КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня бакалавра

за спеціальністю 121 Інженерія програмного забезпечення
на тему:

ПРОЦЕДУРНА ГЕНЕРАЦІЯ ЛАНДШАФТІВ ДЛЯ ІГРОВИХ ПРОГРАМ НА ОСНОВІ СИМПЛЕКС ШУМУ ТА ТРІАНГУЛЯЦІЇ ДЕЛОНЕ

Виконав студент 4-го курсу
Роман ГОФМАН _____

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Максим ВЕРЕС _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«____» _____ 202_ р.,
протокол № ___
Завідувач кафедри
Олександр ПРОВОТАР _____

Київ – 2021

РЕФЕРАТ

Дипломна робота: 46 сторінок, 32 рисунки, 11 джерел.

Ключові слова: ГЕНЕРАЦІЯ ЛАНДШАФТУ, ІГРОВІ ПРОГРАМИ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, СИМПЛЕКС ШУМ, ТРІАНГУЛЯЦІЯ ДЕЛОНЕ, UNITY.

Об’єкт дослідження: методи для генерації ландшафтів для ігрових програм.

Мета роботи: дослідження методів створення ігрових ландшафтів на основі алгоритмів симпекс шуму та тріангуляції Делоне.

Методи та інструменти розроблення: Unity 2020.3.9, мова програмування C#.

Рекомендації щодо використання роботи: для дослідження методів генерації ландшафтів для ігрових програм.

Сфера застосування: розробка відеоігор.

Значимість роботи: наведено використання різнотипних методів генерації та способи їх комбінування.

Висновки: були проаналізовані методи генерації нескінченних ландшафтів для ігрових програм, ефективність їх роботи та сфери застосування.

ЗМІСТ

ВСТУП **4**

РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ГЕНЕРАЦІЇ ЛАНДШАФТІВ **6**

1.1 Принципи генерації ландшафтів 6

1.2 Особливості генерації нескінченних ландшафтів 7

1.3 Огляд існуючих аналогів 9

1.4 Огляд алгоритмів шумів 10

1.4.1 Загальні відомості 10

1.4.2 Шум Перлина 11

1.4.3 Алгоритм Diamond-Square 13

1.4.4 Симплекс шум 14

1.5 Огляд алгоритмів поділу 17

1.5.1 Загальні відомості 17

1.5.2 Тріангуляція Делоне 18

1.6 Способи комбінування методів генерації 19

РОЗДІЛ 2 ПРАКТИЧНА ЧАСТИНА РОЗРОБКИ **23**

2.1 Засоби розробки 23

2.1.1 Ігровий двигун 23

2.1.2 Мова програмування 26

2.2 Імплементація 28

2.2.1 Генерація на основі алгоритмів шумів 28

2.2.2 Генерація на основі тріангуляції Делоне 34

2.2.3 Модель нескінченного ландшафту 37

2.2.4 Покращення отриманих результатів 39

2.3 Конфігурування 42

ВИСНОВКИ **44**

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ **45**

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. Розробка ігрових програм складається з великої кількості етапів. Одним з найважливіших є створення ігрового світу, що включає в себе розробку простору, в якому гравець буде перебувати. Враховуючи велику кількість ігор, їх жанрів та швидких темпів їх розробки, на створення ігрового простору накладають обмеження, що значно ускладнюють цей процес. Серед них – розмірність гри, позиція камери, розмір та специфіка ігрового світу.

Ігровий простір може бути динамічно згенерований, створений вручну або з використанням обох методів. При розробці ігор з нескінченним або випадково створеним світом доцільна лише випадкова генерація. Для більшості ігор з невипадковим світом фіксованого розміру використовується гібридний метод. Динамічна процедурна генерація вирішує проблему випадковості, оскільки базується на використанні генератора випадкових чисел, що дає можливість створювати світи довільного розміру, в тому числі нескінченного.

Актуальність роботи та підстави для її виконання. Наразі існує багато інструментів для генерацій ландшафту, в тому числі декорацій для зйомок фільмів: World Machine, World Creator, Instant Terra, плагіни для ігрових двигунів такі, як terraintoolkit, тощо. В більшості випадків при розробці інді-проектів використання таких потужних інструментів може бути зайвим. Створюваний прототип може використовуватися для генерації простих ландшафтів та легко конфігуруватися.

Мета й завдання роботи. Метою цієї роботи є розробка прототипу для генерації нескінченного ігрового ландшафту з використанням різних підходів, на прикладі симплекс шуму та тріангуляції.

Для досягнення визначеної мети необхідно виконати наступні завдання:

1. Розглянути сфери застосування алгоритмів симплекс шуму та тріангуляції Делоне.
2. Проаналізувати кожний з алгоритмів окремо.
3. Проаналізувати способи комбінування даних алгоритмів.
4. Розробити прототип для демонстрації роботи алгоритмів.
5. Налаштувати параметри алгоритмів для різних типів генерованих ландшафтів.

Об’єкт, методи й засоби дослідження або розроблення. Одним з найбільш поширених методів є генерація на основі алгоритмів шумів. Більшість з них мають невелику складність та можуть бути детально сконфігуровані, зокрема для досягнення реалістичності. Однак в деяких випадках використання шумів не є достатнім. Такі проблеми можуть бути вирішені модифікаціями алгоритму (в основному складними), або комбінуванням з іншими методами, наприклад такими, що розбивають ландшафт на частини. Серед них – тріангуляція та діаграми Вороного.

Можливі сфери застосування. Робота може використовуватися при розробці невеликих ігрових програм з наперед заданим ігроми світом або генерації окремих рівнів. Також можливе використання для ігор з нескінченним світом.

РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ГЕНЕРАЦІЇ ЛАНДШАФТІВ

1.1 Принципи генерації ландшафтів

Задача побудови ландшафту полягає у знаходженні координати zдля кожної точки площини (x; y). Для представлення згенерованого ландшафту будемо використовувати матрицю $n \times m$, значення якої відповідають висоті відповідних точок. Значення цієї матриці можуть бути як цілочисельними, так і з рухомою комою.

Для відображення на площині буде доцільним використання значень з проміжку [0; 256], що дає змогу перетворити значення в точці на колір, де 0 - чорний, а 256 - білий. Тоді згенерований ландшафт у 2D просторі буде мати вигляд чорно-білої текстури розміру $n \times m$ пікселів, де світлі її частини будуть позначати виступи, а темні – западини. Інші комбінації кольорів можуть використовуватися для позначення шарів ландшафту, кліматичних зон, типів поверхні, недосяжних зон, тощо.

Генерація ландшафту для ігрових програм має такі особливості:

а) максимізація ігрового простору

Ландшафт має бути створений з урахуванням специфіки роботи двигуна гри та можливості взаємодії на максимальній площі, якщо це передбачено ігровою логікою. Він має відповідати особливостям світу, але не перешкоджати та запобігати ігровому процесу.

б) відсутність недосяжних ділянок

При деяких типах ігрових світів, потрібно запобігати можливого випадкового створення ділянки, навколо неактивних елементів, які перешкоджають прохід або іншу взаємодію з нею. Такі випадки потребують додаткової обробки, а в загальному випадку – алгоритму для їх видалення або уникнення.

в) врахування генерації ігрових об’єктів

Процес генерації ігрового світу не обмежується створенням поверхні. Алгоритм має враховувати розміщення окремих об’єктів або великих структур. Для них мають бути виділені окремі, часто рівні, ділянки відповідного розміру.

г) багаторівневість світу

Особливості роботи ігрового двигуна можуть вимагати розділення ландшафту на шари, що потребує модифікації існуючого алгоритму або пошуку інших підходів для генерації.

Для отримання реалістичних результатів генерація висот площини не є достатньою. На створення ландшафтів в природі впливає безліч факторів, більшість з яких неможливо відтворити в програмній реалізації. Одним з ефективних методів трансформації згенерованого простору для збільшення його реалістичності є послідуюча ерозія. В спрощеному для нашого випадку вигляді ерозія розчиняє матеріал з крутих схилів та переносить його вниз, де він накопичується на менших нахилах. Це призводить до того, що круті схили стають ще більш крутими, а маловисотна місцевість вирівнюється. Враховуючи той факт, що такий процес займає тривалий час, його програмна реалізація може бути реалізована у вигляді послідовності окремих кроків [5].

1.2 Особливості генерації нескінченних ландшафтів

Оскільки процес генерації базується на заповненні прямокутної матриці значеннями, то для нескінченних ландшафтів він буде полягати у послідовному заповненні матриць такого ж, або іншого розміру, як зображено на рис. 1.1, де матриця під номером 1 є початковою.

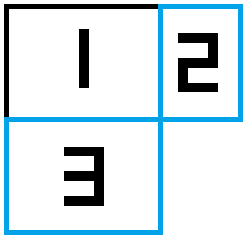

Рисунок 1.1 – Приклад генерації суміжних матриць

Можливість налаштування розміру суміжних матриць дозволяє контролювати навантаження процесу та запобігати генерації частин ландшафту, які не будуть використовуватися під час ігрового процесу.

Більш комплексним підходом є часткова генерація великих ділянок та повна генерація їх частин. Як зображено на рис. 1.2, матриця під номером 1 є початковою, а 2, 3 та 4 суміжними частково згенерованими. При цьому частини цих матриць, що розміщені найближче до початкової матриці (зображені червоним кольором) будуть повністю згенерованими, а інші частини 2, 3 та 4 – лише частково.

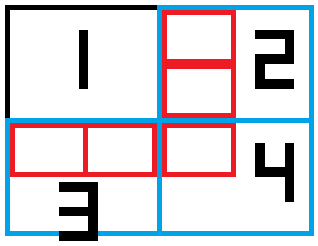

Рисунок 1.2 – Приклад часткової генерації суміжних матриць

Такий підхід дає можливість відображати далекі частини ландшафту без їх повної обробки. Вони будуть в поганій якості, але враховуючи відстань до них цим можна знехтувати, оскільки це ніяк не позначиться на ігровому процесі. Цей метод покращить швидкодію та візуальну частину гри. В окремих випадках методи вибору розміру та позицій генерованих матриць потребують окремого алгоритму, для відповідності вимогам ігрового процесу та швидкодії.

1.3 Огляд існуючих аналогів

Програма World Machine

а) безкоштовна у базовій комплектації

б) не підтримує нескінченну генерацію, однак платні версії можуть генерувати надвеликі мапи

в) згенерована мапа може конфігуруватися вручну використовуючи фільтри пензля або фігури

г) не має підтримки розміщення об'єктів

д) доступна на Windows та (неофіційно) MacOS

Програма/Плагін World Creator

а) не має безкоштовної версії

б) не підтримує нескінченну генерацію, однак може генерувати надвеликі мапи

в) має великий асортимент фільтрів та модифікаторів генератора

г) має підтримку розміщення різних об'єктів, процедурної генерації трави та іншої рослинності

д) доступна на Windows та MacOS як окрема програма, або як плагін для Unity

Плагін MapMagic

а) не має безкоштовної версії

б) підтримує генерацію нескінченних ландшафтів

в) не має інструментів для ручного налаштування або корегування згенерованого ландшафту, кількість доступних модифікаторів значно менша, ніж у конкурентів

г) має підтримку розміщення різних об’єктів

д) доступний на всіх платформах Unity

1.4 Огляд алгоритмів шумів

1.4.1 Загальні відомості

Функція шуму *noise(x, y)* повертає набір випадкових значень, в залежності від параметрів x та y, які виступають координатами на площині. Функція шуму може бути створена таким чином, що згенеровані значення будуть відрізнятися на деяку випадкову величину ε від оточуючих:

а) значення a_i залежить від a_{i-1} та a_{i+1}

б) a_{i-1} та a_i залежать від b_k; a_i та a_{i+1} залежать від b_{k+1}

Алгоритми шумів мають широке використання. Якщо розглядати їх в якості методів генерації 2D та 3D об'єктів, то вони можуть застосовуватися для створення випадкових текстур (каменю, землі), природних явищ (хмар, туману), руху рідин (вогню, води) та спецефектів (розмивання, спотворення зображення). Також їх можна використовувати для для імітації рукописного малюнку або тексту, шляхом незначного спотворення форми запису.

В більшості випадків алгоритми шумів мають невелику складність та є простими у імплементації. Вони базуються на генераторі випадкових чисел, що дозволяє не обмежуватися розмірами генерованої площини. Це дає змогу використовувати їх для генерації об'ємних ландшафтів, в тому числі нескінченних. Слід зауважити, що деякі алгоритми можуть використовувати наперед згенеровану матрицю випадкових чисел заданого розміру, що дещо ускладнює процес створення нескінченних ландшафтів, однак дозволяє значно покращити швидкодію процесу генерації [6].

Алгоритми шумів мають широкі можливості конфігурування для відповідності різним типам ігрових світів. Часто їх використання є достатнім для генерації природного ландшафту, оскільки він базується на випадковій послідовності висот, де кожна наступна відрізняється від попередньої на деяке

випадкове невелике ε. Більш складніші випадки потребують використання інших методів, або модифікацію алгоритму з урахуванням відповідних умов світу та ігрового простору.

Серед найпоширеніших алгоритмів шумів, що використовуються для генерації ландшафтів можна виділити:

а) шум Перлина

в) алгоритм Diamond-Square

г) симплекс шум

1.4.2 Шум Перлина

Шум Перлина розроблений Кеном Перлином у 1983 році. Він належить до градієнтних шумів. Шум Перлина зазвичай реалізується як дво-, три-, або чотиривимірна функція, але може бути визначений довільною кількістю вимірів. Реалізація складається з трьох кроків: визначення сітки, обчислення скалярного добутку градієнтних векторів, та інтерполяція між цими значеннями. Його складність – $O(2^n)$, тому для кожної точки у двовимірній сітці процес вимагатиме 4 операції, у тривимірній — 8.

Основним недоліком алгоритму є його складність при великій розмірності сітки. Навіть у тривимірному просторі кожна множина складається з восьми точок, тому необхідно виконати вісім скалярних добутків та сім лінійних інтерполяцій. Для задачі генерації ландшафту на площині, тобто у двовимірному просторі, шум Перлина можна вважати одним з кращих варіантів.

Для генерації детальних ландшафтів використовується метод накладання отриманих частин, кожна з яких більш детальна за попередню, але має меншу вагу. Детальність визначається зміщенням відповідних координат, тому більш детальна частина має більший розмір, але масштабована до розміру

попередньої частини. На рис. 1.3 пунктирною лінією зображено результуючу частину для одновимірного простору, а сірими лініями частини, які накладаються, де лінії позначені більш насиченим кольором мають більшу вагу.

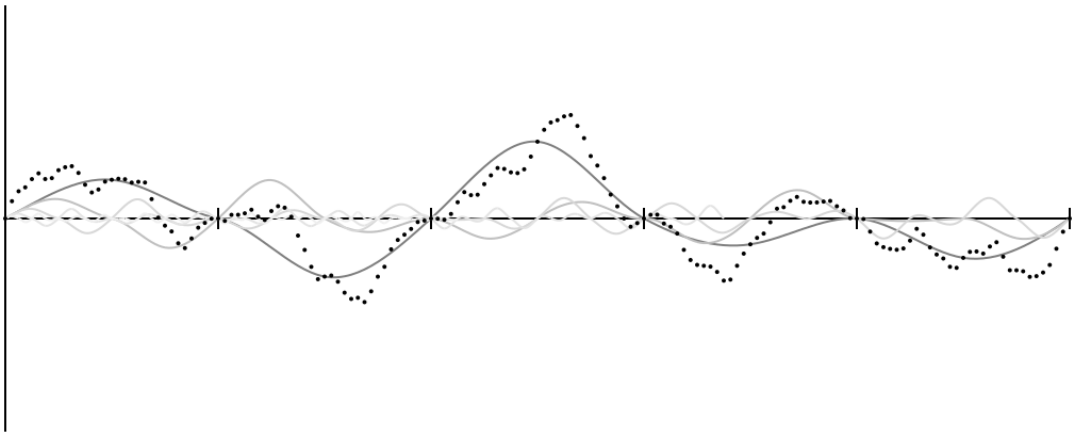

Рисунок 1.3 – Приклад накладання згенерованих частин різної детальності

Кожна з таких частин називається октавою. На рис. 1.4 зображені текстури згенеровані алгоритмом шуму Перлина з двома та чотирма октавами відповідно. Для більш гладких ландшафтів доцільно використовувати меншу кількість октав. Додавання великої кількості октав не має сенсу, оскільки для частин з великим номером октави їх вага буде занадто мала. Такі випадки можуть негативно впливати на швидкодію, тому потребують детального попереднього налаштування.

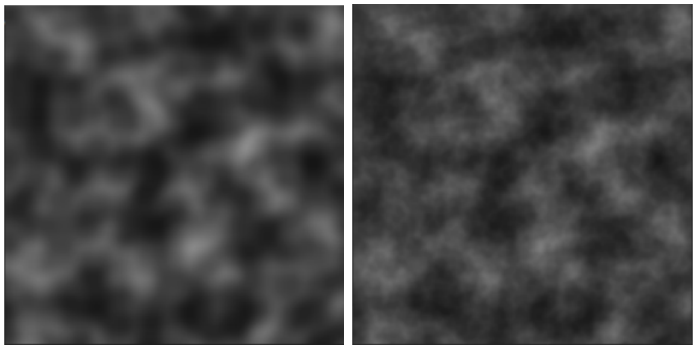

Рисунок 1.4 – Приклад текстури згенерованої шумом Перлина з різною кількістю октав

1.4.3 Алгоритм Diamond-Square

Ідея цього алгоритму вперше була озвучена на конференції siggraph в 1982 році. Він полягає у покроковому заповненні сітки, на основі оточуючих значень у формі ромба або квадрата, як зображено на рис. 1.5.

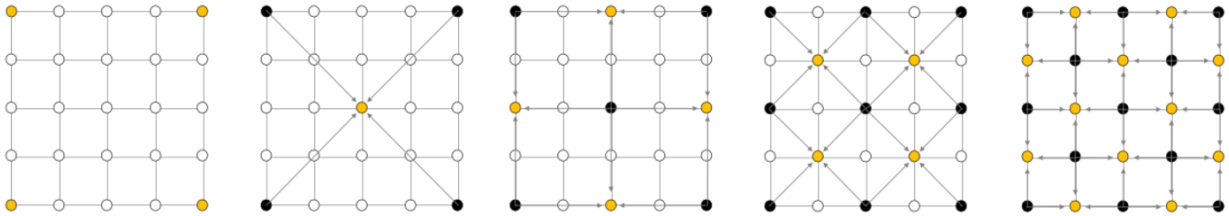

Рисунок 1.5 – Алгоритм заповнення сітки в Diamond-Square

Алгоритм diamond-square починає роботу з двовимірного масиву розміру $2^n + 1$. У чотирьох кутових точках масиву встановлюються початкові значення висот. Кроки diamond і square виконуються по черзі до тих пір, поки всі значення масиву не будуть встановлені [3].

Крок diamond (ромб): для кожного квадрата в масиві, встановлюється серединна точка, якій присвоюється середнє арифметичне з чотирьох кутових точок плюс випадкове значення.

Крок square (квадрат): для кожного ромба в масиві, знаходиться серединна точка, в яку встановлюється середнє значення чотирьох кутових точок плюс випадкове значення.

Випадкове число зазвичай обирається в проміжку $[-Ri, Ri]$, де R це фактор нерівності в проміжку від 0 до 1, а i це номер ітерації. Відповідно, при кожній ітерації випадкове значення, яке додають до серединних точок, зменшується.

З недоліків алгоритму можна виділити (див. рис. 1.6):

а) нерівномірність значень на межах сітки (позначено блакитним кольором): як видно з прикладу, висота на межах сітки майже завжди більше середньої.

б) можливі артефакти у вигляді ліній під кутом 90 або 45 градусів (позначено червоним кольором)

в) розмірність сітки лише $2^n + 1$

г) складнощі об'єднання кількох згенерованих частин: можливі артефакти, як і у пункті б на межі частин

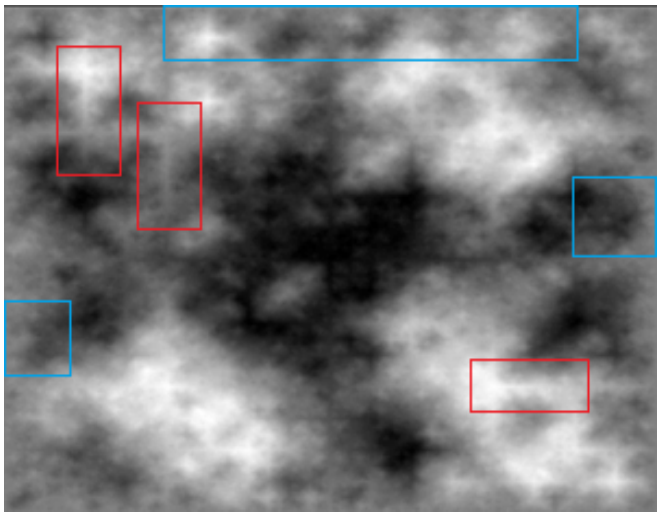

Рисунок 1.6 – Приклад текстури згенерованої алгоритмом Diamond-Square

1.4.4 Симплекс шум

Кен Перлин розробив алгоритм у 2001 році щоб усунути обмеження функції шуму Перлина, особливо за великої кількості вимірів. В той час як класичний шум інтерполює значення градієнтів вузлових точок сітки, симплекс-шум поділяє простір на симплекси (тобто n-вимірні трикутники). Це зменшує кількість вузлів. Гіперкуб у n вимірах має 2^n кутів, симплекс — лише $n + 1$. У 2D трикутники рівносторонні.

Переваги симплекс-шуму над шумом Перлина:

а) має меншу обчислювальну складність, бо вимагає менше операцій множення

б) за більшої кількості вимірів є продуктивнішим, складність – $O(n^2)$

в) не має помітних артефактів, проте вигляд погіршується зі збільшенням кількості вимірів

г) має добре визначений неперервний градієнт, який обчислюється доволі швидко

д) симплекс-шум легко реалізувати

Алгоритм складається з наступних кроків:

а) відхилення координат

б) поділ на симплекси

в) вибір градієнта

г) сумування ядер

Спочатку вхідні координати xта y трансформують за формулою

$$x` = x + (x + y) * F$$

$$y` = y + (x + y) * F,$$

де $F = \frac{\sqrt{n+1}-1}{n}$. Вихідні координати $(x`, y`)$ дозволяють визначити, в яку комірку зміщеного одиничного гіперкуба потрапляє вхідна точка,

$$(x_b` = floor(x`), y_b` = floor(y`)),$$

і її внутрішні координати

$$(x_i` = x` - x_b`, y_i` = y` - y_b`).$$

Коли вищезгадане визначено, значення внутрішніх координат $(x_i`, y_i`)$ сортуються за спаданням, щоб визначити, якій ортосхемі Шлефлі симплекса належить точка. Тоді результуючий симплекс складають з вершин, що відповідають впорядкованому обходу від (0,0, …, 0) до (1, 1, …, 1), і є $n!$ варіантів, що можуть відповідати одній перестановці координат. Іншими словами, починаємо від нульової координати, потім послідовно додаємо ті, що відповідають найбільшому значенню внутрішньої координати, закінчуючи відповідником найменшого значення. Наприклад, точка (0.4, 0.5, 0,3) буде

лежати всередині симплекса з вершинами (0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1). Значення по осі ординат найбільше, тому визначення координат симплекса починатимемо з нього. Потім збільшуємо значення по осі абсцис, і так далі.

Кожна вершина симплекса додається до базової координати відхиленого гіперкуба, потім хешується у псевдовипадковий градієнтний вектор. Варто звертати увагу на вибір множини градієнтів, щоб мінімізувати появу артефактів.

Внесок кожної з *n + 1* вершин симплекса враховується за допомогою суми радіально симетричних ядер з центром у кожній вершині. Спочатку за допомогою оберненої формули визначаються невідхилені координати:

$$x = x` + (x` + y`) * G$$

$$y = y` + (x` + y`) * G,$$

де $G = \frac{1/\sqrt{n+1}-1}{n}$. Ця точка віднімається від вхідних координат, щоб отримати невідхилений вектор переміщення. Цей вектор потрібен для того, щоб порахувати екстрапольоване значення градієнта з використанням скалярного добутку та квадрат відстані до точки.

Далі внесок ядра кожної суми визначається з рівняння

$$(r^2 - d^2)^4 * (< \Delta x, \Delta y > \cdot < grad.x, grad.y >),$$

де r^2 зазвичай обирають рівним 0,5 чи 0,6. 0,5 забезпечує відсутність розривів, в той час як 0,6 може покращити зовнішній вигляд у застосунках, де розриви непомітні. 0,6 використовувався у оригінальній реалізації Кена Перлина.

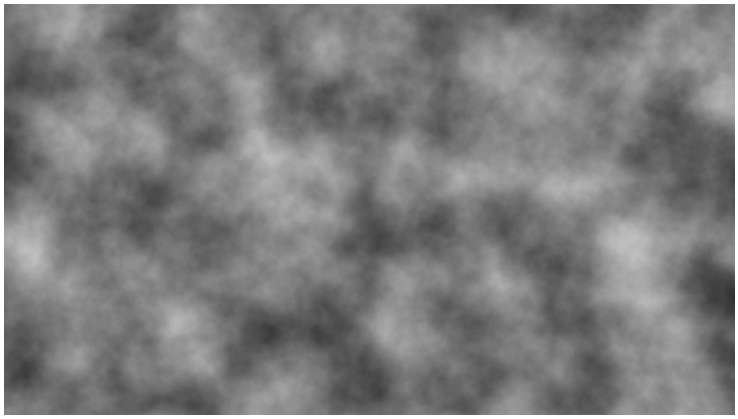

Рисунок 1.7 – Приклад текстури згенерованої симплекс шумом

1.5 Огляд алгоритмів поділу

1.5.1 Загальні відомості

Поділ ландшафту на частини може використовуватися для високорівневої генерації локацій та діючих зон ігрового простору. В більшості випадків використання таких методів для створення ландшафту не є доцільним або вимагає значних зусиль для адаптації алгоритму. Тому часто їх комбінують з методами зазначеними вище, для покращення або корегування отриманого результату. Інший спосіб використання такого підходу – часткова генерація, тобто попереднє налаштування ландшафту перед комбінуванням з іншими методами або виділення окремих його ділянок. Результат залежить від попереднього розміщення опорних точок, між якими надалі будуються ребра, тому алгоритм можна легко сконфігурувати під різні типи та розміри місцевості [4].

Для випадкового поділу можна використовувати діаграми Вороного або тріангуляцію Делоне. В більшості випадків методи побудови таких складних структур вимагають значних розрахунків, порівняно з методами на основі алгоритмів шумів, тому їх часто уникають. Але для створення специфічних або

реалістичних ландшафтів ці алгоритми є доцільними. Приклади часткового та повного поділу для методу тріангуляції зображені на рис. 1.8.

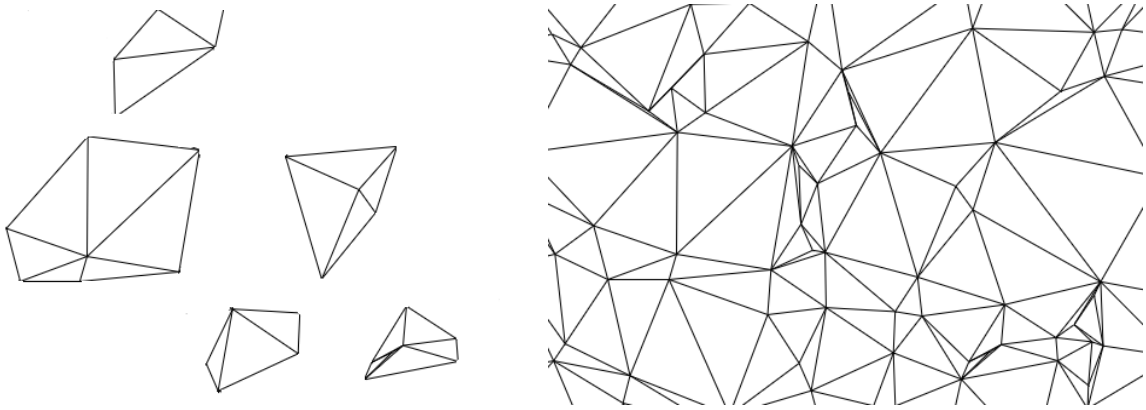

Рисунок 1.8 – Приклади часткового та повного поділу

При частковому поділі нескінченного ландшафту варто створювати ділянки ближче до середини оброблювальної частини, аби уникати проблем пов’язаних з їх подальшим об’єднанням. Однак якщо алгоритм інкрементний або працює по принципу “розділяй і володарюй”, то такі випадки не є проблемними. При повному поділі об’єднання ділянок є неминучим.

Ланцюги ребер отриманих з поділу можна використати для створення природних явищ (гірського хребта, річки) або доріг. Поодинокі фігури або їх об’єднання можуть слугувати водоймами, поселеннями або іншими локаціями.

1.5.2 Тріангуляція Делоне

Тріангуляція Делоне – це така тріангуляція, що жодна точка не знаходиться всередині описаних навколо трикутників кіл. Вона дозволяє зменшити кількість малих кутів. Цей спосіб тріангуляції був винайдений Борисом Делоне в 1934 році.

З’єднання центрів описаних кіл трикутників утворених тріангуляцією Делоне, які мають спільне ребро, утворює діаграму Вороного (див рис. 1.9).

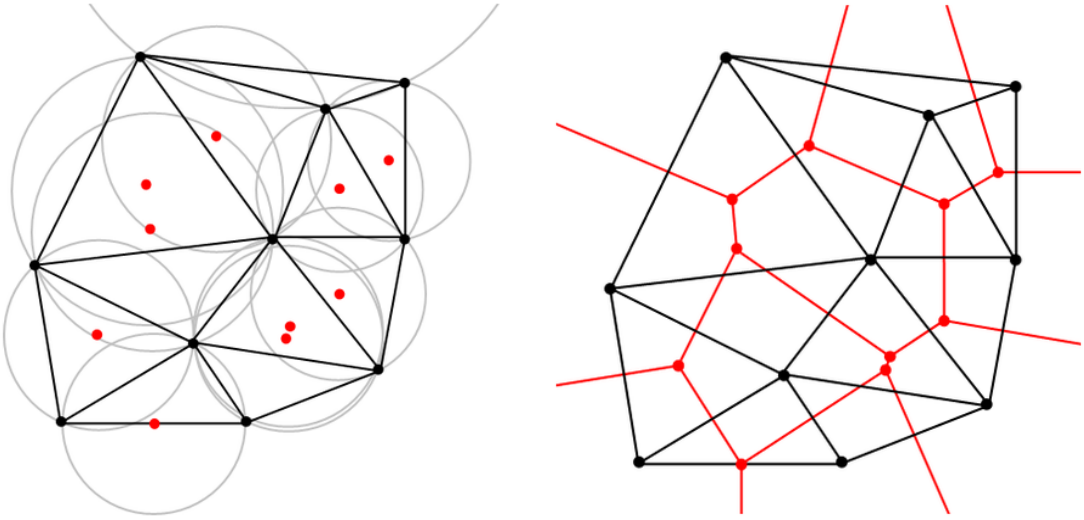

Рисунок 1.9 – Зв’язок Тріангуляція Делоне з діаграмою Вороного

Існує кілька алгоритмів знаходження тріангуляції Делоне:

а) заміни ребра: побудувати хоч якусь тріангуляцію, а потім замінювати ребра аж поки вона не буде задовольняти умові Делоне

б) інкрементний: додавати вершину, та розбивати трикутник, що її містить на три частини, а потім застосувати алгоритм заміни ребра

в) розділяй і володарюй: ділимо всі вершини на дві множини прямою, будуємо тріангуляцію Делоне для кожної з них, потім об’єднуємо їх в одну

г) замітання: алгоритм Форчуна на основі подій, які виникають внаслідок перетину з замітаючою прямої, яка рухається вздовж всіх точок на площині

Основною перевагою тріангуляції Делоне є максимізація найменшого кута в трикутниках, тому отримані фігури будуть більш гладкими та мати приблизно однакову форму. Це зменшить кількість небажаних артефактів та дозволить уникати їх подальшого видалення.

1.6 Способи комбінування методів генерації

У випадку використання алгоритмів поділу для побудови матриці висот отримані результати будуть нереалістичними. Декілька прикладів наведені на рис. 1.10. На перших двох показана мапа з інкрементною генерацією, тобто вершини утворюють виступи, а на третьому – декрементний, де вони утворюють западини. При невеликій кількості вершин такі генерації можна використовувати для попереднього окреслення островів, гірських хребтів або водойм [1].

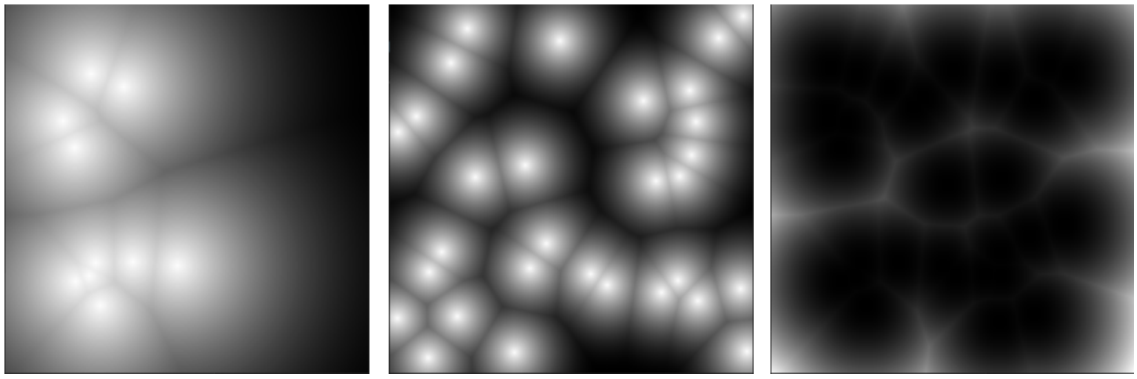

Рисунок 1.10 – Приклади генерації мапи висот на основі діаграми Вороного

Об’єднання їх з матрицею згенерованою на основі алгоритмів шумів дасть кращі результати. Для прикладу розглянемо результати комбінації алгоритмів Diamond-Square та діаграми Вороного. На рис. 1.11 наведені мапи висот згенеровані кожним з алгоритмів та результат їх об’єднання. Як видно з прикладу, кінцевий результат значно кращий, ніж використання лише алгоритму поділу, однак реалістичним його назвати не можна. Межі поділу діаграми Вороного добре помітні, тому це буде мати вигляд прямих впадин між горами [2].

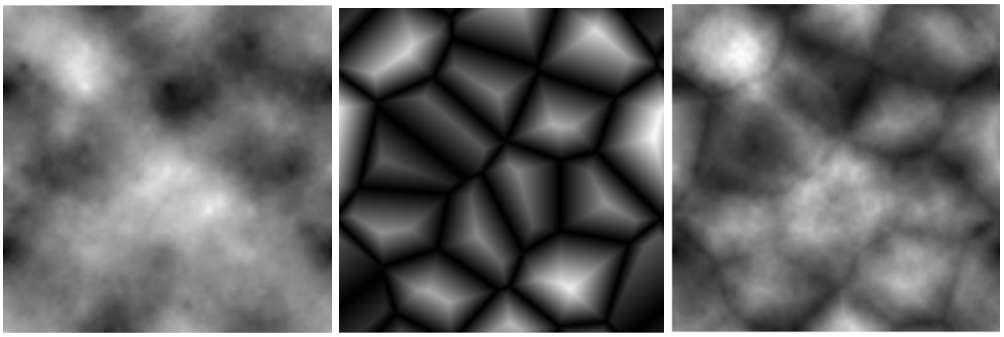

Рисунок 111 – Комбінація алгоритмів Diamond-Square та діаграми Вороного

Для підвищення реалістичності можна виконати декілька кроків пертурбації. Цей процес полягає у довільному невеликому зміщенні частин текстури, яке загладить явні артефакти та додасть мапі випадковості. Результат зображений на рис. 1.12. Як видно з прикладу впадни більше не помітні, а гори виглядають більш природними та випадковими.

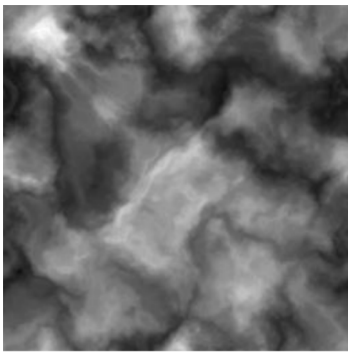

Рисунок 1.12 – Комбінація алгоритмів Diamond-Square та діаграми Вороного з ефектом пертурбації

Такі об’єднання можуть бути конфігуровані вагою кожної з мап. Об’єднання більше двох таких текстур рідко дає прийнятні результати, або його можна звести до об’єднання лише двох текстур з іншими налаштуваннями. Алгоритм об’єднання полягає у сумуванні значень обох мап з відповідними коефіцієнтами:

$z` = c * z_1 + (1 - c) * z_2$, де c–вага першої мапи, $c \in (0;\ 1)$.

Як приклад об'єднання з мапою побудованою на базі алгоритму часткового поділу можна навести генерацію річки. Як показано на рис. 1.13 (виділені блакитним кольором) їх шляхи базуються на ребрах многокутників, які утворилися після побудови діаграми Вороного [4].

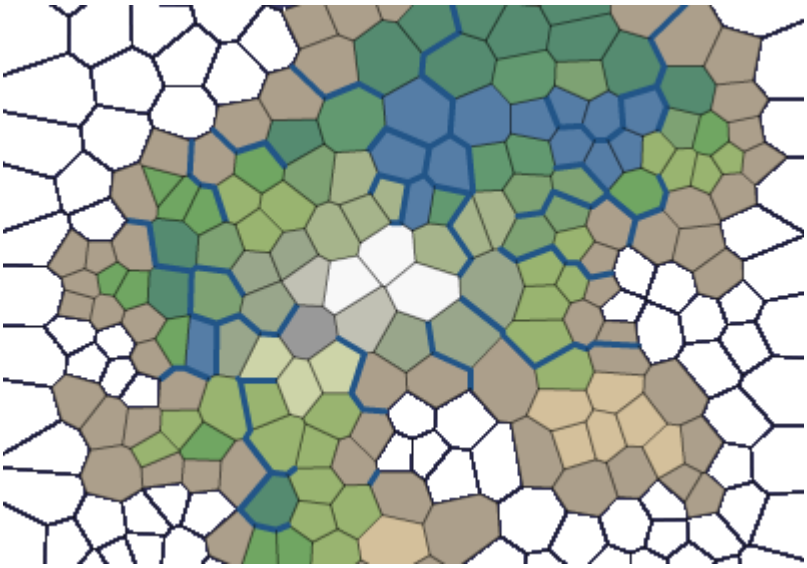

Рисунок 1.13 – Генерація шляху річки на основі діаграми Вороного

Також поділ може використовуватися для позначення кліматичних зон або типу поверхні, окреслення окремих ігрових ділянок або локацій. Як показано на рис. 1.13, білим кольором навколо фігури позначений неігровий простір, а блакитним кольором – водойми. Кольори від зеленого до коричневого означають тип поверхні та відповідну кліматичну зону.

РОЗДІЛ 2 ПРАКТИЧНА ЧАСТИНА РОЗРОБКИ

2.1 Засоби розробки

2.1.1 Ігровий двигун

Іноді для представлення згенерованого ландшафту недостатньо чорно-білої текстури. Якість незначних модифікацій таких, як пертурбація, згладжування та ерозія буде добре помітна лише у тривимірному представленні. На моделювання поведінки камери та відображення базових елементів можна витратити багато часу, тому доцільним буде використання ігрового двигуна, де ці задачі уже вирішені. Це дасть змогу сфокусуватись на безпосередньо алгоритмах генерації ландшафтів.

Для відображення ландшафту та його конфігурування буде використовуватися ігровий двигун Unity. Це багатоплатформний інструмент для розробки відеоігор і застосунків, і рушій, на якому вони працюють. Створені за допомогою Unity програми працюють на настільних комп'ютерах, мобільних пристроях та гральних консолях у дво- та тривимірній графіці, та на пристроях віртуальної чи доповненої реальності. Застосунки, створені за допомогою Unity, підтримують DirectX та OpenGL.

Редактор Unity має інтерфейс, що складається з різних вікон та віджетів, які можна розташувати на свій розсуд. Завдяки цьому можна проводити налаштування програми прямо в редакторі та спостерігати за відповідними змінами.

Проект в Unity складається зі сцен. Це окремі файли зі своїми об’єктами, налаштуваннями, сценаріями, тощо. Сцени також можуть містити об'єкти які не мають моделі, проте задають поведінку інших об'єктів (тригери, точки збереження та інші). Їх дозволяється розташовувати, обертати, масштабувати, застосовувати до них скрипти. Кожний з них має назву, може мати тег та шар, а

якому він має бути відображений Тому у кожного об’єкта на сцені обов'язково наявний компонент Transform, який зберігає в собі координати місця розташування, повороту і розміру. У об'єктів з мають бути відображеними за умовчанням присутній компонент Mesh Renderer, що робить модель видимою. Декілька моделей можуть об’єднуватися в ассети для швидкого доступу до них.

Unity підтримує фізику твердих тіл, тканини, тощо. У редакторі присутня система наслідування об’єктів: дочірні сутності будуть повторювати всі зміни позиції, повороту і масштабу батьківської сутності. Скрипти в редакторі прикріплюються до об'єктів у вигляді окремих компонентів.

У 2D іграх Unity переважно використовує спрайти, а в 3D іграх – тривимірні моделі, на які накладаються текстури, які надають сутності колір та зображення, матеріали, які визначають як поверхня реагуватиме на різні фактори та шейдери, які вираховують зміну кольору кожного пікселя згідно заданих параметрів (наприклад розсіювання відбитого світла). В обох видах застосовуються системи часток для відображення субстанцій, таких як рідини, вогонь та дим.

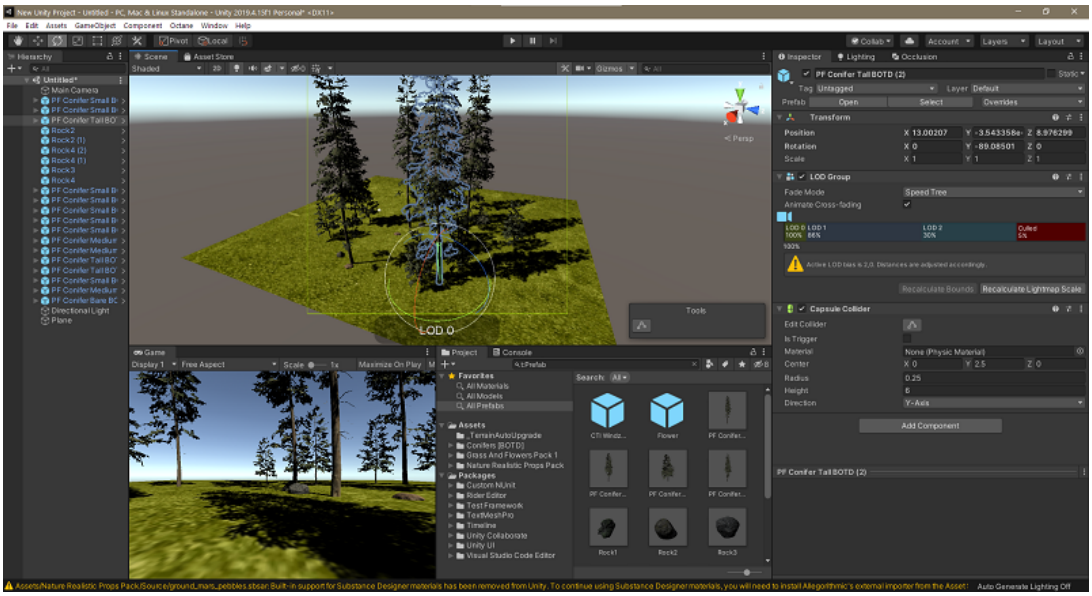

Рисунок 2.1 – Приклад роботи в Unity

Цей ігровий двигун підтримує стиснення текстур, міпмапінг і різні конфігурації роздільності екрана для відповідних платформи, забезпечує бамп-мапінг, паралакс-мапінг, мапінг відображень, затінення світла, динамічні тіні за їх картами та повноекранні ефекти, такі як глибина чіткості, зернистість, розмиття в русі, відблиски або ореол навколо джерел світла [11].

Зображення рендериться через віртуальну камеру огляду. В робочій області редактора ігрова сцена може розміщуватися як завгодно, а при рендерингу – так, як її видно з камери. Камери можуть рухатися за об’єктами чи за визначеною траєкторією. Зображення з камери може бути двовимірним чи тривимірним (в перспективі або ортографічно). Також доступні налаштування оточення, що за умовчанням має вигляд скайбоксу.

Скриптова система ігрового рушія розроблена на Mono, який є відкритим проектом з реалізацією на .NET Framework. Також можна використовувати власну скриптову мову UnityScript, подібну до ECMAScript та JavaScript, Boo або C#. Починаючи з версії 3.0, до Unity входить перероблена версія MonoDevelop. Починаючи з версії 5.2 э можливість редагувати скрипти в інтегрованому середовищі розробки Visual Studio.

Unity включає Unity Asset Server – систему контролю версій для ігрових об'єктів та скриптів на базі PostgreSQL. Для роботи зі звуком використовує роботу бібліотеку FMOD (з можливістю програвати Ogg Vorbis аудіофайли), а для відео – кодек Theora. Також присутній двигун для побудови ландшафтів рослинності, вбудовану систему мап освітлення, мережу для багатокористувацьких ігор та вбудовані навігаційні об’єкти для пошуку шляхів [10].

2.1.2 Мова програмування

Однією з основних мов програмування в ігровому двигуні Unity є C#. Це об’єктно-орієнтована мова програмування з безпечною типізацією. Розроблена Скотом Вілтамутом, Андерсом Гейлсбергом та Пітером Гольде для платформи .NET, яка належить Microsoft. Синтаксис цієї мови С-подібний, тому близький до Java та C++. Мова підтримує поліморфізм, має статичну типізацію, вказівники на функції-члени класів, перевантаження операторів, атрибути, винятки, коментарі, події та інше. Спираючись на практику використання попередніх мов (C++, Object Pascal і Smalltalk), C# не має деяких проблематичних моделей, наприклад множинного успадкування класів.

Мова програмування C# дуже близька до Java, яка була створена в той час, коли з’явилася потреба у розподілених обчисленнях. Java взяла за основу мову C++, не включаючи потенційно небезпечні моделі такі, як вказівники без контролю виходу за межі. Також була створена концепція машинно-незалежного байт-коду та віртуальної машини, яка знаходилася між апаратним рівнем та кодом програми, чим набула значної популярності [7].

Microsoft почала створювати власний аналог Java, яким вона зможе володіти. Ця мова отримала назву C#. Вона унаслідувала від Java концепції байт-коду (MSIL), віртуальної машини (середовище .NET), і більшої безпеки вихідного коду програм, за рахунок врахування досвіду програмування на Java.

C# полегшила взаємодію з кодом програм, написаних на інших мовах програмування, що позитивно позначилося на розробці великих проектів. Коли програми, написані на різних мовах, виконуються на платформі .NET, то вона дбає про сумісність програм та типів даних.

Станом на сьогодні, C# називають мовою корпорації Microsoft, бо вона найповніше використовує нові можливості .NET та займає головну позицію у покращенні існуючого та створенні нового функціоналу цієї платформи. Ця

мова є однією з найбільш потужних та розвивається досить швидко, а платформа .NET існує достатньо довго та налічує безліч можливостей [9].

Можна виділити наступні переваги цієї платформи:

а) підтримка декількох мов програмування

Платформа побудована на базі загальномовного середовища CLR, тому вона підтримує декілька мов програмування (C#, VB.NET, C++, F#) та діалекти інших мов

б) мультиплатформеність

Використовуючи технології на платформі .NET можна розробляти застосунки для Windows, Linux, MacOS, Android iOS та Tizen.

в) велика бібліотека класів

Незалежно від створюваного застосунку програмісти так чи інакше використовують бібліотеку класів

г) технології

Загальномовне середовище CLR та базова бібліотека класів є основою для великого стеку технологій для роботи з базами даних, графічних програм, мобільних застосунків, веб-сервісів, тощо

д) швидкодія

Проведені тести показують, що веб-додатки створені на базі .NET 5 сильно випереджають веб-додатки, побудовані за допомогою інших технологій. Це стосується і інших видів програм.

Також слід відзначити таку особливість мови C # і фреймворка .NET, як garbage collection. В більшості випадків, на відміну від C++, програмістам не доводиться керувати деалокацією пам’яті, що значно прискорює та полегшує процес розробки [8].

2.2 Імплементація

2.2.1 Генерація на основі алгоритмів шумів

Для представлення мапи будемо використовувати двомірний масив типу float, значення якого знаходяться в проміжку [0; 1], де 0 - рівень моря, а 1 максимальна висота, причому значення які перевищують 1 будемо вважати за 1. На основі цих даних будуть будуватися полігони з різними координатами по осі *y*, що і буде визначати висоту ландшафту у тривимірному представленні.

Для зручності відображення буде доцільним розбити весь проміжок можливих значень висоти на шари та зафарбовувати полігони в кольори шару, в якому вони знаходяться. Кольори та рівні шарів можуть конфігуруватися для визначати різних клітамичних зон та типів ландшафтів. Для прикладу будемо використовувати наступні шари:

а) водойма [0; 0.2) синій колір

б) водойма [0.2; 0.3) блакитний колір

в) берег [0.3; 0.45) жовтий колір

г) долина [0.45; 0.65) зелений колір

д) долина [0.65; 0.8) темно-зелений колір

е) гірська місцевість [0.8; 0.9) коричневий колір

є) гірська місцевість [0.9; 1) темно-коричневий колір

ж) гірська вершина [1; ∞) білий колір

Розмірність вихідного масиву 250x250.

Функція знаходження значення шуму в конкретній точці буде мати наступну сигнатуру:

float SimplexNoise(int x, int y),

де *x* та *y* відповідні координати обчислювальної точки на мапі.

Алгоритм ініціалізується параметром seed. Він використовується для ініціалізації генератора випадкових чисел. Різні значення цього параметру призведуть до повністю різного результату генерації. Для випадкових генерацій можна ініціалізувати його значенням часу інтерпретованим до вигляду цілого числа, що дасть різні результати в кожний момент часу. Інший підхід полягає у збереженні набору цих параметрів для повторної ініціалізації таких же ландшафтів у будь-який момент часу, наприклад для наперед згенерованих мап.

Алгоритм генерації мапи ініціалізується розміром ландшафту size, вказаним вище параметром seed, а також

а) *scale*. Використовується для масштабування результуючого шуму. Як видно з прикладу на рис. 2.2, зі збільшенням цього значення зменшується площа геренованого шуму. Тому для місцевості з невеликими виразними ділянками слід використовувати менші значення scale. Також вын визначає якість результуючого розраження, тому чим більше значення, тим більш детальною буде результуюча мапа.

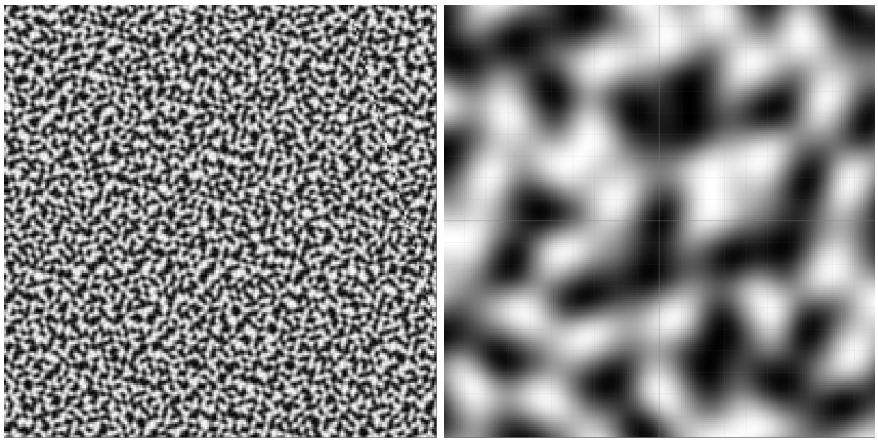

Рисунок 2.2 – Приклад використання параметру scale зі значеннями 5 та 50

б) *octaves*. Використовується для визначення кількості шарів, які будуть накладатися, причому кожний наступний шар буде більш детальним за попередній. Як зображено на рис. 2.3, зі збільшенням кількості шарів збільшується деталізація результуючого ландшафту. Для різного типу

ландшафтів може використовуватися різна кількість шарів. Так для більш гладких ділянок слід використовувати меншу кількість шарів. Для мап великого розміру можливе використання більшої кількості октав, бо вони мають бути більш детальними. Великі значення параметру не дадуть очікуваного результату, оскільки кожний наступний буде мати меншу вагу в результуючій мапі, та лише погішить швидкодію.

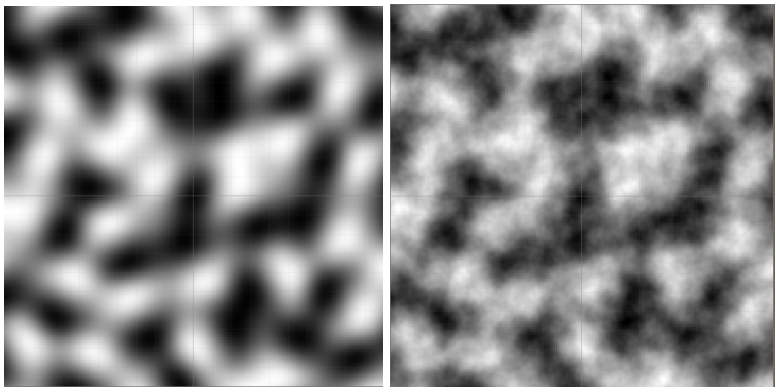

Рисунок 2.3 – Приклад використання параметру octaves зі значеннями 1 та 4

в) *persistence*. Цей параметр визначає вагу кожного з шарів, що накладаються. Чим більше це значення, тим меншу різницю ваг будуть мати сусідні шари. При значенні в 1 всі шари будуть мати однакову вагу, а в 0 – враховуватися лише перший шар.

г) *lacunarity*. Використовується для збільшення зернистості кожного з шарів. Як видно з прикладу на рис. 2.4, при більшому значенні цього параметру послідуючі шари стають менше подібні до попередніх, від чого результат має більше невеликих деталей.

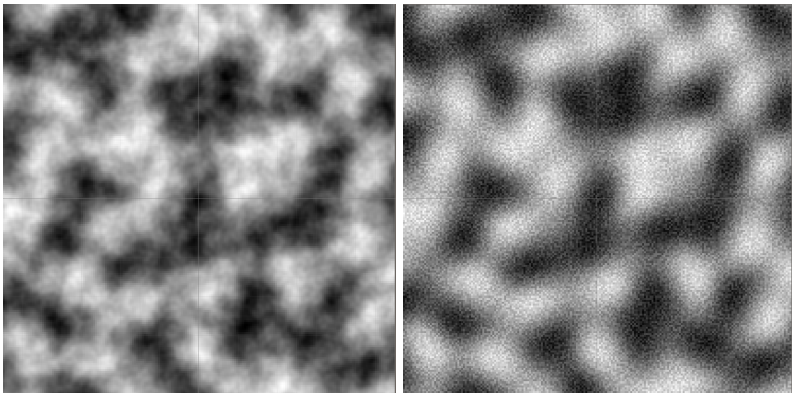

Рисунок 2.4 – Приклад використання параметру lacunarity зі значеннями 3 та 30

д) *offset*. Значення цього параметру визначають зміщення мапи по відповідним осям. На прикладі на рис. 2.5 на другому зображенні відбулося зміщення по осі x на 20 пунктів, а на третьому ще на 20 по осі y . В більшості випадків це не впливає на ландшафт, якщо він випадковий, однак це може використовуватися для специфічних ігрових механік.

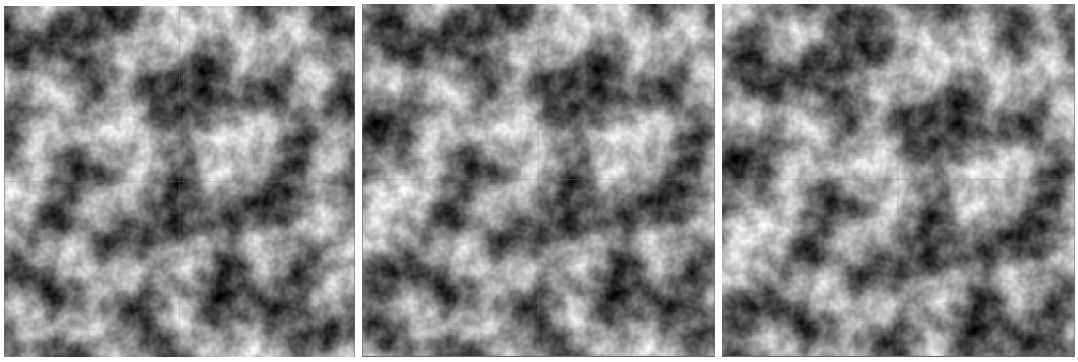

Рисунок 2.5 – Приклад використання параметру offset зі значеннями 20 пок кожній осі

Псевдокод генерації має наступний вигляд:

створюємо результуючий двовимірний масив noiseMap[ширина, висота]
ініціалізуємо генератор випадкових чисел параметром seed
для кожного рядку в мапі
  для кожного значення в рядку
    амплітуда = 1;
    частота = 1;

висота = 0;
для кожної октави
*sampleX = (x - (ширина/2)) + зміщенняоктави.x) / scale * частота;*
*sampleY = (y - (висота/2) + зміщенняоктави.y) / scale * частота;*
*шум = noise.Generate(sampleX, sampleY) * 2 - 1;*
*висота += шум * амплітуда;*
*амплітуда *= persistance;*
*частота *= lacunarity;*
noiseMap[x, y] = висота;
нормалізувати отримані значення noiseMap до [0; 1]
повернути noiseMap

Для побудови полігонів необхідно розбити результуючу мапу на трикутники, як показано на рис. 2.6. Кількість отриманих полігонів дорівнює *width * height * 2*. При розбитті за годинниковою стрілкою для кожної вершини візьмемо її нижніх на правих сусідів: для точки 1 – (2, 5, 4) для точки 2 – (3, 6, 5) і т. д. Точки з останнього ряду та стовпчику не обробляються.

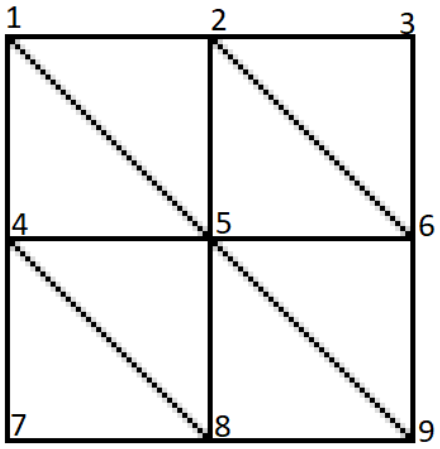

Рисунок 2.6 – Розбиття мапи на полігони

Псевдокод створення полігонів має наступний вигляд:

topLeftX = (ширина - 1) / -2;
topLeftZ = (висота - 1) / 2;
створити структуру meshData
vertexIndex = 0;
для кожного рядка, окрім останнього

```
для кожного значення, окрім останнього
    meshData.vertices[vertexIndex] =
        (topLeftX + x, висота[x, y], topLeftZ - y);
    meshData.uvs[vertexIndex] = (x / (float)width, y / (float)height);
    meshData.AddTriangle
        (vertexIndex, vertexIndex + ширина + 1, vertexIndex + ширина);
    meshData.AddTriangle
        (vertexIndex + ширина + 1, vertexIndex, vertexIndex + 1);
  інкрементуємо vertexIndex;
повернути meshData;
```

Тоді отримаємо наступні полігони: (1, 2, 5), (1, 5, 4), (2, 3, 6), (2, 6, 5) і т. д. Для відображення кожної вершини необхідно змінити значення по координаті *y* на відповідне з отриманої мапи. Кожний полігон зафарбовується у відповідний колір зазначений при розбитті на шари (шар враховується по значенню першої вершини).

Псевдокод створення кольорової текстури має наступний вигляд:

```
створити двомірний масив кольорів colorMap
для кожного рядка
  для кожного значення
    висота = noiseMap[x, y];
    для кожного шару
      якщо (висота >= шар[i].висота)
        colorMap[y * ширина + x] = шар[i].колір;
      інакше
        break;

повернути colorMap;
```

Отриманий тривимірний результат з урахуванням описах вище шарів та кольорів зображено на рис. 2.7. Як видно з другого зображення, двигун Unity дозволяє створювати власні віджети, що значно спрощує процес розробки та дозволяє відразу відобразити ландшафт для даних налаштувань. Надалі вони

будуть активно використовуватися для спроб налаштування алгоритму під різні види ландшафтів та їх шарів.

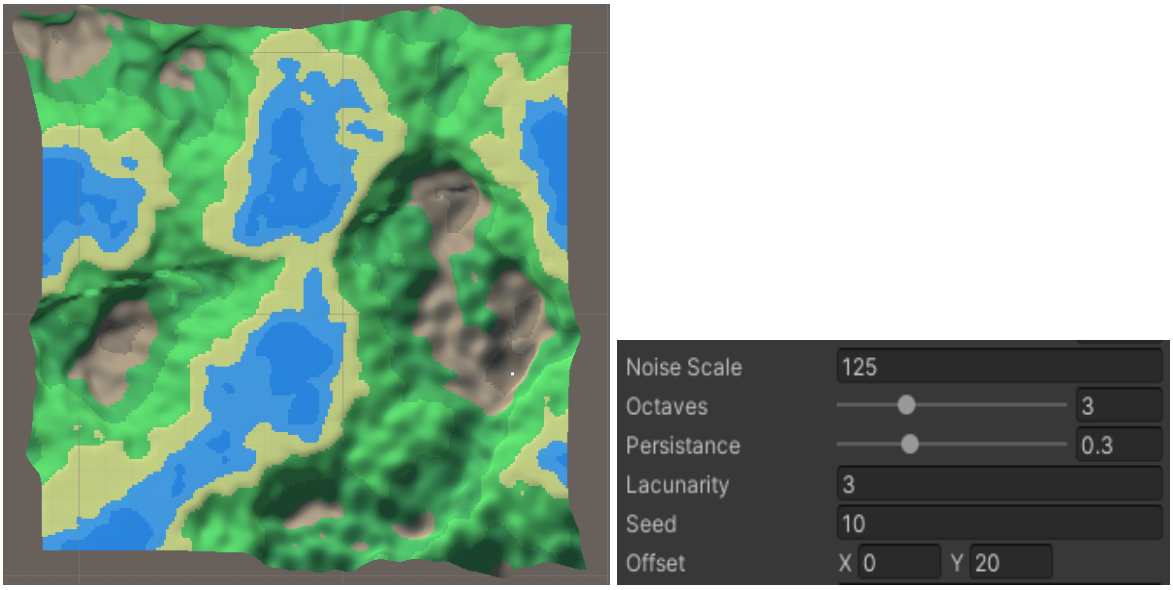

Рисунок 2.7 – Приклад отриманого результату та відповідна конфігурація

2.2.2 Генерація на основі тріангуляції Делоне

При генерації на основі алгоритмів поділу необхідно враховувати, що побудова матриці висот вимагає більших зусиль, оскільки значення в кожній точці залежить від найближчих вершин або ребер фігур (в залежності від алгоритму), що потребує більшої складності розрахунку. Тому такі алгоритму можуть використовуватися для високорівневого проектування ландшафту. Для прикладу розглянемо способи генерації біомів (зон з різною конфігурацією алгоритмів побудови ландшафту, кліматичних зон, тощо).

Кожний біом буде представляти собою набір трикутників. Для зручності представлення вони будуть позначені різними кольорами відповідно до їх типу.

Після побудови тріангуляції Делоне необхідно створити структуру даних, щоб мати можливість трансформувати ці фігури. Вибір структури даних залежить від способів їх обробки.

При використанні методу послідовної випадкової обробки трикутників, доцільним буде побудувати граф, для швидкого доступу до сусідніх трикутників. Як приклад такого алгоритму можна навести алгоритм “вклаплення”, приклад роботи якого зображений на рис. 2.8.

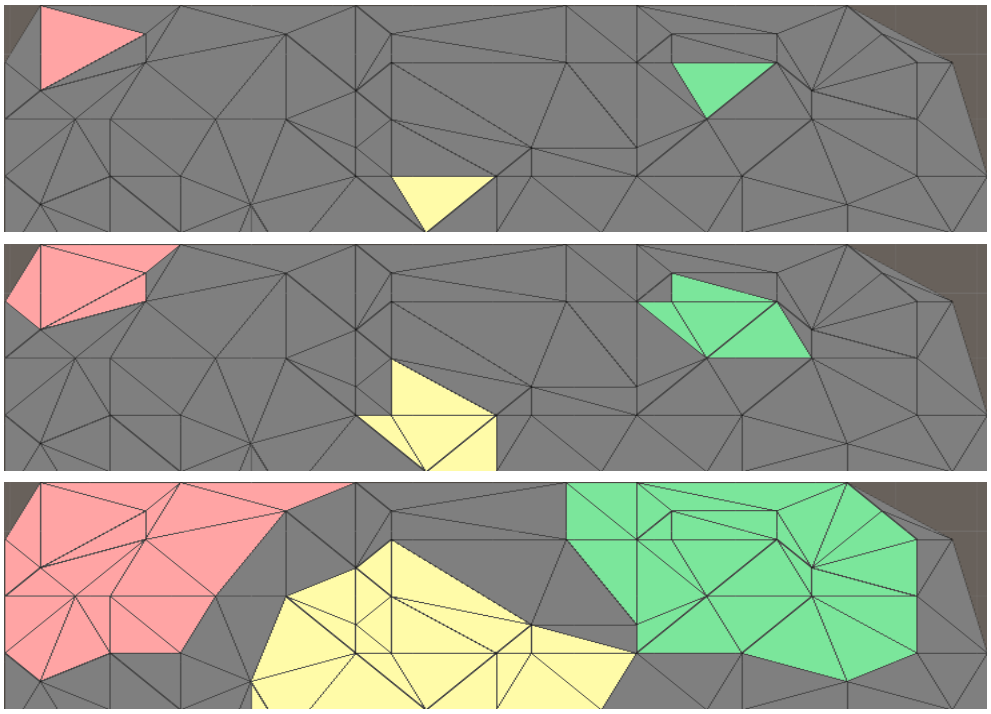

Рисунок 2.8 – Приклад роботи алгоритму “вкраплення”

Псевдокод цього алгоритму має наступний вигляд:

```
випадково обираємо N трикутників
для кожного трикутнику
    створюємо чергу Qi
    додаємо цей трикутник в чергу Qi
    оброблюємо цей трикутник

допоки хоч одна з черг Qi не порожня
    для кожної черги Qi
        переносимо всі трикутники в масив Ti
        для кожного трикутнику з масиву Ti
            якщо трикутний необроблений
                оброблюємо цей трикутник
                для кожного сусіднього трикутнику
                    якщо він не оброблений
                        додаємо в чергу Qi
```

Алгоритм є інтуїтивно зрозумілим і простим в імплементації. Слід звернути увага на вибір початкових трикутників та їх кількість: краще розміщувати їх на більшій відстані один від одного, аби оброблювані зони були приблизно однакового розміру.

Інший підхід полягає у представленні набору трикутників у вигляді сітки. Такий метод дозволяє застосовувати різні підходи до послідовності їх обробки, в тому числі алгоритми шумів. Для створення сітки можна використати рекурсивний алгоритм , який полягає у розбитті множини всіх трикутників на підмножини по якійсь з осей координат, допоки в кожній підмножині не залишиться по 1 або 0 трикутників. Приклад роботи алгоритму зображено на рис. 2.9. В ролі точок виступає середнє значення координат вершин кожного з трикутників, що дозволяє з невеликою похибкою знайти центр фігури. В результаті розбиття, отримаємо сітку, у клітинках якої або буде трикутник або ні.

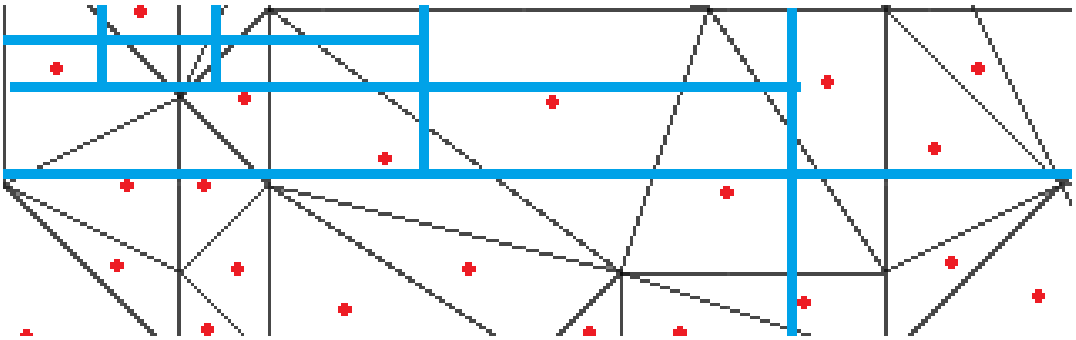

Рисунок 2.9 – Приклад розбиття трикутників у сітку

Маючи двовимірний масив його можна обробити за допомогою алгоритму шуму. У випадку якщо необхідне чітке розбиття на різні типи біомів доцільним буде модифікувати алгоритм шуму для таких випадків. Алгоритм подібний до розбиття на кліматичні зони кольорами, але отримана текстура буде мати набагато менше точок, оскільки кількість трикутників значно менша за кількість точок на мапі. Тоді кожному трикутнику в сітці присвоюється

значення біому зі згенерованої мапи з відповідними координатами. Приклад зображений на рис. 2.10.

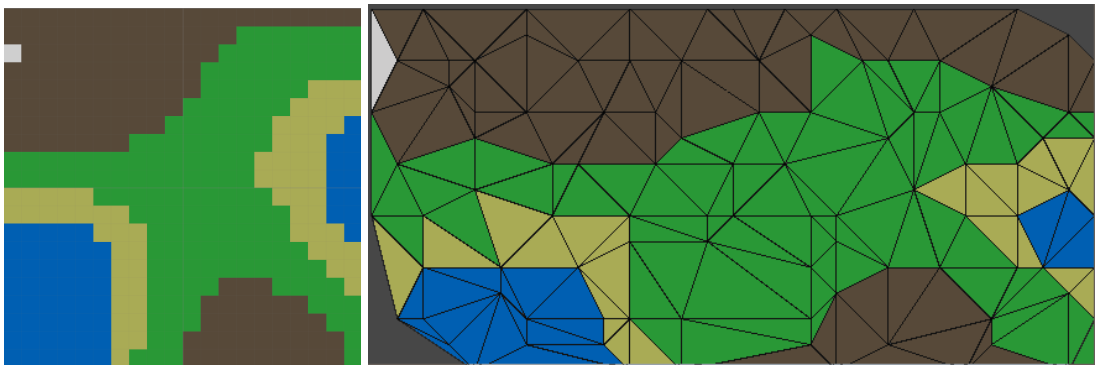

Рисунок 2.10 – Приклад шуму згенерованого для різних біомів

2.2.3 Модель нескінченного ландшафту

Задача генерації нескінченного ландшафту полягає у послідовній генерації частин мапи, що знаходяться поряд. Які саме частини будуть генеруватися виначається ігровим процесом, але в більшості випадків це позиція гравця або камери. Кількість частин, що генеруються визначається параметром максимальної дистанції, тому чим більший цей параметр, тим більше ділянок буде створено навколо вихідної позиції. На рис. 2.11 зображена модель генерації.

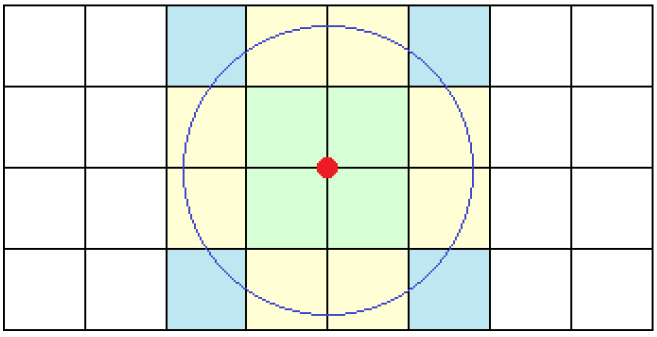

Рисунок 2.11 – Модель генерації нескінченного ландшафту

Вихідна позиція зображена червоною крапкою, тоді зелені частини будуть згенеровані повністю (1: 1), жовті – частково (1: k), блакиті – частково (1: k^2 або

$1: m$, де $m = f(k)$), а білі залишаються порожніми. Синє коло відображає максимальну дистанцію, при якій частина мапа вважається видимою. Зелені частини входять в коло повністю, жовті – більшою частиною, а блакитні – меншою частиною.

Для збереження мап та їх відповідних позицій у сітці будемо використовувати словник, де ключем буде позиція, а значенням – матриця висот. Перевагою використання словника для таких цілей є швидкий доступ за вказаним ключем.

Ігровий процес вимагає постійного оновлення відображених частин мапи, тому алгоритм має відслідковувати зміни позиції гравця. Для оптимізації обчислювального процесу оновлення буде відбуватися тільки коли гравець зміститься на деяку відстань d , яка визначається алгоритмом. Процес оновлення буде полягати у відображенні або невідображені частин мапи в залежності від пройденої гравцем відстані після останнього оновлення.

Псевдокод оновлення відображених частин мапи має наступний вигляд:

```
для кожної відображуваної частини
   сховати частину
частинаX = позиція.x / ширина;
частинаY = позиція.y / висота;
для кожної позиції в видимому радіусі
   визначаємо поточну позицію в сітці
   якщо частина з такою позицією існує
      відображаємо її
   інакше
      генеруємо нову частину
      додаємо її в словник
```

Дотепер позицією для генерації мапи вважався початок координат. Оскільки частини генеруються в різних позиціях, необхідно врахувати це при створенні матриці висот, інакше всі частини будуть мати однаковий вигляд. Для цього існує параметр offset, про який зазначалося вище. Тому замість передачі в

функцію генерації мапи висот сконфігурованого параметру offset, туди буде передаватися center + offset, це center – позиція центру генерованої частини, або позиція в сітці * (ширина, висота).

На рис. 2.12 показаний приклад генерації частин ландшафту при русі гравця вперед. Частини, що вийшли з поля зору, зникають, а нові – з’являються. Причому при повторній появі попередніх частин буде використовуватися вже згенеровані раніше мапи.

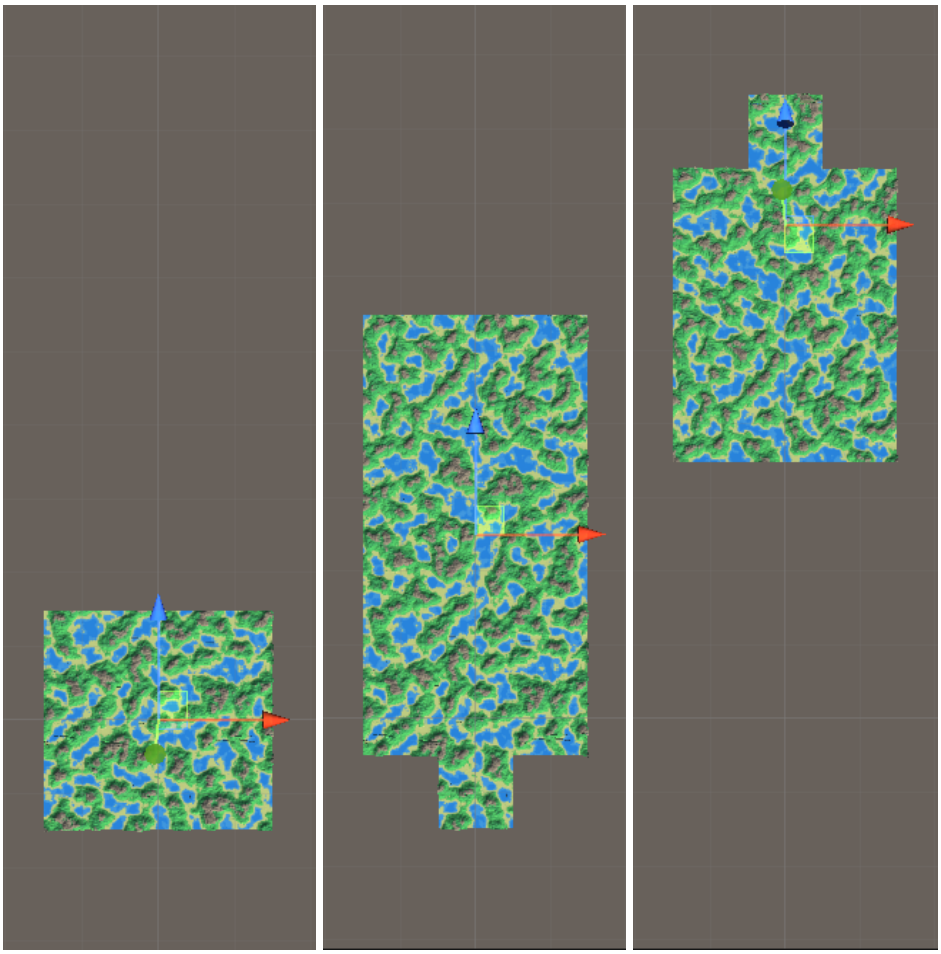

Рисунок 2.12 – Приклад генерації нескінченного ландшафту

2.2.4 Покращення отриманих результатів

Для покращення отриманих результатів можна виконати симуляцію процесу ерозії, що збільшить реалізм ландшафту. Цей процес призведе до того,

що гірська місцевість буде мати більш круті схили, а рівнинні ділянки позбудуться пагорбів. В основі його дії лежить зсув порід через дію гравітації, погодних умов, тощо. Приклад принципу роботи одного з алгоритмів ерозії наведено на рис. 2.13. Для виконання зсуву порід буде обраховуватися різниця висот з сусідніми ділянками, потім це значення буде додано або віднято від кожного зі значень.

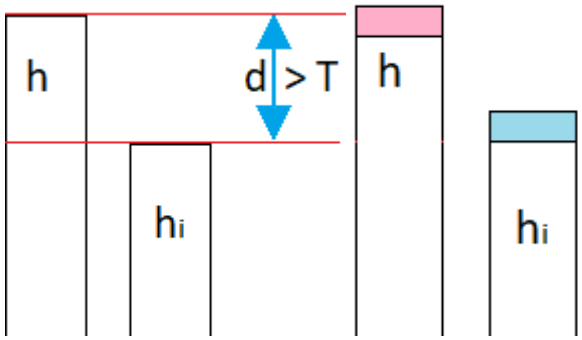

Рисунок 2.13 – Приклад роботи одного з алгоритмів ерозії

Псевдокод використаного алгоритму має наступний вигляд:

```
для кожної ділянки H
    для кожної сусідньої ділянки Hi
        D = H - Hi
        якщо D > Dmax
            Dmax = D
            imax = i
якщо 0 < Dmax < Limit
        delta = Dmax * c
        H = H - delta
        Himax = Himax + delta
```

Є декілька способів вибору сусідніх ділянок *Hi* (див. рис. 2.14):

а) брати всі 8 значень

б) брати 4 значення по краях

в) брати 4 значення по кожну сторону

Різниці між варіантами б та в майже немає, за виключенням кінцевого вигляду (кількість операцій зчитування буде однакова). При виборі першого варіанту отримані результати будуть більш точними та реалістичними.

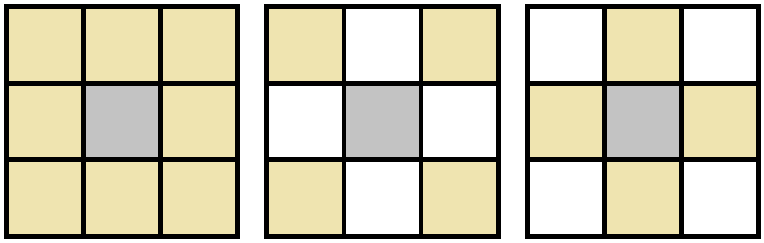

Рисунок 2.13 – Приклад вибору сусідніх значень

На рис. 2.14 та 2.15 зображені приклади генерації без та з використанням модифікатору ерозії. Налаштування алгоритму містить 3 параметри: кількість симуляцій (кроків) ерозії, мінімальна різниця висот для зсуву породи та тип вибору сусідніх ділянок.

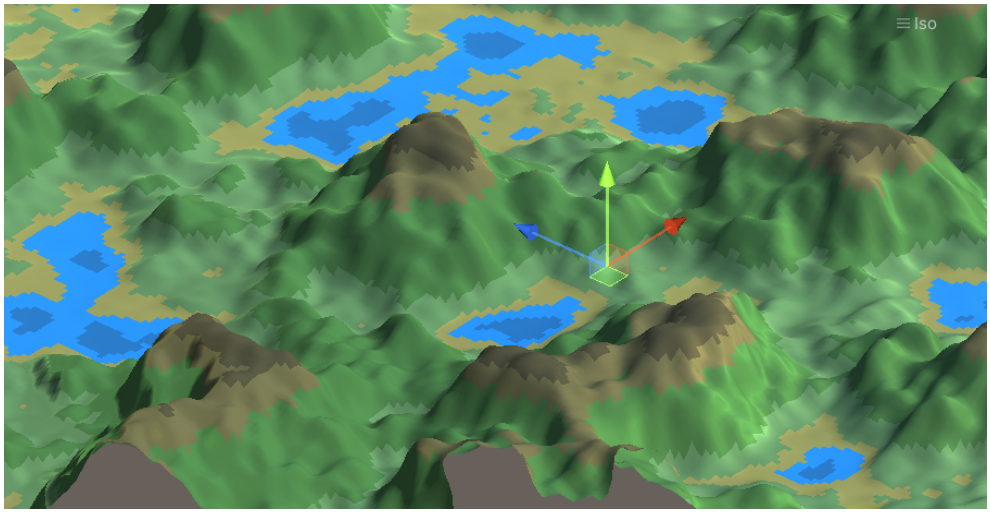

Рисунок 2.14 – Приклад генерації без модифікатору ерозії

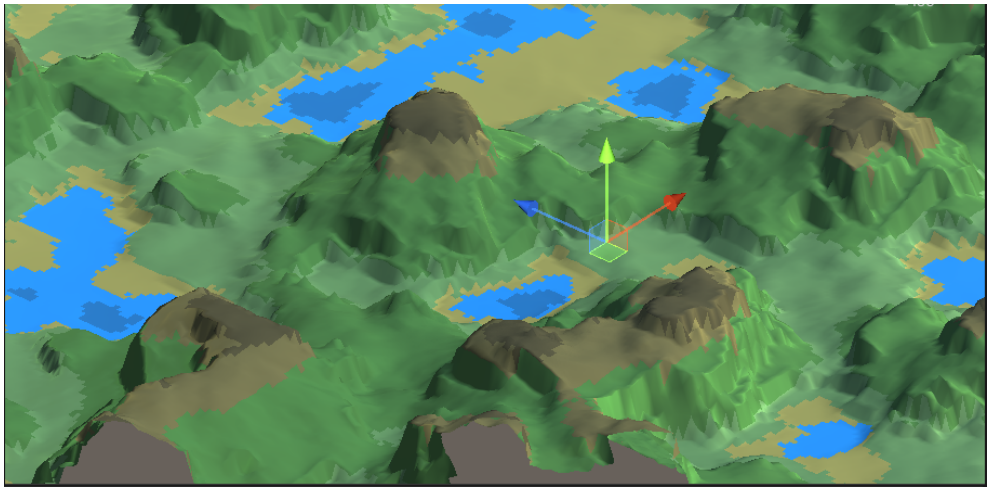

Рисунок 2.15 – Приклад генерації з модифікатором ерозії

2.3 Конфігурування

Конфігурація генерованого ландшафту відбувається змінюючи масштаб шуму, кількість та вагу шарів, що накладаються, а також зернистість. Налаштування типів поверхні відбувається через заданий діапазонів висот, де ці типи допустимі. Для тривимірного відображення використовується параметр multiplier, який збільшує висоту окремих ділянок. Далі наведені приклади генерацій для різних типів біомів.

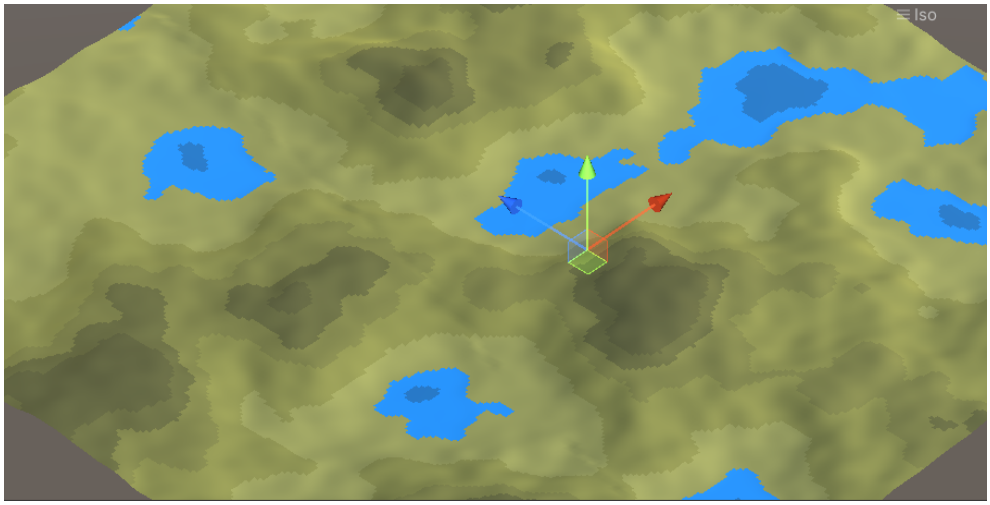

Рисунок 2.16 – Пустеля. scale=126, octaves=4, persistence=0.41, lacunarity=2.14, multiplier=100

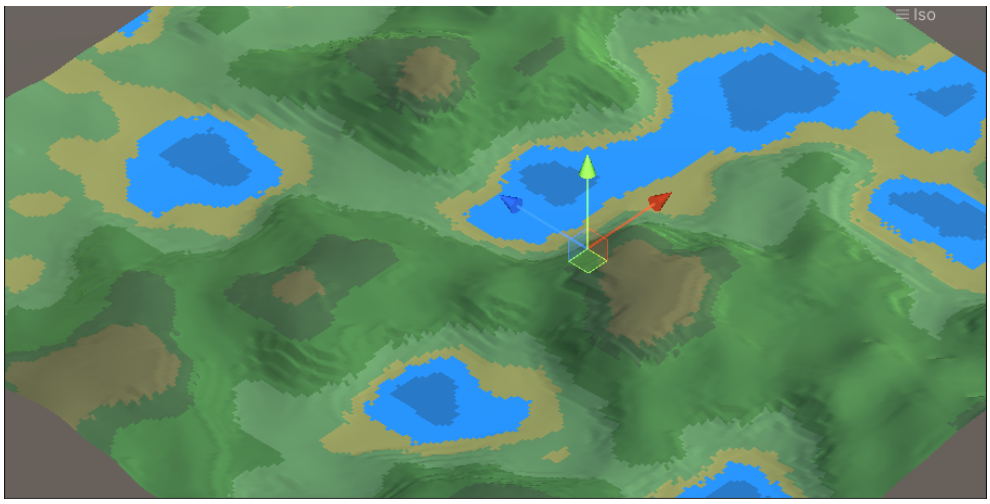

Рисунок 2.17 – Долина. scale=126, octaves=3, persistence=0.41, lacunarity=2.14, multiplier=130, erosionSteps=2, erosionLimit=0.041

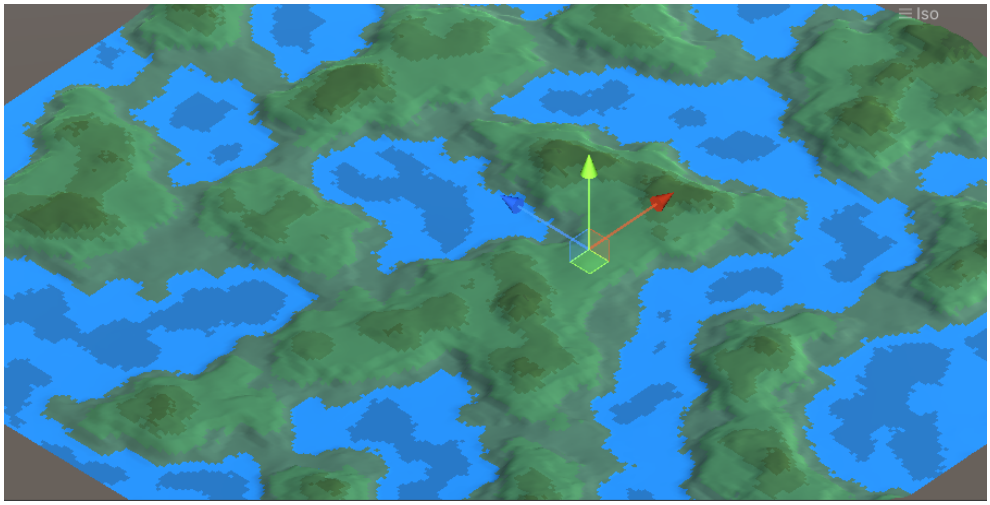

Рисунок 2.18 – Болото. scale=62, octaves=3, persistence=0.29, lacunarity=3.0, multiplier=150, erosionSteps=2, erosionLimit=0.06

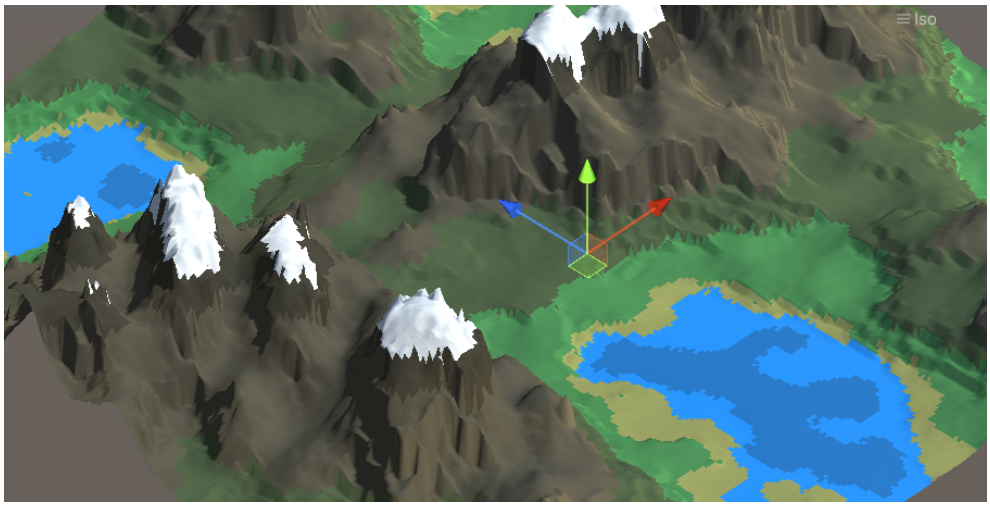

Рисунок 2.19 – Гори. scale=150, octaves=4, persistence=0.22, lacunarity=5.1, multiplier=700, erosionSteps=4, erosionLimit=0.04

ВИСНОВКИ

Наразі існує достатньо багато способів генерації ландшафтів. Однак обмеження, що накладаються ігровим світом, двигуном, вимогами до швидкодії та низки налаштувань значно звужують вибірку. Наразі найбільш часто використовувані методи побудовані на базі функцій шуму та алгоритмів поділу на частини, таких як тріангуляція та діаграми Вороного.

В роботі була поставлена наступна мета: дослідити методи генерації нескінченного ігрового ландшафту з використанням різних підходів, на прикладі симплекс шуму та тріангуляції Делоне.

В процесі досягнення мети були виконані наступні задачі:

1. Досліджені сфери використання методів на базі функцій шуму.
2. Досліджені сфери використання методів на базі алгоритмів поділу.
3. Розглянуті способи комбінації цих алгоритмів.
4. Розроблений прототип для візуалізації отриманих результатів.
5. Додані конфігурації для різних типів генерованих ландшафтів.

Таким чином можна вважати, що результати розробки повністю відповідають усім вимогам технічного завдання, поставлена мета досягнута. Робота має закінчений характер.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Jean-David Genevaux, Eric Galin, Eric Guérin, Adrien Peytavie, Bedrich Benes. [Електронний ресурс] Terrain Generation Using Procedural Models Based on Hydrology – 2013. https://hal.archives-ouvertes.fr/hal-01339224/document
2. Jacob Olsen. [Електронний ресурс] Realtime Synthesis of Eroded Fractal Terrain for Use in Computer Games – 2004. https://web.mit.edu/cesium/Public/terrain.pdf
3. Денис Ольшин. [Електронний ресурс] Алгоритм «diamond-square» для построения фрактальных ландшафтов – 2011. https://habr.com/ru/post/111538/
4. Amit Patel. [Електронний ресурс] Polygonal Map Generation for Games – 2010. http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/#source
5. Юрій Кулагін. [Електронний ресурс] Как процедурная генерация помогает создавать открытые миры – 2020. https://www.school-xyz.com/kak-procedurnaya-generaciya-pomogaet-sozdavat-otkrytye-miry
6. Amit Patel. [Електронний ресурс] Making maps with noise functions – 2015. https://www.redblobgames.com/maps/terrain-from-noise/
7. A tour of the C# language. [Електронний ресурс] : (частина офіційної документації) – 2021. https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
8. Jon Skeet C# in Depth / Jon Skeet : Manning, 2019.
9. The history of C#. [Електронний ресурс] : (частина офіційної документації) – 2020.

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history

10. Joe Hocking Unity in Action: Multiplatform Game Development in C# / Joe Hocking – Manning, 2015.
11. John Haas [Електронний ресурс] A History of the Unity Game Engine – 2012. https://digital.wpi.edu/downloads/2f75r821k