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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод генерації ігрового світу та структур для гри в жанрі RPG»

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

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

_____ Микита БОЙКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Микита БОЙКО

Керівник: Андрій БОНДАРЧУК
д.т.н., професор

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ - 2025

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

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
Ірина ЗАМРІЙ

“____“ _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бойку Микиті Сергійовичу

1. Тема кваліфікаційної роботи: «Метод генерації ігрового світу та структур для гри в жанрі RPG»
керівник кваліфікаційної роботи: Андрій БОНДАРЧУК, д.т.н., проф., професор кафедри ІПЗАС,

затверджені наказом Державного університету інформаційно-телекомунікаційних технологій «15» жовтня 2024 р. №320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вхідні дані до кваліфікаційної роботи: науково-технічна література, методи генерації ігрових локацій та будівель

4. Аналіз наявних методів та технологій для розробки процедурної генерації будівель та локацій для всесвіту гри в жанрі RPG

1. Дослідження зацікавленості ігрового процесу в іграх з процедурною генерацією ігрового світу.
2. Аналіз методів генерації процедурно створених ігрових світів в жанрі RPG
3. Розробка вимог до власного методу генерації ігрових структур

5. Перелік демонстраційного матеріалу: *презентація*

1. Мета, об’єкт та предмет дослідження

2. Аналіз існуючих методів генерації ігрових світів та будівель
3. Поєднання лінійного конгруентного методу та перлинного шуму.
4. Підсумковий результат роботи алгоритму генерації структур
5. Аналіз ефективності генератора псевдовипадкових чисел
6. Створення світу на основі заданого ключа

6. Дата видачі завдання «16» жовтня 2024 року

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10.2024 - 26.10.2024	Виконано
2	Аналіз існуючих методів вирішення поставленої задачі	27.10.2024 - 06.11.2024	Виконано
3	Розробка математичних функціональних моделей випадкової генерації ігрових структур для рольових ігор (RPG).	07.11.2024 - 18.11.2024	Виконано
4	Створення програмного забезпечення для розробки унікального методу генерації ігрових структур для рольових ігор (RPG).	19.11.2024 - 27.11.2024	Виконано
5	Впровадження індивідуального методу в програмне забезпечення для генерації ігрових структур у грі	28.11.2024 - 04.12.2024	Виконано
6	Оформлення роботи: вступ, висновки, реферат	05.12.2024 - 13.12.2024	Виконано
7	Розробка демонстраційних матеріалів	14.12.2024 - 16.12.2024	Виконано

Здобувач вищої освіти _____ Микита БОЙКО
(підпис)

Керівник
кваліфікаційної роботи _____ Андрій БОНДАРЧУК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра роботи 74 с., 28 рис., 29 фор., 3 табл., 20 джерел.

Мета роботи – покращення продуктивності та якості процедурного контенту шляхом вдосконалення алгоритмів, заснованих на моделі клітинних автоматів для генерації ігрових структур.

Об'єкт дослідження – процес генерації ігрових структур для створення ігрових світів.

Предмет дослідження – методи, засоби та алгоритми для автоматизованої генерації ігрових світів та структур.

Короткий зміст роботи: Результати проведеного дослідження свідчать, що використані математичні методи, алгоритмічні моделі та програмні засоби сприяють значному вдосконаленню процесу створення ігрових світів та структур для RPG-ігор.

Практична цінність отриманих результатів полягає у розробці нового методу генерації, який забезпечує підвищену якість створення ігрових рівнів та структур. Застосування цього методу дозволяє не лише покращити візуальну та структурну частину ігрового світу, а й оптимізувати роботу алгоритмів, прискорюючи процес генерації рівнів у майбутніх проєктах.

КЛЮЧОВІ СЛОВА: МЕТОДИ ГЕНЕРАЦІЇ, ІГРОВІ ПРЕДМЕТИ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ , ШУМ ПЕРЛІНА, RP

ABSTRACT

Text part of the master's qualification work: 74 pages, 28 figures, 29 formulas, 3 tables, 20 sources.

The purpose of the work – to improve the productivity and quality of procedural content by improving algorithms based on the cellular automata model for generating game structures..

таблworlds.

Subject of Research – methods, tools and algorithms for automated generation of game worlds and structures.

Summary of the Work: A short summary of the work: The results of the research show that the various mathematical methods, algorithmic models and software are suitable for the significant and thorough process of creating game worlds and structures for RPG games.

The practical value of obtaining results lies in the development of a new generation method, which will ensure increased efficiency in the creation of game levels and structures. This method allows not only to improve the visual and structural structure of the game world, but also to optimize the work of algorithms, speeding up the process of generating ranks in upcoming projects.

KEYWORDS: GENERATION METHODS, GAME ITEMS, PROCEDURAL GENERATION, PERLIN NOISE, RPG

ЗМІСТ

3. РОЗРОБКА МОДЕЛІ ВИПАДКОВОЇ ГЕНЕРАЦІЇ СВІТУ ТА СТРУКТУР ДЛЯ ГРИ В ЖАНРІ RPG.....52

ВСТУП

У світі сьогодні існує велика кількість мобільних ігор різних жанрів, які викликають все більший інтерес у користувачів. Одним з найпопулярніших жанрів є "Tower Defense", який поєднує в собі елементи стратегії та аркади. Для розробки таких ігор використовуються різноманітні інструменти, такі як: мова програмування C#, ігровий двигун Unity, програма для створення 3D-моделей Blender, система контролю версій GitHub, рекламна платформа Unity Ads та магазин додатків Google Play.

Задачею даного дипломного проєкту є розробка гри жанру "Tower Defense" з елементами RPG для мобільних пристроїв. Об'єктом дослідження є методи розробки гри на ігровому двигуні Unity з використанням мови програмування C# та програми для створення 3D-моделей Blender. Предметом дослідження є створення гри з механіками "Дерева пасивних умінь" та "Дерево діалогів", що розширюють можливості гравців та забезпечують більш глибоке занурення в гру. Метою роботи є дослідження методів проєктування та розробки програмного забезпечення, а також оптимізації гри для підвищення її продуктивності. Проєкт буде зберігатися на платформі GitHub, а для монетизації гри будуть використовуватися рекламні інструменти Unity Ads та магазин додатків Google Play. Результатом роботи стане гра "Tower Dimension", яка зможе знайти свою аудиторію в ігровій спільноті та стати додатковим інструментом в популяризації жанру "Tower Defense" на мобільних пристроях.

Дослідження передбачає вивчення методів проєктування та розробки програмного забезпечення, методів створення 3D-моделей та методів оптимізації програмного забезпечення. Науковою новизною є розробка гри з використанням міксу механік: механіки дерева пасивних умінь та механіки дерева діалогів. Галуззю використання є ігрова спільнота, яка становить значну частину ринку мобільних ігор.

1. ТЕОРЕТИЧНІ ОСНОВИ ЖАНРУ RPG ТА ПІДХОДІВ ДО СТВОРЕННЯ ІГРОВИХ РІВНІВ ТА ЛОКАЦІЙ

1.1 Формулювання жанру RPG

Жанр рольових ігор (RPG, від англ. Role-Playing Game) є одним із найпопулярніших і найвпливовіших у світі відеоігор. RPG-ігри характеризуються глибоким зануренням у вигаданий світ, що дозволяє гравцям брати на себе ролі вигаданих персонажів і взаємодіяти з багатогранним сюжетом та складною механікою гри. Основні риси RPG включають наступні аспекти:

Сюжет та історія. В основі більшості RPG лежить розгорнутий сюжет з численними персонажами, місіями та подіями. Сюжет часто має різноманітні розгалуження та кінцівки, залежно від рішень гравця. Такі ігри часто включають детальні описи світів, їх історії, культури та політичних структур. Сюжетні лінії можуть включати епічні битви, інтриги, любовні історії та інші елементи, що роблять гру цікавою та захопливою. Наприклад, у грі The Witcher 3: Wild Hunt гравці занурюються в складний сюжет, що розгортається навколо головного героя, його пошуків та взаємин із численними NPC. Кожне прийняте рішення може суттєво змінити хід подій, що забезпечує унікальний ігровий досвід.

Розвиток персонажа. Однією з ключових складових RPG є система прокачування персонажа. Гравці можуть підвищувати рівні персонажів, отримувати нові навички, здібності та спорядження. Це створює відчуття прогресу та досягнень, стимулюючи подальшу гру. Система розвитку часто включає можливість вибору класів, спеціалізацій та різних шляхів розвитку персонажа. Наприклад, у грі TES: Skyrim гравці можуть обирати між воїном, магом або злодієм, розвиваючи навички відповідно до свого стилю гри. Цей елемент додає грі глибини та варіативності, дозволяючи гравцям адаптувати персонажа під свої потреби та уподобання.

Рис 1.1 Розвиток персонажа у грі TES: Skyrim

Світ гри. Світ RPG-ігор зазвичай великий і детально пророблений. Він може бути відкритим, дозволяючи гравцям вільно досліджувати його, або мати чітко визначені рівні. Важливими елементами є реалістичні та взаємопов'язані екосистеми, NPC, які мають власні історії та мотивації, і численні місії та завдання. Наприклад, у грі Elder Scrolls гравці можуть вільно досліджувати великий відкритий світ, знаходити нові місії, зустрічати різних персонажів та відкривати нові локації. Світ гри часто має свої унікальні особливості, кліматичні умови та ландшафти, що робить його більш живим та захопливим.

Відкритий світ. Багато сучасних RPG пропонують відкриті світи, де гравці можуть вільно переміщуватися і досліджувати без обмежень. Це забезпечує відчуття свободи та можливість самостійного визначення цілей і завдань. Наприклад, у грі The Legend of Zelda: Breath of the Wild гравці можуть досліджувати великий відкритий світ, виконуючи місії у будь-якому порядку та знаходячи нові цікаві місця. Такий підхід дозволяє гравцям створювати власні історії та визначати свій шлях у грі, що робить кожне проходження унікальним.

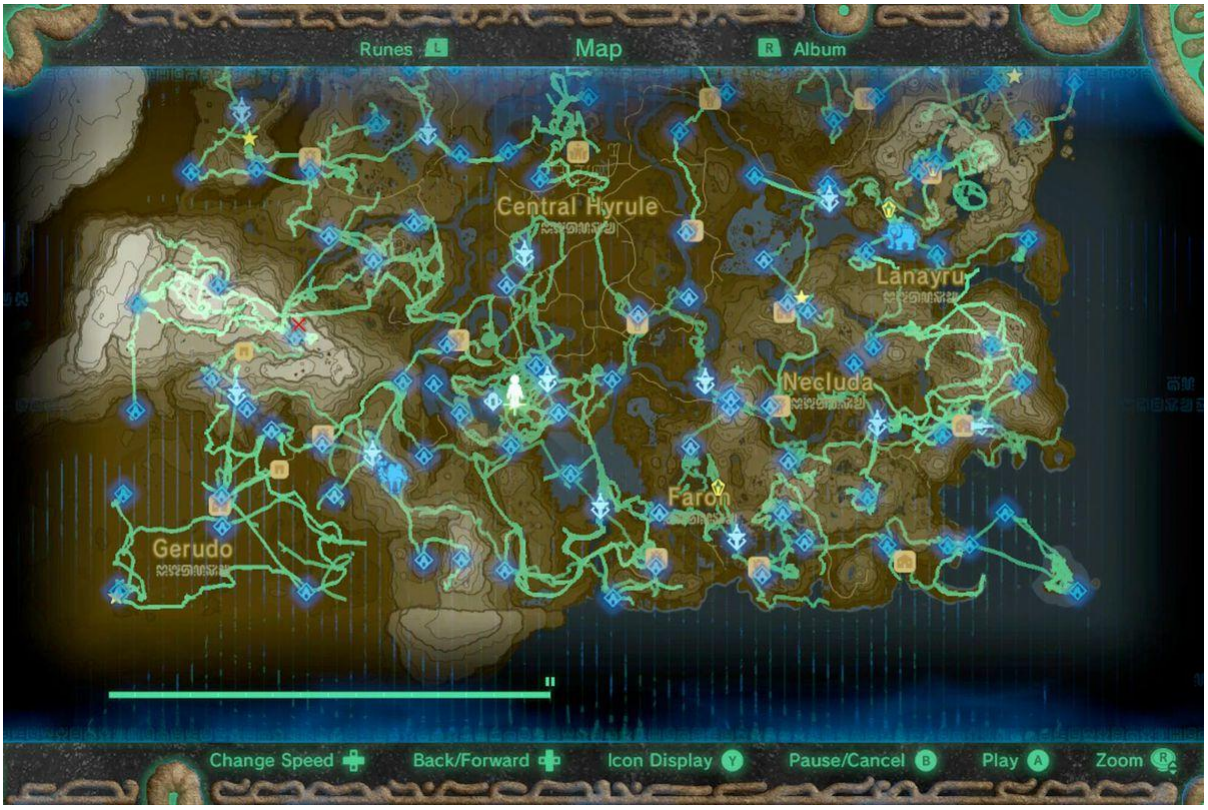


Рис 1.2 Ігрова мапа гри The Legend of Zelda: Breath of the Wild

Деталізація. Світ RPG часто насичений деталями, які допомагають створити глибоке занурення. Це можуть бути як візуальні елементи (архітектура, ландшафти, клімат), так і сюжетні елементи (легенди, історії, записи). Наприклад, у грі Red Dead Redemption 2 деталі, такі як різноманітні види рослин, тварин та архітектурні стилі різних регіонів, створюють відчуття реалістичності та занурення у світ гри. Розробники часто включають дрібні деталі, що роблять світ більш живим та автентичним, від мініігор до випадкових подій, які гравець може зустріти під час подорожі.

Інтерактивність. RPG надають гравцям значну свободу дій, дозволяючи взаємодіяти з оточенням, вибирати різні стратегії проходження та впливати на розвиток сюжету через свої рішення. Наприклад, у грі Mass Effect гравці можуть впливати на хід подій через свої діалоги та дії, змінюючи відносини з іншими персонажами та кінцівки гри. Цей аспект робить гру більш гнучкою та адаптивною до стилю гри кожного гравця, дозволяючи їм створювати унікальні ігрові моменти.

Взаємодія з NPC. Неписані персонажі (NPC) в RPG грають важливу роль. Гравці можуть спілкуватися з ними, отримувати завдання, торгувати або вступати в конфлікти. Від рішень гравця залежить, як NPC реагуватимуть на його дії. Наприклад, у грі The Elder Scrolls V: Skyrim NPC мають свої розклади, професії та відносини з іншими персонажами, що створює живий і динамічний світ. Взаємодія з NPC може включати виконання квестів, участь у політичних інтригах або просто обмін інформацією, що збагачує ігровий досвід та додає глибини сюжету.

Рішення та наслідки. Рішення, прийняті гравцем, часто мають довгострокові наслідки, що впливають на хід сюжету та відносини з іншими персонажами. Це додає грі глибини та робить кожне проходження унікальним. Наприклад, у грі The Witcher 3: Wild Hunt кожне прийняте рішення може суттєво вплинути на долю персонажів і розвиток подій, що робить гру більш інтерактивною та цікавою. Це створює ілюзію живого світу, де дії гравця мають значення, що стимулює обдумане прийняття рішень та повторні проходження гри.

Рис 1.3 Прийняття рішень у The Witcher 3: Wild Hunt

Ігрові механіки. В RPG часто використовуються різноманітні механіки, такі як бойова система, магія, ремесло, алхімія та інші, що додають різноманіття ігровому процесу і дозволяють гравцям вибирати свій власний стиль гри. Наприклад, у грі Diablo III гравці можуть обирати між різними класами персонажів, кожен із яких має свої унікальні здібності та стиль бою. Це дозволяє гравцям експериментувати з різними підходами до проходження гри, вибираючи ті механіки, які найбільше відповідають їхнім уподобанням.

Таким чином, жанр RPG є надзвичайно різноманітним, що робить його привабливим для широкої аудиторії гравців. Завдяки глибоким сюжетам, складним ігровим механікам та великому відкритому світу, RPG-ігри надають гравцям можливість зануритися у вигаданий світ і прожити унікальну історію, створену власними рішеннями та діями.

1.2 Підхід до створення рівнів та ігрових локацій

Процес створення рівнів та ігрових локацій у RPG вимагає ретельного планування та міждисциплінарного підходу, що включає як художні, так і технічні аспекти. Існують різні підходи до створення рівнів, які залежать від таких ключових етапів, як концептуалізація, планування та інтеграція ігрової механіки.

Концептуалізація є першочерговим етапом, що визначає загальну ідею та напрямок розробки локації. Вона містить такі аспекти, як функціональність, тематика та стиль. Тематика та стиль визначають загальну атмосферу та візуальну естетику локацій. На цьому етапі розробники вирішують, чи буде локація лісом, містом, підземеллям чи іншою тематичною зоною. Вибір кольорової гами, архітектурних елементів та деталей оточення є важливими для створення унікального візуального стилю, що відповідає загальній тематиці гри. Наприклад, у грі Skyrim лісові локації мають густу рослинність, темні відтінки та природні звуки, що створює відчуття занурення у природу.

Функціональність локації є не менш важливою. Вона визначає, яке саме завдання виконує певна локація у грі. Це може бути бойова зона, де гравець бореться з ворогами, або місце для відпочинку та взаємодії з NPC. Функціональність також включає визначення місць для отримання квестів та інших ігрових активностей. Наприклад, у грі The Witcher 3 локації поділяються на зони для боїв, зони для дослідження та зони для взаємодії з іншими персонажами, що забезпечує різноманітність ігрового процесу [1].

Планування охоплює створення детальних карт та планів локацій. На цьому етапі розробники визначають розташування основних об'єктів, шляхів руху гравця та NPC. Важливо забезпечити логічну структуру рівня, яка сприяє зручній навігації та цікавому геймплею. Створення планів включає врахування ландшафтних особливостей, розміщення будівель, доріг, мостів та інших архітектурних елементів. Наприклад, у грі Dragon Age: Inquisition детальні плани локацій допомагають створити логічну структуру рівнів, що дозволяє гравцям легко орієнтуватися у просторі.

Ігрова механіка інтегрується в процесі планування. Це включає створення різних ігрових викликів, таких як пастки, головоломки та взаємодії з оточенням. Локації повинні пропонувати різні виклики та можливості для гравців, що додає грі динамічності та глибини. Наприклад, у грі Dark Souls інтеграція пасток і складних головоломок робить кожну локацію унікальною, стимулюючи гравців до обережного і стратегічного підходу.

Архітектура та розміщення об'єктів включають моделювання тривимірних моделей локацій та об'єктів. На цьому етапі розробники створюють моделі будівель, ландшафтів, рослинності та внутрішнього оздоблення. Важливо забезпечити високу деталізацію та реалістичність, щоб створити занурення у гру. Наприклад, у грі Red Dead Redemption 2 детальні моделі локацій і об'єктів створюють реалістичний ігровий світ, що сприяє повному зануренню гравців у гру.

Рис 1.4 Деталізація ігрового світу Red Dead Redemption 2

Наповнення локацій відбувається після моделювання. Розробники розміщують об'єкти та елементи оточення відповідно до плану, включаючи розташування ворогів, скарбів, NPC та інших інтерактивних об'єктів. Наприклад, у грі The Elder Scrolls V: Skyrim розміщення ворогів і скарбів забезпечує цікавий ігровий процес, стимулюючи гравців до дослідження світу.

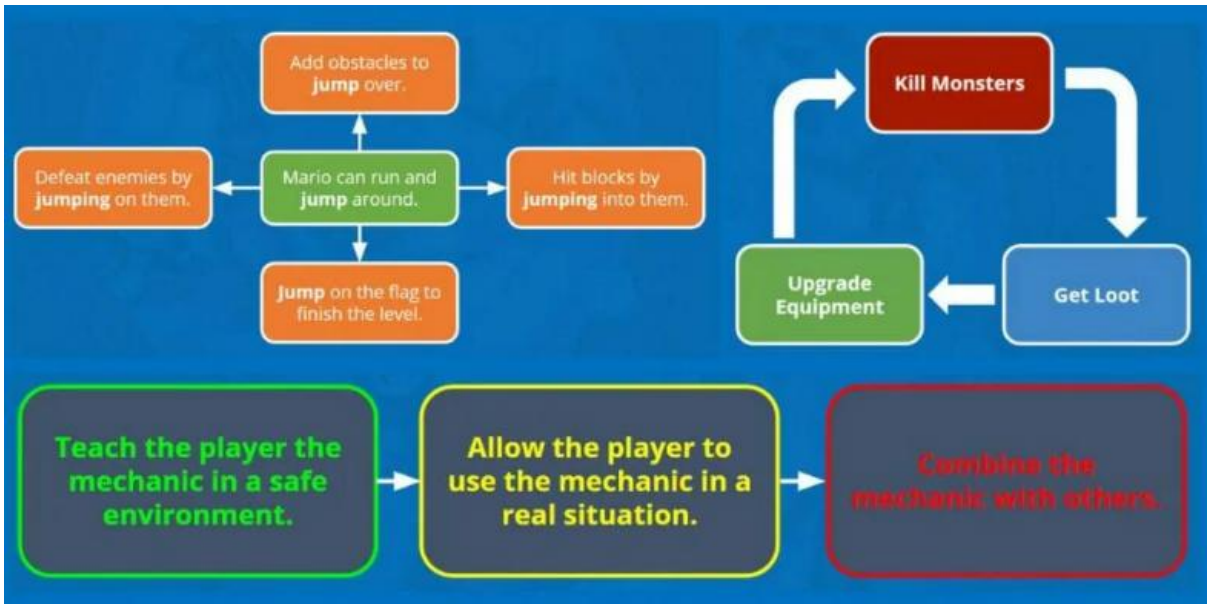


Рис 1.5 Основні принципи геймдизайну

На відміну від традиційних RPG, у грі Minecraft процес генерації рівнів використовує процедурну генерацію для створення великих, випадкових світів. Кожен новий світ у Minecraft генерується на основі насіння, що дозволяє створювати унікальні ландшафти, включаючи гори, річки, печери та різноманітні біоми. Цей підхід забезпечує безмежні можливості для дослідження та креативного будівництва, надаючи гравцям неповторний досвід кожного разу, коли вони починають нову гру.

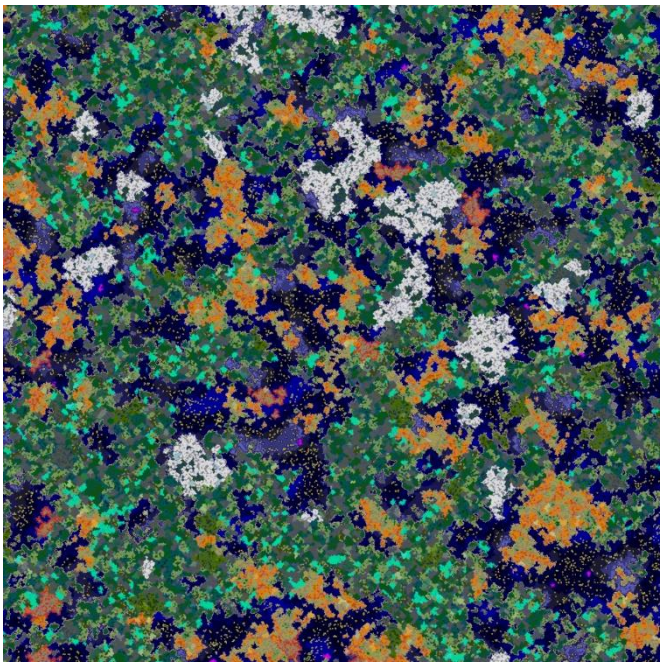


Рис 1.6 Генерація ігрового світу Minecraft

Скриптування та інтерактивність охоплюють написання сценаріїв, що визначають поведінку NPC, події та взаємодії у грі. Це включає діалоги, реакції на дії гравця та інші динамічні елементи. Наприклад, у грі Mass Effect написання складних сценаріїв і діалогів дозволяє створити багатовимірні взаємодії між гравцем і NPC, що робить гру більш захоплюючою.

Завершальним етапом є тестування. На цьому етапі розробники проводять тестування для виявлення та виправлення помилок, а також для оптимізації ігрового процесу. Важливо забезпечити плавний та цікавий геймплей, без багів та збоїв, що сприяє задоволенню гравців і успіху гри.

Таким чином, створення рівнів та ігрових локацій у RPG є складним і багатогранним процесом, що вимагає координації різних аспектів розробки для досягнення високоякісного ігрового досвіду.

1.3 Методи створення рівнів та ігрових локацій в іграх RPG

1. Ручне створення (Handcrafted) дозволяє створювати унікальні та високо деталізовані локації, що відповідають задуму розробників. Ручне створення забезпечує повний контроль над дизайном та естетикою, дозволяє вносити творчі ідеї та експериментувати з різними елементами. Однак, цей процес є дуже трудомістким та вимагає значних ресурсів, як людських, так і тимчасових. Ручне створення може бути менш гнучким для змін у процесі розробки та потребує високої кваліфікації розробників.

2. Процедурна генерація (Procedural Generation) [2] дозволяє автоматично створювати великі та різноманітні світи з мінімальними затратами часу та ресурсів. Вона забезпечує варіативність ігрового процесу та можливість повторного проходження гри з новими рівнями та локаціями. Водночас, процедурна генерація може мати обмежену контрольованість та передбачуваність результатів. Такі рівні можуть бути менш детально опрацьованими та менш цікаво структурованими, ніж вручну створені локації.

3. Алгоритмічний дизайн використовує складні алгоритми для створення специфічних типів локацій, таких як лабіринти або підземелля. Цей метод дозволяє створювати логічні структури та виклики для гравців, враховуючи їх дії та попередні рішення. Алгоритмічний дизайн забезпечує динамічну генерацію локацій з урахуванням дій гравця та ігрових механік, підвищуючи варіативність гри та адаптуючи рівні до поточних навичок гравця. Однак, він може бути складним для налаштування та потребує високої кваліфікації програмістів, а також ретельного тестування для забезпечення стабільності та збалансованості гри.

4. Ітераційний дизайн полягає у постійному вдосконаленні локацій через цикл розробки, тестування та зворотного зв'язку. Це дозволяє поступово покращувати якість рівнів та забезпечити відповідність геймплейних аспектів очікуванням гравців. Ітераційний дизайн дозволяє швидко реагувати на виявлені проблеми та побажання гравців, забезпечуючи високий рівень адаптації до змін у процесі розробки. Проте, цей підхід може бути довготривалим процесом, що потребує постійного ресурсу на тестування та збір зворотного зв'язку. Він вимагає налагодженої комунікації між командою розробників та спільнотою гравців.

5. Модульний дизайн використовує набір попередньо створених модулів або блоків для складання рівнів та локацій. Це дозволяє швидко створювати нові області, комбінуючи готові елементи, що знижує час і витрати на розробку. Модульний дизайн забезпечує високу узгодженість стилю та дизайну між різними локаціями гри. Однак, він може обмежувати унікальність і різноманітність рівнів, оскільки використання одних і тих самих модулів може створювати відчуття повторюваності. Такий підхід потребує ретельного дизайну модулів для забезпечення їх сумісності та цікавого геймплею.

6. Процедурна генерація (Procedural Generation) є перспективним та ефективним підходом до створення ігрових рівнів, що дозволяє автоматично генерувати великі та різноманітні світи з мінімальними затратами часу та ресурсів. Цей метод забезпечує унікальність кожного проходження гри, що значно підвищує її реіграбельність і стимулює інтерес гравців. Прикладом успішного застосування процедурної генерації є популярна гра Minecraft, яка стала еталоном у цій сфері.

У Minecraft гравці досліджують величезний відкритий світ, що складається з блоків різних матеріалів. Ці блоки можна руйнувати та використовувати для створення нових структур. Процедурна генерація в Minecraft забезпечує унікальність кожного світу, дозволяючи гравцям щоразу відкривати нові області для дослідження. Гравці мають можливість створювати власні структури, досліджувати печери, шукати ресурси та боротися з ворогами, що додає грі різноманітності та глибини.

Цей підхід дозволяє зменшити витрати на розробку, оскільки великі та варіативні світи створюються автоматично, без необхідності в ручній роботі над кожною локацією. Процедурна генерація також забезпечує значну варіативність ігрового процесу, оскільки кожен новий світ створюється за допомогою алгоритмів, що враховують різні параметри та умови. Це робить гру цікавою навіть після численних проходжень, оскільки гравці постійно стикаються з новими викликами та можливостями.

Однак, процедурна генерація має свої виклики [3]. Автоматично створені світи можуть бути менш структурованими та деталізованими, ніж світи, створені вручну. Для досягнення високої якості процедурно згенерованих рівнів необхідно використовувати складні алгоритми та ретельно налаштовувати їх для забезпечення цікавого та збалансованого геймплею. Успіх Minecraft у значній мірі пояснюється здатністю гри поєднувати процедурну генерацію з інтуїтивно зрозумілими механіками, які дозволяють гравцям творчо взаємодіяти зі світом.

Вибір процедурної генерації як основного підходу до створення ігрових рівнів може значно підвищити ефективність розробки та залучення гравців. Застосування цього методу дозволяє створювати великі, динамічні та варіативні світи, забезпечуючи унікальний досвід для кожного гравця. Успішний приклад Minecraft демонструє, що за допомогою процедурної генерації можна досягти високої якості та популярності гри, стимулюючи творчість та дослідницький інтерес гравців.

Для процедурної генерації Minecraft використує такі алгоритми:

1. Шум Перліна [4] є популярним методом для створення природних ландшафтів. Він генерує псевдовипадкові значення [5], які плавно змінюються, що дозволяє створювати реалістичні гори, долини та інші географічні особливості. Формула для 2D-шуму Перліна може виглядати так:

$$P(x,y) = \sum_{i=0}^{n} a_i \cdot noise(2^i \cdot x, 2^i \cdot y) \quad (1.1)$$

де *P(x,y)* — значення шуму в точці *(x, y)*

a_i — амплітуда на і-му рівні октави, noise — функція шуму,

n — кількість октав.

Однак, іноді можуть використовуватися інші типи шуму, такі як сімейство шуму Перліна (Simplex Noise) або фрактальний шум (Fractal Noise), що забезпечує більш складні та деталізовані результати.

Фрактальний шум є сумою декількох шарів (октав) шуму Перліна

$$Fractal\ Noise(x,y) = \sum_{i=0}^{n} \frac{1}{2^i} \cdot Perlin\ Noise\ (2^i x, 2^i y) \quad (1.2)$$

Де n– кількість октав, кожна наступна октава зменшується в амплітуді і збільшується в частоті.

Simplex Noise є більш оптимізованою версією шуму Перліна, що краще підходить для тривимірних і вищих розмірностей. Основні кроки включають:

$$Simple\ Noise(x,y) = \sum_{i=0}^{n} a_i \cdot Gradient\ Noise(x_i, y_i) \quad (1.3)$$

Де a_i — вагові коефіцієнти,

x_i, y_i — координати точок на гексагональній сітці.

Для створення плавних переходів між різними значеннями шуму використовується інтерполяція. Прикладом є дволінійна інтерполяція, яка допомагає створювати більш гладкі і природні ландшафти. Формула дволінійної інтерполяції виглядає наступним чином:

1. Визначаємо координати сусідніх точок і значення градієнтів в цих точках:

$$P = (x,y), \quad i = floor(x), \quad j = floor(y) \quad (1.4)$$

$$g_{00} = gradient\ at(i,j), \quad g_{10} = gradient\ at(i+1,j) \quad (1.5)$$

$$g_{01} = gradient\ at(i,j+1), \quad g_{11} = gradient\ at(i+1,j+1) \quad (1.6)$$

2. Обчислюємо вагові коефіцієнти:

$$u = x - i, \quad v = y - j \quad (1.7)$$

3. Виконуємо інтерполяцію:

$$n_{00} = g_{00} \cdot [u,v], \quad n_{10} = g_{10} \cdot [u-1,v] \quad (1.8)$$

$$n_{01} = g_{01} \cdot [u,v-1], \quad n_{11} = g_{11} \cdot [u-1,v-1] \quad (1.9)$$

$$n_{x0} = n_{00}\big(1 - f(u)\big) + n_{10} f(u) \quad (1.10)$$

$$n_{x1} = n_{01}\big(1 - f(u)\big) + n_{11} f(u) \quad (1.11)$$

$$n_{xy} = n_{x0}\big(1 - f(u)\big) + n_{x1} f(v) \quad (1.12)$$

де f(u) — функція згладжування, що часто використовується для створення плавних переходів між значеннями.

Однак, процедурна генерація має свої виклики. Автоматично створені світи можуть бути менш структурованими та деталізованими, ніж світи, створені вручну. Для досягнення високої якості процедурно згенерованих рівнів необхідно використовувати складні алгоритми та ретельно налаштовувати їх для забезпечення цікавого та збалансованого геймплею. Успіх Minecraft у значній мірі пояснюється здатністю гри поєднувати процедурну генерацію з інтуїтивно зрозумілими механіками, які дозволяють гравцям творчо взаємодіяти зі світом.

Вибір процедурної генерації як основного підходу до створення ігрових рівнів може значно підвищити ефективність розробки та залучення гравців. Застосування цього методу дозволяє створювати великі, динамічні та варіативні світи, забезпечуючи унікальний досвід для кожного гравця. Процедурна генерація також дає змогу адаптувати ігровий досвід під різні вподобання гравців, забезпечуючи більшу різноманітність контенту. Крім того, вона може зменшити витрати часу та ресурсів на розробку, що особливо важливо для невеликих команд. Успішний приклад Minecraft демонструє, що за допомогою процедурної генерації можна досягти високої якості та популярності гри, стимулюючи творчість та дослідницький інтерес гравців.

2. АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ВИПАДКОВОЇ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ ТА ІГРОВИХ СТРУКТУР В ГРІ ЖАНРУ RPG

2.1 Аналіз існуючих методів створення ігрових світів

При розробці RPG ігор використовується кілька різних підходів до генерації ігрових світів, кожен з яких має свої переваги та недоліки. Основні методи можна поділити на три категорії: процедурна генерація, попередньо створені шаблони та комбіновані підходи.

Процедурна генерація [6] дозволяє створювати великі світи на основі алгоритмів, таких як шум Перліна або фрактали [7]. Вона широко використовується для створення варіативних світів без необхідності ручної розробки кожної деталі. Наприклад, у Minecraft цей підхід забезпечує високу варіативність проходжень, хоча інколи призводить до аномалій, як-от неприродні переходи між біомами.

Попередньо створені шаблони надають розробникам повний контроль над дизайном кожної локації, що дозволяє створювати більш деталізовані світи. Однак, повторюваність елементів з часом може знижувати відчуття унікальності кожної локації.

Комбіновані підходи поєднують випадковість процедурної генерації з контролем над ключовими зонами. У The Witcher 3 це дозволило розробникам поєднати детальну ручну роботу над містами з процедурно згенерованими природними ландшафтами. Такий підхід дозволяє створювати унікальні ігрові світи, де основні локації виглядають деталізованими, а навколишні території зберігають варіативність. Проте, одним з викликів цього підходу є узгодженість між вручну створеними елементами та згенерованими, щоб уникнути різких контрастів та нелогічних переходів між ними.

2.1.1 Процедурна генерація

Процедурна генерація: Це один з найпоширеніших методів, що використовує математичні алгоритми для автоматичного створення великих ігрових світів. Одним із найвідоміших алгоритмів є шум Перліна, який дозволяє генерувати випадкові, але органічні ландшафти, такі як гори, долини, печери та водойми. Інші техніки включають використання фракталів [8], симуляції фізичних процесів (наприклад, ерозії) або галузевих структур для побудови складних світів.

Переваги:

1. Висока варіативність: Світ кожного разу унікальний, що забезпечує високий рівень варіативності проходжень.
2. Економія ресурсів на розробку: Оскільки світ генерується автоматично, немає потреби вручну проєктувати кожну його частину.

Приклад: У Minecraft шум Перліна використовується для генерації великих природних ландшафтів, таких як гори, долини та річки. Це дозволяє гравцям досліджувати світи, які завжди відрізняються, навіть після багатьох годин гри, що сприяє відчуттю відкритого світу і пригод. Ця варіативність є ключовою причиною популярності Minecraft протягом багатьох років.

Недоліки:

1. Передбачуваність та одноманітність: Всупереч випадковості, алгоритми можуть створювати схожі структури, що призводить до одноманітних і передбачуваних ландшафтів. Наприклад, Minecraft відомий своїми великими площами з подібними шаблонами.
2. Відсутність сюжетного контексту: Процедурна генерація зазвичай не враховує сюжетних вимог RPG-ігор, що може призвести до слабкої інтеграції наративу та випадкової структури, яка не підтримує історію гри.
3. Аномалії у створенні світу, неприродні ландшафти: Хоча шум Перліна дозволяє створювати природні ландшафти, в деяких випадках генерація може створювати "аномалії", коли сегменти світу не виглядають логічно або природно.

Приклад: В Minecraft, наприклад, часто можна побачити висячі шматки землі або окремі блоки, що левітують у повітрі без видимої причини. Інший частий приклад — випадковий перехід між біомами, коли поряд можуть знаходитися пустелі та снігові гори. Це руйнує цілісність світу та знижує рівень занурення гравця. Також у старих версіях Minecraft гравці часто зустрічали аномалії, коли маленькі частини світу не з'єднувались належним чином через зміни у версіях або особливості процедурної генерації, що призводило до створення "дивних" переходів між біомами, де, наприклад, ліс міг раптово перейти в океан або пустелю

2.1.2 Попередньо створені шаблони

Попередньо створені шаблони: У цьому підході використовуються заздалегідь розроблені області або структури, які поєднуються або комбінуються під час процесу генерації. Це дозволяє створювати детально опрацьовані елементи світу, але обмежує варіативність.

Переваги:

1. Контрольованість дизайну: Дизайнери можуть гарантувати високу якість кожної області та зберігати гармонію між наративом та архітектурою.
2. Підтримка наративу: Локації можна легко інтегрувати в сюжетні завдання, що важливо для RPG-ігор.
3. Контроль над дизайном та деталізацією: Попередньо створені шаблони дозволяють розробникам створювати складні та детальні локації з повним контролем над архітектурою і розміщенням елементів світу, що покращує якість гри та атмосферу.

Приклад: У грі Dragon Age: Inquisition детально створені міста, замки та інші ключові локації дозволяють гравцеві зануритися у насичений і добре спроєктований світ, де кожна локація має свою унікальну атмосферу. Такий підхід дозволяє розробникам зосередитися на створенні сюжетно важливих локацій, що значно покращує наративну частину гри.

Недоліки:

1. Обмежена варіативність: Випадковість світу обмежена наперед заданими шаблонами, що робить світ передбачуваним після декількох проходжень.
2. Трудомісткість: Створення великих і детальних шаблонів вручну вимагає значних витрат часу та ресурсів.
3. Повторюваність структури: Коли використовуються попередньо створені шаблони, варіативність світу значно знижується. Це може призвести до того, що гравці помічають повторювані локації або структури в різних частинах світу.

Приклад: у багатьох RPG або стратегіях гравець може стикатися з однаковими архітектурними елементами(мости або печери), що робить гру менш захопливою. У грі The Elder Scrolls V: Skyrim деякі печери та підземелля використовували повторювані архітектурні елементи. Гравці часто відзначали, що ці локації відчувалися дуже схожими, хоча розміри світу та його деталізація повинні були створювати відчуття унікальності

2.1.3 Комбінований підхід

Комбіновані підходи: У багатьох сучасних іграх використовується комбінація процедурної генерації та попередньо створених шаблонів. Наприклад, деякі частини світу можуть бути створені процедурно, але ключові локації, такі як міста, замки або сюжетні зони, проєктуються вручну.

Переваги:

1. Баланс між варіативністю та контролем: Можна створювати варіативні світи з певним рівнем контролю над ключовими елементами.
2. Підтримка гібридного підходу: Дає змогу інтегрувати як випадковість, так і наратив.
3. Баланс між варіативністю та ручним контролем: Комбіновані підходи забезпечують баланс між випадковістю процедурної генерації та контрольованістю ручної розробки ключових елементів світу. Це дозволяє

створювати великі варіативні світи з деталізованими, сюжетно важливими локаціями.

Приклад: У The Witcher 3 деякі регіони світу створюються процедурно для підтримки великої площі відкритого світу, тоді як міста, села та інші сюжетні зони детально спроєктовані вручну. Це дозволяє зберігати унікальність світу і забезпечувати цікаві зони для дослідження, водночас не перевантажуючи розробників необхідністю вручну створювати кожен шматок ландшафту.

Недоліки:

1. Складність реалізації: Інтеграція двох підходів вимагає значних зусиль та правильної організації архітектури світу.
2. Проблеми з узгодженістю: Іноді процедурно створені частини світу можуть конфліктувати з заздалегідь розробленими локаціями.
3. Непослідовність між ручною та процедурною генерацією: При комбінуванні процедурної генерації з попередньо створеними шаблонами можуть виникати ситуації, коли згенеровані частини світу виглядають незв’язаними або нелогічними поруч з ручними елементами. Це створює відчуття, що різні частини світу були згенеровані окремо і не пов'язані між собою.

Приклад: У грі The Elder Scrolls IV: Oblivion комбінована система часто створювала "шви" між процедурно згенерованими локаціями (наприклад, лісами) та ручними елементами, такими як міста. Це призводило до того, що перехід між цими локаціями виглядав штучно та незграбно, з чіткими змінами текстур і рельєфу.

2.1.4 Основні проблеми

Основні проблеми наявних підходів:

Лінійність та передбачуваність: Хоча процедурна генерація забезпечує варіативність, після кількох ігрових сесій гравці можуть почати помічати повторювані патерни. Це робить світ менш захопливим для дослідження і може

призвести до втрати інтересу. Наприклад, у Terraria або Minecraft різні світи, хоча й генеруються випадково, часто мають подібні структури (печери, біоми, ландшафти).

Недостатня адаптивність до жанру RPG: Алгоритми генерації світу часто не враховують специфічні потреби RPG. Важливими є не тільки візуальні чи архітектурні аспекти, а й інтеграція наративу, рольових елементів та взаємодія з гравцем. Наприклад, RPG-локації можуть вимагати певної логіки для побудови місць з важливими сюжетними подіями.

Проблеми масштабу: Генерація великих ігрових світів, які повинні бути одночасно детальними та різноманітними, може призвести до перевантаження системи. Наприклад, у No Man's Sky, хоча всесвіт генерується практично нескінченно, технічні обмеження іноді призводять до повторюваних елементів або проблем з продуктивністю, коли гравець переміщається між великими зонами.

Нестача інтерактивності: Традиційні методи генерації часто не враховують можливості зміни ігрового світу на основі дій гравця. Для RPG ігор це є суттєвим недоліком, оскільки гравці очікують, що їхні рішення будуть мати вплив на світ. Наприклад, якщо гравець зруйнує замок або побудує нову фортецю, ці зміни повинні мати довготривалі наслідки у грі.

2.2 Можливі напрями розвитку методів генерації ігрових світів та структур для RPG-ігор

Технології, які використовуються для створення ігрових світів у жанрі RPG, постійно розвиваються, що відкриває нові можливості для покращення якості генерації рівнів та структур. Одним із перспективних напрямків є розвиток процедурної генерації з використанням складних алгоритмів і технологій штучного інтелекту. Зокрема, це містить застосування нейронних мереж для створення реалістичних ігрових середовищ, що підлаштовуються під дії гравця.

Один із прикладів, де активно розвиваються модифікації до генерації світів — Minecraft. Моди на кшталт Epic Terrain і TerraFirmaCraft демонструють, як можна поліпшити стандартну генерацію світу. Epic Terrain робить генерацію більш реалістичною та різноманітною, а TerraFirmaCraft додає нові елементи до екосистем і складних механік виживання. Такі модифікації вказують на потенційні напрямки розвитку в генерації ігрових структур, де середовища можуть ставати не тільки візуально різноманітними, а й глибше інтегрованими в ігрову механіку.

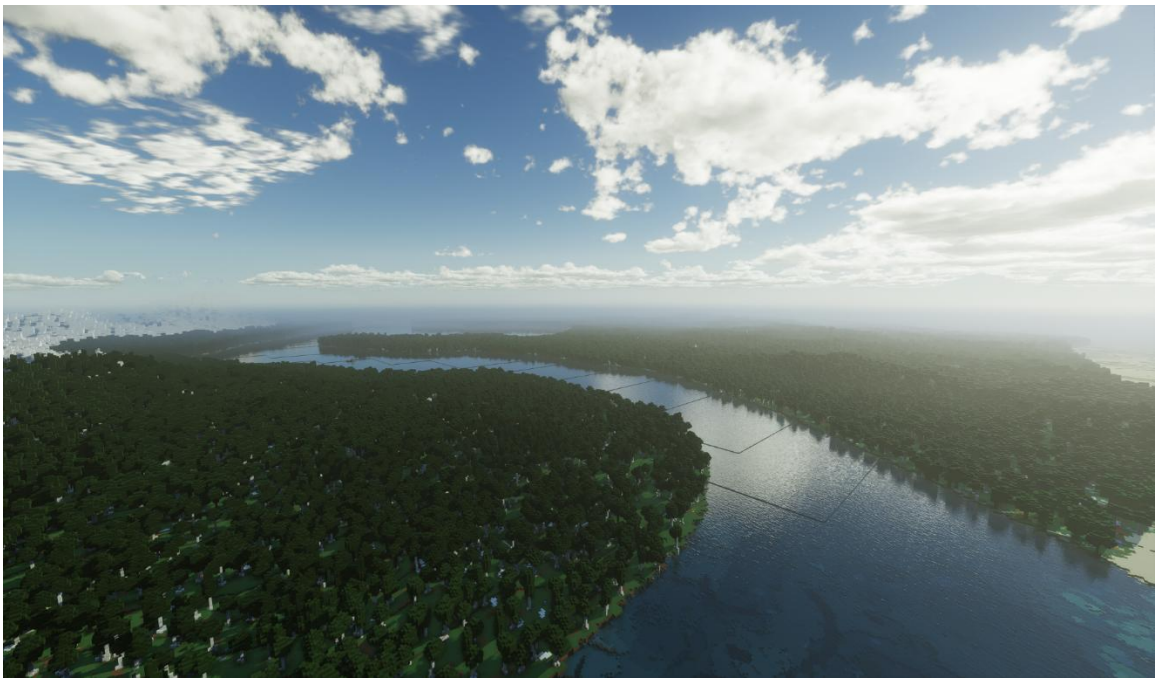


Рис. 2.1 Генерація світу Minecraft з використанням алгоритму Epic Terrain

Ще один напрямок — це створення інтерактивних світів, що реагують на дії гравця. Процедурна генерація, підсилена елементами машинного навчання, дозволяє створювати динамічні світи, які розвиваються разом з персонажем і реагують на його рішення. Наприклад, у RPG-іграх це може бути реалізовано у вигляді змін в архітектурі міст чи ландшафтах після певних подій, наприклад, як це зроблено в модифікаціях для Skyrim, де світ змінюється залежно від рішень гравця. Це може стати основою для нових механік, коли гравець бачить безпосередній результат своїх дій, що підсилює рольову частину гри.

Один із прикладів модифікації, де світ змінюється залежно від дій гравця, — це Holds: The City Overhaul для Skyrim. Ця модифікація розширює міста, додає нові поселення та змінює архітектуру, надаючи кожному місту унікальний вигляд. Світ гри також динамічно реагує на дії гравця через зміну архітектурних елементів і нові інтерактивні елементи, що робить ігровий процес більш захопливим. Наприклад, рішення гравця можуть впливати на події в містах, що може змінювати загальний вигляд цих локацій. Цей приклад ілюструє, як можна інтегрувати динамічну реакцію світу на дії гравця в RPG-іграх, що підсилює рольову частину ігрового процесу.

Окрім того, розвиток генерації у бік більш органічної інтеграції наративних елементів є ще одним перспективним напрямком. У багатьох RPG-іграх важливою частиною є сюжет і його взаємодія з ігровим світом. Створення процедурно згенерованих світів, які можуть динамічно адаптуватися до сюжету, дозволить уникнути одноманітності. В таких системах наратив може впливати на зміну ландшафтів або появу нових структур, що додасть світу більшої інтерактивності.

Наукові дослідження також підтверджують цей потенціал. В статтях, присвячених процедурній генерації в іграх, досліджується можливість використання нейромереж для створення світів, що краще адаптуються до наративу та дій гравців. Одним із таких досліджень є Procedural Content Generation in Games [9], де розглядаються різні підходи до генерації ігрових світів та їх застосування в різних жанрах.

Ця книга охоплює кілька важливих аспектів генерації контенту в іграх. Зокрема, досліджується, як алгоритми можуть адаптувати ігровий світ у реальному часі відповідно до дій гравця, а також як методи машинного навчання можуть вдосконалювати процедуру створення ігрових структур. У книзі описано використання різних видів алгоритмів для створення унікальних ландшафтів, персонажів і сюжетних ліній. Особлива увага приділяється тому, як нейронні мережі можуть вдосконалити цей процес, дозволяючи створювати динамічні й

адаптивні ігрові світи, що реагують на дії гравців. Це допомагає іграм стати менш передбачуваними та підвищує глибину взаємодії.

У книзі Procedural Content Generation in Games розглянуто кілька важливих алгоритмів для генерації ігрового контенту, кожен з яких має свої унікальні особливості. Ось найважливіші з них:

1. L-System (системи Лінденмаєра)
2. Алгоритм клітинних автоматів (Cellular Automata)
3. Wave Function Collapse (WFC)
4. Генетичні алгоритми

2.2.1 L System (системи Лінденмаєра)

L-системи (системи Лінденмаєра) [10] — це формальні граматики, запропоновані біологом Арістідом Лінденмаєром у 1968 році для моделювання росту рослин, проте пізніше вони знайшли застосування в комп'ютерній графіці для генерації фрактальних структур, дерев, рослин та інших природних об'єктів. Ці системи використовують прості правила для створення складних, часто фрактальних структур через процес ітерації.

Основні компоненти L-системи:

1. Алфавіт: набір символів, які представляють будівельні блоки або елементи системи. Наприклад, це можуть бути символи A, B, F, +, -, які позначають різні операції або частини структури.
2. Аксіома: початкова послідовність символів, з якої починається генерація. Це своєрідна "зернина", яка через ітерації розростається у складнішу структуру.
3. Правила переписування (продукції): це набір правил, які визначають, як кожен символ замінюється на нову послідовність символів. Ці правила застосовуються до кожного символу послідовності одночасно під час кожної ітерації

4. Ітерації: кожен символ у поточній послідовності замінюється відповідно до правил продукції, після чого цей процес повторюється кілька разів для досягнення складніших структур.

$$\mathrm{A} \rightarrow AB \tag{2.1}$$

$$B \rightarrow A \tag{2.2}$$

Кожна ітерація повторює ці правила для заміни символів, створюючи все складніші структури. Наприклад, для початкового символу A послідовні ітерації будуть виглядати так:

$$\mathrm{A} \rightarrow AB \tag{2.3}$$

$$\mathrm{A}B \rightarrow AB\mathrm{A} \tag{2.4}$$

$$\mathrm{A}BA \rightarrow AB\mathrm{AAB} \tag{2.5}$$

$$AB\mathrm{AAB} \rightarrow ABAABABA \tag{2.6}$$

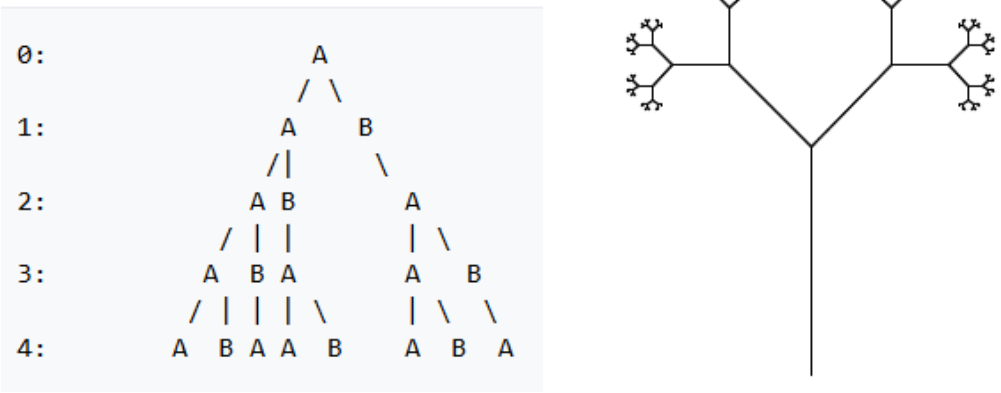


Рис 2.2 Візуальне зображення системи Ліденмаєра

Таким чином, при кожній ітерації структура ускладнюється, що дозволяє створювати фрактальні та самоорганізовані патерни.

Для створення графічних зображень за допомогою L-систем, наприклад, дерев або рослин, часто використовують додаткові символи, які позначають поворот напрямку або рух вперед (черепашкова графіка). Символи можуть мати такі значення:

F: рух уперед на певну відстань.

+: поворот вліво на певний кут.

-: поворот вправо на певний кут.

[: збереження поточного положення та напрямку (початок нової гілки).

]: повернення до збереженого положення (кінець гілки).

Ця система малює класичний фрактал — Крива Коха. Кожен відрізок замінюється на чотири нових, причому два з них створюють кут, що збільшує складність структури з кожною ітерацією.

L-системи можуть використовуватися для процедурної генерації ландшафтів або структур у відеоіграх. Наприклад, вони допомагають автоматично створювати дерева, кущі та інші природні об'єкти, які виглядають природно, але є різними на кожній ітерації або для кожної нової локації у грі.

L-системи — це потужний математичний інструмент для моделювання органічного росту та створення фрактальних структур. Їх проста логіка переписування символів дозволяє генерувати складні візуальні форми на основі простих правил, що робить їх популярними в комп'ютерній графіці та процедурній генерації в іграх.

2.2.2 Алгоритм клітинних автоматів (Cellular Automata)

Клітинні автомати (Cellular Automata) — це дискретні математичні моделі, які використовуються для моделювання систем, що змінюються з часом за певними правилами. Вперше цей концепт був введений в 1940-х роках Джоном фон Нейманом і Станіславом Уламом, але найбільшої популярності клітинні автомати набули завдяки роботі Джона Конвея і його "Грі життя" (Game of Life). Цей метод знайшов застосування в різних галузях, включаючи фізику, біологію, комп'ютерну науку, а також процедурну генерацію в ігровій індустрії.

Основні компоненти клітинних автоматів:

1. Сітка (Ґратка): Клітинні автомати складаються з дискретної сітки клітин, яка може бути одновимірною, двовимірною або навіть багатовимірною.

Найпоширенішими є одновимірні та двовимірні сітки. Одновимірна сітка являє собою лінію з клітин, кожна з яких може перебувати в одному з декількох можливих станів, а двовимірна сітка — це площина, де кожна клітина має вісім сусідів (по горизонталі, вертикалі та діагоналі).

2. Стани клітин: Кожна клітина на сітці може знаходитися в одному з обмеженого числа можливих станів. Наприклад, в "Грі життя" Конвея клітини можуть бути "живими" або "мертвими", що позначається двома станами: чорний (жива) або біла (мертва).
3. Правила переходу: Стан кожної клітини змінюється відповідно до правил переходу, які залежать від поточного стану клітини та станів її сусідів. Правила визначають, як клітини взаємодіють між собою і як зміни в станах клітин викликають еволюцію всієї системи. Наприклад, у "Грі життя" Конвея правила переходу виглядають так: Якщо у живої клітини є менше ніж два або понад трьох живих сусідів, вона помирає (симуляція перенаселення або самотності). Якщо у мертвої клітини є рівно три живих сусіди, вона "оживає".
4. Час (ітерації): Еволюція клітинного автомата відбувається в дискретних кроках часу (ітераціях). На кожному кроці застосовуються правила переходу для кожної клітини сітки одночасно. Це призводить до змін у структурі всієї системи з кожним наступним кроком.

Розглянемо "Гру життя" Джона Конвея, яку було згадано раніше. У ній клітини розташовані на двовимірній сітці, і кожна клітина може бути або живою (чорна), або мертвою (біла). Правила дуже прості, але дозволяють створювати надзвичайно складні структури, що еволюціонують з часом.

Приклад початкової конфігурації (стартової сітки А) з кількох живих клітин:

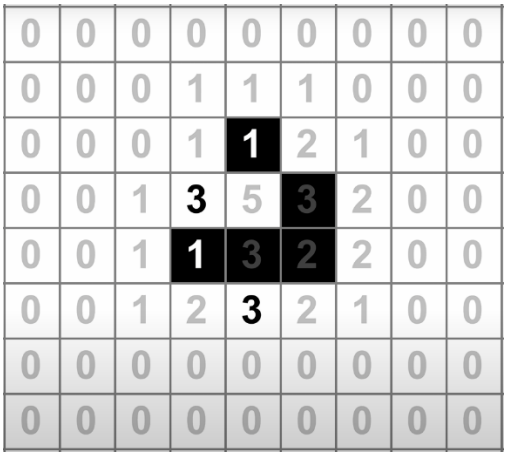


Рис 2.3 Створення сітка A

Ця конфігурація в наступній ітерації може змінитися, залежно від того, скільки сусідів мають живі клітини, а скільки мертвих клітин оточують кожну з них. Після кількох ітерацій система може або стабілізуватися (стати нерухомою), або вийти в нескінченну еволюцію.

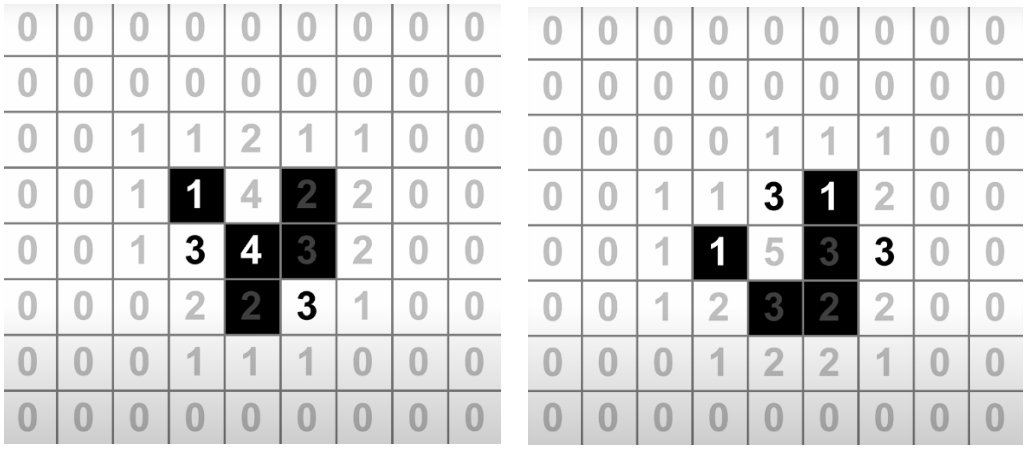


Рис 2.4 Перша і друга зміна сітки A

Таким чином, "Гра життя" здатна створювати різноманітні складні структури, включаючи осцилятори (структури, що циклічно повторюються після певної кількості ітерацій), глайдери (рухомі фігури), та стабільні конфігурації, які з часом не змінюються. Завдяки таким базовим взаємодіям, ця система стає чудовим прикладом того, як прості правила можуть генерувати неймовірно складну та різноманітну поведінку.

Існує безліч варіацій клітинних автоматів, створених на основі різних правил і алгоритмів. Наприклад, одновимірний клітинний автомат "Правило 30", яке генерує хаотичні шаблони з простих початкових умов, використовується в генерації псевдовипадкових чисел та шифруванні даних. Ці правила та варіації клітинних автоматів дозволяють моделювати не лише фізичні процеси, а й біологічні та екологічні системи, а також створювати процедурно згенеровані світи у відеоіграх.

Одним із прикладів біологічної системи, яка моделюється за допомогою клітинного автомата, є модель Ленгтона, відома як "Мураха Ленгтона". Це приклад клітинного автомата, який демонструє складну поведінку з простими правилами, але в контексті біологічних та екологічних систем він може бути використаний для моделювання руху організмів або колоній у пошуках їжі або ресурсів.

Правила цієї моделі такі: кожна клітина може бути або чорною, або білою. "Мураха" рухається по сітці клітин і змінює напрямок на 90 градусів (вліво або вправо) залежно від кольору клітини, на якій вона стоїть. Після цього вона змінює колір цієї клітини (з чорного на білий або з білого на чорний) і пересувається вперед. Здавалося б, проста система поступово може створювати складні, організовані структури, які нагадують певні форми життєдіяльності колоніальних організмів, що рухаються в пошуках ресурсів.

Ця модель, хоч і є спрощеною, дозволяє досліджувати, як прості правила можуть призводити до виникнення складної поведінки. В екологічному контексті вона може бути використана для вивчення поведінки комах, бактерій або інших мікроорганізмів у середовищі, де є обмежені ресурси.

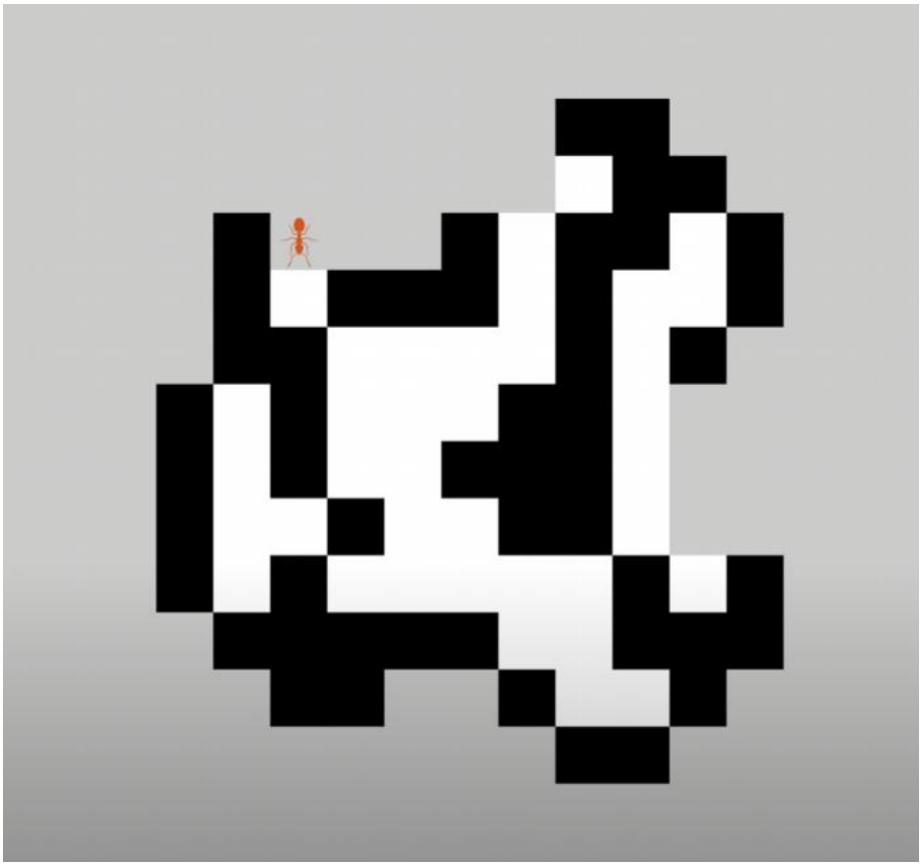


Рис 2.5 Приклад моделі Ленгтона

Також існують Одновимірні клітинні автомати, які працюють за схожим принципом, але мають лише один ряд клітин. Стан кожної клітини залежить лише від самої себе та двох сусідів (зліва та справа). Одновимірні автомати зазвичай використовуються для генерації псевдовипадкових чисел або для симуляції простих систем. Одним із найвідоміших прикладів одновимірних клітинних автоматів є автомат Rule 30, де кожна клітина змінюється на основі простого правила: її новий стан залежить від поточного стану сусідів і її самої.

Усі ці правила застосовуються до кожної клітини, що призводить до утворення складних патернів у процесі еволюції системи.

Застосування клітинних автоматів:

1. Фізичні симуляції: клітинні автомати використовуються для моделювання різних фізичних процесів, таких як дифузія, хвильові процеси та ріст кристалів.

2. Генерація ландшафтів: у процедурній генерації клітинні автомати можуть використовуватися для створення природних форм, таких як рельєф, океанські хвилі або структури міст.
3. Симуляція біологічних систем: моделі клітинних автоматів використовуються для симуляції росту популяцій, еволюційних процесів, захворювань та інших біологічних явищ.
4. Генерація ігрового контенту: клітинні автомати часто застосовуються для створення процедурних світів у відеоіграх, таких як лабіринти, планети або ландшафти.

Клітинні автомати — це потужний інструмент для моделювання складних систем, де взаємодії між простими елементами можуть призвести до виникнення складної поведінки та структури. Завдяки своїй простоті й гнучкості, клітинні автомати знайшли застосування в багатьох галузях науки та техніки, від комп'ютерних симуляцій до процедурної генерації контенту для відеоігор.

2.2.3 Wave Function Collapse (WFC)

Wave Function Collapse (WFC) [11] — це алгоритм генерації контенту, який використовується для створення складних візуальних або структурних шаблонів на основі простого набору правил та вхідних зразків. Алгоритм був створений програмістом Максимом Ґуммелем і став популярним завдяки своїй здатності процедурно генерувати як двовимірні, так і тривимірні структури, зберігаючи при цьому когерентність і стиль вихідного шаблону. WFC застосовується в різних областях, таких як відеоігри, генерація текстур та створення рівнів.

Основні компоненти алгоритму WFC:

1. Вхідний зразок: це вихідна сітка, яка може бути зображенням або набором об’єктів, що містить певні шаблони. Наприклад, це може бути піксельна картинка або карта місцевості, на основі якої буде генеруватися нова структура.

2. Клітинна сітка: WFC працює на сітці клітин (подібно до клітинних автоматів), де кожна клітина може містити певний можливий стан або варіант (набір значень або шаблонів).
3. Правила сумісності: кожна клітина повинна відповідати правилам, які визначають, як різні сусідні клітини можуть співіснувати. Ці правила забезпечують, щоб генерована структура не порушувала логіку зразка. Наприклад, якщо в зразку певні клітини мають прямі лінії, то генерована карта також повинна дотримуватися таких шаблонів.
4. Колапс хвильової функції: це основний етап алгоритму, де кожній клітині призначається один зі станів на основі правил імовірностей. Клітина "колапсує" в один із можливих варіантів, а це призводить до того, що інші клітини в сітці повинні адаптуватися, дотримуючись правил сумісності.
5. Розв'язок обмежень: алгоритм поступово зменшує кількість можливих варіантів для кожної клітини на основі контексту, до того, як уся сітка заповниться узгодженими клітинами.

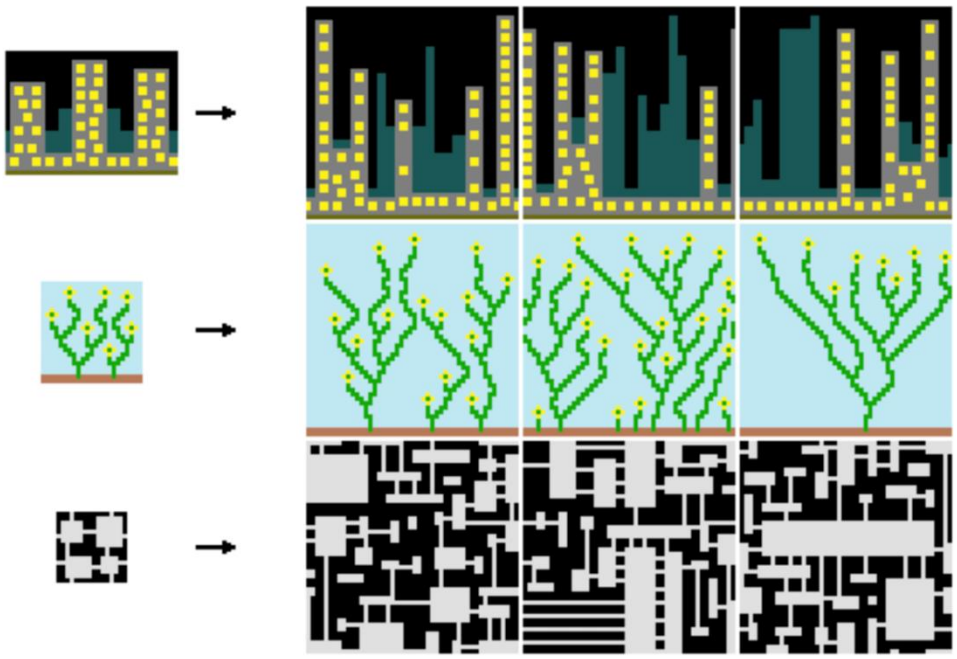


Рис 2.6 Приклад роботи WFC

Процес роботи:

1. Спочатку алгоритм отримує вхідний зразок і генерує сітку, на якій буде виконуватись процедура WFC, окрім цього кожна клітина містить набір усіх можливих варіантів (станів), які згодом будуть зведені до одного.
2. На кожному кроці алгоритм обирає клітину з найменшою кількістю можливих варіантів (максимально невизначену) і колапсує її, обравши один варіант. Після цього, сусідні клітини адаптуються, щоб дотримуватися правил сумісності.
3. Процес повторюється доти, доки вся сітка не буде заповнена.

Генерація рівнів за допомогою Wave Function Collapse (WFC) стала одним із найпотужніших методів процедурної генерації контенту, особливо в відеоіграх. Цей алгоритм дозволяє автоматично створювати складні та узгоджені рівні на основі набору простих правил і вхідних даних.

Алгоритм WFC використовується в багатьох галузях, зокрема:

1. Процедурна генерація рівнів у відеоіграх: WFC дозволяє автоматично створювати карти та світи, дотримуючись при цьому заданих стилістичних або логічних обмежень, наприклад, для генерування лабіринтів, підземель або міст.
2. Текстури та піксельна графіка: WFC може використовуватися для створення текстур з заданих зразків, забезпечуючи плавні переходи між різними елементами текстури, що робить їх природними на вигляд.
3. Архітектурні рішення: завдяки здатності генерувати узгоджені шаблони, WFC застосовується для проєктування міських ландшафтів або будівель, зберігаючи архітектурний стиль вихідного шаблону.

Алгоритм Wave Function Collapse (WFC) не має однозначної математичної формули, як у випадку з класичними алгоритмами, але він базується на концепціях з теорії інформації та ймовірнісних процесів. Однак, деякі ключові математичні ідеї та принципи, що використовуються в алгоритмі, можна представити так Ентропію та ймовірності:

Ентропія: У кожній клітині сітки існує множина можливих станів. Під час генерації ми намагаємося вибрати стан для кожної клітини, який мінімізує ентропію (невизначеність). Чим менше можливих станів для клітини, тим менша її ентропія. Та ймовірності, де в кожній клітині ймовірності можливих станів можуть визначатися на основі частоти цих станів у вихідному шаблоні.

Ймовірності: В кожній клітині ймовірності можливих станів можуть визначатися на основі частоти цих станів у вихідному шаблоні.

Для кожної клітини c_i, яка має множину можливих станів S_i, ентропія клітини визначається за формулою Шеннона:

$$H(c_i) = -\sum_{8 \in S_i} P(s) \log P(s) \quad (2.7)$$

Де $P(s)$ — це ймовірність того, що клітина c_i перебуватиме в стані s.

Для клітини з мінімальною ентропією вибирається стан s. із множини можливих станів S_i на основі ймовірностей $P(s)$. Це можна представити як операцію вибору аргументу максимуму для випадкової змінної s:

$$s = argmax_s\, P(s) \quad (2.8)$$

Для кожної клітини накладаються обмеження на основі станів її сусідів. Набір можливих станів для клітини c_i після вибору стану сусідньої клітини c_j стає підмножиною вихідної множини станів S_i. Це можна описати функцією оновлення:

$$S_i' = S_i \cap R(c_j) \quad (2.9)$$

Де $R(c_j)$ — це множина дозволених станів для сусідньої клітини c_j.

Оновлення сітки: Після того, як колапс відбувся в одній клітині, алгоритм оновлює можливі стани для всіх її сусідів, зменшуючи їх множини можливих станів на основі правил сумісності. Це призводить до того, що ентропія клітин-сусідів зменшується.

Основні етапи роботи алгоритму:

1. Вибір клітини з мінімальною ентропією: min_{c_i} H(c_i).
2. Вибір стану для цієї клітини на основі ймовірностей: $s = argmax_s\, P(s)$
3. Оновлення сусідів на основі правил сумісності.

4. Повторення процесу для інших клітин до заповнення всієї сітки.

Приклад простої формалізації WFC: Нехай сітка складається з $N \times N$ клітин, кожна з яких може приймати одне з $\underline{k}$ можливих значень. Для кожної клітини ми маємо множину можливих станів:

$$S_i \subseteq \{1, \dots, k\} \tag{2.10}$$

Процес WFC зводиться до таких кроків:

1. Ініціалізувати множини можливих станів для кожної клітини.
2. На кожній ітерації вибрати клітину з мінімальною кількістю можливих станів (мінімум ентропії).
3. Вибрати стан для цієї клітини випадковим чином з урахуванням ймовірностей.
4. Оновити множини можливих станів для сусідів вибраної клітини.
5. Повторювати, поки всі клітини не матимуть єдиний можливий стан.

Таким чином, алгоритм Wave Function Collapse не є традиційним формальним рівнянням [12], а скоріше використовує імовірнісну модель для поступового зменшення ентропії в сітці та побудови узгодженої структури на основі початкового шаблону та правил сумісності.

Переваги використання WFC для генерації рівнів:

1. Узгодженість і стилістична цілісність: Алгоритм WFC генерує рівні, які виглядають узгодженими та відповідають стилю вхідного зразка. Це особливо важливо в іграх, де необхідно зберегти логіку світу (наприклад, щоб стіни з'єднувалися правильно або щоб різні кімнати мали правильні переходи).
2. Варіативність: Хоча алгоритм базується на вхідному зразку, кожен новий рівень унікальний. Це дозволяє створювати багато варіантів рівнів, зберігаючи загальну стилістику та правила, але при цьому уникаючи повторень.

3. Оптимізація генерації: WFC не генерує рівень відразу цілим шматком. Завдяки поступовому колапсу клітин і правилам сумісності, результатом є логічно побудована карта без необхідності ручного втручання.
4. Різноманітні застосування: WFC можна використовувати не тільки для 2D ігор, але й для 3D світів. Наприклад, можна генерувати міста, лабіринти, приміщення та інші типи середовищ, де важливо дотримуватися певних топологічних правил..

Застосування WFC у відеоіграх:

"Bad North" — стратегічна гра, в якій WFC використовується для процедурної генерації карт островів.

"The Last Clockwinder" — гра, в якій WFC допомагає генерувати складні тривимірні світи та архітектуру.

Отже, головною перевагою WFC в ігровій індустрії є здатність створювати унікальні рівні з дотриманням загальних правил і стилю, що дає змогу підтримувати високу варіативність ігрового контенту. Це потужний алгоритм процедурної генерації, який поєднує елементи з теорії інформації та комп'ютерної графіки для створення складних і різноманітних візуальних або логічних структур на основі заданих вхідних даних. Завдяки використанню принципів супротивності (ентропії) та правилам сусідства, WFC здатний створювати візуально привабливі й структурно послідовні шаблони, які можуть мати широкий спектр застосувань, від дизайну рівнів у відеоіграх до створення текстур та архітектурних планів.

Алгоритм WFC показує, як можна застосувати математичні концепції для генерації різноманітного і водночас контрольованого контенту, що робить його незамінним інструментом у сучасних процедурах генерації.

2.3 Аналіз та можливі покращення методу генерації ігрового світу та структур найвідомішої RPG гри

Одна з найвідоміших ігор з процедурної генерацією світу є гра minecraft, яку було раніше згадано. Генерація світу в Minecraft — це складний процес, який базується на використанні декількох алгоритмів, кожен з яких відповідає за різні аспекти створення ігрового світу. Основні компоненти, які впливають на генерацію, включають створення рельєфу, розподіл біомів, розміщення структур (таких як села, печери, мінорні та мажорні структури), а також рослинність і мінерали.

Основою для генерації рельєфу в Minecraft є алгоритми шумів. Найпоширенішим алгоритмом є шум Перліна, який використовується для створення плавних переходів висот. Він працює за принципом накладення декількох шарів шуму, кожен з яких має різну частоту та амплітуду, що дозволяє створювати детальний рельєф. Окрім цього, використовуються Octaves (октави) — це термін, що означає кількість шарів шуму, кожен з яких додає більш детальні зміни до рельєфу. Менші октави додають деталі, такі як пагорби, тоді як більші октави відповідають за основний ландшафт, наприклад, великі гори або долини.

Розподіл біомів здійснюється на основі системи, схожої на Перліновий шум, але з додатковим урахуванням таких параметрів, як температура, вологість і висота. Це дозволяє створювати логічні переходи між біомами, де, наприклад, пустеля може переходити в савану, але не безпосередньо в тайгу. Модифікація генерації біомів може включати заміну наявних алгоритмів або додавання нових біомів зі специфічними характеристиками. Наприклад, можна створити біом на основі реальних екосистем, таких як амазонські джунглі або альпійські гори, з відповідною флорою, фауною та рельєфом.

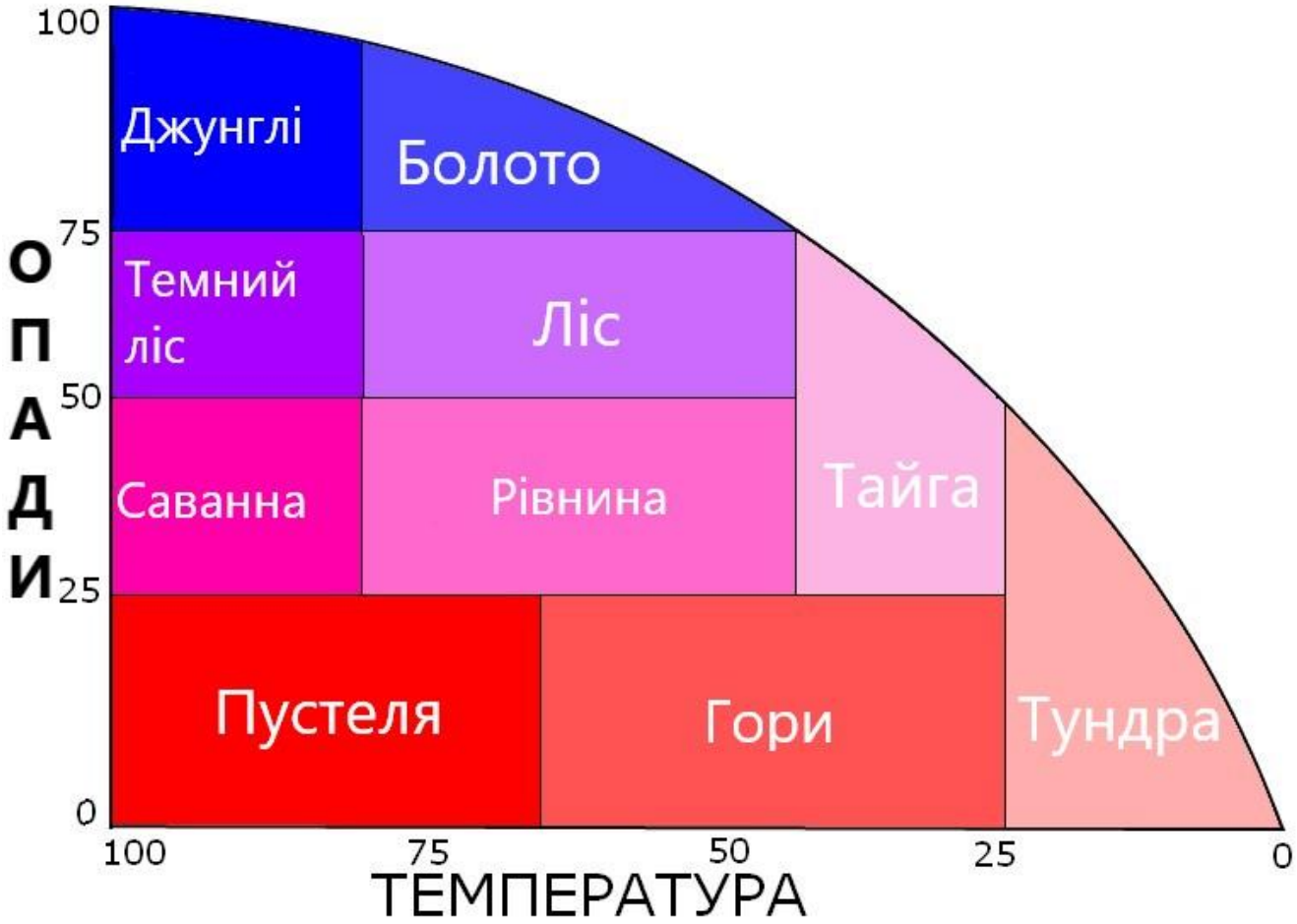


Рис 2.7 схема розподілу біомів у Minecraft

Генерація структур в Minecraft здійснюється з використанням детермінованих шаблонів, які можуть бути змінені або створені з нуля. Наприклад, такі структури, як села, фортеці, і навіть менш значні об'єкти, як дерева, мають визначені шаблони, які розташовуються у світі на основі певних правил. При використанні WFC (Wave Function Collapse) для генерації структур можна досягти більш складної та різноманітної архітектури. Цей алгоритм працює на основі принципу "розпаду хвильової функції", де кожен новий блок або секція будівлі генерується на основі попередніх, враховуючи обмеження і зв'язки з сусідніми елементами.

Реалізацію алгоритму Wave Function Collapse (WFC) в Minecraft дозволяють створювати структури, які виглядають більш органічно і природно. Наприклад,

замість детермінованих прямокутних будівель можна отримати структурно складні замки або міста, що мають безліч унікальних архітектурних елементів.

За допомогою налаштування параметрів шумів можна створити більш реалістичний рельєф. Це може включати детальні гори, долини та рівнини, які будуть відповідати реальним ландшафтам. Наприклад, для генерації гірського масиву можна використовувати декілька шарів шуму з різними октавами, що дозволить створити різноманітні схили й вершини.

Щоб користувач міг вибирати біоми на основі реальних локацій, потрібно створити набір параметрів для кожного біому. Це можуть бути нові комбінації висоти, температури й вологості, а також особливі рослини, тварини та мінерали, притаманні цим біомам.

Для генерації структур, таких як міста або підземелля, можна використовувати WFC. Це дозволить створити неповторні архітектурні ансамблі, де кожна структура буде мати свою унікальну форму і розташування елементів. Це зробить світ більш різноманітним і цікавим для дослідження.

Використовуючи описані алгоритми та техніки, можна створити модифікацію для Minecraft, яка перетворить генерацію світу, зробивши її більш реалістичною й адаптованою до бажань гравця. Ця модифікація дозволить створювати світи, які будуть відображати реальні біоми та структури, що робить гру ще більш захопливою та індивідуальною.

Таблиця 2.1

Порівняльна характеристика алгоритмів генерації ігрових мап

Критерій	**L-System**	**Клітинні автомати**	**Wave Function Collapse (WFC)**
Призначення	Моделювання органічного росту, створення дерев, рослин.	Моделювання фізичних, біологічних, екологічних систем.	Генерація складних візуальних і структурних шаблонів.
Гнучкість	Менш універсальний, підходить для природних структур.	Дуже гнучкий, підходить для моделювання різних систем.	Універсальний, орієнтований на структуровану генерацію.
Простота реалізації	Простий у реалізації, але обмежений для інших цілей.	Вимагає налаштування правил, середня складність.	Складний, потребує правил сумісності й оптимізації.
Динаміка роботи	Статичний (ітеративне ускладнення структури).	Динамічний (зміна стану клітин із часом).	Статичний (генерація завершеної сітки за зразком).
Застосування	Генерація дерев, фракталів, рослин.	Створення ландшафтів, екосистем, лабіринтів.	Процедурна генерація карт, текстур, рівнів.
Вимоги до ресурсів	Мінімальні ресурси.	Середні ресурси, залежить від розміру сітки.	Високі ресурси для великих сіток та складних правил.
Рівень реалізму	Високий для органічних структур.	Залежить від налаштувань правил, підходить для абстрактних моделей.	Високий за умови якісного зразка.

Для роботи було обрано метод клітинних автоматів, тому що він має низку переваг, які роблять його оптимальним вибором для вирішення завдань процедурної генерації:

Простота реалізації: алгоритми клітинних автоматів легко імплементувати та налаштовувати під конкретні потреби. Вони не потребують складних математичних обчислень, що прискорює процес розробки.

Гнучкість і універсальність: клітинні автомати дозволяють моделювати широкий спектр явищ — від генерації ландшафтів і структури рівнів до симуляції фізичних і біологічних процесів.

Складна поведінка з простих правил: навіть при мінімальному наборі правил клітинні автомати здатні генерувати складні та реалістичні структури. Це особливо важливо для створення природних елементів ігрового світу, таких як рельєфи, лабіринти чи екосистеми.

Процедурність і унікальність результатів: завдяки випадковим стартовим умовам і варіативності правил, кожна сесія генерації дає унікальний результат. Це дозволяє створювати різноманітний контент без значних витрат на його ручну розробку.

Локальність обчислень: зміна стану кожної клітини залежить лише від її найближчих сусідів, що спрощує паралельну обробку і підвищує продуктивність на великих сітках.

Ці характеристики роблять клітинні автомати ідеальним вибором для завдань, де потрібно балансувати між складністю, якістю результату та швидкістю виконання.

Окрім цього, клітинні автомати було порівняно ще з такими відомими методами, як «Рекурсивний поділ простору» [13], «Діаграми Вороного» [14] та «Алгоритми на основі графів» [15], протягом написання наукової статті: «Застосування методів генерації ігрових структур для гри з відкритим всесвітом у жанрі RPG». Результати цієї роботи у таблиці 2.2:

Аналізуючи різноманітні підходи до створення ігрових світів, можна побачити, що кожен алгоритм має свої переваги та недоліки. Наприклад, BSP (Binary Space Partitioning) [16] вирізняється простотою реалізації та здатністю створювати структуровані світи, тоді як графові алгоритми забезпечують більше можливостей для створення зв’язних і гнучких локацій. Проте в кожному з підходів існують обмеження: складність BSP зростає при роботі з великими світами, а при використанні діаграм Вороного виникають труднощі з однорідністю границь.

Вибір конкретного алгоритму залежить від потреб розробки, продуктивності системи, вимог до візуального стилю та особливостей створюваного ігрового світу. Розглянемо основні особливості популярних методів:

BSP (Binary Space Partitioning): Цей підхід приваблює простотою реалізації, що робить його зручним для початківців. Він дозволяє створювати зв’язні та чітко структуровані світи. Недоліком є ускладнення реалізації при збільшенні масштабу світу та менш природний вигляд локацій.

Cellular Automata [17]: Цей метод забезпечує гнучкість і здатність адаптуватися до різних умов. Він підходить для створення складних світів з багатогранними взаємодіями між елементами. Однак для досягнення якісного результату потрібні додаткові обчислення та налаштування.

Графові алгоритми [18]: Вони надають більше можливостей для створення зв’язних, структурованих ігрових світів із різноманітними локаціями. Ефективні для великих проєктів, особливо за умов оптимізації графа.

Діаграми Вороного: Ідеальні для генерації ландшафтів і розміщення ресурсів. Вони швидко генеруються і створюють природні границі. Водночас потребують значних ресурсів і не завжди забезпечують органічний вигляд світів.

Для детального порівняння алгоритмів і їхніх особливостей можна звернутися до таблиці 2.2, де наведено ключові характеристики кожного методу.

Таблиця 2.2

Результат порівняння методу клітинних автоматів
з іншими відомими методами

Методи	**Рекурсивний поділ простору**	**Діаграми Вороного**	**Алгоритми на основі графів**	**Клітинні автомати**
Основний принцип роботи	Рекурсивне поділення простору на підрегіони	Поділ простору на ділянки за найближчою точкою	Графи як вузли та з'єднання	Клітинна симуляція за правилами сусідства
Складність реалізації	Низька: Простий алгоритм без складних обчислень.	Висока: Складна математика для побудови й оптимізації країв.	Висока: Потрібно будувати графи, налаштовувати пошук шляхів.	Середня: Простий код, але важке налаштування правил.
Гнучкість	Низька: Придатний лише для регулярних форм	Висока: Придатний для природних зон	Висока: Контроль структури та деталей	Висока: Легко змінювати правила
Використання в 3D	Підходить для будівель, кімнат	Добре для біомів, водойм	Легке створення складних структур	Добре підходить для печер, рельєфів, ландшафту
Переваги	Легко реалізувати	Генерація реалістичних зон	Придатний для лабіринтів, доріг	Простота, природний вигляд, безліч можливостей налаштування
Недоліки	Обмежений для природних середовищ	Складна обробка країв	Складний у налаштуванні	Складний у налаштуванні

Клітинні автомати мають середній рівень складності реалізації та високу гнучкість у налаштуванні, що робить їх зручними для створення ландшафтів і природних середовищ. Вони забезпечують простоту реалізації правил, але можуть бути складними в налаштуванні, порівняно з іншими методами, такими як діаграми Вороного чи алгоритми на основі графів.

3. РОЗРОБКА МОДЕЛІ ВИПАДКОВОЇ ГЕНЕРАЦІЇ СВІТУ ТА СТРУКТУР ДЛЯ ГРИ В ЖАНРІ RPG

3.1 Вибір програмних засобів реалізації

Для розробки алгоритму процедурної генерації ігрових світів і структур у рамках цієї роботи було використано сучасні інструменти, які забезпечили високу продуктивність і гнучкість у реалізації проєкту. Нижче наведено детальний опис основних програмних засобів, які були залучені.

1. Unity — це одна з найпопулярніших платформ для створення ігор, яка забезпечує зручний інструментарій для реалізації ігрових механік, побудови 2D і 3D-середовищ та налагодження процедурних алгоритмів. Для цієї роботи Unity обрано через кілька ключових переваг:

Можливість процедурної генерації: Unity має вбудовані інструменти для роботи зі спрайтами, сітками (Meshes) і рельєфами, які ідеально підходять для реалізації алгоритмів генерації ігрових світів.

Підтримка C#: Unity працює з мовою програмування C#, яка дозволяє ефективно реалізовувати складні алгоритми.

Швидке прототипування: Завдяки вбудованим компонентам, інтуїтивному інтерфейсу та багатій документації Unity дозволяє швидко створювати й тестувати нові ідеї.

Підтримка 2D та 3D: У проєкті спочатку реалізовано 2D-карту, яка згодом перетворюється в 3D-середовище. Unity надає інструменти для плавного переходу між цими форматами.

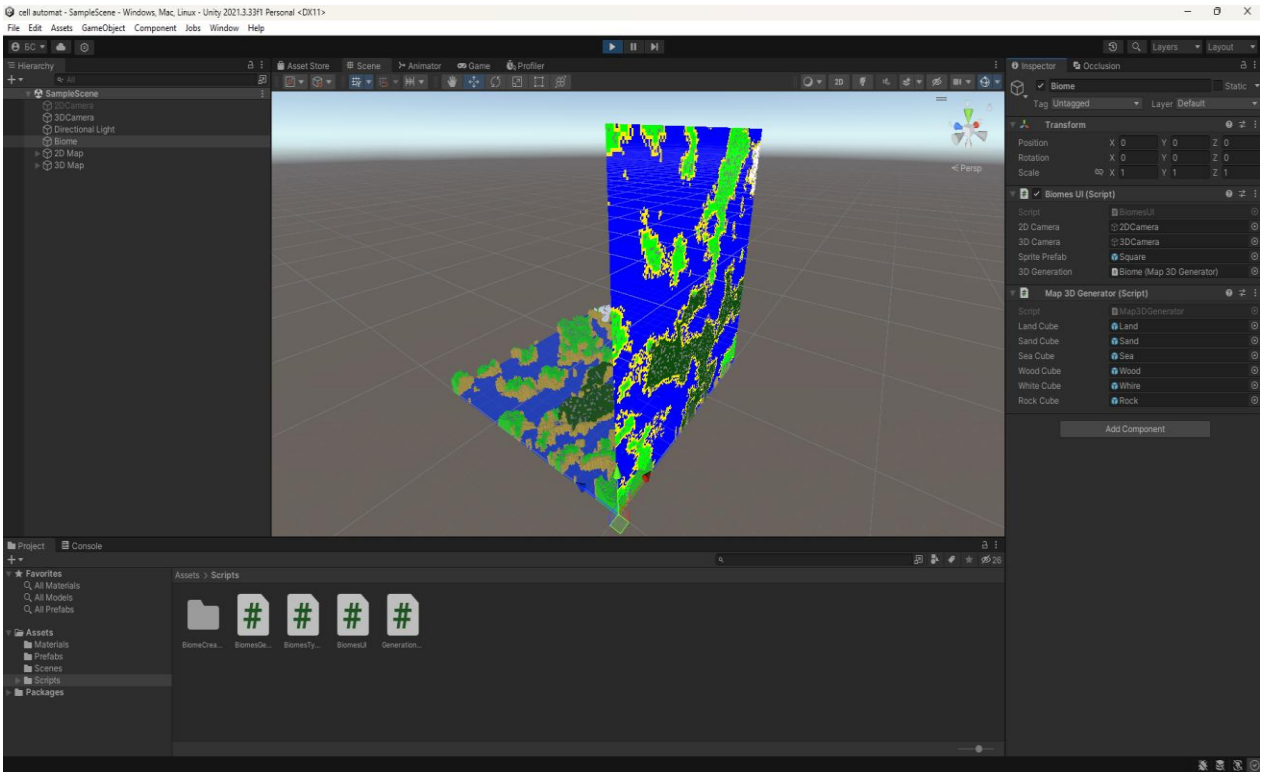


Рис. 3.1 Демонстрація ігрового рушія Unity

2. Visual Studio була використана як основне інтегроване середовище розробки для написання коду мовою C#. Її обрання обумовлене широкими можливостями для створення, налагодження та оптимізації програмного забезпечення, зокрема для проєктів, пов'язаних з Unity. Середовище забезпечує зручність роботи завдяки підтримці інтелектуального автозаповнення (IntelliSense), що суттєво пришвидшує написання коду та зменшує кількість помилок. Вбудований потужний дебагер допоміг швидко знаходити та виправляти помилки в алгоритмах процедурної генерації, що є критично важливим для таких проєктів.

Крім цього, Visual Studio має інтеграцію з Unity, що дозволило безперебійно синхронізувати розробку коду з ігровим рушієм, включаючи миттєве відображення змін. Можливість роботи з системами контролю версій, такими як Git, дозволила ефективно організувати роботу над проєктом і зберігати історію змін, що

спрощувало повернення до попередніх етапів розробки. Visual Studio зарекомендувала себе як зручний, надійний і функціональний інструмент, який суттєво прискорив реалізацію всіх етапів розробки.

3. Мова програмування C# стала основним засобом написання логіки генерації, оскільки вона є стандартною мовою програмування для Unity і чудово підходить для роботи з об'єктноорієнтованими концепціями.

4. Golly — це спеціалізована програма для моделювання клітинних автоматів. Хоча вона не використовувалася як основний інструмент для розробки, Golly була корисною для прототипування та вивчення поведінки клітинних автоматів, які використовувалися в роботі. Golly надає можливість візуально перевіряти, як змінюється стан клітин у процесі виконання правил. Це допомогло вдосконалити логіку клітинних автоматів, що використовувались для генерації. За допомогою Golly було проведено попередні тести на різних правилах клітинних автоматів, що дозволило вибрати оптимальний набір правил для алгоритму генерації. Програма надихнула на розробку унікальних підходів до моделювання структур.

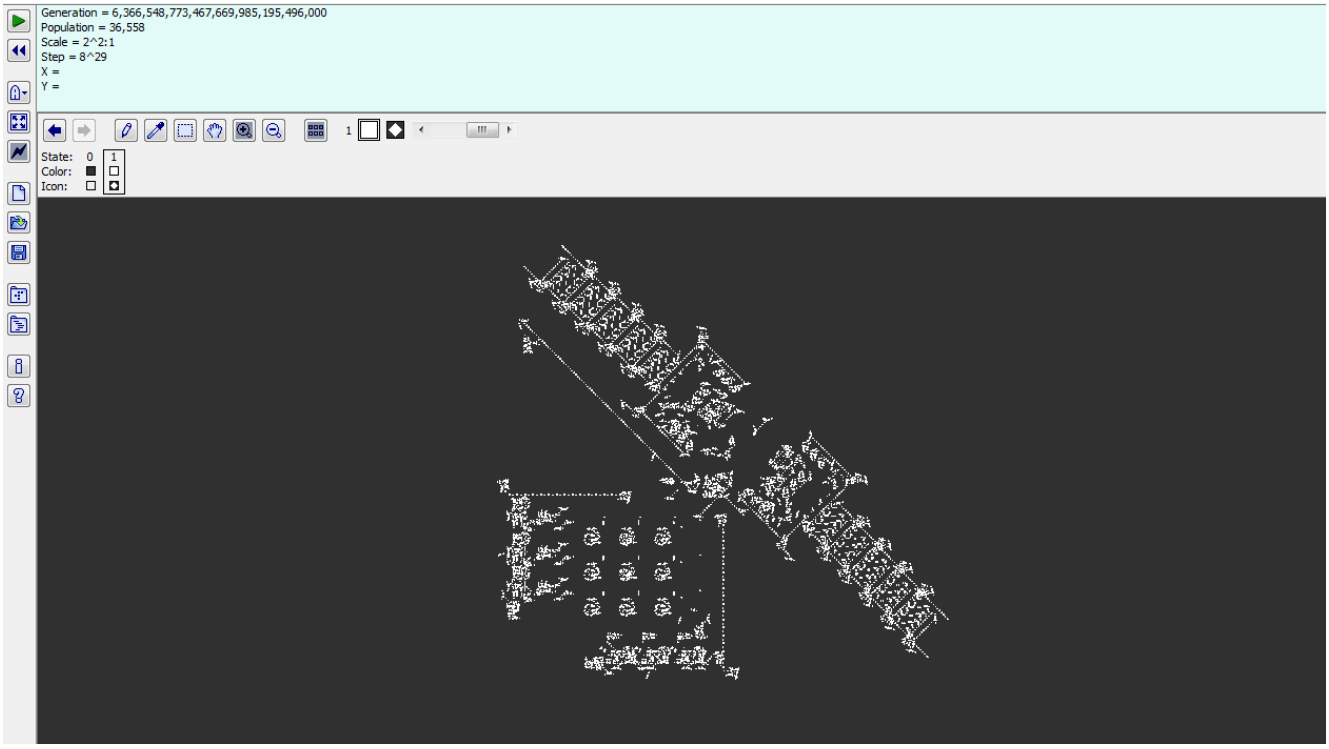


Рис 3.2 Демонстрація програми для генерації клітин «Golly»

3.2 Математична модель генерації ігрового світу

Математична модель, представлена у цьому проєкті, слугує основою для автоматизованої генерації ігрового світу та його структур, дозволяючи створювати багатокомпонентні ігрові простори за допомогою обчислювальних алгоритмів. Вона забезпечує формалізацію процесів генерації, враховуючи як базові параметри світу, так і специфічні особливості, такі як топографія, розподіл ресурсів і зонування.

Модель покликана реалізувати процедурну генерацію світу таким чином, щоб забезпечити реалістичний ігровий простір зі структурою, що відповідає заданим параметрам. Також автоматизувати процес створення рівнів, зменшуючи залежність від ручної роботи та забезпечити високу варіативність, необхідну для підтримки інтересу гравців.

Ця модель визначає процедуру генерації ігрової карти за допомогою клітинного представлення, де кожна клітинка має свій стан: вода (SEA), земля (LAND) чи пісок (SAND). Опис складається з наступних кроків:

1. Генерація стартової карти: На першому етапі створюється матриця M розміром $m \times n$, де кожен елемент M_{mn} відповідає певному стану. Ініціалізація матриці здійснюється випадковим способом або заздалегідь заданими параметрами. Множина станів клітинок: {SEA,LAND}; Умова: $M_{mn} = SEA$, якщо випадкове значення відповідає певному порогу; Інакше $M_{mn} = LAND$.

$$M_{mn} = \begin{cases} SEA, \text{ якщо } rand(1,3) = 1, \\ LAND, \text{ інакше.} \end{cases} \quad (3.1)$$

Цей початковий розподіл формує базову карту для подальшої обробки.

2. Функція перевірки сусідів: Модель використовує концепцію локальних взаємодій між клітинками. Для кожної клітинки визначаються сусіди (x', y'), що знаходяться у радіусі 1:

Функція перевірки сусідів: Нехай *S(x,y), L(x,y)* — лічильники для морських *(SEA)* та земельних *(LAND)* сусідів для клітинки *(x,y)*. Для перевірки сусідів визначимо:

$$Neighbor(x', y') = \begin{cases} SEA, \text{ якщо } M(x', y') = SEA, \\ LAND, \text{якщо } M(x', y') = LAND, \\ \emptyset, \text{якщо } (x', y') \text{ поза межами матриці} \end{cases} \quad (3.2)$$

Алгоритм підрахунку сусідів:

Для кожної клітинки (x,y) обчислюється кількість сусідів певного типу:

$S(x, y)$ кількість сусідів із станом SEA; $L(x, y)$ кількість сусідів із станом LAND.

$$S(x, y) = \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x + \Delta x, y + \Delta y) = SEA) \quad (3.3)$$

$$L(x, y) = \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x + \Delta x, y + \Delta y) = LAND) \quad (3.4)$$

де $\delta(condition) = 1$, якщо $condition$ істинне, і 0 — в іншому випадку.

3. Логіка створення островів: Для кожної клітинки визначається новий стан $M'(x,y)$ алежно від поточного стану: $M(x,y)$ і кількості сусудів:

$$\text{Якщо } M(x,y) = LAND: M'(x, y) = \begin{cases} SEA, \text{ якщо } S(x, y) \in \{3,6,7,8\}, \\ LAND, \text{ інакше.} \end{cases} \quad (3.5)$$

$$\text{Якщо } M(x,y) = SEA: M'(x, y) = \begin{cases} LAND, \text{ якщо } L(x, y) \in \{3,6,7,8\}, \\ SEA, \text{ інакше.} \end{cases} \quad (3.6)$$

Ця логіка забезпечує формування природних контурів островів у карті.

4. Логіка створення піску: Якщо клітинка LAND має сусідство з водою SEA, вона перетворюється на SAND. Це реалізується умовою:

$$L(x,y) \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x + \Delta x, y + \Delta y) = SAND) \quad (3.7)$$

Переваги моделі цієї моделі у простоті реалізації: Використання клітинного підходу з чіткими правилами. Також у гнучкості, адже підхід має легке налаштування параметрів для зміни характеристик світу. Найголовніше, це – природність, застосування локальних правил дозволяє отримати реалістичні контури ландшафтів.

Алгоритм фактично використовує базові ідеї клітинних автоматів, де кожна клітинка змінює свій стан на основі "оточення". це базовий, але ефективний інструмент для генерації ігрового світу. Її цінність полягає у гнучкості та можливості адаптації під різні завдання в процедурній генерації. Вона демонструє, як можна моделювати складні природні процеси за допомогою локальних правил і

взаємодій між клітинками. Таким чином, ця модель є "відправною точкою", яку можна поступово ускладнювати, зберігаючи простоту реалізації.

На цьому принципі можна будувати й більш складні алгоритми. Наприклад:Розширення моделі для генерації підземних печер або тунелів та додавання нових типів клітинок, таких як ліси, гори, будівлі.

Розроблена модель вирізняється серед інших рішень процедурної генерації завдяки балансуванню між простотою і можливістю отримати складний результат. В іграх, таких як Terraria чи Minecraft, схожі принципи застосовуються для генерації ландшафтів і структур. Однак розроблений алгоритм дозволяє задати більш чіткі правила взаємодії між елементами карти (наприклад, зв'язок між сушею та водою). Також дозволяє швидко налаштовувати параметри для досягнення бажаного результату та Легше інтегрувати нові елементи у вже наявну систему.

Ця математична модель є основою для процедурної генерації ігрових світів, створюючи деталізовані карти з реалістичною топографією. Вона об'єднує простоту клітинних автоматів із потужністю налаштовуваних правил, що робить її універсальним інструментом у розробці ігор. Це не просто "готовий алгоритм", а певна платформа, яку можна адаптувати для різних ігор і застосунків. Її основна перевага — модульність і простота налаштування. На її основі можна створювати різноманітні світи, які будуть як функціональними, так і візуально привабливими. Тому вона є базовим, але дуже перспективним рішенням для процедурної генерації контенту в ігровій індустрії.

3.3 Загальна структура розробленої системи

Система, яка розроблена в межах цього проєкту, являє собою програмне рішення для генерації біомів на 2D- та 3D-картах. Головна мета системи — створення процедурно згенерованих середовищ з різними типами біомів, такими як земля, море, пляж, скелі, ліс, білі землі тощо, для інтерактивної візуалізації та

використання в ігрових проєктах. Основний функціонал реалізований з використанням мови програмування C# та рушія Unity.

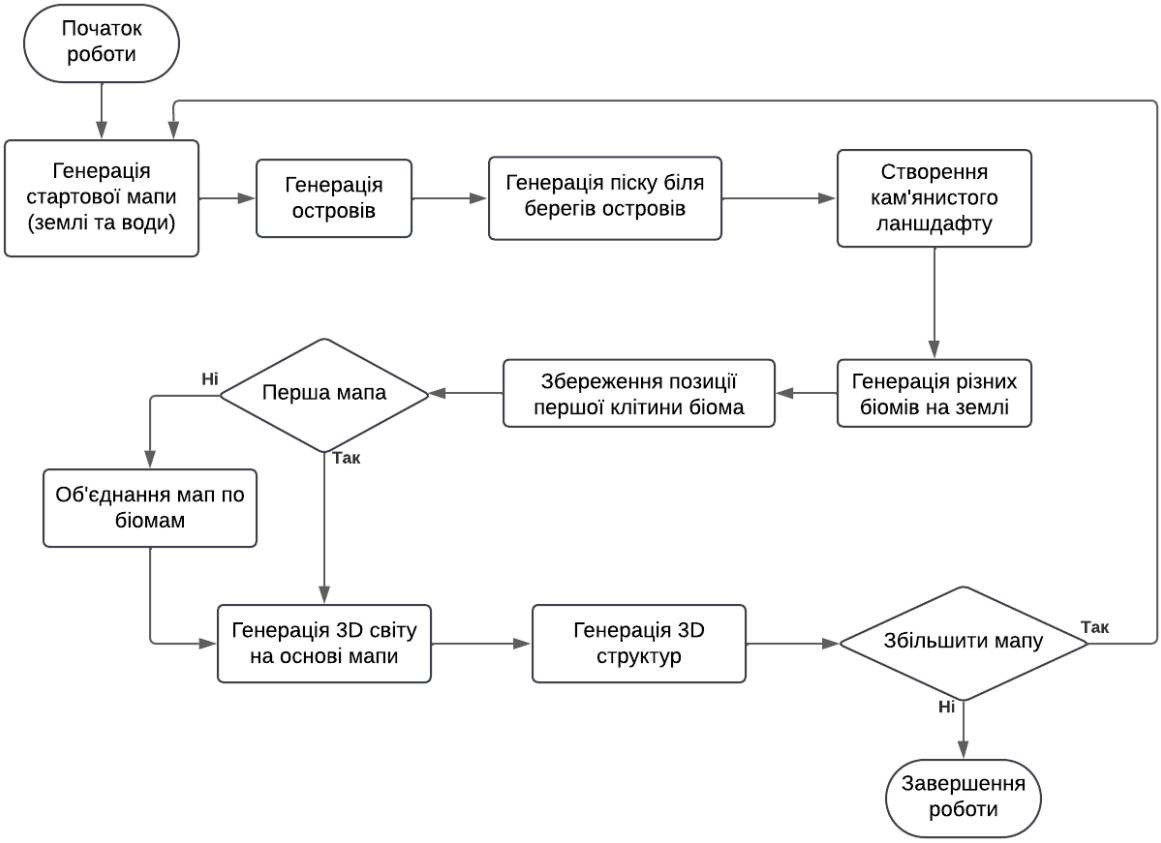


Рис 3.3 Блок-схема роботи алгоритму процедурної генерації.

Система складається з кількох ключових компонентів, які взаємодіють для забезпечення повного циклу генерації та візуалізації карт:

1. Налаштування генерації: зберігає загальні параметри (розміри карти, кольори біомів тощо).
2. Класи для управління біомами: відповідальні за логіку створення різних типів біомів.
3. Генератор 2D та 3D карт: забезпечує створення та відображення карти у двох і трьох вимірах.
4. Інтерфейс користувача: забезпечує інтерактивне перемикання між 2D та 3D режимами, а також активацію різних типів генерації.

Основні параметри генерації визначаються у класі GenerationSettings. У ньому задаються розміри карти, базові координати для обчислення положення кожного елемента, а також кольори, які відповідають різним типам біомів:

1. Розміри карти: 150×150 клітинок.
2. Кольори біомів: земля (зелений), море (синій), пісок (жовтий), скелі (сірий), ліс (темно-зелений), білі землі (білий).

У системи є декілька компонентів, такі як : Клас «BiomesUI», «MapGenerationController», «StartMapCreator», «Creators», «BiomesGenerator», «WorldGenerationController», «BuildingController» тощо. Розберемось з кожним з них по черзі.

1. Клас «BiomesUI» - відповідає за управління інтерфейсом користувача. Його основні функції це керування всім процесом процедурної генерації, завдяки вводу користувачем даних. Окрім даних користувач може або створити повність випадкову мапу, або трохи керувати цим процесом, завдяки вводу клавіш. Ввід клавіш впливає тільки на час, який буде генеруватись мапа.

2. Клас «MapGenerationController», є найголовнішим класом, який відповідає за увесь процес генерації. Він дає команди іншим класам, коли генерувати ту чи іншу структуру. Цей клас має послідовність виклику усіх «Creators» та передає їм основні дані, такі як розмірність мапи, метод фарбування клітин тощо. Також він є провідником між класами «BiomesGenerator» та «WorldGenerationController», оскільки передає останньому список класів «Biome». Це необхідно для того, щоб завдяки цим класам потім згенерувати на їх місці будівлі, по типу замку або міста. Така структура необхідно для того, щоб відділити 2D та 3D генерації один від одного, і для цього було використано «MapGenerationController» я провідника.

3. Клас «StartMapCreator» відповідає за випадкову генерацію двох типів клітин, а саме Land та Sand. Це необхідно для того, щоб зробити генерацію світу кожен раз випадковою. Основна логіка випадковості – 1 до 3, де 3 – це острів, 1 – це вода. Проте значення можливо легко змінити у налаштуваннях.

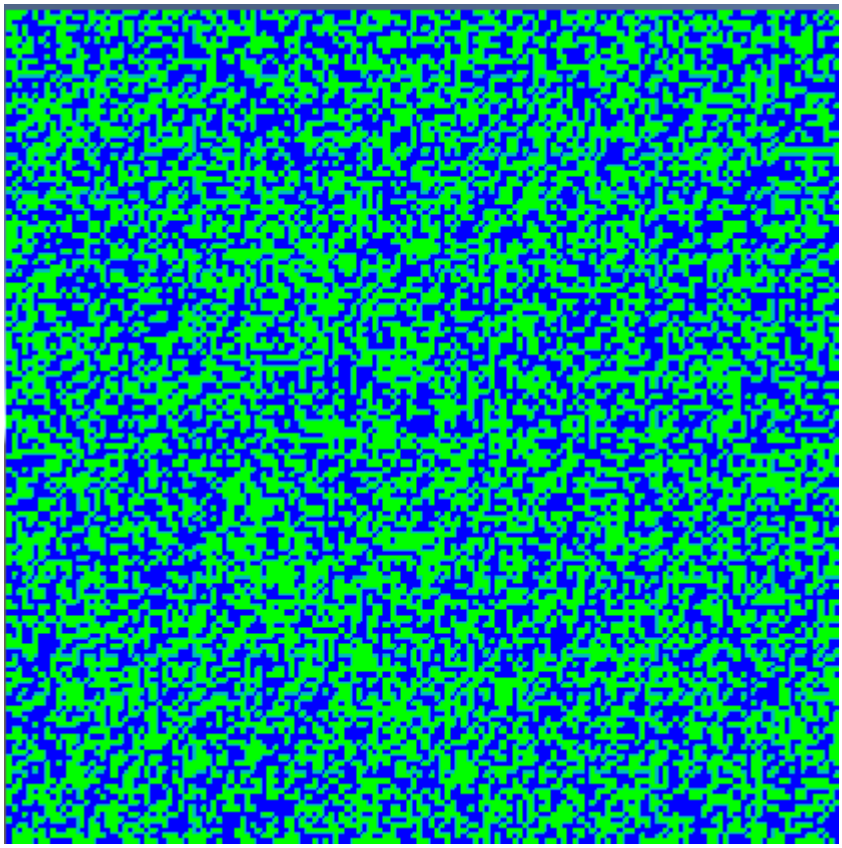


Рис 3.4 Стартова генерація мапи

4. Класи «Creators» необхідні для генерації різних природних елементів на мапі. Оскільки всі вони мають свою логіку появи, вони були розбити на декілька підкласів, які можливо об’єднати в одну категорію «Creators». Всі вони використовують метод клітинного автомата для того, щоб не порушувати логіку метода, який передає «MapGenerationController», а саме методу відповідального за фарбування клітин та передачі їм типу у двовимірному масиві. Всі значення, які були використані для налаштування методу клітинного автомата є підібрані методом пробу, адже алгоритм природної генерації занадто складний для такого ігрового двигуна як «Unity». Тому однієї з цілей було не тільки зробити його природним, а ще й оптимізувати, саме через це програма використовує метод об’єднання декількох згенерованих мап в одну.

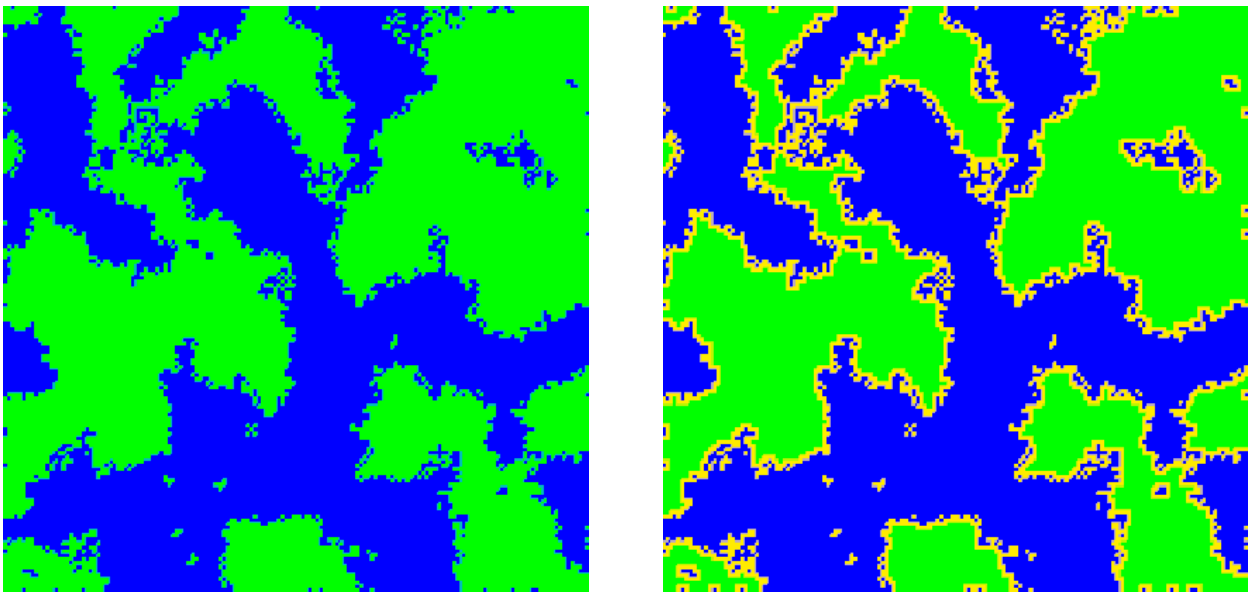


Рис 3.5 Генерація островів та піску на 2D мапі.

5. Клас «BiomesGenerator» - Центральний компонент для генерації біомів. Основні його завдання - Використання різних генераторів (WHITE_LAND, DARK_FORES тощо) для створення специфічних біомів та управління кольорами клітинок. Їх оновлення відбувається на основі змін у матриці біомів. Цей клас використовує такі алгоритми як: Початкова генерація карти (Випадкове розташування клітинок типів "земля" і "море") та Розширення біомів (Пошук сусідніх клітинок для зміни типу (наприклад, створення пляжу навколо моря).

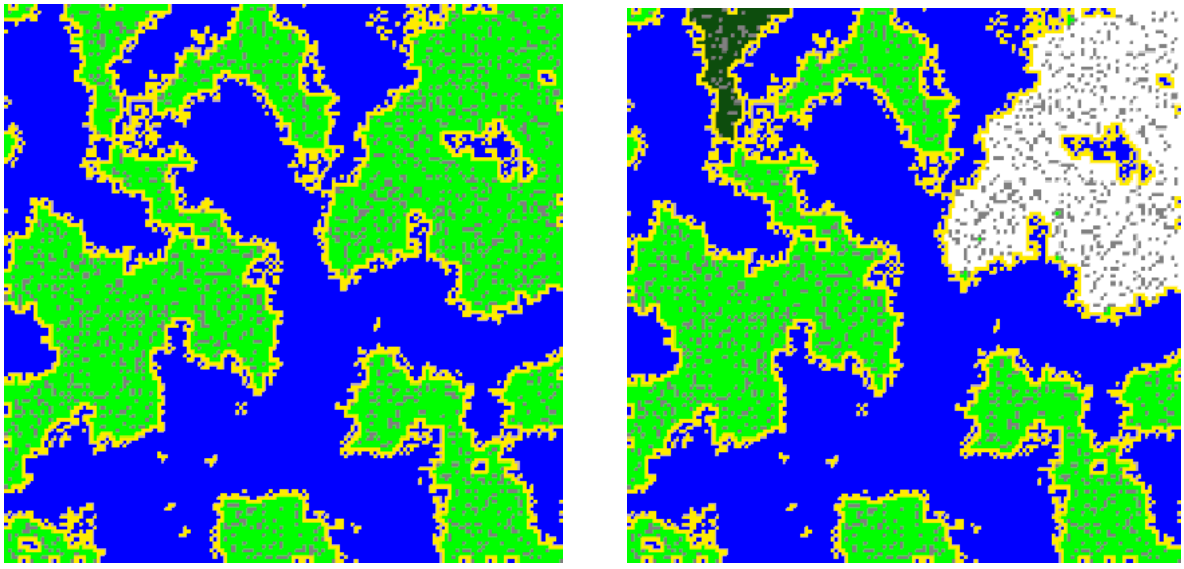


Рис. 3.6 Генерація каміння та біомів.

6. Клас «WorldGenerationController» - відповідальний за 3D генерацію світу та ігрових структур, таких як замки, міста, дерева тощо. Спочатку він дає команду класу «WorldCreator», щоб той згенерував 3D світ, на основі матриці із MapGenerationController. Після генерації світу, «WorldCreator» дає команду «WorldGenerationController», що він закінчив свою роботу, а той своєю чергою дає команду класу «BuildingsCreator», щоб той почав створювати різні структури. Вони створюються на основі стартової позиції початку генерації кожного біому.

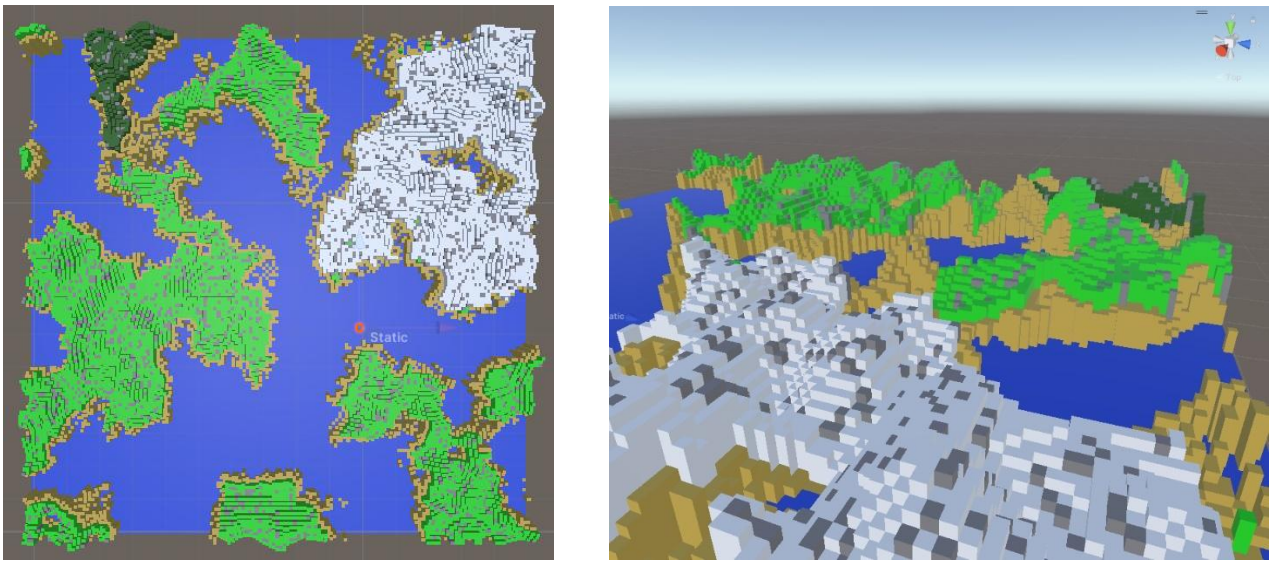


Рис 3.7 Створений 3D світ з видом зверху та в близі.

7. Клас «BuildingController» - відповідальний за генерацію різних будівель, дерево тощо. Він будує на основі позицій біомів різні унікальні структури. Це можуть бути як різні дерева, так й великі селища. За основу береться центрально точка, звідки і починається генерація структур. Для прикладу було зроблено різних видів дерева, щоб показати, як могла б відбуватись генерація селища, або лісу.

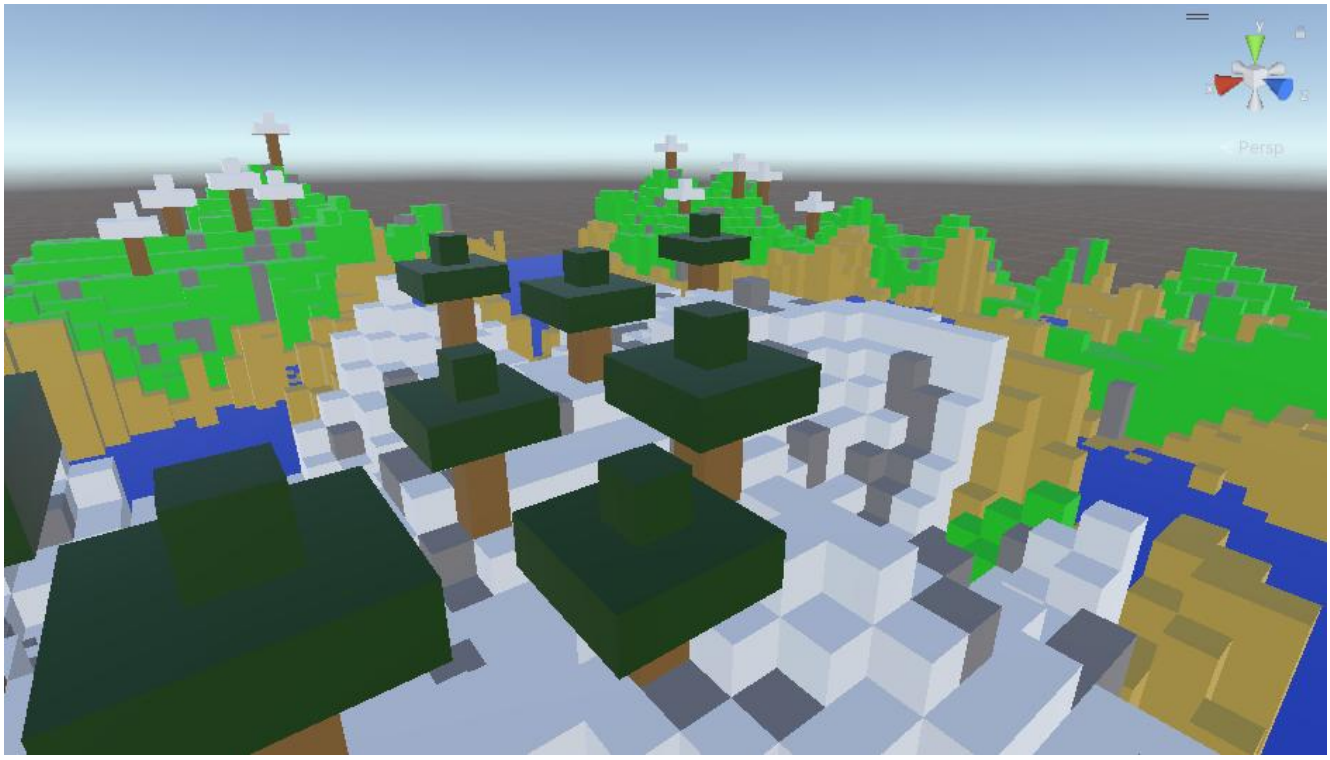


Рис 3.8 Закінчена генерація.

Отже, система забезпечує повний набір функцій для процедурної генерації карт з різними типами біомів, інтерактивного управління і переходу між 2D і 3D режимами. Архітектура є гнучкою для майбутнього розширення, наприклад, додавання нових типів біомів чи інтеграції зі сторонніми системами. Висока точність алгоритму гарантує природність і різноманітність створених ландшафтів [19]. Крім того, система підтримує оптимізацію ресурсів, що дозволяє її використання на середньопродуктивних пристроях.

Розроблений підхід також відзначається зменшеним часом генерації, що робить його ефективнішим у порівнянні з наявними аналогами. Підтримка налаштування параметрів генерації дозволяє адаптувати карти під специфічні вимоги проєкту. Система забезпечує стабільну роботу навіть при створенні великих карт, зберігаючи високу деталізацію. Усі ці переваги роблять її конкурентоспроможною для використання в різноманітних ігрових жанрах

3.4 Опис логіки роботи розроблених класів

У цьому розділі розглядаються основні класи, що були реалізовані для забезпечення роботи алгоритмів генерації ігрового світу. Кожен клас відповідає за конкретний аспект процесу [20], який починається зі створення 2D карти та завершується побудовою 3D середовища.

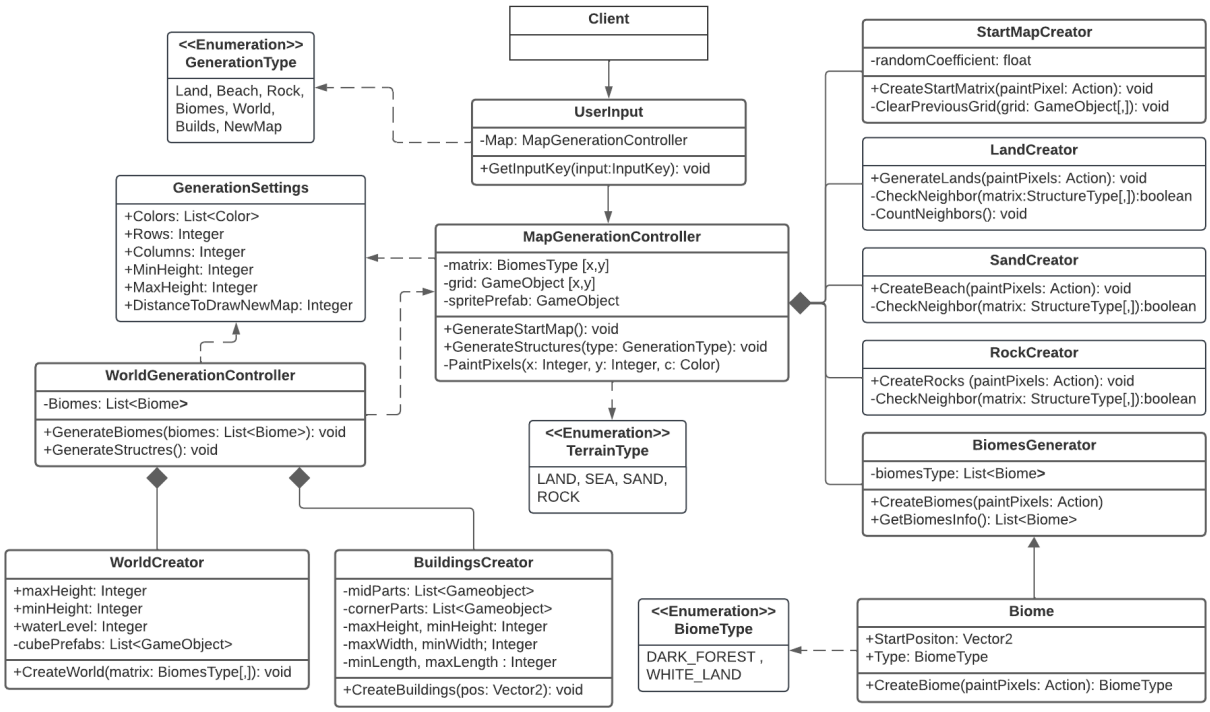


Рис. 3.9 Діаграми класів процедурної генерації ігрового світу.

Клас «StartMapCreator» відповідає за створення стартової мапи, його роботу викликає клас «MapGenerationController», при натисканні кнопки користувачем, або при автоматизованому створенню світу, в залежності від того, який режим гри обрав гравець для генерації.

```
Скрипт Unity | Ссылок: 2
public class StartMapCreator : MonoBehaviour
{
    ссылка: 1
    private void ClearPreviousGrid(GameObject[,] grid)
    {
        if (grid == null) return;

        for (int i = 0; i < grid.GetLength(0); i++)
        {
            for (int j = 0; j < grid.GetLength(1); j++)
            {
                if (grid[i, j] != null)
                {
                    Destroy(grid[i, j]);
                }
            }
        }

        // Очищаємо сітку
        grid = new GameObject[GenerationSettings.Rows, GenerationSettings.Columns];
    }

    ссылка: 1
    public void CreateStartMatrix(ref BiomesType[,] matrix, ref GameObject[,] grid,
        GameObject parent, GameObject spritePrefab, Action<GameObject, BiomesType> paintPixel)
    {
        ClearPreviousGrid(grid);
        int rows = GenerationSettings.Rows;
        int cols = GenerationSettings.Columns;
        matrix = new BiomesType[rows, cols];
        var random = new System.Random();

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i, j] = random.Next(1, 3) == 1 ? BiomesType.SEA : BiomesType.LAND;

                // Створення квадрата
                Vector3 position = new Vector3(i * GenerationSettings.BasicX, j * GenerationSettings.BasicY, 0);
                var sprite = UnityEngine.Object.Instantiate(spritePrefab, position, Quaternion.identity, parent.transform);
                grid[i, j] = sprite;

                // Початковий колір
                paintPixel(sprite, matrix[i, j]);
            }
        }
    }
}
```

Рис 3.10 Генерація стартової карти з двох типів клітин(Land, Sea).

Логіка генерації островів, як було згадано раніше, працює за логікою методу клітинних автоматів. Підрахунок сусід відбувається за наступною логікою:

```
7       Ссылок: 8
        private void CheckNeighbor(BiomesType[,] matrix, int x, int y, ref int counterSea, ref int counterLand)
8       {
9           if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
10          {
11              if (matrix[x, y] == BiomesType.SEA)
12              {
13                  counterSea++;
14              }
15              else if (matrix[x, y] == BiomesType.LAND)
16              {
17                  counterLand++;
18              }
19          }
20      }
```

Рис 3.11 Підрахунок сусід для клітини.

У більшості випадків всі перетворення у клітинних автоматів виглядають схожим образом:

```
if (matrix[x, y] == BiomesType.LAND)
{
    if (counterSea == 3 || counterSea == 6 || counterSea == 7 || counterSea == 8)
    {
        newMatrix[x, y] = BiomesType.SEA;
    }
}
else if (matrix[x, y] == BiomesType.SEA)
{
    if (counterLand == 3 || counterLand == 6 || counterLand == 7 || counterLand == 8)
    {
        newMatrix[x, y] = BiomesType.LAND;
    }
}
```

Рис 3.12 Перетворення клітин в залежності від сусідів.

Проте, іноді бувають вийнятки, наприклад, у генерації піску:

```
private bool CheckNeighborForSand(BiomesType[,] matrix, int x, int y)
{
    if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
    {
        return matrix[x, y] == BiomesType.SEA;
    }
    return false;
}
```

Риск 3.13 Перевірка сусідів піску.

Адже якщо у клітини є сусід море, отже вона має бути піском.

Генерація 3D світу, відбувається за трохи складнішим алгоритмом, при цьому використовуючи шум Перліна для своєї роботи. Рівень завжди має нульове значення по у, проте, це можливо дуже легко змінити. На даному етапі воно має таке значення, адже у води немає фізики, тому, летючі куби будуть виглядати некоректно. Генерація структур генерує заздалегідь створений префаб будівлі, який також має вкладену логіку для того, щоб будівлі виглядали по різному. Код генерації світу та структур:

```
public void Generate3DMap(BiomesType[,] matrix, GameObject parent)
{
    int rows = matrix.GetLength(0);
    int cols = matrix.GetLength(1);

    _parent = parent.transform;
    for (int x = 0; x < rows; x++)
    {
        for (int y = 0; y < cols; y++)
        {
            float perlinValue = Mathf.PerlinNoise(x * noiseScale, y * noiseScale);
            int height = Mathf.RoundToInt(Mathf.Lerp(minHeight, maxHeight, perlinValue));

            if (matrix[x, y] == BiomesType.SEA)
            {
                height = waterLevel;
            }

            for (int h = 0; h < height; h++)
            {
                GameObject cube;
                Vector3 position = new Vector3(x, h, y);
                switch (matrix[x, y])
                {
                    case BiomesType.LAND:
                        cube = Instantiate(_landCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.SAND:
                        SanCorrect(out cube, x, h, y);
                        break;
                    case BiomesType.WHITE_LAND:
                        cube = Instantiate(_whiteCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.ROCK:
                        cube = Instantiate(_rockCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.WOODS:
                        cube = Instantiate(_woodCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.SEA:
                        cube = Instantiate(_seaCube, position, Quaternion.identity, _parent);
                        break;
                    default:
                        break;
                }
            }
        }
    }
```

Рис 3.14 Генерація 3D мапи світу.

```
public void GenerateBiomes(List<Biome> biomes)
{
    foreach (Biome biome in biomes)
    {
        GameObject prefabToInstantiate = null;

        switch (biome.Type)
        {
            case BiomeType.DARK_FOREST:
                prefabToInstantiate = darkForestPrefab;
                break;
            case BiomeType.WHITE_LAND:
                prefabToInstantiate = whiteLandPrefab;
                break;
        }
        if (prefabToInstantiate != null)
        {
            Instantiate(prefabToInstantiate, new Vector3(biome.Position.x, 0, biome.Position.y), Quaternion.identity);
        }
    }
}
```

Рис 3.15 Генерація 3D структур та будівель.

У роботі реалізовано модульну систему процедурної генерації ігрового світу, яка ефективно поєднує алгоритми клітинних автоматів, шум Перліна та принципи об'єктноорієнтованого програмування. Вона забезпечує створення як 2D-карти, так і її трансформацію у 3D-середовище з різноманітними біомами, структурами та ландшафтами. Завдяки чіткій структурі класів [21] і використанню готових префабів, система легко розширюється та налаштовується під різні потреби ігрового процесу. Таким чином, вона забезпечує баланс між функціональністю, гнучкістю та продуктивністю, що робить її універсальним рішенням для розробки ігрових світів.

3.5 Результати розробленого алгоритму

Розроблений метод процедурної генерації показує себе як потужне й перспективне рішення для створення ігрових світів. Його головними перевагами є висока гнучкість налаштувань, що дозволяє адаптувати карту під різні жанри ігор та специфічні потреби проєктів. Здатність створювати великі світи з реалістичними ландшафтами ставить цей метод на рівень із найкращими сучасними алгоритмами, а у деяких аспектах навіть перевершує їх.

Особливо виділяється швидкість роботи алгоритму: час генерації та завантаження значно знижений, що робить його зручним як для гравців, так і для розробників. Завдяки середнім вимогам до ресурсів, алгоритм здатний працювати на більшості сучасних пристроїв без зниження продуктивності. Ця оптимізація може суттєво розширити його застосування, включаючи мобільні платформи та інді-проєкти.

Єдиним аспектом, що може потребувати вдосконалення, є спрощення інтеграції зі сторонніми системами та автоматизація підбору параметрів для новачків. Незважаючи на це, розроблений метод вже є потужним інструментом для створення процедурно-генерованих світів, демонструючи баланс між продуктивністю, якістю та універсальністю.

Сильні сторони:

1.Розроблений алгоритм демонструє високу гнучкість у налаштуванні параметрів. Це дає змогу створювати різноманітні світи залежно від потреб користувача або типу гри. У порівнянні з алгоритмами Minecraft, Terraria та Starbound, які мають низьку гнучкість, ваш метод є більш адаптивним.

2. Розроблений метод здатний генерувати карти розміром до 12000x12000 блоків, що є одним із найбільших показників серед порівнюваних систем. Це перевищує можливості Terraria (8400x2400) та Starbound (12000x3600).

3. Алгоритм забезпечує високу реалістичність середовища, яка перевершує Minecraft і Terraria, наближаючись до Starbound. Це досягається завдяки використанню спеціалізованих біомів та адаптивної логіки генерації.

4. Розроблений алгоритм має значно швидший час генерації світу — в середньому 1.5 хвилини. Це вдвічі менше, ніж у Minecraft (4 хвилини), і значно краще, ніж у Starbound (3 хвилини).

5. Завантаження карт триває лише 3-5 секунд, що значно швидше у порівнянні з Minecraft (10-15 секунд) та Starbound (7-9 секунд).

Проте, на жаль, не обійшлось без слабких сторін:

1. Алгоритм має середні вимоги до апаратного забезпечення, які дещо перевищують Terraria, проте вони залишаються нижчими, ніж у Minecraft. Це може обмежувати його використання на слабких пристроях.

2. Хоча розмір світу є значною перевагою, генерація настільки великих карт може спричиняти проблеми з оптимізацією у реальному часі на деяких системах. У цьому аспекті Terraria виграє завдяки більш компактним світам.

3. Алгоритм не може змагатись з такою великою генерацією, як «Minecraft», алгоритм підходить для більш малих, та можливо, сюжетно-рольових ігор.

Таблиця 3.1

Порівняльна характеристика алгоритмів генерації ігрових мап

Критерій	**Minecraft**	**Terraria**	**Starbound**	**Розроблений алгоритм**
Можливість налаштування	Низька	Низька	Низька	Висока
Розмір світу, блоків	До 60 млн об'єктів	8400x2400	12000x3600	12000x12000
Природність ландшафту	Середня	Середня	Висока	Висока
Час генерації (середній), хв	4	2	3	1.5
Час завантаження карт, сек	10-15	5-7	7-9	3-5
Вимоги до ресурсів	Високі	Низькі	Середні	Середні

Розроблений алгоритм має низку унікальних переваг, які роблять його потужним інструментом для створення великих, адаптивних та реалістичних світів. Він перевершує конкурентів у налаштуванні параметрів, швидкості роботи та підтримці великих розмірів карти. Однак, слід звернути увагу на оптимізацію великих світів та вимоги до ресурсів, щоб зробити алгоритм ще більш універсальним.

Цей метод використовує унікальний підхід до генерації мапи, генеруючи окремі сегменти (карти) незалежно одна від одної, а потім об’єднуючи їх в єдину цілісну структуру. Цей підхід дав значний приріст оптимізації, оскільки дозволяє розподілити навантаження на обчислювальні ресурси більш рівномірно.

Генерація сегментів окремо забезпечує можливість паралельної обробки, що значно знижує час генерації навіть для дуже великих карт. Об'єднання сегментів у кінцевий світ реалізується через спеціальні алгоритми зшивання, які зберігають

природність ландшафту та забезпечують плавні переходи між зонами. Цей метод дозволив скоротити час обробки майже вдвічі, порівняно з традиційними підходами, які генерують мапу як єдине ціле.

Крім того, така модульність генерації відкриває можливість для динамічного оновлення або розширення карти під час гри без помітних затримок, що робить цей алгоритм універсальним і придатним для багатьох сценаріїв використання, включаючи ігри з відкритим світом і симулятори.

Розроблений метод також реалізує механізм динамічного відображення 3D об'єктиву, який генерується в режимі реального часу навколо позиції персонажа на 2D-мапі. Якщо персонаж знаходиться на позиції, наприклад, 2x3, то 3D-мапа малюється в радіусі 1 від цієї точки. Це означає, що відображаються всі відповідні 3D-сегменти: 1x2, 1x3, 1x4, 2x2, 2x3, 2x4, 3x2, 3x3 і 3x4.

Такий підхід дозволяє суттєво оптимізувати обчислювальні ресурси, адже рендеринг 3D-ландшафту обмежується лише необхідною для взаємодії з персонажем областю. Решта світу залишається у 2D-форматі, що знижує навантаження на систему. Це рішення забезпечує плавність переходів між 2D- і 3D-режимами, а також підтримує високу продуктивність навіть у великих світах із деталізованою графікою.

Також було використано переваги матеріалів і рендер методів Unity, що дозволяють малювати всі необхідні сегменти 3D-мапи за один раз. Завдяки таким інструментам Unity, як Instancing (економічне повторне використання однакових об'єктів) та Batching (групування об'єктів для одноразового рендерингу), система мінімізує кількість викликів до графічного процесора (GPU).

Цей підхід ще більше підвищує продуктивність, дозволяючи відображати великий обсяг деталей у реальному часі, без значного навантаження на ресурси. Наприклад, використання матеріалів із підтримкою Shader Graph забезпечує оптимальну взаємодію між освітленням, текстурами та тінями, що створює реалістичний вигляд світу з мінімальними витратами на обчислення. Завдяки цим

можливостям Unity, метод досягає високої ефективності у відображенні як великих, так і локальних зон карти.

Окрім оптимізацій, пов’язаних із матеріалами та рендерингом, розроблений метод ефективно використовує Occlusion Culling, що дозволяє значно зменшити навантаження на графічний процесор. Ця технологія забезпечує малювання тільки тих об'єктів, які знаходяться в межах видимості камери, тоді як всі інші сегменти світу, що перекриваються або знаходяться поза межами огляду, не обробляються.

Застосування Occlusion Culling у поєднанні з динамічним рендерингом 3D-мапи в радіусі відносно позиції персонажа забезпечує надзвичайно високу продуктивність навіть для карт із великою кількістю деталей. Це дозволяє зосередити обчислювальні ресурси на відображенні лише важливих для гравця елементів, що робить метод не тільки швидким, але й масштабованим для великих світів.

Додатково, завдяки інтеграції Occlusion Culling з процедурним генеруванням, система демонструє чудову адаптивність до різних конфігурацій обладнання. Це означає, що навіть на менш потужних пристроях метод забезпечує плавний геймплей без помітного зниження якості. У поєднанні з можливістю відмальовувати матеріали за допомогою ефективних методів Unity, таких як Batching та LOD (Level of Detail), розроблений підхід стає універсальним і ідеально підходить як для малих, так і для великих відкритих світів.

Розроблений метод демонструє значний потенціал для застосування у відкритих світах, які вимагають деталізації та динамічності. Його оптимізація дозволяє інтегрувати складні ландшафти без суттєвого навантаження на ресурси системи, забезпечуючи плавність роботи навіть у масштабних проєктах.

ВИСНОВКИ

У процесі роботи було підібрано та опрацьовано відповідну науково-технічну літературу, яка забезпечила фундамент для подальших досліджень у сфері процедурної генерації контенту. Проведено глибокий аналіз наявних алгоритмів, таких як Binary Space Partitioning, Cellular Automata, Graph-based algorithms та Voronoi Diagrams. Цей аналіз дозволив оцінити їх сильні сторони та недоліки, що стало основою для вибору найбільш доцільного підходу до генерації ігрових структур.

На основі клітинних автоматів було створено алгоритми для процедурної генерації, що забезпечило підвищення якості створюваного контенту та ефективності процесу генерації. Розроблена математична модель дозволила моделювати 2D мапи з урахуванням природних зон (земля, вода, пісок), що забезпечило більш реалістичний вигляд ігрових просторів. Реалізація функції перетворення 2D мапи у 3D середовище надала змогу візуалізувати згенеровані структури у вигляді об’ємного ландшафту, створюючи нові можливості для інтерактивності та занурення гравця у віртуальний світ.

У середовищі Unity була розроблена архітектура програмного забезпечення для генерації ігрових структур, яка включає модулі для створення стартових карт, біомів, пляжів, скель, будівель та інших об’єктів. Завдяки цьому вдалося інтегрувати різноманітні елементи генерації в єдину систему, що дозволяє розробникам ігор легко налаштовувати параметри та створювати унікальні ігрові світи.

Проведено тестування розроблених алгоритмів, яке підтвердило їх ефективність. Результати засвідчили можливість створення реалістичних ігрових просторів із мінімальними витратами ресурсів, що відкриває перспективи для застосування цих алгоритмів у сучасних ігрових проєктах.

ПЕРЕЛІК ПОСИЛАНЬ

1. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games, Fourth Edition / Tracy Fullerton., 2019. – 522 с. – (A K Peters/CRC Press). – (4).
2. Shekr N. Procedural Content Generation in Games / N. Shekr, J. Togelius, M. J. Nelson., 2018. – 237 с.
3. Shaker N. Procedural Content Generation in Games [Електронний ресурс] / N. Shaker, J. Togelius, M. Nelson. – 2016. – Режим доступу до ресурсу: https://www.pcgbook.com/.
4. Wu Z. Perlin noise and its improvements: A literature review [Електронний ресурс] / Z. Wu, J. Liu // Stavros Shiaeles. – 2024. – Режим доступу до ресурсу: https://www.ewadirect.com/proceedings/ace/article/view/14225.
5. Random [Електронний ресурс] // .NET Platform.. – 2023. – Режим доступу до ресурсу: https://github.com/dotnet/coreclr/blob/ebda0e66df0dd19688cf8a88375bb31c184f5037/src/mscorlib/shared/System/Random.cs.
6. Procedural Content Generation for Games: A Survey [Електронний ресурс] / M.Hendrikx, S. Meijer, J. Van Der Velden, A. Iosup. – 2013. – Режим доступу до ресурсу:https://www.researchgate.net/publication/262327212_Procedural_Content_Generation_for_Games_A_Survey
7. Schell J. The Art of Game Design / Jesse Schell., 2019. – 610 с. – (A K Peters/CRC Press).
8. Procedural Content Generation for Games: A Survey [Електронний ресурс] / M.Hendrikx, S. Meijer, J. Van Der Velden, A. Iosup. – 2013. – Режим доступу до

pecypcy:https://www.researchgate.net/publication/262327212_Procedural_Content_Generation_for_Games_A_Survey

9. Adams T. Procedural Generation in Game Design / T. Adams, T. Short., 2017. – 336 с.
10. L-system [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/L-system.
11. Hasan Sabit N. Is the Collapse of Wave Function at the Heart of Reality? [Електронний ресурс] / Nahidul Hasan Sabit. – 2024. – Режим доступу до ресурсу: https://medium.com/@sabit.hasan006/is-the-collapse-of-wave-function-at-the-heart-of-reality-15f67a5af2e2
12. Prusinkiewicz P. The Algorithmic Beauty of Plants / P. Prusinkiewicz, A. Lindenmayer., 2012. – 228 с. – (Springer New York, NY).
13. Binary Space Partitioning [Електронний ресурс] // Geeksforgeeks. – 2020. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/binary-space-partitioning/. (date of access: 01.06.2024)
14. The fascinating world of Voronoi diagrams [Електронний ресурс] // Towards Data Science. – 2022. – Режим доступу до ресурсу: https://towardsdatascience.com/the-fascinating-world-of-voronoi-diagrams-da8fc700fa1b. (date of access: 01.06.2024)
15. An Intro to Graphs For Procedural World Generation [Електронний ресурс] // Wintermutedigital. – 2020. – Режим доступу до ресурсу: https://wintermutedigital.com/post/graphsforproceduralworlds/. (date of access: 01.06.2024)
16. Buckley M. Game Design: Level Generation Using Binary Space Partitioning [Електронний ресурс] / Matt Buckley // Medium. – 2018. – Режим доступу до ресурсу: https://medium.com/nice-things-ios-android-development/game-design-level-generation-using-binary-space-partitioning-e229230ab8b5. (date of access: 01.06.2024)

17. Zhang C. Cellular Automata DEVS: A Modeling, Simulation, and Visualization Environment [Електронний ресурс] / C. Zhang, H. Sarjoughian // Digital library. – 2017. – Режим доступу до ресурсу: https://dl.acm.org/doi/10.1145/3173519.3173534.
18. Babiac M. DISCOVERING GRAPH GENERATION ALGORITHMS [Електронний ресурс] / M. Babiac, M. Karolis, R. Wattenhofer // NeSy-GeMs workshop. – 2023. – Режим доступу до ресурсу: https://nesygems.github.io/assets/pdf/papers/discovering.pdf.
19. Кнут Д. Мистецтво програмування / Дональд Ервін Кнут., 2020.
20. Refactoring Guru [Електронний ресурс]. – 2014. – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns.
21. Gang of Four Design Patterns [Електронний ресурс] // scholarhat. – 2024. – Режим доступу до ресурсу: https://www.scholarhat.com/tutorial/designpatterns/gang-of-four-gof-design-patterns

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО - КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерськаробота

«МЕТОД ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ ТА СТРУКТУР ДЛЯ ГРИ В ЖАНРІ RPG»

Виконав: студент групи ПДМ-63 Бойко Микита Сергійович

Керівник: д.т.н., проф., професор кафедри ІПЗАС Бондарчук Андрій Петрович

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення продуктивності та якості процедурного контенту шляхом вдосконалення алгоритмів, заснованих на моделі клітинних автоматів для генерації ігрових структур.

Об'єкт дослідження: процес генерації ігрових структур для створення ігрових світів.

Предмет дослідження: методи, засоби та алгоритми для автоматизованої генерації ігрових світів та структур.

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ

Методи / Критерії	Рекурсивний поділ простору	Діаграми Вороного	Алгоритми на основі графів	Клітинні автомати
Основний принципроботи	Рекурсивне поділення простору на підрегіони	Поділпростору наділянки за найближчоюточкою	Графи як вузли та з'єднання	Клітинна симуляція за правилами сусідства
Складність реалізації	Низька: Простийалгоритм без складних обчислень	Висока: Складна математика для побудови й оптимізаціїкраїв.	Висока: Потрібно будувати графи, налаштовувати пошук шляхів.	Середня: Простий код, але важке налаштування правил.
Гнучкість	Низька: Придатний лише для регулярнихформ	Висока: Придатний для природнихзон	Висока: Контроль структури та деталей	Висока: Легко змінювати правила
Використання в 3D	Підходить для будівель, кімнат	Добре для біомів, водойм	Легке створення складних структур	Добре підходить для печер, рельєфів, ландшафту
Переваги	Легко реалізувати	Генерація реалістичних зон	Придатний для лабіринтів, доріг	Простота, природний вигляд, безліч можливостей налаштування
Недоліки	Обмеженийдля природних середовищ	Складна обробка країв	Складний у налаштуванні	Складний у налаштуванні

3

МАТЕМАТИЧНА МОДЕЛЬ АЛГОРИТМУ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ

1. Генерація стартовоїмапи: Нехай M— матрицястанів розміром $m \times n$, де M_{mn} приймаєзначенняз множин {SEA,LAND}; $M_{mn} = \begin{cases} SEA, \text{якщо } rand(1,3) = 1, \\ LAND, \text{інакше.} \end{cases}$

2. Функція перевірки сусідів: Нехай $S(x,y)$, $L(x,y)$ — лічильники для морських *(SEA)* та земельних *(LAND)* сусідів для клітинки (x,y). Для перевіркисусідів визначимо:

$$Neighbor(x', y') = \begin{cases} SEA, \text{якщо } M(x', y') = SEA, \\ LAND, \text{якщо } M(x', y') = LAND, \\ \emptyset, \text{якщо } (x', y') \text{ поза межами матриці} \end{cases}$$

Алгоритмпідрахункусусідів для клітинки (x,y):

$$S(x,y) = \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x+\Delta x, y+\Delta y) = SEA)$$

$$\mathrm{L}(x,y) = \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x+\Delta x, y+\Delta y) = LAND)$$

де $\delta(condition) = 1$, якщо $condition$ істинне, і 0— в іншому випадку.

3. Логіка створенняострівків: Зміна стану клітинки $M(x,y)$ визначається поточним станом і числом сусідів:

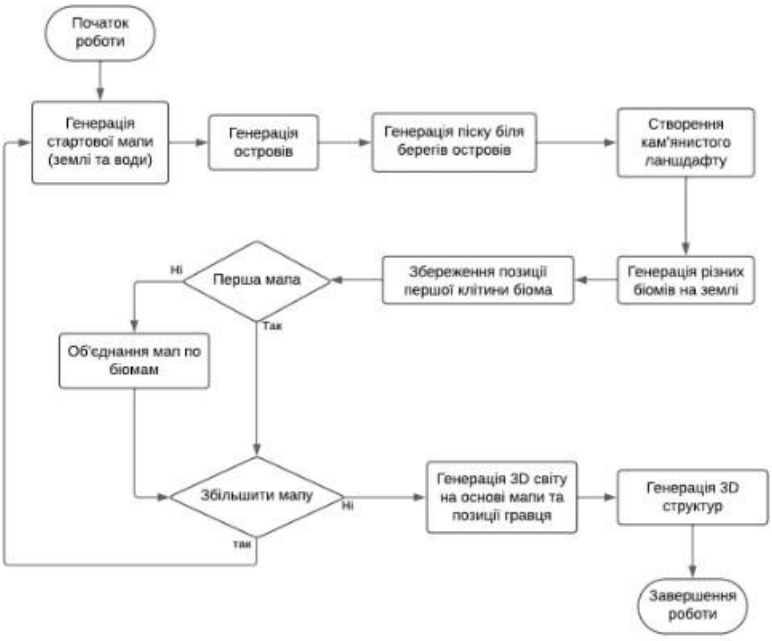
Якщо $M(x,y) = LAND$, то новийстан: $M'(x,y) = \begin{cases} SEA, \text{якщо } S(x,y) \in \{3,6,7,8\}, \\ LAND, \text{інакше.} \end{cases}$

Якщо $M(x,y) = SEA$, то новий стан: $M'(x,y) = \begin{cases} LAND, \text{якщо } L(x,y) \in \{3,6,7,8\}, \\ SEA, \text{інакше.} \end{cases}$

4. Логіка створенняпеску: Якщо LAND має сусідство з водою, він перетворюється на SAND
$\mathrm{L}(x,y) \sum_{\Delta x, \Delta y \in \{-1,0,1\} \setminus \{(0,0)\}} \delta(M(x+\Delta x, y+\Delta y) = SAND)$

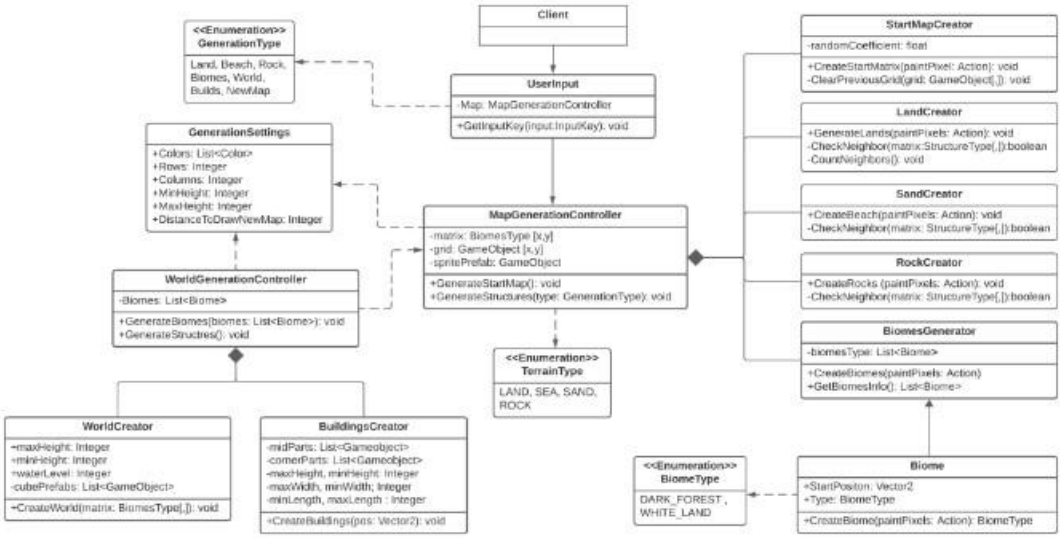
4

БЛОК-СХЕМА РОБОТИ АЛГОРИТМУ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ



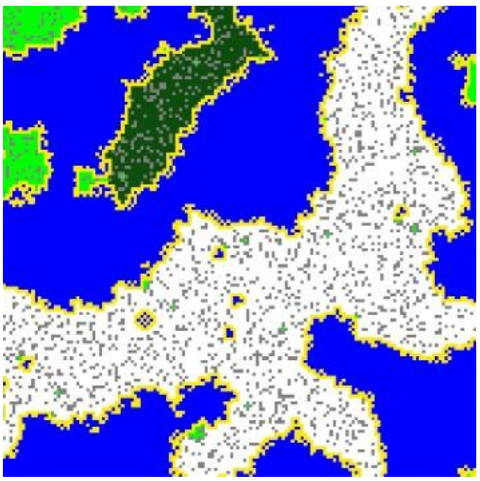
5

ДІАГРАМИ КЛАСІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ



6

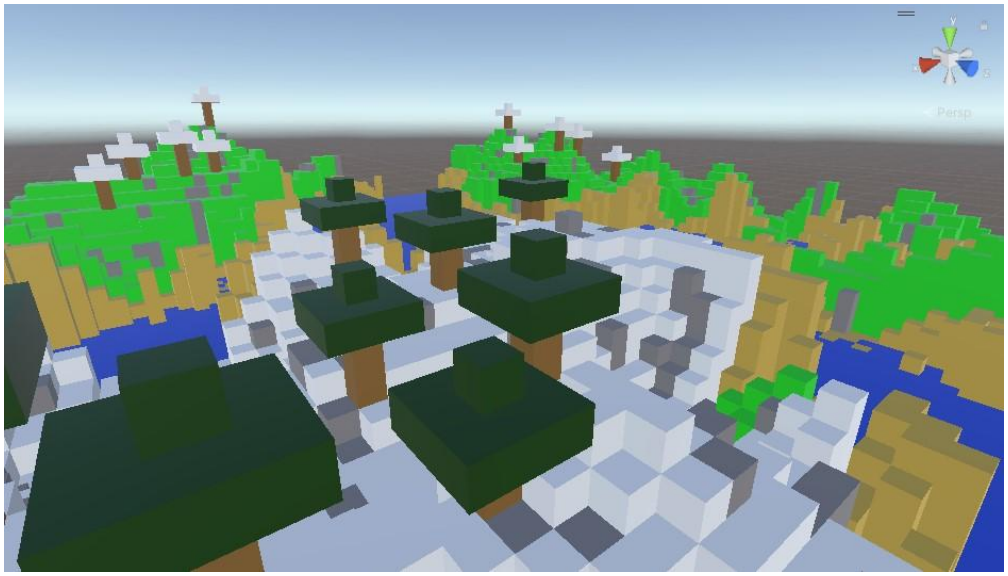
ЕКРАННІ ФОРМИ



Згенерована 2D мапа

Згенерований 3D світ

ЕКРАННІ ФОРМИ



Зовнішний вигляд світу від першого обличчя

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ

Методи / Критерії	Minecraft	Terraria	Starbound	Розроблений алгоритм
Налаштування генерації	-	-	-	+
Придатність до 3D-генерації	+	-	-	+
Висока продуктивність	+	+	+	+
Реалістичність створеного світу	-	-	-	+

ВИСНОВКИ

1. Проведено аналіз існуючих алгоритмів процедурної генерації, таких як «Рекурсивний поділ простору», «Клітинний автомат», «Алгоритми на основі графів», та «Діаграми Вороного».
2. Створено алгоритми на моделі клітинних автоматів для процедурної генерації ігрових структур, що дозволило досягти підвищення якості створюваного контенту та продуктивності генерації.
3. Розроблено математичну модель генерації 2D мапи із врахуванням природних зон (земля, вода, пісок).
4. Реалізовано функцію перетворення 2D мапи у 3D середовище, що забезпечило візуалізацію згенерованих структур у вигляді об'ємного ландшафту.
5. Розроблено архітектуру програмного забезпечення для генерації ігрових структур у середовищі Unity, що включає модулі генерації стартової карти, біомів, пляжів, скель, будівель та інших об'єктів.
6. Здійснено тестування роботи алгоритмів, що підтвердило їх ефективність у створенні реалістичних ігрових світів із мінімальними витратами ресурсів.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Бойко М.С., Бондарчук А. П., Застосування методів генерації ігрових структур для гри з відкритим всесвітом у жанрі RPG // Зв'язок. №6 2024 подана до друку

Тези доповідей:

1. Бойко М.С., Бондарчук А. П., Застосування шаблонів проєктування Factory та Observer на ігровому двигуні Unity для створення інвентарю головного // Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії»
Подано до піблукації. 24.11.2024, ДУІКТ, м. Київ

2. Бойко М.С., Бондарчук А. П., Використання та редагування базового алгоритму псевдовипадкових чисел C# для генерацій подій у грі // Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії».
Подано до піблукації. 24.11.2024, ДУІКТ, м. Київ

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
using BiomeCreator;
using UnityEngine;

public class MapGenerationController : MonoBehaviour
{
    private BiomesType[,] _matrix;
    private GameObject[,] _grid;
    private GameObject _spritePrefab;
    private StartMapCreator _startMap = new StartMapCreator();
    private LandCreator _land = new LandCreator();
    private SandCreator _sand = new SandCreator();
    private RockCreator _rock = new RockCreator();
    private WhiteLandCreator _whiteLand = new WhiteLandCreator();
    private DarkForestCreator _woods = new DarkForestCreator();
    private Map3DGenerator _3DGenerator;

    private void PaintPixelElement(GameObject sprite, BiomesType biome)
    {
        var renderer = sprite.GetComponent<SpriteRenderer>();
        switch (biome)
        {
            case BiomesType.LAND:
                renderer.color = GenerationSettings.COLOR_LAND;
                break;
            case BiomesType.SEA:
                renderer.color = GenerationSettings.COLOR_SEA;
                break;
            case BiomesType.SAND:
                renderer.color = GenerationSettings.COLOR_SAND;
                break;
            case BiomesType.WHITE_LAND:
                renderer.color = GenerationSettings.COLOR_WHITELAND;
                break;
            case BiomesType.WOODS:
                renderer.color = GenerationSettings.COLOR_WOODS;
                break;
            case BiomesType.ROCK:
                renderer.color = GenerationSettings.COLOR_ROCKS;
                break;
            default:
                renderer.color = Color.black; // Для невідомих біомів
                break;
        }
    }

    private void UpdateGridColors()
    {
        for (int x = 0; x < GenerationSettings.Rows; x++)
        {
            for (int y = 0; y < GenerationSettings.Columns; y++)
            {
                PaintPixelElement(_grid[x, y], _matrix[x, y]);
            }
        }
    }

    public MapGenerationController(GameObject spritePrefab, Map3DGenerator mapGen)
    {
```

```
        _spritePrefab = spritePrefab;
        _grid = new GameObject[GenerationSettings.Rows, GenerationSettings.Columns];
        _3DGenerator = mapGen;
    }

    public void GenerateStartMap(GameObject parent)
    {
        _startMap.CreateStartMatrix(ref _matrix, ref _grid, parent, _spritePrefab, PaintPixelElement);
    }

    public void GenerateLands()
    {
        _land.GenerateLands(ref _matrix, UpdateGridColors);
    }

    public void GenerateBeach()
    {
        _sand.CreateBeach(ref _matrix, ref _grid, PaintPixelElement);
    }

    public void GenerateRocks()
    {
        _rock.CreateRocks(ref _matrix, UpdateGridColors);
    }
    public void GenerateWhiteLand()
    {
        _whiteLand.CreateWhiteLand(ref _matrix, UpdateGridColors);
    }

    public void GenerateWoods()
    {
        _woods.CreateWoods(ref _matrix, UpdateGridColors);
    }
    public void Generate3DWorld(GameObject parent)
    {
        _3DGenerator.Generate3DMap(_matrix, parent);
    }
    public void GenerateStructures()
    {

    }
    public void AddNewMap()
    {

    }
}
using UnityEngine;

public class GenerationSettings
{
    public const int Rows = 150;
    public const int Columns = 150;
    public const int BasicX = 1;
    public const int BasicY = 1;
    public const int MaxYLevel = 10;
    public static readonly Color COLOR_LAND = Color.green;
    public static readonly Color COLOR_SEA = Color.blue;
    public static readonly Color COLOR_SAND = Color.yellow;
    public static readonly Color COLOR_SEA_SHORE = Color.cyan;
    public static readonly Color COLOR_ROCKS = Color.grey;
    public static readonly Color COLOR_WOODS = new Color(15 / 255f, 77 / 255f, 15 / 255f);
    public static readonly Color COLOR_WHITELAND = Color.white;
```

```
}

using UnityEngine;

public class BiomesUI : MonoBehaviour
{
    [SerializeField] private GameObject _2DCamera;
    [SerializeField] private GameObject _3DCamera;
    [SerializeField] private GameObject _spritePrefab;
    [SerializeField] private Map3DGenerator _3DGeneration;
    private MapGenerationController _biomes;
    private GameObject _2Dparent;
    private GameObject _3Dparent;
    private bool _is3DGenerated = false;

    void Start()
    {
        _2Dparent = new GameObject("2D Map");
        _3Dparent = new GameObject("3D Map");
        _biomes   = new MapGenerationController(_spritePrefab, _3DGeneration);
        _biomes.GenerateStartMap(_2Dparent);
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            _biomes.GenerateStartMap(_2Dparent);
        }
        if (Input.GetKey(KeyCode.F))
        {
            _biomes.GenerateLands();
        }
        if (Input.GetKeyDown(KeyCode.G))
        {
            _biomes.GenerateBeach();
        }
        if (Input.GetKeyDown(KeyCode.H))
        {
            _biomes.GenerateRocks();
        }
        if(Input.GetKey(KeyCode.J))
        {
            _biomes.GenerateWhiteLand();
        }
        if (Input.GetKey(KeyCode.K))
        {
            _biomes.GenerateWoods();
        }
        if(Input.GetKeyDown(KeyCode.M))
        {
            if (!_is3DGenerated)
            {
                _is3DGenerated = true;
                _2DCamera.SetActive(false);
                _3DCamera.SetActive(true);
                _2Dparent.SetActive(false);
                _biomes.Generate3DWorld(_3Dparent);
            }
        }
    }
}
```

```
public enum BiomesType
{
   LAND = 1,
   SEA = 2,
   SAND = 3,
   SEA_SHORE = 4,
   WOODS = 5,
   WHITE_LAND = 6,
   ROCK = 7,
}
using System;
using UnityEngine;

namespace BiomeCreator
{
   public class WhiteLandCreator : MonoBehaviour
   {
      private bool _isCreated = false;
      public void CreateWhiteLand(ref BiomesType[,] matrix, Action UpdateGridColor)
      {
         if(!_isCreated)
         {
            _isCreated = true;
            SetStartWhiteLandPoint(ref matrix, UpdateGridColor);
         }
         else
         {
            SpreadWhiteLand(ref matrix, UpdateGridColor);
         }
      }

      private void SetStartWhiteLandPoint(ref BiomesType[,] matrix, Action UpdateGridColor)
      {

         int rows = GenerationSettings.Rows;
         int cols = GenerationSettings.Columns;

         // Знайти випадкову клітинку LAND та зробити її WHITE_LAND
         for (int attempt = 0; attempt < 100; attempt++) // Обмежимо кількість спроб
         {
            int x = UnityEngine.Random.Range(0, rows);
            int y = UnityEngine.Random.Range(0, cols);

            if (matrix[x, y] == BiomesType.LAND)
            {
               matrix[x, y] = BiomesType.WHITE_LAND;
               UpdateGridColor.Invoke(); // Оновлюємо кольори після зміни                                    return;
            }
         }
      }

      private void SpreadWhiteLand(ref BiomesType[,] matrix, Action UpdateGridColor)
      {
         int rows = GenerationSettings.Rows;
         int cols = GenerationSettings.Columns;

         // Проходимо по всій карті та шукаємо WHITE_LAND
         for (int x = 0; x < rows; x++)
         {
            for (int y = 0; y < cols; y++)
            {
               if (matrix[x, y] == BiomesType.WHITE_LAND)
```

```
                {
                    // Пробуємо розповсюджувати навколо
                    TryConvertToWhiteLand(matrix, x - 1, y);
                    TryConvertToWhiteLand(matrix, x + 1, y);
                    TryConvertToWhiteLand(matrix, x, y - 1);
                    TryConvertToWhiteLand(matrix, x, y + 1);
                }
            }
        }

        // Оновлюємо кольори після обробки
        UpdateGridColor.Invoke();
    }

    private void TryConvertToWhiteLand(BiomesType[,] matrix, int x, int y)
    {
        if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
        {
            if (matrix[x, y] == BiomesType.LAND)
            {
                matrix[x, y] = BiomesType.WHITE_LAND; // Перетворюємо LAND на WHITE_LAND
            }
        }
    }
}
using System;
using UnityEngine;

namespace BiomeCreator
{
    public class StartMapCreator : MonoBehaviour
    {
        private void ClearPreviousGrid(GameObject[,] grid)
        {
            if (grid == null) return;

            for (int i = 0; i < grid.GetLength(0); i++)
            {
                for (int j = 0; j < grid.GetLength(1); j++)
                {
                    if (grid[i, j] != null)
                    {
                        Destroy(grid[i, j]);
                    }
                }
            }

            // Очищаємо сітку
            grid = new GameObject[GenerationSettings.Rows, GenerationSettings.Columns];
        }

        public void CreateStartMatrix(ref BiomesType[,] matrix, ref GameObject[,] grid,
            GameObject parent, GameObject spritePrefab, Action<GameObject, BiomesType> paintPixel)
        {
            ClearPreviousGrid(grid);
            int rows = GenerationSettings.Rows;
            int cols = GenerationSettings.Columns;
            matrix = new BiomesType[rows, cols];
            var random = new System.Random();

            for (int i = 0; i < rows; i++)
```

```
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i, j] = random.Next(1, 3) == 1 ? BiomesType.SEA : BiomesType.LAND;

                // Створення квадрата
                Vector3 position = new Vector3(i * GenerationSettings.BasicX, j * GenerationSettings.BasicY, 0);
                var sprite = UnityEngine.Object.Instantiate(spritePrefab, position, Quaternion.identity, parent.transform);
                grid[i, j] = sprite;

                // Початковий колір
                paintPixel(sprite, matrix[i, j]);
            }
        }
    }
}
using System;
using UnityEngine;

namespace BiomeCreator
{
    public class SandCreator : MonoBehaviour
    {
        private bool CheckNeighborForSand(BiomesType[,] matrix, int x, int y)
        {
            if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
            {
                return matrix[x, y] == BiomesType.SEA;
            }
            return false;
        }

        public void CreateBeach(ref BiomesType[,] matrix, ref GameObject[,] grid, Action<GameObject, BiomesType>
paint)
        {

            int rows = GenerationSettings.Rows;
            int cols = GenerationSettings.Columns;

            for (int x = 0; x < rows; x++)
            {
                for (int y = 0; y < cols; y++)
                {
                    if (matrix[x, y] == BiomesType.LAND)
                    {
                        bool hasSeaNeighbor = false;
                        if (CheckNeighborForSand(matrix, x - 1, y - 1)) hasSeaNeighbor = true;
                        if (CheckNeighborForSand(matrix, x - 1, y)) hasSeaNeighbor = true;
                        if (CheckNeighborForSand(matrix, x - 1, y + 1)) hasSeaNeighbor = true;

                        if (CheckNeighborForSand(matrix, x, y - 1)) hasSeaNeighbor = true;
                        if (CheckNeighborForSand(matrix, x, y + 1)) hasSeaNeighbor = true;

                        if (CheckNeighborForSand(matrix, x + 1, y)) hasSeaNeighbor = true;
                        if (CheckNeighborForSand(matrix, x + 1, y - 1)) hasSeaNeighbor = true;
                        if (CheckNeighborForSand(matrix, x + 1, y + 1)) hasSeaNeighbor = true;

                        if (hasSeaNeighbor)
                        {
                            matrix[x, y] = BiomesType.SAND;
                            paint(grid[x, y], BiomesType.SAND);
```

```
                }
            }
        }
    }
}


}
using System;
using UnityEngine;

namespace BiomeCreator
{
    public class RockCreator : MonoBehaviour
    {
        private void CheckNeighbor(BiomesType[,] matrix, int x, int y, ref int counterSea, ref int counterLand)
        {
            if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
            {
                if (matrix[x, y] == BiomesType.SEA)
                {
                    counterSea++;
                }
                else if (matrix[x, y] == BiomesType.LAND)
                {
                    counterLand++;
                }
            }
        }

        public void CreateRocks(ref BiomesType[,] matrix, Action UpdateGridColor)
        {
            int rows = GenerationSettings.Rows;
            int cols = GenerationSettings.Columns;

            BiomesType[,] newMatrix = (BiomesType[,])matrix.Clone();

            for (int x = 0; x < rows; x++)
            {
                for (int y = 0; y < cols; y++)
                {
                    if (matrix[x, y] == BiomesType.LAND)
                    {
                        int counterLand = 0;
                        int counterSea = 0;

                        CheckNeighbor(matrix, x - 1, y - 1, ref counterSea, ref counterLand);
                        CheckNeighbor(matrix, x - 1, y, ref counterSea, ref counterLand);
                        CheckNeighbor(matrix, x - 1, y + 1, ref counterSea, ref counterLand);

                        CheckNeighbor(matrix, x, y - 1, ref counterSea, ref counterLand);
                        CheckNeighbor(matrix, x, y + 1, ref counterSea, ref counterLand);

                        CheckNeighbor(matrix, x + 1, y - 1, ref counterSea, ref counterLand);
                        CheckNeighbor(matrix, x + 1, y, ref counterSea, ref counterLand);
                        CheckNeighbor(matrix, x + 1, y + 1, ref counterSea, ref counterLand);

                        if ((counterLand == 3 || counterLand == 6 || counterLand == 7 || counterLand == 8) &&
UnityEngine.Random.value > 0.9f)
                        {
                            newMatrix[x, y] = BiomesType.ROCK;
```

```
                }
            }
        }
    }

    matrix = newMatrix;
    UpdateGridColor.Invoke();
  }
 }
}
using UnityEngine;

public class Map3DGenerator : MonoBehaviour
{
  [SerializeField] private GameObject _landCube;
  [SerializeField] private GameObject _sandCube;
  [SerializeField] private GameObject _seaCube;
  [SerializeField] private GameObject _woodCube;
  [SerializeField] private GameObject _whiteCube;
  [SerializeField] private GameObject _rockCube;
  private Transform _parent;

  private const float noiseScale = 0.1f; // Масштаб шуму
  private const int minHeight = 1;
  private const int maxHeight = 15;
  private const int waterLevel = 1;

  private void SanCorrect(out GameObject cube, int x, int h, int y)
  {
    if(h - 3 >= 0)
    {
      h -= 3;
    }
    else if(h - 2 >= 0)
    {
      h -= 2;
    }
    else if (h - 1 >= 0)
    {
      h -= 1;
    }

    Vector3 position = new Vector3(x, h, y);
    cube = Instantiate(_sandCube, position, Quaternion.identity, _parent);
  }

  public void Generate3DMap(BiomesType[,] matrix, GameObject parent)
  {
    int rows = matrix.GetLength(0);
    int cols = matrix.GetLength(1);

    _parent = parent.transform;
    for (int x = 0; x < rows; x++)
    {
      for (int y = 0; y < cols; y++)
      {
        float perlinValue = Mathf.PerlinNoise(x * noiseScale, y * noiseScale);
        int height = Mathf.RoundToInt(Mathf.Lerp(minHeight, maxHeight, perlinValue));

        if (matrix[x, y] == BiomesType.SEA)
        {
          height = waterLevel;
```

```
            }

            for (int h = 0; h < height; h++)
            {
                GameObject cube;
                Vector3 position = new Vector3(x, h, y);
                switch (matrix[x, y])
                {
                    case BiomesType.LAND:
                        cube = Instantiate(_landCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.SAND:
                        SanCorrect(out cube, x, h, y);
                        break;
                    case BiomesType.WHITE_LAND:
                        cube = Instantiate(_whiteCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.ROCK:
                        cube = Instantiate(_rockCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.WOODS:
                        cube = Instantiate(_woodCube, position, Quaternion.identity, _parent);
                        break;
                    case BiomesType.SEA:
                        cube = Instantiate(_seaCube, position, Quaternion.identity, _parent);
                        break;
                    default:
                        break;
                }
            }
        }
    }
}
using System;

namespace BiomeCreator
{
    public class LandCreator
    {
        private void CheckNeighbor(BiomesType[,] matrix, int x, int y, ref int counterSea, ref int counterLand)
        {
            if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
            {
                if (matrix[x, y] == BiomesType.SEA)
                {
                    counterSea++;
                }
                else if (matrix[x, y] == BiomesType.LAND)
                {
                    counterLand++;
                }
            }
        }

        public void GenerateLands(ref BiomesType[,] matrix, Action UpdateGridColor)
        {
            int rows = GenerationSettings.Rows;
            int cols = GenerationSettings.Columns;

            BiomesType[,] newMatrix = (BiomesType[,]) matrix.Clone();
```

```
            for (int x = 0; x < rows; x++)
            {
                for (int y = 0; y < cols; y++)
                {
                    int counterSea = 0;
                    int counterLand = 0;

                    // Перевірка сусідів
                    CheckNeighbor(matrix, x - 1, y - 1, ref counterSea, ref counterLand);
                    CheckNeighbor(matrix, x - 1, y, ref counterSea, ref counterLand);
                    CheckNeighbor(matrix, x - 1, y + 1, ref counterSea, ref counterLand);

                    CheckNeighbor(matrix, x, y - 1, ref counterSea, ref counterLand);
                    CheckNeighbor(matrix, x, y + 1, ref counterSea, ref counterLand);

                    CheckNeighbor(matrix, x + 1, y, ref counterSea, ref counterLand);
                    CheckNeighbor(matrix, x + 1, y - 1, ref counterSea, ref counterLand);
                    CheckNeighbor(matrix, x + 1, y + 1, ref counterSea, ref counterLand);

                    // Логіка зміни біома
                    if (matrix[x, y] == BiomesType.LAND)
                    {
                        if (counterSea == 3 || counterSea == 6 || counterSea == 7 || counterSea == 8)
                        {
                            newMatrix[x, y] = BiomesType.SEA;
                        }
                    }
                    else if (matrix[x, y] == BiomesType.SEA)
                    {
                        if (counterLand == 3 || counterLand == 6 || counterLand == 7 || counterLand == 8)
                        {
                            newMatrix[x, y] = BiomesType.LAND;
                        }
                    }
                }
            }

            // Оновлюємо матрицю і кольори
            matrix = newMatrix;
            UpdateGridColor.Invoke();
        }
    }
}

using System;
using UnityEngine;

namespace BiomeCreator
{
    public class DarkForestCreator : MonoBehaviour
    {
        private bool _isCreated = false;

        public void CreateWoods(ref BiomesType[,] matrix, Action UpdateGridColor)
        {
            if (!_isCreated)
            {
                _isCreated = true;
                SetStartWoodsLandPoint(ref matrix, UpdateGridColor);
            }
            else
            {
```

```
            SpreadWoodsLand(ref matrix, UpdateGridColor);
        }
    }

    private void SetStartWoodsLandPoint(ref BiomesType[,] matrix, Action UpdateGridColor)
    {

        int rows = GenerationSettings.Rows;
        int cols = GenerationSettings.Columns;

        // Знайти випадкову клітинку LAND та зробити її WHITE_LAND
        for (int attempt = 0; attempt < 100; attempt++) // Обмежимо кількість спроб                    {
            int x = UnityEngine.Random.Range(0, rows);
            int y = UnityEngine.Random.Range(0, cols);

            if (matrix[x, y] == BiomesType.LAND)
            {
                matrix[x, y] = BiomesType.WOODS;
                UpdateGridColor.Invoke(); // Оновлюємо кольори після зміни
                return;
            }
        }
    }

    private void SpreadWoodsLand(ref BiomesType[,] matrix, Action UpdateGridColor)
    {
        int rows = GenerationSettings.Rows;
        int cols = GenerationSettings.Columns;

        // Проходимо по всій карті та шукаємо WHITE_LAND
        for (int x = 0; x < rows; x++)
        {
            for (int y = 0; y < cols; y++)
            {
                if (matrix[x, y] == BiomesType.WOODS)
                {
                    // Пробуємо розповсюджувати навколо
                    TryConvertToWhiteLand(matrix, x - 1, y);
                    TryConvertToWhiteLand(matrix, x + 1, y);
                    TryConvertToWhiteLand(matrix, x, y - 1);
                    TryConvertToWhiteLand(matrix, x, y + 1);
                }
            }
        }

        // Оновлюємо кольори після обробки
        UpdateGridColor.Invoke();
    }

    private void TryConvertToWhiteLand(BiomesType[,] matrix, int x, int y)
    {
        if (x >= 0 && y >= 0 && x < GenerationSettings.Rows && y < GenerationSettings.Columns)
        {
            if (matrix[x, y] == BiomesType.LAND)
            {
                matrix[x, y] = BiomesType.WOODS; // Перетворюємо LAND на WHITE_LAND
            }
        }
    }
  }
}
```