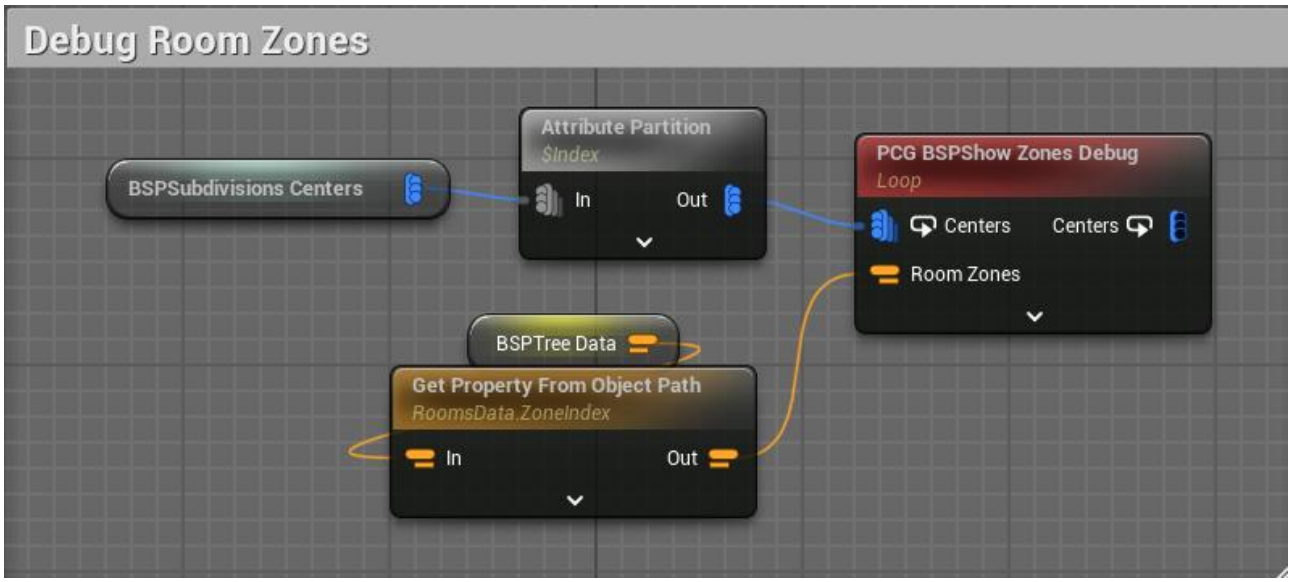

Рис. В.7 – Граф PCG_RoomMain з логікою відлагодження сформованих центрів кімнат

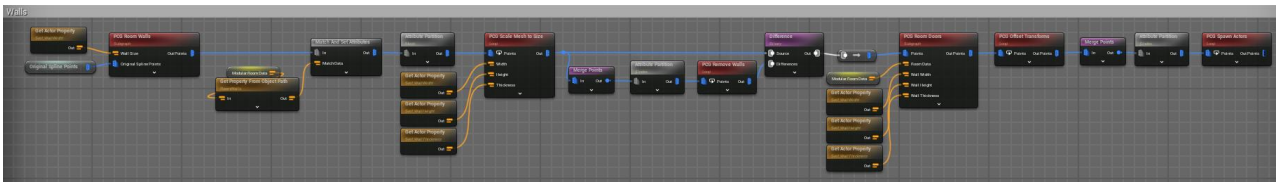

Рис. В.8 – Граф PCG_RoomMain з логікою генерації стін

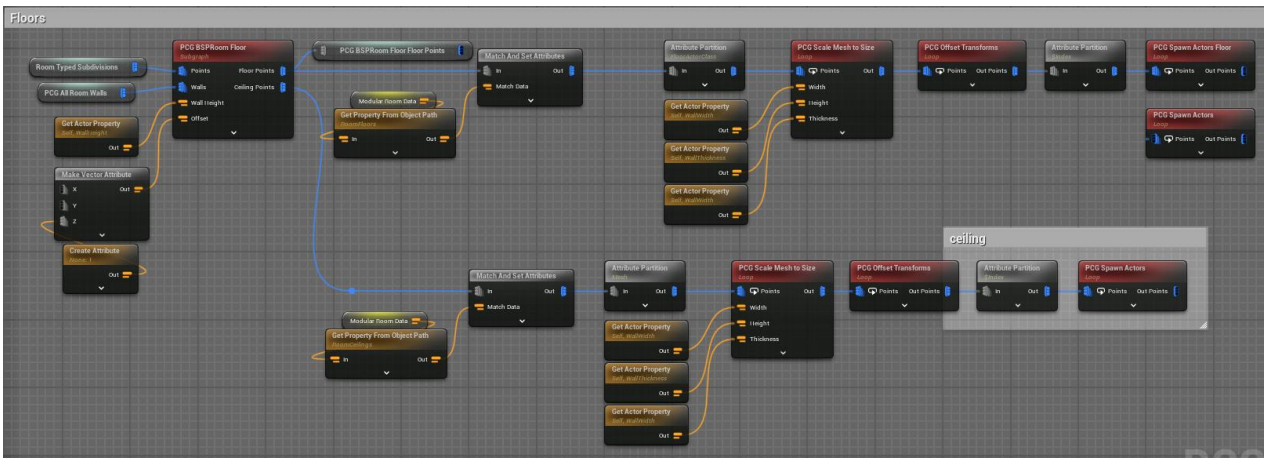

Рис. В.9 – Граф PCG_RoomMain з логікою генерації підлоги і стелі

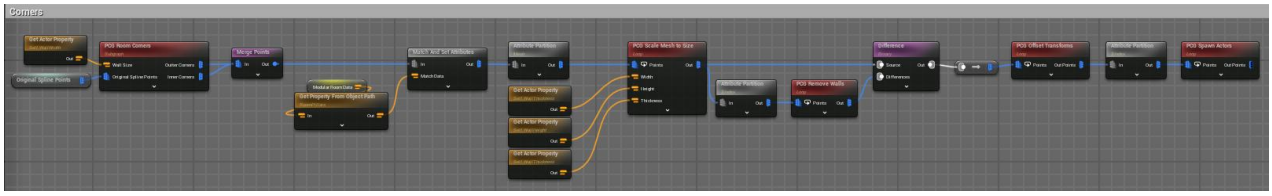

Рис. В.10 – Граф PCG_RoomMain з логікою генерації кутів

Додаток Г

```r
library(jsonlite)
library(e1071)
library(ggplot2)
library(dplyr)
library(tidyr)
library(ggthemes)


data <- fromJSON("D:\\HorrorGame\\HorrorGame\\Saved\\PersistentDownloadDir\\behaviourdata.json")
data <- as.data.frame(data)

k <- 4

fcm_result <- cmeans(data, centers = k, m = 1.2, method = "cmeans")

print(fcm_result$membership)

pca <- prcomp(data, scale. = TRUE)
pca_data <- as.data.frame(pca$x)

membership <- as.data.frame(fcm_result$membership)
colnames(membership) <- paste0("Cluster_", 1:4)

pca_with_membership <- bind_cols(pca_data, membership)

plot_data <- pca_with_membership %>%
  pivot_longer(cols = starts_with("Cluster_"),
               names_to = "Cluster",
               values_to = "Membership")

plot_pca_clusters <- function(pc_x = 1, pc_y = 2)
{
  pc_x_name <- paste0("PC", pc_x)
  pc_y_name <- paste0("PC", pc_y)

  plot(ggplot(plot_data, aes_string(x = pc_x_name, y = pc_y_name, color = "Membership")) +
         geom_point(size = 2, alpha = 0.8) +
         scale_color_gradient(low = "grey90", high = "red") +
         facet_wrap(~ Cluster) +
         theme_minimal(base_size = 14) +
         labs(title = paste0("Міра належності до кластеру у PCA-просторі (PC", pc_x, "/PC", pc_y, ")"),
              color = "Належність",
              x = paste0("PC", pc_x), y = paste0("PC", pc_y)))
}

plot_pca_clusters(1, 2)

print(colMeans(fcm_result$membership))
```

Рис. Г.1 – Код на R для кластеризації поведінкових даних гравця

Додаток Д

```
void UBehaviourAnalysis::Execute()
{
    if(BehaviourData.Num() < MinCountForClustering)
    {
        return;
    }

    TArray<TArray<float>> FCMMatrix;
    TArray<TArray<float>> Centroids;
    FuzzyCMeans( [&] FCMMatrix,  [&] Centroids,  ClustersCount: 4);
    InterpretClusterResult( [&] Centroids,  [&] FCMMatrix);
}
```

Рис. Д.1 – Функція входу у виконання кластеризації

```
void UBehaviourAnalysis::InitializeFCMMatrix(TArray<TArray<float>>& FCMMatrix, int32 ClusterCount)
{
    FCMMatrix.SetNum(BehaviourData.Num());
    for(int32 i = 0; i < FCMMatrix.Num(); ++i)
    {
        FCMMatrix[i].SetNum(ClusterCount);
        float LeftMembership = 1.f;
        for(int32 j = 0; j < FCMMatrix[i].Num() - 1; ++j)
        {
            const float Membership = FMath::RandRange( InMin: 0.f,  InMax: LeftMembership);
            LeftMembership -= Membership;

            FCMMatrix[i][j] = Membership;
        }
        FCMMatrix[i][FCMMatrix[i].Num() - 1] = LeftMembership;
    }
}
```

Рис. Д.2 – Функція ініціалізації матриці приналежності даних

```cpp
void UBehaviourAnalysis::CalculateCentroids(TArray<TArray<float>>& OutCentroids,
    const TArray<TArray<float>>& FCMMatrix)
{
    OutCentroids.Empty();

    TArray<TArray<float>> BehaviourDataRow;
    GetRawBehaviourData([&]BehaviourDataRow);

    for(int32 i = 0; i < FCMMatrix[0].Num(); ++i)
    {
        float MembershipSum = 0.f;
        for(int32 j = 0; j < FCMMatrix.Num(); ++j)
        {
            MembershipSum += FMath::Pow(A:FCMMatrix[j][i], B:FuzzinessParameter);
        }

        TArray<float> RawCentroid;
        for(int32 j = 0; j < BehaviourDataRow[0].Num(); ++j)
        {
            float MembershipDataSum = 0.f;
            for(int32 k = 0; k < FCMMatrix.Num(); ++k)
            {
                MembershipDataSum += FMath::Pow(A:FCMMatrix[k][i], B:FuzzinessParameter) * BehaviourDataRow[k][j];
            }
            RawCentroid.Add(Item:MembershipDataSum / MembershipSum);
        }

        OutCentroids.Add(RawCentroid);

    }
}
```

Рис. Д.3 – Функція обрахування центроїдів за матрицею приналежності

```cpp
void UBehaviourAnalysis::CalculateDistancesToCentroids(TArray<TArray<float>>& OutDistances,
    const TArray<TArray<float>>& Centroids)
{
    TArray<TArray<float>> RawBehaviourData;
    GetRawBehaviourData([&]RawBehaviourData);

    for(int32 i = 0; i < RawBehaviourData.Num(); ++i)
    {
        TArray<float> Distance;
        for(int32 j = 0; j < Centroids.Num(); ++j)
        {
            Distance.Add(Item:GetEuclideanDistance(Start:RawBehaviourData[i], Finish:Centroids[j]));
        }
        OutDistances.Add(Distance);
    }
}
```

Рис. Д.4 – Функція обрахування відстаней до центроїдів

```cpp
void UBehaviourAnalysis::CalculateNewFCMMatrix(TArray<TArray<float>>& NewFCMMatrix,
    const TArray<TArray<float>>& DistancesToCentroids)
{
    NewFCMMatrix.Empty();
    for(int32 i = 0; i < DistancesToCentroids.Num(); ++i)
    {
        TArray<float> NewMembershipValues;
        for(int32 j = 0; j < DistancesToCentroids[i].Num(); ++j)
        {
            float MembershipValue = 0.f;

            for(int32 k = 0; k < DistancesToCentroids[i].Num(); ++k)
            {
                MembershipValue +=
                    (DistancesToCentroids[i][j] * DistancesToCentroids[i][j]) /
                        (DistancesToCentroids[i][k] * DistancesToCentroids[i][k]);
            }
            MembershipValue = FMath::Pow( A: MembershipValue, B: 1.f/(FuzzinessParameter - 1.f));
            MembershipValue = FMath::Pow( A: MembershipValue, B: -1.f);

            NewMembershipValues.Add(MembershipValue);
        }
        NewFCMMatrix.Add(NewMembershipValues);
    }
}
```

Рис. Д.5 – Функція обрахування нової матриці приналежності

```cpp
float UBehaviourAnalysis::GetEuclideanDistance(const TArray<float>& Start, const TArray<float>& Finish)
{
    float QuadDistance = 0.f;
    for(int32 i = 0; i < Start.Num() && Finish.Num(); ++i)
    {
        QuadDistance += (Finish[i] - Start[i]) *  (Finish[i] - Start[i]);
    }

    return FMath::Sqrt(QuadDistance);
}
```

Рис. Д.6 – Функція обрахування евклідової відстані

```cpp
float UBehaviourAnalysis::GetTolerance(const TArray<TArray<float>>& Old, const TArray<TArray<float>>& New)
{
    float Eps = 0.f;

    for(int32 i = 0; i < Old.Num() && i < New.Num(); ++i)
    {
        for(int32 j = 0; j < Old[i].Num() && j < New[i].Num(); ++j)
        {
            Eps += (Old[i][j] - New[i][j]) * (Old[i][j] - New[i][j]);
        }
    }

    return FMath::Sqrt(Eps);
}
```

Рис. Д.7 – Функція обрахування допустимого значення за старою і новою матрицями приналежності

Додаток Е

```
void UBehaviourAnalysis::GetCentroidsInterpretation(const TArray<TArray<float>>& Centroids,
    TArray<BehaviourType>& OutInterpretation)
{
    for(int32 i = 0; i < Centroids.Num(); ++i)
    {
        TMap<int32, float> IndexToValueCentroid;
        for(int32 j = 0; j < Centroids[i].Num(); ++j)
        {
            IndexToValueCentroid.Add( InKeyValue: TPair<int32, float>( [&] j, Centroids[i][j]));
        }
        IndexToValueCentroid.ValueSort( Predicate: [](float A, float B) ->bool
        {
            return A > B;
        });

        TArray<int32> Indexes;
        IndexToValueCentroid.GetKeys( [&] Indexes);

        TMap<BehaviourType, float> ClusterProbability = {
            TPair<BehaviourType, float>(BehaviourType::Killer, 0.f),
            TPair<BehaviourType, float>(BehaviourType::Explorer, 0.f),
            TPair<BehaviourType, float>(BehaviourType::Achiever, 0.f),
            TPair<BehaviourType, float>(BehaviourType::Socializer, 0.f)
        };
```

Рис. Е.1 – Початок функції GetCentroidsInterpretation. Цикл проходження по всіх центроїдах, задання мапи ваг належності до кожного кластеру

```
for(int32 k = 0; k < Indexes.Num(); ++k)
{
    switch (Indexes[k])
    {
    case 0:
        ClusterProbability[BehaviourType::Explorer] += (Indexes.Num() - k) / 3.f;
        break;
    case 1:
        ClusterProbability[BehaviourType::Explorer] += (Indexes.Num() - k) / 3.f;
        ClusterProbability[BehaviourType::Achiever] += (Indexes.Num() - k) / 2.f;
        break;
    case 2:
        ClusterProbability[BehaviourType::Achiever] += (Indexes.Num() - k) / 2.f;
        break;
    case 3:
        ClusterProbability[BehaviourType::Socializer] += (Indexes.Num() - k) / 2.f;
        break;
    case 4:
        ClusterProbability[BehaviourType::Socializer] += k / 2.f;
        break;
    case 5:
        ClusterProbability[BehaviourType::Killer] += (Indexes.Num() - k) / 2.f;
        break;
    case 6:
        ClusterProbability[BehaviourType::Killer] += k / 2.f;
        break;
    case 7:
        ClusterProbability[BehaviourType::Explorer] += (Indexes.Num() - k) / 3.f;
        break;
    default:
        break;
    }
}
```

Рис. Е.2 – Функція GetCentroidsInterpretation. Проходження по всіх атрибутах центроїда

```
        ClusterProbability.ValueSort( Predicate: [](float A, float B) ->bool
        {
            return A > B;
        });

        TArray<BehaviourType> Clusters;
        ClusterProbability.GetKeys( [&] Clusters);

        for(BehaviourType Behaviour : Clusters)
        {
            if(!OutInterpretation.Contains(Behaviour))
            {
                OutInterpretation.Add(Behaviour);
                break;
            }
        }

    }
}
```

Рис. Е.3 – Функція GetCentroidsInterpretation. Віднесення центру до конкретного кластеру за класифікацією Бартла

Додаток Є

```
void UTraitGeneticAlgorithm::Initialize(UGeneticAlgorithmSave* NewGeneticAlgorithmSave,
    UBehaviourAnalysis* BehaviourAnalysis,int32 MinPopulationCount)
{
    if(!IsValid(NewGeneticAlgorithmSave))
    {
        return;
    }

    GeneticAlgorithmSave = NewGeneticAlgorithmSave;

    TArray<FSavedRoomTraits> SavedRoomTraits;
    GeneticAlgorithmSave->GetTraitsByRoomType([&]SavedRoomTraits, Type);

    for(const FSavedRoomTraits& Traits : SavedRoomTraits)
    {
        URoomTraits* NewRoomTraits = NewObject<URoomTraits>(Outer:this, RoomTraitsClass);
        NewRoomTraits->RoomType = Traits.RoomType;
        NewRoomTraits->Traits = Traits.RoomTraits;

        if(BehaviourAnalysis != nullptr)
        {
            NewRoomTraits->FuzzyClusteringResult = BehaviourAnalysis->FuzzyClusteringResult;
        }

        TraitsPopulation.Add(NewRoomTraits);
    }
```

Рис. Є.1 – Початок функції ініціалізації генетичного алгоритму. Підвантаження збережених індивідів

```
    if(TraitsPopulation.Num() < MinPopulationCount)
    {
        const int32 AdditionalTraitsCount = MinPopulationCount - TraitsPopulation.Num();
        for(int32 i = 0; i < AdditionalTraitsCount; ++i)
        {
            URoomTraits* NewRoomTraits = NewObject<URoomTraits>(Outer: this, RoomTraitsClass);
            NewRoomTraits->InitializeDefault();

            if(BehaviourAnalysis != nullptr)
            {
                NewRoomTraits->FuzzyClusteringResult = BehaviourAnalysis->FuzzyClusteringResult;
            }

            TraitsPopulation.Add(NewRoomTraits);
        }
    }
    else
    {
        URoomTraits* NewRoomTraits = NewObject<URoomTraits>(Outer: this, RoomTraitsClass);
        NewRoomTraits->InitializeDefault();

        if(BehaviourAnalysis != nullptr)
        {
            NewRoomTraits->FuzzyClusteringResult = BehaviourAnalysis->FuzzyClusteringResult;
        }

        TraitsPopulation.Add(NewRoomTraits);
    }

    bIsExploration = GetExplorationExploitationType(BehaviourAnalysis);
}
```

Рис. Є.2 – Продовження функції ініціалізації генетичного алгоритму. Ініціалізація індивідів за замовчування за потреби, визначення етапу «Exploration–Exploitation»

```
void UTraitGeneticAlgorithm::Execute(TArray<URoomTraits*>& OutTraits, int32 BestCount)
{
    if(TraitsPopulation.Num() == BestCount)
    {
        for(int32 i = 0; i < BestCount; ++i)
        {
            OutTraits.Add(TraitsPopulation[i]);
            GeneticAlgorithmSave->AddRoomTraitsToSave(TraitsPopulation[i]);
        }
        return;
    }

    TArray<URoomTraits*> NewPopulation;
    RouletteWheelSelection([&]NewPopulation);

    TArray<URoomTraits*> CrossoverPopulation = NewPopulation;
    const int32 CrossoverPopulationCount = FMath::RoundToInt32(F:NewPopulation.Num() * CrossoverPercent);
    for(;CrossoverPopulation.Num() > CrossoverPopulationCount;)
    {
        CrossoverPopulation.Pop(bAllowShrinking:false);
    }
    for(int32 i = 0; i < CrossoverPopulation.Num() / 2; ++i)
    {
        TraitsPopulation.Add(Item:CrossoverPopulation[i]->
            Crossover(CrossoverPopulation[CrossoverPopulation.Num() / 2 + i]));
    }
    if(CrossoverPopulation.Num() % 2 == 1)
    {
        TraitsPopulation.Add(Item:CrossoverPopulation[0]->Crossover(CrossoverPopulation.Last()));
    }
```

Рис. Є.3 – Основна функція виконання генетичного алгоритму. Селекція та схрещування

```
    TArray<URoomTraits*> MutatePopulation = NewPopulation;
    const int32 MutatePopulationCount = NewPopulation.Num() - CrossoverPopulationCount;
    for(;MutatePopulation.Num() > MutatePopulationCount;)
    {
        MutatePopulation.RemoveAt(Index:0);
    }
    for(URoomTraits* Traits : MutatePopulation)
    {
        TraitsPopulation.Remove(Traits);
        Traits->Mutate(MutateGrowChance, MutateCutChance, MinIterationCount:1, MaxIterationCount:1);
        TraitsPopulation.Add(Traits);
    }

    GetSortedPopulationByFitnessValue([&]NewPopulation);

    for(int32 i = 0; i < BestCount; ++i)
    {
        OutTraits.Add(NewPopulation[i]);
        GeneticAlgorithmSave->AddRoomTraitsToSave(NewPopulation[i]);
    }
}
```

Рис. Є.4 – Продовження функції виконання генетичного алгоритму. Мутація та вибір найкращих індивідів

Abstract.

Laitaruk I. F. - Research on world procedural generation based on the development of a computer game with player behavior analysis. Manuscript.

Qualification work for the degree of "Master" in the field of Computer Science, educational program "Computer Science and Information Technologies." – Lesya Ukrainka Volyn National University. – 2025.

This paper represents a study of world procedural generation based on the development of a model for constructing a room-distributed house integrated into a computer game. The initial stage involves the analysis of modern algorithms such as graph rewriting, gradient noise, and Voronyi diagram. The project is developed using Unreal Engine 5.4 game engine and has an integrated auxiliary tool – the PCG Plugin. A modified BSP tree-based algorithm is applied for implementation, where nodes are interpreted as room centers, a stage of random tree formation takes into account the type of rooms, introduction of a post-merge queue for the stage of searching for passages between rooms. The characteristics of the rooms are determined using a genetic algorithm based on fuzzy clustering of player behavior with the Fuzzy C-Means method. Balancing the adaptability of generation is performed by integrating the "Exploration–Exploitation" dilemma and Shannon information entropy into the genetic algorithm. The results of the study include generated houses under different scenarios and an analysis of entropy change across multiple gameplay sessions.

Keywords: procedural generation, BSP tree, genetic algorithm, fuzzy clustering, Unreal Engine, «Exploration–Exploitation» dilemma, Shannon information enthropy.

Анотація

Лайтарук І. Ф. – Дослідження процедурної генерації світу на основі розробки комп'ютерної гри з аналізом поведінки гравця. – Рукопис.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки, освітньої програми Комп'ютерні науки та інформаційні технології. – Волинський національний університет імені Лесі Українки. – 2025 р.

У роботі проведено дослідження процедурної генерації світу на основі розробки моделі формування розподіленого будинку для проходження гри. Початковим етапом був аналіз сучасних алгоритмів: переписування графу, градієнтних шумів, діаграми Вороного. Розробка відбувалася на ігровому рушії Unreal Engine 5.4 та інтегрованим допоміжним інструментом – PCG Plugin. Для реалізації використано алгоритм на основі BSP дерева із модифікаціями – вузли інтерпретуються як центри кімнат, етап випадкового формування дерева із врахуванням типу кімнат, введення черги пост-злиття для етапу пошуку проходів між кімнатами. Характеристики кімнат визначаються за допомогою генетичного алгоритму на основі нечіткої кластеризації поведінки гравця методом Fuzzy C-Means. Балансування адаптивності генерації відбувалось за допомогою введення принципу «Exploration–Exploitation» та інформаційної ентропії Шеннона у генетичний алгоритм. У результатах дослідження описано згенеровані будинки в різних випадках та аналіз зміни значення ентропії протягом декількох проходжень гри.

Ключові слова: процедурна генерація, BSP дерево, генетичний алгоритм, нечітка кластеризації, Unreal Engine, дилема «Exploration–Exploitation», інформаційна ентропія Шеннона.