

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ
ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ РІВНІВ У 2D-ГРІ
RESEARCH AND SOFTWARE IMPLEMENTATION OF A
PROCEDURAL LEVEL GENERATION SYSTEM IN A 2D GAME

Виконав(ла): студент(ка) 2 курсу, групи КНТ-114м

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ВЛАСЕНКО К.В.

(ПРИЗВИЩЕ та ініціали)

Керівник ДУБРОВІН В.І.

(ПРИЗВИЩЕ та ініціали)

Рецензент БАБЕНКО Н.В.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

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

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ВЛАСЕНКА Костянтина Віталійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація системи процедурної генерації рівнів у 2D-грі
Research and Software Implementation of a Procedural Level Generation System in a 2D Game

керівник проєкту (роботи) к.т.н., професор, ДУБРОВІН Валерій Іванович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 08 грудня 2025 року

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

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

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ДУБРОВІН В.І., професор		
Нормоконтролер	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “30” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	4-5 тижні	Розділ 2
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7-8 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	9 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

(підпис) Костянтин ВЛАСЕНКО
(Імя ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Валерій ДУБРОВІН
(Імя ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
118 с., 12 табл., 38 рис., 2 дод., 28 джерел.

ПРОЦЕДУРНА ГЕНЕРАЦІЯ, ЛАБІРИНТ, ПЕЧЕРА, ОЦІНКА
РІВНІВ, МЕТРИКИ ЯКОСТІ, PYTHON.

Об'єкт дослідження – процес розроблення програмного забезпечення для процедурної генерації ігрового контенту з автоматизованим оцінюванням якості.

Предмет дослідження – методи проектування, реалізації та тестування програмних систем процедурної генерації з формалізованими метриками якості для алгоритмів створення лабіринтів і печерних структур.

Мета роботи – розроблення програмного забезпечення для автоматизації процесу створення ігрових рівнів шляхом реалізації системи процедурної генерації з інтегрованими метриками якості та порівняльним аналізом алгоритмів.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, алгоритми процедурної генерації, бібліотеки Python, методи комп'ютерного зору та аналізу графів, персональний комп'ютер під управлінням ОС Windows 10.

Результати. Створено програмний застосунок з графічним інтерфейсом для генерації лабіринтів і печер шістьма алгоритмами та автоматичного обчислення чотирьох метрик якості. Проведено експериментальне дослідження, у ході якого порівняно алгоритми за середніми показниками кожної метрики.

Висновки. Розроблена система дозволяє оцінювати та порівнювати алгоритми процедурної генерації за формальними критеріями. Отримані результати демонструють, що різні підходи до генерації створюють

принципово відмінні просторові структури й можуть бути свідомо обрані під конкретні завдання геймдизайну.

Галузь використання – дослідження процедурної генерації, розроблення ігрових рівнів.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 118 pages, 12 tables, 38 figures, 2 appendixes, 28 sources.

PROCEDURAL GENERATION, MAZE, CAVE, LEVEL EVALUATION, QUALITY METRICS, PYTHON.

Object of study – the process of developing software for procedural generation of game content with automated quality assessment.

Subject of study – methods for designing, implementing, and testing procedural generation software systems with formalized quality metrics for algorithms that create mazes and cave structures.

Aim of the work – the development of software to automate the process of creating game levels by implementing a procedural generation system with integrated quality metrics and comparative analysis of algorithms.

Materials, methods, and tools: structural and object-oriented programming, procedural generation algorithms, Python libraries, computer vision techniques and graph analysis methods, a personal computer running Windows 10.

Results. A software application with a graphical user interface was created to generate mazes and caves using six algorithms and to automatically calculate four quality metrics. An experimental study was conducted in which the algorithms were compared according to the average values of each metric.

Conclusions. The developed system makes it possible to evaluate and compare procedural generation algorithms according to formal criteria. The obtained results show that different generation approaches create fundamentally distinct spatial structures and can be consciously selected for specific game design tasks.

The field of use is research in procedural generation and game level development.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області.....	10
1.1 Критерії оцінки алгоритмів процедурної генерації.....	10
1.2 Огляд існуючих методів процедурної генерації	15
1.3 Висновок за розділом 1.....	30
2 Обґрунтування вибору програмних засобів розробки	32
2.1 Вибір мови програмування для розробки системи.....	32
2.2 Вибір інтегрованого середовища розробки.....	35
2.3 Огляд використаних бібліотек python.....	39
2.4 Висновок за розділом 2.....	44
3 Основні рішення щодо реалізації компонентів системи.....	46
3.1 Архітектура системи процедурної генерації.....	46
3.2 Реалізація класу mazegenerator	51
3.3 Реалізація класу metricscalculator	62
3.4 Реалізація графічного інтерфейсу користувача	70
3.5 Висновок за розділом 3	75
4 Експлуатація, тестування та експериментальне дослідження програми	77
4.1 Інструкція користувача та демонстрація роботи системи	77
4.2 Методика проведення експериментів	80
4.3 Результати експериментального дослідження.....	81
4.4 Оптимізація застосування алгоритмів	85
4.5 Висновок за розділом 4.....	87
Висновки	89
Перелік джерел посилання	90
Додаток А Текст програми.....	93
Додаток Б Слайди презентації	112

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
BSP	– Binary Space Partitioning;
CSV	– Comma-Separated Values;
GUI	– Graphical User Interface;
IDE	– Integrated Development Environment;
NumPy	– Numerical Python;
OpenCV	– Open Source Computer Vision Library;
БД	– база даних;
ПЗ	– програмне забезпечення.

ВСТУП

Сфера розроблення відеоігор є однією з найпотужніших галузей цифрової індустрії. Світовий ринок ігор у 2024 році перевищив 450 млрд доларів США і продовжує стабільно зростати [1]. Зі збільшенням масштабів ігрових проєктів та ускладненням очікувань користувачів важливим стає використання методів, що дозволяють автоматично генерувати різноманітний ігровий контент.

Процедурна генерація контенту є однією з ключових технологій сучасного геймдизайну. Вона забезпечує створення рівнів, локацій, карт та структур з високим рівнем варіативності та реіграбельності [2]. Такі підходи широко застосовуються як у незалежному, так і у AAA-сегменті ігрової індустрії – забезпечуючи унікальний досвід кожного проходження.

Попри популярність методів процедурної генерації, проблема оцінювання якості згенерованих рівнів залишається актуальною. Недостатньо лише створити лабіринт чи підземелля — важливо мати можливість виміряти їх складність, відкритість, збалансованість та інші ключові параметри. У наукових публікаціях останніх років підкреслюється необхідність формальних метрик, які дозволяють порівнювати різні алгоритми генерації та оцінювати їх придатність до ігрового процесу [3].

Таким чином, розроблення інструменту, що поєднує кілька алгоритмів генерації та формалізовані метрики для їх оцінювання, є актуальним завданням у контексті сучасних досліджень у галузі процедурного контенту та практичної розробки відеоігор.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Критерії оцінки алгоритмів процедурної генерації

Щоб порівняти алгоритми процедурної генерації, потрібно спочатку визначити параметри для їх оцінювання. У контексті задачі побудови ігрових рівнів до основних критеріїв можна віднести складність, відкритість, зв'язність та поблажливість рівня.

1.1.1 Поблажливість рівня

Поблажливість – це метрика, яка вимірює, наскільки рівень поблажливий до помилок гравця. Вона розраховується шляхом усереднення показників поблажливості для кожної з потенційних перешкод у рівні та нормалізації цього значення до діапазону від 0 до 1. Такими перешкодами могли бути розрив або ворог (зі значенням поблажливості $l = -1$), стрибок без пов'язаного з ним розриву ($l = 1$) та інші елементи.

Для гри лабіринту не одразу зрозуміло, як застосувати цю метрику, оскільки в ній відсутні очевидні перешкоди. Тому був обраний метод обчислення поблажливості як частки шляхів, що призводять у глухі кути. Для кожної незаповненої плитки t знаходиться найкоротший шлях від старту s до t . Потім цей шлях «заповнюється» і визначається, чи існує ще шлях від t до цілі g . Плитка t позначається як глухий кут лише в тому випадку, якщо такого шляху не існує [4].

1.1.2 Складність рівня

Ця метрика оцінює рівень «складності» у структурі печери. Ідеально гладка печера, наприклад правильне коло, отримає низький бал складності. Навпаки, печера з нерівними стінами, розгалуженими проходами та загальною заплутаністю матиме вищий бал. Складність є ключовим

фактором при оцінці якості печери, адже вона безпосередньо впливає на візуальну привабливість і природність згенерованих печерних систем. Природні печери часто мають різні рівні складності/шорсткості: від гладких ділянок до більш нерівних поверхонь, відкритих залів та розлогих взаємопов'язаних проходів. Тому оцінка складності печери дозволяє забезпечити більш природний та деталізований вигляд, підвищуючи загальну атмосферність і естетичну привабливість ігрового середовища.

На рисунку 1.1 зображено оцінені бали метрики для печерних систем зі зростаючою складністю.

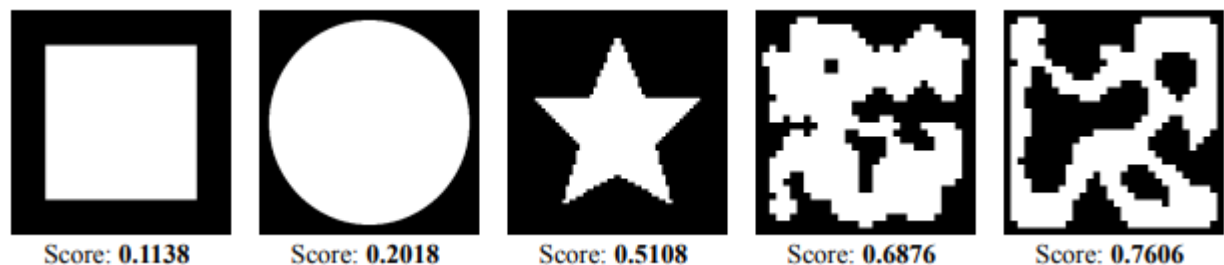


Рисунок 1.1 – Оцінені бали метрики для печерних систем зі зростаючою складністю

Реалізація метрики складності включала кілька кроків для кількісної оцінки заплутаності структури печер.

Крок 1. Спочатку алгоритм обчислює периметр і площу як зовнішніх (parent), так і внутрішніх (child) контурів на зображенні печери. Зовнішні контури відображають загальну форму печери без внутрішніх деталей, тоді як внутрішні контури відповідають внутрішнім особливостям або «порожнинам» у структурі печери.

Крок 2. Далі алгоритм визначає максимальну «шорсткість» печери, яка визначається як половина радіуса кола з площею, що дорівнює загальній площі зовнішнього контуру. Це значення використовується як еталон для подальшого розрахунку.

Крок 3. Потім фактична шорсткість печери обчислюється як відношення різниці площі зовнішніх і внутрішніх контурів до загального периметра. Це співвідношення відображає, наскільки нерівними є межі печери.

Крок 4. Щоб отримати підсумковий бал складності, алгоритм нормалізує фактичну шорсткість, поділивши її на максимальну (крок 2). Це гарантує, що результат лежить у діапазоні від 0 до 1. Чим ближче бал до 1, тим складніша і більш заплутана структура печери з нерівними стінами й відгалуженнями. Бал, близький до 0, означає більш гладку і просту структуру.

Такий підхід до реалізації ефективно враховує як нерівність меж печерної системи, так і наявність внутрішніх особливостей.

1.1.3 Відкритість рівня

Ця метрика оцінює баланс між «відкритістю» та «звуженістю» у структурі печери. Відкритість характеризує простір і ширину залів та проходів, тоді як звуженість відображає тісноту й компактність. Ідеальне планування печери поєднує обидва аспекти: широкі зали для дослідження та вузькі коридори для навігації. Печера з великими просторами та широкими проходами отримує високий бал, тоді як печера з вузькими й обмеженими ходами – низький. Таким чином, значення, близьке до середини (0.5), вважається оптимальним балансом між цими параметрами. Оцінка відкритості та звуженості є важливою для створення візуально привабливих печер із різноманітними структурами, що забезпечують цікаве й різнопланове дослідження.

На рисунку 1.2 зображено оцінені бали метрики для печерних систем із різним рівнем відкритості.

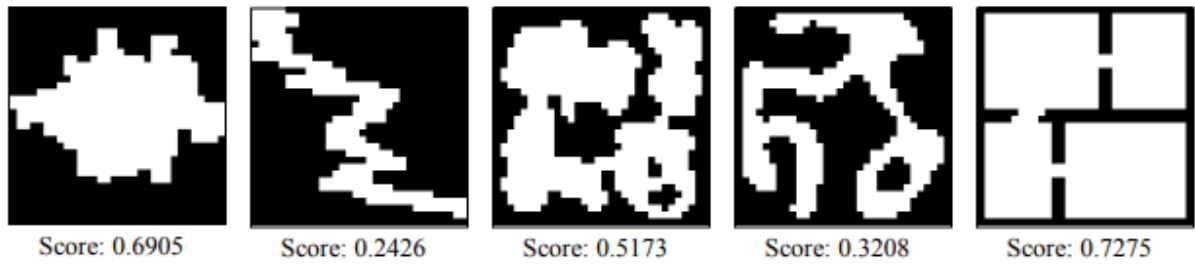


Рисунок 1.2 – Оцінені бали метрики для печерних систем із різним рівнем відкритості

Реалізація метрики відкритості/звуженості включає кілька етапів кількісної оцінки просторового балансу.

Етап 1. На початковому етапі алгоритм створює сітку, що представляє план печери, де кожна клітинка класифікується як тверда (стіна) або порожня (простір печери).

Етап 2. Далі алгоритм ітеративно присвоює значення порожнім клітинкам залежно від відстані до найближчих твердих клітин. Клітинки, що межують зі стінами, отримують значення 1, вказуючи на безпосередню близькість до меж печери. Чим далі клітинка розташована від стін, тим вищим є її значення, що відображає більший рівень відкритості. Призначення триває доти, доки всі порожні клітинки не отримають свої значення. У результаті формується карта відкритості/звуженості печери.

Етап 3. Після цього обчислюється бал відкритості/звуженості як відношення кількості порожніх клітин зі значенням більшим за 1 до загальної кількості порожніх клітин. Це співвідношення відображає частку відкритих просторів у плануванні печери.

Етап 4. Нормалізація цього значення гарантує, що підсумковий бал лежить у діапазоні від 0 до 1: чим він вищий, тим печера відкритіша, а нижчий – тим вона більш вузька.

Такий підхід дозволяє ефективно кількісно оцінити баланс між відкритими просторами та вузькими проходами у згенерованих печерних системах. Метрика забезпечує надійну основу для оцінки візуальної

привабливості, просторової динаміки та потенціалу дослідження в ігровому середовищі.

1.1.4 Зв'язність рівня

Ця метрика оцінює рівень зв'язності у системі печер, тобто наскільки взаємопов'язані між собою різні зали та проходи. Ідеальною вважається система, де всі елементи з'єднані між собою, що забезпечує безперервне дослідження та вільний перехід між різними ділянками. Навпаки, ізольовані печери з обмеженими переходами ускладнюють навігацію й роблять середовище менш цікавим для дослідження. Тому оцінка зв'язності є ключовою для створення цілісних, візуально привабливих і захопливих печерних систем.

На рисунку 1.3 зображено оцінені значення метрики для печерних систем із різним рівнем зв'язності.



Рисунок 1.3 – Оцінені бали метрики для печерних систем із різним рівнем зв'язності

Реалізація метрики зв'язності включає кілька етапів кількісної оцінки.

Етап 1. Спочатку алгоритм виконує попередню обробку вхідного зображення за допомогою морфологічної операції opening із спеціальним ядром. Це дозволяє закрити діагональні отвори, підвищити чіткість структури печери та забезпечити коректне відокремлення ізольованих систем.

Приклад: вихідне зображення печери до та після застосування морфологічної обробки.

Етап 2. Далі з попередньо обробленого зображення витягуються контури. Для підвищення точності алгоритм відкидає так звані карликові контури (занадто малі печери), які не мають значного впливу на ігровий процес. Для розрахунків враховуються лише ті печерні системи, площа яких перевищує 20% площі найбільшої печери у розташуванні.

Приклад: на вихідному зображенні маленька печера в нижньому правому куті була відкинута під час попередньої обробки.

Етап 3. Нарешті, обчислюється показник зв'язності. Для цього використовується спеціальна формула, яка враховує кількість знайдених контурів і посилює відмінності при низьких балах, що дозволяє коректно відобразити ступінь ізолюваності за формулою (1.1):

$$\text{Connectivity Score} = \frac{1}{2^{x-1}}, \quad (1.1)$$

де x – кількість контурів.

Бал 1.0 означає повну зв'язність печерної системи, тоді як нижчі значення вказують на наявність ізолюваних ділянок.

Таким чином, алгоритм дозволяє надійно оцінювати ступінь зв'язності печерних систем, забезпечуючи корисну метрику для аналізу їхньої цілісності та зручності навігації [5].

1.2 Огляд існуючих методів процедурної генерації

Автоматичне створення ігрового контенту за допомогою процедурної генерації рівнів набуло статусу ключової технології в сучасній ігровій індустрії. Процедурна генерація передбачає алгоритмічне формування ігрових елементів: рівнів, карт, об'єктів без необхідності ручного проектування. Використання цього методу сприяє значному розширенню

варіативності, економії робочих ресурсів та гнучкому налаштуванню процесу генерації залежно від вхідних даних або побажань гравців [6].

Процедурна генерація може ґрунтуватися на різноманітних підходах: рекурсивних методах, алгоритмах випадкового блукання, клітинних автоматах, функціях шуму, формальних граматиках та інших. Кожен із цих методів створює унікальні архітектурні рішення – від складних лабіринтів до масштабних печерних просторів.

У даному підрозділі здійснено огляд сучасних підходів до генерації, характерних для двовимірних ігор: проаналізовано їхні базові принципи, позитивні аспекти та обмеження.

1.2.1 Алгоритм рекурсивного пошуку з поверненням

Алгоритм рекурсивного пошуку з поверненням є доволі простим методом випадкової генерації лабіринтів. Як випливає з назви, його робота ґрунтується на принципі повернення назад (backtracking), що реалізується за допомогою рекурсії.

На рисунку 1.4 зображено приклад лабіринту згенерованого за допомогою цього алгоритму.

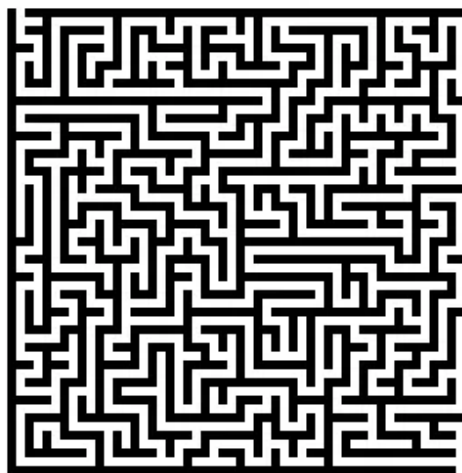


Рисунок 1.4 – Лабіринт, створений за допомогою алгоритму рекурсивного пошуку з поверненням

Метод вирізняється високою швидкістю виконання, простотою розуміння та відносною легкістю реалізації. Водночас він вимагає достатнього обсягу пам'яті для збереження всієї структури лабіринту, а також додаткового стекового простору, пропорційного розміру лабіринту. Через це при надзвичайно великих розмірах лабіринтів алгоритм може виявитися неефективним.

Основні кроки алгоритму можна подати у такій послідовності.

Крок 1. Вибрати випадкову клітинку як початкову.

Крок 2. Обрати випадкову сусідню клітинку. Прохід створюється лише в тому випадку, якщо ця клітинка ще не відвідана.

Крок 3. Повторювати процес, доки залишаються сусідні клітинки для вибору.

Крок 4. Якщо сусідніх клітинок немає, виконати повернення назад до попередньої клітинки, з якої ще можна зробити вибір.

Крок 5. Алгоритм завершується тоді, коли відбувається повернення до початкової клітинки [7].

1.2.2 Алгоритм рекурсивного поділу

Алгоритм рекурсивного поділу належить до методів генерації лабіринтів, які реалізуються шляхом поступового додавання стін. Особливу зацікавленість він викликає завдяки своїй фрактальній природі: теоретично процес можна продовжувати нескінченно, розділяючи простір на дедалі менші ділянки та досягаючи все вищого рівня деталізації.

На рисунку 1.5 зображено приклад лабіринту згенерованого за допомогою цього алгоритму.

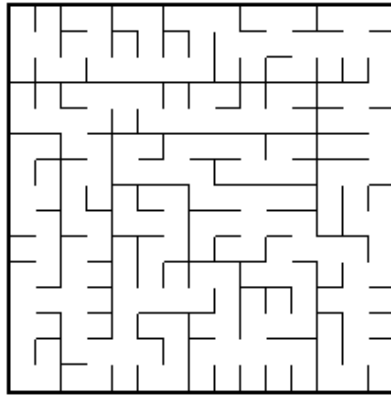


Рисунок 1.5 – Лабіринт, створений за допомогою алгоритму рекурсивного поділу

Принцип роботи алгоритму полягає в такому.

Крок 1. Почати з порожнього поля.

Крок 2. Поділити поле стіною навпіл, горизонтально або вертикально, залишивши один прохід у стіні.

Крок 3. Повторити крок 2 для кожної з отриманих ділянок.

Крок 4. Продовжувати рекурсивний поділ доти, доки лабіринт не досягне необхідного рівня деталізації.

Важливо відзначити, що навіть при ранніх ітераціях утворюється коректний лабіринт. Кожна наступна ітерація алгоритму лише підвищує рівень структурної складності та деталізації створеного середовища [8].

1.2.3 Алгоритм випадкової прогулянки

Алгоритм випадкової прогулянки є відносно простим методом, який може використовуватися для генерації рівнів із середовищем, подібним до печерного. Використання однієї траєкторії випадкової прогулянки гарантує отримання повністю зв'язного рівня з різноманітним поєднанням вузьких проходів та відкритих просторів.

На рисунку 1.6 зображено приклад печери згенерованої з використанням кроків лише у чотирьох традиційних кардинальних напрямках. Водночас алгоритм може бути налаштований на застосування інших схем

кроків (вісім напрямів, гексагональна сітка тощо) залежно від обраної реалізації.

```
#####
#####.#####
#####.##.#####
#####.##.#####
#####.....#####
#####.....#####
#####.....#####
#####.<...#.#####
#####.....#####
####.#.###.###.#.###.#####.##
####.#.###.#####.....#####.##
####.#.#.....##.....#####.....#
####.###.....#.....#.#####.....#
#####.###.....#.....#.....###
##.....####.....###
###.#####.....#......>.####
###.#####.....#####.#####
###.#####.#####.....#.#####
####.#####.#####.....#.#####
###.....#####.#####.##...##.....###
###.....#####.#####.#####
###.....###.#####.#####
###.#.....#####
####.....##.#####
#####.....#####
#####.#####
#####.#####
#####.#####
```

Рисунок 1.6 – Печера, створена за допомогою алгоритму випадкової прогулянки

Для досягнення оптимальних результатів алгоритм вимагає динамічного зміни розмірів карти; без цього згенеровані рівні можуть стикатися з межами поля, що призводить до естетично незадовільних результатів.

Псевдокод алгоритму:

- а) ініціалізувати всі клітинки карти як стіни;
- б) вибрати випадкову клітинку як початкову точку;
- в) перетворити вибрану клітинку на підлогу;
- г) поки кількість клітинок-підлоги є недостатньою:
 - 1) зробити крок у випадковому напрямі;
 - 2) якщо нова клітинка є стіною, перетворити її на підлогу та збільшити лічильник клітинок-підлоги.

Таким чином, алгоритм випадкової прогулянки дозволяє поступово формувати структуру лабіринту, забезпечуючи при цьому зв'язність та варіативність просторової організації рівня.

1.2.4 Клітинні автомати

Використання клітинних автоматів для генерації печероподібних структур є відомим і достатньо добре задокументованим підходом. Базова ідея полягає в тому, що початкове поле заповнюється випадковим чином, після чого до нього послідовно застосовуються правила клітинних автоматів. Найчастіше використовується правило «4–5»: клітинка перетворюється на стіну, якщо вона вже була стіною і принаймні 4 з 8 сусідніх клітинок також були стінами, або якщо вона була не стіною, але принаймні 5 сусідніх клітинок були стінами.

Іншими словами, клітинка стає стіною, якщо у 3×3 області з центром у цій клітинці нараховується щонайменше 5 стін. Кожна наступна ітерація робить клітинки більш подібними до їхніх сусідів, поступово зменшуючи кількість випадкового «шуму» та надаючи карті впорядкованішої структури.

На рисунку 1.7 зображено приклад зменшення шуму за ітераціями.

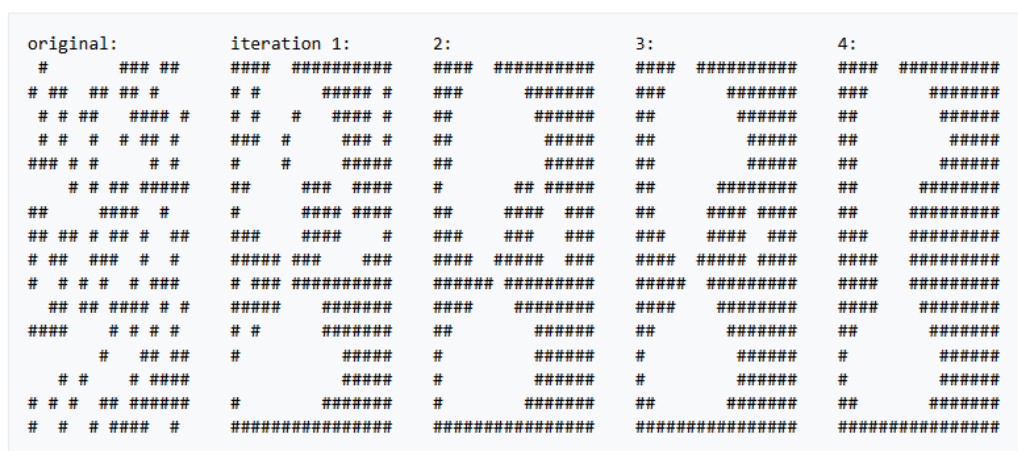


Рисунок 1.7 – Лабіринт, створений за допомогою клітинних автомат

Проблема полягає в тому, що результати є непослідовними: алгоритм має схильність генерувати роз'єднані карти [9].

Алгоритм роботи з використанням клітинних автоматів можна узагальнити у три основні фази:

- створення початкової карти шляхом випадкового заповнення клітинок стінами та порожнім простором;
- застосування циклів правил клітинного автомата для поступового наближення карти до печероподібної структури;
- з'єднання окремих печер таким чином, щоб кожна клітинка карти залишалася досяжною з будь-якої іншої.

Завдяки своїй простоті та ефективності метод клітинних автоматів дозволяє швидко отримати варіативні карти з високим рівнем схожості на природні печерні системи [10].

1.2.5 Бінарне розбиття простору

Алгоритм бінарного розбиття простору у спрощеному вигляді передбачає, що кореневий вузол ділиться на два вузли, після чого кожен із нових вузлів також ділиться на два, і цей процес повторюється до досягнення заданої глибини. Кінцеві вузли, так звані «листки», потім перетворюються на кімнати [11].

Процес генерації починається з прямокутного підземелля, повністю заповненого клітинками-стінами. Підземелля рекурсивно ділиться, доки кожне його під підземелля(subdungeon) не набуде приблизного розміру майбутньої кімнати. Операція розбиття включає такі кроки.

Крок 1. Вибір випадкового напрямку: горизонтальне або вертикальне розділення.

Крок 2. Вибір випадкової позиції (x для вертикального поділу, y для горизонтального).

Крок 3. Розбиття підземелля на два підпідземелля.

На рисунку 1.8 зображено приклад першої ітерації.

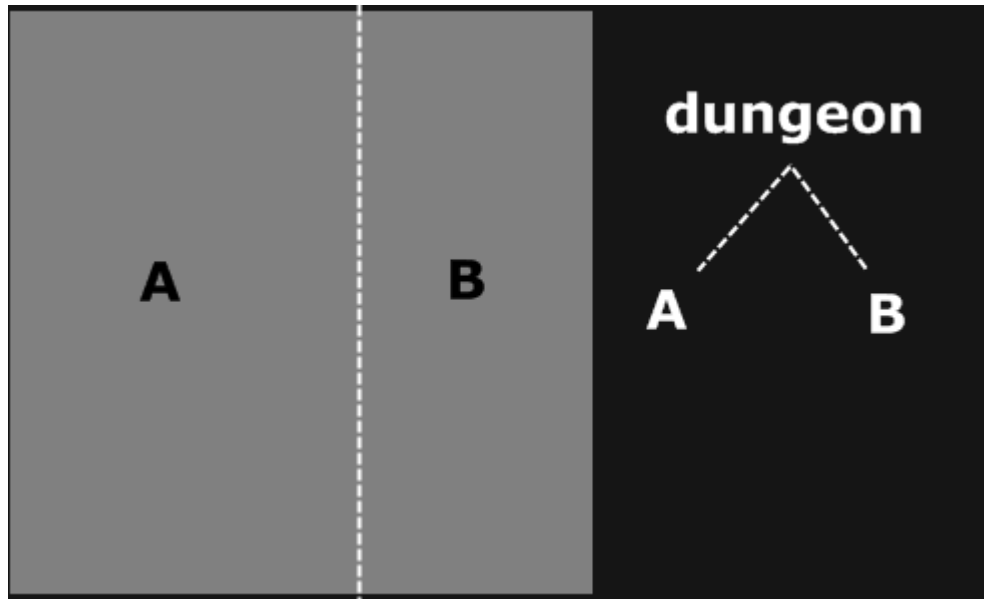


Рисунок 1.8 – Перша ітерація алгоритму бінарного розбиття

Після першої ітерації отримуються два підпідземелля, позначені як A і B. Ту саму операцію можна застосувати до кожного з них рекурсивно.

На рисунку 1.9 зображено приклад другої ітерації.

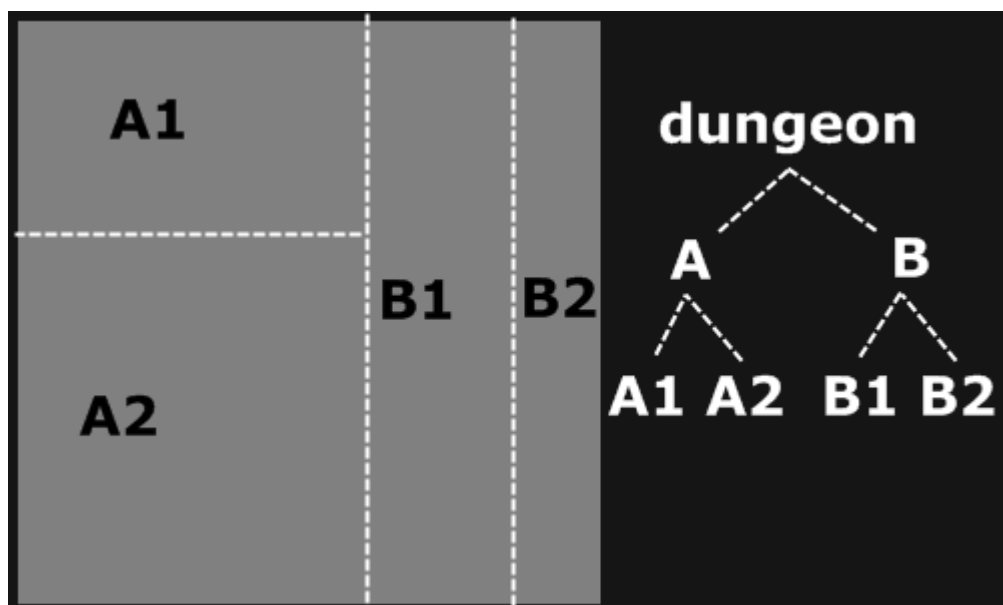


Рисунок 1.9 – Друга ітерація алгоритму бінарного розбиття

При виборі позиції розділення слід враховувати, щоб вона не була занадто близько до меж підземелля, оскільки необхідно забезпечити можливість розміщення кімнати в кожному з утворених підпідземель. Процес повторюється до тих пір, поки найменші підпідземелля не досягнуть приблизного розміру бажаних кімнат.

На рисунку 1.10 зображено приклад після 4 ітерацій.

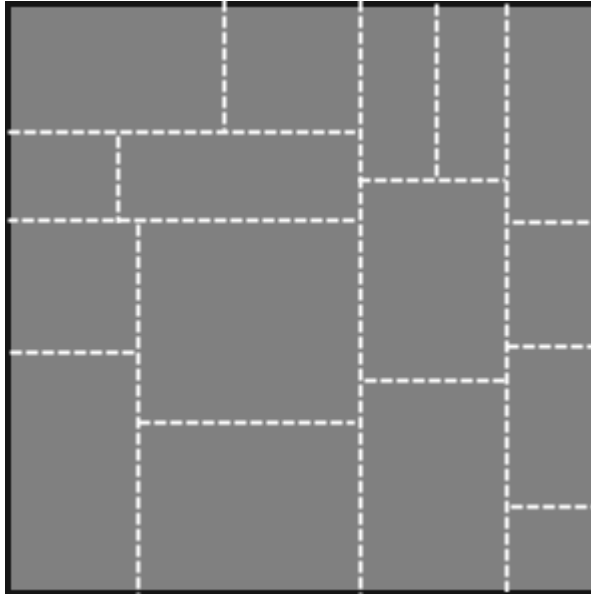


Рисунок 1.10 – П'ята ітерація алгоритму бінарного розбиття

Варіації правил вибору позиції розділення можуть призводити до утворення більш однорідних підпідземель (позиція між 0,45 і 0,55) або більш різномірних (позиція між 0,1 і 0,9). Також можливо застосувати глибшу рекурсію в окремих частинах підземелля, щоб отримати там менші кімнати.

На наступному етапі в кожному «листку» BSP-дерева створюється кімната випадкового розміру. При цьому кімната обов'язково повинна повністю міститися всередині відповідного підпідземелля. Завдяки структурі BSP-дерева виключається можливість накладання кімнат одна на одну.

На рисунку 1.11 наведено створені кімнати.

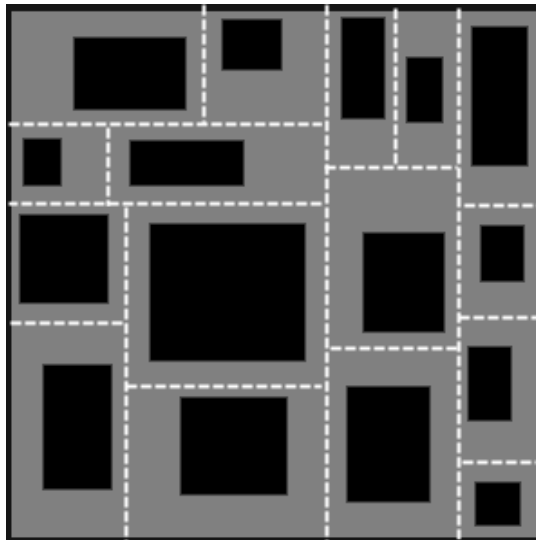


Рисунок 1.11 – Створені кімнати

Для побудови коридорів здійснюється обхід усіх листків дерева, підключаючи кожен листок до його «сестринського». Якщо дві кімнати мають стіни, що дивляться одна на одну, можна використати прямий коридор. В іншому випадку застосовується коридор форми зигзагу.

На рисунку 1.12 наведено з'єднання чотирьох підпіземель.

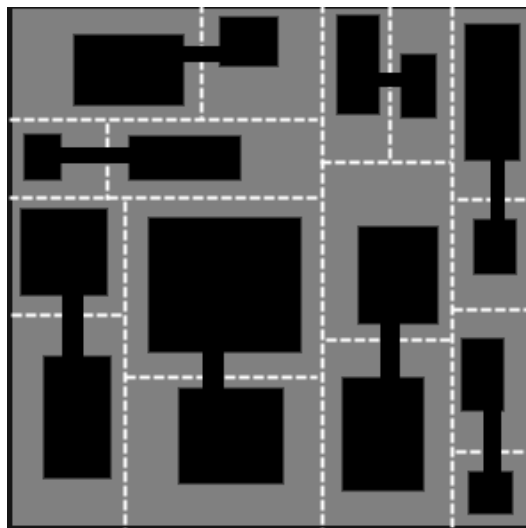


Рисунок 1.12 – З'єднання чотирьох підпіземель

Після цього піднімаємося на рівень вище в дереві і повторюємо процес для батьківських підрегіонів. На цьому рівні можна з'єднувати дві

підрегіональні області через два приміщення, приміщення та коридор, два коридори.

На рисунку 1.13 наведено з'єднання трьох підпідземель.

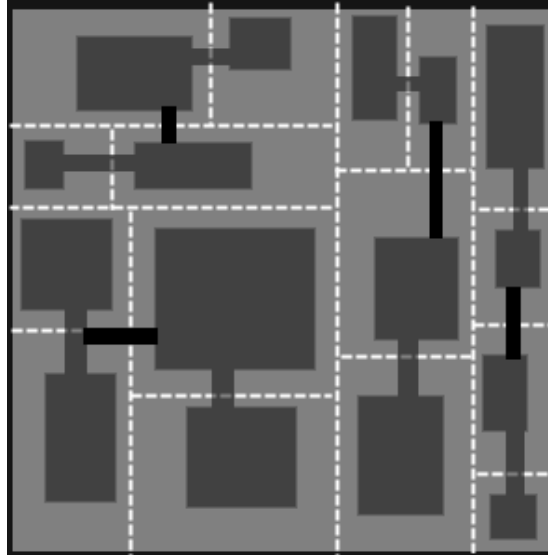


Рисунок 1.13 – З'єднання трьох підпідземель

Процес повторюється рекурсивно, поки не будуть з'єднані перші два підпідземелля А і В.

На рисунку 1.14 наведено фінальну ітерацію алгоритма.

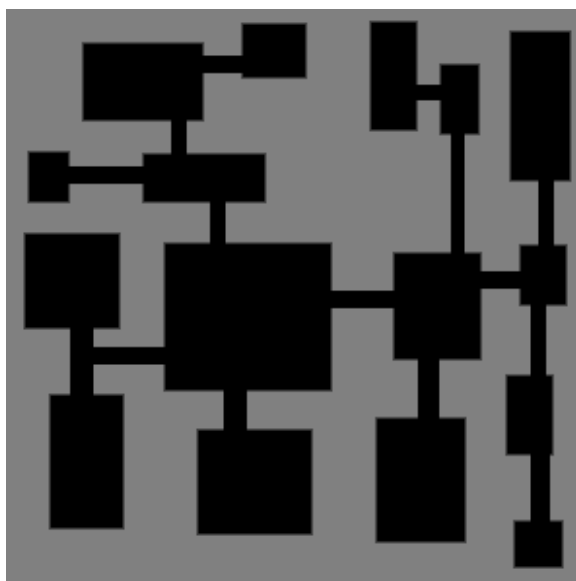


Рисунок 1.14 – Фінальна ітерація алгоритма

Дозволяючи деяким кімнатам заповнювати весь листок, можна отримати більш різноманітні та цікаві підземелля [12].

1.2.6 Perlin worms

Шум Перліна є типом градієнтного шуму та представляє собою базову техніку для створення природно виглядаючих текстур у комп'ютерній графіці. Цей метод дозволяє генерувати візуально привабливі візерунки, що корисні для відтворення природної складності цифрових середовищ.

На основі плавної та органічної природи шуму Перліна був розроблений метод Perlin worms, який адаптує його принципи для генерації складних печерних систем із реалістичними, звивистими проходами. Використовуючи властиву шуму Перліна випадковість і плавність, Perlin worms забезпечують цікаву методику створення печерних структур із високим рівнем деталізації та природності.

Спочатку використовується шум Перліна для створення градієнтної сітки значень у діапазоні від -1 до 1.

На рисунку 1.15 наведено шум Перліна.

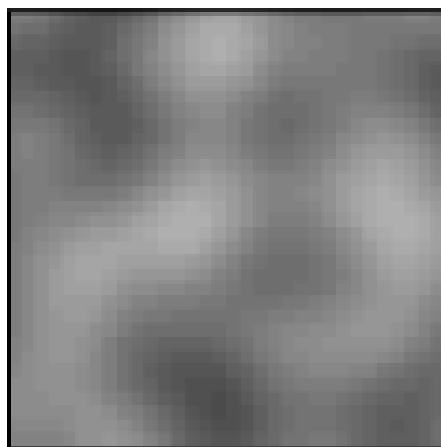


Рисунок 1.15 – Шум Перліна

Встановивши мінімальний і максимальний пороги, кожен клітинку сітки класифікували як «Повітря» або «Твердий об'єкт» залежно від того, чи

потрапляє її значення шуму в межі порогів. Такий підхід дозволяє формувати «червоподібні» структури, виділяючи переплетені сірі пікселі в шумі.

Однак отримані структури будуть часто виходити за межі сітки, створюючи візуально непривабливі обрізки. Щоб усунути цю проблему, застосовано обмеження генерації печер межами сітки, підвищуючи як візуальну, так і структурну цілісність та забезпечуючи створення складних і природно виглядаючих печерних систем.

На рисунку 1.16 наведено сгенеровану печеру.



Рисунок 1.16 – Згенерована печера

В даній реалізації було обрано кілька параметрів для контролю поведінки процесу генерації. Параметр «октави» визначає кількість шарів шуму Перліна, що генеруються, причому збільшення цього значення додає більше деталей і складності шуму. Параметри «мінімальний поріг» та «максимальний поріг» встановлюють діапазон значень, у межах якого клітинки класифікуються як «Повітря» або «Твердий об'єкт». Менший поріг призводить до більш вузьких проходів і компактнішого середовища, тоді як більший поріг створює ширші та більш відкриті простори. Параметри клітинних автоматів залишаються такими ж, як описано в попередньому розділі, з додаванням параметра ширини кордону для обмеження печери.

На рисунку 1.17 наведено Perlin Worms з різними значеннями мінімального та максимального порогів.



Рисунок 1.17 – Perlin Worms з різними значеннями мінімального та максимального порогів

Після визначення параметрів розглянемо процедуру реалізації алгоритму Midpoint Displacement. Спершу відбувається ініціалізація градієнтної сітки шуму Перліна з використанням параметра октав, а також встановлюються мінімальний та максимальний порогі і параметри клітинних автоматів, включаючи кількість ітерацій, ймовірність «живих» клітинок, порогі правил α і β , а також параметр ширини кордону.

У процедурі Perlin Worms для кожної клітинки сітки визначається її класифікація: якщо значення шуму потрапляє в межі мінімального та максимального порогів, клітинку вважають «Повітря», в іншому випадку – «Твердий об'єкт».

Далі застосовується обмеження генерації печер. Всі зовнішні клітинки кордону класифікуються як тверді, після чого ітеруються клітинки кордону у межах встановленої ширини, випадково визначаючи їх як «Повітря» або «Твердий об'єкт». Під час кожної ітерації клітинних автоматів проводиться обхід усіх клітинок сітки з застосуванням правил α та β , що визначають, чи клітинка змінює свій стан на «Повітря» або «Твердий об'єкт».

Для проведення дослідження було встановлено такі значення параметрів: три октави шуму Перліна, мінімальний поріг -0,12 та максимальний поріг 0,12. Для адаптованих клітинних автоматів: 10 ітерацій, ймовірність «живої» клітинки 0,8, якщо вихідна клітинка Perlin Worm була

тверда, інакше 0,6; поріг правила α – 4, поріг правила β – 3, ширина кордону – 4. Розміри сітки становлять 32×32 [13].

На рисунку 1.18 наведено приклади згенерованих печер, що ілюструють роботу алгоритму за цих параметрів.



Рисунок 1.18 – Приклади згенерованих печер

1.3 Висновок за розділом 1

В ході дослідження предметної області процедурної генерації ігрових рівнів були зроблені такі висновки:

- визначено та обґрунтовано основні критерії оцінювання алгоритмів процедурної генерації, а саме: поблажливість, складність, відкритість та зв'язність рівня, які дозволяють комплексно оцінити якість згенерованих структур;
- розроблено методики кількісної оцінки кожного критерію з використанням математичних формул та алгоритмічних підходів, що забезпечують об'єктивне порівняння різних методів генерації;
- оцінка поблажливості базується на аналізі частки шляхів, що призводять у глухі кути, що дозволяє визначити толерантність рівня до помилок гравця;
- оцінка складності ґрунтується на обчисленні шорсткості структури через відношення периметра до площі контурів, що відображає заплутаність та деталізацію печерних систем;
- оцінка відкритості визначається через аналіз відстані клітинок до найближчих стін, що характеризує баланс між просторими залами та вузькими проходами;
- оцінка зв'язності використовує морфологічну обробку та підрахунок ізольованих контурів, що забезпечує визначення цілісності та навігаційної зручності рівня;
- проведено систематичний огляд існуючих методів процедурної генерації, включаючи рекурсивний пошук з поверненням, рекурсивний поділ, випадкову прогулянку, клітинні автомати, бінарне розбиття простору та Perlin worms;
- встановлено, що більшість розглянутих методів мають специфічні переваги та обмеження: алгоритми рекурсивного типу забезпечують високу швидкість та простоту реалізації, але вимагають

значних ресурсів пам'яті; клітинні автомати створюють природні структури, проте схильні до генерації роз'єднаних карт;

— одним з найбільш перспективних методів серед розглянутих є Perlin worms, який поєднує органічність шуму Перліна з можливістю точного контролю параметрів генерації через систему порогів та адаптовані клітинні автомати, забезпечуючи створення візуально привабливих та функціонально збалансованих печерних систем.

2 ОБҐРУНТУВАННЯ ВИБОРУ ПРОГРАМНИХ ЗАСОБІВ РОЗРОБКИ

2.1 Вибір мови програмування для розробки системи

Вибір мови програмування є одним із найважливіших рішень на етапі проектування програмної системи, оскільки він безпосередньо впливає на швидкість розробки, продуктивність, підтримуваність та можливість подальшого розширення функціональності. Для розробки системи процедурної генерації та оцінки печерних систем було розглянуто кілька сучасних мов програмування, кожна з яких має свої особливості та сфери застосування.

2.1.1 Мова програмування Python

Python є інтерпретованою мовою програмування високого рівня, яка широко використовується в наукових дослідженнях, розробці прототипів та створенні застосунків із графічним інтерфейсом користувача. Основною перевагою Python є його читабельність, лаконічність синтаксису та динамічна типізація що роблять цю мову ідеальною для створення скриптів і швидкої розробки застосунків у багатьох сферах на більшості платформ [14].

Python підтримує множинні парадигми програмування, включаючи об'єктно-орієнтоване, процедурне та функціональне програмування. Це дозволяє розробнику обирати найбільш підходящий підхід для вирішення конкретної задачі. Мова має величезну екосистему бібліотек, які охоплюють практично всі сфери розробки програмного забезпечення.

Для роботи з багатовимірними масивами та матричними операціями в Python існує бібліотека NumPy, яка забезпечує потужний, компактний і виразний синтаксис для доступу, маніпулювання та оперування даними у векторах, матрицях та багатовимірних масивів [15]. Бібліотека підтримує векторизовані операції, що дозволяє уникнути повільних циклів Python та досягти продуктивності, близької до компільованих мов.

Для обробки зображень та комп'ютерного зору доступна бібліотека OpenCV [16]. Ця бібліотека використовується для виявлення зв'язних компонентів, обчислення контурів, морфологічних операцій та інших завдань обробки зображень, необхідних для реалізації метрик оцінювання згенерованих рівнів.

Python також має вбудовану бібліотеку Tkinter для створення графічних інтерфейсів користувача. Tkinter є кросплатформенною та не вимагає додаткової інсталяції, оскільки входить до стандартної поставки Python [17]. Це робить Python ідеальним вибором для швидкої розробки застосунків із візуальним інтерфейсом.

2.1.2 Мова програмування C++

C++ є компільованою мовою програмування, яка забезпечує максимальний контроль над апаратними ресурсами та пам'яттю [18]. Основною перевагою C++ є висока обчислювальна швидкість, що робить цю мову ідеальною для розробки систем реального часу, ігрових рушіїв та високопродуктивних обчислювальних застосунків.

Мова підтримує як низькорівневе програмування з прямим доступом до пам'яті через вказівники, так і високорівневі концепції об'єктно-орієнтованого програмування. C++ має потужну систему типів та підтримку шаблонів, що дозволяє створювати ефективний та типобезпечний код.

Для розробки графічних застосунків у C++ існує бібліотека wxWidgets, яка забезпечує кросплатформенність та багатий набір віджетів [19]. Для роботи з матрицями доступна бібліотека Eigen, яка використовує шаблони C++ для досягнення високої продуктивності.

Важливою перевагою C++ є наявність повної та оптимізованої версії бібліотеки OpenCV, розробленої саме для цієї мови. Реалізація для C++ забезпечує максимальну продуктивність завдяки використанню апаратних оптимізацій, SIMD-інструкцій та багатопотоковості. Це робить C++

найефективнішим середовищем для виконання алгоритмів комп'ютерного зору в режимі реального часу.

Однак C++ має і значні недоліки для даного проєкту. Мова вимагає ручного управління пам'яттю, що збільшує ймовірність помилок та ускладнює процес розробки. Час компіляції великих проєктів може бути значним, що сповільнює цикл розробки та тестування.

2.1.3 Мова програмування JavaScript

JavaScript є інтерпретованою мовою програмування, яка спочатку розроблялася для створення інтерактивних веб-сторінок, але згодом набула значного поширення у серверній розробці через платформу Node.js. Основною перевагою JavaScript є можливість створення веб-застосунків, які працюють безпосередньо у браузері без необхідності інсталяції додаткового програмного забезпечення [21].

Для візуалізації у JavaScript доступний Canvas API, який дозволяє малювати графіку безпосередньо на веб-сторінці. Існують також потужні бібліотеки для роботи з графікою, такі як Three.js для 3D-візуалізації та P5.js для створення інтерактивних візуальних застосунків.

Для роботи з матрицями та науковими обчисленнями в JavaScript існують бібліотеки math.js та numeric.js, однак JavaScript не має вбудованих інструментів для роботи з комп'ютерним зором, і наявні бібліотеки значно поступаються OpenCV за функціональністю та швидкодією.

Основним недоліком JavaScript для даного проєкту є обмежені можливості для наукових обчислень та відсутність стабільних бібліотек для обробки зображень. Створення десктопного застосунку потребувало б використання додаткових технологій, таких як Electron, що значно ускладнює архітектуру проєкту.

2.1.4 Порівняння мов програмування

Порівняльний аналіз розглянутих мов програмування наведено у табл. 2.1.

Таблиця 2.1 – Порівняння мов програмування для розробки системи

Критерій	Python	C++	JavaScript
Швидкість розробки	Висока	Низька	Середня
Продуктивність	Середня	Висока	Низька
Бібліотеки для наукових обчислень	NumPy	Eigen	math.js, numeric.js
Бібліотеки для комп'ютерного зору	OpenCV	OpenCV	Обмежені
Створення GUI	Tkinter	wxWidgets	Electron
Підтримка кросплатформеності	Висока	Середня	Висока

На основі проведеного аналізу для розробки системи процедурної генерації та оцінки печерних систем було обрано мову програмування Python. Ключовими факторами вибору стали наявність потужних бібліотек для наукових обчислень та обробки зображень, швидкість розробки прототипу, простота створення графічного інтерфейсу та читабельність коду. Оскільки генерація рівнів відбувається за запитом користувача, а не в реальному часі, перевага C++ за продуктивністю не є значущою.

2.2 Вибір інтегрованого середовища розробки

Вибір інтегрованого середовища розробки (IDE) суттєво впливає на зручність налагодження коду та загальну якість розробки. Для роботи з

Python існує кілька популярних IDE, кожне з яких має свої особливості та призначення.

2.2.1 Visual Studio Code

Visual Studio Code є безкоштовним редактором коду з відкритим вихідним кодом, основними перевагами якого є його простота, швидкість роботи та велика кількість розширень [21]. Редактор запускається практично миттєво та споживає значно менше оперативної пам'яті порівняно з більш складними IDE.

VS Code має вбудовану підтримку Git, що дозволяє керувати версіями коду безпосередньо з інтерфейсу редактора. Інтегрований термінал забезпечує можливість виконання команд без необхідності перемикання між вікнами.

Для роботи з Python у VS Code доступне офіційне розширення від Microsoft, яке додає підтримку налагодження, лінтингу, форматування коду, запуску тестів та роботи з віртуальними середовищами.

Система розширень VS Code налічує десятки тисяч плагінів для різних мов програмування, фреймворків та інструментів розробки. Це дозволяє налаштувати редактор під специфічні потреби конкретного проєкту.

2.2.2 PyCharm

PyCharm є повноцінним IDE для Python, розробленим компанією JetBrains. Середовище існує у двох версіях: безкоштовній Community Edition та платній Professional Edition з розширеним функціоналом [22].

Основною перевагою PyCharm є його потужні інструменти для роботи з Python. Наприклад система рефакторингу, яка дозволяє безпечно змінювати структуру коду, автоматично оновлюючи всі посилання.

PyCharm має розширені можливості налагодження, включаючи точки зупинки з умовами, перегляд стеку викликів, оцінювання виразів під час виконання програми. Вбудований профілювальник дозволяє аналізувати продуктивність.

IDE підтримує роботу з віртуальними середовищами, менеджерами пакетів, системами контролю версій, базами даних та веб-фреймворками. Professional Edition додатково включає підтримку наукових бібліотек, Jupyter Notebook та інструменти для веб-розробки.

Основним недоліком PyCharm є його значні вимоги до апаратних ресурсів. IDE споживає багато оперативної пам'яті та процесорного часу, що може призводити до сповільнення роботи на слабких комп'ютерах. Час запуску PyCharm є значно довшим порівняно з більш простими редакторами.

2.2.3 Jupyter Notebook

JupyterLab — це веб-орієнтоване інтерактивне середовище розробки для ноутбуків, коду та даних. Його гнучкий інтерфейс дає змогу користувачам налаштовувати й упорядковувати робочі процеси в галузях дата-саєнсу, наукових обчислень, комп'ютерної журналістики та машинного навчання. Модульний дизайн дає можливість розширювати й збагачувати функціональність за допомогою додатків [23].

Основною особливістю Jupyter є інтерактивність. Код виконується у комірках, і результати відображаються безпосередньо під комірками. Це дозволяє швидко експериментувати з кодом, перевіряти гіпотези та візуалізувати результати без необхідності повного перезапуску програми.

Jupyter підтримує вбудовану візуалізацію графіків та зображень, що робить його ідеальним інструментом для дослідницької роботи. Можливість додавання тексту у форматі Markdown дозволяє створювати документи, які поєднують код, результати та пояснення.

Однак Jupyter Notebook не підходить для розробки повноцінних застосунків. Середовище не має інструментів налагодження, рефакторингу та інших функцій, необхідних для розробки складних програмних систем. Jupyter краще використовувати для прототипування, експериментів та аналізу даних.

2.2.4 Результат порівняння середовищ розробки

Порівняльний аналіз розглянутих середовищ розробки наведено у табл. 2.2.

Таблиця 2.2 – Порівняння інтегрованих середовищ розробки

Критерій	VS Code	PyCharm	Jupyter
Швидкість виконання коду	Висока	Середня	Висока
Споживання пам'яті	Низьке	Високе	Середнє
Можливості налагодження	Добре	Відмінно	Обмежене
Інтерактивність	Низька	Низька	Висока
Функціонал для розробки застосунків	Добре	Відмінно	Погано
Вартість	Безкоштовно	Community — безкоштовно, Professional — платно	Безкоштовно

На основі проведеного аналізу для розробки системи було обрано Visual Studio Code. Ключовими факторами вибору стали швидкість роботи,

низьке споживання ресурсів, зручність інтерфейсу та достатній набір функцій для складності розроблюваного застосунку.

2.3 Огляд використаних бібліотек Python

Успішна реалізація системи процедурної генерації та оцінки печерних систем неможлива без використання спеціалізованих бібліотек, які забезпечують необхідну функціональність для роботи з масивами даних, обробки зображень та створення графічного інтерфейсу.

2.3.1 Бібліотека NumPy

У світі Python масиви NumPy є стандартним представленням числових даних і дозволяють ефективно виконувати числові обчислення мовою високого рівня. Вони надають підтримку багатовимірних масивів та матриць, а також велику колекцію математичних функцій для ефективної роботи з цими структурами даних.

Основним об'єктом NumPy є `ndarray` – багатовимірний масив фіксованого розміру з елементами одного типу [24]. На відміну від вбудованих списків Python, масиви NumPy зберігаються у безперервних блоках пам'яті, що забезпечує значно вищу швидкість доступу до елементів та виконання операцій.

У проєкті NumPy використовується для представлення сітки лабіринту. Функція `np.ones()` створює масив, заповнений одиницями, що представляють стіни, а функція `np.zeros()` створює масив з нулів для порожніх клітинок. Ці функції приймають параметр `dtype`, який визначає тип даних елементів масиву. Використання типу `int` замість `float` дозволяє зменшити споживання пам'яті вдвічі.

Функція `np.random.choice()` використовується для випадкової ініціалізації сітки в алгоритмі клітинних автоматів. Вона дозволяє вказати

ймовірності для кожного можливого значення через параметр `p`, що забезпечує контроль над щільністю початкового заповнення.

Метод `np.argwhere()` повертає координати всіх елементів масиву, які задовольняють певну умову. У проєкті він використовується для знаходження всіх порожніх клітинок під час обчислення метрики поблажливості. Результатом є двовимірний масив, де кожен рядок містить координати однієї клітинки.

Функція `np.sum()` обчислює суму елементів масиву. При роботі з булевими масивами вона підраховує кількість елементів зі значенням `True`, що використовується для обчислення різних метрик. Параметр `axis` дозволяє виконувати підсумовування вздовж певної осі для багатовимірних масивів.

NumPy підтримує векторизовані операції, які виконуються на рівні C без необхідності явних циклів у Python. Наприклад, вираз `grid == 0` створює булевий масив того ж розміру, де кожен елемент вказує, чи дорівнює відповідний елемент вихідного масиву нулю. Така операція виконується в десятки разів швидше, ніж еквівалентний цикл на чистому Python.

Бібліотека також надає функції для роботи з випадковими числами через модуль `np.random`. Функція `shuffle()` перемішує елементи масиву на місці, що використовується в алгоритмі рекурсивного пошуку для випадкового вибору напрямку руху. Функція `randint()` генерує випадкові цілі числа у заданому діапазоні, що застосовується для вибору початкових координат та позицій розділення в алгоритмах генерації.

2.3.2 Бібліотека OpenCV

OpenCV є відкритою бібліотекою комп'ютерного зору та машинного навчання, яка містить понад 2500 оптимізованих алгоритмів. Бібліотека розроблена на мові C++ для максимальної продуктивності, але має зручні обгортки для Python.

Функція `cv2.connectedComponents()` виконує аналіз зв'язності зображення, виявляючи окремі зв'язні компоненти. Вона приймає бінарне зображення, де ненульові пікселі вважаються об'єктами, а нульові – фоном. Функція повертає кількість знайдених компонентів та масив міток, де кожен піксель має мітку компоненти, до якої він належить.

У проєкті ця функція використовується для обчислення метрики зв'язності. Спочатку сітка лабіринту перетворюється на бінарне зображення, де порожні клітинки мають значення 255, а стіни – 0. Потім `connectedComponents()` знаходить всі окремі печерні системи. Якщо виявлено більше однієї компоненти, це означає, що рівень є роз'єднаним.

Для обчислення метрики складності використовується функція `cv2.findContours()`, яка виявляє контури об'єктів на зображенні. Контур представляє собою криву, що з'єднує всі граничні точки об'єкта. Функція може працювати в різних режимах, заданих параметром `mode`.

Режим `cv2.RETR_TREE` будує повну ієрархію контурів, включаючи зовнішні контури (батьківські) та внутрішні (дочірні). Параметр `method` визначає метод апроксимації контуру. Значення `cv2.CHAIN_APPROX_SIMPLE` стискає горизонтальні, вертикальні та діагональні сегменти, залишаючи тільки їхні кінцеві точки, що зменшує споживання пам'яті [25].

Функція повертає список контурів та масив ієрархії. Кожен елемент ієрархії містить чотири значення: індекс наступного контуру того ж рівня, індекс попереднього контуру, індекс першого дочірнього контуру та індекс батьківського контуру. Значення -1 означає відсутність відповідного контуру.

Для кожного знайденого контуру можна обчислити його характеристики. Функція `cv2.contourArea()` повертає площу контуру, а `cv2.arcLength()` обчислює периметр. Параметр `closed` вказує, чи є контур замкненим. Ці значення використовуються для обчислення показника шорсткості структури печери.

Метрика відкритості обчислюється за допомогою функції `cv2.distanceTransform()`, яка створює карту відстаней. Для кожного ненульового пікселя вхідного зображення функція обчислює відстань до найближчого нульового пікселя. Параметр `distanceType` визначає метрику відстані [26].

У проєкті використовується манхеттенська відстань `cv2.DIST_L1`, яка обчислюється як сума абсолютних різниць координат. Параметр `maskSize` визначає розмір маски для обчислення відстані. Значення 3 означає використання маски 3×3 , яка забезпечує баланс між точністю та швидкістю.

Результатом `distanceTransform()` є масив дійсних чисел, де кожен елемент представляє відстань відповідного пікселя до найближчої стіни. Клітинки з відстанню більше 1 вважаються відкритими просторами, а клітинки з відстанню 1 знаходяться поруч зі стінами і представляють вузькі проходи.

2.3.3 Бібліотека Tkinter

Tkinter є стандартною бібліотекою Python для створення графічних інтерфейсів користувача. Вона базується на інструментарії Tk, розробленому для мови Tcl, але надає зручний об'єктно-орієнтований інтерфейс для Python.

Основою кожного застосунку Tkinter є кореневе вікно, яке створюється класом Tk. Це вікно служить контейнером для всіх інших віджетів та керує головним циклом обробки подій. Метод `mainloop()` запускає цей цикл, який очікує на події від користувача та операційної системи.

Клас `Frame` створює контейнер для групування віджетів. Фреми можуть бути вкладеними один в одного, що дозволяє створювати складні макети інтерфейсу. Параметр `padding` додає відступи всередині фрейму, покращуючи зовнішній вигляд інтерфейсу.

`LabelFrame` є спеціалізованим фреймом, який має видиму рамку та текстовий заголовок. Він використовується для візуального групування

пов'язаних елементів управління. У проєкті `LabelFrame` застосовується для розділення області візуалізації лабіринту та області відображення метрик.

Віджет `Label` відображає текст або зображення. Він підтримує різні шрифти, кольори та вирівнювання. У проєкті `Label` використовується для відображення назв метрик та їхніх значень. Параметр `font` дозволяє задати гарнітуру, розмір та стиль шрифту.

`Canvas` є найбільш потужним віджетом `Tkinter` для малювання графіки. Він надає методи для створення різних геометричних примітивів: ліній, прямокутників, овалів, багатокутників та тексту. У проєкті `Canvas` використовується для візуалізації згенерованих лабіринтів.

Метод `create_rectangle()` створює прямокутник на полотні. Він приймає координати лівого верхнього та правого нижнього кутів, а також параметри `fill` (колір заповнення) та `outline` (колір обведення). Повернуте значення є ідентифікатором створеного об'єкта, який можна використовувати для подальших операцій з ним.

Метод `delete()` видаляє об'єкти з полотна. Параметр `"all"` видаляє всі об'єкти, що корисно для очищення `Canvas` перед малюванням нового лабіринту. Метод `wininfo_width()` повертає поточну ширину віджета у пікселях, що використовується для динамічного масштабування візуалізації [27].

Віджет `Spinbox` надає поле введення з кнопками збільшення та зменшення значення. Параметри `from_` та `to` визначають діапазон допустимих значень. Параметр `textvariable` пов'язує `Spinbox` зі змінною `Tkinter`, яка автоматично оновлюється при зміні значення.

Класи `IntVar` та `StringVar` представляють змінні `Tkinter`, які можуть бути пов'язані з віджетами. Вони надають методи `get()` для отримання значення та `set()` для його встановлення. Використання змінних `Tkinter` дозволяє створювати реактивні інтерфейси, де зміна одного елемента автоматично відображається в інших.

`Combobox` з модуля `ttk` створює випадальний список для вибору одного з декількох варіантів. Параметр `values` задає список доступних опцій,

a state="readonly" забороняє ручне редагування значення. Це забезпечує, що користувач може вибрати тільки один з передбачених алгоритмів генерації.

Button створює кнопку, яка виконує певну дію при натисканні. Параметр command вказує функцію, яка буде викликана при натисканні кнопки. У проєкті кнопка "Генерувати" запускає процес створення нового лабіринту з обраними параметрами.

Менеджери геометрії Tkinter визначають розташування віджетів у вікні. Метод pack() розташовує віджети послідовно у порядку їх створення. Параметр side визначає, з якої сторони батьківського контейнера розміщувати віджет. Параметр fill вказує, чи повинен віджет розтягуватися для заповнення доступного простору.

Метод grid() розташовує віджети в табличній структурі з рядками та стовпцями. Параметри row та column визначають позицію віджета в таблиці. Параметр sticky вказує, до яких сторін комірки прив'язувати віджет, використовуючи сторони світу: "w" (west), "e" (east), "n" (north), "s" (south) [28].

2.4 Висновок за розділом 2

У даному розділі було проведено аналіз та обґрунтування вибору програмних засобів для розробки системи процедурної генерації та оцінки печерних систем. На основі порівняльного аналізу було прийнято наступні рішення.

Для реалізації системи обрано мову програмування Python, яка забезпечує оптимальний баланс між швидкістю розробки та продуктивністю. Основними факторами вибору стали наявність бібліотек, які можуть реалізувати обрані методи генерації та робити наукові обчислення та обробляти зображення для візуалізації результатів генерації.

Як інтегроване середовище розробки обрано Visual Studio Code завдяки його швидкодії, низькому споживанню ресурсів та достатньому

набору функцій для розробки застосунків середньої складності. Система розширень VS Code дозволяє гнучко налаштувати редактор під потреби даного проекту.

Для реалізації системи використано три ключові бібліотеки. NumPy забезпечує ефективну роботу з багатовимірними масивами та швидкі математичні операції. OpenCV надає потужні алгоритми обробки зображень для обчислення метрик якості згенерованих рівнів. Tkinter дозволяє створити інтуїтивний графічний інтерфейс користувача без додаткових залежностей.

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Архітектура системи процедурної генерації

3.1.1 Загальна структура проєкту

Система процедурної генерації та оцінки печерних систем побудована за принципом модульної архітектури, що забезпечує чітке розділення відповідальності між компонентами та полегшує подальше розширення функціональності. Проєкт складається з трьох основних модулів, кожен з яких відповідає за окремий аспект роботи системи.

Перший модуль MazeGenerator відповідає за генерацію лабіринтів та печерних структур. Він інкапсулює всі алгоритми процедурної генерації, забезпечуючи уніфікований інтерфейс для створення різноманітних типів рівнів. Модуль працює з внутрішнім представленням лабіринту у вигляді двовимірного масиву, де кожен елемент вказує на тип клітинки.

Другий модуль MetricsCalculator призначений для обчислення метрик якості згенерованих рівнів. Він аналізує структуру лабіринту та повертає числові оцінки за чотирма критеріями. Модуль використовує алгоритми обробки зображень та графів для комплексного аналізу просторових характеристик.

Третій модуль MazeApp реалізує графічний інтерфейс користувача. Він забезпечує взаємодію з користувачем, відображає згенеровані лабіринти та результати обчислення метрик. Модуль керує життєвим циклом застосунку та координує роботу інших компонентів.

Таке розділення дозволяє незалежно модифікувати кожен компонент без впливу на інші частини системи. Наприклад, можна додати новий алгоритм генерації, не змінюючи модуль обчислення метрик чи інтерфейс користувача.

3.1.2 Взаємодія між компонентами

Взаємодія між компонентами системи відбувається за чітко визначеними інтерфейсами, що забезпечує слабке зв'язування та високу підтримуваність коду. Процес генерації та оцінювання лабіринту складається з послідовності операцій, кожна з яких виконується відповідним модулем.

Користувач взаємодіє з системою через графічний інтерфейс, встановлюючи параметри генерації: розміри сітки та обраний алгоритм. При натисканні кнопки генерації інтерфейс створює екземпляр класу генератора з вказаними параметрами та викликає відповідний метод алгоритму.

Генератор виконує процедурну генерацію та повертає двовимірний масив, що представляє структуру лабіринту. Цей масив зберігається в інтерфейсі як поточний стан системи. Далі інтерфейс передає згенерований масив модулю візуалізації для графічного відображення.

Паралельно з візуалізацією інтерфейс ініціює процес обчислення метрик. Для кожної метрики викликається відповідний статичний метод класу калькулятора, який приймає масив лабіринту та повертає числове значення оцінки. Отримані значення відображаються в окремій панелі інтерфейсу.

Взаємодія компонентів системи наведена на рисунку 3.1.

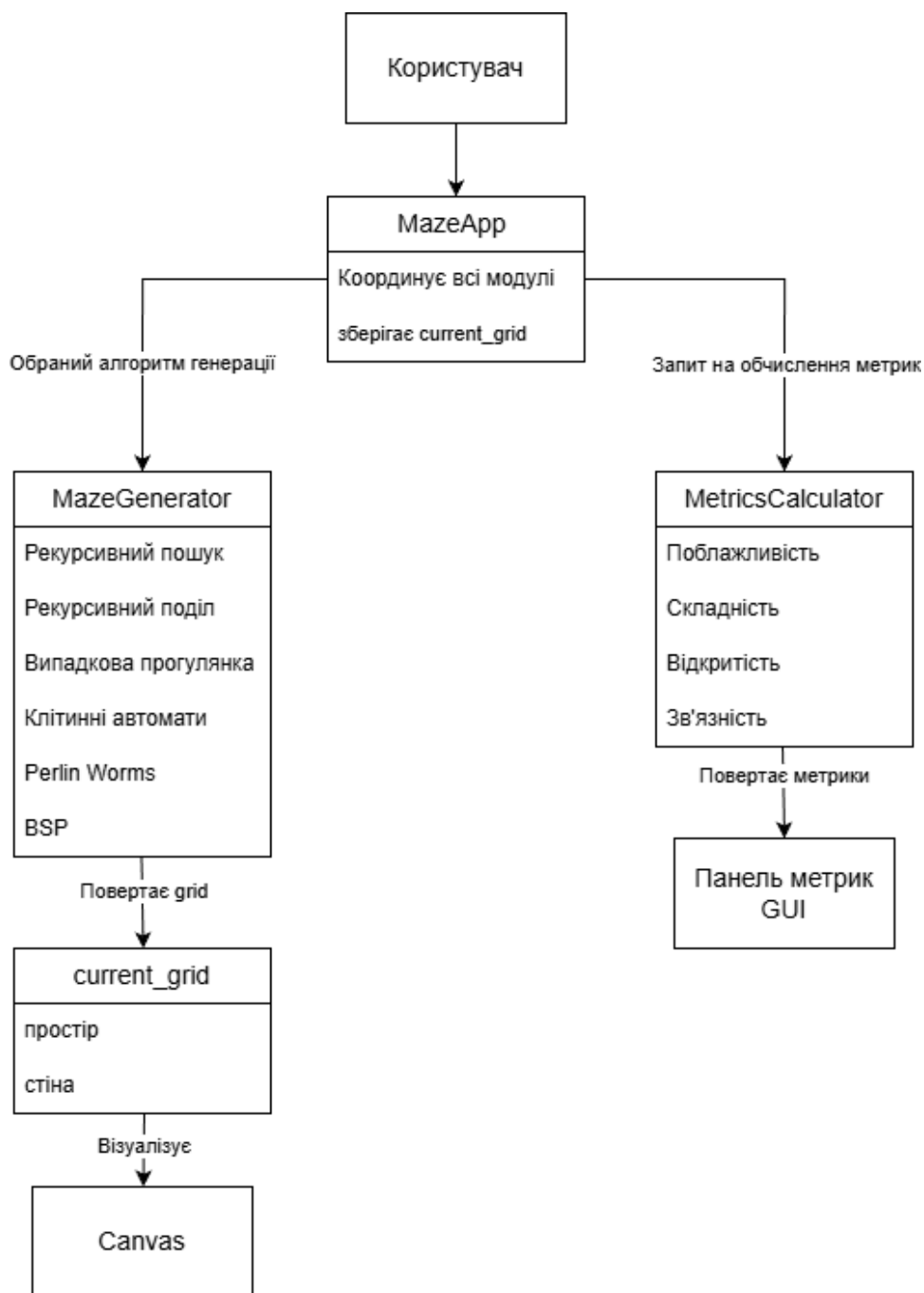


Рисунок 3.1 – Діаграма взаємодії компонентів системи

Важливою особливістю архітектури є незалежність модуля обчислення метрик від способу генерації лабіринту. Калькулятор працює виключно з готовою структурою даних, не маючи інформації про те, яким алгоритмом вона була створена. Це дозволяє використовувати одні й ті самі метрики для оцінювання результатів різних алгоритмів генерації.

3.1.3 Діаграма класів

Об'єктно-орієнтована структура системи представлена трьома основними класами, кожен з яких має чітко визначену відповідальність та набір методів для виконання своїх функцій.

Клас `MazeGenerator` відповідає за створення лабіринтів різними алгоритмами. Він містить атрибути `width` та `height`, що визначають розміри генерованої сітки, а також атрибут `grid` для зберігання поточного стану лабіринту. Конструктор класу ініціалізує ці атрибути та створює початковий масив, заповнений стінами.

Клас надає шість публічних методів, кожен з яких реалізує окремий алгоритм генерації: `recursive_backtracking`, `recursive_division`, `random_walk`, `cellular_automata`, `perlin_worms` та `bsp_dungeon`. Усі ці методи мають однаковий інтерфейс - не приймають параметрів і повертають двовимірний масив `NumPy`.

Додатково клас містить чотири приватних допоміжних методи. Метод `_count_neighbors` обчислює кількість сусідніх стін для заданої клітинки. Метод `_connect_caves` з'єднує роз'єднані печерні системи коридорами. Метод `_create_corridor` створює прохід між двома точками. Метод `_generate_perlin_noise` генерує сітку шуму Перліна для алгоритму `Perlin Worms`.

Клас `MetricsCalculator` реалізований як колекція статичних методів для обчислення метрик якості. Він не має атрибутів екземпляра і не потребує створення об'єктів. Кожна метрика обчислюється окремим статичним методом: `calculate_leniency`, `calculate_complexity`, `calculate_openness` та `calculate_connectivity`.

Усі методи метрик мають однаковий інтерфейс - приймають двовимірний масив `NumPy` та повертають дійсне число в діапазоні від 0 до 1. Така уніфікація спрощує процес обчислення та відображення метрик в інтерфейсі користувача.

Клас `MazeApp` реалізує графічний інтерфейс та координує роботу інших компонентів. Він містить посилання на кореневе вікно Tkinter, об'єкт `Canvas` для малювання лабіринту, словник для зберігання віджетів відображення метрик, а також атрибути для зберігання поточного лабіринту та обчислених метрик.

Метод `setup_ui` створює всі елементи інтерфейсу: поля введення параметрів, кнопки управління, полотно для візуалізації та мітки для відображення метрик. Метод `generate_maze` обробляє подію натискання кнопки генерації, створює відповідний лабіринт та ініціює його візуалізацію. Метод `display_maze` відображає структуру лабіринту на полотні `Canvas`. Метод `calculate_metrics` обчислює всі метрики для поточного лабіринту та оновлює відповідні мітки.

Взаємодія між класами наведена на рисунку 3.2.

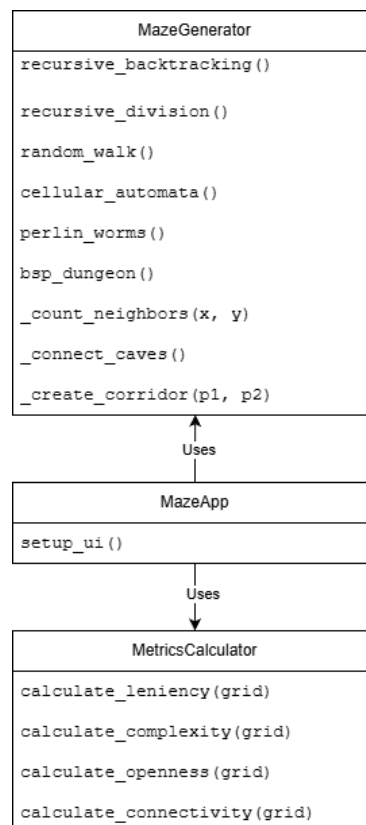


Рисунок 3.2 – Діаграма класів системи процедурної генерації

Основні класи системи та їх відповідальність наведені у таблиці 3.1.

Таблиця 3.1 – Основні класи системи та їх відповідальність

Клас	Відповідальність	Ключові методи
MazeGenerator	Генерація лабіринтів різними алгоритмами	recursive_backtracking, cellular_automata, perlin_worms
MetricsCalculator	Обчислення метрик якості	calculate_leniency, calculate_complexity, calculate_connectivity
MazeApp	Графічний інтерфейс та координація	setup_ui, generate_maze, display_maze, calculate_metrics

3.2 Реалізація класу MazeGenerator

3.2.1 Структура класу та основні методи

Клас MazeGenerator є центральним компонентом системи, що інкапсулює всю логіку процедурної генерації лабіринтів та печерних структур. Архітектура класу побудована таким чином, щоб забезпечити максимальну гнучкість та можливість легкого додавання нових алгоритмів генерації без зміни існуючого коду.

При ініціалізації класу створюється внутрішнє представлення лабіринту у вигляді двовимірного масиву NumPy. Використання NumPy забезпечує високу продуктивність операцій з масивами та зручні засоби для маніпуляції даними. Початково весь масив заповнюється одиницями, що представляють стіни, а алгоритми генерації поступово перетворюють певні клітинки на порожні простори.

Кожен алгоритм генерації реалізований як окремий метод класу, що дозволяє легко переключатися між різними підходами до створення лабіринтів. Усі методи генерації слідуєть єдиному контракту - вони не

приймають зовнішніх параметрів, використовують внутрішні атрибути класу та повертають згенерований масив.

Така уніфікація інтерфейсу спрощує інтеграцію з графічним інтерфейсом користувача. Незалежно від обраного алгоритму, інтерфейс може викликати відповідний метод однаковою способом та очікувати результат у стандартному форматі.

3.2.2 Алгоритм рекурсивного пошуку

Метод `recursive_backtracking` реалізує класичний алгоритм генерації досконалих лабіринтів, що гарантує існування єдиного шляху між будь-якими двома точками без циклів. Алгоритм базується на методі пошуку в глибину з поверненням назад.

Генерація починається з ініціалізації сітки, повністю заповненої стінами. Випадково обирається початкова клітинка з непарними координатами, що перетворюється на порожній простір. Використання непарних координат забезпечує правильну структуру лабіринту з коридорами шириною в одну клітинку.

Алгоритм підтримує стек відвіданих клітинок для реалізації механізму повернення назад. На кожному кроці з поточної позиції випадково обирається один з чотирьох можливих напрямків руху. Якщо сусідня клітинка в обраному напрямку ще не відвідана, створюється прохід до неї шляхом видалення стіни між поточною та сусідньою клітинкою.

Ключовою особливістю реалізації є використання кроку розміром у дві клітинки. Це означає, що алгоритм переміщується через одну клітинку, а потім видаляє стіну між початковою та кінцевою позицією. Такий підхід автоматично гарантує, що всі коридори матимуть ширину в одну клітинку, а стіни - також одну клітинку.

Коли з поточної клітинки неможливо зробити жоден хід до невідвіданої сусідньої клітинки, алгоритм виконує повернення назад до

попередньої клітинки зі стеку. Процес продовжується доти, доки стек не стане порожнім, що означає відвідування всіх досяжних клітинок.

Результатом роботи алгоритму є лабіринт з довгими звивистими коридорами та високою звивістю. Відсутність циклів робить такі лабіринти особливо складними для проходження, оскільки кожна помилка може призвести до тривалого блукання глухими кутами.

3.2.3 Алгоритм рекурсивного поділу

Метод `recursive_division` реалізує протилежний підхід до генерації лабіринтів – замість видалення стін алгоритм поступово додає їх до початково порожнього простору. Цей метод створює лабіринти з більш правильною структурою та чіткими кімнатами.

Генерація починається з порожньої сітки, де всі клітинки представляють вільний простір. Далі простір рекурсивно ділиться стінами на менші підобласті. На кожному рівні рекурсії обирається напрямок поділу - горизонтальний або вертикальний.

Вибір напрямку поділу залежить від пропорцій поточної підобласті. Якщо висота значно перевищує ширину, обирається горизонтальне розділення. Якщо ширина більша за висоту - вертикальне. При приблизно рівних розмірах напрямок обирається випадково. Така стратегія запобігає створенню надто вузьких коридорів.

Після вибору напрямку визначається позиція стіни. Стіна розміщується на непарній координаті для забезпечення коректної структури лабіринту. Важливо, щоб стіна не розташовувалася занадто близько до меж підобласті, інакше одна з утворених частин буде занадто малою.

Під час створення стіни в ній обов'язково залишається один прохід. Позиція проходу обирається випадково серед непарних координат вздовж стіни. Це забезпечує зв'язність лабіринту - можливість дістатися з будь-якої точки до будь-якої іншої.

Після поділу простору стіною процес рекурсивно повторюється для кожної з утворених підобластей. Рекурсія припиняється, коли розміри підобласті стають занадто малими для подальшого поділу. Мінімальний розмір підобласті становить три клітинки, що дозволяє розмістити стіну та прохід.

3.2.4 Алгоритм випадкової прогулянки

Метод `random_walk` реалізує один з найпростіших підходів до генерації печероподібних структур. Алгоритм імітує випадкове блукання в просторі, формуючи проходи в масиві.

Початковий стан сітки представляє повністю заповнений простір стінами. Алгоритм починає роботу з центральної точки сітки, яка перетворюється на порожній простір. Вибір центру як стартової точки забезпечує симетричне розширення печери в усіх напрямках.

На кожній ітерації алгоритм випадково обирає один з чотирьох кардинальних напрямків руху - вгору, вниз, ліворуч або праворуч. Якщо наступна клітинка в обраному напрямку знаходиться в межах сітки, алгоритм переміщується до неї. Якщо клітинка була стіною, вона перетворюється на порожній простір, і лічильник заповнених клітинок збільшується.

Важливою особливістю реалізації є можливість повторного відвідування вже порожніх клітинок. Це призводить до створення просторів різної ширини - деякі ділянки траєкторії випадкової прогулянки можуть перетинатися, утворюючи ширші зали, тоді як інші залишаються вузькими коридорами.

Алгоритм продовжує роботу доти, доки кількість порожніх клітинок не досягне заданого відсотка від загальної площі сітки. Параметр заповнення за замовчуванням встановлений на рівні 40%, що забезпечує баланс між відкритими просторами та твердими ділянками.

Результатом роботи алгоритму є органічна печероподібна структура з різноманітними проходами та залами. Випадковість траєкторії створює природний вигляд, проте не гарантує рівномірного розподілу порожніх просторів по сітці. Іноді можуть утворюватися ізольовані ділянки біля країв сітки.

3.2.5 Алгоритм клітинних автоматів

Метод `cellular_automata` використовує принципи клітинних автоматів для створення складних органічних структур, що нагадують природні печерні системи. Алгоритм базується на простих правилах, які застосовуються ітеративно до кожної клітинки сітки.

Процес генерації починається з випадкової ініціалізації сітки. Кожна клітинка незалежно встановлюється як стіна або порожній простір відповідно до заданої ймовірності. Параметр початкової ймовірності контролює щільність заповнення - значення 0.45 означає, що приблизно 45% клітинок будуть порожніми, а 55% - стінами.

Після ініціалізації до сітки послідовно застосовуються правила клітинного автомата. На кожній ітерації для кожної клітинки підраховується кількість сусідніх стін у вікні 3×3 . Правила визначають, чи повинна клітинка змінити свій стан на наступній ітерації.

Правило для стіни: якщо клітинка є стіною і має чотири або більше сусідніх стін, вона залишається стіною. В іншому випадку перетворюється на порожній простір. Це правило поступово видаляє ізольовані стіни та згладжує контури печерних систем.

Правило для порожнього простору: якщо клітинка є порожньою і має п'ять або більше сусідніх стін, вона перетворюється на стіну. В іншому випадку залишається порожньою. Це правило заповнює невеликі порожнини та зміцнює структуру стін.

Особливістю реалізації є обробка країв сітки. Клітинки, що знаходяться за межами сітки, вважаються стінами для цілей підрахунку сусідів. Це запобігає утворенню проходів біля країв та забезпечує замкненість печерної системи.

Кількість ітерацій правил клітинного автомата контролює рівень згладженості результату. За замовчуванням виконується п'ять ітерацій, що забезпечує достатнє видалення випадкового шуму при збереженні складної структури. Більше ітерацій призводить до більш гладких та округлих печер.

Після застосування правил клітинного автомата викликається допоміжний метод для з'єднання роз'єднаних печерних систем. Це необхідно, оскільки випадкова ініціалізація та локальні правила можуть призвести до утворення декількох ізольованих печер.

3.2.6 Алгоритм Perlin Worms

Метод `perlin_worms` поєднує шум Перліна з адаптованими правилами клітинних автоматів для створення складних печерних систем із звивистими червоподібними проходами. Алгоритм складається з кількох етапів обробки, кожен з яких вносить свій внесок у фінальну структуру.

Перший етап полягає у генерації сітки шуму Перліна. Допоміжний метод `_generate_perlin_noise` створює плавне градієнтне поле значень у діапазоні від -1 до 1. Параметр октав визначає кількість шарів шуму різної частоти, які накладаються один на одного для створення багат шарової текстури.

Генерація шуму Перліна базується на інтерполяції випадкових градієнтних векторів у вузлах регулярної сітки. Для кожної точки обчислюється відстань до чотирьох найближчих вузлів та виконується білінійна інтерполяція відповідних градієнтів. Функція згладжування забезпечує плавні переходи між значеннями.

Другий етап полягає у пороговій обробці шуму Перліна. Для кожної клітинки сітки перевіряється, чи потрапляє значення шуму в межі між мінімальним та максимальним порогами. Клітинки, що відповідають цій умові, перетворюються на порожні простори, формуючи червоподібні структури.

Значення порогів критично впливають на характер результату. Вузький діапазон порогів створює тонкі звивисті проходи, тоді як широкий діапазон призводить до утворення більших відкритих просторів. Параметри за замовчуванням встановлені на рівні -0.12 та 0.12 , що забезпечує баланс між тонкими коридорами та доступними залами.

Третій етап забезпечує обмеження печери межами сітки. Клітинки на краях та в межах заданої ширини кордону примусово встановлюються як стіни. Це запобігає виходу печерних структур за межі сітки та створює візуально завершений вигляд.

Четвертий етап застосовує адаптовані правила клітинних автоматів для згладжування та покращення структури. На відміну від базового алгоритму клітинних автоматів, тут правила застосовуються тільки до внутрішньої частини сітки, виключаючи область кордону. Це зберігає цілісність меж печери.

Кількість ітерацій клітинних автоматів встановлена на рівні десяти, що забезпечує достатнє згладжування при збереженні характерних червоподібних форм. Результатом є печерні системи з органічним виглядом, що поєднують складність шуму Перліна з чіткістю клітинних автоматів.

Етапи роботи алгоритму наведені на рисунку 3.3.

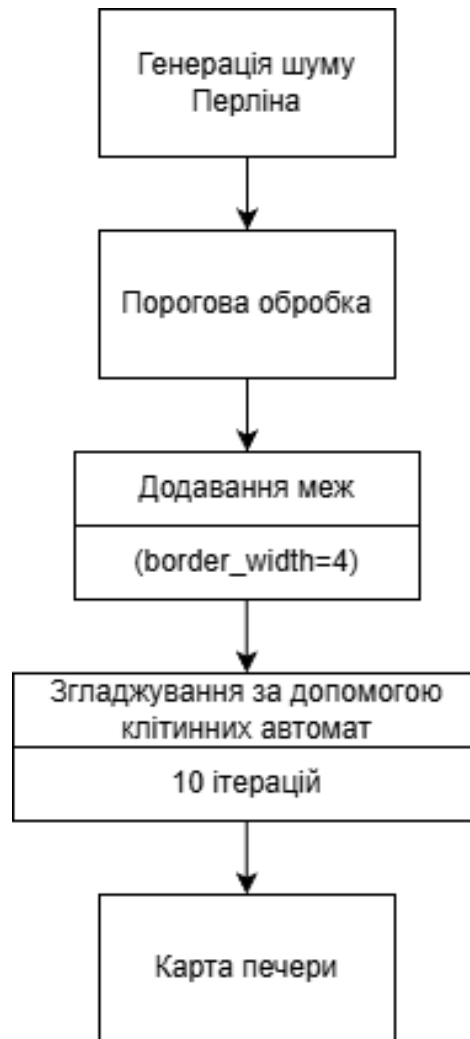


Рисунок 3.3 – Етапи роботи алгоритму Perlin Worms

3.2.7 Алгоритм BSP підземель

Метод `bsp_dungeon` реалізує алгоритм бінарного розбиття простору для створення структурованих підземель з кімнатами та коридорами. На відміну від органічних печер, цей алгоритм генерує більш архітектурні структури з чіткими прямокутними приміщеннями.

Основою алгоритму є рекурсивне розбиття простору на менші прямокутні регіони. Процес починається з кореневого регіону, що охоплює всю сітку. На кожному рівні рекурсії регіон ділиться на два підрегіони горизонтальною або вертикальною лінією.

Вибір напрямку поділу залежить від пропорцій регіону. Якщо висота значно перевищує ширину, використовується горизонтальне розділення.

Якщо ширина більша - вертикальне. При близьких розмірах напрямок обирається випадково. Така стратегія забезпечує створення регіонів з відносно збалансованими пропорціями.

Позиція лінії поділу вибирається випадково в межах допустимого діапазону. Важливо залишити достатньо місця з обох боків для розміщення кімнат мінімального розміру. Мінімальна відстань від країв регіону визначається параметром `min_room_size` плюс додатковий запас.

Рекурсія продовжується до досягнення максимальної глибини або до того моменту, коли регіон стає занадто малим для подальшого поділу. Глибина рекурсії встановлена на рівні чотирьох, що забезпечує достатню деталізацію без надмірної фрагментації простору.

У кожному листковому регіоні, де рекурсія припиняється, створюється кімната випадкового розміру. Розміри кімнати обмежені параметрами `min_room_size` та `max_room_size`, які за замовчуванням встановлені на рівні 4 та 10 клітинок відповідно. Позиція кімнати всередині регіону також обирається випадково.

Після створення всіх кімнат алгоритм з'єднує їх коридорами. З'єднання відбувається ієрархічно відповідно до структури розбиття. Спочатку з'єднуються кімнати в сусідніх листкових регіонах, потім з'єднання поширюються на вищі рівні ієрархії.

Для з'єднання двох кімнат обирається по одній випадковій точці з центральної області кожної кімнати. Між цими точками створюється коридор L-подібної форми. Спочатку коридор йде в одному напрямку до рівня другої точки, а потім повертає під прямим кутом для досягнення цільової точки.

Напрямок першого сегмента коридору обирається випадково - або спочатку горизонтальний, потім вертикальний, або навпаки. Така варіативність додає різноманітності структурі підземелля. Ширина коридорів становить одну клітинку, що створює відчуття тісних проходів між просторими кімнатами.

Ієрархічна структура BSP-дерева для підземелля наведена на рисунку 3.4.

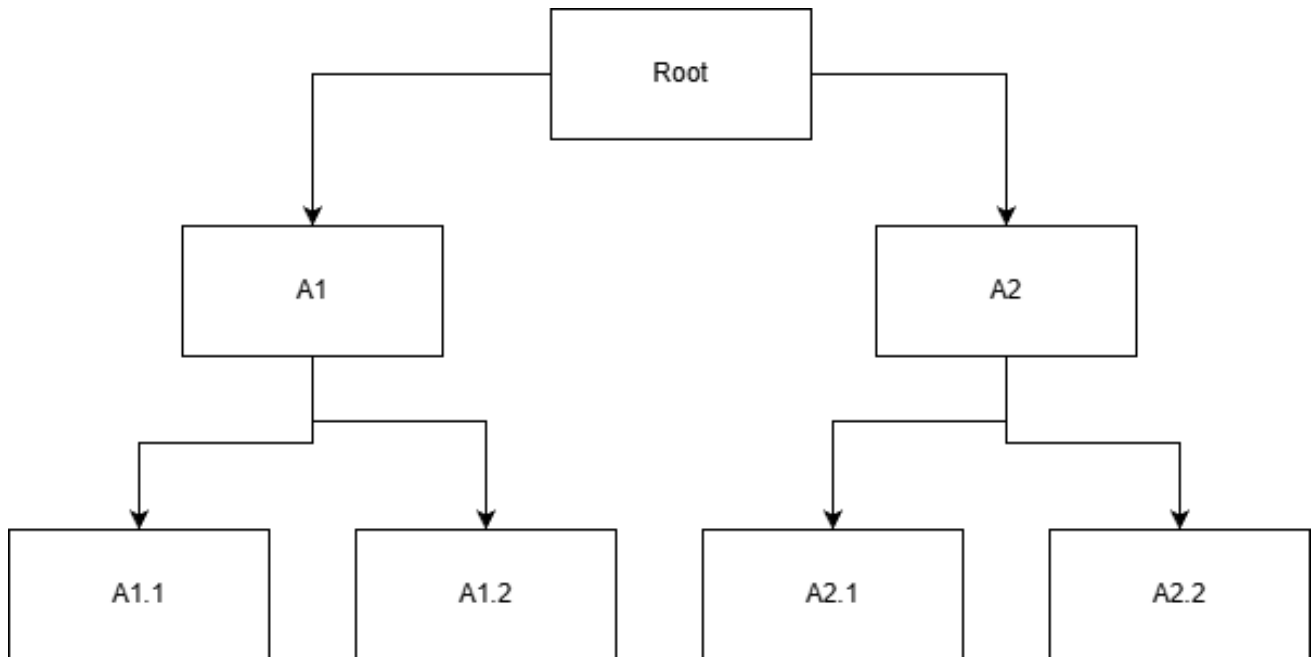


Рисунок 3.4 – Ієрархічна структура BSP-дерева для підземелля

3.2.8 Допоміжні методи

Ефективна робота основних алгоритмів генерації забезпечується набором допоміжних методів, кожен з яких виконує специфічну функцію, що використовується в різних частинах класу.

Метод `_count_neighbors` обчислює кількість сусідніх стін для заданої клітинки. Він перевіряє всі вісім сусідніх позицій у вікні 3×3 з центром у цільовій клітинці. Для кожної сусідньої позиції метод перевіряє, чи знаходиться вона в межах сітки і чи є стіною.

Особливістю реалізації є обробка клітинок за межами сітки. Якщо сусідня позиція виходить за межі масиву, вона вважається стіною. Це важливо для коректної роботи алгоритмів клітинних автоматів біля країв сітки, де необхідно запобігти утворенню проходів назовні.

Метод повертає загальну кількість сусідніх стін, включаючи віртуальні стіни за межами сітки. Це значення використовується в правилах

клітинних автоматів для визначення, чи повинна клітинка змінити свій стан. Діапазон можливих значень від 0 до 8.

Метод `_connect_caves` призначений для з'єднання роз'єднаних печерних систем в єдину зв'язну структуру. Він використовує алгоритм пошуку зв'язних компонентів з бібліотеки `OpenCV` для виявлення окремих печер.

Спочатку метод перетворює внутрішнє представлення лабіринту в бінарне зображення, де порожні клітинки мають значення 255, а стіни - 0. Функція `connectedComponents` аналізує це зображення та присвоює унікальну мітку кожній зв'язній компоненті порожніх клітинок.

Якщо виявлено більше однієї компоненти, метод обчислює центральну точку кожної печери та послідовно з'єднує найближчі пари. Для кожної печери центр визначається як серединна точка масиву координат, що належать до цієї компоненти.

Процес з'єднання відбувається ітеративно. На кожній ітерації метод знаходить пару печер з мінімальною манхеттенською відстанню між їхніми центрами. Між цими центрами створюється коридор за допомогою методу `_create_corridor`. Після з'єднання мітки обох печер об'єднуються, і процес повторюється до залишення однієї компоненти.

Метод `_create_corridor` створює прохід між двома точками на сітці. Він приймає координати початкової та кінцевої точок і формує L-подібний коридор, що їх з'єднує. Випадково обирається порядок сегментів - спочатку горизонтальний чи вертикальний.

Під час створення коридору метод перетворює на порожні простори всі клітинки вздовж обраного шляху. Перевірка меж сітки перед зміною значення клітинки запобігає помилкам при створенні коридорів біля країв або за межами масиву.

Допоміжні методи класу `MazeGenerator` наведені у таблиці 3.2.

Таблиця 3.2 – Допоміжні методи класу MazeGenerator

Метод	Призначення	Параметри	Повертає
<code>_count_neighbors</code>	Підрахунок сусідніх стін	x, y: координати клітинки	int: кількість стін
<code>_connect_caves</code>	З'єднання роз'єднаних печер	—	—
<code>_create_corridor</code>	Створення коридору між точками	p1, p2: координати точок	—
<code>_generate_perlin_noise</code>	Генерація шуму Перліна	octaves: кількість октав	ndarray: сітка шуму

3.3 Реалізація класу MetricsCalculator

3.3.1 Структура класу для обчислення метрик

Клас MetricsCalculator організований як колекція статичних методів, що не вимагає створення екземплярів та підтримки внутрішнього стану. Такий підхід відповідає функціональній природі обчислення метрик, де кожна метрика є чистою функцією від вхідного масиву до числового результату.

Відсутність стану в класі забезпечує потокобезпечність та дозволяє викликати методи незалежно один від одного без ризику конфліктів. Кожен метод приймає двовимірний масив NumPy, що представляє структуру лабіринту, та повертає нормалізоване значення метрики в діапазоні від 0 до 1.

Уніфікація інтерфейсу методів спрощує інтеграцію з іншими компонентами системи. Графічний інтерфейс може викликати всі методи однаковою способом, передаючи їм поточний лабіринт та отримуючи стандартизовані оцінки. Це також полегшує додавання нових метрик у майбутньому.

Внутрішня реалізація кожного методу використовує алгоритми обробки зображень та аналізу графів з бібліотек OpenCV та NumPy. Складні обчислення розбиті на послідовність простіших операцій, що підвищує читабельність коду та спрощує налагодження.

3.3.2 Метод обчислення поблажливості

Метод `calculate_leniency` оцінює толерантність рівня до помилок гравця через аналіз глухих кутів. Метрика базується на припущенні, що шляхи, які ведуть у глухі кути, збільшують складність навігації та покарання за неправильні рішення.

Обчислення починається з перевірки базових умов. Метод підраховує загальну кількість порожніх клітинок у лабіринті. Якщо їх занадто мало, метрика повертає нульове значення, оскільки оцінювання поблажливості не має сенсу для майже порожніх структур.

Далі виконується аналіз зв'язності для виділення найбільшої зв'язної компоненти порожніх клітинок. Використовується алгоритм `connectedComponents` з OpenCV, що присвоює унікальну мітку кожній групі з'єднаних порожніх клітинок. Це дозволяє відкинути дрібні ізольовані порожнини та зосередитися на основній печерній системі.

Після виділення основної компоненти визначаються початкова та кінцева точки для аналізу шляхів. Початковою точкою обирається перша клітинка з масиву координат компоненти, кінцевою – остання. Хоча цей вибір є довільним, він забезпечує стабільні результати для однієї структури.

Ключовим етапом є визначення глухих кутів через аналіз альтернативних шляхів. Для кожної клітинки з вибірки спочатку знаходиться найкоротший шлях від початку до цієї клітинки за допомогою алгоритму пошуку в ширину. Потім клітинки цього шляху блокуються, і перевіряється існування альтернативного шляху від клітинки до кінцевої точки.

Якщо альтернативного шляху не існує після блокування прямого маршруту, клітинка класифікується як глухий кут. Це означає, що гравець, який пройшов до цієї клітинки, не зможе дістатися до цілі без повернення назад тим самим шляхом.

З міркувань продуктивності аналізується не вся множина клітинок, а репрезентативна вибірка. Розмір вибірки встановлюється як 20% від загальної кількості клітинок компоненти, з обмеженням між 100 та 500 клітинками. Це забезпечує баланс між точністю оцінки та швидкодією.

Підсумкове значення поблажливості обчислюється як частка клітинок, що не є глухими кутами. Метрика визначається за формулою (3.1):

$$\text{leniency} = 1 - \frac{\text{dead_ends}}{\text{checked_cells}}, \quad (3.1)$$

де `dead_ends` — кількість клітинок, для яких не існує альтернативного шляху до цілі;

`checked_cells` — загальна кількість проаналізованих клітинок вибірки.

Значення нормалізується до діапазону від 0 до 1, де 1 означає повну відсутність глухих кутів, а 0 — що всі проаналізовані клітинки ведуть у безвихідь.

Алгоритм визначення глухого кута через блокування шляху наведено на рисунку 3.5.

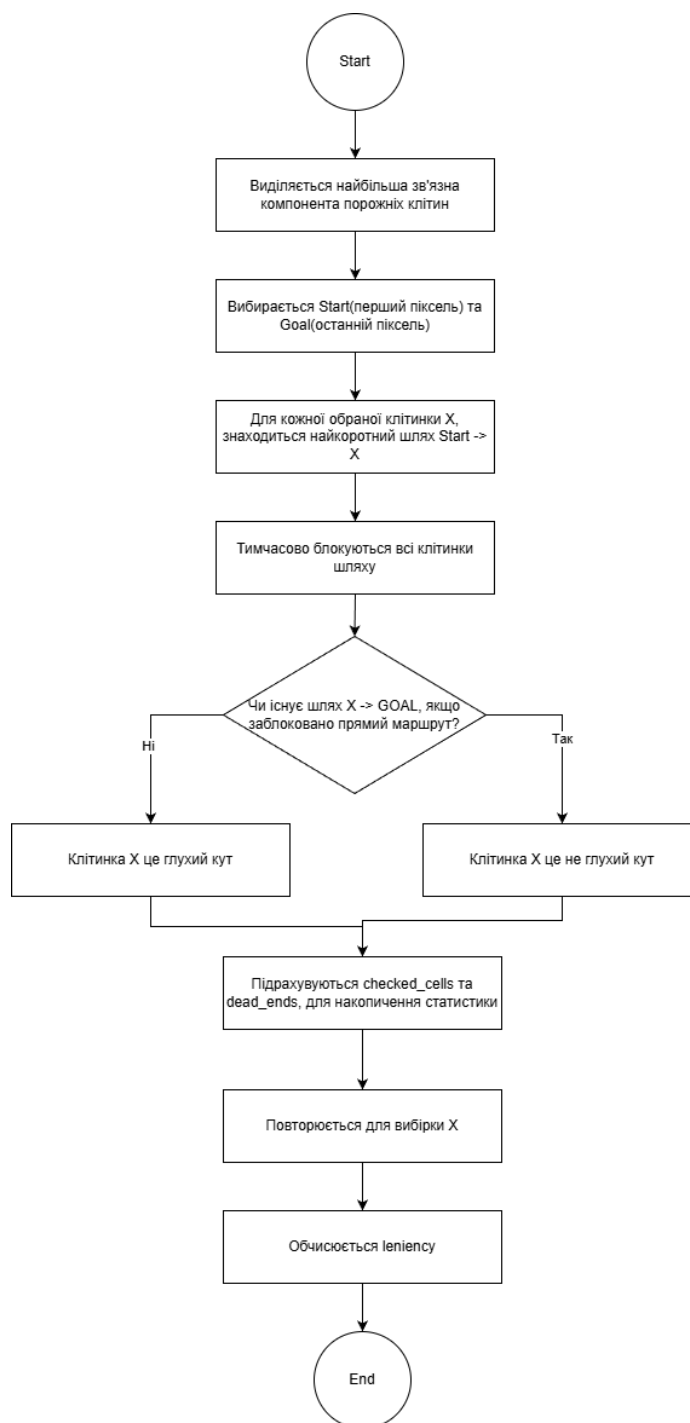


Рисунок 3.5 – Алгоритм визначення глухого кута через блокування шляху

3.3.3 Метод обчислення складності

Метод `calculate_complexity` оцінює рівень заплутаності структури через аналіз шорсткості контурів печери. Метрика базується на геометричних характеристиках меж печерної системи.

Обчислення починається з перетворення внутрішнього представлення лабіринту в бінарне зображення. Порожні клітинки перетворюються на білі пікселі зі значенням 255, а стіни - на чорні з значенням 0. Це зображення використовується для пошуку контурів.

Функція `findContours` з `OpenCV` виявляє всі замкнені контури на бінарному зображенні. Параметр `RETR_TREE` будує повну ієрархію контурів, що дозволяє розрізняти зовнішні контури печери та внутрішні отвори. Параметр `CHAIN_APPROX_SIMPLE` стискає контури, залишаючи тільки кінцеві точки сегментів.

Після виявлення контурів вони класифікуються на батьківські та дочірні на основі ієрархічної структури. Батьківські контури не мають батька в ієрархії і представляють зовнішні межі печери. Дочірні контури мають батька і представляють внутрішні порожнини або острівці стін всередині печери.

Дрібні контури з площею менше 10 пікселів відфільтровуються як шум. Це запобігає впливу артефактів дискретизації на результати обчислень. Для кожного контуру, що залишився, обчислюється площа за допомогою функції `contourArea` та периметр за допомогою `arcLength`.

Загальна площа структури визначається як різниця між сумою площ батьківських контурів та сумою площ дочірніх. Це дає фактичну площу твердої частини печерної системи, виключаючи порожнини. Загальний периметр обчислюється як сума периметрів всіх контурів.

Максимальна можлива шорсткість визначається для ідеально гладкого контуру - кола з такою самою площею. Радіус такого кола обчислюється з площі, а максимальна шорсткість встановлюється як половина цього радіуса. Це значення служить еталоном для нормалізації.

Фактична шорсткість обчислюється як відношення площі структури до загального периметра. Чим більша площа при тому самому периметрі, тим більш компактною є структура. Чим більший периметр при тій самій площі, тим більш звивистими є контури.

Підсумковий бал складності отримується діленням фактичної шорсткості на максимальну. Результат нормалізується до діапазону від 0 до 1, де більші значення відповідають більш складним структурам з нерівними межами.

3.3.4 Метод обчислення відкритості

Метод `calculate_openness` оцінює співвідношення між просторами залами та вузькими коридорами в структурі лабіринту, використовуючи аналіз відстаней до найближчих стін. Ідея метрики полягає в тому, що чим більше клітинок розташовано на певній відстані від стін, тим відкритішою є структура, і навпаки — велика частка клітинок біля стін вказує на вузькі, коридорні області.

На першому етапі вихідна матриця лабіринту перетворюється у бінарне зображення, де прохідні клітинки позначаються одиницями, а стіни — нулями. Це забезпечує коректну роботу алгоритму трансформації відстаней. Далі за допомогою функції `distanceTransform` з бібліотеки `OpenCV` створюється карта відстаней, у якій кожен елемент визначає найкоротшу манхеттенську відстань (L1) до найближчої стіни. Така форма метрики узгоджується з дискретною природою ґратки, де рух дозволено лише по вертикалі та горизонталі.

Після побудови карти відстаней аналізується розподіл значень. Клітинки з відстанню 1 знаходяться безпосередньо біля стін і відповідають вузьким проходам. Клітинки з відстанню понад 1 утворюють зони, які можна вважати більш відкритими. На основі цих даних визначаються два ключові показники: загальна кількість порожніх клітинок та кількість клітинок, що розташовані на відстані більше одиниці від стіни.

Підсумкова метрика `openness` обчислюється як відношення відкритих клітинок до загальної кількості прохідних клітин та нормалізується до діапазону $[0; 1]$. Високе значення вказує на наявність великих просторів, тоді

як низьке — на домінування вузьких коридорів. Таким чином, метрика дозволяє кількісно оцінити ступінь відкритості структури, що є важливим параметром у генерації ігрових рівнів.

Показник відкритості визначається за формулою (3.2):

$$\text{openness} = \frac{A}{B}, \quad (3.2)$$

де A – кількість відкритих клітинок;

B – кількість усіх прохідних клітинок.

Алгоритм обчислення параметра openness наведено на рисунку 3.6.



Рисунок 3.6 – Алгоритм обчислення оцінки відкритості структури

3.3.5 Метод обчислення зв'язності

Метод `calculate_connectivity` оцінює ступінь цілісності печерної системи через аналіз кількості незалежних компонентів. Метрика визначає, наскільки добре з'єднані різні частини структури між собою.

Обчислення починається з перетворення лабіринту в бінарне зображення аналогічно іншим метрикам. Функція `connectedComponents` виявляє всі окремі групи з'єднаних порожніх клітинок, присвоюючи унікальну мітку кожній компоненті.

Якщо виявлено тільки одну або дві компоненти, метрика повертає ідеальне значення 1.0. Дві компоненти вважаються прийнятними, оскільки одна з них є фоном, а друга - основною печерною системою. Відсутність додаткових компонентів означає повну зв'язність.

Для кожної виявленої компоненти обчислюється її розмір як кількість клітинок. Дуже малі компоненти можуть бути артефактами генерації або незначними порожнинами. Встановлюється поріг у 10% від розміру найбільшої компоненти - тільки компоненти більші за цей поріг вважаються значущими.

Підрахунок значущих компонентів дає кількість окремих печерних систем у структурі. Якщо виявлено тільки одну значущу компоненту, це означає повну зв'язність. Кожна додаткова компонента зменшує оцінку зв'язності.

Формула обчислення зв'язності використовує експоненційне зменшення залежно від кількості компонентів. Значення обчислюється як 1.0 ділене на 1.5 в степені кількості компонентів мінус один. Така формула забезпечує швидке зменшення оцінки при збільшенні фрагментації.

Результат обмежується максимальним значенням 1.0 для випадків з однією компонентою. Мінімальне теоретичне значення наближається до нуля при великій кількості ізольованих печер. При цьому значення рідко опускаються нижче 0.3 для структур з прийнятною генерацією.

Методи обчислення метрик наведені в таблиці 3.3.

Таблиця 3.3 – Методи обчислення метрик та їх характеристики

Метрика	Метод	Основний алгоритм
Поблажливість	calculate_leniency	Пошук в ширину
Складність	calculate_complexity	Аналіз контурів
Відкритість	calculate_openness	Трансформація відстані
Зв'язність	calculate_connectivity	Зв'язні компоненти

3.4 Реалізація графічного інтерфейсу користувача

3.4.1 Структура класу MazeApp

Клас MazeApp інкапсулює всю логіку графічного інтерфейсу та координує взаємодію між компонентами системи. Архітектура класу побудована навколо кореневого вікна Tkinter та організована за принципом відділення представлення від логіки.

При ініціалізації класу створюється головне вікно застосунку з встановленими розмірами та заголовком. Початковий розмір 1200×800 пікселів забезпечує достатній простір для відображення згенерованого лабіринту та панелі метрик без необхідності прокручування.

Клас підтримує внутрішній стан, що включає посилання на поточний згенерований лабіринт та обчислені значення метрик. Атрибут `current_grid` зберігає двовимірний масив останнього згенерованого лабіринту, тоді як словник `metrics` містить пари ключ-значення для кожної метрики.

Важливим аспектом архітектури є розділення процесу створення інтерфейсу та обробки подій. Метод `setup_ui` викликається під час ініціалізації для створення всіх елементів інтерфейсу, тоді як методи обробки подій викликаються у відповідь на дії користувача.

3.4.2 Створення елементів інтерфейсу

Метод `setup_ui` організовує створення всіх візуальних компонентів інтерфейсу, розподіляючи їх на логічні групи для зручності сприйняття та використання. Структура інтерфейсу складається з трьох основних областей: панель управління, область візуалізації та панель метрик.

Панель управління розміщується в верхній частині вікна та містить елементи для налаштування параметрів генерації. Використовується фрейм з додатковими відступами для покращення візуального відокремлення від інших частин інтерфейсу.

Елементи управління організовані в горизонтальний ряд за допомогою менеджера геометрії `grid`. Кожен елемент має текстову мітку, що пояснює його призначення. Для введення числових значень ширини та висоти використовуються віджети `Spinbox`, що обмежують діапазон допустимих значень від 20 до 100.

`Spinbox` автоматично валідує введення користувача, дозволяючи тільки цілі числа в заданому діапазоні. Використання змінних Tkinter типу `IntVar` забезпечує двосторонній зв'язок між віджетом та внутрішнім представленням значення. Зміна значення у віджеті автоматично оновлює змінну, і навпаки.

Для вибору алгоритму генерації використовується випадаючий список `Combobox`. Параметр `state="readonly"` робить список тільки для читання, запобігаючи введенню довільних значень користувачем. Список заповнюється назвами всіх доступних алгоритмів українською мовою для зручності користувача.

Кнопка генерації розміщена праворуч від елементів налаштування та пов'язана з методом `generate_maze` через параметр `command`. При натисканні кнопки автоматично викликається відповідний метод обробки події.

Основна частина вікна розділена на дві панелі, які організовані як окремі `LabelFrame` з описовими заголовками. Ліва панель призначена для

відображення згенерованого лабіринту, права - для виведення обчислених метрик.

Область візуалізації містить віджет Canvas з білим фоном та початковими розмірами 600×600 пікселів. Canvas налаштований на автоматичне розширення для заповнення доступного простору при зміні розмірів вікна. Це забезпечує адаптивність інтерфейсу до різних розмірів екрану.

Панель метрик містить чотири пари міток для відображення назв метрик та їхніх значень. Мітки назв використовують жирний шрифт розміром 11 пунктів для чіткого виділення, тоді як мітки значень мають більший розмір 14 пунктів для легкого читання результатів.

Посилання на мітки значень зберігаються в словнику `metric_labels` з ключами, що відповідають назвам метрик. Це дозволяє легко оновлювати відображувані значення після обчислення метрик без необхідності пошуку відповідних віджетів.

Структура графічного інтерфейсу наведена на рисунку 3.7.

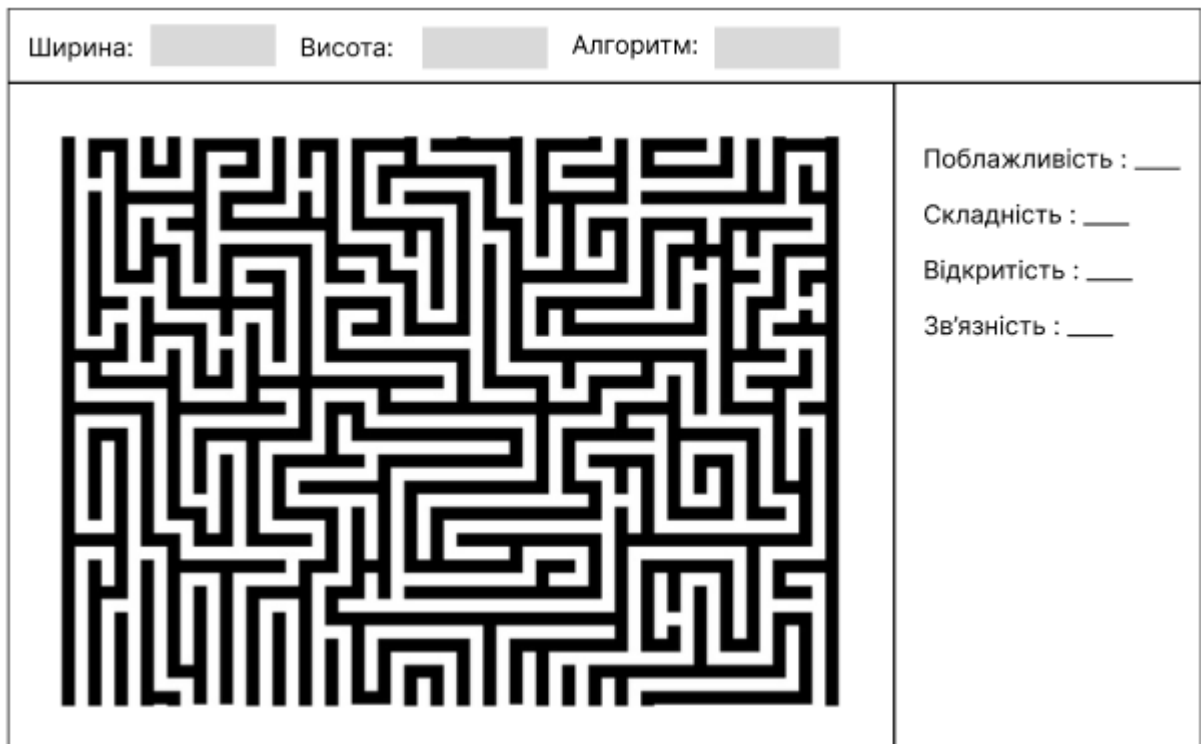


Рисунок 3.7 – Структура графічного інтерфейсу системи

3.4.3 Обробка подій користувача

Метод `generate_maze` є центральним обробником події натискання кнопки генерації. Він координує послідовність операцій від отримання параметрів до відображення результатів.

Процес обробки починається з отримання поточних значень параметрів з відповідних віджетів через пов'язані змінні Tkinter. Метод `get()` повертає актуальне значення, введене або вибране користувачем. Ці значення використовуються для створення екземпляра генератора з потрібними розмірами.

Залежно від обраного алгоритму викликається відповідний метод генератора. Використовується послідовність умовних операторів для співставлення назви алгоритму з відповідним методом. Кожна назва перевіряється українською мовою відповідно до того, як вона відображається у випадяючому списку.

Результат генерації зберігається в атрибуті `current_grid` для подальшого використання іншими методами. Це забезпечує збереження стану між різними операціями без необхідності передачі лабіринту як параметра.

Після успішної генерації послідовно викликаються методи `display_maze` та `calculate_metrics` для відображення результатів. Така послідовність гарантує, що користувач одразу бачить як візуальне представлення лабіринту, так і його числові характеристики.

Вся логіка обробки загорнута в блок `try-except` для перехоплення можливих помилок під час генерації. Якщо виникає виняток, користувачу відображається діалогове вікно з повідомленням про помилку. Це запобігає аварійному завершенню програми та інформує користувача про проблему.

Метод `calculate_metrics` викликає всі методи обчислення метрик з класу `MetricsCalculator`, передаючи їм поточний лабіринт. Результати

зберігаються у внутрішньому словнику та одразу відображаються в інтерфейсі через оновлення текстових позначок.

Для кожної метрики значення форматується до трьох знаків після коми для забезпечення однакового вигляду та достатньої точності. Метод `config()` оновлює текст відповідної мітки, що призводить до негайного оновлення відображення на екрані.

3.4.4 Візуалізація згенерованих лабіринтів

Метод `display_maze` відповідає за графічне представлення структури лабіринту на полотні `Canvas`. Візуалізація реалізована через малювання кольорових прямокутників для кожної клітинки сітки.

Перед початком малювання метод очищає полотно від попередніх об'єктів за допомогою методу `delete("all")`. Це забезпечує, що новий лабіринт відображається на чистому тлі без накладання на попередні результати.

Розмір кожної клітинки обчислюється динамічно на основі поточних розмірів `Canvas` та розмірів сітки лабіринту. Методи `wininfo_width()` та `wininfo_height()` повертають актуальні розміри віджета в пікселях. Розділення цих значень на відповідні розміри сітки дає розмір однієї клітинки.

Використовується мінімальне значення з ширини та висоти клітинки для забезпечення квадратної форми та уникнення спотворень. Це гарантує, що клітинки завжди мають правильні пропорції незалежно від співвідношення сторін `Canvas`.

Для кожної клітинки в сітці обчислюються координати верхнього лівого та нижнього правого кутів прямокутника. Координати визначаються множенням індексів клітинки на розмір комірки. Метод `create_rectangle()` створює заповнений прямокутник за цими координатами.

Колір заповнення визначається типом клітинки. Стіни малюються темно-сірим кольором, тоді як порожні клітинки - світло-сірим.

Використання приглушених відтінків замість чистих чорного та білого робить візуалізацію більш приємною для ока.

Параметр `outline` вимикає обведення прямокутників, створюючи суцільне зображення без видимих меж між клітинками. Це покращує візуальне сприйняття структури, особливо для великих сіток з малими клітинками.

3.5 Висновок за розділом 3

У третьому розділі було детально розглянуто реалізацію всіх компонентів системи процедурної генерації та оцінки печерних систем. Проведений аналіз архітектурних рішень та особливостей імплементації дозволяє зробити такі висновки.

Модульна архітектура системи забезпечує чітке розділення відповідальності між компонентами. Клас `MazeGenerator` – інкапсулює алгоритми генерації, клас `MetricsCalculator` – обчислення метрик, а клас `MazeApp` – графічний інтерфейс. Така організація полегшує підтримку та розширення функціональності системи.

Реалізовано шість різноманітних алгоритмів процедурної генерації, кожен з яких створює структури з унікальними характеристиками. Алгоритми рекурсивного типу генерують лабіринти з довгими коридорами, клітинні автомати та `Perlin Worms` створюють органічні печерні системи, а `BSP` забезпечує архітектурні підземелля з чіткими кімнатами.

Система метрик дозволяє об'єктивно оцінювати якість згенерованих структур за чотирма критеріями. Метрика поблажливості аналізує наявність глухих кутів через алгоритми пошуку шляхів. Метрика складності оцінює шорсткість через геометричний аналіз контурів. Метрика відкритості визначає баланс просторів через трансформацію відстані. Метрика зв'язності оцінює цілісність через виявлення зв'язних компонентів.

Графічний інтерфейс користувача забезпечує зручну взаємодію з системою. Панель управління дозволяє налаштовувати параметри генерації, область візуалізації відображає структуру лабіринту, а панель метрик показує числові оцінки. Адаптивний дизайн інтерфейсу підтримує різні розміри екрану.

Використання бібліотек NumPy та OpenCV забезпечує високу продуктивність обчислень. Векторизовані операції NumPy прискорюють роботу з масивами, а оптимізовані алгоритми OpenCV ефективно виконують обробку зображень для обчислення метрик.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Інструкція користувача та демонстрація роботи системи

4.1.1 Запуск та початкове налаштування програми

Для запуску системи процедурної генерації та оцінки печер необхідно виконати файл `main.py` у середовищі Python 3.8 або вище з встановленими бібліотеками NumPy, OpenCV та Tkinter.

При успішному запуску відкривається головне вікно застосунку розміром 1200×800 пікселів. Інтерфейс розділений на три основні області: панель управління у верхній частині, область візуалізації зліва та панель метрик справа.

Вікно наведено на рисунку 4.1.

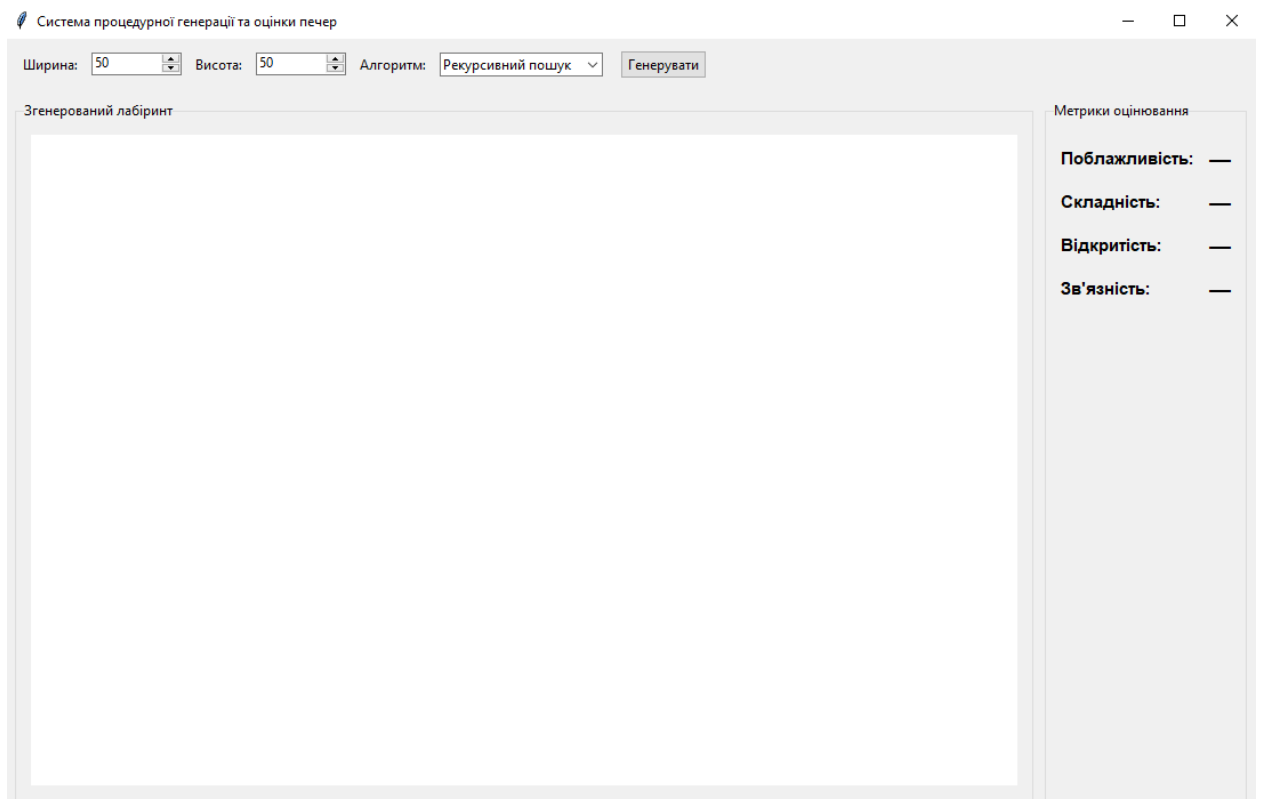


Рисунок 4.1 – Головне вікно програмного забезпечення

Панель управління містить елементи для налаштування параметрів генерації:

- поле "Ширина" – встановлює горизонтальний розмір сітки;
- поле "Висота" – встановлює вертикальний розмір сітки;
- випадаючий список "Алгоритм" – дозволяє обрати один з шести доступних методів генерації;
- кнопка "Генерувати" – запускає процес створення лабіринту з обраними параметрами.

Область візуалізації відображає згенеровану структуру, де темно-сірим кольором позначені стіни, а світло-сірим – прохідні клітинки. Панель метрик показує чотири числові оцінки згенерованої печери: поблагливість, складність, відкритість, зв'язність.

4.1.2 Процес генерації печери

Для створення нового лабіринту користувач встановлює бажані розміри сітки та обирає алгоритм з випадаючого списку. Після натискання кнопки "Генерувати" система виконує наступну послідовність операцій:

- зчитування параметрів з полів введення;
- створення екземпляра класу `MazeGenerator` із заданими розмірами;
- виклик відповідного методу генерації;
- візуалізація результату на полотні `Canvas`;
- обчислення та відображення метрик якості.

Весь процес займає від декількох мілісекунд для простих алгоритмів на малих сітках до декількох секунд для складних алгоритмів на великих розмірах. Користувач може одразу бачити результат та оцінки без затримок.

Сгенеровану печеру наведено на рисунку 4.2.

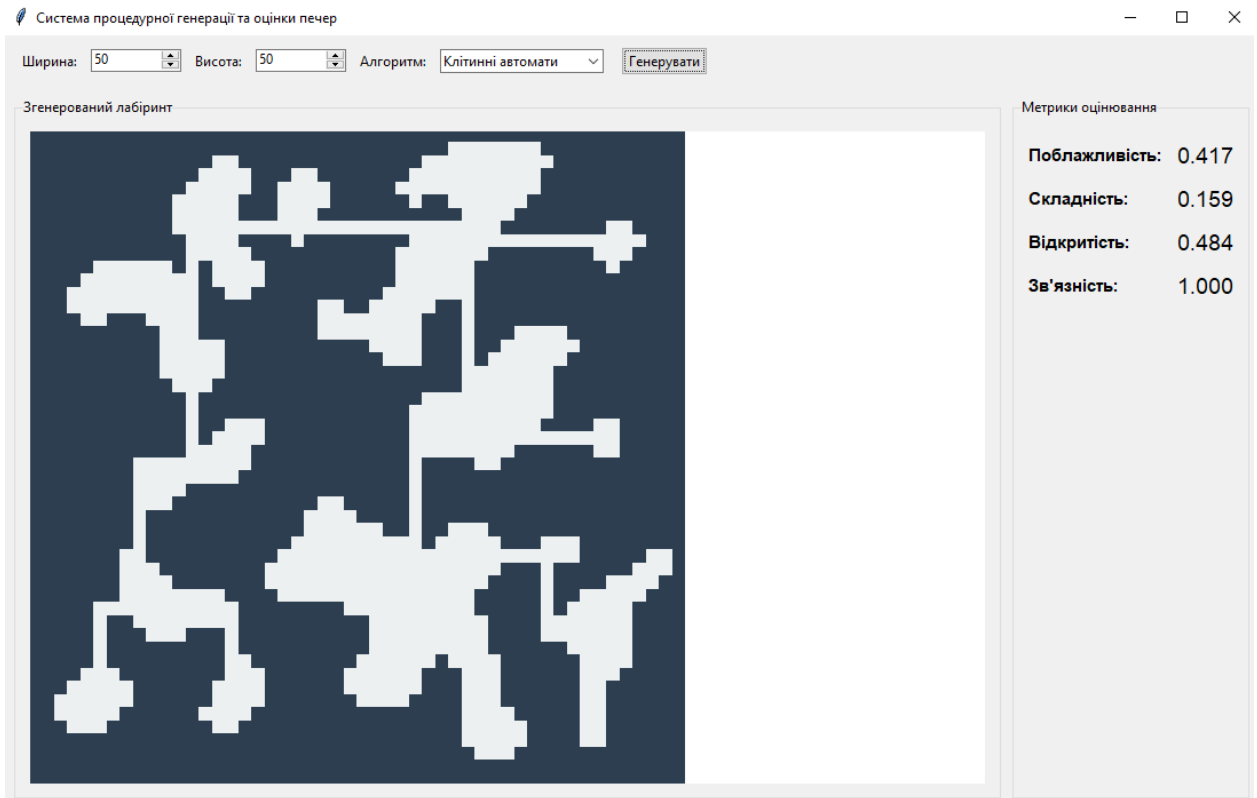


Рисунок 4.2 – Печера сгенерована за допомогою клітинних автомат

4.1.3 Інтерпретація метрик

Система обчислює чотири метрики, кожна з яких характеризує певний аспект згенерованої структури:

Поблажливість оцінює толерантність рівня до помилок гравця. Високе значення означає, що більшість шляхів ведуть до цілі і гравець може легко виправити помилки. Низьке значення вказує на велику кількість глухих кутів, де неправильний вибір напрямку призводить до тривалого блукання.

Складність відображає рівень заплутаності структури. Низьке значення відповідає простим геометричним формам з гладкими контурами. Високі значення характеризують складні структури з багатьма відгалуженнями та нерівними межами.

Відкритість вимірює баланс між просторими залами та вузькими коридорами. Низькі значення (0.0–0.3) означають переважання тісних проходів. Середні значення (0.3–0.6) вказують на збалансоване поєднання.

Високі значення (0.6–1.0) характеризують відкриті простори з великими залами.

Зв'язність оцінює цілісність печерної системи. Значення 1.0 означає повну зв'язність - всі частини структури з'єднані між собою. Нижчі значення вказують на наявність ізольованих печер або роз'єднаних компонентів.

4.2 Методика проведення експериментів

4.2.1 Мета експериментального дослідження

Метою експериментального дослідження є порівняльний аналіз шести реалізованих алгоритмів процедурної генерації за критеріями якості згенерованих структур. Дослідження спрямоване на виявлення характерних особливостей кожного алгоритму та визначення сфер їх оптимального застосування для різних типів ігор.

4.2.2 Параметри експериментів

Для забезпечення коректності порівняння всі алгоритми тестувалися з однаковими параметрами. Розмір сітки встановлено як 50×50 клітинок, що є оптимальним балансом між деталізацією структури та швидкістю обчислень.

Для кожного з шести алгоритмів виконано п'ять незалежних генерацій, що забезпечить репрезентативність вибірки.

Після кожної генерації автоматично обчислювалися чотири метрики якості. Результати фіксувалися з точністю до трьох знаків після коми. Для кожного алгоритму розраховувалися статистичні характеристики: середнє значення, мінімум, максимум та стандартне відхилення.

4.3 Результати експериментального дослідження

4.3.1 Дані генерацій для кожного алгоритму

Результати п'яти генерацій для кожного з шести алгоритмів наведено в таблицях 4.1– 4.6.

Таблиця 4.1 – Результати генерацій алгоритму рекурсивного пошуку

Поблажливість	Складність	Відкритість	Зв'язність
0.250	0.023	0.006	1.000
0.398	0.023	0.006	1.000
0.301	0.023	0.006	1.000
0.217	0.023	0.005	1.000
0.327	0.023	0.006	1.000

Таблиця 4.2 – Результати генерацій алгоритму рекурсивного поділу

Поблажливість	Складність	Відкритість	Зв'язність
0.095	0.028	0.039	1.000
0.095	0.028	0.046	1.000
0.136	0.028	0.056	1.000
0.148	0.028	0.047	1.000
0.125	0.028	0.043	1.000

Таблиця 4.3 – Результати генерацій алгоритму випадкової прогулянки

Поблажливість	Складність	Відкритість	Зв'язність
0.785	0.296	0.689	1.000
0.934	0.316	0.669	1.000
0.753	0.309	0.603	1.000
0.885	0.247	0.601	1.000
0.905	0.248	0.585	1.000

Таблиця 4.4 – Результати генерацій алгоритму клітинних автоматів

Поблажливість	Складність	Відкритість	Зв'язність
0.552	0.153	0.444	1.000
0.533	0.120	0.365	1.000
0.277	0.134	0.425	1.000
0.376	0.134	0.445	1.000
0.577	0.139	0.468	1.000

Таблиця 4.5 – Результати генерацій алгоритму Perlin Worms

Поблажливість	Складність	Відкритість	Зв'язність
0.943	0.307	0.704	1.000
0.843	0.273	0.683	1.000
0.697	0.134	0.425	1.000
0.376	0.292	0.606	0.667
0.962	0.291	0.641	1.000

Таблиця 4.6 – Результати генерацій алгоритму BSP підземель

Поблажливість	Складність	Відкритість	Зв'язність
0.605	0.195	0.532	1.000
0.812	0.220	0.529	1.000
0.591	0.183	0.524	1.000
0.342	0.195	0.557	1.000
0.333	0.160	0.533	1.000

4.3.2 Порівняльний аналіз алгоритмів

Середнє значення метрик для кожного алгоритму наведено у таблиці

4.7.

Таблиця 4.7 – Середнє значення метрик для кожного алгоритму

Алгоритм	Поблажливість	Складність	Відкритість	Зв'язність
Рекурсивний пошук	0.299	0.023	0.006	1.000
Рекурсивний поділ	0.120	0.028	0.046	1.000
Випадкова прогулянка	0.852	0.283	0.629	1.000
Клітинні автомати	0.463	0.136	0.430	1.000
Perlin Worms	0.764	0.259	0.612	0.933
BSP підземелля	0.537	0.191	0.535	1.000

4.3.3 Аналіз за метрикою поблажливості

Найвищі показники поблажливості демонструють алгоритми випадкової прогулянки (0.852) та Perlin Worms (0.764). Ці алгоритми створюють структури з великою кількістю альтернативних шляхів, що дозволяє гравцеві легко виправляти помилки та знаходити вихід.

Алгоритм рекурсивного поділу показує найнижчу поблажливість (0.120), що пояснюється його особливістю створювати чітко розділені кімнати з обмеженою кількістю проходів між ними. Рекурсивний пошук також має низьку поблажливість (0.299) через велику кількість глухих кутів у згенерованих лабіринтах.

BSP підземелля (0.537) та клітинні автомати (0.463) займають середню позицію, забезпечуючи збалансоване співвідношення між складністю навігації та можливістю виправлення помилок.

4.3.4 Аналіз за метрикою складності

Рекурсивні алгоритми (пошук та поділ) створюють найпростіші структури з точки зору геометричної складності – 0.023 та 0.028 відповідно. Їхні контури є практично ідеально прямолінійними, що пояснюється алгоритмічною природою генерації з чіткими правилами.

Алгоритм випадкової прогулянки генерує найскладніші структури (0.283) з органічними звивистими формами. Perlin Worms має схожий показник (0.259), що обумовлено використанням процедурного шуму для створення природних контурів.

BSP підземелля (0.191) та клітинні автомати (0.136) займають проміжне положення. BSP створює структури середньої складності завдяки комбінації прямокутних кімнат з L-подібними коридорами. Клітинні автомати згладжують початкову випадковість, що призводить до помірної складності.

4.3.5 Аналіз за метрикою відкритості

Найвищу відкритість демонструють випадкова прогулянка (0.629) та Perlin Worms (0.612), створюючи структури з великими відкритими просторами та широкими проходами. Такі печери нагадують природні утворення з чергуванням просторих залів та вузьких коридорів.

BSP підземелля (0.535) та клітинні автомати (0.429) створюють середньо відкриті структури. BSP забезпечує стабільну відкритість завдяки контрольованим розмірам кімнат. Клітинні автомати генерують більш варіативні результати з нерівномірним розподілом відкритих просторів.

Рекурсивні алгоритми створюють найтісніші структури – рекурсивний пошук (0.006) практично не має відкритих просторів, складаючись виключно з вузьких коридорів. Рекурсивний поділ (0.046) дещо відкритіший через наявність кімнат, але все одно залишається дуже тісним.

4.3.6 Аналіз за метрикою зв'язності

П'ять з шести алгоритмів забезпечують ідеальну зв'язність (1.000), гарантуючи повну досяжність усіх частин згенерованої структури. Це досягається через вбудовані механізми з'єднання компонентів або алгоритмічні властивості, що виключають створення ізольованих областей.

Лише Perlin Worms демонструє іноді зниження зв'язності до 0.933 у середньому з мінімумом 0.667. Це пояснюється природою алгоритму, де порогова обробка шуму Перліна може іноді призводити до утворення роз'єднаних печерних систем.

Висока стабільність зв'язності у більшості алгоритмів робить їх придатними для ігор, де критично важлива повна досяжність всіх областей рівня.

4.4 Оптимальне застосування алгоритмів

4.4.1 Алгоритми для складних викликів

Рекурсивний пошук та рекурсивний поділ ідеально підходять для ігор, де головною метою є подолання складного лабіринту. Низька поблажливість (0.120-0.299) створює значну складність, оскільки гравці часто потрапляють у глухі кути та змушені повертатися назад.

Мінімальна відкритість (0.006-0.046) забезпечує тісні коридори з обмеженою видимістю, що підвищує напруженість дослідження. Ідеальна зв'язність гарантує, що вихід завжди існує, але знайти його потрібно через методичне дослідження.

Ці алгоритми рекомендовано використовувати для:

- класичних лабіринтів у головоломках;
- *dungeon-crawlers* з акцентом на навігації;
- пазлів з провадженням лінії до виходу.

4.4.2 Алгоритми для свободи дослідження

Випадкова прогулянка та Perlin Worms створюють відкриті органічні структури, ідеальні для ігор, де акцент робиться на дослідженні, а не на подоланні лабіринту. Висока поблажливість (0.764-0.852) дозволяє гравцям вільно пересуватися без страху застрягти у безвиході.

Значна відкритість (0.612-0.629) забезпечує простір для маневру та створює відчуття великих печерних систем. Висока складність (0.259-0.283) додає візуальної цікавості та природності оточенню.

Рекомендовано для:

- рогаляйк-ігор з акцентом на дослідженні;
- survival-ігор у печерному середовищі;
- sandbox-ігор з процедурною генерацією локацій.

1.4.3 Алгоритми для збалансованого геймплею

Клітинні автомати та BSP підземелля займають середню позицію за більшістю метрик, забезпечуючи баланс між складністю та доступністю. Поблажливість на рівні 0.463-0.537 створює помірну складність – гравці будуть відчувати середній виклик, та не проводити занадто багато часу у печері.

Відкритість 0.429-0.535 забезпечує чергування тісних проходів та відкритих кімнат, що додає різноманітності геймплею. Середня складність (0.136-0.191) створює цікаву візуальну структуру без надмірної заплутаності.

BSP особливо підходить для RPG та тактичних ігор завдяки чіткій кімнатній структурі, яка природно підходить для розміщення противників, скарбів та інтерактивних об'єктів.

Клітинні автомати рекомендовано для action-RPG, де потрібні природні печери з помірною складністю навігації, але з великими областями

для боїв. Вища варіативність створює більше різноманітності між генераціями.

4.4.4 Параметри налаштування для різних контекстів

Для мобільних ігор з обмеженим екранним простором рекомендовано розміри 30×30 до 40×40 . Менші лабіринти забезпечують кращу видимість на малих екранах та швидшу генерацію.

Для десктопних ігор оптимальний діапазон становить 50×50 до 70×70 , що забезпечує достатню деталізацію при прийнятній продуктивності. Експерименти показали, що на розмірі 50×50 всі алгоритми працюють без помітних затримок.

Для масштабних відкритих рівнів можна використовувати сітки до 100×100 та більше, але з урахуванням збільшення часу генерації та споживання пам'яті.

Для створення багаторівневих підземель є можливість комбінувати різні алгоритми на різних поверхах:

- верхні рівні: BSP або рекурсивний поділ для структурованих будівель;
- середні рівні: клітинні автомати для природних печер;
- глибокі рівні: рекурсивний пошук для складних викликів.

Така прогресія забезпечує поступове зростання складності та різноманітність оточення.

4.5 Висновок за розділом 4

У четвертому розділі проведено комплексне експериментальне дослідження шести реалізованих алгоритмів процедурної генерації. Виконано 30 генерацій (по 5 для кожного алгоритму) на сітці розміром 50×50 клітинок з подальшим статистичним аналізом отриманих метрик.

Дослідження виявило чіткі відмінності між алгоритмами за всіма метриками якості. Рекурсивні методи створюють прості геометричні структури з низькою поблажливістю, що робить їх ідеальними для складних викликів. Алгоритми на основі випадковості та процедурного шуму генерують органічні відкриті структури, придатні для ігор з акцентом на дослідженні. BSP та клітинні автомати забезпечують збалансований підхід для універсального застосування.

Сформульовано оптимальні застосування кожного алгоритму для різних типів ігор. Запропоновано стратегії комбінування алгоритмів та налаштування параметрів для досягнення бажаних характеристик згенерованих рівнів.

Результати експериментів підтверджують, що розроблена система забезпечує різноманітний інструментарій для процедурної генерації з можливістю об'єктивної оцінки якості через систему метрик. Це дозволяє розробникам ігор свідомо обирати алгоритми відповідно до геймдизайн-цілей проєкту.

ВИСНОВКИ

У дипломній роботі розроблено повноцінну систему процедурної генерації та оцінювання печерних структур, яка поєднує шість алгоритмів створення рівнів та чотири метрики їх аналізу. Система має модульну архітектуру, що забезпечує розділення логіки генерації, обчислення характеристик та графічного інтерфейсу користувача. Це підвищує гнучкість, масштабованість і зручність подальшого розширення.

Проведене експериментальне дослідження продемонструвало, що різні алгоритми створюють структури з унікальними характеристиками: рекурсивні методи формують лабіринти з мінімальною відкритістю та найнижчою варіативністю, тоді як випадкова прогулянка та Perlin Worms генерують найбільш органічні та відкриті печери. Клітинні автомати та BSP-підземелля займають проміжне місце, забезпечуючи збалансовані показники за більшістю метрик.

Результати підтверджують ефективність розробленої системи як інструменту для дослідження процедурної генерації та аналізу ігрових просторів. Платформа може бути використана як основа для майбутніх проєктів у галузі геймдизайну, штучного інтелекту або автоматизованого створення рівнів, а також легко розширюється новими алгоритмами та метриками.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Statista – Video Game Market Worldwide, 2024 [Electronic resource].
– Access mode: <https://www.statista.com/topics/868/video-games/>
2. PCG Wiki – Procedural Content Generation Overview [Electronic resource]. – Access mode: <https://pcg.wikidot.com/>
3. Summerville A. et al. Procedural Content Generation via Machine Learning, 2018 [Electronic resource]. – Access mode: <https://arxiv.org/pdf/1702.00539>
4. Procedural Content Generation for Games [Electronic resource]. – Access mode: <https://arxiv.org/pdf/2201.10334>.
5. Salinas J. Procedural Dungeon Generation Using Graph Grammars [Electronic resource]. – Access mode: https://scholarworks.uark.edu/cgi/viewcontent.cgi?params=/context/csceuht/article/1138/&path_info=Salinas_Thesis.pdf.
6. Procedural Generation: A Primer for Game Devs [Electronic resource]. – Access mode: <https://www.gamedeveloper.com/design/procedural-generation-a-primer-for-game-devs>.
7. Aryanab. Maze Generation: Recursive Backtracking [Electronic resource]. – Access mode: <https://aryanab.medium.com/maze-generation-recursive-backtracking-5981bc5cc766>.
8. Buck J. Maze Generation Algorithm Recap [Electronic resource]. – Access mode: <https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap>.
9. Cellular Automata Method for Generating Random Cave-Like Levels [Electronic resource]. – Access mode: https://www.roguebasin.com/index.php/Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels.
10. Cellular Automata Cave Generation Example [Electronic resource]. – Access mode: http://pixelenvy.ca/wa/ca_cave.html.

11. Complete Roguelike Tutorial using Python3 + libtcod: BSP Dungeon Generator [Electronic resource]. – Access mode: https://www.roguebasin.com/index.php/Complete_Roguelike_Tutorial,_using_Python3%2Blibtcod,_extras#BSP_Dungeon_Generator.
12. Basic BSP Dungeon Generation [Electronic resource]. – Access mode: https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation.
13. Salinas J. Procedural Dungeon Generation Thesis [Electronic resource]. – Access mode: <https://scholarworks.uark.edu/cgi/viewcontent.cgi?article=1138&context=csceuh>.
14. Python Language Reference. Introduction [Electronic resource]. – Access mode: <https://docs.python.org/3/reference/introduction.html>.
15. Array programming with NumPy [Electronic resource] / [C. R. Harris, K. J. Millman, S. J. van der Walt et al.]. – Access mode: <https://arxiv.org/abs/2006.10256>.
16. Introduction to OpenCV-Python Tutorials [Electronic resource]. – Access mode: https://docs.opencv.org/4.x/d0/de3/tutorial_py_intro.html.
17. Shipman J. W. Tkinter 8.5 reference: a GUI for Python [Electronic resource] / J. W. Shipman. – Access mode: <https://tkdocs.com/shipman/tkinter.pdf>.
18. Stroustrup B. A Tour of C++: The Basics [Electronic resource] / B. Stroustrup. – Access mode: <https://isocpp.org/images/uploads/2-Tour-Basics.pdf>.
19. About wxWidgets [Electronic resource]. – Access mode: <https://wxwidgets.org/about/>.
20. What is JavaScript? MDN Web Docs [Electronic resource]. – Access mode: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/What_is_JavaScript.
21. Visual Studio Code. Microsoft Learn [Electronic resource]. – Access mode: <https://learn.microsoft.com/uk-ua/shows/visual-studio-code/>.
22. Unified PyCharm documentation [Electronic resource]. – Access mode: <https://www.jetbrains.com/help/pycharm/unified-pycharm.html>.

23. Project Jupyter [Electronic resource]. – Access mode: <https://jupyter.org/>.
24. NumPy ndarray Documentation [Electronic resource]. – Access mode: <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>.
25. Structural Analysis and Shape Descriptors. OpenCV Documentation [Electronic resource]. – Access mode: https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html.
26. Image Segmentation with Distance Transform and Watershed Algorithm. OpenCV Tutorials [Electronic resource]. – Access mode: https://docs.opencv.org/3.4/d2/dbd/tutorial_distance_transform.html.
27. Tkinter Canvas Widget Documentation [Electronic resource]. – Access mode: <https://tkinter-docs.readthedocs.io/en/latest/widgets/canvas.html>.
28. tkinter.ttk — Tk themed widgets [Electronic resource]. – Access mode: <https://docs.python.org/uk/3.13/library/tkinter.ttk.html>.

ДОДАТОК А
Текст програми

```

import tkinter as tk
from tkinter import ttk, messagebox
import numpy as np
import random
from PIL import Image, ImageTk
import cv2
import math
from collections import deque

class MazeGenerator:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.grid = np.ones((height, width), dtype=int)

    def recursive_backtracking(self):
        self.grid = np.ones((self.height, self.width), dtype=int)
        stack = []
        start_x, start_y = random.randrange(1, self.width, 2),
random.randrange(1, self.height, 2)
        self.grid[start_y][start_x] = 0
        stack.append((start_x, start_y))
        directions = [(0, -2), (2, 0), (0, 2), (-2, 0)]
        while stack:
            x, y = stack[-1]
            random.shuffle(directions)
            found = False
            for dx, dy in directions:
                nx, ny = x + dx, y + dy

```

```

        if 0 < nx < self.width and 0 < ny < self.height and
self.grid[ny][nx] == 1:
            self.grid[ny][nx] = 0
            self.grid[y + dy//2][x + dx//2] = 0
            stack.append((nx, ny))
            found = True
            break
    if not found:
        stack.pop()
return self.grid

```

```

def recursive_division(self):
    self.grid = np.zeros((self.height, self.width), dtype=int)
    def divide(x, y, w, h):
        if w < 3 or h < 3:
            return
        horizontal = random.choice([True, False]) if w == h else h > w
        if horizontal:
            wall_y = random.randrange(y + 1, y + h - 1, 2)
            passage_x = random.randrange(x, x + w, 2)
            for i in range(x, x + w):
                if i != passage_x:
                    self.grid[wall_y][i] = 1
            divide(x, y, w, wall_y - y)
            divide(x, wall_y + 1, w, y + h - wall_y - 1)
        else:
            wall_x = random.randrange(x + 1, x + w - 1, 2)
            passage_y = random.randrange(y, y + h, 2)
            for i in range(y, y + h):
                if i != passage_y:

```

```

        self.grid[i][wall_x] = 1
        divide(x, y, wall_x - x, h)
        divide(wall_x + 1, y, x + w - wall_x - 1, h)
    divide(0, 0, self.width, self.height)
    return self.grid

```

```

def random_walk(self, fill_percent=0.4):
    self.grid = np.ones((self.height, self.width), dtype=int)
    x, y = self.width // 2, self.height // 2
    self.grid[y][x] = 0
    target_cells = int(self.width * self.height * fill_percent)
    filled = 1
    directions = [(0, -1), (1, 0), (0, 1), (-1, 0)]
    while filled < target_cells:
        dx, dy = random.choice(directions)
        nx, ny = x + dx, y + dy
        if 0 <= nx < self.width and 0 <= ny < self.height:
            if self.grid[ny][nx] == 1:
                filled += 1
            self.grid[ny][nx] = 0
            x, y = nx, ny
    return self.grid

```

```

def cellular_automata(self, initial_prob=0.45, iterations=5):
    self.grid = np.random.choice([0, 1], size=(self.height, self.width),
p=[initial_prob, 1-initial_prob])
    for _ in range(iterations):
        new_grid = self.grid.copy()
        for y in range(self.height):
            for x in range(self.width):

```

```

        neighbors = self._count_neighbors(x, y)
        if self.grid[y][x] == 1:
            new_grid[y][x] = 1 if neighbors >= 4 else 0
        else:
            new_grid[y][x] = 1 if neighbors >= 5 else 0
    self.grid = new_grid
    self._connect_caves()
    return self.grid

```

```

def _connect_caves(self):
    binary = (1 - self.grid).astype(np.uint8) * 255
    num_labels, labels = cv2.connectedComponents(binary)
    if num_labels <= 2:
        return
    cave_centers = []
    for i in range(1, num_labels):
        coords = np.argwhere(labels == i)
        if len(coords) > 1:
            center = coords[len(coords) // 2]
            cave_centers.append((center[0], center[1], i))
    while len(cave_centers) > 1:
        min_dist = float('inf')
        best_pair = None
        for i in range(len(cave_centers)):
            for j in range(i + 1, len(cave_centers)):
                y1, x1, _ = cave_centers[i]
                y2, x2, _ = cave_centers[j]
                dist = abs(y2 - y1) + abs(x2 - x1)
                if dist < min_dist:
                    min_dist = dist

```

```

        best_pair = (i, j)
    if best_pair is None:
        break
    i, j = best_pair
    y1, x1, label1 = cave_centers[i]
    y2, x2, label2 = cave_centers[j]
    self._create_corridor((x1, y1), (x2, y2))
    labels[labels == label2] = label1
    cave_centers.pop(j)

def _generate_perlin_noise(self, octaves=3):
    def fade(t):
        return t * t * t * (t * (t * 6 - 15) + 10)
    def lerp(t, a, b):
        return a + t * (b - a)
    def gradient(h, x, y):
        vectors = [[0,1],[0,-1],[1,0],[-1,0],[1,1],[-1,1],[1,-1],[-1,-1]]
        g = vectors[h % 8]
        return g[0] * x + g[1] * y
    p = list(range(256))
    random.shuffle(p)
    p = p + p
    noise_grid = np.zeros((self.height, self.width))
    for y in range(self.height):
        for x in range(self.width):
            total = 0
            frequency = 1
            amplitude = 1
            max_value = 0
            for _ in range(octaves):

```

```

    fx = x / 20.0 * frequency
    fy = y / 20.0 * frequency
    xi = int(fx) & 255
    yi = int(fy) & 255
    xf = fx - int(fx)
    yf = fy - int(fy)
    u = fade(xf)
    v = fade(yf)
    aa = p[p[xi] + yi]
    ab = p[p[xi] + yi + 1]
    ba = p[p[xi + 1] + yi]
    bb = p[p[xi + 1] + yi + 1]
    x1 = lerp(u, gradient(aa, xf, yf), gradient(ba, xf - 1, yf))
    x2 = lerp(u, gradient(ab, xf, yf - 1), gradient(bb, xf - 1, yf - 1))
    total += lerp(v, x1, x2) * amplitude
    max_value += amplitude
    amplitude *= 0.5
    frequency *= 2
    noise_grid[y][x] = total / max_value
return noise_grid

```

```

def    perlin_worms(self,    octaves=3,    min_threshold=-0.12,
max_threshold=0.12, ca_iterations=10, border_width=4):
    noise_grid = self._generate_perlin_noise(octaves)
    self.grid = np.ones((self.height, self.width), dtype=int)
    for y in range(self.height):
        for x in range(self.width):
            if min_threshold <= noise_grid[y][x] <= max_threshold:
                self.grid[y][x] = 0
    for i in range(border_width):

```

```

self.grid[i, :] = 1
self.grid[self.height - 1 - i, :] = 1
self.grid[:, i] = 1
self.grid[:, self.width - 1 - i] = 1
for _ in range(ca_iterations):
    new_grid = self.grid.copy()
    for y in range(border_width, self.height - border_width):
        for x in range(border_width, self.width - border_width):
            neighbors = self._count_neighbors(x, y)
            if self.grid[y][x] == 1:
                new_grid[y][x] = 1 if neighbors >= 4 else 0
            else:
                new_grid[y][x] = 1 if neighbors >= 5 else 0
    self.grid = new_grid
return self.grid

def bsp_dungeon(self, min_room_size=4, max_room_size=10):
    self.grid = np.ones((self.height, self.width), dtype=int)
    class Room:
        def __init__(self, x, y, w, h):
            self.x, self.y, self.w, self.h = x, y, w, h
    def split(room, depth=0):
        max_depth = 4
        if depth >= max_depth or room.w < min_room_size * 2 + 2 or
room.h < min_room_size * 2 + 2:
            w = min(max_room_size, max(min_room_size, room.w - 4))
            h = min(max_room_size, max(min_room_size, room.h - 4))
            if w >= min_room_size and h >= min_room_size:
                max_x = max(room.x + 2, room.x + room.w - w - 2)
                max_y = max(room.y + 2, room.y + room.h - h - 2)

```

```

        x = random.randint(room.x + 2, max_x) if max_x > room.x + 2
    else room.x + 2

    y = random.randint(room.y + 2, max_y) if max_y > room.y + 2
    else room.y + 2

    for i in range(y, min(y + h, self.height)):
        for j in range(x, min(x + w, self.width)):
            self.grid[i][j] = 0
    return [(x + w//2, y + h//2)]
return []

horizontal = random.choice([True, False]) if abs(room.w - room.h)
< 5 else room.h > room.w
try:
    if horizontal:
        min_split = room.y + min_room_size + 2
        max_split = room.y + room.h - min_room_size - 2
        if min_split >= max_split:
            return split(room, depth + max_depth)
        split_pos = random.randint(min_split, max_split)
        r1 = Room(room.x, room.y, room.w, split_pos - room.y)
        r2 = Room(room.x, split_pos, room.w, room.y + room.h -
split_pos)
    else:
        min_split = room.x + min_room_size + 2
        max_split = room.x + room.w - min_room_size - 2
        if min_split >= max_split:
            return split(room, depth + max_depth)
        split_pos = random.randint(min_split, max_split)
        r1 = Room(room.x, room.y, split_pos - room.x, room.h)
        r2 = Room(split_pos, room.y, room.x + room.w - split_pos,
room.h)

```

```

        centers1 = split(r1, depth + 1)
        centers2 = split(r2, depth + 1)
        if centers1 and centers2:
            c1 = random.choice(centers1)
            c2 = random.choice(centers2)
            self._create_corridor(c1, c2)
        return centers1 + centers2
    except:
        return split(room, depth + max_depth)
root = Room(0, 0, self.width, self.height)
split(root)
return self.grid

def _create_corridor(self, p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    if random.choice([True, False]):
        for x in range(min(x1, x2), max(x1, x2) + 1):
            if 0 <= y1 < self.height and 0 <= x < self.width:
                self.grid[y1][x] = 0
        for y in range(min(y1, y2), max(y1, y2) + 1):
            if 0 <= y < self.height and 0 <= x2 < self.width:
                self.grid[y][x2] = 0
    else:
        for y in range(min(y1, y2), max(y1, y2) + 1):
            if 0 <= y < self.height and 0 <= x1 < self.width:
                self.grid[y][x1] = 0
        for x in range(min(x1, x2), max(x1, x2) + 1):
            if 0 <= y2 < self.height and 0 <= x < self.width:
                self.grid[y2][x] = 0

```

```

def _count_neighbors(self, x, y):
    count = 0
    for dy in [-1, 0, 1]:
        for dx in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.width and 0 <= ny < self.height:
                count += self.grid[ny][nx]
            else:
                count += 1
    return count

```

```

class MetricsCalculator:

```

```

    @staticmethod
    def calculate_leniency(grid):
        h, w = grid.shape
        empty_cells = np.sum(grid == 0)
        if empty_cells < 3:
            return 0.0
        empty_coords = np.argwhere(grid == 0)
        binary = (1 - grid).astype(np.uint8) * 255
        num_labels, labels = cv2.connectedComponents(binary)
        if num_labels <= 1:
            return 0.0
        largest_component = 1
        largest_size = 0
        for i in range(1, num_labels):

```

```

size = np.sum(labels == i)
if size > largest_size:
    largest_size = size
    largest_component = i
connected_coords = np.argwhere(labels == largest_component)
if len(connected_coords) < 3:
    return 0.0
start = tuple(connected_coords[0])
goal = tuple(connected_coords[-1])
def bfs_path(start_pos, goal_pos, blocked=set()):
    queue = deque([(start_pos, [start_pos])])
    visited = {start_pos}
    while queue:
        (y, x), path = queue.popleft()
        if (y, x) == goal_pos:
            return path
        for dy, dx in [(0,1), (1,0), (0,-1), (-1,0)]:
            ny, nx = y + dy, x + dx
            if (0 <= ny < h and 0 <= nx < w and labels[ny][nx] ==
largest_component and (ny, nx) not in visited and (ny, nx) not in blocked):
                visited.add((ny, nx))
                queue.append(((ny, nx), path + [(ny, nx)]))
    return None
dead_ends = 0
checked_cells = 0
proportional_sample = int(len(connected_coords) * 0.2)
sample_size = min(max(100, proportional_sample), min(500,
len(connected_coords)))
sampled_indices = random.sample(range(len(connected_coords)),
sample_size)

```

```

sampled_cells = [tuple(connected_coords[i]) for i in sampled_indices]
for tile in sampled_cells:
    if tile == start or tile == goal:
        continue
    checked_cells += 1
    path_to_tile = bfs_path(start, tile)
    if path_to_tile is None:
        dead_ends += 1
        continue
    blocked_path = set(path_to_tile[:-1])
    path_to_goal = bfs_path(tile, goal, blocked_path)
    if path_to_goal is None:
        dead_ends += 1
    if checked_cells == 0:
        return 1.0
    leniency = 1.0 - (dead_ends / checked_cells)
    return max(0.0, min(1.0, leniency))

```

@staticmethod

```

def calculate_complexity(grid):
    h, w = grid.shape
    binary = ((1 - grid) * 255).astype(np.uint8)
    if np.sum(binary) == 0:
        return 0.0
    contours, hierarchy = cv2.findContours(binary, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
    if not contours or hierarchy is None:
        return 0.0
    parent_contours = []
    child_contours = []

```

```

for i, h_item in enumerate(hierarchy[0]):
    area = cv2.contourArea(contours[i])
    if area < 10:
        continue
    if h_item[3] == -1:
        parent_contours.append(contours[i])
    else:
        child_contours.append(contours[i])
if not parent_contours:
    return 0.0
parent_area = sum(cv2.contourArea(c) for c in parent_contours)
child_area = sum(cv2.contourArea(c) for c in child_contours)
structure_area = parent_area - child_area
total_perimeter = sum(cv2.arcLength(c, True) for c in parent_contours
+ child_contours)
if total_perimeter == 0 or structure_area <= 0:
    return 0.0
max_roughness = math.sqrt(structure_area / math.pi) / 2
if max_roughness == 0:
    return 0.0
actual_roughness = structure_area / total_perimeter
base_complexity = actual_roughness / max_roughness
return max(0.0, min(1.0, base_complexity))

@staticmethod
def calculate_openness(grid):
    h, w = grid.shape
    empty_cells = (grid == 0)
    if not np.any(empty_cells):
        return 0.0

```

```

binary = (1 - grid).astype(np.uint8)
distance_map = cv2.distanceTransform(binary, cv2.DIST_L1, 3)
open_cells = np.sum(distance_map > 1)
total_empty = np.sum(empty_cells)
openness = open_cells / total_empty if total_empty > 0 else 0
return max(0.0, min(1.0, openness))

```

```
@staticmethod
```

```
def calculate_connectivity(grid):
```

```

    binary = (1 - grid).astype(np.uint8) * 255
    if np.sum(binary) == 0:
        return 0.0
    num_labels, labels = cv2.connectedComponents(binary)
    if num_labels <= 2:
        return 1.0
    component_sizes = []
    for i in range(1, num_labels):
        size = np.sum(labels == i)
        component_sizes.append(size)
    if not component_sizes:
        return 0.0
    max_size = max(component_sizes)
    threshold = max_size * 0.10
    significant = sum(1 for size in component_sizes if size > threshold)
    if significant <= 1:
        return 1.0
    connectivity = 1.0 / (1.5 ** (significant - 1))
    return min(1.0, connectivity)

```

```
class MazeApp:
```

```

def __init__(self, root):
    self.root = root
    self.root.title("Система процедурної генерації та оцінки печер")
    self.root.geometry("1200x800")
    self.current_grid = None
    self.metrics = {}
    self.setup_ui()

def setup_ui(self):
    control_frame = ttk.Frame(self.root, padding="10")
    control_frame.pack(side=tk.TOP, fill=tk.X)
    ttk.Label(control_frame, text="Ширина:").grid(row=0, column=0,
padx=5)

    self.width_var = tk.IntVar(value=50)
    ttk.Spinbox(control_frame, from_=20, to=100,
textvariable=self.width_var, width=10).grid(row=0, column=1, padx=5)
    ttk.Label(control_frame, text="Висота:").grid(row=0, column=2,
padx=5)

    self.height_var = tk.IntVar(value=50)
    ttk.Spinbox(control_frame, from_=20, to=100,
textvariable=self.height_var, width=10).grid(row=0, column=3, padx=5)
    ttk.Label(control_frame, text="Алгоритм:").grid(row=0, column=4,
padx=5)

    self.algorithm_var = tk.StringVar(value="Рекурсивний пошук")
    algorithms = ["Рекурсивний пошук", "Рекурсивний поділ",
"Випадкова прогулянка", "Клітинні автомати", "Perlin Worms", "BSP
підземелля"]
    ttk.Combobox(control_frame, textvariable=self.algorithm_var,
values=algorithms, state="readonly", width=20).grid(row=0, column=5, padx=5)

```

```

        ttk.Button(control_frame, text="Генерувати",
command=self.generate_maze).grid(row=0, column=6, padx=10)

        main_container = ttk.Frame(self.root)
        main_container.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
        left_frame = ttk.LabelFrame(main_container, text="Згенерований
лабіринт", padding="10")
        left_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        self.canvas = tk.Canvas(left_frame, bg="white", width=600,
height=600)
        self.canvas.pack(fill=tk.BOTH, expand=True)
        right_frame = ttk.LabelFrame(main_container, text="Метрики
оцінювання", padding="10")
        right_frame.pack(side=tk.RIGHT, fill=tk.BOTH, padx=(10, 0))
        metrics_labels = [("Поблажливість:", "leniency"), ("Складність:",
"complexity"), ("Відкритість:", "openness"), ("Зв'язність:", "connectivity")]
        self.metric_labels = {}
        for i, (label, key) in enumerate(metrics_labels):
            ttk.Label(right_frame, text=label, font=("Arial", 11,
"bold")).grid(row=i, column=0, sticky="w", pady=(10, 2))
            value_label = ttk.Label(right_frame, text="—", font=("Arial", 14))
            value_label.grid(row=i, column=1, sticky="w", padx=(10, 0),
pady=(10, 2))
            self.metric_labels[key] = value_label

    def generate_maze(self):
        try:
            width = self.width_var.get()
            height = self.height_var.get()
            algorithm = self.algorithm_var.get()
            generator = MazeGenerator(width, height)

```

```

if algorithm == "Рекурсивний пошук":
    self.current_grid = generator.recursive_backtracking()
elif algorithm == "Рекурсивний поділ":
    self.current_grid = generator.recursive_division()
elif algorithm == "Випадкова прогулянка":
    self.current_grid = generator.random_walk()
elif algorithm == "Клітинні автомати":
    self.current_grid = generator.cellular_automata()
elif algorithm == "Perlin Worms":
    self.current_grid = generator.perlin_worms()
elif algorithm == "BSP підземелля":
    self.current_grid = generator.bsp_dungeon()
self.display_maze()
self.calculate_metrics()
except Exception as e:
    messagebox.showerror("Помилка", f"Виникла помилка при
генерації: {str(e)}")

def display_maze(self):
    if self.current_grid is None:
        return
    self.canvas.delete("all")
    canvas_width = self.canvas.winfo_width() if
self.canvas.winfo_width() > 1 else 600
    canvas_height = self.canvas.winfo_height() if
self.canvas.winfo_height() > 1 else 600
    cell_width = canvas_width / self.current_grid.shape[1]
    cell_height = canvas_height / self.current_grid.shape[0]
    cell_size = min(cell_width, cell_height)
    for y in range(self.current_grid.shape[0]):

```

```

        for x in range(self.current_grid.shape[1]):
            x1, y1 = x * cell_size, y * cell_size
            x2, y2 = x1 + cell_size, y1 + cell_size
            color = "#2C3E50" if self.current_grid[y][x] == 1 else
"#ECF0F1"

            self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="")

def calculate_metrics(self):
    if self.current_grid is None:
        return

    calculator = MetricsCalculator()

    self.metrics['leniency'] =
calculator.calculate_leniency(self.current_grid)

    self.metrics['complexity'] =
calculator.calculate_complexity(self.current_grid)

    self.metrics['openness'] =
calculator.calculate_openness(self.current_grid)

    self.metrics['connectivity'] =
calculator.calculate_connectivity(self.current_grid)

    for key, value in self.metrics.items():
        self.metric_labels[key].config(text=f"{value:.3f}")

def main():
    root = tk.Tk()
    app = MazeApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```

ДОДАТОК Б
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів
Дипломна кваліфікаційна робота магістра

Дослідження та програмна реалізація системи процедурної генерації рівнів у 2D-грі

Виконав:

ст. групи КНТ-114м

Костянтин ВЛАСЕНКО

Керівники:

к.т.н, професор

Валерій ДУБРОВІН

Рисунок Б.1 – Слайд 1

Мета, об'єкт, предмет і задачі дослідження

- Мета – розроблення програмного забезпечення для автоматизації процесу створення ігрових рівнів шляхом реалізації системи процедурної генерації з інтегрованими метриками якості та порівняльним аналізом алгоритмів.
- Об'єкт – процес розроблення програмного забезпечення для процедурної генерації та оцінювання ігрових рівнів..
- Предмет – методи проєктування, реалізації та тестування програмних систем процедурної генерації з формалізованими метриками якості для алгоритмів створення лабіринтів і печерних структур.
- Завдання:
 - аналіз алгоритмів;
 - реалізація метрик оцінювання;
 - визначити оптимальне застосування алгоритмів.

2

Рисунок Б.2 – Слайд 2

Аналіз предметної області

Критерій	Традиційне проектування	Процедурна генерація
Трудовитрати	Високі	Низькі
Варіативність контенту	Обмежена	Висока
Масштабованість	Низька	Висока
Оцінювання якості	Суб'єктивне	Формалізоване
Залежність від людини	Висока	Знижена

3

Рисунок Б.3 – Слайд 3

Таблична порівняльна характеристика мов програмування

Критерій	Python	C++	JavaScript
Швидкість розробки	Висока	Низька	Середня
Продуктивність	Середня	Висока	Низька
Бібліотеки для наукових обчислень	NumPy	Eigen	math.js, numeric.js
Бібліотеки для комп'ютерного зору	OpenCV	OpenCV	Обмежені
Створення GUI	Tkinter	wxWidgets	Electron
Підтримка кросплатформеності	Висока	Середня	Висока

4

Рисунок Б.4 – Слайд 4

Таблична порівняльна характеристика середовищ програмування

Критерій	VS Code	PyCharm	Jupyter
Швидкість виконання коду	Висока	Середня	Висока
Споживання пам'яті	Низьке	Високе	Середнє
Можливості налагодження	Добре	Відмінно	Обмежене
Інтерактивність	Низька	Низька	Висока
Функціонал для розробки застосунків	Добре	Відмінно	Погано
Вартість	Безкоштовно	Community — безкоштовно, Professional — платно	Безкоштовно

5

Рисунок Б.5 – Слайд 5

Схема структури системи



6

Рисунок Б.6 – Слайд 6

Основні класи системи та їх відповідальність

Клас	Відповідальність	Ключові методи
MazeGenerator	Генерація лабіринтів різними алгоритмами	recursive_backtracking, cellular_automata, perlin_worms
MetricsCalculator	Обчислення метрик якості	calculate_leniency, calculate_complexity, calculate_connectivity
MazeApp	Графічний інтерфейс та координація	setup_ui, generate_maze, display_maze, calculate_metrics

7

Рисунок Б.7 – Слайд 7

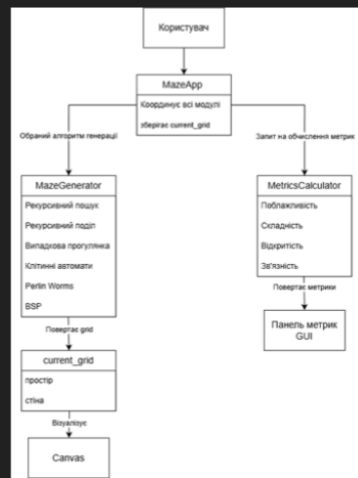
Допоміжні методи класу MazeGenerator

Метод	Призначення	Параметри	Повертає
_count_neighbors	Підрахунок сусідніх стін	x, y: координати клітинки	int: кількість стін
_connect_caves	З'єднання роз'єднаних печер	–	–
_create_corridor	Створення коридору між точками	p1, p2: координати точок	–
_generate_perlin_noise	Генерація шуму Перліна	octaves: кількість октав	ndarray: сітка шуму

8

Рисунок Б.8 – Слайд 8

Схема функціонування системи



9

Рисунок Б.9 – Слайд 9

Методи обчислення метрик якості

Метрика	Метод	Основний алгоритм
Поблажливість	calculate_leniency	Пошук в ширину
Складність	calculate_complexity	Аналіз контурів
Відкритість	calculate_openness	Трансформація відстані
Зв'язність	calculate_connectivity	Зв'язні компоненти

10

Рисунок Б.10 – Слайд 10

Висновки

У дипломній роботі створено систему процедурної генерації та оцінювання печерних структур, що поєднує шість алгоритмів і чотири метрики аналізу. Модульна архітектура розділяє генерацію, обчислення характеристик та інтерфейс, забезпечуючи гнучкість і можливість розширення.

Експерименти показали, що рекурсивні методи формують найзакритіші лабіринти, тоді як випадкова прогулянка та Perlin Worms дають найбільш відкриті й органічні карти, а клітинні автомати та BSP — проміжні збалансовані структури. Система підтвердила ефективність як інструмент аналізу процедурної генерації та може бути основою для подальших проектів у геймдизайні.

11

Рисунок Б.11 – Слайд 11