ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально–науковий інститут Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи
на ступень вищої освіти магістр

на тему «**РОЗРОБКА МЕТОДУ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ДЛЯ ГРИ В ЖАНРІ ROGUELIKE НА ОСНОВІ КЛІТИННОГО АВТОМАТУ»**

Виконав: студент 6 курсу, групи ПДМ - 61
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Сачик Микита Олександрович
(прізвище та ініціали)
Керівник Золотухіна О.А.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

КИЇВ – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально–науковий інститут Інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти «Магістр»
Спеціальність підготовки 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного
забезпечення
_____О.В.Негоденко
“____” _____ 2022 року

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Сачик Микита Олександрович
(прізвище, ім’я, по батькові)

1.Тема роботи: «Розробка методу автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату»

Керівник роботи Золотухіна Оксана Анатоліївна, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від « 12 » жовтня 2022 року № 122

2. Строк подання студентом роботи «31» грудня 2022 року

3. Вихідні дані до роботи: Матеріали переддипломної практики, методи автоматизованої генерації, науково-технічна література з питань, пов’язаних з програмним забезпеченням, клітинними автоматами, мовою програмування C#.

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

4.1 Огляд предметної області
4.2 Розробка моделей та методів автоматизованої генерації ігрової карти
4.3 Розробка програмного забезпечення моделей
4.4 Проведення моделювання та аналіз отриманих результатів

5. Перелік графічного матеріалу (презентація)

5.1 Мета, об’єкт та предмет дослідження
5.2 Ключові елементи ігрової карти в іграх жанру roguelike
5.3 Аналіз існуючих методів автоматизованої генерації ігрової карти
5.4 Критерії якості ігрової карті жанру roguelike
5.5 Модель карти на основі клітинного автомату

5.6 Метод автоматизованої генерації ігрової карти
5.7 Схема функціонування додатку для автоматизованої генерації ігрової карти
5.8 Параметри для створення ігрової карти
5.9 Екранна форма додатку для автоматизованої генерації ігрових карт
5.10 Результат генерації ігрової карти типу «суцільна печера»
5.11 Результат генерації ігрової карти типу «багато маленьких крапель-печер»
5.12 Результат генерації ігрової карти типу «лабіринт»
5.13 Аналіз характеристик створених карт
5.14 Висновки
5.15 Апробація роботи

6. Дата видачі завдання «14» жовтня 2022 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
2	Огляд предметної області	14.10-19.10	Виконано
3	Аналіз методів автоматизованої генерації ігрових карт	19.10-25.10	Виконано
4	Розробка моделі клітинного автомату та методу генерації карт для використання в іграх жанру roguelike	25.10-8.11	Виконано
5	Розробка програмного забезпечення для автоматизованої генерації ігрових карт	8.11-20.11	Виконано
6	Моделювання та аналіз результатів	20.11-26.11	Виконано
7	Написання та оформлення пояснювальної записки	26.11-20.12	Виконано
8	Розробка графічних та презентаційних матеріалів	19.12-20.12	Виконано
9	Попередній захист роботи	20.12-22.12	Виконано
10	Захист роботи	17.01	Виконано

Студент _____ Сачик М.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ О.А. Золотухіна
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи: 78с., 44 рис., 1 дод., 16 джерел.

МОДЕЛЬ, КЛІТИННИЙ АВТОМАТ, АВТОМАТИЗОВАНА ГЕНЕРАЦІЯ, ROGUELIKE, ІГРОВА КРТА, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, C#

Об`єкт дослідження – автоматизована генерація ігрової карти для гри в жанрі Roguelike.

Предмет дослідження – метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату.

Мета роботи – покращення процесу автоматизованого створення ігрової карти для гри в жанрі Roguelike за рахунок застосування клітинного автомату.

Методи дослідження – математичні: теорія ймовірності, статистичні, методи автоматизованої генерації ігрових карт, клітинні автомати; емпірико-теоретичні: абстрагування, аналіз, синтез, евристики побудови печер та тунелів, моделювання; методи проектування та розробки програмного забезпечення.

У роботі розглянуто типові підходи для реалізації автоматизованої генерації ігрової карти для ігор жанру Roguelike та виявлені ключові елементи ігрової карти, що притаманні іграм цього жанру. Розроблена модель карти на основі клітинного автомату. Розглянуто алгоритми, що розширюють можливості клітинного автомату. Розроблено метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату.

Виконано огляд використаних програмних засобів та середовищ розробки. На основі вдосконалених алгоритмів заливки та тунелювання розроблено програмне забезпечення, що реалізує автоматизовану генерацію ігрової карти для ігор жанру Roguelike. За результатами моделювання отримано проведено аналіз характеристик створених карт та визначено прийнятні параметри для генерації карт різних типів.

ЗМІСТ

ВСТУП

Ігрова індустрія на сьогоднішній день захопила багато сфер діяльності, та привертає до себе увагу мільйонів людей щоденно. У всьому світі проходять змагання з різних ігор, призові яких вже перевищують багато спортивних дисциплін, з кожним днем кіберспорт розвивається, усе більше людей грають у свій вільний час, під час подорожей, деякі витрачають багато років аби поставити новій рекорд у певній грі, для когось це головний спосіб заробітку, а хтось вирішив поєднати своє життя з розробкою ігор. Грати можна на комп'ютерах, в найприголомшливіші ігри з графікою, яку ледь можна відрізнити від реальності, на консолях, які купляють спеціально щоб пограти в ексклюзивні ігри, на ігрових автоматах, які у свій час покорили серця мільйонів підлітків по усьому світу, мобільних та планшетах, які дають змогу грати у любому місці, й у що завгодно. З впевненістю можна сказати, що ігрова індустрія є дуже актуальною сферою на сьогоднішній день, і з часом тільки поширює свій вплив.

Важливими показниками ефективності будь-якої гри виступають сюжет, графіка, можливості, що пропонує гра, якість фізики та ігровий світ, що представляється в багатьох жанрах за допомогою ігрових карт. В залежності від характеристик карти, гра може приваблювати гравця та надихати на її повне дослідження, або навпаки, відбити усе бажання грати у гру, ось наскільки важливо добре продумати карту для гри. В залежності від жанру, цінність можуть становити як відкриті ігрові карти з високим рівнем деталізації, гарною графікою та величезним розміром, так піксельні карти невеликих розмірів, які при якісь події, наприклад смерті гравця, будуть повністю змінюватися. У кожного типу карт є свої шанувальники, які полюбляють певний дизайн.

Одним з актуальних питань при розробці елементів ігрового світу є його автоматизована генерація. Процедурна генерація спряє автоматизованому створенню різноманітних ігрових елементів за допомогою алгоритмів прописаних розробником. Тобто, у випадку зі створенням карт при автоматизованій генерації розробнику не потрібно вимальовувати та створювати кожний елемент карти

окремо, а потім прописувати суцільну карту, він може один раз описати загальні правила, за якими ігрова карта буде створюватися. Це є дуже корисним інструментом, оскільки замість однієї карти, яка швидко може набриднути гравцям, створюється можливість отримати нескінченну кількість карт, які увесь час будуть генерувати різні патерни, і якість яких буде залежати тільки від прописаних розробником алгоритмів та правил.

Процедурне створення карти є доцільним для використання в іграх, де випадковість ігрової карти позитивно вплине на геймплей, і найкращим жанром для її реалізації є Roguelike. Він розкриває усі її позитивні сторони, та саме при створенні гри у цьому жанру вперше була застосована процедурна генерація. Автоматизована генерація ігрової карти вирішує проблему з реіграбельністю, оскільки гравці кожен раз будуть отримувати щось нове при проходженні гри та бажати дослідити ще більш різноманітні варіації карти. Завдяки її алгоритмам, економиться значна частина ресурсів команди-розробників на розробку гри, оскільки більшість контенту генерується автоматизовано, а вивільнені ресурси можна витратити на дослідження та впровадження методів та алгоритмів цієї автоматизованої генерації. Таким чином, актуальним є дослідження, покращення та розробка ефективних алгоритмів, правил та методів за допомогою яких реалізується процедурна генерація ігрових карт.

В роботі розглянуто існуючі алгоритми та методи для автоматизованої генерації ігрової карти, в тому числі, на основі клітинного автомату. Створено відповідну модель та метод для покращення автоматизованої генерації ігрової карти на основі клітинного автомату для гри в жанрі Roguelike та проведено моделювання процесу створення карт з різними показниками.

Об'єкт дослідження – автоматизована генерація ігрової карти для гри в жанрі Roguelike.

Предмет дослідження – метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату.

Мета роботи – покращення процесу автоматизованого створення ігрової карти для гри в жанрі Roguelike за рахунок застосування клітинного автомату.

Методи дослідження – математичні: теорія ймовірності, статистичні, методи автоматизованої генерації ігрових карт, клітинні автомати; емпірико-теоретичні: абстрагування, аналіз, синтез, евристики побудови печер та тунелів, моделювання; методи проектування та розробки програмного забезпечення.

Практична значущість результатів полягає в використанні розробленої моделі для отримання статистичних даних у автоматизованій генерації ігрової карти, а саме кількість створених печер, розміри ігрової карти, кількість тунелів, довжина тунелів, їх прямолінійність, кількість входів та виходів у печері, відсутність печер, не поєднаних з іншими, можливість згладжування печер.

Для досягнення мети вирішено наступні завдання:

1. Аналіз ключових елементів ігрової карти в іграх жанру roguelike.
2. Аналіз існуючих методів автоматизованої генерації ігрової карти.
3. Визначення критеріїв якості ігрової карті жанру roguelike.
4. Розробка моделі карти на основі клітинного автомату.
5. Розробка методу автоматизованої генерації ігрової карти на основі клітинного автомату.
6. Визначення параметрів для створення ігрової карти.
7. Розробка додатку для автоматизованої генерації ігрових карт.
8. Аналіз характеристик створених карт.

1 АНАЛІЗ МОДЕЛЕЙ, МЕТОДІВ ТА ЗАСОБІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ В ІГРАХ

1.1 Актуальність процедурної генерації в іграх

При розробці ігор значну увагу приділяють створенню рівнів, карт, ландшафту, бо вони можуть зробити гру значно цікавіше, або ж навпаки, погано зроблена карта може перетворити гру в неграбельну. Автоматизована генерація може створити безліч цікавих варіацій карт, ігрових предметів, що може повністю змінити сприйняття гри, та притягувати увагу гравців кожного разу, що одразу вирішує проблему деяких гравців, котрим набридає проходження однієї карти під час кожної гри.

Оскільки замість створення кожного рівня, карти, або світу вони будуть генеруватися автоматично, це збереже багато часу, грошей та зусиль при розробці. Головний аспект при розробці гри, у якій необхідно застосовувати автоматизовану генерацію – це приділити увагу написанню якісних алгоритмів: вони повинні працювати швидко, ефективно, та не містити помилок в генерованих картах, які потім можуть привести до проблем в ігровому процесі.

Автоматизована генерація має безліч плюсів, хоча використовується далеко не у кожні грі. По-перше, вона відкриває можливості створення такого великого рівня, або карти, яку б дизайнери не створили б навіть за декілька років, це не кажучи про те, що вона буде унікальною. Ще значною мірою заощаджується пам'ять, бо кожен об’єкт карти зберігається не окремо, а зберігається вся карта. При автоматизованій генерації, гравець ніколи не знає що його очікує, і кожен раз отримує нові емоції [1]. Кількість розробників, яка працює над створенням гри, набагато менша ніж при інших підходах. Таким чином, автоматизована генерація є важливим й затребуваним інструментом у розробці багатьох ігор. Найпоширеніше використання автоматизована генерація карт набула для ігор в жанрі Roguelike [2], [3].

1.2 Особливості розробки ігор жанру Roguelike

Визначення терміну Roguelike полягає в тому, що він характеризується іграми, які були прямо чи опосередковано натхненні грою «Rogue», скріншот якої можна побачити на рисунку 1.1. Це піджанр рольових ігор, які мають певні риси жанру, наприклад остаточна смерть гравця, випадкові, або процедурно згенеровані підземелля, покроковий рух, випадкові властивості предметів. Це гра, яка пропонує досліджувати рівні, щоб усунути якомога більше ворогів і знайти якомога більше скарбів у всесвіті, населеному монстрами та магією. Гравець ніколи не сприймає весь рівень, який залишається обмеженим полем зору гравця. Хід гри дуже відрізняється від інших відеоігор [4], оскільки тут потрібно чекати своєї черги, щоб виконати дію.

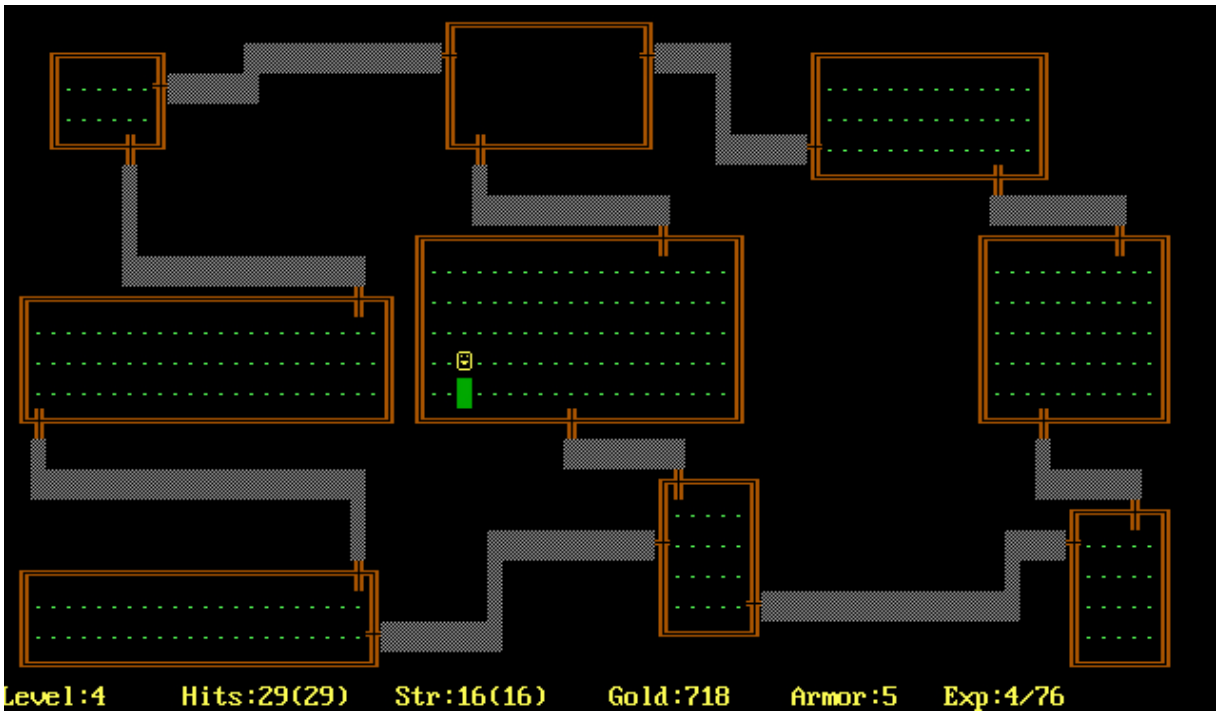


Рисунок 1.1 – Скріншот гри «Rogue»

Іграм жанру Roguelike притаманні такі характеристики як:

– генерація випадкового середовища: світ генерується процедурно, і швидше за все гравець ніколи не побачить один і той же рівень підземелля двічі;

– миттєва смерть: коли персонаж гравця помирає, ним більше не можна буде грати;

– послідовність: гра відбувається не в реальному часі, тому усе змінюється, лише коли користувач діє якимось чином;

– карта на основі сітки: світ представлено рівномірною сіткою плиток;

– складність: гра достатньо складна, тому є кілька рішень для проходження. це досягається за допомогою взаємодій з предметами та монстрами;

– управління ресурсами: гравець має обмежені ресурси, і повинен придумати стратегії для найкращої взаємодії з ними для просування в грі;

– бий та руби: боротьба з великою кількістю монстрів є важливою частиною гри;

– дослідження: гравець має досліджувати нові рівні підземелля для кожної нової гри, та повинен це робити обережно, усе продумувати заздалегідь;

Як і в рольових іграх, в Roguelike гравці можуть у більшості випадків вибирати свій клас і стиль гри. Від мага до лучника, або воїна та інших, можна вибрати, як підходити до небезпек, з якими стикається ігровий персонаж. Підземелля процедурно генеруються з кожною новою грою, а численні кімнати та монстри розташовуються у випадковому порядку, щоб кожна пригода була унікальною [5], [6].

Іншою особливістю є те, що кожна смерть є перманентною. Гравці повинні перезапустити всю гру. Збереження дозволяють лише призупинити гру, а всі наявні збереження стираються, коли персонаж помирає. Ця функція була створена, щоб зробити гру більш захоплюючою, щоб змусити гравця думати про всі свої рухи; кожна помилка може бути фатальною. Щоб гравець не розчаровувався через нестачу швидкості в своїх діях, усі ігри цього жанру є покроковими, і кожен вибір і зіткнення можна вирішити кількома способами. Наприклад, замкнені двері можна відчинити за допомогою ключа, вдарити ногою або навіть обійти, пройшовши іншим шляхом [7]. Тому покрокова гра пропонує можливість подумати, щоб прийняти найкраще рішення.

Надзвичайна складність цього жанру приваблює гравців, які люблять виклик і хочуть перевершити свої досягнення у грі. Складність, як правило, прогресує, і вороги, які спочатку є складними, стають простими після кількох зіткнень. Таким чином, навіть якщо гра є вимогливою, гравець бачить, як прогресує та просувається в грі. Цю перманентну смерть можна порівняти з аркадними іграми 80-х років, де збереження було неможливим і кожна нова гра починалася знову з першого рівня [8].

Крім того, випадкова частина здобичі та процедурна генерація рівнів сприяють на бажання гравців кожен раз повертатися до гри, чим вирішують проблеми з реіграбельністю. Кожна гра унікальна, навіть якщо використовується один і той самий сценарій з розташуванням кімнат чи потребою знову збирати арсенал. Таким чином, гравець повинен постійно адаптуватися, показувати свої рефлекси та навички, набуті в ході пригод. Навіть після перемоги над фінальним босом іноді можна перезапустити гру, щоб перевірити нові комбінації предметів або закінчити її за всіх персонажів.

Більшість Roguelike ігор мають дуже стислий вступ, і більша частина історії та сюжету прихована в грі. Під час перших кількох запусків персонаж вирушає в пригоду, не знаючи кінцевої мети, але вся історія відкривається в залежності від кроків проходження, й гравець може в деяких випадках пропустити багато цікавих елементів. Часто буває, що гра перезапускається лише тому, що гравець заглиблений в атмосферу гри і хоче знати більше про ігровий простір.

1.3 Процедурна генерація елементів гри в жанрі Roguelike

Одна з ключових властивостей жанру Roguelike є процедурно згенеровані елементи, бо саме вони викликають інтерес та вражають гравців своїм розмаїттям. За допомогою гарної реалізації цього елементу можна значно покращити геймплей, заохочуючи гравця більш ретельно дослідити печери, створити стратегію проходження карти, ворогів які будуть з’являтись у неочікуваних

місцях, та нарешті зміг отримати винагороду за старання у вигляді цінних предметів.

Процедурна генерація – це створення цифрового вмісту, такого як рівні відеоігор, зображення, музика, персонажі, локації, і так далі, автоматизованим способом із відповіддю на набір правил, визначених алгоритмами [9]. Процедурна генерація використовує програму, яка самостійно створює певний вміст карти, спираючись на базу даних, надану її розробником. Але це не обов'язково випадково, оскільки карта має відповідати правилам, розробленим розробником.

Великий плюс такого підходу є те, що він легкий. Такий дизайн справді економить багато місця на диску, бо він сам генерує вміст із пулу попередньо визначених активів, а не завантажує для кожного рівня повне створення.

Процедурна генерація містить в собі елемент випадковості. Алгоритм враховуватиме набір правил і випадковим чином генеруватиме для гри вміст. Навпаки, створюючи контент вручну, існує обмеження щодо того, чого можна досягти за обумовлений час.

Процедурна генерація також використовується з обробкою природної мови для створення реалістичних сюжетів в іграх [10]. Зазвичай реалізується в текстових іграх.

Генерація підземель є важливим прикладом процедурної генерації. У підземеллі кілька кімнат розміщені на сітці, а потім намагаються з'єднатися одна з одною через коридори.

Схеми коридорів, що з'єднують кімнати, генеруються процедурно, і з такою кількістю комбінацій і можливостей кінцевий дизайн може стати справді складним. Чим вище складність, тим більше задоволення гравців.

Вимоги до автоматизованої генерації передбачають наступне:

– заощадження часу: генерація елементів значно швидша ніж прописування кожного елементу уручну;

– різноманітність: навіть коли елементи будуть генеруватися випадково, гравцю може швидко набриднути ігрове середовище, яке побудовано за

незмінним алгоритмом; випадкова генерація повинна спиратись на алгоритми, що дозволяють легко змінювати параметри генерації.

Часто процедурну генерацію використовують для створення карт [11]. Приклади карт створених завдяки процедурній генерації можна побачити на рисунку 1.2. Але при цьому не існує єдиного вірного методу на всі випадки, їх підбирають залежно від гри яку розробляють, й за потреби комбінують.

Рисунок 1.2 – Приклади карт створених завдяки процедурній генерації

1.4 Огляд методів та алгоритмів для автоматизованої процедурної генерації ігрової карти

1.4.1 Просте кімнатне розміщення

Просте кімнатне розміщення це один із найпростіших і, мабуть, найбільш використовуваних алгоритмів для створення підземелля. Результат генерації ігрової карти завдяки його алгоритмам можна побачити на рисунку 1.3. Цей метод розміщує кімнати випадковим чином у сітці, потім обводить кожну кімнату та

намагається їх з’єднати. Якщо буде така можливість, можна також вирити випадкові тунелі, щоб надати підземеллю більш природного відчуття. Його можна реалізувати за допомогою наступних кроків.

1. Визначається кількість кімнат, які потрібно створити на основі вибраного макета кімнати.

2. Використовуємо метод для створення кімнати випадкової ширини та висоти у випадковому місці на сітці. На стінах кімнати розміщено випадкову кількість отворів.

3. Перевіряємо, чи кімната не знаходиться за межами сітки, чи заблокована іншою кімнатою чи коридором . Якщо заблоковано, видаляємо кімнату та повторюємо крок 2.

4. Після ініціалізації всіх кімнат потрібно викликати метод, що буде здатний з’єднати отвори в кожній кімнаті. Метод Манхеттена використовується для пошуку оптимального шляху від одних дверей до інших. Обважнювачі використовуються для того, щоб коридори були більш прямими та з’єднувалися один з одним.

5. Якщо є якісь отвори, які залишилися незв’язаними, випадковим чином тунелюються коридори з отвору. Якщо коридор досягає стіни кімнати, туди додаються двері.

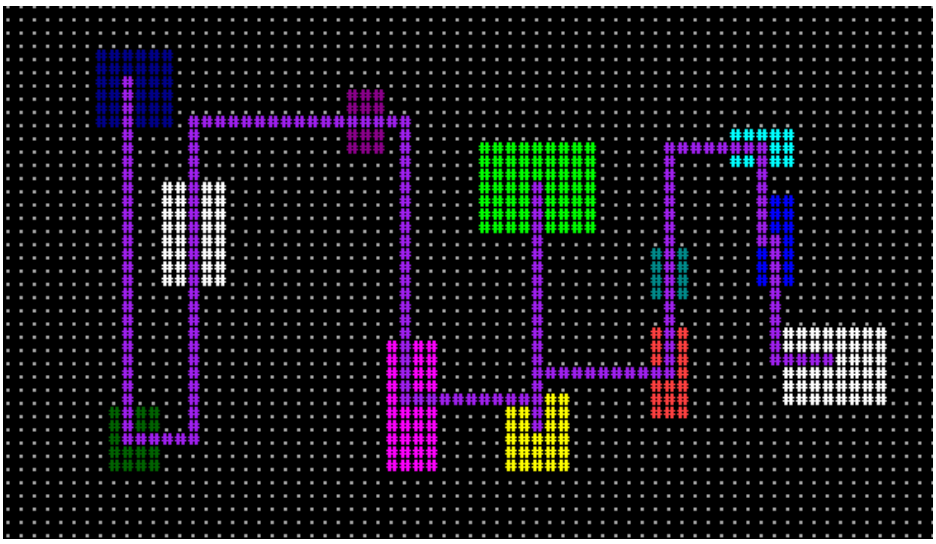


Рисунок 1.3 – Зображення результату генерації завдяки алгоритмам простого кімнатного розміщення

Різні типи підземель можна створити на основі мінімального та максимального розміру кімнати та кількості кімнат. Цей алгоритм використовує метод Манхеттена для обчислення своєї евристики.

Манхеттенська відстань — це метрика відстані між двома точками в N-вимірному векторному просторі. Це сума довжин проекцій відрізка між точками на осі координат. Простіше кажучи, це сума абсолютної різниці між мірами в усіх вимірах двох точок.

Цей метод, який ще називають відстанню таксі, більш корисний для векторів, які описують об’єкти на єдиній сітці, як-от шахова дошка чи міські квартали. Назва таксі для міри зумовлена тим, що міра обчислює найкоротший шлях, який проїде таксі між кварталами міста (координати на сітці).

Манхеттенська відстань обчислюється як сума абсолютних різниць між двома векторами:

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|, \quad (1.1)$$

де d1 – Мангеттенська відстань,

p – вектор, p = (p1, p2, …, pn);

q – вектор, (q1, q2, …, qn);

n – кількість вимірів у дійсному просторі.

1.4.2 Бінарний розподіл простору

Бінарний розподіл простору часто використовується для 3D-відеоігор, зокрема у шутерах від першої особи. Результат генерації завдяки алгоритмам бінарного розподілу простору можна побачити на рисунку 1.4. Алгоритм бінарного розподілу простору був вперше застосований фахівцями компанії LucasArts на початку 80-х років. Популярність у розробників він здобув завдяки компанії id Software, яка розробила ігри Doom і Quake, скріншот з гри Doom де він був вперше застосований можна побачити на рисунку 1.5. Бінарний розподіл простору виник, бо використовуючи комп’ютерну графіку, потрібно було

якнайшвидше створювати тривимірні сцени, які створюються з багатокутників. Простий спосіб намалювати такі сцени — це алгоритм художника. Цей підхід має два недоліки:

1. Час, необхідний для сортування багатокутників у зворотному порядку.

2. Можливість помилок у багатокутниках, що накладаються, також була високою.

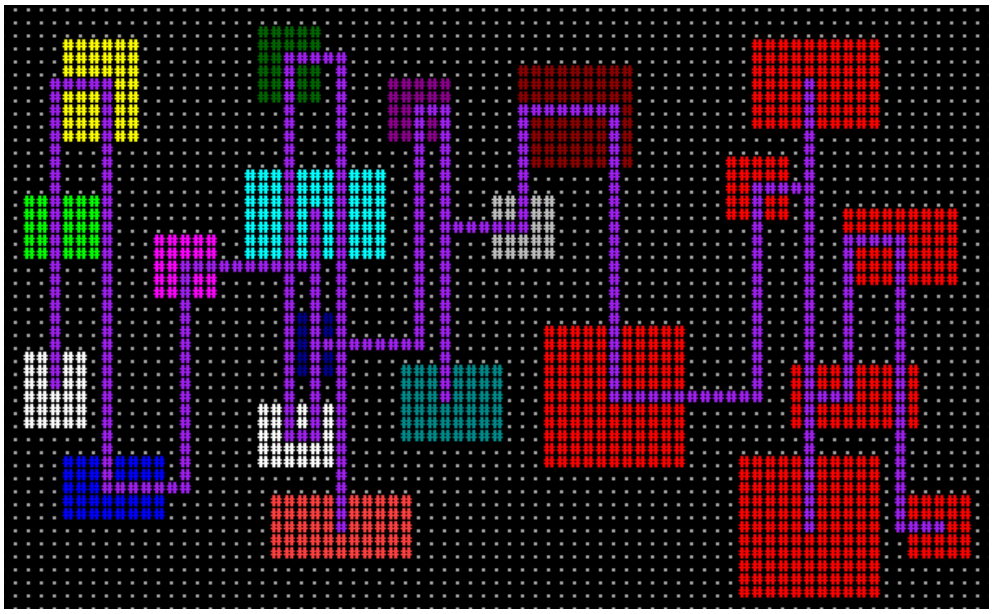


Рисунок 1.4 – Зображення результату генерації завдяки алгоритмам бінарного розподілу простору

Концепція бінарного дерева розподілу простору дозволяє компенсувати ці недоліки. Бінарний алгоритм розподілу простору рекурсивно ділить поверхню на дві частини протягом кількох ітерацій. Бінарне дерево забезпечує найбільш очевидну структуру даних для проходження різних рівнів розділів. Використання Binary Space Partition Tree (BSP):

– для виявлення зіткнень у 3D-відеоіграх і робототехніці;
– в задачах трасування променів;
– в обробці складних просторових сцен.

Для розділення в кожній ітерації напрямок поділу (горизонтальний або вертикальний), призначається випадковим чином, а також розмір першого

розділу, тоді як другий просто займає простір, що залишився. Пізніше, на листках дерева (найнижні вузли), кімнати можна вирощувати всередині контейнера, будь то випадковим чином або шляхом призначення деяких обмежень, які підходять для генерації. Нарешті, з'єднавши центр кожної з перегородок із сусіднім елементом, усі перегородки стають з'єднаними та доступними. Якщо необхідно створити випадкову карту, це можна зробити різними способами. Можна написати просту логіку для створення прямокутників довільного розміру у випадкових місцях, але це може призвести до появи карт, повних перекриваючими, згрупованими або дивно розташованими кімнатами. Це також ускладнює з'єднання кімнат одна з одною та гарантує, що немає ізольованих кімнат без доступу.

Рисунок 1.5 – Скріншот гри DOOM

З BSP можна гарантувати більш рівномірне розміщення кімнат, а також гарантувати, що можна з'єднати всі кімнати разом.

Загальний алгоритм генерації дерева BSP виглядає наступним чином.

1. Виберіть багатокутник P зі списку.

2. Зробіть вузол N у дереві BSP і додайте P до списку багатокутників у цьому вузлі.

3. Для кожного багатокутника в списку:

– якщо цей багатокутник повністю знаходиться перед площиною, що містить P, перемістіть цей багатокутник до списку вузлів перед P;

– якщо цей багатокутник повністю знаходиться позаду площини, що містить P, перемістіть цей багатокутник до списку вузлів позаду P;

– якщо цей багатокутник перетинається площиною, що містить P, розділіть його на два багатокутники та перемістіть їх у відповідні списки багатокутників позаду та перед P;

– якщо цей багатокутник лежить у площині, що містить P, додайте його до списку багатокутників у вузлі N.

4. Застосуйте цей алгоритм до списку багатокутників перед P.

5. Застосуйте цей алгоритм до списку багатокутників позаду P.

Таким чином, розподіл двійкового простору – це метод рекурсивного поділу простору на дві опуклі множини за допомогою гіперплощин як розділів. Отримана структура даних є двійковим деревом , а дві підплощини називають передньою та задньою. Лінія поділу кореня розділяє геометрію на два набори. Весь простір посилається на кореневий вузол, це розбивається шляхом вибору гіперплощини розділу, а дві підплощини, які називають передньою та задньою , містять більше вузлів, і, отже, їх слід розділити, щоб отримати більше підплощин.

Цей процес має рекурсивно повторюватися у кожному підпросторі, створеному для остаточного відтворення повного бінарного дерева, де кожен листовий вузол містить окремі кола, й там ми досягаємо нашого розділеного дерева бінарного простору.

Недоліки бінарного розподілу простору:

– BSP не вирішує проблему визначення видимої поверхні;

– Час, витрачений на створення дерева BSP може бути завеликий.

1.4.3 Прогулянка п'яниці

Прогулянка п'яниці – це тип випадкової прогулянки, який є одним із найпростіших алгоритмів створення підземель. Свою назву він отримав через приголомшливі візерунки, які створює, результат генерації отриманої завдяки використанню цього алгоритму можна побачити на рисунку 1.6.

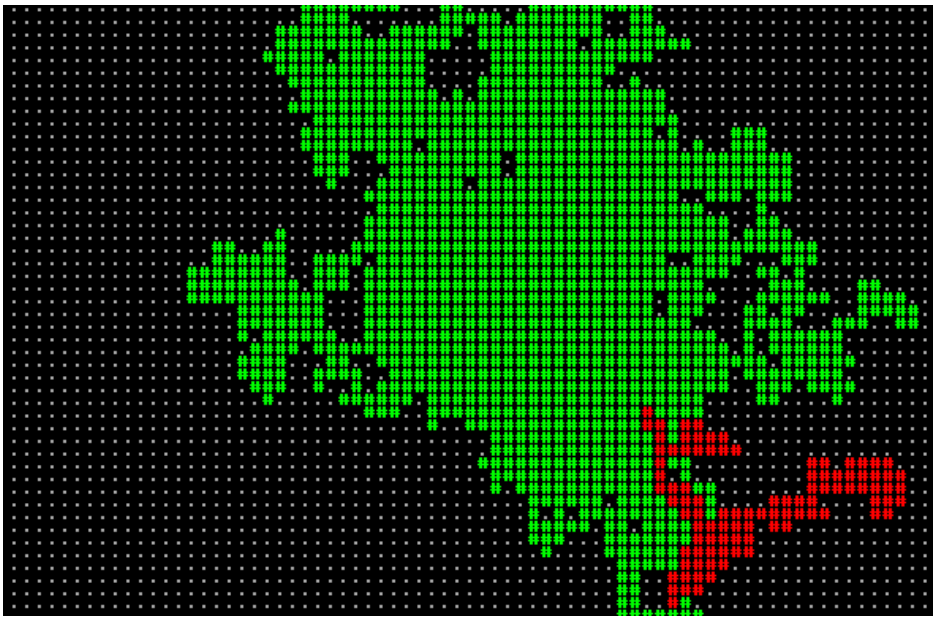


Рисунок 1.6 – Зображення результату генерації завдяки алгоритму «Прогулянка п'яниці»

Алгоритм «Прогулянка п'яниці» починається з повністю заповненого рівня, а потім видаляє його одну клітинку за раз. Алгоритм «прогулянки п’яниці» наступний:

1. Вибрати випадкову точку на заповненій сітці та позначити її порожньою.

2. Вибрати довільний кардинальний напрямок.

3. Рухатись у цьому напрямку та позначити його порожнім, якщо він ще не був.

4. Повторити кроки 2-3, доки не очиститься стільки сіток, скільки необхідно.

«Прогулянка п'яниці» гарантує підключення з першої вибраної сітки, і також може гарантувати, що визначений відсоток сітки було вирізано. Якщо сітка

не велика, необхідно зміщувати напрямок, вибраний до центру сітки, оскільки «п’яниця» може неприродньо стикатися з краями. Також можна змінити крок «п’яниці», щоб вибрати останній напрямок, у якому він рухався, щоб створити довші коридори.

Для отримання оптимальних результатів цей алгоритм вимагає динамічної зміни розміру карти, бо без таких заходів рівні, згенеровані таким чином, можуть виглядати естетично незадовільними.

Зазвичай алгоритм генерує цікаві для гравців печерні рівні, і вони завжди будуть з’єднані, бо ніколи не утворюється двох окремих підпечер, відокремлених одна від одної стіною.

Основна проблема алгоритму полягає в тому, що неможливо передбачити кінцевий результат – він настільки випадковий, що іноді він генерує дуже якісні рівні тунельної печери, а іноді він просто генерує нудну на вигляд краплю, що не сприяє підвищенню рівня зацікавленості гравця.

1.4.4 Дифузійна обмежена агрегація

Агрегація з обмеженою дифузією – це процес, у якому частинки речовини «злипаються», коли вони хаотично рухаються через середовище, що забезпечує певну силу опору. Коли ці частинки з часом «злипаються» разом, вони утворюють характерні фрактальні розгалужені структури, відомі як броунівські дерева.

Дифузійна обмежена агрегація складається з частинок, які приєднуються одна до одної. Алгоритм починається з абсолютно порожнього простору, в який поміщується випадковим чином одна частинка. Інша частинка «запускається» на нескінченну відстань від цієї частинки. Ця нова частинка продовжує рухатися випадковим чином у порожньому просторі, поки не зіткнеться з уже розміщеною частинкою. У цей момент, частинки приєднуються одна до одної й осідають. Тепер третя частинка запускається так само, і та саме процедура повторюється знову і знову. Цей алгоритм був створений внаслідок досліджень броунівського руху.

Алгоритм дифузійної обмеженої агрегації:

1. Розташовується на карті початкова клітина, що зафарбовується.
2. Вибирається випадкова точка будь-де на карті.
3. Вибирається довільний напрямок.
4. Зафарбовується випадкова клітинка:
   – продовжується заливка клітин, поки не потрапить на вже зафарбовану;
   – якщо потрапили в зафарбовану область, вирізається остання суцільна ділянка, через яку було пройдено.

За допомогою даного алгоритму створюються дуже звивисті відкриті карти. Він гарантовано буде суміжним, та є багато способів налаштувати алгоритму так, щоб він видавав цікаві для гравців результати. Результат генерації завдяки алгоритму дифузійної обмеженої агрегації можна побачити на рисунку 1.7.

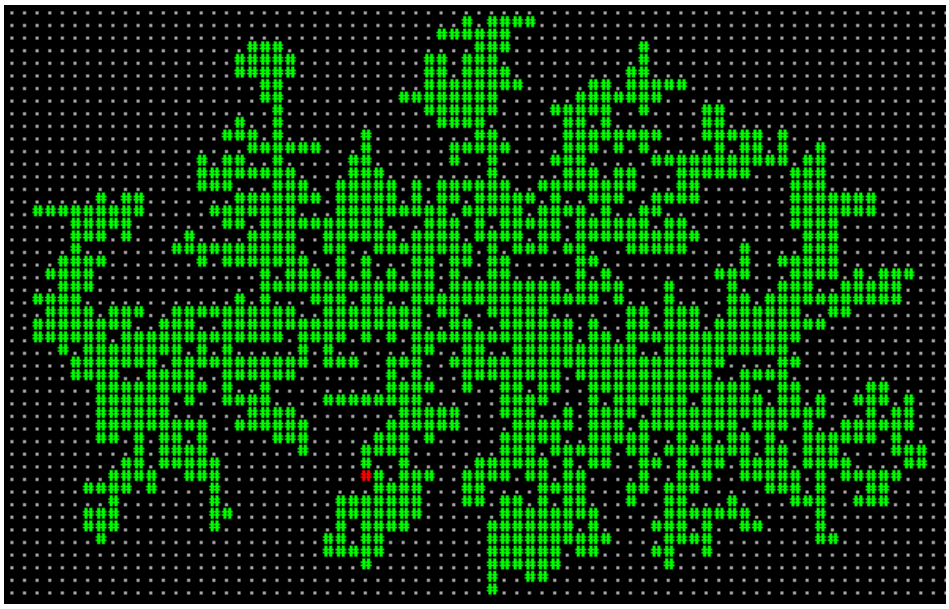


Рисунок 1.7 – Зображення результату генерації завдяки алгоритму дифузійної обмеженої агрегації

Також є такий різновид цієї моделі, як дифузійна обмежена агрегація з центральним атрактом. Особливості моделі дифузійної обмеженої агрегації з центральним атрактом:

– більша ймовірність завжди влучити в ціль;

– випадкове створення початкової точки, а потім вистрілювання частинкою в середину карти;

– допомагає отримати відкритий простір посередині карти;

– більш цікавий візерунок по краях карти;

– можна застосувати симетрію вниз та по вертикалі.

1.4.5 Діаграми Вороного

Діаграми Вороного — це алгоритм, у якому береться будь-який довільний (зазвичай випадковий) набір точок на карті (або таблиці тощо), а після цього алгоритм підраховує найближчу точку для кожного пікселя і призначає піксель для цієї точки. Використання діаграм Вороного часто призводить до більш органічного поділу, ніж прямокутні чи круглі системи, як можна побачити на рисунку 1.8. Різні точки та їх площі можна використовувати для демонстрації різних країн, рослинності, морів та суші, чи майже будь-чого. Його використання також не обмежується рельєфом, його можна використовувати його для штучного інтелекту, або генерації текстури, або будь-чого, що потребує різних значень у 1d, 2d, або 3d таблицях.

Рисунок 1.8 – Зображення результату генерації завдяки алгоритму «Діаграми Вороного»

Припустімо, що є n точок, розкиданих на площині. Діаграма Вороного цих точок ділить площину рівно на n клітинок, які охоплюють частину площини, яка є

найближчою до кожної точки. Це створює тесселяцію, яка повністю покриває площину. Приклади алгоритмів які комбінуються з діаграмами Вороного: триангуляції Делоне, груба сила.

Основні характеристики, притаманні діаграмам Вороного:

– довільно (або навмисно) розміщені точки;

– кожна плитка приєднується до найближчої точки;

– для різних ефектів може бути застосовано різні евристики відстані;

– повторити кожну точку на карті, і приєднати її до області, що належить найближчий точці;

– можна налаштувати результат за допомогою іншого алгоритму відстані, щоб визначити, до якої групи приєднується кожна плитка: піфагорова відстань; манхеттенська відстань;

– при знаходженні краю та розташування там стін утворюється структура багатокутника;

– може використовуватися для визначення розташування/поведінки точки на основі розташування клітини;

– можна використовувати для ефективної генерації міста.

1.4.6 Шум Перліна і симплексний шум

Шум Перліна є основою багатьох процедурних текстур і алгоритмів моделювання. Його можна використовувати для створення мармуру, дерева, хмар, вогню та карт висот для місцевості. Це також дуже корисно для мозаїчних сіток для імітації органічних областей і змішування текстур для цікавих переходів.

Його дуже просто реалізувати, створюючи ряд масивів, що містять «гладкий» шум. Кожен масив називається октавою, плавність різна для кожної октави. Після цього потрібно змішати їх разом.

Коли шум Перліна інтерпретується як карта висот, за його допомогою можна створити цікавий рельєф, як можна побачити на рисунку 1.9. Його також можна використовувати для більш природного розміщення об’єктів на сітці, ніж це можна зробити за допомогою рівномірного випадкового розміщення.

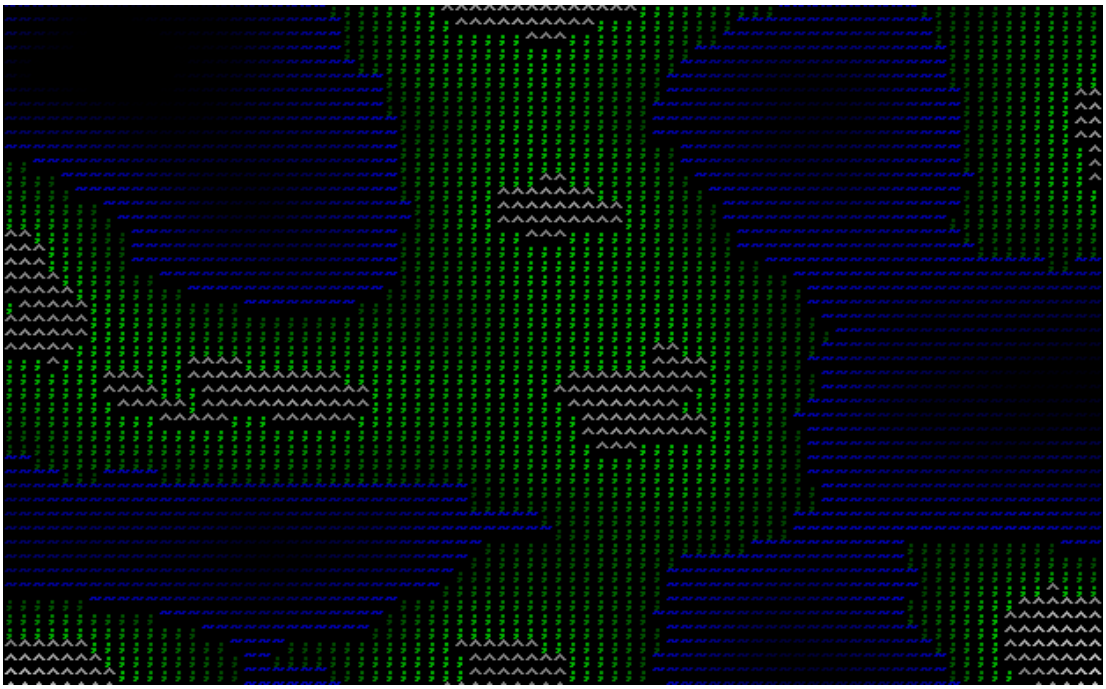


Рисунок 1.9 – Зображення результату генерації завдяки алгоритму «Шум Перліна»

Загальний опис алгоритму наведено далі.

1. Визначити *k* наборів схожих на вигляд об’єктів.

2. Створити шум Перліна, достатньо великий, щоб охопити сітку. Кожен піксель шуму повинен відповідати одній клітинці сітки.

3. Для кожної комірки в сітці потрібно знайти відповідний піксель шуму, щоб вирішити, який слід вибрати об’єкт.

Можна використовувати цей метод, навіть якщо не використовується сітка для розміщення об’єктів. В цьому випадку сітка визначається так, щоб у кожній клітинці був приблизно один об’єкт. Для кожного об’єкта, який потрібно розмістити, спочатку визначається комірка, якій він відповідає, а потім виконуються дії, описані вище. Якщо для цього потрібна занадто велика сітка, можна використати меншу сітку, щоб у кожній клітинці було більше одного об’єкта. Щоб отримати значення Перліна для об’єкта, використовується лінійну інтерполяція.

Щоб побачити характерний малюнок Перліна, світ має бути досить великим. Однак можна отримати користь від розміщення Перліна навіть для маленьких світів. У маленькому світі буде набагато більше варіацій між послідовно згенерованими світами, ніж було б, якби вони були згенеровані іншим методом.

Особливістю алгоритму є те, що навіть при однакових налаштуваннях параметрів генерації світу шуми Перліну можуть виглядати зовсім по різному. Тому доцільно запускати алгоритм кілька разів з однаковими налаштуваннями, щоб переконатися у відсутності аномалій.

Шум Перліна можна зробити плитковим, використовуючи другу ступінь для розмірів масиву. Також можна створити набір плиток, використовуючи набір базових масивів шуму з тим самим першим рядком і першим стовпцем. Усі плитки в наборі будуть рівно укладатися одна з одною.

1.4.7 Карти Дейкстри

Алгоритм Дейкстри працює на основі того, що будь-який підшлях B - D найкоротшого шляху A - D між вершинами A і D також є найкоротшим шляхом між вершинами B і D. Дейкстра використав цю властивість у протилежному напрямку, тобто відбувається переоцінювання відстані кожної вершини від початкової вершини. Потім відвідується кожен вузол і його сусіди, щоб знайти найкоротший підшлях до цих сусідів. Традиційним застосуванням алгоритму є пошук шляху. Взагалі, алгоритм Дейкстри – це жадібний алгоритм, тобто алгоритм робить оптимальний вибір на кожному кроці, намагаючись знайти загальний оптимальний спосіб вирішення всієї проблеми.

Щоб отримати карту Дейкстри, формується масив цілих чисел, який представляє карту, з деяким набором цільових комірок, встановленим на нуль, а всі інші – в дуже велике число. Далі послідовно перебираються комірки «підлоги» карти, непрохідні комірки стін пропускаються. Якщо будь-яка підлогова плитка має значення, що перевищує 1 відносно її сусідньої плитки підлоги з найнижчим значенням (у кардинальному напрямку, тобто вгору, вниз, ліворуч або праворуч,

клітинка поряд з тією, яка перевіряється), встановлюється значення на одиницю більше, ніж його сусід із найменшим значенням. Кроки повторюються, доки не буде внесено жодних змін. Отримана сітка чисел представляє кількість кроків, які знадобляться, щоб дістатися від будь-якої плитки до найближчої мети. Результат генерації карти завдяки алгоритму «Карти Дейкстри» зображено на рисунку 1.10.

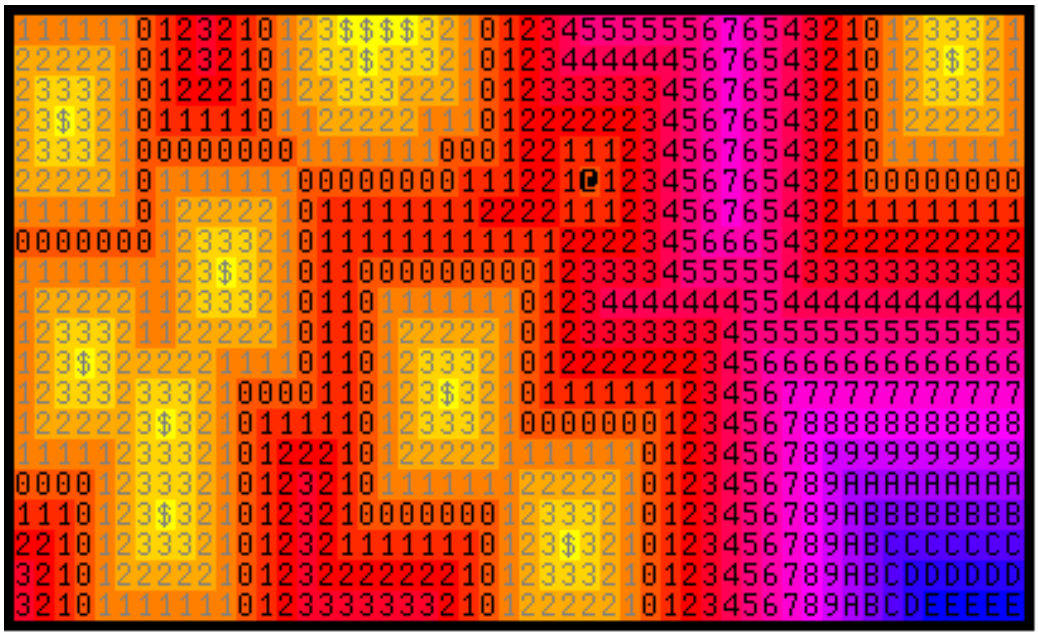


Рисунок 1.10 – Зображення результату генерації завдяки алгоритму «Карти Дейкстри»

1.4.8 Алгоритми тунелювання

Алгоритми тунелювання викопують коридори та кімнати з суцільної місцевості, подібно до того, як міг би це зробити справжній архітектор підземель, хоча іноді трапляються марні або зайві шляхи. Результат генерації карти завдяки алгоритмам тунелювання зображено на рисунку 1.11.

Його ідея полягає в тому, що кожна нова кімната з'єднується з іншою через тунель. Так, крок за кроком, на рівні з'являється все більше і більше кімнат, які обов'язково мають бути пов'язані одна з одною.

Особливість цього підходу в тому, що в кожній кімнаті є лише один вхід і вихід, а це не завжди цікаво з точки зору геймплею. Ця проблема вирішується завдяки наданню кожному осередку певної вартості переміщення: у підлоги вона

нижча, а біля стін – вища. Саме тому, зазвичай алгоритм вважає за краще створити тунель, пересуваючись по підлозі, але іноді він «пробиває» стіну – так виходять закільцьовані карти або кімнати з кількома проходами.

Рисунок 1.11 – Зображення результату генерації завдяки алгоритмам тунелювання

Інше правило, якого можна дотримувався для покращення методу тунелювання: кімнати не повинні бути простими прямокутниками. Він створює кімнати, комбінуючи разом випадкову кількість прямокутників – зазвичай від одного до чотирьох. Алгоритм починає з одного прямокутника як основу, а потім з'єднує його з іншими прямокутниками довільного розміру. При цьому, вони можуть накладатися друг на друга.

Під час генерації рівня, кімнати не можуть перекривати одна одну, а відстань між приміщеннями можна налаштовувати.

Коли з'являється нова кімната, вона відокремлена від інших. Щоб з'єднати її з локацією, застосовується алгоритм пошуку шляху. Суть цього алгоритму у тому,

що він вираховує найменшу «вартість» шляху та генерує маршрут. Після цього вздовж тунелю утворюються переходи між кімнатами.

1.4.9 Клітинні автомати

Клітинний автомат – це сітка комірок, кожна з яких має стан і правило для визначення того, до якого стану переходить клітина, на основі її стану та стану сусідів. Усі клітинки мають однаковий набір правил, але клітини водночас мають змогу перебувати у різних станах. Усі переходи між станами відбуваються одночасно; тобто всі клітини оцінюють стан своїх сусідів і вирішують, на що змінити, після цього за один такт часу відбувається зміна стану всіх клітин автомату [12].

Сітка може бути будь-якого розміру (наприклад, лінія або куб клітинок). Можливі більш складні приклади, наприклад спіральні клітинні автомати, які складаються з кількох орбіт, що обертаються з різними періодами, де кожна орбіта містить різну кількість клітин: сусіди включають початкові та кінцеві клітини, а також будь-які клітини, які в даний момент є сусідніми в навколишніх внутрішній та зовнішній орбітах [13].

Загалом, клітинний автомат – це модель системи «клітинних» об’єктів з такими характеристиками:

- клітини живуть на сітці;
- кожна клітинка має стан, кількість можливостей станів, як правило, обмежена; у найпростішому прикладі є дві можливості: 1 і 0 (інакше їх називають «увімкнена» і «вимкнена» або «живий» і «мертвий»);
- кожна клітинка має околиці - це можна визначити різними способами, але зазвичай це список суміжних клітинок. Найбільш відомі типи сусідства клітин, це окіл фон Неймана і окіл Мура, вони зображені на рисунку 1.12.

Сітка – це просто таблиця довжиною M x N клітинок.

Кожна клітинка – це комірка, яка в будь-який момент часу відображає один із двох чи більше станів. Стан може приймати різні форми, проте

найпоширенішою є просто двійкове значення, тобто 0 або 1, яке зазвичай позначається порожньою (білою) або заштрихованою (чорною) клітинкою.

Околиці – це визначений набір сусідів, які впливають на стан однієї комірки за допомогою певної функції.

Набір правил – це функція, яка виводить стан окремої клітинки на основі виводу її околиць.

У загальному вигляді клітинний автомат можна продемонструвати за допомогою формули (1.3):

$$(V, X, C, f) \tag{1.3}$$

де V – решітка (нескінченна або скінченна); X – кінцевий набір значень або станів клітинки; C – кінцева околиця; F – локальна функція переходу, що правилом, або таблицею переходів.

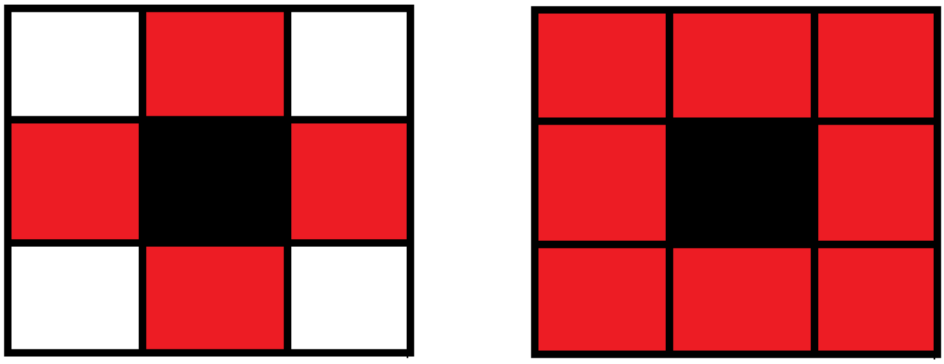


Рисунок 1.12 – Окіл фон Неймана (а) та окіл Мура (б)

Кожна комірка може приймати один із можливих станів k, і стани змінюються від 0 до k 1. Значення комірки і в момент часу t позначається як xt. Околиці завжди складаються з діапазону r найближчих сусідів комірки, яку потрібно оновити (xt), і її самої. Отже, околиця складається з 2r + 1 клітинок. Клітинний автомат, який містить ряд станів k = 2 і r = 1 (один сусід ліворуч і один

сусід праворуч, на додаток до самої комірки, яка буде оновлюватися), визначають як одновимірний елементарний клітинний автомат.

Елементарний клітинний автомат – це одновимірний клітинний автомат, у якому є два можливих стани, які позначаються 0 і 1, а правило для визначення стану клітини в наступному поколінні залежить лише від поточного стану клітини та двох її безпосередніх сусідів.

Можливе число комбінацій з двома сусідами і самою коміркою, та двома станами це 8 комбінацій, вони зображені на рисунку 1.13.

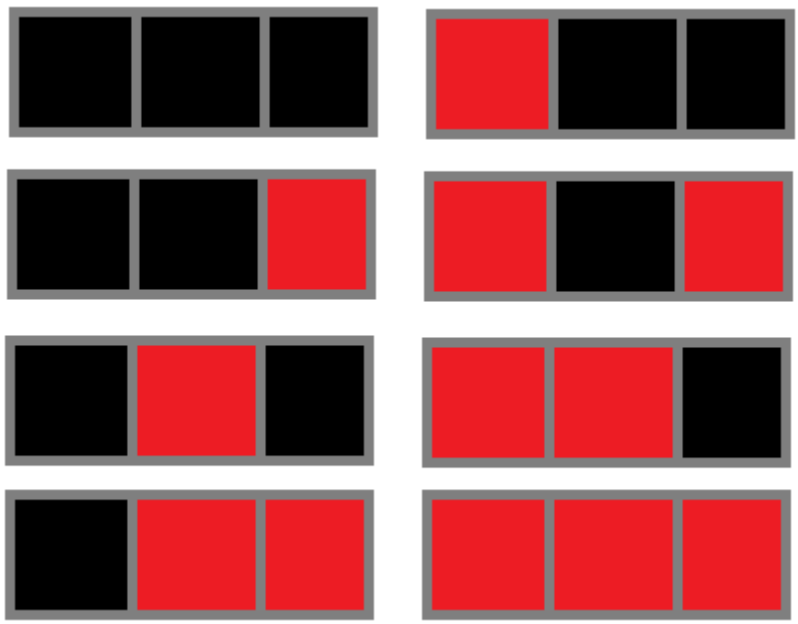


Рисунок 1.13 – Можливі комбінації елементарного клітинного автомату

Комірки матимуть тільки один з двох наявних станів, тобто число числу можливих станів що може набувати клітинка відповідає 2. Клітинка може приймати, наприклад, 0 або 1, біле або чорне, і так далі. Число 3 відповідає кількості сусідів, тобто лівому сусіду, самій комірці та правому сусіду, що передбачає діапазон околиць, що дорівнює r = 1.

З цими 8 можливими комбінаціями можна сформувати загалом 256 локальних правил, тобто для кожної комбінації клітинка, яка буде оновлюватися, може отримати 0 або 1. Тоді ми можемо визначити, що кількість комбінацій дорівнює k2r+1, а кількість локальних правил дорівнює kk2r+1.

У двовимірному клітинному автомату кожна клітинка має більшу околицю у порівнянні з елементарним, а отже варіативність застосування також зростає. Можливі стани що приймають двовимірні клітинні автомати так само 0 та 1, кількість сусідніх клітин збільшилося до дев’яти. З трьома клітинками отримується 3-розрядне число, або вісім можливих конфігурацій, а з дев'ятьма клітинками 9 бітів, або 512 можливих околиць.

Гра «Життя» – це двовимірний клітинний автомат, винайдений британським математиком Джоном Конвеєм у 1970 році. Відмінною рисою спеціального набору правил Конвея «Game of Life» є те, що він повний за Тьюрингом. Повнота за Тьюрингом — це властивість, яка описує, що мова програмування, моделювання або логічна система в принципі придатні для вирішення будь-якої обчислювальної проблеми. У грі життя кожна клітина сітки може прийняти один із двох станів: мертвий або живий. Ігрове поле поділено на рядки та стовпці та в ідеалі має нескінченний розмір. Гра в життя керується чотирма простими правилами, які застосовуються до кожної клітинки сітки в області моделювання:

– жива клітина гине, якщо вона має менше двох живих сусідніх клітин;
– жива клітина з двома-трьома живими сусідами продовжує жити;
– жива клітина з більш ніж трьома живими сусідніми клітинами гине на наступному часовому етапі;
– мертва клітина оживає, якщо в ній є рівно три живі сусідні клітини.

На рисунку 1.14 можна побачити як ці правила будуть відображатися на стані клітинки. У наступній ітерації у випадку a, центральна комірка помре, а у випадку b, навпаки, повернеться до жття, або народиться, бо клітини повністю притримуються правил гри «Життя».

Клітинні автомати часто використовують тороїдальну топологію області моделювання. Це означає, що протилежні краї сітки з’єднані один з одним. Стовпець сітки що крайній праворуч, є сусідом із крайнім стовпцем сітки ліворуч, верхній рядок сітки є сусідом із нижнім рядком сітки, і навпаки, відображення цього можна побачити на рисунку 1.15. З такою топологією гарантується безперешкодна передача інформації про стан комірок через межі мережі.

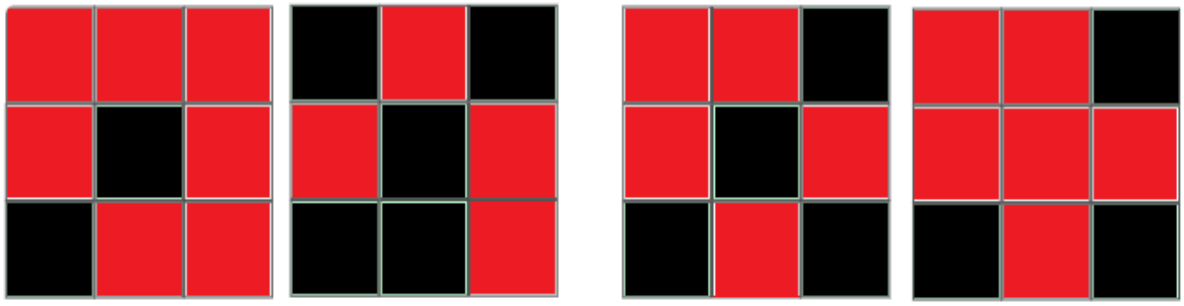


Рисунок 1.14 – Зображення правил гри «Життя»

Інший тип обмежень полягає в тому, щоб обробляти всі клітинки за межами сітки так, ніби вони перебувають у фіксованому стані. Для гри в життя це означає, що вони розглядаються як мертві Перевага цих граничних умов для гри в життя полягає в тому, що результуючі шаблони, такі як планери, не можуть знову потрапити в зону моделювання на протилежній стороні краю. Це можна використовувати, наприклад, щоб запобігти руйнуванню планерної гармати планерами, які вона виробляє.

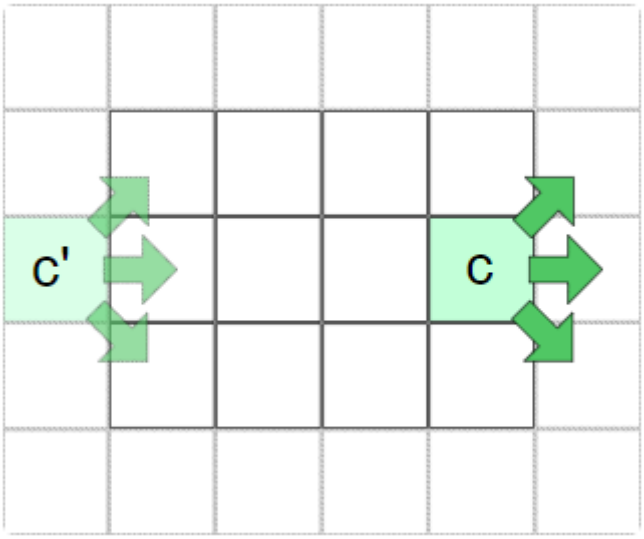


Рисунок 1.15 – Зображення як протилежні рядки/стовпці з’єднані та утворюють тороїдальну топологію області моделювання

Під час типового проходження гри в життя певні шаблони з’являтимуться часто. Деякі з них статичні (натюрморт), інші коливаються, а треті рухаються в

зоні моделювання. Деякі візерунки навіть можуть створювати інші візерунки. У наведеній нижче таблиці подано огляд шаблонів, які часто зустрічаються в грі життя.

Натюрморти – це статичні споруди, які не змінюються з часом, приклад такої споруди зображено на рисунку 1.16.

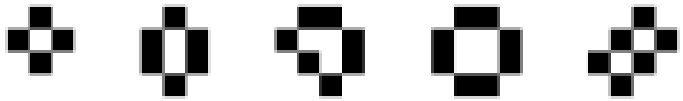


Рисунок 1.16 – Зображення статичних споруд, які не змінюються з часом

Осцилятори – це структури, які періодично повторюються з часом, зберігаючи своє положення, приклад такої споруди зображено на рисунку 1.17.

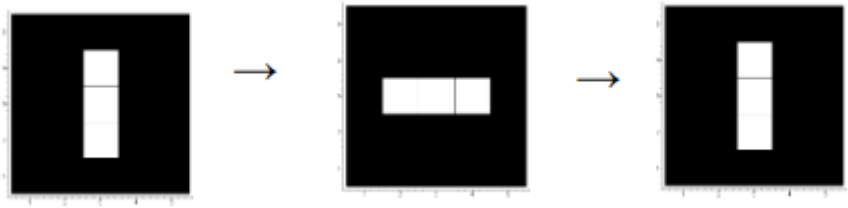


Рисунок 1.17 – Зображення Осцилятора «Індикатор»

Космічні кораблі – це конструкції, які пересуваються по ігровому полю "Гри в життя", приклад такої споруди зображено на рисунку 1.18.

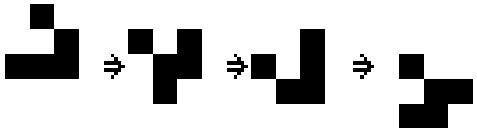


Рисунок 1.18 – Зображення Конструкції космічного корабля «Планер»

2 РОЗРОБКА МЕТОДУ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ДЛЯ ГРИ В ЖАНРІ ROGUELIKE НА ОСНОВІ КЛІТИННОГО АВТОМАТУ

2.1 Постановка задачі автоматизованої генерації ігрової карти для гри в жанрі Roguelike

Процедурна генерація контенту відноситься до алгоритмічного створення вмісту. Вони дозволяють автоматично генерувати вміст і, отже, може значно зменшити робоче навантаження розробників. Деякі процедурні методи генерації контенту поступово стають звичайною практикою в ігровій індустрії, але здебільшого обмежуються дуже конкретними контекстами та елементами гри, як наприклад процедурна генерація ігрової карти.

Загалом, те, чого бракує поточним процедурним методам генерації ігрових карт, це не продуктивність, а більш потужний, точний і багатший контроль над процесом генерації. Прописані заздалегідь критерії якості, пов’язані з грою, можуть забезпечити високий рівень контролю.

Дуже важливим елементом будь-якої процедурної генерації є контроль, який він забезпечує при створенні карти для гарних результатів. Цей контроль досягається завдяки властивостям генерації.

Потрібно визначити набір опцій, для керування процесом генерації, які впровадити у створюване програмне забезпечення, щоб цілеспрямовано керувати процесом генерації рівня, а також параметри, які можна буде змінювати. Контроль також визначає, якою мірою редагування цих опцій і параметрів спричиняє значущі зміни для генерації кінцевої карти.

Належний контроль гарантує, що генерація буде створювати узгоджені результати (наприклад, доступні для гри рівні, або такі патерни карт, на які ми очікуємо при встановленні певних параметрів), зберігаючи як набір бажаних властивостей, так і варіативність.

Відсутність відповідних параметрів або відсутність розуміння того, що саме робить параметр, майже завжди означає поганий контроль над автоматизованою генерацією, що може призвести до небажаних результатів або навіть катастрофічних збоїв.

Раніше при розробці елементів автоматизованої генерації у іграх акцент був більше зосереджений на тому, як працювали методи та що можна було згенерувати. На сьогоднішній день, розробників ігор стало більше турбувати те, як вони можуть досягти бажаних результатів. Значущі параметри були введені, щоб допомогти створити вміст для певного результату. У міру ускладнення автоматизованої генерації і поєднання різних методів за допомогою яких її можна реалізувати, також виникла потреба в більшому контролі. Таким чином, контроль елементів утворених завдяки автоматизованій генерації розширився до ще більш природної взаємодії, включаючи використання ігрового процесу як значущого параметра керування.

Тісний зв’язок між ігровим процесом і ігровим простором є важливим аспектом при створенні карт, а тим більш підземель. Таким чином, контроль над автоматизованою генерацією при створенні підземель повинен залежати від вимог ігрового процесу. Це можна зробити більш (наприклад, параметри, що стосуються складності рівня) або менш (наприклад, параметри, що стосуються топології) явними способами.

Для визначення критеріїв якості слід дослідити основи, які забезпечують керування генерацією рівня підземелля на основі ігрового процесу, та дослідити які основні елементи має карта в іграх жанру roguelike.

2.2 Ключові елементи ігрової карти в іграх жанру roguelike

Для визначення ключових елементів ігрової карти гри жанру roguelike, розглянемо основні функції що притаманні іграм цього жанру:

– процедурно згенеровані середовища;
– вічна смерть;

– рандомізація ідентифікаторів предметів.

Жанру притаманні ще покрокові бої, середовище на основі сітки та вбивство монстрів як ключові функції, але саме ці основні три функції знайшли застосування в різних типах ігор і тісно пов’язані з терміном roguelike. Отже, технічно кажучи, roguelike більше стосується спільного набору механік та принципів дизайну, ніж жанру як самого по собі, саме тому дуже важливо притримуватися цього набору механік при розробці будь-якого контенту пов’язаного з іграми цього жанру.

Процедурна генерація вказує на те, що елементи гри, особливо дизайн рівнів, не жорстко закодовані заздалегідь дизайнером, а замість цього генеруються випадковим чином із шаблону кожної гри. У результаті це означає, що гравець не може запам’ятати вигідні локації або місця, яких слід уникати, і тому повинен грати в кожну гру наосліп, кожен раз насолоджуючись ігровою картою якої не бачив до цього. Хоча особливості карти рандомізовані, загальна форма карти залишається незмінною для кожної гри.

Ключові елементи ігрової карти в іграх жанру roguelike такі:

– крапля – обмежена зона певної конфігурації по якій може пересуватися гравець. краплі можуть бути у формі печер та кімнат;

– печера – крапля, що має неправильну, довільну форму;

– кімната – крапля, що має форму багатокутника;

– тунель – елемент карти, який об'єднує між собою краплі; тунелі поєднують як одну краплю з іншою, так і багато крапель разом; тунелі можуть мати довільні розгалуження.

Процедурна генерація має дві основні переваги для загального дизайну ігор жанру roguelike.

1. Оскільки ігрове поле кожного разу інше, це збільшує можливість відтворення, оскільки карта завжди невідома. Це звільняє гравця від необхідності запам’ятовувати конкретні складності рівня самі по собі, тому гравець повинен намагатися адаптуватися до шаблону, що лежить в самій основі складності не одного рівня, а механіки в цілому. Хоча гравець ніколи не можете знати напевно,

що знаходиться за сусідніми дверима, є змога розвинути у нього відчуття того, що може бути за ними.

2. Гарантується, що кожне проходження є дослідженням нового простору. Оскільки карта ніколи не буває однаковою, гравець завжди буде занурюватися в невідоме, незалежно від того, чи це його перша гра, чи тисячна.

Наскільки випадковими мають бути рівні, повинен визначити розробник. Він повинен проаналізувати усі аспекти механіки та критерії якості гри що розроблює, бо процедурна генерація потребує ретельного тестування. Якщо розташування надто випадкове, складність між іграми може надто різко відрізнятися: одні будуть дуже легкими, а інші неможливо виграти. Залежно від складності гри, це може бути дуже інтенсивний процес. Потрібно переконатися, що всі можливі випадкові карти цікаві та доступні для проходження. Найкраще розроблені ігри жанру roguelike змушують гравця відчувати покарання за помилки, а не робити так, щоб сама гра генерувала несправедливо складні карти.

Складність вдосконалення карт зростає, чим більші рівні. Більшість ігор використовують більш обмежений підхід, сегментуючи карту за рівнем або областю, тому створені області є відносно меншими, і тому їх легше повністю перевірити.

Переглянувши ключові елементи, механіки та особливості жанру roguelike, були сформовані критерії якості ігрової карти.

Критерії якості ігрової карті жанру roguelike:

– кількість печер;
– розмір печер;
– згладжування печер;
– кількість тунелів;
– довжина тунелів;
– прямолінійність тунелів;
– кількість входів та виходів печер;
– розміри карти;
– відсутність печер, не поєднаних тунелями з іншими печерами.

Відповідно до цих критеріїв є змога проаналізувати процедурно згенеровану карту, зрозуміти, чи задовольняє вона потреби для застосування у певній грі жанру roguelike.

Карта для гри у жанрі roguelike відіграє одну з найважливіших ролей у всій грі, бо якщо вона виявиться поганою, то гравцю швидко набридне в неї грати. Випадковість це чудово, але потрібно вміти правильно налаштувати її, щоб карта була водночас і випадковою, і підпорядковувалася певним правилам і налаштуванням.

2.3 Використання клітинного автомату при автоматизованій генерації ігрових карт

Одним із методів процедурної генерації підземель є клітинні автомати. Ця структура складається з сітки комірок будь-якої кінцевої кількості вимірів. Кожна клітинка має посилання на набір клітинок, які складають її сусідство, і початковий стан у нульовий момент часу. Щоб обчислити стан комірки в наступному поколінні, набір правил застосовується до поточного стану комірки та сусідніх. Після кількох поколінь у сітці можуть формуватися шаблони, які значною мірою залежать від використовуваних правил і станів комірок [14].

Репрезентативною моделлю клітинного автомата є сітка клітин та їх станів. Прикладом набору дозволених станів може бути клітинка та напрямок, так що комірки представляють або місце, куди гравець може перейти, або місце, куди гравець не може перейти.

В роботі використовуються можливості самоорганізації клітинних автоматів для створення ігрової карти що складається із печерних структур. Вони визначають околиці клітини як вісім оточуючих клітин, використовуються околиці Мура, де їх можливими станами є сама печера, фон, тобто зона у якій немає печери, та край карти.

Після початкового випадкового перетворення комірки від фону до печери, набір правил ітеративно застосовується в кількох поколіннях. Цей набір правил

можна прописувати в залежності від потреб генерації карти, яка буде потрібна у кінцевому результаті [15].

Наприклад, випадковим чином вибираються клітини. Якщо обрана клітина має більше сусідів відповідно до околиці мура ніж обране значення дозволених сусідів, наприклад 3, то клітина закривається. У іншому випадку, клітина відкривається. Цей крок повторюється таку кількість разів, скільки ітерацій було обрано.

На основі клітинного автомату, а саме прописаних у ньому правил, можна створювати структури, схожі на печери, як показано на рисунку 2.1.

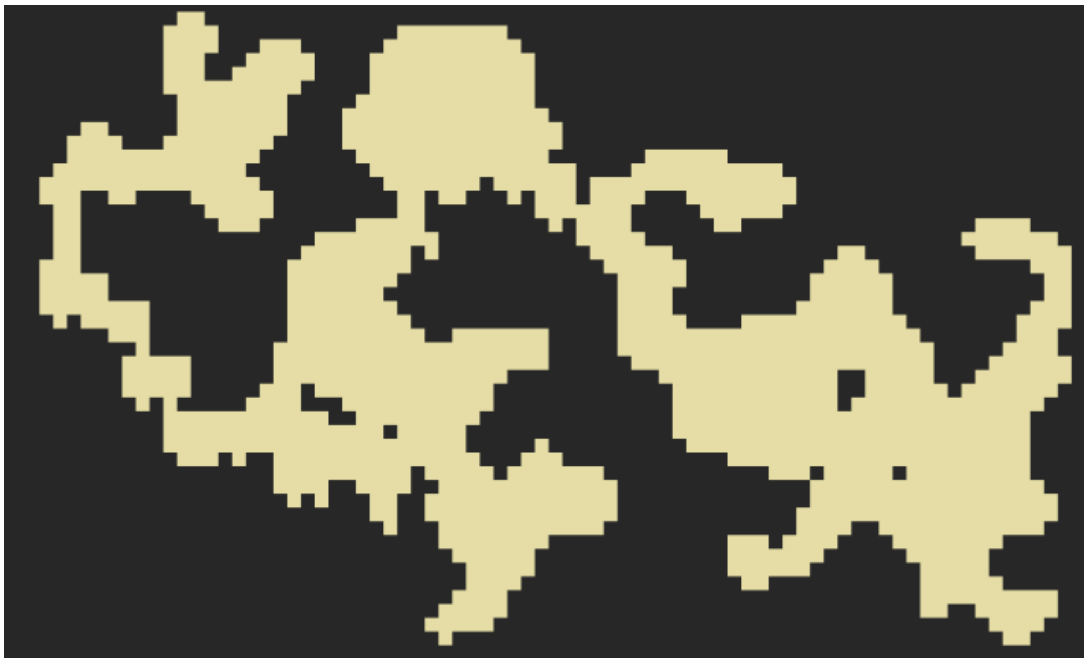


Рисунок 2.1 – Приклад карти згенерованої завдяки використанню клітинного автомату

Серед цікавих особливостей цього методу, можна підкреслити:

– ефективність;

– здатність генерувати нескінченні рівні;

– здатні до розширювання, бо є змога покращити прості правила за допомогою інших методів;

– інтерактивний, бо є змога вносити корективи та модифікувати його коли виконуються операції;

– розмір карти не впливає на складність обчислень, або витрачений на них час;

– відносно простий алгоритм створення (стани та правила, що використовуються, прості);

– природне, хаотичне відчуття, яке мають рівні, створені цим методом.

Основними недоліками методу автоматизованої генерації ігрової карти з використанням клітинних автоматів, є відсутність прямого контролю згенерованих карт без додаткових налаштувань та методів, та той факт, що цей метод застосовується лише до 2D карт [16]. Для створення ігрової карти для ігор жанру Roguelike це не проблема.

Є можливість автоматизованої генерації ігрових карт у 3D з використовуванням клітинних автоматів, однак поточні проблеми керування, ймовірно, будуть гіршими в 3D. Крім того, зв’язок між будь-якими двома згенерованими печерами, тобто доступними областями, не може бути гарантований одним лише алгоритмом, його потрібно систематично перевіряти та додавати, якщо його немає, або використовувати інші методи у поєднанні з клітинним автоматом.

Невелика кількість параметрів і те, що вони відносно інтуїтивно зрозумілі, є перевагою цього підходу. Однак це також є одним із недоліків методу: важко повністю зрозуміти вплив одного параметра на процес генерації, оскільки кожен параметр впливає на кілька функцій згенерованих карт.

Будь-який зв’язок між цим методом генерації та функціями ігрового процесу повинен здійснюватися шляхом проб і помилок.

Для повного розкриття потенціалу клітинних автоматів у автоматизованій генерації ігрової карти, його слід поєднати з іншими методами генерації, наприклад для прокладення тунелів, або для пофарбування елементів карти у певний колір.

2.4 Розробка схеми функціонування додатку для автоматизованої генерації ігрової карти

Для подальшого розуміння бажаного результату, та розробці моделей для реалізації, було створено схему функціонування додатку для автоматизованої генерації ігрової карти, яку можна побачити на рисунку 2.2. Цю схему було використано також при розробці програмного забезпечення що реалізує створення ігрової карти жанру Roguelike на основі клітинного автомату.

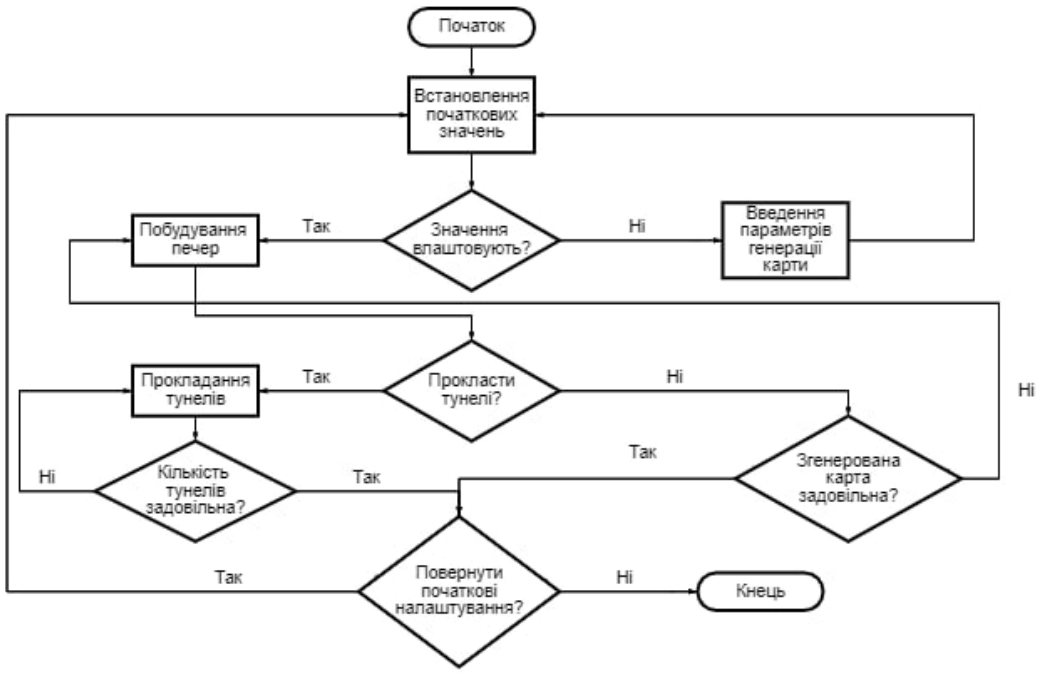


Рисунок 2.2 – Схема функціонування додатку для автоматизованої генерації ігрової карти

Аналізуючи схему, можна позначити, що карта складається з печер, порожніх клітин, можуть бути тунелі, та околиці. Налаштування усіх елементів на карті здійснюється завдяки зміні встановлених початкових значень.

На старті, додаток функціонуватиме з початковими значеннями, які користувач матиме змогу змінювати. Якщо значення не влаштовують, то потрібно змінити його на потрібне. Коли всі значення влаштовують, то відбувається генерація печер. Після цього, за потребою, є змога прокладання тунелів. Кількість

разів, що тунелі покладатимуться, можна контролювати. Якщо утворена карта не задовільна, то можна створювати печери та прокладати тунелі поки результат не влаштовуватиме користувача. У кінці отримується бажана карта, або є варіант змінити налаштування, та отримати зовсім інші карти.

2.5 Модель карти на основі клітинного автомату

Карта що буде автоматизовано згенерована, складається з комірок, й може бути довільної форми, але мати чітке формулювання потреб для реалізації майбутньої карти.

При побудові моделі ігрової карти, уявимо її як дискретний простір, компонентами якого є з печери, порожніх клітини та тунелі. Після генерації об’єкти типу печера та тунелі не можуть змінювати своє розташування на карті, а порожні клітини можуть змінюватися при прокладенні додаткових тунелів. Однак при кожній новій генерації, усі об’єкти на карті вільно переміщуються, й за прописаними алгоритмами та визначеними налаштуваннями розміщуються на карті.

Математична модель карти – множина клітин M, розміщених у вигляді решітки, де фрагменти решітки – клітини $m_{(x,y)}$:

$$m_{(x,y)} \in M. \tag{2.1}$$

Множина клітин M карти, задається сукупністю множин трьох типів, що не перетинаються:

$$\begin{aligned} M = {} & M_P \cup M_e \cup M_t, \\ & M_p \cap M_e \cap M_t = \emptyset, \end{aligned} \tag{2.2}$$

де Mp – множина клітин що утворюють печери,

Me – множина порожніх клітин,

Mt – множина клітин що утворюють тунелі.

Графічне зображення моделі ігрової карти відображено на рисунку 2.3. На рисунку елементи множини Mp зображені зеленим кольором (клітини що утворюють печери), елементи множини Me – сірим (порожні клітини), елементи множини Mt – блакитним (клітини, що утворюють тунелі).

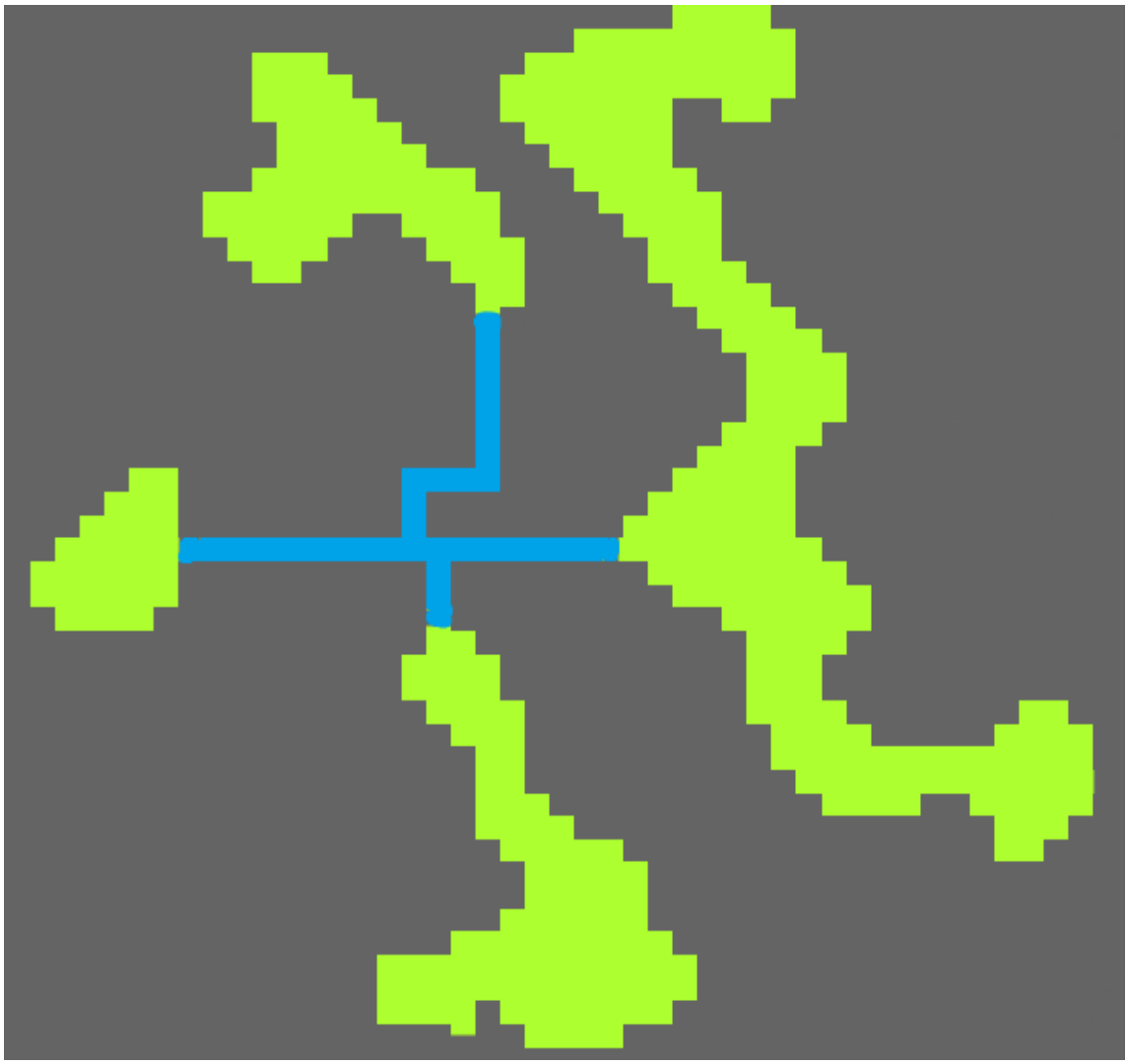
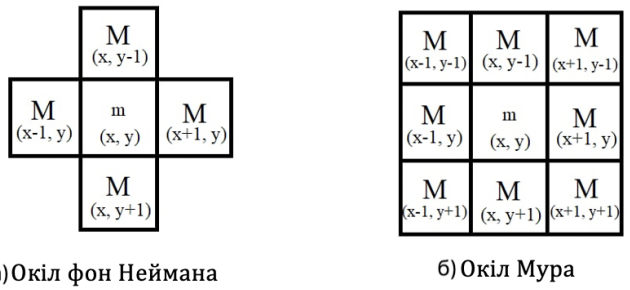


Рисунок 2.3 – Модель ігрової карти

Множини Mp та Mt є константними та не змінюють свого вмісту (об’єкти типу печер та тунелі після генерації не пересуваються). У множині Me може змінюватися склад (створення тунелю). Об’єднання множин Mp та Mt на множині Me є константною множиною (окрім себе, тунелі можуть простягатися тільки у множині порожніх клітин, не переходячи за грані множин печер).

Усі сусіди клітин на карті визначаються за околицею фон Неймана, що зображено на рисунку 2.4а, або за околицею Мура, що зображено на рисунку 2.4б.



а) Окіл фон Неймана б) Окіл Мура

Рисунок 2.4 – Зображення а) окіл фон Неймана, та б) окіл Мура

Правила, за якими змінюються стани клітин автомату для задачі автоматизованої генерації карти визначаються наступним списком.

Правило для генерації початкових печер.

1. Якщо клітина помічена, як печера, то її стан не змінюється.

2. Інакше для клітини генерується випадкове значення p - ймовірності того, що клітина буде печерою. Якщо p більше заданого порогового значення, то клітина помічається, як печера.

Правило для згладжування печер.

1. Якщо обрана клітина має 3 або більше сусіда-печери відповідно до околиці Мура, то клітина помічається, як печера.

2. Інакше, клітина помічається як пуста.

Правило для уникнення пустот в печерах

Якщо клітина має 4 сусіда-печери відповідно до околиці Мура, то клітина помічається, як печера.

2.6 Алгоритм заливки

У поєднанні з клітинним автоматом було вирішено використовувати алгоритм заливки, який допомагає відвідати кожну точку карти на певній території.

Алгоритм заливки – це алгоритм, який в основному використовується для визначення обмеженої області, підключеної до певного вузла в багатовимірному масиві.

Маючи матрицю з деякою початковою точкою та деяким пунктом призначення з кількома перешкодами між ними, цей алгоритм допомагає знайти шлях від джерела до пункту призначення.

Коли виявляється порожня клітинка, цей алгоритм допомагає виявити сусідні клітинки. Цей крок виконується рекурсивно, доки не будуть виявлені клітинки з номерами.

Алгоритм заливки можна просто змоделювати як задачу обходу графа, представляючи задану область у вигляді матриці та розглядаючи кожну клітинку цієї матриці як вершину, яка з’єднана з точками над нею, під нею, праворуч від неї та ліворуч від неї а у випадку 8-зв'язків також до точок обох діагоналей.

Наприклад, береться кольорове зображення, яке можна представити у вигляді двовимірного масиву пікселів. Кожен піксель у цьому двовимірному масиві має колір. За допомогою цього алгоритму потрібно змінити колір певної області на новий колір.

Спочатку алгоритм заливки приймає два параметри:

Початкова позиція – параметр, який представляє індекс комірки, яку збираються перефарбувати з усіма пов’язаними комірками, які мають однаковий початковий колір.

Новий колір – параметр, який представляє новий колір, який дана область матиме після застосування алгоритму заливки.

Для простоти вважатимемо сусідами кожної комірки, клітинки вгорі, внизу, ліворуч і праворуч від клітинки, тобто за околицями фон Неймана. Крім того, щоб розглядати комірку як частину зв'язної області, вона повинна мати той же колір, що і початкова.

Наприклад, створюється 2D масив як зображено на рисунку 2.5 а, параметр початкової позиції вказується j(2, 2), а як параметр нового кольору передається блакитний. Як можна побачити на рисунку, колір комірки j(2, 2) зелений, тому

перефарбовуються усі пов’язані комірки, які мають однаковий колір, на блакитний, як зображено на рисунку 2.5 б.

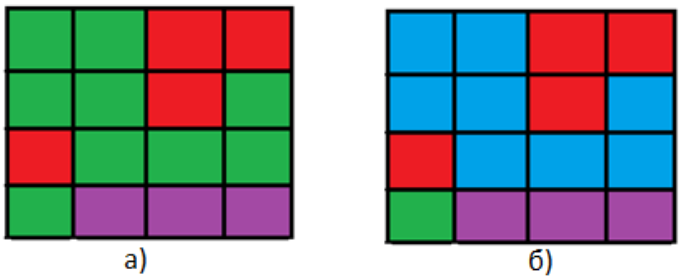


Рисунок 2.5 – Зображення зміни кольору клітин а) після застосування алгоритму заливки б)

Потрібно зауважити, що розфарбовуються не лише сусіди початкової клітинки, розглядається сусіди початкової комірки та продовжується знаходження сусідів кожної нововиявленої комірки. Операція триває до тих пір, поки не буде виявлено нових сусідніх клітин.

Крім того, нижня ліва клітинка все ще має зелений колір після завершення операцій. Причина полягає в тому, що початкова клітинка не є частиною зв’язаної області.

Сфери використання алгоритму заливки наведено нижче.

1. Використовується в інструменті заповнення ковша в програмах малювання для заповнення з’єднаних областей однакового кольору іншим кольором. Можна помітити, що при натиску лише на один піксель зображення, всі клітинки пов’язаної області, пов’язані з клітинкою на яку натиснули, змінюють колір на новий.

2. У іграх, для визначення того, які фігури очищені. За приклад візьмемо гру «Сапер», при натисканні на якусь клітинку сітки, яка не має ні міни, ні номера. У цьому випадку гра відкриє всі клітинки, які порожні та знаходяться в одній сполучній області. Однак у цьому випадку є одне оновлення алгоритму, яке зупиняється, коли досягається клітинки, яка містить число.

3. Перевірити, чи дві клітинки знаходяться в одній з’єднаній області. Для цього використання можна надати унікальний ідентифікатор для кожної підключеної області заздалегідь. Пізніше, коли отримується запит про дві клітинки всередині сітки, потрібно лише перевірити їх відповідні ідентифікатори, та перевірити, чи вони збігаються чи ні.

В роботі цей алгоритм використовується для пошуку клітин, що містять у собі печери, беручи до уваги околиці фон Неймана.

2.7 Розробка методу автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату

Метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike розроблено на основі клітинного автомату, а саме використовуючи модифіковану форму гри «Життя». Схематично створений метод зображено на рисунку 2.6.

До гри «Життя» додаються такі правила для створення печер:

1. Відвідується кожна клітинка на карті, і використовується випадкова ймовірність, щоб визначити, чи закривати цю клітинку;
2. Клітинки відвідуються випадковим чином, і кількість сусідів у клітинці використовується для визначення того, закривати чи відкривати її, тобто якщо випадково вибрана комірка має більше сусідів ніж дозволено (найчастіше 3), то клітинка закривається, у іншому випадку – відкривається;
3. Кожна клітина переглядається окремо, виконавши кілька проходжень по карті, видаляються клітини з 3 або більше порожніми сусідами, це потрібно для згладжування;
4. Відвідується кожна клітинка на карті, та заповнюється кожна порожня клітина, що має 4 заповнених сусіда.

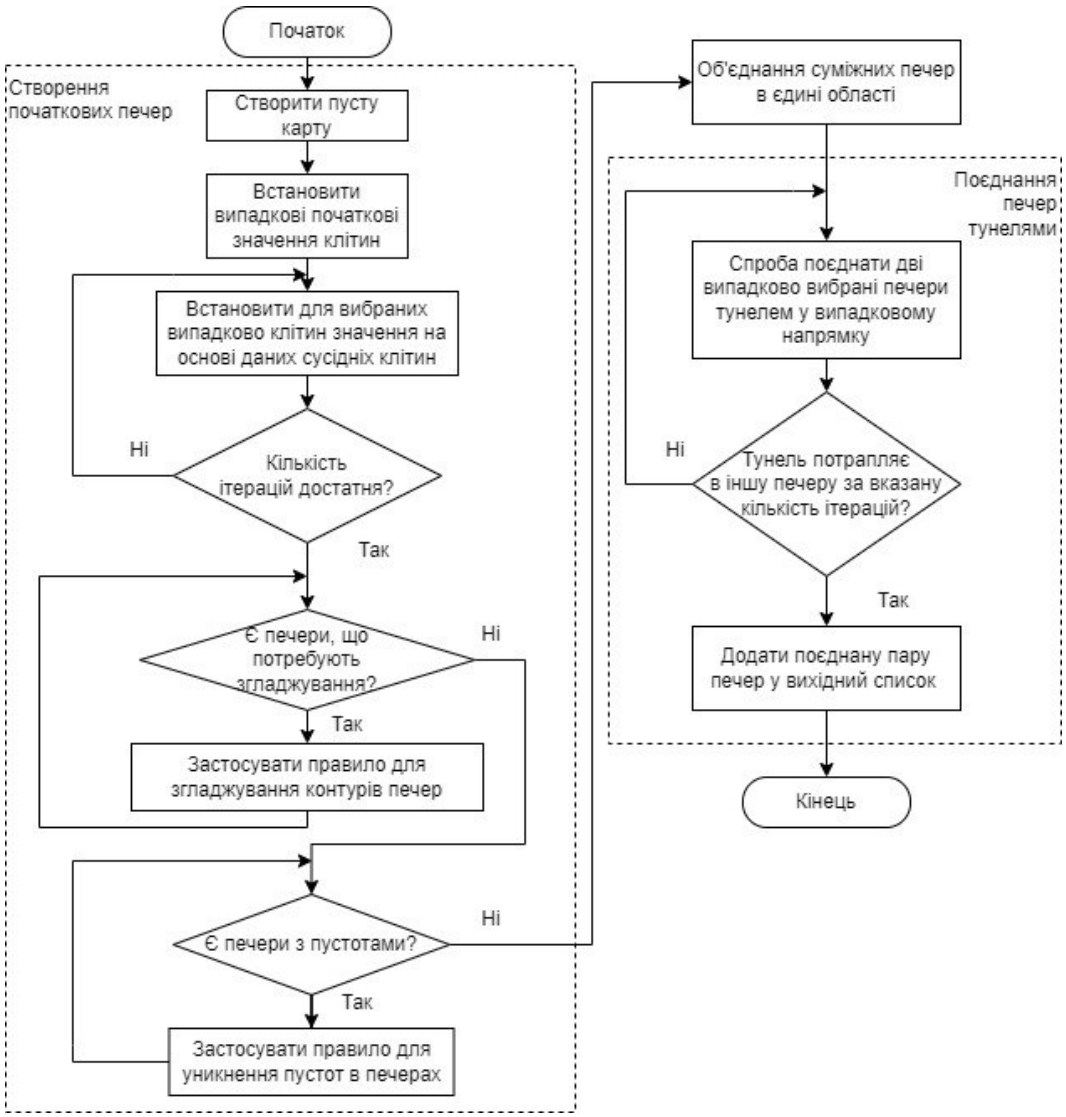


Рисунок 2.6 – Схема створеного методу автоматизованої генерації ігрової карти для гри в жанрі Roguelike розробленого на основі клітинного автомату

Повторюючи другий крок тисячі разів, починають утворюватися краплі-печери. Закрита клітинка вважається краплею, відкрита – порожньою клітиною.

Шляхом коригування різних доступних властивостей характеристики сформованих печер можуть значно змінитися, що може призвести до формування дуже цікавих печерних систем.

Після створення печер, алгоритм на основі заливки розміщує кожну печеру та дані для кожної печери в загальному списку. За допомогою цього списку печер, їх можна легко з’єднати разом.

Щоб спробувати з’єднати печери, використовується процедура побудови тунелю, завдяки алгоритмам тунелювання. Це працює шляхом випадкового вибору печери та розвитку тунелю у випадковому напрямку та спостереження, чи потрапляє він в іншу печеру протягом визначеного періоду часу. У разі успіху дві з’єднані печери розміщуються в списку, і робиться подальша спроба з’єднати одну з цих печер з іншою.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ДЛЯ ГРИ В ЖАНРІ ROGUELIKE

Програму було розроблено за допомогою комп’ютеру, який має наступні характеристики:

– процесор – AMD Ryzen 5 3600 3.6(4.2)GHz 32MB sAM4 Tray;

– відеокарта – Gigabyte GeForce RTX 3060 Gaming OC 12288MB;

– материнська плата – MSI B450M Mortar Max;

– оперативна пам'ять HyperX DDR4-3200 16384MB PC4-25600 (Kit of 2x8192).

Продукт було розроблено за допомогою операційної системи Windows 10, та може бути використаним на комп’ютерах з операційною системою Windows.

3.1 Опис використаних програмних засобів

3.1.1 Figma

Для проектування майбутнього програмного забезпечення було обрано використання Figma. Figma – це універсальний інструмент для проектування, який досить популярний серед дизайнерів, він є одним із найдосконаліших інструментів для створення цифрових інтерфейсів. Figma є інструментом, який найчастіше використовують ті, хто працює з прототипуванням, дизайном інтерфейсу користувача UI і користувальницьким досвідом UX. Як інструмент із функціями дизайну та редагування векторів, Figma можна використовувати для різних цілей. Загалом, її використовують для досягнення таких цілей:

- розробка адаптивних інтерфейсів для додатків, веб-сайтів і програмного забезпечення, тобто інтерфейсів, які адаптуються до формату екрана пристроїв з різними розмірами та конфігураціями;
- розробка прототипів і навігаційних потоків;
- створення та впровадження Design Systems (систем проектування);
- створення електронних листів та мистецтва для соціальних мереж;
- розробка презентацій, електронних книг, інфографіки тощо.

За допомогою деяких плагінів і функцій, таких як Figmotion , LottieFiles і Chart, також можна виконувати деякі інші види діяльності, такі як створення анімації та графіки з реальними даними за допомогою інструменту.

Для проектування додатку автоматизованої генерації ігрової карти використовувалися екрани прототипів і потоків. Прототип — це «версія» продукту від середньої до високої точності, яка імітує, як кінцевий продукт буде представлено на екранах. Створюючи екранні потоки, він також імітує, як може відбуватися взаємодія користувача з екранами цього продукту. Екранні потоки – це шляхи, якими користувач може пройти між різними екранами та меню програми чи веб-сайту. Наприклад, коли ви відкриваєте програму соціальної мережі, ви, ймовірно, матимете головний екран, екран повідомлень, екран самої розмови, ще один екран із вашим профілем і так далі. Користувач може пройти незліченну кількість шляхів через екрани програми чи веб-сайту. Роль прототипів і їхніх екранних потоків полягає в тому, щоб вказати, як виглядатимуть усі ці можливості. Також є змога перевірити, як працюють екранні потоки, у шаблоні презентації відтворення. Під час входу в режим відтворення, окрім переходу з одного екрана на інший, можна натискати кнопки та посилання прототипу та мати реалістичне уявлення про те, як просувається проект.

Таким чином, Figma за своїми характеристиками повністю підійшла для створення прототипу додатку для автоматизованої генерації ігрової карти.

3.1.2 Visual Studio

Visual Studio – це інтегроване середовище розробки (IDE). Тобто це програмне забезпечення текстового редактора, яке дозволяє користувачам писати свій код певною мовою, який потім перетворюється в команди для комп’ютеру. Інтерфейс Visual Studio зображено на рисунку 3.1.

За допомогою Visual Studio відбувається розробка веб-програм ASP.NET, веб-служб XML, настільних програм та мобільних додатків.

Visual Studio допомагає у розробці програмного забезпечення. Незалежно від мови, яка використовується, він показує доступні ярлики API, а також команди автозавершення, щоб пришвидшити створення та написання коду.

Коли закінчується написання коду або є бажання перевірки коду, Visual Studio надає можливість його налагодити. Проблему великого обсягу коду Visual Studio вирішує завдяки можливості стежити впродовж написання коду за допомогою початкових тегів, фільтрів і функцій.

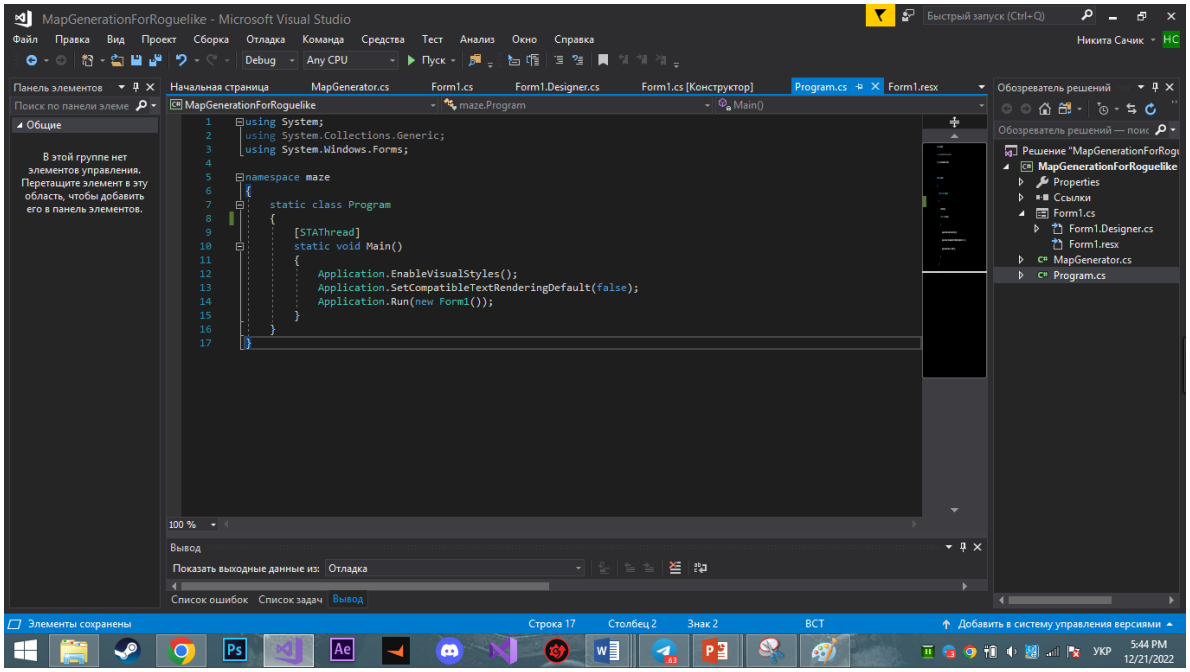


Рисунок 3.1 – Інтерфейс Visual Studio

Такі параметри, як «Знайти всі посилання», дозволяють шукати інформацію та відображати результати пошуку в коді. Також можна відобразити код у структурованому вигляді, що полегшує пошук проблем і завжди інформує про розташування структур у написаному коді. Крім того, є маленькі піктограми у формі лампочки, які допомагають перевіряти та вирішувати поширені проблеми програмування під час написання коду для виконання гнучких подій, таких як рефакторинг і впровадження інтерфейсів у самому редакторі. Під час налагодження коду Visual Studio покаже список помилок. За допомогою нього є змога визначити, в якому рядку коду є синтаксична чи логічна помилка. У деяких

мовах відображаються навіть варіанти вирішення деяких помилок. Visual Studio дозволяє зберігати налаштування. Для цього потрібно просто увійти у профіль програми, де зберігаються зміни конфігурації. Для області проектування, наприклад, нема потреби вводити код для проектування компонентів. Зрештою, Visual Studio має палітру інструментів, у якій є можливість просто перетягувати компоненти – значки, кнопки, тощо, у вікно створення інтерфейсу та впорядкувати їх відповідно до потреб. Visual Studio можна використовувати для написання коду на C#, C++, VB (Visual Basic), Python, JavaScript та багатьох інших мовах програмування. Він підтримує 36 різних мов програмування, та доступний для Windows і macOS.

3.1.3 Мова програмування C#

Для створення додатку що реалізуватиме автоматизовану генерацію ігрової карти, було обрано мову програмування C#. Мова програмування C# – це об’єктно-орієнтована мова програмування, яка була розроблена Microsoft і є частиною платформи .NET. Хоча мова C# була створена з нуля, вона була заснована на мові C++ і містить багато елементів мов Pascal і Java. Однією з характерних рис C# є те, що ця мова програмування не містить жодного набору бібліотек класів або функцій. Натомість мов працює з посиланням на платформу .NET, звідки походять її класи або функції виконання.

Як і Java, C# використовує концепцію віртуальної машини. Це CLR (Common Language Runtime), який є свого роду віртуальним комп’ютером, який керує виконанням програм, які використовують платформу «.net».

Оскільки C# є об’єктно-орієнтованою мовою, він підтримує такі поняття, як інкапсуляція, успадкування та поліморфізм. Усі змінні та методи інкапсульовані у визначеннях класу. Він використовується в більшості класів .NET framework. Це була перша компонентно-орієнтована мова сімейства C/C++, вона відповідає стандартам ООП, де все походить від спільного батьківського елемента, у випадку C# це System.Object.

Мова програмування C# спрощує багато складнощів C++, надаючи такі потужні можливості, як типи нульових значень, перерахування, делегування, лямбда-вирази та прямий доступ до пам’яті, підтримку загальні методи та типи, що веде до кращої безпеки типу та продуктивності.

C# суворо типізована мова, має підтримку DLL, COM і COM+, чутлива до регістру, і її класи можуть реалізувати кілька інтерфейсів, але успадкування за розширенням просте. Створені програми працюють у керованому середовищі, залишаючи платформу .NET для здійснення контролю пам’яті. Мова має функцію Language Integrated Query (LINQ), яка забезпечує вбудовані можливості запитів до різноманітних джерел даних.

Методи та типи не потрібно оголошувати по порядку, файл C# може містити численні класи, структури та інтерфейси. Інновації C# дозволяють швидко писати програми, зберігаючи при цьому виразність і елегантність мов C-Style. Це дозволяє його розробникам створювати безліч програм, сумісних із платформою .NET, таких як традиційні програми Windows, веб-служби, розподілені компоненти, клієнт-серверні програми та програми з інтеграцією бази даних, серед інших типів.

C# є кросплатформною мовою. Тому можна використовувати її для розробки веб-платформ, мобільних пристроїв і настільних програм. Завдяки практичності цієї мови можна відносно легко переходити від простіших проектів до складних і кросплатформних проектів.

Деякі функції C# допомагають створювати довговічні та надійні програми, наприклад:

– збірка сміття: автоматично відновлює пам'ять, зайняту невикористаними та недоступними файлами;

– обробка особливих ситуацій: надає комплексний і структурований підхід для виявлення та відновлення помилок;

– типізований дизайн: унеможливлює читання неініціалізованих змінних та масивів індексів за їх межами. це також запобігає виконанню неперевірених перетворень типів.

Можна програмувати на C# за допомогою фреймворку «.Net Core». Це відкритий вихідний код, який дозволяє програмам нормально працювати на інших платформах, включаючи Linux і Mac. Він також дозволяє використовувати інші веб-сервери, наприклад Apache. Важливо підкреслити, що C# постійно оновлюється, щоб стати доступнішим і, отже, полегшити розробку програмного забезпечення.

3.1.4 Windows Forms

Для зображення інтерфейсу додатку що реалізуватиме автоматизовану генерацію ігрової карти, було обрано Windows Forms. Windows Forms – це набір керованих бібліотек у .NET Framework, розроблених для створення багатофункціональних клієнтських програм. Це графічний API для відображення даних і керування взаємодією користувачів зі спрощеним розгортанням і кращою безпекою в клієнтських програмах. Створення форми можна побачити на рисунку 3.2.

API (Application Programming Interface) – це набір коду, який доступний в операційній системі для виконання найрізноманітніших функцій. Вони включають створення візуального вигляду елементів інтерфейсу користувача, таких як кнопки, текстові поля, прапорці тощо. Як наслідок, ці компоненти по суті не підлягають налаштуванню, навпаки, вони дуже схожі на розглянуту операційну систему.

Windows Forms пропонує велику бібліотеку клієнтів, що забезпечує інтерфейси для доступу до власних елементів інтерфейсу Windows і графіки з керованого коду. Він побудований на керованій подіями архітектурі, подібній до клієнтів Windows, тому програми що розробляються очікують на введення даних користувача для запуску.

Windows Forms схожа на бібліотеку MFC (Microsoft Foundation Class) у розробці клієнтських програм. Він надає оболонку, що складається з набору класів C++ для розробки програм Windows. Однак він не надає стандартну структуру додатків, як це робить MFC.

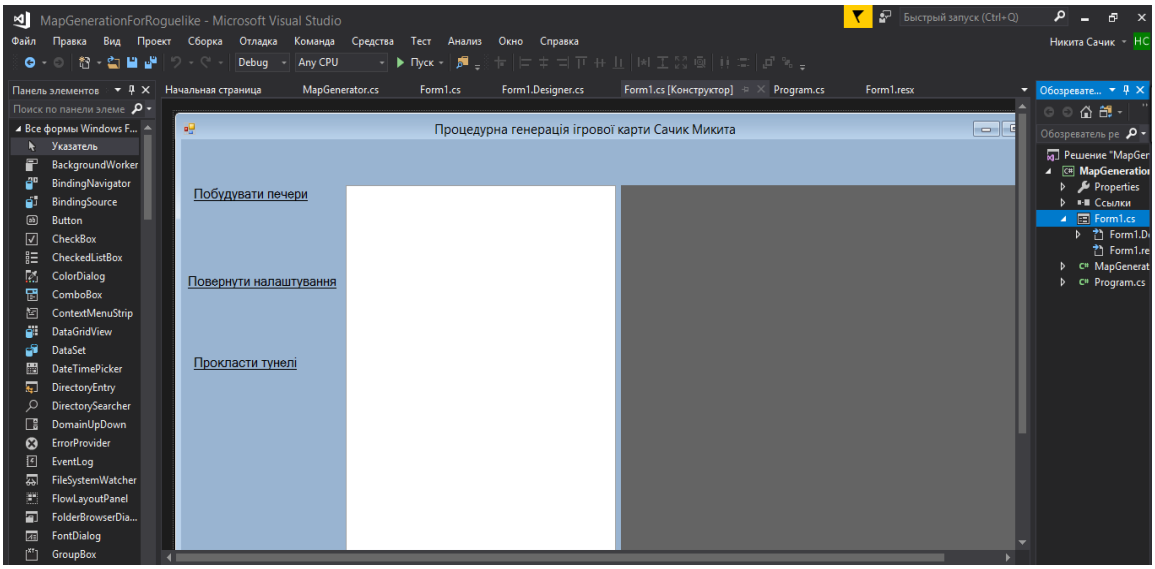


Рисунок 3.2 – Скріншот форми, розробленої завдяки Windows Forms

Кожен елемент керування в програмі Windows Forms є конкретним екземпляром класу. Макет елемента керування в графічному інтерфейсі користувача та його поведінка керуються за допомогою методів і засобів доступу. Windows Forms надає різноманітні елементи керування, такі як текстові поля, кнопки та веб-сторінки, а також параметри для створення спеціальних елементів керування. Він також містить класи для створення пензлів, шрифтів, значків та інших графічних об’єктів (таких як лінія та коло).

Класи Windows Forms можна розширити за допомогою успадкування, щоб спроектувати структуру програми, яка може забезпечити високий рівень абстракції та повторного використання коду.

3.2 Опис структури проекту

Структуру проекту «Map Generation For Roguelike» можна побачити на рисунку 3.3.

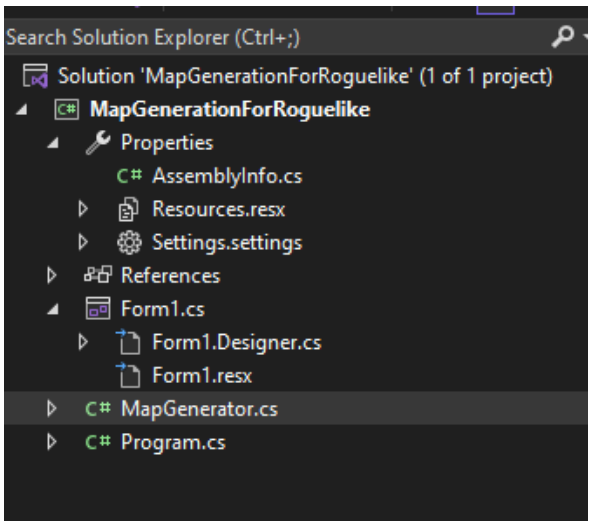


Рисунок 3.3 – Структура проекту Map Generation For Roguelike

Структура проекту складається з таких елементів:

– MapGenerationForRoguelike – головний файл проекту, він містить всі налаштування проекту, посилання на всі ресурси що ми використовуємо;

– Properties – налаштування проекту;

– AssemblyInfo.cs – інформація про збірку проекту;

– Resources.resx – використовуються для зберігання налаштувань програми та інших ресурсів програми;

– References – посилання на бібліотеки що використовуються;

– Form1.cs – Графічний дизайн додатку;

– Form1.Designer.cs – логіка та загальні налаштування головної сторінки проекту;

– MapGenerator.cs – програмний код що реалізує основний функціонал;

– Program.cs – точка входу в програму;

3.3 Опис інтерфейсу

Інтерфейс розробленого додатку містить три кнопки, область для заповнення данних, та PictureBox у якому буде знаходитись згенерована карта, сам інтерфейс можна побачити на рисунку 3.4.

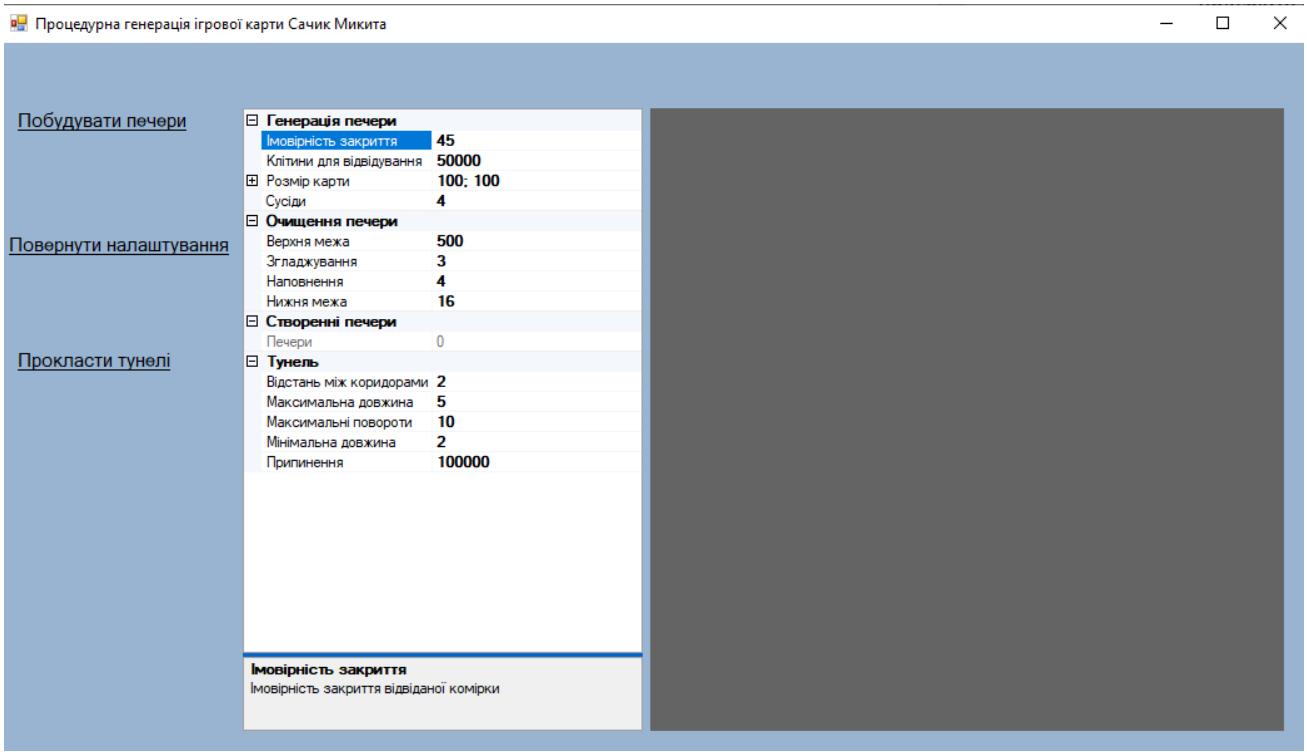


Рисунок 3.4 – Головна сторінка розробленої програми

Функціонал, який реалізується при натисканні кнопки можна описати так:

1. Натиск на кнопку «Побудувати печери» за допомогою прописаних методів створює ігрову карту та виводить її у PictureBox (рисунок 3.5).

2. Натиск на кнопку «прокласти тунелі» створює коридори між острівками на вже створеній нами раніше карті (рисунок 3.6).

3. Натиск на кнопку «Повернути налаштування» видаляє данні які ми заповнювали у ListBox, й повертає всі значення до початкових.

Тобто для початкової генерації карти натискаємо на «Побудувати печери», а далі вже у нас є можливість змінювати її зовнішній вигляд переписуючи властивості.

Розроблена програма має властивості, які можна налаштовувати, й отримувати зовсім різні результати.

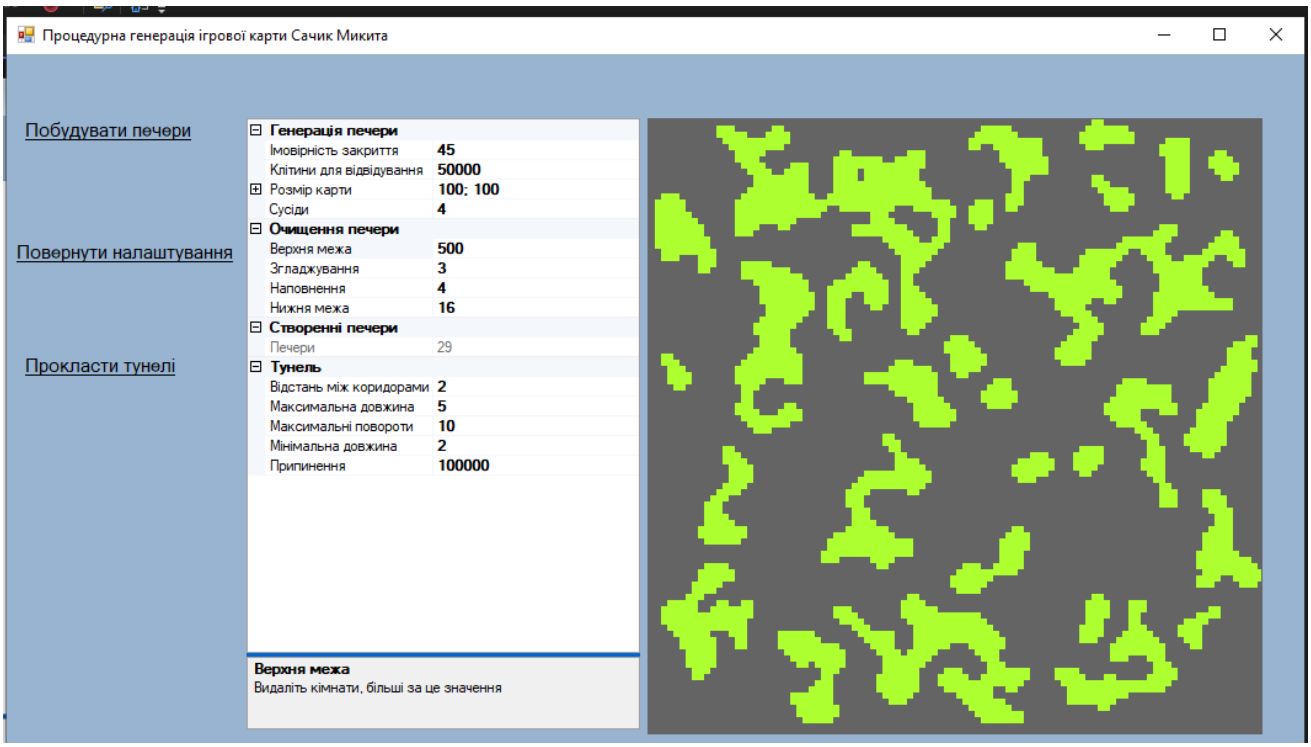


Рисунок 3.5 – генерація ігрової карти

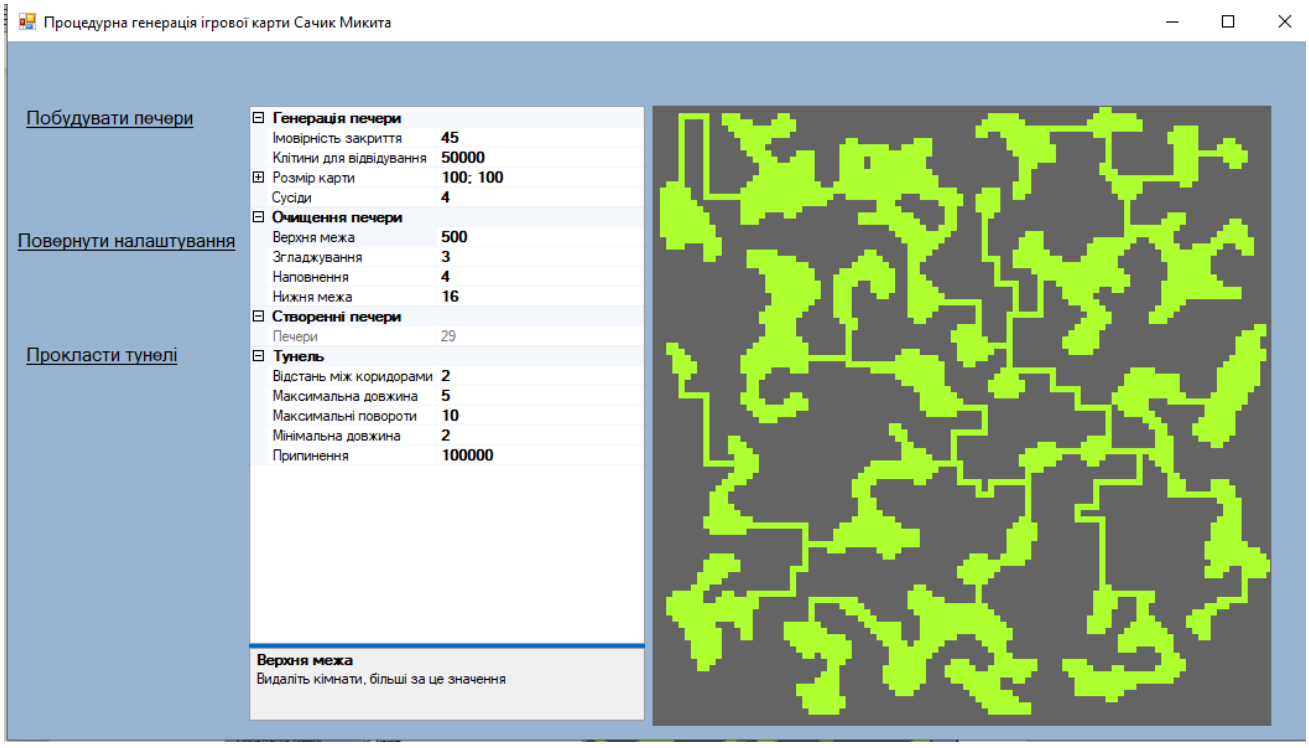


Рисунок 3.6 – Прокладання тунелів між створеними островами

Властивості за допомогою яких можна регулювати генерацію ігрової карти та за що вони відповідають:

1) Генерація печери:

– імовірність закриття – це значення визначає ймовірність закриття комірки яку ми відвідали;

– клітини для відвідування – потрібно вказати кількість клітин, які потрібно відвідати під час процесу генерації;

– розмір карти – потрібно ввести розмір карти, яку ми збираємося створити;

– сусіди – коли це значення дорівнює або перевищує кількість сусідів, змінює стан комірки;

2) Очищення печери;

– верхня межа – максимальний дозволений розмір печер. якщо печера більша за це число, її буде видалено;

– згладжування – видаляє окремі клітини з країв печери, клітинка з такою кількістю порожніх сусідів видаляється;

– наповнення – заповнює отвори в печерах, відкрита клітина з цією кількістю закритих сусідів закрита;

– нижня межа – мінімально допустимий розмір печер. якщо печера менше цього числа, її буде видалено;

3) Створені печери;

– печери – загальна кількість згенерованих печер;

4) Тунель;

– відстань між коридорами – кількість порожніх клітинок, які маємо мати по обидва боки коридору, щоб його можна було побудувати; це залежить від напрямку, в якому ми рухаємося. якщо ми подорожуємо на північ, то повинні мати таку кількість порожніх клітин на схід і захід від нас;

– максимальна довжина – максимальна довжина коридору;

– максимальні повороти – максимальна кількість змін напрямку, яку може зробити коридор під час його будівництва;

– мінімальна довжина – мінімальна довжина коридору;

– припинення – значення лічильника, яке при перевищенні припиняє спроби створення коридорів. зупиняє процес побудови коридору від зависання.

Як можна побачити на рисунку 3.6, створено карту, що містить не тільки окремі острівки, а повноцінну систему підземель, у якій кожний елемент за допомогою нашого алгоритму поєднується з іншим, таким чином створюючи повноцінну печерну систему. Можна створювати більш ніж одну систему прокладання тунелів, як показано на рисунку 3.7.

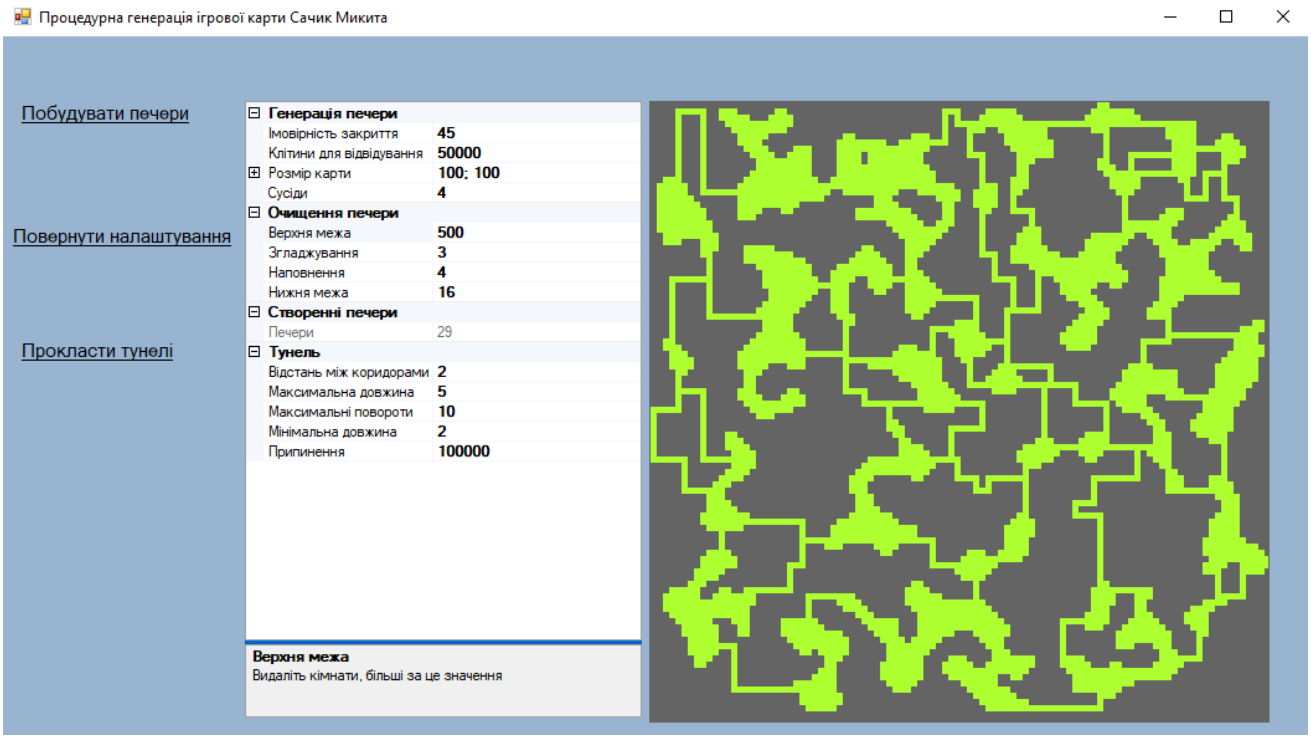


Рисунок 3.7 – Демонстрація багаторазового прокладання тунелів на карті

Збільшення кількості тунелів може зробити створену нами карту ще більш неочікуваною та цікавою. Не слід генерувати карту та створювати якомога більше тунелів коли це не потрібно, треба визначитися яка саме карта нам потрібна, та вже виходячи з цього, налаштовувати параметри карти.

3.4 Реалізація патернів ігрової карти

Завдяки великій кількості налаштувань, що можна змінити, з’являється необмежена кількість цікавих з точки зору ігрового процесу, рельєфів ігрової карти. При установці деяких параметрів на певні значення, можна отримувати одну концепцію, але карти усе одно будуть різними. Далі розглянуто декілька патернів, що реалізують типові концепції.

Перш за все потрібно поставити задачу, яка повинна виконувати карта при генерації. Можна створювати зовсім різні карти, від карт, які нагадують лабіринти (рисунок 3.8), так і карти які являють собою суцільний острів (рисунок 3.9).

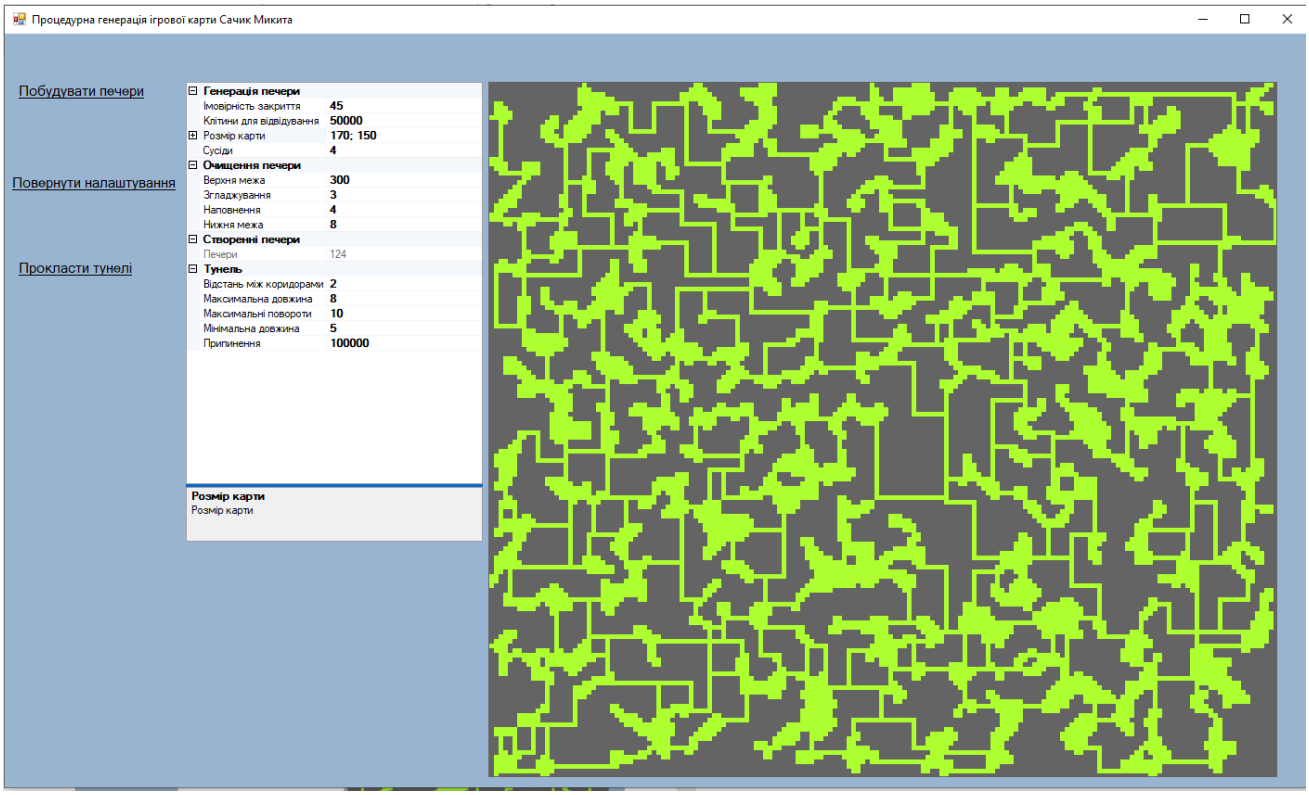


Рисунок 3.8 – Приклад ігрової карти з великою кількістю тунелів

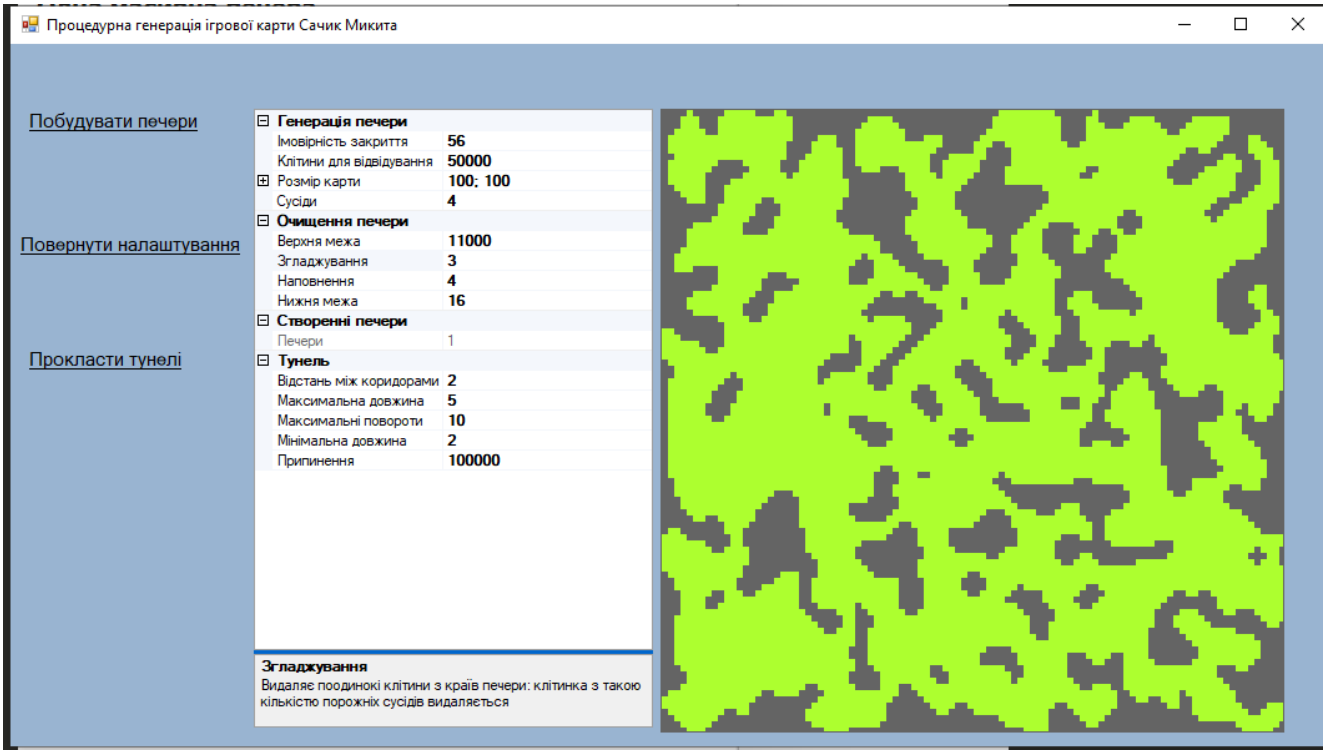


Рисунок 3.9 – Згенерована карта суцільного острова

Карта яка зображена на рисунку 3.8 нагадує лабіринт, який містить у собі багато островів, які з’єднані великою кількістю тунелів. Головна ідея у тому, що створюється велика кількість невеликих островів, та вони поєднуються за допомогою довгих тунелів, які утворюють систему довгих коридорів, що перетинаються один з одним, і мають багато розгалужень.

Такий результат досягається завдяки зміні деяких налаштувань у програмі, а саме:

– зміна верхньої межі;
– зміна нижньої межі;
– максимальна довжина тунелів;
– мінімальна довжина тунелів;
– розмір карти.

При зміні верхньої межі та нижньої межі, острови на карті стають значно дрібнішими, це потрібно щоб збільшити їх кількість та для більшої кількості тунелів між ними.

Коли змінюється максимальна та мінімальна довжина тунелів, то визначається які саме тунелі потрібні на даний момент. У цьому випадку потрібні більші тунелі, тож потрібно збільшити їх значення максимальної та мінімальної довжини.

Карта збільшується щоб простір, кількість тунелів та згенерованих печер також збільшилася, а ландшафт ще більше був схожий на лабіринт. У результаті змін в властивостях, отримуємо 126 островів, а потім поєднуємо їх завдяки тунелям.

Створена карта з великою кількістю довгих коридорів що поєднують дрібні островки чудово пасуватиме такому жанру ігор як Roguelike. Ці тунелі можуть стати мостами, вентиляційними шахтами, тощо, й у них можуть міститися цінні скарби для гравців які ретельно досліджують карту й кожен піксель, або ж можна заповнити тунелі монстрами.

Протилежністю до карти з великою кількістю островів та згенерованих між ними тунелів є мапа що складається з суцільного острова, й не має потреби у генерації додаткових тунелів (рисунок 3.9).

Для отримання такого візерунку у розробленому додатку, потрібно змінити два параметри:

– значення верхньої межі повинно бути більше за 10000;

– імовірність закриття клітини повинне перевищувати 55.

Оскільки ця карта створюється лише з одного острова, то це означає, що не потрібно видаляти кімнату (або острів) який буде великого розміру, саме тому ми й змінюємо його значення з 500 до 11000. Імовірність закриття комірки теж потрібно підвищити, у цьому прикладі генерації карти змінено значення 45 на 56, тим самим підвищивши шанс закриття клітини. Змінивши параметри на зазначені, отримуємо цільну карту, яка не потребує додаткового прокладання тунелів – це вже повноцінно сформована печера.

3.5 Опис розроблених класів

Для реалізації програми було розроблено такі класи:

– class MapGenerator;

– partial class Form1;

– static class Program.

Більш детально розглянемо розроблені класи.

Клас Program – це головна вхідна точка додатку. Він має метод static void Main() з якого програма й починає працювати. Він містить виклик таких методів:

– Application.EnableVisualStyles() – цей метод включає стилі візуальних елементів для програми;

– Application.SetCompatibleTextRenderingDefault(false) – задає значення за промовчанням для всієї програми для властивості, визначеної UseCompatibleTextRendering для певних елементів керування;

– Application.Run(new Form1()) – запуск стандартного циклу обробки повідомлень програми в поточному потоці і робить вказану форму видимою.

Клас Form1 наслідується від класу Form. Він описує графічний дизайн додатку, логіку та загальні налаштування головної сторінки. Клас Form1 має такі методи:

– public Form1() – це конструктор класу Form1, він визиває метод InitializeComponent();

– private void Form1_Load(object sender, EventArgs e) – відбувається до початкового відображення форми. У цьому методу створюється об’єкт класу MapGenerator. Цей об’єкт записується у змінну, що зберігає у собі його властивості;

– private void pictureBox1_Paint(object sender, PaintEventArgs e) – цей метод використовується для малювання створюваної карти у PictureBox;

– private void lblBuildCaves_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e) – обробка події натискання на кнопку «Побудувати печери». У PictureBox задається висота та ширина, відповідно до

початкових налаштувань, або вони змінюються через зміну властивостей розміру карти. Викликаються методи cavgen.Build(), prop.Refresh() та pictureBox1.Invalidate();

– private void lblConnectCaves_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e) – обробка події натискання на кнопку «Прокласти тунелі». Викликаються методи cavgen.ConnectCaves() та pictureBox1.Invalidate();

– private void lblResetProperties_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e) – обробка події натискання на кнопку «Повернути налаштування». У цьому методу створюється об’єкт класу MapGenerator. Цей об’єкт записується у змінну, що зберігає у собі його властивості;

– protected override void Dispose(bool disposing) – очищує всі використовувані ресурси. Якщо керовані ресурси повинні бути утилізовані то true, в іншому випадку false;

– private void InitializeComponent() – це метод, який автоматично створюється та керується дизайнером Windows Forms, і він визначає все, що знаходиться у формі. Він задає початкові значення, розмір, знаходження, назву, вказує чи відображаються елементи Form1, pictureBox1, t, prop, lblConnectCaves, lblResetProperties, lblBuildCaves.

У класі MapGenerator реалізований головний функціонал та логіка усього додатку. Клас MapGenerator має такі методи:

– public MapGenerator() – конструктор класу MapGenerator. У цьому конструкторі відбувається присвоєння початкових значень для змінних Rnd, Neighbours, Iterations, CloseCellProb, LowerLimit, UpperLimit, MapSize, EmptyNeighbours, EmptyCellNeighbours, CorridorSpace,Corridor_MaxTurns, Corridor_Min, Corridor_Max, BreakOut;

– public int Build() – метод, що викликає методи BuildCaves() та GetCaves() й повертає метод Caves.Count();

– private void BuildCaves() – цей метод створює печери, згладжує їх і заповнює будь-які «дірки» у печерах. У ньому проходить ініціалізація ігрової карти. Створюється масив карти nxm. Відвідується кожна клітинка на карті, і використовуючи випадкову ймовірність, визначається, закривати клітинку, чи ні. Усі значення зберігаються у масиві. Перебирається масив та випадковим чином вибираються клітини. Якщо обрана клітина має більше сусідів відповідно до околиці мура ніж n, то клітина закривається. У іншому випадку, клітина відкривається. Цей крок повторюється і кількість разів;

– private void Cave_GetEdge(List<Point> pCave, ref Point pCavePoint, ref Point pDirection) – метод який знаходить краї печери. Вибирається випадкова точка в печері та записується в pCavePoint. Відбувається виклик методу Direction_Get(pDirection) за допомогую якого отримується випадковий напрямок, так результат записується у змінну pDirection;

– private void GetCaves() – метод, що знаходить усі створені печери, та поміщає їх у загальний список печер. У ньому, у змінну Caves записується новій порожній єекземпляр класу List, створюються структура List<Point> Cave, та локальна змінна cell з типом даних Point. Переглядається кожна клітинка на карті. Змінній cell присвоюється значення нового екземпляру класу Point, у конструктор якого ми передаємо x та y. Якщо клітинка закрита, і ця клітинка не зустрічається в списку печер, то у змінну cave ми записуємо новий екземпляр класу List<Point>, та запускаємо рекурсивний метод LocateCave(cell, Cave). Потім потрібно перевірити, чи співпадає печера із заданим діапазоном властивостей. Якщо співпадає, то перебираємо її завдяки foreach (Point p in Cave), та встановлюємо клітині задане значення Point_Set(p, 0). Якщо не співпадає, то додаємо печеру у колекцію Caves.Add(Cave);

– private void LocateCave(Point pCell, List<Point> pCave) – це рекурсивний метод пошуку клітин, що містять печеру, на основі алгоритму flood fill. За допомогою циклу foreach перебираємо список дійсних сусідніх клітин вибраної клітини за околицею фон Неймана: foreach (Point p in Neighbours_Get(pCell).Where(n => Point_Get(n) > 0)). Якщо змінна p не входить до

листа pCave, до додає цей об’єкт у кінець списку, та визиває початковий метод LocateCave(p, pCave);

– public bool ConnectCaves() – це метод, який намагається з’єднати печери. Він перевіряє, чи кількість створених печер дорівнює нулю, та у випадку коли вони відсутні, виходить з методу. Створюється структура List з типом даних Point що називається currentcave, вона відповідає за теперішню печеру. Створюється структура List з типом даних <List<Point>> з назвою ConnectedCaves, який одразу ініціалізується, вона відповідає за підключені печери. Створюються дві змінні класу Point, cor_point що відповідає за місцезнаходження тунелю, та cor_direction яка буде вказувати у якому напрямку він буде простягатися. Відбувається ініціалізація potentialcorridor структури List з типом даних Point яка відповідає за потенційні коридори. Присвоюється значення 0 змінній breakoutctr, що відповідає за контроль завантаженості печер у теперішній момент часу. Створюється екземпляр типу List<Point>, та записується у Corridors, що відповідає за збереження утворених тунелів. Випадковим чином вибирається печера, та записується у змінну currentcave. Після цього, вона додається у ConnectedCaves, та прибирається з Caves. Далі створюється початковий конструктор. Якщо коридорів нема, будується печера, викликається метод Cave_GetEdge у який передаються потрібні значення currentcave, ref cor_point та ref cor_direction. Якщо тунель існує, тоді випадковим чином вибирається з чого починати створення точки, з коридору чи печери, тобто якщо випадкове число більше за 50, то змінній currentcave присвоюється ConnectedCaves[rnd.Next(0, ConnectedCaves.Count())], та визивається метод Cave_GetEdge у який передаються параметри currentcave, ref cor_point, ref cor_direction, а якщо випадкове число менше за 50, то змінній currentcave присвоюється null, та викликається метод Corridor_GetEdge, у який передаються параметри ref cor_point, ref cor_direction. Використовуючи точки, які було визначено напередодні, відбувається спроба побудувати коридор від них, у змінну potentialcorridor записується результат виклику методу Corridor_Attempt з параметрами cor_point, cor_direction, true. Якщо змінна potentialcorridor не дорівнює null, печеру було знайдено, й виконуються наступні кроки. Потрібно

обстежити всі існуючи печери, й у випадку, коли остання точка коридорів знаходиться у печері, та ми побудували коридор або змінна currentcave не дорівнює теперішній печері Caves, то відбувається наступне:

1. Остання точка тунелю вторгається в печеру, тому видаляємо її;
2. Додається тунель, який зараз знаходиться у potentialcorridor до колекції коридорів Corridors за допомогою методу AddRange();
3. За допомогою циклу, відтворюється тунель на карті;
4. Досягнута печера додається до підключеного списку ConnectedCaves;
5. Видаляється розглянута печера зі списку Caves;
6. Обстеження печер припиняється.

Інкрементуємо змінну breakoutctr, та у випадку коли вона більша за значення змінної BreakOut, то метод повертає false. Усі дії від початкового конструктора, й до перевірки breakoutctr будуть повторюватися доки кількість печер у Caves буде більше за нуль. Після цього у Caves викликається метод AddRange() у який передається ConnectedCaves. У ConnectedCaves викликаємо метод Clear(), який очищує прокладені тунелі. Відбувається вихід з методу, й повертається true;

– private void Corridor_GetEdge(ref Point pLocation, ref Point pDirection) – це метод, за допомогою якого випадково отримується точка в існуючому тунелі. Створюється екземпляр типу List<Point>, та записується у validdirections. У pLocation записується Corridors[rnd.Next(1, Corridors.Count - 1)], завдяки цим модифікаторам уникаємо ситуації коли обирається перша або остання клітина. Відбувається спроба знайти всі порожні точки на карті навколо визначеної околиці. Використовуються Directions для зміщення випадково обраної точки. Використовується цикл у якому перебираються об’єкти типу Point в Directions, у ньому викликається метод Point_Check який перевіряє чи клітина закрита, у який передається new Point(pLocation.X + p.X, pLocation.Y + p.Y), та метод Point_Get який отримує значення даної клітинки, у який передається new Point(pLocation.X + p.X, pLocation.Y + p.Y). Якщо метод Point_Check повертає true, та значення яке повертає метод Point_Get дорівнює нулю, то у validdirections додається теперішній

екземпляр класу Point. Ці дії будуть повторюватися поки кількість об’єктів у validdirections дорівнює нулю. Потім змінній pDirection присвоюється validdirections[rnd.Next(0, validdirections.Count)], й змінна pLocation викликає метод Offset(), у який передає pDirection;

– private List<Point> Corridor_Attempt(Point pStart, Point pDirection, bool pPreventBackTracking) – це метод який намагається прокласти тунелі. Створюється екземпляр типу List<Point>, та записується у lPotentialCorridor. У lPotentialCorridor викликається метод Add() у який передається об’єкт pStart з типом даних Point. Створюється змінна corridorlength з цілочисельним типом даних, яка потрібна для зберігання інформації про довжину тунелю. Створюється екземпляр класу Point з назвою startdirection, у конструктор якого передається параметри pDirection.X та pDirection.Y. Відбувається створення та ініціалізація змінної pTurns з цілочисельним типом даних у яку передається значення змінної Corridor_MaxTurns, яка визначає максимальну кількість поворотів що можуть зробити прокладені тунелі. Доки pTurns більше, або дорівнює нулю, зменшуємо значення цієї змінної на одиницю, у змінну corridorlength , що відповідає за довжину тунелю, записуємо випадкове значення між максимальним та мінімальним значенням розміру тунелю завдяки виклику метода rnd.Next, та передачі у нього значень Corridor_Min та Corridor_Max. Далі прокладаємо сам тунель. Доки змінна corridorlength більша за нуль, зменшуємо її значення на одиницю, вибираємо клітинку pStart та завдяки методу Offset просуваємо її, передаючи як аргумент у метод pDirection. Перевіряємо, якщо знайдена точка існує Point_Check(pStart) та її значення Point_Get(pStart) дорівнює одиниці, то у змінну яка відповідає за потенційний коридор lPotentialCorridor, додаємо цю клітину pStart, та виходимо з методу повертаючи значення lPotentialCorridor. Якщо ж обрана клітина не існує, то метод повертає null, якщо ж метод Corridor_PointTest у який ми передаємо значення pStart та pDirection повертає false, то наш метод також повертає null. у змінну яка відповідає за потенційний коридор lPotentialCorridor, додаємо pStart. На цьому ітерація спроби побудови коридору закінчується. Якщо pTurns більше за одиницю, то перевіряємо значення

змінної pPreventBackTracking, та якщо воно ложе, то у pDirection записується значення, що отримується при виклику мктоду Direction_Get та передачі у нього як аргумент pDirection. Якщо ж значення змінної pPreventBackTracking істина, то у pDirection записується значення, що отримується при виклику мктоду Direction_Get та передачі у нього як аргументи pDirection та startdirection. Відбувається вихід з методу, та повертається null;

– private bool Corridor_PointTest(Point pPoint, Point pDirection) – це метод який перевіряє, чи порожні клітинки по обидві сторони від обраної клітини. За допомогою цикле foreach перебираємо єлементи у структурі Enumerable.Range(-CorridorSpace, 2 * CorridorSpace + 1).ToList(). Коли значення CorridorSpace перевищено, припиняються спроби з’єднати печери, значення цій змінній надаються як по замовчуванням, так їх також може змінити користувач завдяки інтерфейсу програми. Якщо значення pDirection.X дорівнює 0, тобто розглядається тільки північ та південь, то перевіряється чи задана точка дійсна передаючи у метод Point_Check новий об’єкт new Point(pPoint.X + r, pPoint.Y)), та якщо метод повертає true, викликаємо метод Point_Get у який як аргумент передаємо Point_Get(new Point(pPoint.X + r, pPoint.Y), і якщо метод повертає значення що не дорівнює нулю, то наш метод private bool Corridor_PointTest(Point pPoint, Point pDirection) повертає значення false. Якщо ж pDirection.Y дорівнює нулю, тобто розглядаємо тільки захід та схід, перевіряється чи задана точка дійсна передаючи у метод Point_Check новий об’єкт new Point(pPoint.X, pPoint.Y + r)), та якщо метод повертає true, викликаємо метод Point_Get у який як аргумент передаємо Point_Get(new Point(pPoint.X, pPoint.Y + r), і якщо метод повертає значення що не дорівнює нулю, то наш метод private bool Corridor_PointTest(Point pPoint, Point pDirection) повертає значення false. Відбувається вихід з методу, та повертається true;

– private List<Point> Neighbours_Get(Point p) – це метод, який повертає список дійсних сусідніх клітинок наданої точки, використовуючи лише північ, південь, схід і захід, тобто за околицею фон Неймана;

– private List<Point> Neighbours_Get1(Point p) – це метод, який повертає список дійсних сусідніх клітинок наданої точки, відповідно до околиці Мура;

– private Point Direction_Get(Point p) – це метод, за допомогою якого отримується випадковий напрямок, за умови, що він не дорівнює протилежному. Створюється об’єкт класу Point, що відповідатиме за напрямок, та має назву newdir. Створеному об’єкту присвоюється значення Directions[rnd.Next(0, Directions.Count())] до тих пір, доки newdir.X не дорівнює -p.X і водночас newdir.Y не дорівнює -p.Y. Відбувається вихід з методу, та повертається newdir;

– private Point Direction_Get(Point pDir, Point pDirExclude) – це метод, за допомогою якого отримуємо випадковий напрямок, за винятком наданих напрямків і протилежного від вже проведеного напрямку, для того щоб запобігти зворотному простяганню тунелю. Параметр pDirExclude є першим напрямком, вибраним для тунелю, і, щоб запобігти його використанню, треба запобігати поверненню тунелю до самого себе. Створюється обєкт класу Point та називається NewDir. У нього записується значення Directions[rnd.Next(0, Directions.Count())], доки об’єкт що повертає виклик методу Direction_Reverse з передачею у нього як аргумент NewDir дорівнює pDir, або дорівнює pDirExclude. Відбувається вихід з методу, та повертається NewDir;

– private Point Direction_Reverse(Point pDir) – це метод, який повертає інвертований напрям;

– private bool Point_Check(Point p) – це метод, який перевіряє чи передана у метод клітина існує, та повертає true якщо існує, та false якщо клітина не існує;

– private void Point_Set(Point p, int val) – метод, який встановлює клітинку карти, на вказане значення;

– private int Point_Get(Point p) – це метод, за допомогою якого можна отримати значення наданої точки.

3.6 Проведення моделювання та аналіз результатів

Для отримання даних про відповідність карт, розроблених за допомогою вдосконалених методів та алгоритмів процедурної генерації, які було реалізовано у створеному програмному забезпеченні, якісним вимогам, проведемо ряд експериментів із генерацією карт завдяки розробленого додатку, та зміні відповідних характеристик для отримання та порівняння різних результатів.

Оскільки автоматизована генерація випадкова, та утворені карти навіть за однаковими налаштуваннями відрізняються, результати наведених графіків беруть середнє арифметичне з 10 створених карт.

Початкові налаштування для генерації ігрової карти: p = 45, i = 50000, M[100, 100], n = 4, h1 = 500, h2 = 16, z = 3, f = 4, t = 2, tmax = 5, tmin = 2, tu = 10, s = 100000.

При використанні виду карт типу «Лабіринт» деякі налаштування змінюються, а саме: M[170, 150], h1 = 300, h3 = 8, tmax = 17, tmin = 3, tu = 18.

При використанні виду карт типу «Багато маленьких крапель-печер», деякі налаштування змінюються, а саме: M[170, 150], h1 = 300, h3 = 8, tmax = 8.

При використанні виду карт типу «Суцільна печера», деякі налаштування змінюються, а саме: p = 56, M[120, 120], h1 = 10000.

Результат порівняння згенерованих карт різного патерну за такими ознаками як кількість печер, не поєднаних з іншими, кількість тунелів, кількість печер та максимальна кількість входів та виходів печери можна побачити на рисунку 3.10.

Відповідно до результатів, можна з впевненістю сказати що програмне забезпечення що генерує ігрові карти розроблено завдяки поєднанню покращених методів клітинного автомату та алгоритмів тунелювання працює відповідно до очікувань. Усі карти що було згенеровано, окрім створених за патерном «Суцільна печера», не має печер, не поєднаних з іншими. Суцільна печера, як можна зрозуміти з назви, й не повинна поєднуватися з іншими, бо вона одна заповнює простір усієї карти, така її задумка.

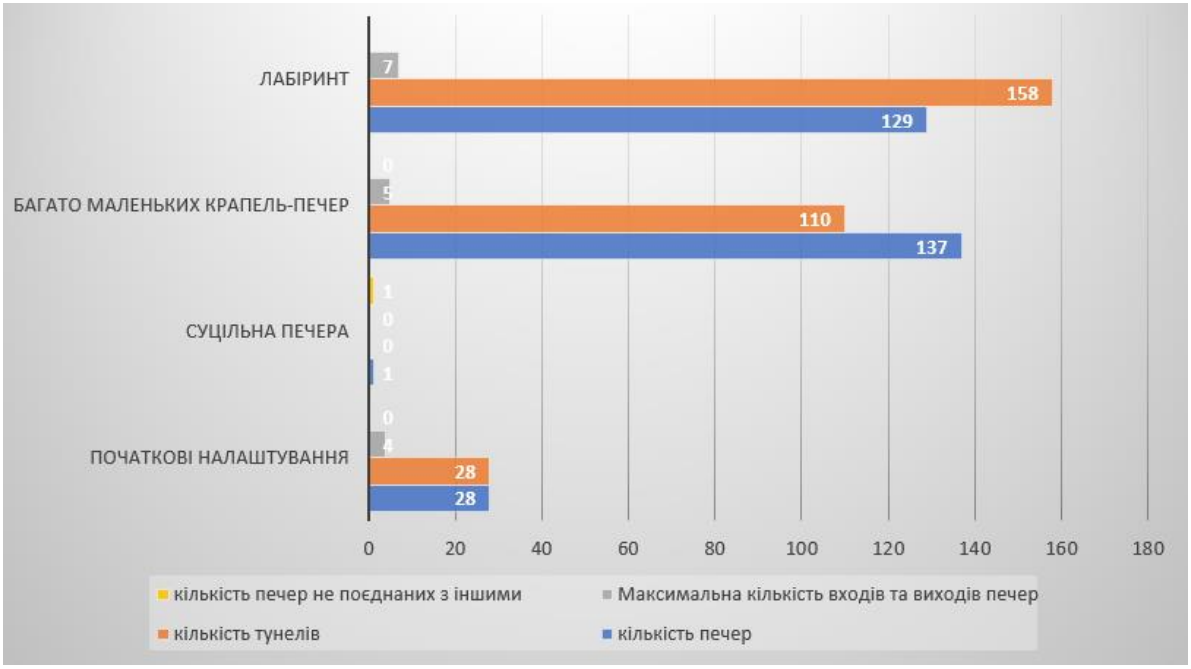


Рисунок 3.10 – Порівняння середнього арифметичного значення кількості тунелів, печер, печер не поєднаних з іншими та входів виходів 10 згенерованих карт з початковими налаштуваннями, та використовуючи патерни «Лабіринт», «Багато маленьких крапель-печер» та «Суцільна печера»

Кількість печер залежить від зміни розміру карти та від зміни ймовірності закриття клітинки, на рисунку 3.11 можна побачити графік що показує зміну кількості печер відповідно до зміни імовірності закриття клітинок, при початкових налаштуваннях.

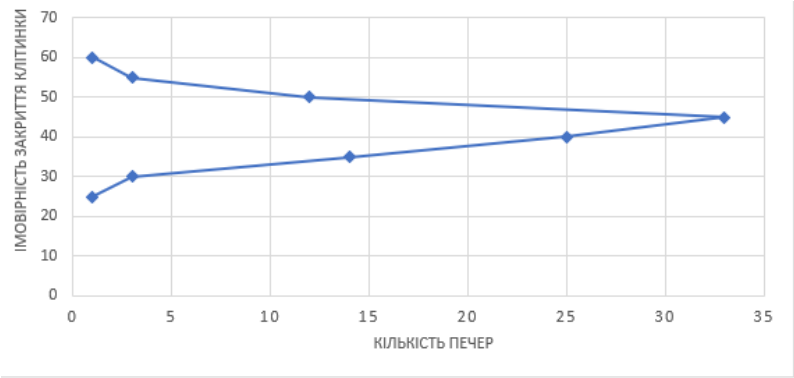


Рисунок 3.11 – Графік залежності кількості генеруємих печер у залежності від імовірності закриття клітинки

Кількість прокладених тунелів напряму залежить від кількості утворених печер, графік їх залежності при першому прокладанні тунелів та при початкових налаштуваннях можна побачити на рисунку 3.12.

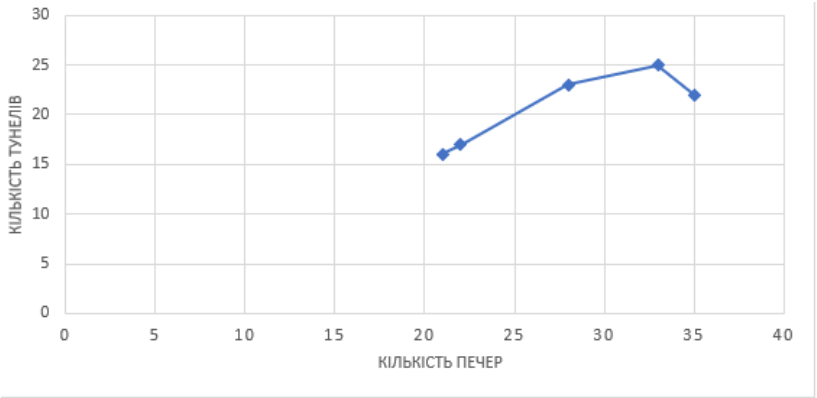


Рисунок 3.12 – Графік залежності кількості тунелів при першому прокладанні та утворених печер

Завдяки створеної можливості прокладання тунелів неодноразово, розроблений додаток має змогу прокладати тунелі декілька разів, якщо кількість прокладених тунелів недостатня для користувача. Графік кількості тунелів після п’яти спроб прокласти тунелі між печерами зображено на рисунку 3.13.

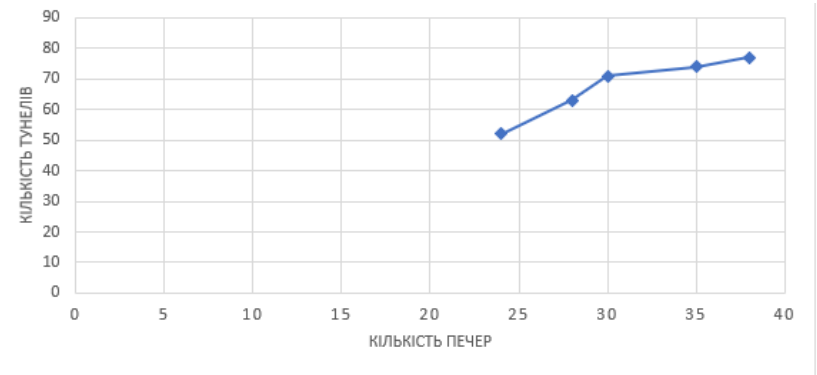


Рисунок 3.13 – Графік залежності кількості тунелів після п’яти прокладань та утворених печер

Для більш детального аналізу генеруємих карт, трохи змінимо деякі початкові налаштування. У h, що відповідає за верхню межу, тобто максимальний

розмір печери, записується число 10000, та поступово змінюючи p, що відповідає за імовірність закриття, з 40 до 60 отримуємо залежність кількості печер від імовірності закриття клітинки при зміненому максимальному розміру печери, результат чого відображено у графіку що зображений на рисунку 3.14.

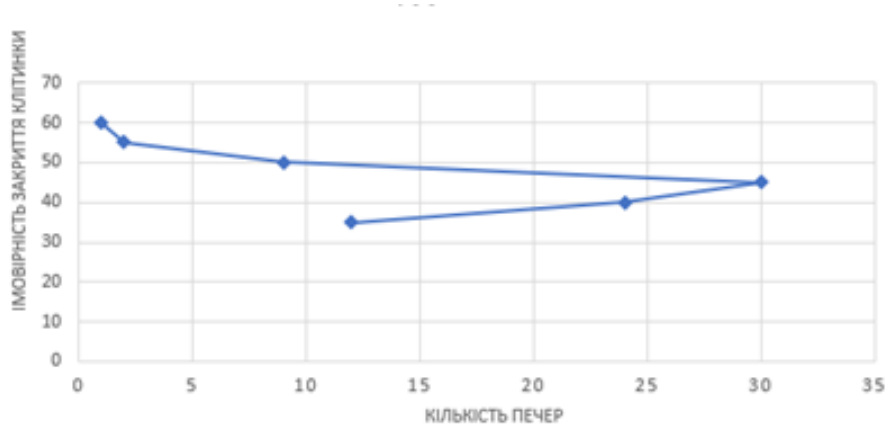


Рисунок 3.14 – Графік залежності кількості тунелів та утворених печер при поступовій зміні значення p, та установленню h значення 10000

Хоча на перший погляд графік зображений на рисунку 3.14 схожий на графік на рисунку 3.11, але розмір наприклад однієї печери при p = 60 у першому випадку буде великим, як при використанні патерну «Суцільна печера», а у другому це буде просто маленька крапля-печера.

Кількість печер також залежить від кількості ітерацій. Змінивши початкові налаштування h1 на 50, h2 на 4, та поступово змінюючи кількість ітерацій з 1000 до 35000 результат буде таким, як зображено на рисунку 3.15.

Для остаточного підбиття підсумків розробленого додатку який реалізує покращенні методи та алгоритми автоматизованої генерації ігрової карти для гри жанру Roguelike, було створено чотири карти різних патернів, та узяті перші їх генерації за зразки.

Першою розглянутою картою була карта згенерована за початковими налаштуваннями, вона зображена на рисунку 3.16.

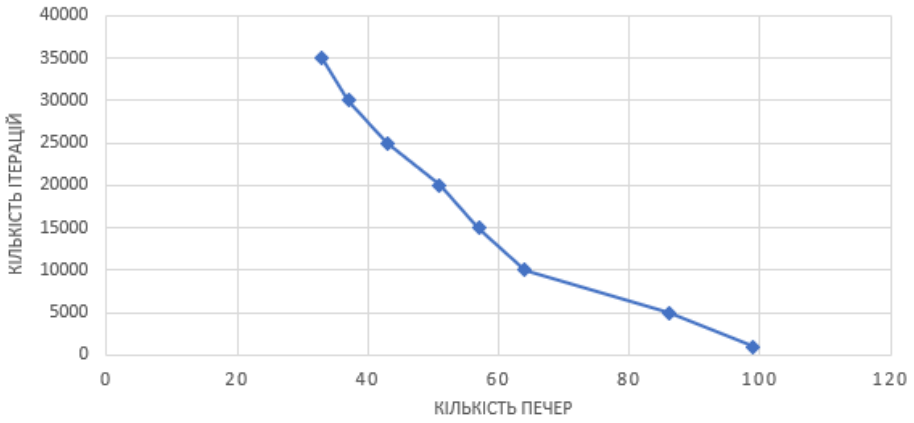


Рисунок 3.15 – Графік залежності кількості утворених печер від кількості ітерацій, при зміні h1 на 50, h2 на 4

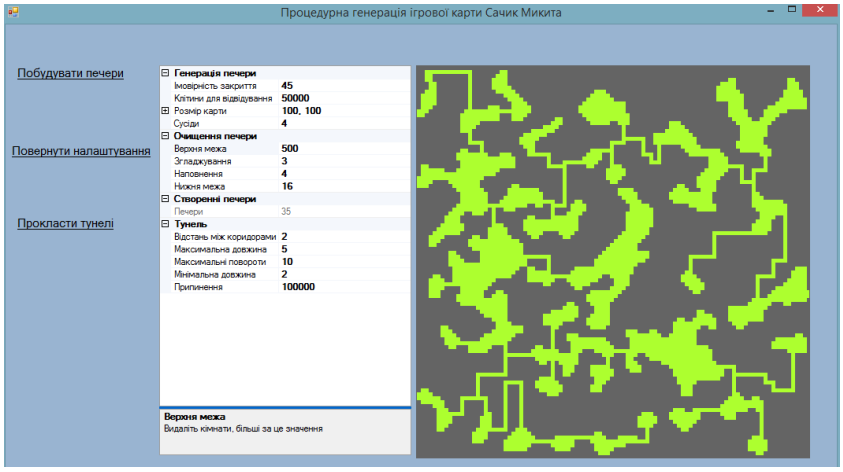


Рисунок 3.16 – Ігрова карта згенерована використовуючи початкові налаштування

У подальшому, згенеровані карти створювалися зі зміною параметрів, відповідно до їх патернів. Було створено карти з патернами «Лабіринт», що зображена на рисунку 3.17, «Суцільна печера», зображена на рисунку 3.18, та «Багато маленьких крапель-печер», що зображена на рисунку 3.19.

Ці патерни були використані для заповнення таблиці для оцінювання згенерованих карт за критеріями якості які було визначено. Результати оцінювання якості створених карт можна побачити на рисунку 3.20.

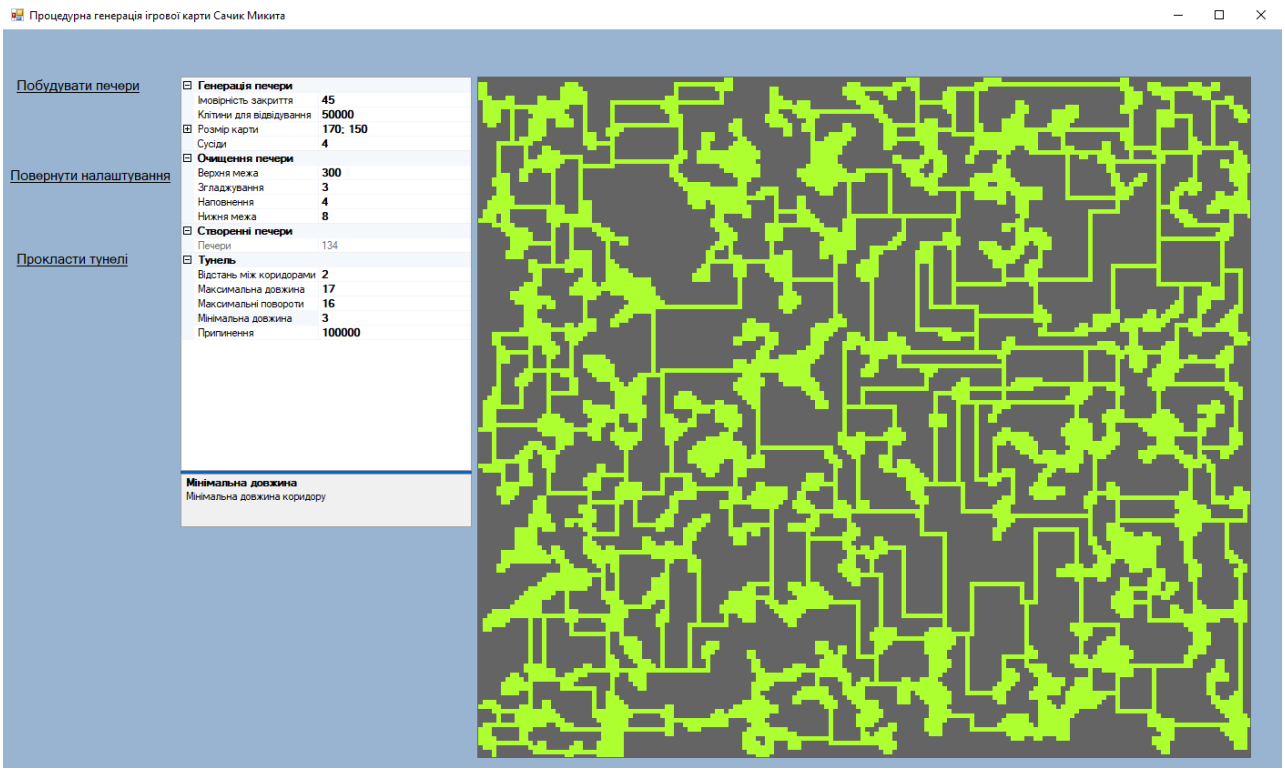


Рисунок 3.17 – Ігрова карта згенерована використовуючи патерн «Лабіринт»

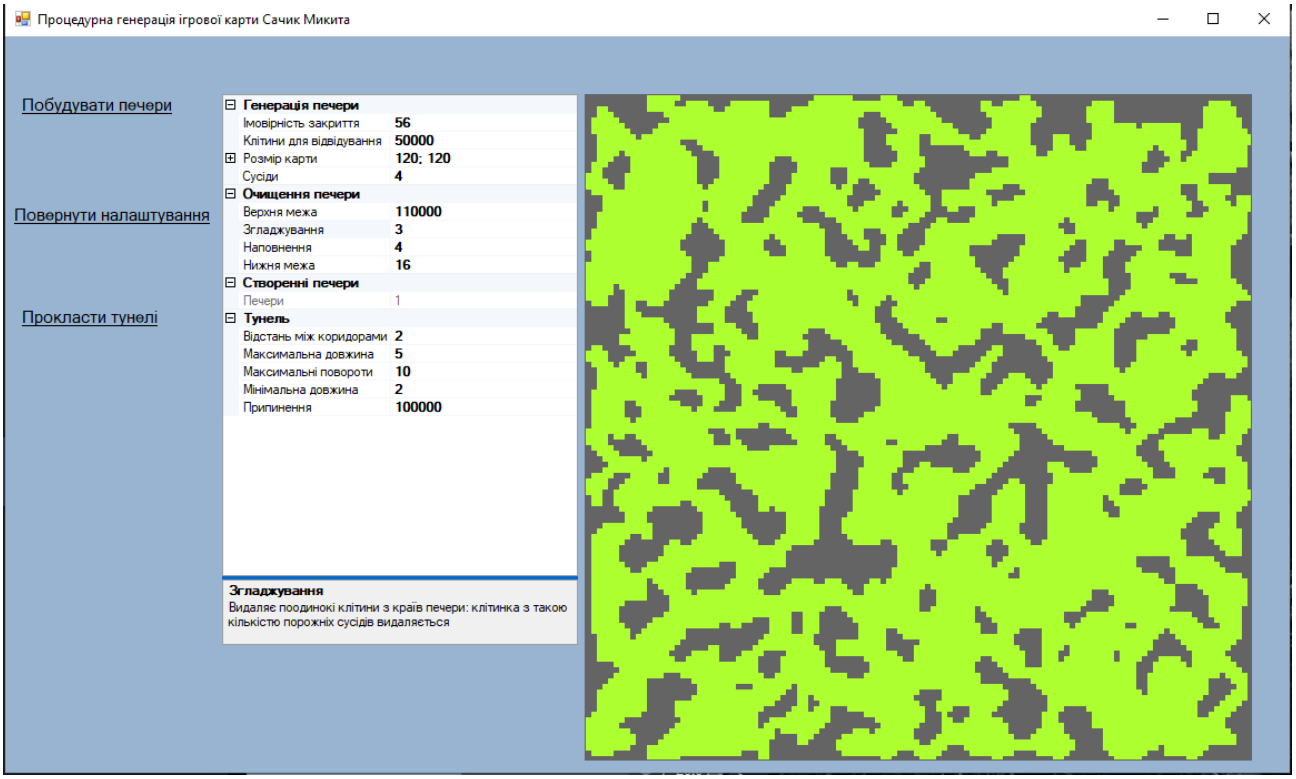


Рисунок 3.18 – Ігрова карта згенерована використовуючи патерн «Суцільна печера»

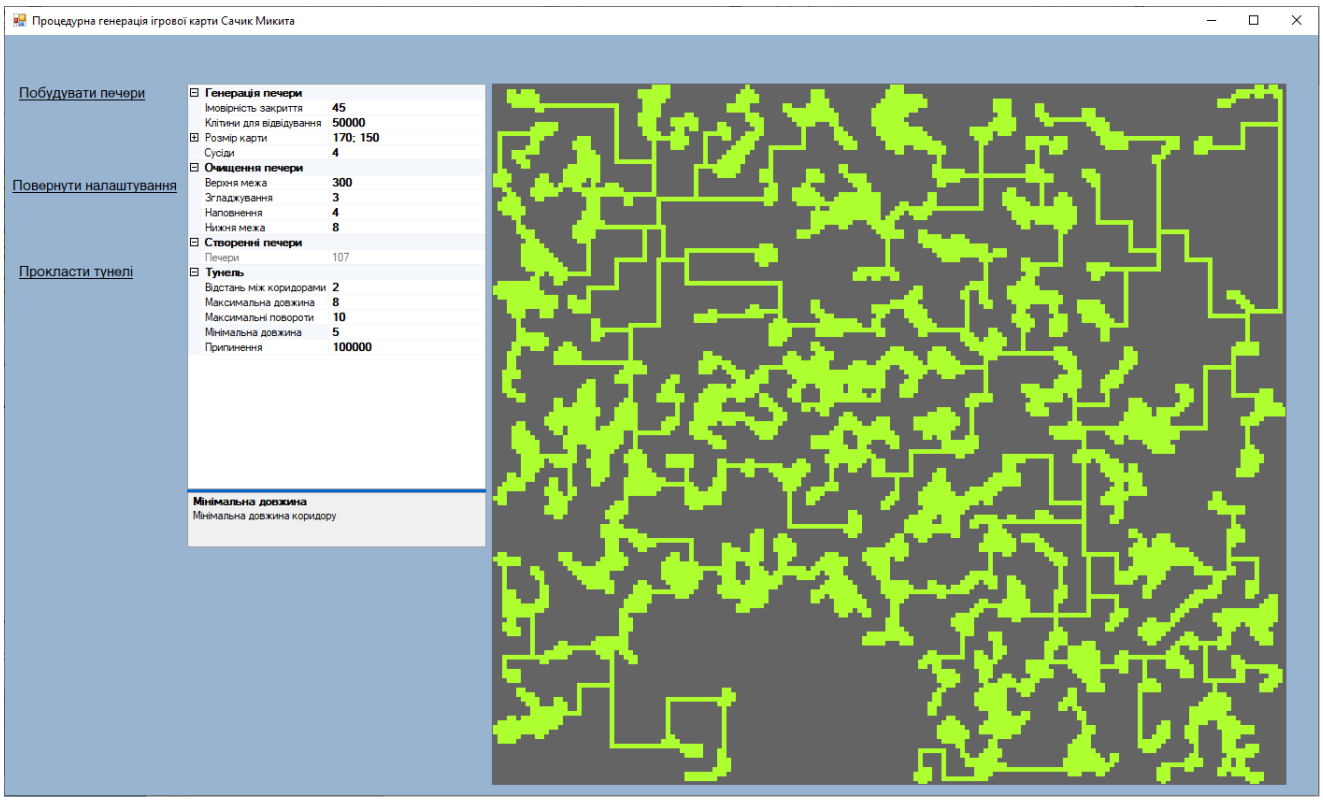


Рисунок 3.19 – Ігрова карта згенерована використовуючи патерн «Багато маленьких крапель-печер»

Критерії якості	Початкові налаштування	Лабіринт	Суцільна печера	Багато маленьких крапель-печер
Кількість печер	35	134	1	107
Розмір печер	Від 16 до 500, можна налаштувати	Від 8 до 300, можна налаштувати	Від 16 до 110000, можна налаштувати	Від 8 до 300, можна налаштувати
Згладжування печер	3, можна налаштувати	3, можна налаштувати	3, можна налаштувати	3, можна налаштувати
Кількість тунелів	23	67	0	43
Довжина тунелів	Від 2 до 5, можна налаштувати	Від 3 до 17, можна налаштувати	Від 2 до 5, але не використовуються	Від 5 до 8, можна налаштувати
Прямолінійність тунелів	Кількість поворотів 10, можна налаштувати	Кількість поворотів 16, можна налаштувати	Кількість поворотів може бути 10, але не використовується	Кількість поворотів 10, можна налаштувати
Кількість входів та виходів печер	Мінімальна 1, максимальна 5	Мінімальна 1, максимальна 6	0	Мінімальна 1, максимальна 4
Розміри карти	100 на 100, можна налаштувати	170 на 150, можна налаштувати	120 на 120, можна налаштувати	170 на 150, можна налаштувати
Печери, не поєднанні тунелями з іншими печерами	0	0	1	0

Рисунок 3.20 – Результати оцінювання згенерованих карт за критеріями якості

Беручи до уваги результати, отримані на рисунку 3.19, можна сказати що критерії якості створення ігрової карти були успішно реалізовано. Більш того,

додаток що реалізує покращені методи, створює унікальні карти що цілком підходять для ігор жанру Roguelike, та користувач має змогу самостійно налаштовувати створення карт, завдяки можливості зміни налаштувань.

ВИСНОВКИ

У результаті виконання магістерської роботи було розглянуто головні підходи для реалізації автоматизованої генерації ігрової карти для ігор жанру Roguelike та виявлені ключові елементи ігрової карти, що притаманні іграм цього жанру.

Розглянуто такі алгоритми та методи процедурної генерації, як просте кімнатне розміщення, бінарний розподіл простору, прогулянка п'яниці, дифузійна обмежена агрегація, діаграми Вороного, шум Перліна і симплексний шум, карти Дейкстри, алгоритми тунелювання, заливання, та клітинні автомати. Виконано аналіз особливостей та недоліків генерації ігрової карти на основі цих методів,

Розроблена модель карти на основі клітинного автомату, що являє собою множину клітин, які поступово змінюють свій стан в залежності від заданих правил. Модель є модифікацією «Гри в життя» Конвея. Розроблено метод автоматизованої генерації ігрової карти для гри в жанрі roguelike на основі клітинного автомату та у поєднанні з алгоритмами заливання для пошуку клітин що є складовими печери, та алгоритмами тунелювання за допомогою яких відбувається поєднання утворених печер між собою.

Визначено критерії якості генерованих печер, за допомогою яких можна визначити, чи згенерована ігрова карта є грабельною, доцільною з точки зору ігрового процесу, чи реалізує закладений початковий функціонал.

Розроблено програмне забезпечення, що реалізує автоматизовану генерацію ігрової карти на основі клітинного автомату для ігор жанру Roguelike. Визначено параметри для створення ігрової карти, за допомогою яких можна контролювати процес генерації ігрової карти, створювати різні патерни, варіації відповідно до критеріїв якості, які було визначено. Вони впливають на генерацію самих печер, їх налаштування, очищення та прокладання тунелів.

Проведено моделювання карт з різними параметрами налаштування. Аналіз характеристик створених карт демонструє їх відповідність заданим показникам якості. Є можливість повністю налаштувати генерацію карти під потреби

користувача, що робить карти унікальними. Не зважаючи на випадковий характер автоматизованої генерації ігрової карти, завдяки модифікованому методу на основі клітинних автоматів, досягнена можливість контролювати її варіативність.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Joris Dormans. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In Proceedings of the 2010 Workshop on Procedural Content Generation in Games (Monterey, California) (PCGames ’10). Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages.

2. Joris Dormans. 2011. Level Design as Model Transformation: A Strategy for Automated Content Generation. In Proceedings of the 2nd International Workshop on Procedural Content Generation in Games (Bordeaux, France) (PCGames ’11). Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages.

3. G. Smith, E. Gan, A. Othenin-Girard, and J. Whitehead, “PCG-based game design: enabling new play experiences through procedural content generation,” in Second International Workshop on Procedural Content Generation in Games (PCG 2011), Bordeaux, France, June 28, 2011 2011.

4. Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Search-Based Procedural Generation of Maze-Like Levels. IEEE Transactions on Computational Intelligence and AI in Games 3, 3 (Sep. 2011), 260–273.

5. Vojtech Cerny and Filip Dechterenko. 2015. Rogue-Like Games as a Playground for Artificial Intelligence – Evolutionary Approach. In Entertainment Computing - ICEC 2015, Konstantinos Chorianopoulos, Monica Divitini, Jannicke Baalsrud Hauge, Letizia Jaccheri, and Rainer Malaka (Eds.). Springer International Publishing, Cham, 261–271.

6. D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis. 2019. Procedural Content Generation through Quality Diversity. In 2019 IEEE Conference on Games (CoG). 1–8.v

7. E. Hastings, R. Guha, and K. Stanley, “Automatic content generation in the Galactic Arms Race video game,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 1, no. 4, pp. 245–263, 2009

8. L. Johnson, G. N. Yannakakis, and J. Togelius, “Cellular automata for real-time generation of infinite cave levels,” in PCG ’10: Proceedingsof the 2010 Workshop

on Procedural Content Generation in Games. New York, NY, USA: ACM, 2010, pp. 10:1–10:4

9. J. Dormans, “Level design as model transformation: a strategy for automated content generation,” in Proceedings of the 2nd InternationalWorkshop on Procedural Content Generation in Games. New York, NY, USA: ACM, 2011, pp. 2:1–2:8.

10. K. Hartsook, A. Zook, S. Das, and M. Riedl, “Toward supporting stories with procedurally generated game worlds,” in IEEE Conferenceon Computational Intelligence and Games (CIG), September 2011, pp. 297 –304.

11. Rosin, P. L., Adamatzky, A., & Xianfang, Sun. (2014). Cellular Automata in Image Processing and Geometry. Springer

12. Hoekstra A.G., Kroc J., Sloot P. (2010). Simulating complex systems by cellular automata. London: Springer, 235 p.

13. Mohammed, J., & Deepak, R. N. (2010). An Effcient Edge Detection Technique by Two Dimensional Rectangular Cellular Automata.

14. Wuensche A. Classifying Cellular Automata Automatically / Wuensche A. COMPLEXITY. – 1999. – №4. – pp. 47–66.

15. Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. PCGRL: Procedural Content Generation via Reinforcement Learning. arXiv:cs.LG/2001.09212

16. Penelope Sweetser and Janet Wiles. 2005. Combining Influence Maps and Cellular Automata for Reactive Game Agents. In Intelligent Data Engineering and Automated Learning - IDEAL 2005, Marcus Gallagher, James P. Hogan, and Frederic Maire (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 524–531.

ДОДАТОК

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

МАГІСТЕРСЬКА РОБОТА

«РОЗРОБКА МЕТОДУ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ДЛЯ ГРИ В ЖАНРІ ROGUELIKE НА ОСНОВІ КЛІТИННОГО АВТОМАТУ»

Виконав: студент групи ПДМ-61, Сачик Микита Олександрович

Керівник: к.т.н., доц., доцент кафедри ІПЗ Золотухіна Оксана Анатоліївна

Київ - 2022

МЕТА, ОБ'ЄКТ, ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення процесу автоматизованого створення ігрової карти для гри в жанрі Roguelike за рахунок застосування клітинного автомату.

Об'єкт дослідження: автоматизована генерація ігрової карти для гри в жанрі Roguelike.

Предмет дослідження: метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату.

КЛЮЧОВІ ЕЛЕМЕНТИ ІГРОВОЇ КАРТИ В ІГРАХ ЖАНРУ ROGUELIKE

Крапля – обмежена зона певної конфігурації по якій може пересуватися гравець. Краплі можуть бути у формі печер та кімнат.

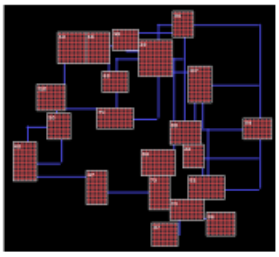
Печера – крапля, що має неправильну, довільну форму.

Кімната – крапля, що має форму багатокутника.

Тунель – елемент карти, який об'єднує між собою краплі. Тунелі поєднують як одну краплю з іншою, так і багато крапель разом. Тунелі можуть мати довільні розгалуження.

Карта з печерами

Карта з кімнатами



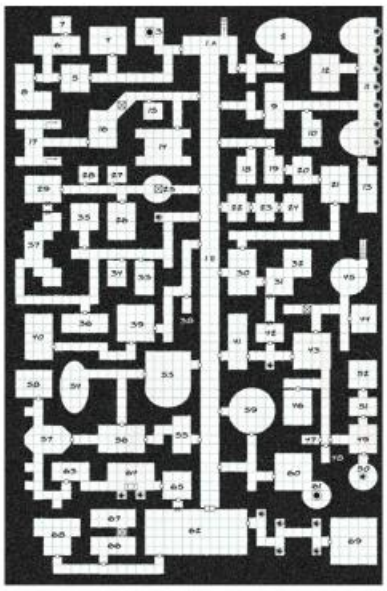
Карта з прокладеними між кімнатами тунелями

АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ

Модель	Особливості генерації крапель та тунелів	Недоліки
Просте кімнатне розміщення	Проста, поширена, корисна модель для векторів, які описують об'єкти на єдиній сітці, як-от шахова дошка чи міські квартали.	Не підходить для створення крапель-печер.
Бінарний розподіл простору	Можна гарантувати створення більш рівномірно розподілених крапель-кімнат та забезпечити їхнє з'єднання. Використовується для виявлення зіткнень у 3D-відеоіграх, робототехніці, обробці складних просторових сцен та трасуванні променів.	Найбільш очевидна структура даних для проходження різних рівнів розділів, її моделювання займає більше часу у порівнянні з іншими методами, не вирішує проблему визначення видимої поверхні.
Тунелювання	Створюють у суцільній «землі» коридори та кімнати, підходить для створення печероподібних карт зі змішуванням відкритих та замкнутих просторів.	В кожній кімнаті є лише вхід і вихід, а це не завжди цікаво з точки зору геймплею. При генерації часто виходять марні чи зайві шляхи.
Прогулянка п'яниці	Створює різноманітні комбінації карапель-печер, починається з повністю заповненого рівня, а потім видаляє його одну клітинку за раз. Генерує печерні рівні, що завжди будуть з'єднані, тож не потребує додаткового прокладання тунелів.	Вимагає динамічної зміни розміру карти, не дуже добре підходити для створення рівнів для відеоігор. Метод ненадійний, бо занадто випадковий.
Клітинний автомат	**Найкраще підходить для моделювання природніх середовищ, дає змогу легко підстроювати правила за необхідністю. Чим більше світ, то вище має бути варіативність областей на дослідження. Ідеально підходить для генерації великих ділянок.**	**Неструктурований, хаотичний. Після генерування карти інколи потрібно самостійно забезпечити з'єднання, тому що деякі алгоритми мають велику ймовірність створення розділених областей.**

КРИТЕРІЇ ЯКОСТІ ІГРОВОЇ КАРТІ ЖАНРУ ROGUELIKE

- Кількість печер.
- Розмір печер.
- Згладжування печер.
- Кількість тунелів.
- Довжина тунелів.
- Прямолінійність тунелів.
- Кількість входів та виходів печер.
- Розміри карти.
- Відсутність печер, не поєднаних тунелями з іншими печерами.



МОДЕЛЬ КАРТИ НА ОСНОВІ КЛІТИННОГО АВТОМАТУ

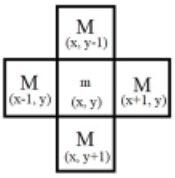
Математична модель карти – множина клітин M, розміщених у вигляді решітки, де фрагменти решітки – клітини $m_{(x,y)}$:

$$m_{(x,y)} \in M.$$

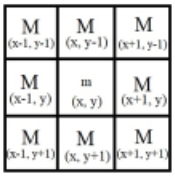
Множина клітин M карти, задається сукупністю множин трьох типів, що не перетинаються:

$$M = M_p \cup M_e \cup M_t,$$
$$M_p \cap M_e \cap M_t = \emptyset,$$

Mp – множина клітин що утворюють печери.
Me – множина порожніх клітин.
Mt – множина клітин що утворюють тунелі.



Окіл фон Неймана



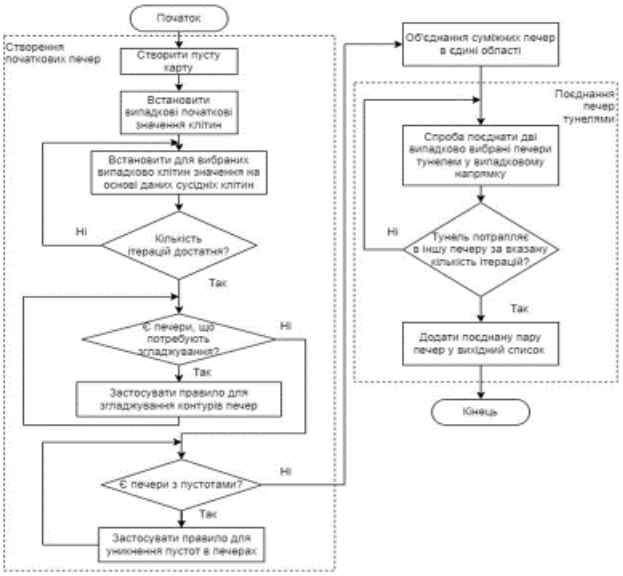
Окіл Мура

Правило для генерації початкових печер:
1. Якщо клітина помічена, як печера, то її стан не змінюється.
2. Інакше для клітини генерується випадкове значення p - ймовірності того, що клітина буде печерою. Якщо p більше заданого порогового значення, то клітина помічається, як печера.

Правило для згладжування печер:
1. Якщо обрана клітина має 3 або більше сусіда-печери відповідно до околу Мура, то клітина помічається, як печера.
2. Інакше, клітина помічається як пуста.

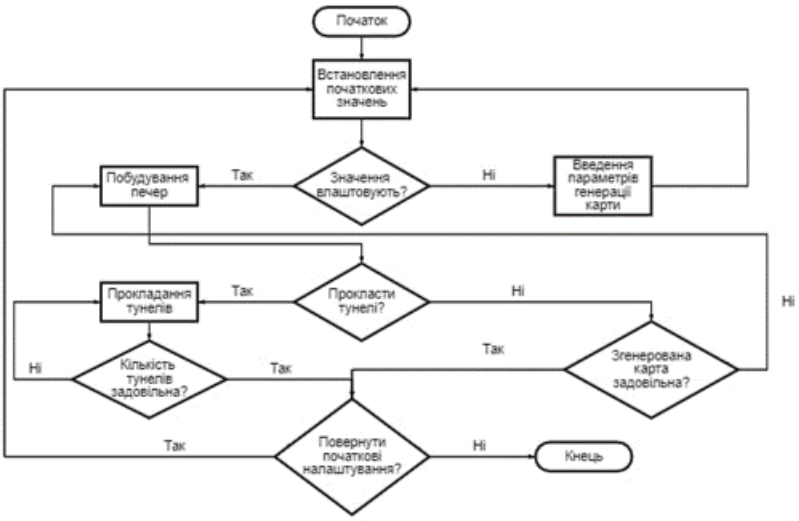
Правило для уникнення пустот в печерах:
Якщо клітина має 4 сусіда-печери відповідно до околу Мура, то клітина помічається, як печера.

Метод автоматизованої генерації ігрової карти для гри в жанрі Roguelike на основі клітинного автомату



7

СХЕМА ФУНКЦІОНУВАННЯ ДОДАТКУ ДЛЯ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ



8

ПАРАМЕТРИ ДЛЯ СТВОРЕННЯ ІГРОВОЇ КАРТИ

Генерація печери

Імовірність закриття (p) – це значення визначає ймовірність закриття комірки яку було відвідано.
Клітини для відвідування (i) – кількість клітин, які потрібно відвідати під час процесу генерації.
Розмір карти (M) – розмір карти в клітинах.
Сусіди (n) – коли це значення дорівнює або перевищує кількість сусідів, змінюється стан комірки.

Очищення печери

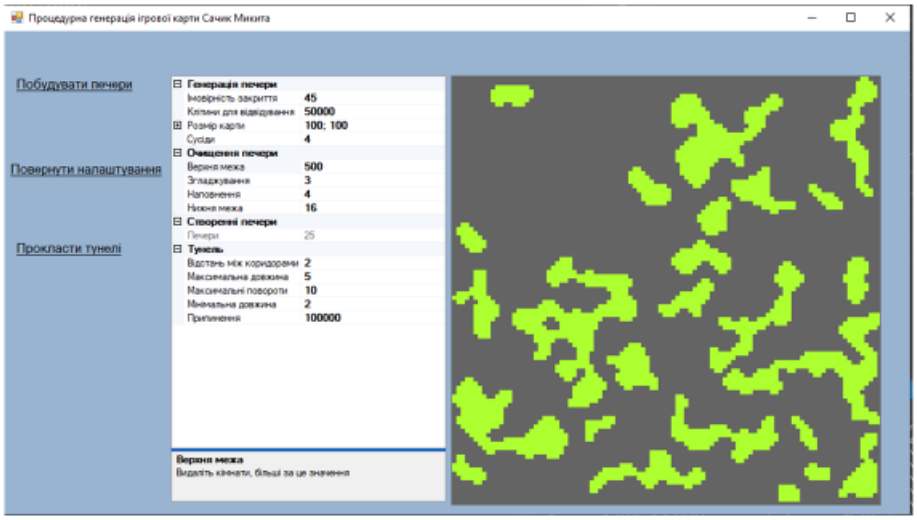
Верхня межа (h1) – максимальний дозволений розмір печер. Якщо печера буде містити більшу кількість клітин, її буде видалено.
Згладжування (h2) – видаляє окремі клітини з країв печери, клітинка з такою кількістю порожніх сусідів видаляється.
Наповнення (z) – заповнює отвори в печерах, відкрита клітина з цією кількістю закритих сусідів закрита.
Нижня межа (f) – мінімально допустимий розмір печер. Якщо печера буде містити меншу кількість клітин, її буде видалено.

Тунель

Відстань між коридорами (t) – кількість порожніх клітинок, які необхідно мати по обидва боки коридору, щоб його можна було побудувати. Це залежить від напрямку, в якому відбувається рух.
Максимальна довжина (tmax) – максимальна довжина коридору.
Максимальні повороти (tu) – максимальна кількість змін напрямку, яку може зробити коридор під час його будівництва.
Мінімальна довжина (tmin) – мінімальна довжина коридору.
Припинення (s) – кількість кроків, після якої припиняється спроба побудувати коридор.

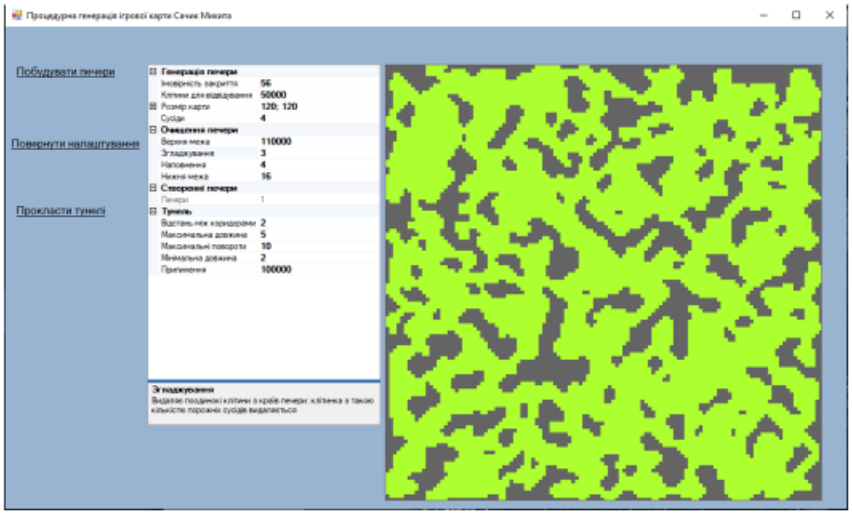
9

ЕКРАННА ФОРМА ДОДАТКУ ДЛЯ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ІГРОВИХ КАРТ



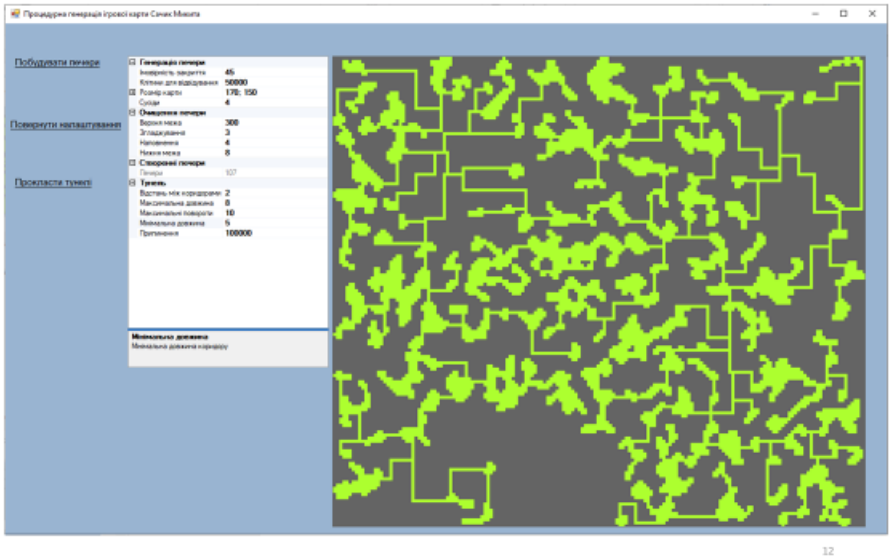
10

РЕЗУЛЬТАТ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ТИПУ «СУЦІЛЬНА ПЕЧЕРА»



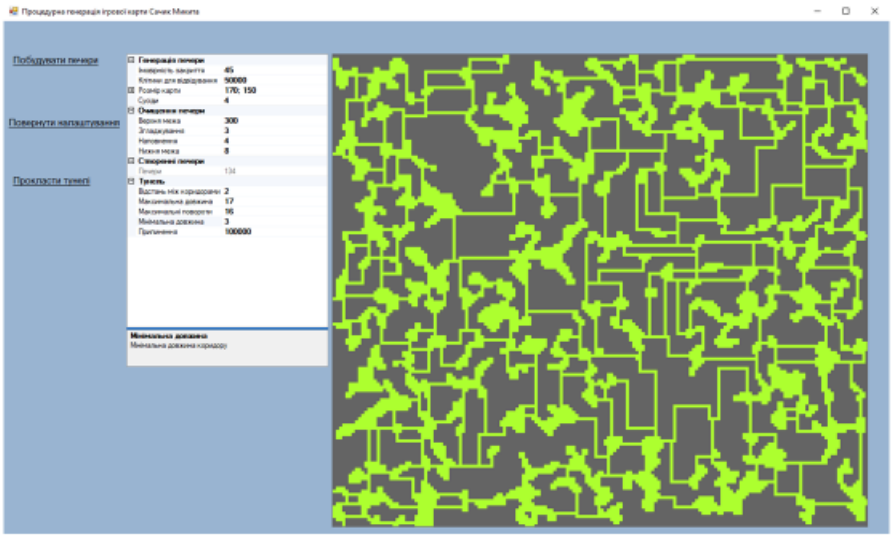
11

РЕЗУЛЬТАТ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ТИПУ «БАГАТО МАЛЕНЬКИХ КРАПЕЛЬ-ПЕЧЕР»



12

РЕЗУЛЬТАТ ГЕНЕРАЦІЇ ІГРОВОЇ КАРТИ ТИПУ «ЛАБІРИНТ»



13

АНАЛІЗ ХАРАКТЕРИСТИК СТВОРЕНИХ КАРТ

Критерії якості	Початкові налаштування	Лабіринт	Суцільна печера	Багато маленьких крапель-печер
Кількість печер	35	134	1	107
Розмір печер	Від 16 до 500, можна налаштувати	Від 8 до 300, можна налаштувати	Від 16 до 110000, можна налаштувати	Від 8 до 300, можна налаштувати
Згладжування печер	3, можна налаштувати	3, можна налаштувати	3, можна налаштувати	3, можна налаштувати
Кількість тунелів	23	67	0	43
Довжина тунелів	Від 2 до 5, можна налаштувати	Від 3 до 17, можна налаштувати	Від 2 до 5, але не використовуються	Від 5 до 8, можна налаштувати
Прямолінійність тунелів	Кількість поворотів 10, можна налаштувати	Кількість поворотів 16, можна налаштувати	Кількість поворотів може бути 10, але не використовується	Кількість поворотів 10, можна налаштувати
Кількість входів та виходів печер	Мінімальна 1, максимальна 5	Мінімальна 1, максимальна 6	0	Мінімальна 1, максимальна 4
Розміри карти	100 на 100, можна налаштувати	170 на 150, можна налаштувати	120 на 120, можна налаштувати	170 на 150, можна налаштувати
Печери, не поєднанні тунелями з іншими печерами	0	0	1	0

14

ВИСНОВКИ

1. Розглянуто головні підходи для реалізації автоматизованої генерації ігрової карти для ігор жанру Rougelike, та виявлені ключові елементи ігрової карти, що притаманні іграм цього жанру
2. Пройшов аналіз особливостей та недоліків генерації ігрової карти завдяки різним методам, та як найбільш доречні для реалізації розробки методу автоматизованої генерації ігрової карти для гри в жанрі roguelike вирішено використовувати клітинні автомати поєднанні з алгоритмами тунелювання та заливанням.
3. Визначенні критерії якості, за допомогою яких проведено аналіз характеристик створених карт, й можна зрозуміти, чи згенерована ігрова карта є грабельною, якісною, реалізує закладений початковий функціонал, чи вона не підходить.
4. Розроблена модель карти на основі клітинного автомату, що являє собою множину клітин, які поступово змінюють свій стан в залежності від заданих правил.
5. Розроблену схему функціонування програмного забезпечення що реалізує створені методи та алгоритми для автоматизованої генерації ігрової карти. Визначені параметри для створення ігрової карти, за допомогою яких можна контролювати генерацію ігрової карти, створювати різні патерни, варіації відповідно до критеріїв якості, які було визначено.
6. Розроблено додаток, який реалізує автоматизовану генерацію ігрової карти на основі клітинного автомату для ігор жанру Rougelike. Проведено моделювання шляхом генерації ігрових карт з різними характеристиками.

АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Золотухіна О.А., Сачик М.О. Автоматизована генерація в іграх. // IX Міжнародна науково-технічна Internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами». – Київ : НУХТ, 2022.

2. Золотухіна О.А., Сачик М.О. Автоматизована генерація з використанням клітинного автомату. // XV науково-технічна Конференція «сучасні інфокомунікаційні технології». - Київ: ДУТ, 2021.

ДЯКУЮ ЗА УВАГУ!