

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерні науки»)

на тему:

**«Програмна реалізація генерації тривимірного
ландшафту ігрового світу в Unity»**

Виконав студент групи КНм-23-1
ХАРЧЕНКО Андрій Володимирович

Керівник роботи
ЛЕВКІВСЬКИЙ Віталій Леонідович

Рецензент
ГРАФ Марина Сергіївна

Житомир – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ЗАТВЕРДЖУЮ

Завідувач кафедри комп'ютерних наук

Марина ГРАФ

«17» жовтня 2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу освітнього ступеня «магістр»

за спеціальністю 122 «Комп'ютерні науки»

(освітня програма «Комп'ютерні науки»)

Здобувач вищої освіти: **ХАРЧЕНКО Андрій Володимирович**

Керівник роботи: **ЛЕВКІВСЬКИЙ Віталій Леонідович**

Тема роботи: **«Програмна реалізація генерації тривимірного ландшафту ігрового світу в Unity»**,

затверджена наказом закладу вищої освіти від **«17» жовтня 2024 р., №505/с**

Термін здачі закінченої роботи: **«20» грудня 2024 р.**

Вихідні дані роботи (зазначається предмет і об'єкт дослідження): об'єктом дослідження є процес процедурної генерації тривимірних ландшафтів ігрового світу. Предметом дослідження є методи і підходи до програмної реалізації процедурної генерації тривимірних ландшафтів ігрового світу.

Консультанти кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Опрацювання літературних джерел.	1 лютого 2024 – 1 квітня 2024	Виконано
2	Дослідження методів та алгоритмів процедурної генерації ландшафтів.	2 квітня 2024 – 20 липня 2024	Виконано
3	Проектування додатку. Написання програмного коду	21 липня 2024 – 25 жовтня 2024	Виконано
4	Тестування системи	26 жовтня 2024 – 14 листопада 2024	Виконано
5	Оформлення пояснювальної записки	15 листопада 2024 – 17 грудня 2024	Виконано
6	Попередній захист	18 грудня 2024	Виконано

Здобувач вищої освіти **Андрій ХАРЧЕНКО**

Керівник **Віталій ЛЕВКІВСЬКИЙ**

РЕФЕРАТ

Кваліфікаційна робота присвячена програмній реалізації генерації тривимірного ландшафту ігрового світу в Unity. Пояснювальна записка складається з вступу, трьох розділів, висновків, списку використаної літератури та додатків. Основна частина викладена на 96 сторінках друкованого тексту. Додатки розміщено на 16 сторінках. Список використаних джерел містить 41 найменувань і займає 4 сторінки. В роботі наведено 61 рисунок та одна таблиця. Загальний обсяг роботи – 112 сторінок.

У роботі було проведено огляд відомих методів та алгоритмів, що використовуються для процедурної генерації тривимірних ландшафтів. Проаналізовано можливі інструменти для реалізації проекту та обрано найбільш підходящі.

Також було спроектовано архітектуру структурних елементів за допомогою діаграм та описано роботу основних методів класів програмного додатку. Здійснено тестування програмного додатку, яке підтвердило його працездатність. Результати роботи алгоритмів було візуалізовано за допомогою Google Colaboratory.

Ключові слова: тривимірний ландшафт, процедурна генерація, мапа висот, шум Перліна, diamond-square, Voronoi, DLA, рівні деталізації, ландшафтні шейдери.

ABSTRACT

The qualification thesis is dedicated to the software implementation of three-dimensional game world landscape generation in Unity. The explanatory note consists of an introduction, three chapters, conclusions, a list of references, and appendices. The main part spans 96 pages of printed text. The appendices cover 16 pages. The list of references includes 41 sources and takes up 4 pages. The thesis contains 61 figures and one table. The total volume of the thesis is 112 pages.

The thesis provides an overview of well-known methods and algorithms used for procedural generation of 3D landscapes. Possible tools for project implementation were analyzed, and the most suitable ones were selected.

Additionally, the architecture of structural elements was designed using diagrams, and the operation of the main methods of the software application classes was described. The software application was tested, confirming its functionality.

The results of the algorithms were visualized using Google Colaboratory.

Keywords: three-dimensional landscape, procedural generation, heightmap, Perlin noise, diamond-square, Voronoi, DLA, levels of detail, landscape shaders.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ ГЕНЕРАЦІЇ ЛАНДШАФТІВ.....	9
1.1. Огляд методів генерації тривимірних ландшафтів.....	9
1.2. Фрактальні підходи в моделюванні та шум Перліна.....	19
1.3. Постановка задачі.....	24
1.4. Вибір інструментів для реалізації генерації тривимірного ландшафту ...	25
Висновки до першого розділу.....	27
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ГЕНЕРАЦІЇ ТРИВИМІРНОГО ЛАНДШАФТУ ІГРОВОГО СВІТУ	28
2.1. Реалізація алгоритмів створення мапи висот	28
2.2. Генерація тривимірної сітки ландшафту	51
2.3. Створення шейдерів для ландшафтів.....	56
2.3. Динамічна генерація ландшафту ігрового світу	62
Висновки до другого розділу	71
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ, ЙОГО ТЕСТУВАННЯ ТА АНАЛІЗ	72
3.1. Опис роботи з програмним додатком	72
3.2. Тестування роботи програмного забезпечення.....	79
3.3. Створення набору даних для аналізу	82
3.4. Візуалізація та аналіз роботи алгоритмів генерації.....	85
Висновки до третього розділу.....	91
ВИСНОВКИ.....	92
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	93
ДОДАТКИ.....	97

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

DLA – Diffusion-Limited Aggregation, агрегація з дифузійним обмеженням.

LOD – Level Of Detail, рівень деталізації.

glTF – GL Transmission Format, це відкритий формат файлу для передачі і зберігання тривимірних моделей і сцен.

UI – User Interface, користувацький інтерфейс.

ВСТУП

Актуальність теми. Сучасна індустрія відеоігор знаходиться на передовій цифрових технологій, створюючи все більш реалістичні та захоплюючі віртуальні світи, що пропонують гравцям унікальні можливості для занурення у ігровий процес. Сьогодні розробники стикаються з новими проблемами масштабу через експоненційне зростання за останнє десятиліття як кількості гравців, так і витрат на виробництво. Процедурне створення контенту для ігор має можливість вирішити ці проблеми шляхом автоматизації або певної допомоги в створенні ігрового вмісту [1].

Ландшафти є важливим компонентом тривимірних сцен і присутні в багатьох програмах комп'ютерної графіки. Методи моделювання рельєфу зосереджені на фіксації його форм з усіма складними деталями, включаючи ерозійні долини, що виникають у результаті взаємодії різноманітних природних явищ, гірські хребти та складні річкові системи [2].

Створення ігрових світів з високим рівнем графічної деталізації, що мають мільйони квадратних кілометрів, вимагає великих ресурсів, зокрема часу, людських ресурсів та фінансування. І якщо для великих ігрових студій це питання вирішується розподілом обов'язків між десятками або навіть сотнями працівників, то для невеликих інді-команд це може стати серйозною перешкодою на шляху до розробки конкурентоспроможних проєктів [3].

Серед найвідоміших прикладів використання процедурної генерації можна згадати такі ігри, як *No Man's Sky*, яка славиться своїми мільярдами процедурно згенерованих планет, або ж *Minecraft*, де кожен новий світ створюється алгоритмічно, з унікальними ландшафтами, системами печер, біомів та іншими елементами [4-7].

Процедурна генерація дозволяє досягти високого рівня деталізації та різноманітності контенту при значно меншій витраті ресурсів. Окрім того, вона дозволяє створювати нелінійні ігрові рівні, де кожен новий запуск гри може пропонувати нові виклики та умови для гравця, тим самим підвищуючи інтерес до гри. З огляду на це, алгоритми процедурної генерації стали невід'ємною

частиною багатьох сучасних ігрових рушіїв, таких як Unity та Unreal Engine [8, 9].

Процедурна генерація використовується у фільмах для створення реалістичних середовищ, таких як ліс, хмари чи поверхня води, анімацій тощо [10]. Ці алгоритми допомагають архітекторам створювати варіанти дизайну будівель, інтер'єрів або міських планів [11]. У моделюванні природних процесів, таких як ерозія, формування рельєфу чи поведінка рідин, процедурні методи використовуються для створення симуляцій [12]. Також автоматично згенеровані сценарії, задачі чи моделі є корисними для навчальних програм і симуляцій [13, 14].

Об'єкт дослідження: процес процедурної генерації тривимірних ландшафтів ігрового світу.

Предмет дослідження: методи і підходи до програмної реалізації процедурної генерації тривимірних ландшафтів ігрового світу.

Методи дослідження: абстрагування, аналіз, моделювання, симуляція, аналіз літературних джерел.

Наукова новизна роботи полягає в дослідженні та реалізації методів процедурної генерації тривимірних ландшафтів. Створений програмний додаток може бути використаний в ігровій індустрії, навчальних цілях та інших областях.

Практична цінність роботи полягає в розробці методів і алгоритмів, які реалізовані у програмному продукті та можуть використовуватись в проектах, де важливо мати можливість процедурно створювати тривимірні ландшафти.

В рамках кваліфікаційної роботи було опубліковано наукові праці:

1. Марчук Г. В., Левківський В. Л., Харченко А. В., Марчук Д. К. Огляд і аналіз алгоритмів процедурної генерації ігрових світів. Таврійський науковий вісник. Серія: Технічні науки. 2024. Вип. 4. С. 101-110.

2. Харченко А. В., Левківський В. Л., Колос К. Р. Процедурна генерація ігрових світів. Тези VI Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення», 29-30 листопада 2023 р. Житомир: «Житомирська політехніка», 2023. С. 251-252.

РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ ГЕНЕРАЦІЇ ЛАНДШАФТІВ

1.1. Огляд методів генерації тривимірних ландшафтів

Генерація тривимірних ландшафтів є важливою складовою сучасного комп'ютерного моделювання та ігрової розробки. Завдяки розвитку графічних технологій, інженери та дизайнери отримали можливість створювати реалістичні та складні ландшафти, які можуть бути інтерактивними і варіюватися залежно від умов навколишнього середовища. Одним з ключових етапів цього процесу є вибір та реалізація алгоритмів генерації ландшафтів. Різні підходи до цієї задачі дозволяють досягти різних результатів: від генерації великих гірських масивів і долин до детальних текстур поверхні, таких як еродовані скелі або м'які пагорби.

Тривимірні ландшафти мають вирішальне значення в багатьох галузях, включаючи ігрову індустрію, симуляції природних процесів, віртуальну реальність, а також архітектурне моделювання. Створення таких ландшафтів може базуватися як на фізичних принципах, що відображають реальні природні процеси (наприклад, ерозія або формування ландшафтів вітром), так і на математичних моделях, які дозволяють контролювати кожен етап генерації. Такий підхід дозволяє не лише точно відтворювати природні явища, але й експериментувати з різними штучними, абстрактними формами, що можуть бути створені з використанням математичних шумів або випадкових процесів [15].

Шуми є основними методами генерації тривимірних ландшафтів. Вони також є ключовими інструментами в процедурній генерації контенту, особливо у створенні віртуальних світів, текстур, ландшафтів та різних природних явищ. Процедурні алгоритми, такі як шум Перліна, Diamond-Square, шум Вороного та Simplex Noise, дозволяють створювати складні й реалістичні патерни, використовуючи прості математичні правила. Ці алгоритми є невід'ємною частиною ігрової індустрії, комп'ютерної графіки та симуляцій природних явищ, допомагаючи уникнути необхідності ручної генерації великих масивів контенту.

Одним із найвідоміших та найвпливовіших алгоритмів є Шум Перліна (Perlin Noise), що використовується в процедурній генерації контенту, зокрема для створення реалістичних текстур і ландшафтів у комп'ютерній графіці. Його

створив **Кен Перлін** у **1983 році** під час роботи над фільмом *Трон* (1982). Перлін розробив цей алгоритм для того, щоб допомогти візуальним ефектам виглядати більш природно, позбавленими чітко видимих артефактів, які часто виникають під час використання простих генеративних методів [16].

До створення шуму Перліна, графічні текстури часто мали непривабливий штучний вигляд через те, що використовувані алгоритми генерували надто регулярні або періодичні візерунки. Це призводило до повторюваних структур, які не мали реалістичного вигляду. Кен Перлін вирішив цю проблему, створивши алгоритм, що генерує гладкий, неперіодичний шум, який імітує природні структури, такі як поверхні землі, хмари, гори, вода тощо.

Шум Перліна мав значний вплив на комп'ютерну графіку та процедурну генерацію в цілому, особливо в ігровій індустрії. Він дав можливість генераторам контенту автоматично створювати реалістичні текстури та ландшафти, які виглядали б природно і неповторно. Зокрема, його використовують для створення гірських хребтів, текстур поверхонь, шумових полів у 3D-моделюванні, симуляцій хмар, води, вогню тощо.

Хоча початковий шум Перліна був революційним, він мав деякі недоліки, зокрема у високовимірних просторах, де він виявляв артефакти через свою сіткову структуру. У **2001 році** Кен Перлін розробив нову версію — **Simplex Noise**, яка усуває ці проблеми. Simplex Noise є більш ефективним у вищих вимірах і працює швидше, оскільки використовує менш регулярну сітку і оптимізовані обчислення. Цей алгоритм особливо добре підходить для 3D-симуляцій і реального часу, таких як ігри або віртуальні світи [17].

Серед переваг шуму Перліна можна виділити його гладкі переходи: значення шуму плавно змінюються, створюючи природні текстури без різких переходів. Також його доволі легко масштабувати, що дозволяє використовувати його на різних рівнях деталізації.

Окрім Шуму Перліна класичним алгоритмом генерації фрактальних ландшафтів також є алгоритм **Diamond-Square**. Ідею створювати масштабовані поверхні з використанням відносно простих математичних операцій вперше

представили Фурньє, Фюсселл і Карпентер у SIGGRAPH (Special Interest Group on Graphics and Interactive Techniques) у 1982 році. Алгоритм базується на принципах фрактальної геометрії та був розроблений для створення реалістичних ландшафтів [18].

Diamond-Square – це ітераційний метод, що використовує розбиття поверхні на квадрати, і дозволяє створювати поверхні з фрактальною нерегулярністю. Основний принцип алгоритму полягає в тому, що на кожному кроці значення висот для нових точок обчислюються на основі середнього значення висот сусідніх точок, із додаванням випадкового шуму. Цей підхід дозволяє створити "гірські" рельєфи, схожі на природні.

Алгоритм складається з двох основних кроків, що повторюються (рис. 1.1):

1. **Крок "Діаманта" (Diamond Step):** У цьому кроці на кожному квадраті генерується центральна точка, значення якої є середнім арифметичним висот чотирьох вершин цього квадрату. До середнього додається випадкова величина для створення нерегулярності.

2. **Крок "Квadrата" (Square Step):** Після цього обчислюються нові точки на середині сторін квадратів. Значення кожної нової точки є середнім арифметичним трьох (якщо точка знаходиться на краю матриці) або чотирьох сусідніх вершин, із додаванням випадкової величини.

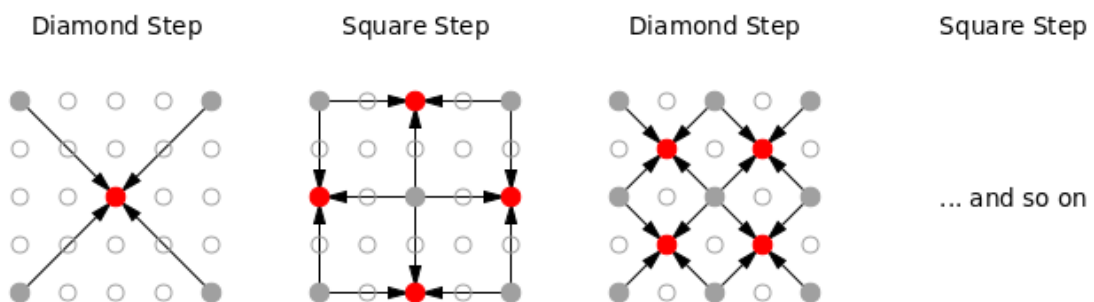


Рис. 1.1. Алгоритм роботи Diamond-Square [19]

На кожному наступному ітераційному кроці "розбивання" сітки повторюється, і площа отримує все більше точок, що надає їй складний та деталізований вигляд, подібний до натуральних гірських хребтів або нерівностей ландшафту.

Diamond-Square став дуже популярним у 1980-х та 1990-х роках, оскільки надавав можливість швидко генерувати правдоподібні ландшафти для ранніх відеоігор. На відміну від інших методів, цей алгоритм був ефективним і відносно простим у реалізації, що дозволяло розробникам використовувати його для створення великих карт або світів з мінімальними затратами ресурсів [20].

За час свого існування алгоритм розвивався та ставав фундаментом для інших досліджень та розробок. Наприклад, Diamond-Square модифікували для роботи в тривимірному просторі за принципом щільності, замість генерації двовимірної мапи висот [21].

Ще одним корисним алгоритмом генерації процедурних текстур є **шум Вороного** (іноді його також називають діаграмою Вороного). Він є одним із основних алгоритмів у процедурній генерації контенту. Шум Вороного був названий на честь українського математика **Георгія Вороного**, який вперше розробив концепцію діаграм Вороного у **1908 році** для вивчення просторових поділів у геометрії. Діаграми Вороного та пов'язані з ними шумові алгоритми широко використовуються в комп'ютерній графіці, геометричному моделюванні, симуляціях, ігрових двигунах та науці загалом для створення органічних, нерегулярних візерунків і текстур. Використання діаграм Вороного у процедурній генерації текстур було описано Стівеном Уорлі [22].

Діаграми Вороного — це поділ простору на області на основі набору точок, які називаються сайтами або генераторами. Для кожної точки в просторі знаходиться найближча точка-генератор, і ця точка належить відповідній комірці Вороного. У двовимірному просторі ці комірки мають форму багатокутників, а в тривимірному просторі — багатогранників.

Ця концепція була вперше застосована в математиці, але згодом знайшла широке застосування в комп'ютерній графіці, зокрема для створення шумів, схожих на природні структури. Шум Вороного використовується для створення природних текстур, які нагадують комірки, кристалічні структури, розподіли водойм або навіть розташування клітин у тканинах організмів.

Алгоритм процедурно генерує текстуру, поділяючи простір на комірки, які відповідають найближчій до кожної точки *сайту* (точці-генератору). Основний алгоритм для двовимірного простору починається з генерації точок-генераторів. Потім кожна точка, або піксель, в просторі прив'язується до найближчого генератора за допомогою метрики відстані, де найчастіше використовується евклідова відстань. Ця відстань до найближчого генератора визначає, до якої комірки Вороного належить точка. Потім алгоритм використовує відстані до найближчого або до кількох найближчих генераторів, щоб визначити колір або інші атрибути кожної точки на текстурі. Це дозволяє створювати візерунки, що нагадують осередкові структури, мозаїки або органічні форми.

Важливо, що метрика відстані може змінюватися – наприклад, інші варіанти, така як мангеттенська відстань, можуть дати інші геометричні ефекти [23].

Шум Вороного є дуже популярним у процедурній генерації завдяки своїй здатності створювати візуально привабливі, нерегулярні візерунки. Він може використовуватися для створення нерегулярних форм рельєфу. Території або області, розділені за принципом Вороного, можуть використовуватися для визначення розташування гір, долин або водойм. Також він часто використовується для генерації органічних текстур, таких як кора дерев, шкіра, граніт або мрамур, та клітинних структур, таких візерунки на рослинах або тваринах. Діаграми Вороного також можна застосовувати в урбаністичному плануванні та моделюванні для створення поділів або територій, що імітують природне розташування будівель або кварталів у містах.

Шум Вороного відіграв важливу роль у розвитку процедурної генерації, оскільки він дозволяє створювати природні й органічні візерунки. Його використання у відеоіграх та комп'ютерній графіці допомогло розробникам автоматизувати створення великих і складних ландшафтів, текстур і об'єктів, без необхідності ручної роботи. У машинному навчанні діаграми Вороного використовуються для класифікації 1-NN [24].

За останні роки алгоритм Шуму Вороного значно еволюціонував. Окрім евклідової відстані, було введено інші метрики, які дають різноманітні геометричні форми й ефекти, залежно від того, яка відстань використовується. Для створення різноманітніших текстур були розроблені варіації шуму, які дозволяють додавати випадковість і нестандартні форми до текстур. Також з розвитком графічних процесорів (GPU) алгоритм був оптимізований для швидкої генерації великих осередкових структур у реальному часі.

Створення реалістичних віртуальних тривимірних рельєфів є доволі важливою частиною як індустрії кіно, так і комп'ютерних ігор та симуляторів. Реалістичний та візуально привабливий ландшафт допомагає споживачу контенту зануритися у віртуальний світ. Один з найбільших та тяжких викликів є саме симуляція ефекту ерозії. **Ерозія** — це природний процес руйнування і переміщення ґрунту, каменю та інших матеріалів під дією вітру, води або льоду. Вода є головним фактором ерозії, особливо через дощі та річкові течії, які змивають ґрунт зі схилів, формуючи долини та каньйони. Вітер також сприяє ерозії в посушливих районах, де з поверхні видувуються дрібні частинки ґрунту, створюючи пустелі та дюни. Наприклад, Великий Каньйон у США сформувався завдяки мільйонам років ерозії річкою Колорадо [25].

Дуже сильний та помітний вплив на використання алгоритму ерозії в розробці ігор та процедурного створення ландшафту зробив Ганс Теобальд Бейєр в своїй дисертації 2015 року. В багатьох проектах, пов'язаних з симуляцією реалістичної ерозії ландшафту, згадується саме його дисертація за темою «Імплементація методу гідравлічної ерозії». Ключовий підхід в моделі гідравлічної ерозії Ганса Теобальда Бейєра полягає в симуляції взаємодії дощових крапель із ландшафтом. Вода, що представляє дощ, падає на поверхню і починає текти вниз, слідуючи шляхам найменшого опору. У процесі свого руху вода змиває осад з вищих ділянок і відкладає його на нижчих, поступово змінюючи топографію. Ця модель ерозії потребує точного опрацювання рівнянь динаміки рідини та транспорту осаду для створення реалістичних патернів ерозії ландшафту.

Одна з інновацій підходу Бейєра полягає в тому, як симуляція врівноважує точність і обчислювальну ефективність. У роботі розглядаються компроміси між використанням методу Ейлера, заснованого на сітці, і Лагранжевим методом, що базується на частинках.

Терміни Ейлеровий та Лагранжевий використовуються у динаміці рідин і фізиці для опису того, як речовини рухаються і змінюються з часом.

В Ейлеровому методі увага зосереджена на конкретних місцях у просторі, через які проходить рідина. Простір поділяється на фіксовану сітку, і властивості рідини, наприклад, швидкість, тиск або густина, розраховуються в кожній точці сітки в міру руху рідини через неї. Тобто, спостерігається, як рідина поводить себе в певній точці простору протягом часу. Цей підхід часто застосовується в симуляціях, які вимагають детального контролю за великими об'ємами рідини, наприклад, у моделюванні погоди або динаміки рідин для інженерії. Він забезпечує детальний огляд того, як вода тече по поверхні, розраховуючи динаміку рідини в кожній точці сітки.

У Лагранжевому методі увага зосереджена на відстеженні окремих частинок у міру їх руху через простір і час. Замість того, щоб спостерігати за потоком у фіксованих точках простору, ви відстежуєте траєкторію конкретних частинок у системі. Цей метод часто є більш ефективним у симуляціях, що базуються на частинках, таких як моделювання транспорту осаду, поведінки крапель дощу або ерозії. Він широко використовується в симуляціях у реальному часі для ігор, оскільки вимагає менше обчислювальних ресурсів порівняно з Ейлеровим підходом. **Лагранжевий** відстежує окремі краплі води або частинки, що робить його обчислювально дешевшим і легшим для використання в симуляціях у реальному часі, що особливо важливо для ігор.

Обидва підходи походять від робіт математиків 18-го століття **Леонарда Ейлера** та **Жозефа-Луї Лагранжа**, які розробили ці підходи для вивчення руху рідини та динаміки в різних фізичних системах.

Метод Ганса Теобальда Бейєра був прийнятий і адаптований у кількох відкритих проєктах з відкритим вихідним кодом, таких як **Erodr** [26], де його

алгоритм використовується для симуляції впливу водної ерозії на ландшафти. Простота та гнучкість цього алгоритму роблять його підходящим для застосувань у реальному часі в ігрових рушіях, де ерозію можна динамічно застосовувати для модифікації місцевості у відповідь на дії гравців або зовнішні фактори.

Проте, хоча робота Бейєра мала значний вплив, інші відомі внески в галузі гідравлічної ерозії включають реалізацію **Себастьяна Лагу** в Unity [27], яка є дуже відомою в спільноті розробників. Робота Лагу будується на концепціях Бейєра, але також включає модифікації для оптимізації продуктивності та розширення функціональності. Крім того, сучасні підходи використовують прискорення за допомогою GPU для значного зменшення часу обчислень у масштабних симуляціях, з чим початковий підхід Бейєра, що базувався на використанні CPU, мав труднощі [28, 29, 30].

Diffusion-Limited Aggregation (DLA), або агрегація з дифузійним обмеженням – це модель, яка описує, як частинки агрегуються під час дифузії. Це явище формує складні фрактальні структури. Алгоритм DLA широко використовується для моделювання природних процесів, таких як ріст кристалів, електроосадження, утворення блискавок і навіть процеси біологічного росту.

Термін "агрегуватися" означає об'єднуватися або накопичуватися в групу чи кластер. У контексті моделі DLA (Diffusion-Limited Aggregation), це означає, що частинки, які рухаються за допомогою дифузії, злипаються при зустрічі з іншими частинками або з ядром. Таким чином, вони створюють більшу групу або структуру, яка з часом розростається. У даному контексті слово "дифузійним" походить від терміну "дифузія", який описує процес переміщення частинок від області з високою концентрацією до області з низькою концентрацією. Тобто, частинки вільно і випадково рухаються до тих пір, поки не зустрінуть інші частинки або перепони, з якими вони можуть взаємодіяти.

Агрегація з дифузійним обмеженням (DLA) була вперше представлена в 1980-х роках Віттеном і Сандером і з того часу застосовується для вивчення різних природних явищ, таких як утворення сніжинок, колоній бактерій і

руйнування діелектриків в електричних системах. Важливість цієї моделі полягає в тому, що вона дає можливість моделювати розгалужені, нерегулярні візерунки, що виникають у цих процесах, надаючи основу для розуміння того, як такі структури розвиваються в умовах, де дифузія обмежена.

Фрактальна природа візерунків, що утворюються за допомогою DLA, відображає самоподібність, тобто схожі структури можна побачити на різних масштабах. Це робить DLA ключовою моделлю для розуміння того, як складні, нерегулярні форми виникають у природі через прості, випадкові процеси.

Сам алгоритм DLA складається з кількох простих, але важливих, етапів:

1. **Ініціалізація:** Створюється початкова "насіннева" частинка, яка розміщується в центрі полотна. Ця насіннева частинка виступає ядром, навколо якого відбувається процес агрегування.
2. **Додавання частинок:** Одна за одною в систему вводяться випадкові частинки, кожна з яких розміщується в випадковій точці, зазвичай далеко від насінневої частинки або вже існуючого кластеру.
3. **Дифузія:** Випущені частинки здійснюють випадковий рух в будь-якому напрямку на кожному кроці.
4. **Агрегація:** Коли частинка, що рухається, контактує з будь-якою іншою частинкою, яка вже є частиною кластеру, вона прилипає і стає частиною цього кластеру.

Цей процес повторюється, нові частинки продовжують вводитися в систему, дифузіївати випадковим чином і прилипати до зростаючого кластеру при контакті з ним.

У міру того, як більше частинок прилипає до кластеру, він починає рости. З часом з'являється розгалужена, деревоподібна структура, причому нові частинки переважно прилипають до найбільш віддалених частин кластеру. Отримана структура має фрактальний візерунок, що характеризується нерегулярним, розгалуженим ростом і самоподібністю. Великі кластери відображають ту саму загальну форму і складність, що й менші кластери, хоча й на більшому масштабі.

DLA часто використовується для симуляції різноманітних природних і фізичних явищ. Вона використовується для моделювання того, як кристали формуються і ростуть у природних умовах. Нерегулярну, розгалужену структуру сніжинок, наприклад, можна змодельовати за допомогою даного процесу, коли окремі молекули води рухаються через повітря і прилипають до зростаючої кристалічної структури. Також DLA використовується для моделювання осадження іонів металу на електроді. Іони металу в розчині рухаються до електрода і прилипають до нього, утворюючи фрактальний візерунок металевих осадів. Ріст бактерій часто слідує дифузійно-обмеженому патерну, коли нові бактерії рухаються від центральної колонії і прилипають до країв вже існуючої колонії. Це призводить до утворення нерегулярних, розгалужених колоній бактерій, схожих на візерунки, які спостерігаються в симуляціях DLA.



Рис. 1.2. Кластер DLA, вирощений із розчину мідного купоросу в комірці електроосадження

Хоча модель DLA ефективно генерує складні, розгалужені структури, вона має деякі виклики. Наприклад, у міру зростання кластеру нові частинки з більшою ймовірністю будуть дифузійувати далеко від кластеру перед тим, як контактувати з ним. Це призводить до повільнішого процесу агрегування, оскільки кластер стає більшим. Також, стохастична природа моделі означає, що точна структура кластеру сильно залежить від шляхів випадкових прогулянок

окремих частинок. Хоча ця варіативність є частиною того, що робить DLA цікавим, вона також може ускладнити передбачення кінцевої форми кластеру.

У міру збільшення кількості частинок обчислювальна вартість симуляції DLA зростає. Кожна частинка повинна пройти потенційно довгу випадкову прогулянку перед тим, як прилипнути до кластеру, і кількість частинок, необхідних для формування великого кластеру, може бути значною [31].

Для створення тривимірного ландшафту згенеровану структуру можна розглядати як гірський хребет, в якому найдаальші від центра частинки приймають найменші значення висоти, а центральні – найбільші. Звісно в процесі дана мапа висот має пройти безліч етапів розмиття, проте в результаті виходить привабливий ландшафт з достовірним ефектом реалістичності.

1.2. Фрактальні підходи в моделюванні та шум Перліна.

Фрактальні підходи в моделюванні є однією з ключових методик у комп'ютерній графіці та процедурній генерації контенту. Вони ґрунтуються на математичних концепціях фракталів, які дозволяють генерувати складні, природні структури, такі як гірські хребти, лісові пейзажі, річки, хмари та інші географічні об'єкти. Однією з головних переваг фрактальних підходів є їх здатність створювати детальні й правдоподібні форми з мінімальними початковими даними.

Фрактали — це математичні структури, що мають властивість самоподібності, тобто їх форма виглядає однаково на різних масштабах. Природні об'єкти, такі як гори, берегові лінії, хмари або дерева, часто мають фрактальні властивості. У моделюванні цей принцип використовується для створення детальних поверхонь і текстур, коли деталі додаються на різних рівнях масштабу, створюючи візуально складні структури.

У моделюванні природних форм обмеження, накладені евклідовою геометрією, можуть бути перевищені фракталами. Фрактальна геометрія відносно молода (перші дослідження — роботи французьких математиків П'єра Фату (1878-1929) і Гастона Джуліа (1893-1978) на початку 20 століття), але лише

завдяки математичній потужності комп'ютерів стало можливим реалізувати зв'язки між фрактальною геометрією та іншими дисциплінами. Зараз фрактали використовуються в різних сферах, від біології до архітектури. Важливе застосування також з'являється в інформатиці, оскільки фрактальна геометрія дозволяє стискати зображення та відтворювати в середовищах віртуальної реальності складні візерунки, присутні в природі, використовуючи прості ітераційні алгоритми, що виконуються комп'ютерами. Останні дослідження застосовують цю геометрію для керування трафіком у комп'ютерних мережах (LAN, MAN, WAN та Інтернет) та для реалізації віртуальних світів [32].

У багатьох іграх або симуляціях фрактальні алгоритми використовуються для створення гірських хребтів та ландшафтів. Наприклад, вже раніше згаданий алгоритм Diamond-Square (або його варіації) генерує ландшафти за допомогою рекурсивного розбиття площини на менші квадрати, кожен з яких отримує випадкове значення висоти, що робить поверхню схожою на природний рельєф.

Шум Перліна — це градієнтний шум, який генерує плавні випадкові зміни між значеннями в просторі. Ідея полягає в тому, щоб інтерполювати між випадковими градієнтами, що розміщені у вузлах регулярної сітки.

Перш за все відбувається генерація градієнтів у точках сітки. Для цього уявимо простір, розділений на сітку квадратів. У кожній вершині квадрата потрібно згенерувати випадковий градієнтний вектор. У 2D це вектор на одиничному колі. У 3D аналогічно. Потім для кожної точки в просторі потрібно знайти її положення в сітці. Для цього визначаємо, між якими чотирма вузлами вона знаходиться. Для 2D це буде квадрат, для 3D – куб.

Для кожної точки і кожного сусіднього вузла потрібно обчислити вектор від цієї точки до вузла і знайти скалярний добуток цього вектора з градієнтним вектором, що зберігається в цьому вузлі. Це дасть величину "впливу" градієнта на цю точку. Для плавних переходів між значеннями, потрібно інтерполювати ці результати скалярного добутку. Для цього використовується функція згладжування ($\text{fade}(t) = 6t^5 - 15t^4 + 10t^3$), щоб отримати плавний перехід між значеннями в різних частинах сітки.

Після того як буде обчислено вплив від кожного з чотирьох (для 2D) або восьми (для 3D) сусідніх вузлів, використовується лінійна інтерполяція для плавного комбінування цих результатів, отримуючи остаточне значення для точки [33-35].

На просторах інтернету можна знайти імплементацію коду Покращеного Шуму Перліна, написану Кеном Перліна у 2002 році. Код написаний на мові програмування Java.

Лістинг програмного коду Імплементації Покращеного Шуму [36]:

```
// JAVA REFERENCE IMPLEMENTATION OF IMPROVED NOISE - COPYRIGHT 2002 KEN PERLIN.

public final class ImprovedNoise {
    static public double noise(double x, double y, double z) {
        int X = (int)Math.floor(x) & 255,          // FIND UNIT CUBE THAT
        Y = (int)Math.floor(y) & 255,          // CONTAINS POINT.
        Z = (int)Math.floor(z) & 255;

        x -= Math.floor(x);          // FIND RELATIVE X,Y,Z
        y -= Math.floor(y);          // OF POINT IN CUBE.
        z -= Math.floor(z);

        double u = fade(x),          // COMPUTE FADE CURVES
        v = fade(y),          // FOR EACH OF X,Y,Z.
        w = fade(z);

        int A = p[X ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,    // HASH COORDINATES OF
        B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;    // THE 8 CUBE CORNERS,

        return lerp(w, lerp(v, lerp(u, grad(p[AA ], x , y , z ), // AND ADD
            grad(p[BA ], x-1, y , z )), // BLENDED
            lerp(u, grad(p[AB ], x , y-1, z ), // RESULTS
            grad(p[BB ], x-1, y-1, z ))), // FROM 8
            lerp(v, lerp(u, grad(p[AA+1], x , y , z-1 ), // CORNERS
            grad(p[BA+1], x-1, y , z-1 )), // OF CUBE
            lerp(u, grad(p[AB+1], x , y-1, z-1 ),
            grad(p[BB+1], x-1, y-1, z-1 ))));
    }

    static double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
    static double lerp(double t, double a, double b) { return a + t * (b - a); }
    static double grad(int hash, double x, double y, double z) {
        int h = hash & 15;          // CONVERT LO 4 BITS OF HASH CODE
        double u = h<8 ? x : y,          // INTO 12 GRADIENT DIRECTIONS.
        v = h<4 ? y : h==12||h==14 ? x : z;
        return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
    }

    static final int p[] = new int[512], permutation[] = { 151,160,137,91,90,15,
131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
};
    static { for (int i=0; i < 256 ; i++) p[256+i] = p[i] = permutation[i]; }
}
```

Цей код використовується для генерації тривимірного шуму Перліна. Він починається з обчислювання координати блоку сітки, в якому знаходиться точка.

```
int X = (int)Math.floor(x) & 255, // FIND UNIT CUBE THAT
    Y = (int)Math.floor(y) & 255, // CONTAINS POINT.
    Z = (int)Math.floor(z) & 255;
```

Спочатку обчислюються цілі значення координат X, Y і Z, відповідно до поданої точки x, y та z, округляючи вниз функцію Math.floor. Потім береться побітова операція AND з 255, тобто & 255, що обмежує координати до діапазону від 0 до 255, що є розміром таблиці пермутацій.

```
x -= Math.floor(x); // FIND RELATIVE X,Y,Z
y -= Math.floor(y); // OF POINT IN CUBE.
z -= Math.floor(z);
```

Потім віднімається ціла частина координат, щоб отримати відносні координати точки всередині блоку сітки, в межах [0, 1).

```
double u = fade(x), // COMPUTE FADE CURVES
    v = fade(y), // FOR EACH OF X,Y,Z.
    w = fade(z);
```

Використовується функція fade для згладжування переходів між значеннями в шумі. Вона допомагає уникнути різких змін між блоками сітки, роблячи шум плавним і природним:

```
static double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
```

Це кубічна функція, яка повільно зростає біля 0 і сповільнюється біля 1, згладжуючи перехід.

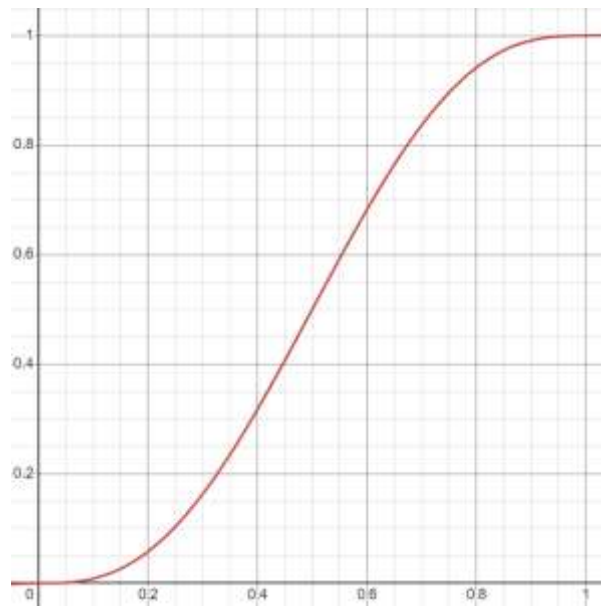


Рис. 1.3. Функція fade (desmos.com)

```
int A = p[X ]+Y, AA = p[A]+Z, AB = p[A+1]+Z, // HASH COORDINATES OF
    B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z; // THE 8 CUBE CORNERS,
```

Здійснюється хешування координат, яке використовується для випадкової генерації градієнтних векторів для кожного вузла блоку сітки. Таблиця p[] містить перестановку індексів, що гарантує випадковість. Потім відбувається

багатокрокова лінійна інтерполяція між результатами функції `grad`, що відповідає впливу кожного з 8 кутів блоку сітки на шукану точку.

```
return lerp(w, lerp(v, lerp(u, grad(p[AA ], x , y , z ), // AND ADD
                                grad(p[BA ], x-1, y , z ), // BLENDED
                                lerp(u, grad(p[AB ], x , y-1, z ), // RESULTS
                                grad(p[BB ], x-1, y-1, z ))), // FROM 8
            lerp(v, lerp(u, grad(p[AA+1], x , y , z-1 ), // CORNERS
                                grad(p[BA+1], x-1, y , z-1 ), // OF CUBE
                                lerp(u, grad(p[AB+1], x , y-1, z-1 ),
                                grad(p[BB+1], x-1, y-1, z-1 ))));
```

Функція `lerp`, linear interpolation, лінійно інтерполює між двома значеннями залежно від параметра:

```
static double lerp(double t, double a, double b) { return a + t * (b - a); }
```

Ця функція вибирає один із 12 можливих градієнтних напрямків на основі хешу, і обчислює скалярний добуток між градієнтом і вектором від точки до вузла сітки.

```
static double grad(int hash, double x, double y, double z) {
    int h = hash & 15; // CONVERT LO 4 BITS OF HASH CODE
    double u = h<8 ? x : y, // INTO 12 GRADIENT DIRECTIONS.
          v = h<4 ? y : h==12||h==14 ? x : z;
    return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}
```

Таблиця `p[]` — це повторення масиву перестановок `permutation[]`. Це забезпечує випадковість і дозволяє згенерувати унікальний шум у кожній точці простору.

Хоча сам по собі шум Перліна генерує лише один шар плавного шуму, у фрактальному моделюванні він часто використовується у вигляді так званої фрактальної суми шуму. Цей підхід передбачає накладання декількох шарів шуму Перліна з різною частотою і амплітудою. Кожен наступний шар (або октава) має вищу частоту (дрібніші деталі) і меншу амплітуду (вплив). Таким чином, фрактальна сума шуму Перліна дозволяє створювати деталізовані, багаторівневі текстури та ландшафти.

Фрактальні підходи у моделюванні, зокрема із використанням шуму Перліна, стали основою для процедурної генерації в комп'ютерній графіці. Вони дозволяють створювати складні та реалістичні текстури й ландшафти з мінімальними обчислювальними затратами. Використання фрактальної суми шуму Перліна дає змогу створювати багаторівневі структури, які виглядають природно, що робить цей метод ідеальним для відеоігор, анімацій та симуляцій природних процесів.

1.3. Постановка задачі

Основною метою кваліфікаційної роботи є розробка програмного додатку, який використовує алгоритми процедурної генерації для створення тривимірних ландшафтів ігрового світу. Створений продукт надасть користувачам можливість генерувати мапи висот з різними параметрами алгоритмів, з'єднувати їх, та експортувати тривимірний об'єкт ландшафту у форматі glTF.

Розробка проекту включатиме в себе декілька етапів, ціллю яких буде створення визначених частин додатку. Основними серед них є:

1. Теоретичні аспекти

1.1. Дослідити шумові алгоритми створення мап висот.

1.2. Дослідити алгоритми, що базуються на симуляції природних явищ, для редагування та створення мап висот.

2. Розробка основних модулів додатку

2.1. Реалізувати алгоритм фрактального шуму Перліна.

2.2. Реалізувати алгоритм Diamond-Square.

2.3. Реалізувати алгоритм генерації діаграм Вороного.

2.4. Реалізувати алгоритм симуляції DLA.

2.5. Реалізувати алгоритм симуляції гідравлічної ерозії.

2.6. Створити меню для роботи з алгоритмами.

2.7. Реалізувати динамічну генерацію ландшафту.

3. Розробка додаткових модулів

3.1. Реалізувати можливість поєднувати мапи висот з коефіцієнтом.

3.2. Створити шейдер для відображення текстур на ландшафті.

3.3. Створити меню для редагування параметрів шейдеру.

3.4. Реалізувати функціонал переміщення персонажа по тривимірному ландшафті, який був створений у редакторі.

3.5. Реалізувати функціонал переміщення персонажа по динамічному ландшафту.

3.6. Реалізація функціоналу експортування тривимірного об'єкта ландшафту.

Результатом виконання кваліфікаційної роботи буде готовий програмний додаток для дослідження процесу створення тривимірних ландшафтів.

1.4. Вибір інструментів для реалізації генерації тривимірного ландшафту

Вибір інструментів для реалізації генерації тривимірного ландшафту є ключовим етапом у розробці складних віртуальних середовищ. Існує доволі широкий набір підходів та інструментів для даного проекту. Кожен із них має свої особливості, залежно від специфіки проекту та вимог до генерації. Розберемо переваги та недоліки основних методів та інструментів.

Можна самостійно розробити систему генерації ландшафтів за допомогою стандартних мов програмування, таких як **C++**, **Python**, або **Rust**, з використанням графічних бібліотек **OpenGL**, **DirectX** або **Vulkan**. Це дає повний контроль над алгоритмами та оптимізацією, але вимагає значних зусиль для реалізації. Серед переваг є повний контроль над усіма аспектами генерації, включно з алгоритмами. Проте з'являється проблема необхідності розробки власного графічного рендерингу, що займатиме значно більше часу на розробку порівняно з використанням готових рішень.

Графічні фреймворки на кшталт **Three.js**, для JavaScript, або **PyOpenGL**, для Python, дозволяють створювати 3D-ландшафти у веб-додатках або десктопних додатках. Ці фреймворки надають засоби для рендерингу тривимірних об'єктів і текстур та мають вбудовані інструменти для процедурної генерації. Проте наявна обмежена продуктивність порівняно з нативними рішеннями на C++ або C#. Також є складнощі в обробці великих ландшафтів у реальному часі.

Використання ігрових рушіїв надасть доступ до корисного вбудованого функціоналу: готові інструменти та технології для роботи з тривимірними об'єктами, текстурами, фізикою та іншими аспектами процедурної генерації. Ігрові рушії є найбільш зручним і ефективним рішенням для інтеграції динамічної генерації у взаємодіючі середовища, тоді як інші інструменти та бібліотеки можуть бути кращими для спеціалізованих завдань або проектів із

високими вимогами до реалістичності. Під ігровими рушіями мається на увазі комплекс програмних рішень, що забезпечують розробку, візуалізацію та симуляцію інтерактивних віртуальних середовищ.

Однією з ключових переваг використання ігрових рушіїв є їх можливість інтерактивної візуалізації результатів у режимі реального часу. Це особливо важливо при процедурній генерації тривимірного ландшафту, де важливо завжди бачити кінцевий результат у тривимірному середовищі й мати можливість динамічно змінювати параметри генерації для отримання бажаного вигляду ландшафту. Також більшість сучасних ігрових рушіїв підтримують кросплатформену розробку, що дозволяє легко адаптувати один і той самий проект для різних платформ — від персональних комп'ютерів до мобільних пристроїв і консолей.

Хоча ігрові рушії включають безліч готових інструментів, вони можуть вимагати використання зовнішніх бібліотек або плагінів для виконання специфічних завдань. Це може ускладнювати процес інтеграції та підтримки проекту. Загалом, вони пропонують потужні інструменти для реалізації процедурної генерації тривимірних ландшафтів, забезпечуючи прискорення розробки, зручність візуалізації та кросплатформеність. Незважаючи на певні обмеження продуктивності та гнучкості, їх використання значно спрощує процес генерації віртуальних середовищ, роблячи його доступним для широкого кола розробників.

Одним з ігрових рушіїв, що стрімко набирає популярність, є **Unreal Engine**. Він забезпечує високу якість графіки, зокрема реалістичні текстури, освітлення та ефекти. Це робить його ідеальним для проектів, де візуальні ефекти мають велике значення. Проте через високу функціональність, рушій може бути складним для початківців, особливо з точки зору налаштування процедурних алгоритмів. Unreal Engine є досить ресурсомістким і вимагає потужних комп'ютерів для повноцінної роботи [37].

Одним з найпопулярніших ігрових рушіїв є **Unity**. Він має інтуїтивний інтерфейс і простий процес налаштування процедурної генерації, що робить його

доступним як для початківців, так і для професіоналів. Також використання C# як основної мови програмування дозволяє легко інтегрувати алгоритми генерації ландшафтів та додаткові інструменти. Хоча Unity добре підходить для багатьох проектів, за якістю графіки він дещо поступається Unreal Engine [38].

Іншим популярний рушієм є **Godot Engine**. Він відомий своєю простотою і зручністю у використанні, завдяки чому він підходить для невеликих проектів та інді-розробників. Проте порівняно з Unreal Engine або Unity, Godot має менш розвинену графічну систему, що може обмежити якість візуалізації складних тривимірних ландшафтів. Також Godot поки що не надає настільки потужних інструментів для процедурної генерації великих тривимірних світів, як інші рушії [39].

Unity є ідеальним вибором для цього проекту завдяки своїй зручності, інтуїтивного інтерфейсу та численних вбудованих бібліотек, які значно спрощують процес розробки. Велика спільнота користувачів Unity, а також багата база ресурсів, яка включає навчальні матеріали, форуми та сторонні плагіни, надає розробникам безліч можливостей для підтримки і розвитку. Ці фактори роблять Unity оптимальним середовищем для створення тривимірних ландшафтів із процедурною генерацією.

Висновки до першого розділу

В результаті написання першого розділу було поставлено задачу на кваліфікаційну роботу, а саме – розробити програмний додаток, який надасть користувачу можливість створювати та редагувати мапи висот для генерації тривимірного ландшафту ігрового світу в Unity. Було проведено аналіз методів генерації тривимірних ландшафтів та визначено їх головні особливості.

Було описано фрактальний підхід в моделюванні та роботу оригінальної імплементації шуму Перліна. Також було проаналізовано та обрано інструменти для розробки даного проекту, наведено їх переваги та недоліки.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ГЕНЕРАЦІЇ ТРИВИМІРНОГО ЛАНДШАФТУ ІГРОВОГО СВІТУ

2.1. Реалізація алгоритмів створення мапи висот

Для створення мапи висот можна використовувати безліч різноманітних методів. Найлегшим та найбільш популярним способом є використання псевдовипадкових шумів, а саме алгоритму, створеного Кеном Перліном. Алгоритм генерації шуму Перліна починає своє життя ще з 1983 року, за весь час якого він був покращений та модернізований. На просторах інтернету можна знайти декілька варіантів його імплементації, проте найпопулярнішим є реалізація покращеного шуму (Improved Noise) самим Кеном Перліном, який він написав використовуючи мову програмування Java у 2002 році. Він є вдосконаленим варіантом оригінального алгоритму шуму Перліна (1983), в результаті отримавши більш плавні результати та підвищену ефективність обчислення [36].

У власній реалізації фрактального шуму використовується вбудована в Unity функція `Mathf.Perlin`. Вона є дуже оптимізованою та працює швидше за будь-яку власну імплементацію на C#. Створену функцію можна поділити на декілька умовних частин: ініціалізація, генерація та нормалізація.

Розберемо хедер функції, який виглядає таким чином:

```
public static float[,] GenerateHeights(int _size, int seed, float _scale, int _octaves, float _persistence, float _lacunarity, Vector2 offset, NormalizeMode normalize_mode)
```

Як можна бачити, функція приймає наступний набір вхідних параметрів:

- **_size**: визначає розмір мапи висот як квадрату з довжиною сторони `_size`;
- **seed**: ініціалізує початкове значення випадковості для обчислень, що її використовують. Однакове значення забезпечує однаковий результат;
- **_scale**: контролює рівень масштабу шуму Перліна. Менше значення призводить до розтягнутих властивостей, коли велике значення додає деталі;
- **_octaves**: відповідає за кількість шарів накладання шуму. Кожна октава додає деталі до загального результату, змішуючи кожен шар з попереднім;

- **_persistence**: визначає амплітуду шуму кожної ітерації накладання шуму, тобто октави. Зазвичай дорівнює 0.5;
- **_lacunarity**: контролює частоту кожну октаву. Зазвичай дорівнює 2;
- **offset**: дозволяє зсувати шум по осі X та Y. Вкрай важливий для генерації нескінченного ландшафту;
- **normalize_mode**: визначає тип нормалізації, локальний чи глобальний.

Спершу відбувається ініціалізація всіх змінних, необхідних для подальшої роботи.

```
float[,] noise_heights = new float[_size, _size];
float max_possible_height = 0;
float frequency = 1;
float amplitude = 1;
System.Random prng = new System.Random(seed);
Vector2[] octave_offsets = new Vector2[_octaves];
for(int i = 0; i < _octaves; i++) {
    float offset_x = prng.Next(-100000, 100000) + offset.x;
    float offset_y = prng.Next(-100000, 100000) - offset.y;
    octave_offsets[i] = new Vector2(offset_x, offset_y);
    max_possible_height += amplitude;
    amplitude *= _persistence;
}
```

Мапа висот: `noise_heights` є масивом, ціллю якого є зберігання фінального результату генерації значень. Є квадратною матрицею.

Частоті та амплітуді надається значення 1, визначаючи першу октаву.

Створюється змінна `prng` із зазначеним `seed`, визначаючи початкове значення в алгоритмі генерації випадкових значень.

Для додавання результату більш унікального вигляду, створюється масив зміщень для кожної октави, що в результаті створює більш натуральний патерн.

Змінна **`max_possible_height`** використовується в майбутньому під час нормалізації значень, якщо обраний тип нормалізації є глобальний. Визначає собою значення, коли при кожній ітерації накладання шуму Перліна було згенеровано значення 1, тобто максимальне. Такий випадок є вкрай рідкісним, проте теоретично можливим.

Зміна значення амплітуди в даному випадку забезпечує, що кожної ітерації октава впливає на можливе максимальне значення коректно, залежно від змінної **persistence**.

Наступним кроком здійснюється ініціалізація значень для локальної нормалізації.

```
float local_max_height = float.MinValue;
float local_min_height = float.MaxValue;
float halfSize = _size / 2f;
if(_scale <= 0) {
    _scale = 0.00001f;
}
```

Значення `local_max_height` та `local_min_height` слідкує за значеннями висоти та зберігає мінімальне та максимальне значення поточного масиву.

Змінна **halfSize** використовується для центрування шуму на мапі висот. Це чітко видно при здійсненні плавної зміни масштабу.

Також для забезпечення відсутності помилок важливо перевірити значення **_scale**, адже воно використовується як дільник та не може бути нулем або менше нуля.

Після ініціалізації важливих змінних можна приступити до безпосередньої генерації мапи висот.

```
for (int x = 0; x < _size; x++) {
    for (int y = 0; y < _size; y++) {
        float noise_height = 0;
        frequency = 1;
        amplitude = 1;
        for (int i = 0; i < _octaves; i++)
        {
            float xCoord = (x-halfSize + octave_offsets[i].x) / _scale * frequency;
            float yCoord = (y-halfSize + octave_offsets[i].y) / _scale * frequency;
            noise_height += (Mathf.PerlinNoise(xCoord, yCoord) * 2 - 1) * amplitude;
            amplitude *= _persistence;
            frequency *= _lacunarity;
        }
        noise_heights[x,y] = noise_height;
    }
}
```

Відбувається проходження по кожній точці (x, y) в масиві `noise_heights` та розраховується значення шуму для кожної октави, використовуючи `offsets`, **frequency** та **amplitude**. На кожній ітерації накладання октави змінні **frequency**

та **amplitude** масштабуються, в результаті змінюючи вплив на кожен наступний шум, за допомогою змін **_persistence** та **_lacunarity**.

Після цього здійснюється оновлення локальних значень максимум та мінімуму.

```
if(noise_height > local_max_height) {local_max_height = noise_height; }  
if(noise_height < local_min_height) {local_min_height = noise_height; }
```

Останнім кроком генерації шуму є нормалізація, в результаті чого всі елементи масиву приймуть значення від 0 до 1.

```
for (int x = 0; x < noise_heights.GetLength(0); x++) {  
    for (int y = 0; y < noise_heights.GetLength(1); y++) {  
        if(normalize_mode == NormalizeMode.Local) {  
            noise_heights[x, y] = Mathf.InverseLerp(local_min_height,  
local_max_height, noise_heights[x, y]);  
        }else if(normalize_mode == NormalizeMode.Global) {  
            float normalized_height = (noise_heights[x,y] + 1) / (2f *  
max_possible_height / 1.75f);  
            noise_heights[x, y] = Mathf.Clamp(normalized_height, 0,  
int.MaxValue);  
        }  
    }  
}
```

В локальній нормалізації використовується вбудована функція **Mathf.InverseLerp**, яка повертає значення від 0 до 1. Вона розраховує, як поточна висота лежить між мінімальним та максимальним значенням.

Під час глобальної нормалізації використовується змінна **max_possible_height**. Вона є завжди однаковою, що є дуже корисним у випадку необхідності отриманні узгодженості між різними частинами (чанками від англ. chunk) нескінченного світу.

В кінці функція повертає створену мапу висот для подальшої обробки та створення тривимірного ландшафту.

```
return noise_heights;
```

Результати роботи реалізованого алгоритму з параметрами size: 512, scale: 100, seed: 10, persistence: 0.5, lacunarity: 2, octaves: 1, 3, 6 зображено на рисунку 2.1.

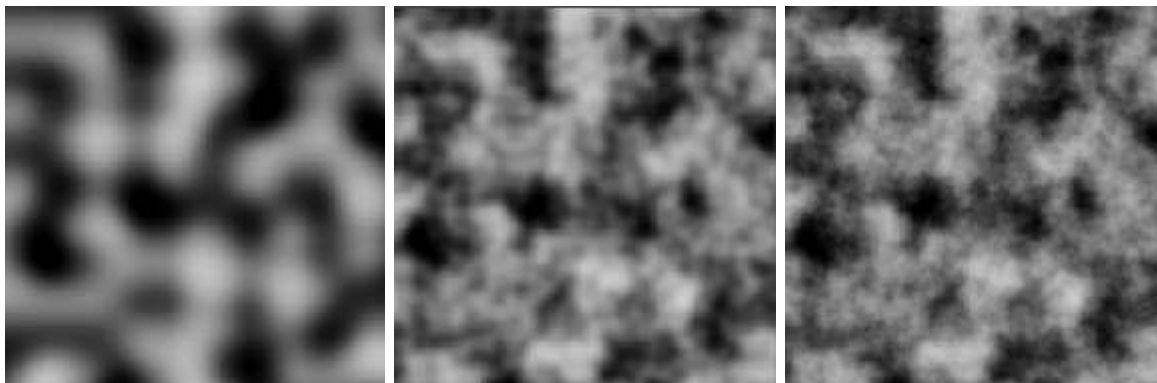


Рис. 2.1. Результат роботи алгоритму генерації фрактального шуму

Як можна бачити, із збільшенням кількості октав збільшується кількість деталей, проте слід сказати, що на певному моменті, приблизно після 5-6 октави, зміну кількості деталей помітити майже неможливо, особливо на маленькому масштабі.

Ще одним популярним та корисним шумом є **шум Вороного**, який ще також використовується для генерації так званих діаграм Вороного. Це підхід, який базується на визначенні пікселю залежно від його близькості до набору точок, що було попередньо розташовані на полотні випадковим чином. Власна імплементація складається з трьох доволі простих функцій.

Створення шуму Вороного починається з виклику **GenerateHeights**.

```
public static float[,] GenerateHeights(int _size, int _pointCount, int seed, bool
random = true) {

    if(random == false) {
        Random.InitState(seed);
    }
    float[,] heights = new float[_size, _size];
    Vector2[] points = GeneratePoints(random, _pointCount, _size);
    for (int x = 0; x < _size; x++) {
        for (int y = 0; y < _size; y++) {
            heights[x, y] = Mathf.Pow(CalculateHeight(x, y, points, _size),
0.35f);
        }
    }
    return heights;
}
```

Даний метод отримує наступний набір вхідних параметрів:

- **_size**: представляє ширину та висоту полотна, на якому буде здійснюватися створення точок та розрахунок висот;

- **_pointCount**: визначає кількість точок, або «сайтів», які будуть розміщені на полотні. Вони будуть відігравати роль так званих центрів регіонів Вороного, та служити фокусними областями, які визначатимуть структуру шуму;
- Параметри **seed** та **random**: визначають початкове значення для генератора випадкових чисел та чи використовувати його;

Перш за все відбувається ініціалізація генератора випадкових значень, якщо вказано, та оголошується двовимірний масив **heights** з розмірами **_size**. Для розрахунку значення висоти потрібно створити визначену кількість точок. Це відбувається в методі **GeneratePoints**.

```
private static Vector2[] GeneratePoints(int _pointCount, float _size) {  
    Vector2[] points = new Vector2[_pointCount];  
    for (int i = 0; i < _pointCount; i++) {  
        points[i] = new Vector2(Random.Range(0, _size), Random.Range(0, _size));  
    }  
    return points;  
}
```

В цій частині коду відбувається створення та ініціалізація масиву двовимірних векторів **points** в діапазоні координат **[0, size]**.

Отримавши інформацію про точки та їх розміщення на полотні, розраховуються значення висот за допомогою методу **CalculateHeights**. Отримане значення підносимо до степеню задля збільшення його яскравості.

```
private static float CalculateHeight(int x, int y, Vector2[] points, int _size)  
{  
    float minDistance = float.MaxValue;  
    foreach (var point in points) {  
        float distance = Vector2.Distance(new Vector2(x, y), point);  
        if (distance < minDistance) {  
            minDistance = distance;  
        }  
    }  
    return minDistance / _size;  
}
```

Для кожної комірки (x, y) знаходиться мінімальна відстань до будь якої точки Вороного, створених в методі **GeneratePoints**. Після цього відбувається нормалізація висоти за допомогою ділення знайденої відстані **minDistance** на

розмір полотна **_size**. Ця операція забезпечує, що значення лежить в діапазоні від 0 до 1.

Взагалі даний алгоритм найкраще себе відображає в генерації процедурних текстур. Для створення ж тривимірних ландшафтів він може бути використаний лише як допоміжний, додаючи цікаві патерни до загальної картини.

Інший доволі відомий алгоритм, **Diamond-Square**, є корисним для генерації мапи висот для тривимірних ландшафтів з ціллю досягнення натурального, переважно гірського, вигляду. В даній реалізації метод, що визивається, отримує на вхід до чотирьох вхідних параметрів.

- **_size**: визначає розмір полотна в одному вимірі, тобто передбачається, що полотно є квадратним. Для коректної роботи алгоритму це значення має слідувати правилу $2^n + 1$, де n – будь-яке натуральне число;
- **_roughness**: контролює рівень випадковості, яка додається до значень висот під час генерації. Вище значення даного параметру призводитиме до більш різких рельєфів, коли нижче – до плавніших;
- **seed**: використовується для ініціалізації генератора випадкових чисел;
- **random**: якщо дорівнює true, то алгоритм кожного разу генерує різний ландшафт, та використовує зазначений seed, якщо false.

```
public static float[,] GenerateHeights(int _size, float _roughness, int seed,
bool random = true) {
    if(IsPowerOfTwo(_size-1) == false) {
        Debug.LogWarning($"_size-1 is not power of 2: {_size-1}");
        return null;
    }
}
```

Перш за все важливо перевірити чи значення розміру слідує вище зазначеному правилу. Воно існує для забезпечення можливості коректного ділення полотна на два.

Загальний алгоритм Diamond-Square можна бачити на рисунку 2.2.

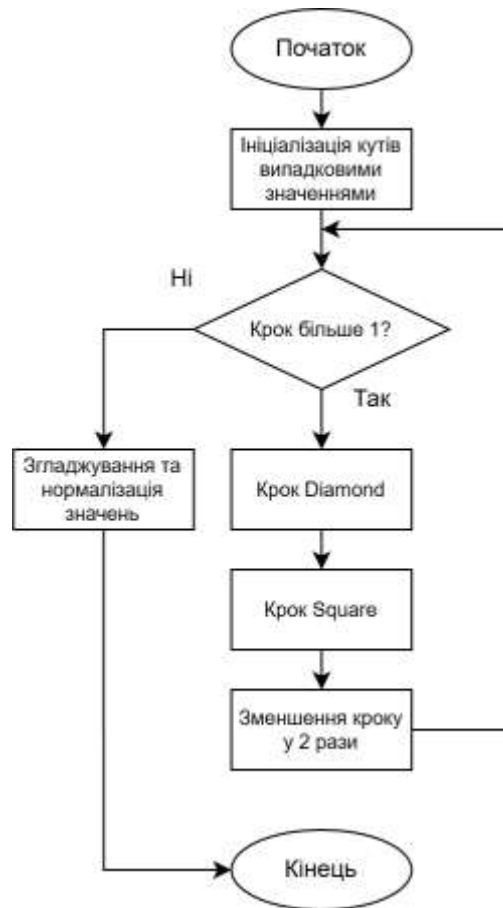


Рис. 2.2. Загальний алгоритм Diamond-Square

Потім здійснюється початкова ініціалізація важливих змінних.

```

float[,] heights = new float[_size, _size];
int edge = _size - 1;

if(random == false) {
    Random.InitState(seed);
}

// Ініціалізуємо кути
heights[0, 0] = Random.Range(0f, 1f);
heights[0, edge] = Random.Range(0f, 1f);
heights[edge, 0] = Random.Range(0f, 1f);
heights[edge, edge] = Random.Range(0f, 1f);

int stepSize = edge;
float scale = _roughness;
  
```

Оголошується мапа висот **heights** із зазначеними розмірами вимірів. Кожен елемент матриці відповідає за значення висоти пікселя з даними координатами на полотні.

Ініціюються значення чотирьох кутів полотна випадковими значеннями. Їх можна назвати фундаментом ландшафту та якорями алгоритму.

Змінні **stepSize** та **scale** контролюють розмір кроку кожної ітерації та кількість випадковості, яка додається до висоти. Значення **scale** ділиться на два кожну ітерацію, щоб зменшувати вплив випадкового значення із зменшенням масштабу.

Алгоритм працює доти, доки розмір відстані до сусіднього кута, квадрату або діаманта, дорівнює більше одиниці. Його роботу можна поділити на два великих кроки (Diamond та Square), кожен з яких відповідає за ініціалізацію висот в своєму діапазоні. Інколи їх називають навпаки, проте сам порядок операцій не змінюється.

Крок Diamond:

```
while (stepSize > 1)
{
    // Diamond step
    for (int x = 0; x < edge; x += stepSize)
    {
        for (int y = 0; y < edge; y += stepSize)
        {
            float avg = (heights[x, y] +
                        heights[x + stepSize, y] +
                        heights[x, y + stepSize] +
                        heights[x + stepSize, y + stepSize]) * 0.25f;
            heights[x + stepSize / 2, y + stepSize / 2] = avg +
Random.Range(-scale, scale);
        }
    }
}
```

Під час кроку **Diamond** алгоритм вираховує центральну точку кожного квадрату визначеного чотирма кутами. Спочатку записується середнє значення в змінну **avg**, після чого до неї додається випадкове значення в діапазоні **[-scale, scale]** та одразу призначається елементу масиву. Випадкове значення вносить до ландшафту певний рівень варіації, імітуючи природні нерівності рельєфу. Згенероване значення на даному кроці впливатиме на сусідні клітинки в наступних кроках, будуючи складність загальної мапи висот.

Крок Square:

```
// Square step
for (int x = 0; x <= edge; x += stepSize / 2)
{
    for (int y = (x + stepSize / 2) % stepSize; y <= edge; y+=stepSize)
    {
```

```

        float avg = (heights[(x - stepSize / 2 + _size) % _size, y] +
                     heights[(x + stepSize / 2) % _size, y] +
                     heights[x, (y + stepSize / 2) % _size] +
                     heights[x, (y - stepSize / 2 + _size) % _size]) *
0.25f;

        heights[x, y] = avg + Random.Range(-scale, scale);
        if (x == 0) heights[edge, y] = heights[x, y];
        if (y == 0) heights[x, edge] = heights[x, y];
    }
}
stepSize /= 2;
scale /= 2f;
}

```

Під час кроку Square розраховуються значення середнім точкам сторін кожного квадрата створеного в кроці Diamond. Так само здійснюється визначення середнього значення кутів, до якого додається випадкове значення в діапазоні `[-scale; scale]`. Крайні клітинки на кордонах матриці також можуть віддзеркалювати значення з протилежного боку, створюючи безшовний ефект або ефект обтікання, який може бути корисним для створення нескінченного ландшафту.

Після заповнення значеннями всіх елементів мапи висот слід виконати ще декілька корисних операцій. Іноді результат Diamond-Square може виглядати занадто різким, для цього здійснюється додатковий етап згладжування та нормалізація значень. В кінці функції повертається готова мапа висот для подальшої обробки.

```

SmoothHeights(heights);
NormalizeHeights(heights);
return heights;

```

Наступний алгоритм, про який піде мова, **DLA**, кардинально відрізняється від Diamond-Square, Voronoi або шуму Перліна, адже DLA зосереджується на симуляції поступового нарощування структури шляхом додавання частинок, які прямують до центрального пікселя та приєднуються до кластеру при зіткненні. Такий підхід дозволяє створювати менш згладжені та значно більш випадкові структури. Так як алгоритм базується на випадковому русі частинки по полотні допоки вона не зіткнеться з іншою, на великих розмірах цей процес може забирати дуже багато часу. Ідеальним рішенням, як мінімум пом'якшенням, цієї

проблеми є ідея збільшення полотна, починаючи з малого. Адже ймовірність того, що частинка зіткнеться з центральною на розмірі полотна 12x12 значно більша, ніж одразу на розмірі 256x256.

Клас DLA містить дві важливі статичні змінні:

```
public const int UPSCALE_FACTOR = 2;  
static int step = 1;
```

Константа **UPSCALE_FACTOR** визначає, наскільки буде збільшуватись полотно, а змінна **step** – скільки разів воно було вже збільшено, починаючи з одиниці, тобто після одного масштабування вона буде приймати значення 2. Алгоритм починається з виводу методу RunDLA.

```
public static float[,] RunDLA(int start_size, int upscale_count)
```

- Змінна **start_size** визначає початковий розмір полотна.
- Параметр **upscale_count** визначає кількість потрібних збільшень полотна.

Загальний алгоритм (рис. 2.3) можна описати наступним чином: після ініціалізації полотна та початкового додавання пікселів відбувається масштабування текстури. Після цього здійснюється масштабування та додавання пікселів таку кількість повторів, скільки було оголошено параметром **upscale_count**. В кінці текстура проходить згладжування та нормалізацію висот.

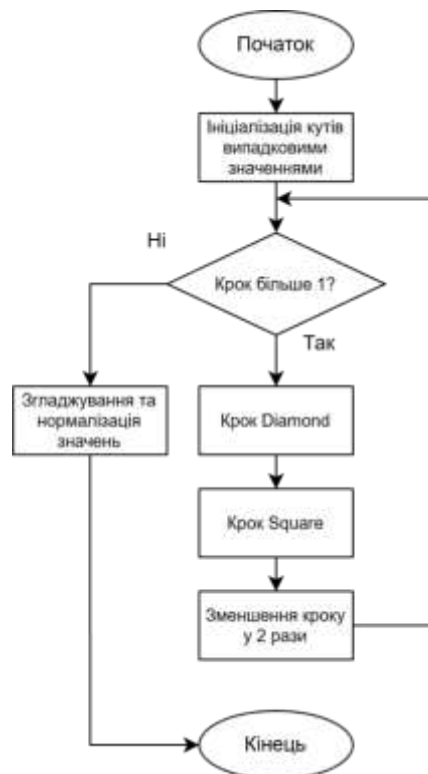


Рис. 2.3. Загальний алгоритм роботи DLA

На початку функції відбувається ініціалізація важливих змінних, яка складається з оголошення певного набору змінних.

```
// Початкова ініціалізація
grid = new Pixel[start_size, start_size];
image = new float[start_size*start_size];
pixels = new List<Pixel>();

// Початковий піксель в центрі
Vector2Int start_pos = new Vector2Int(start_size / 2, start_size / 2);

mainPixel = new Pixel(start_pos, null, Pixel.PixelType.MAIN);
grid[start_pos.x, start_pos.y] = mainPixel;
pixels.Add(mainPixel);
```

Основними структурами даних, з якими відбувається основна частина роботи, є:

- Двовимірний масив об'єктів Pixel **grid**, який відповідає за збереження та візуалізація пікселів на полотні;
- Двовимірний масив типу float **image**, який відіграє роль мапи висот, або, в даному випадку, текстури;
- Список об'єктів типу Pixel **pixels**, який зберігає дані про всі пікселі на полотні та їх зв'язки один з одним;

Дуже важливим кроком під час ініціалізації є розміщення початкового пікселя в центрі полотна, до якого будуть приєднуватися наступні. Для збереження зв'язків було створено вище згаданий клас Pixel (рис. 2.4).

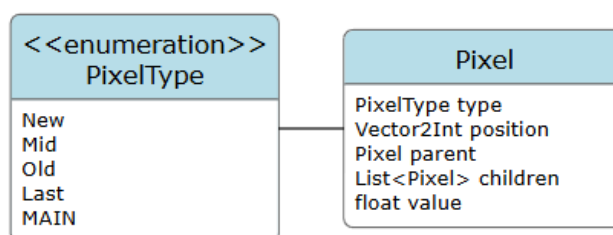


Рис. 2.4. Структура класу Pixel

Він містить інформацію про свій тип (використаний для візуалізації роботи алгоритму), координати свого розміщення на полотні, посилання на батьківський елемент, до якого був приєднаний, список інших частинок, які приєднались до нього, та значення типу float, яке зберігає його відстань до центрального пікселя, починаючи з 1, що є значенням крайніх частинок.

Після оголошення всіх необхідних параметрів відбувається основний цикл алгоритму, в якому здійснюється масштабування та додавання пікселів на полотно.

```
AddPixels(PixelsPerStep(0));
UpscaleTexture();
// Генерація пікселів і зв'язків на кожному етапі
for (int step = 0; step < upscale_count; step++) {
    UpscaleGrid();
    AddPixels(PixelsPerStep(step+1));
    UpscaleTexture();
}
```

Коли було виконано всі операції над пікселями полотна, кінцеву мапу висот **image** необхідно провести через додатковий фінальний етап згладжування за допомогою методу GaussBlur, в якому було реалізовано розмиття по Гаусу, та перевести її в двовимірний вигляд для зручнішого використання в подальшому.

```
image = GaussBlur(image, size);
// 1d -> 2d
float[,] matrix = new float[size, size];
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        matrix[i, j] = image[i * size + j];
    }
}
// знаходження min та max ...
```

Так як завжди є ймовірність, що значення можуть виходити з діапазону [0;1], перед поверненням згенерованої мапи відбувається нормалізація її висот.

```
// Нормалізація
for(int i = 0; i < matrix.GetLength(0); i++) {
    for(int j = 0; j < matrix.GetLength(1); j++) {
        matrix[i,j] = Mathf.InverseLerp(min, max, matrix[i,j]);
    }
}
return matrix;
```

Як можна було помітити, AddPixels(), UpscaleGrid() та UpscaleTexture() є дуже важливими методами в даній імплементації алгоритму DLA.

Метод отримує кількість пікселів, яку потрібно додати до полотна, як параметр **amount**.

```
static void AddPixels(int amount=1) {
    // перевірка значення параметру та оголошення змінних width та height
```

```

...
for(int i = 0; i < amount; i++) {
    randomPos=new Vector2Int(Random.Range(0,width),Random.Range(0,height));
    while(grid[randomPos.x, randomPos.y] != null) {
        randomPos=new Vector2Int(Random.Range(0,width),Random.Range(0,height));
    }
    MovePixelToConnection(randomPos);
}
}

```

Залежно від розмірів полотна здійснюється випадкова генерація координат в діапазоні [0; **width**) та [0, **height**) до тих пір, поки не буде знайдено клітинку, на якій не розміщено піксель, тобто до моменту, коли **grid** з отриманими координатами буде дорівнювати null. Координати розміру зберігаються як тип даних Vector2Int, що зберігає два значення типу int.

Симуляцію руху частинки реалізовано в методі MovePixelToConnection.

```

static void MovePixelToConnection(Vector2Int pos) {

```

Він отримує один вхідний параметр, який визначає координати випадкового розміщеного пікселя в методі AddPixels. Після збереження розміру сторони полотна в змінну **size** основною частиною є цикл while, в якому відбувається рух частинки по масиву.

```

    int size = grid.GetLength(0);
    while (grid[pos.x, pos.y] == null)
    {
        foreach(var offset in offsets) {
            if(pos.x + offset.x > 0 && pos.x + offset.x < size-1 && pos.y +
offset.y > 0 && pos.y + offset.y < size-1) {
                if(grid[pos.x + offset.x, pos.y + offset.y] != null) {
                    Pixel parent = grid[pos.x + offset.x, pos.y + offset.y];
                    Pixel new_pixel = new Pixel(pos, parent, Pixel.PixelType.New);
                    pixels.Add(new_pixel);
                    grid[pos.x, pos.y] = new_pixel;
                    break;
                }
            }
        }
    }
}

```

Допоки поточний елемент масиву є пустим, тобто null, алгоритм виконує зміну значення його координат за допомогою додаванням до них випадкового елементу масиву **offsets**, який зберігає чотири елементи типу Vector2Int із значеннями (0, 1), (1, 0), (0,-1), та (-1,0).

```

if(grid[pos.x, pos.y] == null) {

```

```

        pos += offsets[Random.Range(0, 4)];
    }
    pos.x = Mathf.Clamp(pos.x, 0, size-1);
    pos.y = Mathf.Clamp(pos.y, 0, size-1);
}
}

```

Метод працює поки частинку не буде розташовано біля іншої. Якщо біля неї є елемент, що не дорівнює null, вважаємо що вона знайшла собі батьківський піксель. В такому випадку створюємо новий елемент типу Pixel та додаємо його в список **pixels** та масив **grid**.

В результаті початкової ініціалізації полотно може виглядати наступним чином (рис. 2.5):

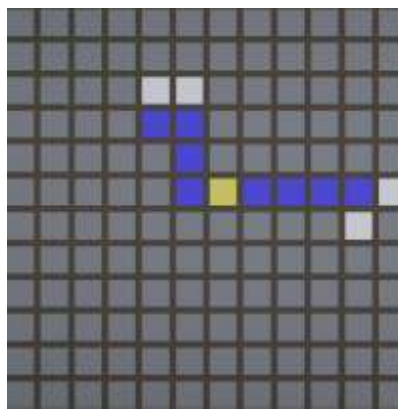


Рис. 2.5. Візуалізація початкового вигляду полотна

Жовтим кольором позначено центральний, початковий піксель, синім – пікселі, що були додані на даному етапі масштабування, сірим – крайні пікселі, які були додані останніми та не мають будь-яких дочірніх елементів.

Після додавання пікселів на полотно відбувається його масштабування за допомогою методу UpscaleGrid.

```

static void UpscaleGrid() {
    int oldSize = grid.GetLength(0);
    int newSize = oldSize * UPSCALE_FACTOR;
    grid = new Pixel[newSize, newSize];
    CreateConnections(mainPixel);
    foreach(var pixel in pixels) {
        grid[pixel.position.x, pixel.position.y] = pixel;
    }
}

```

В змінну **grid** оголошується новий масив з оновленими розмірами та відбувається створення та редагування нових зв'язків методом

CreateConnections, який отримує посилання на головний піксель, що був створений та розміщений під час початкової ініціалізації. Ідея полягає в рекурсивному проходженні по дочірніх елементах, створюючи нові між ними.

```
static private void CreateConnections(Pixel pixel) {  
    Pixel[] children = pixel.children.ToArray();  
    pixel.children.Clear();
```

Перш за все дочірні елементи зберігаються в тимчасовому масиві **children**, а їх список в об'єкті **pixel** очищується, щоб його можна було заповнити пізніше новими. Після цього відбувається проходження по всім дочірнім елементам, де створюється новий екземпляр типу Pixel з координатами, що будуть дорівнювати середині на проміжку від батьківського до дочірнього елементу.

```
foreach(Pixel child in children) {  
    Vector2Int connector_pos=pixel.position*UPSCALE_FACTOR+(child.position-  
pixel.position);  
    Pixel connector = new Pixel(connector_pos, pixel, Pixel.PixelType.Mid);  
    pixels.Add(connector);
```

В даному випадку він зберігається в змінні **connector**, що можна зрозуміти як «з'єднувач».

Так як частинки приєднуються ігноруючи діагональне розміщення один одного, вони будуть виглядати занадто рівними, адже з'єднання в такому випадку може бути лише горизонтальне та вертикальне (рис. 2.6).

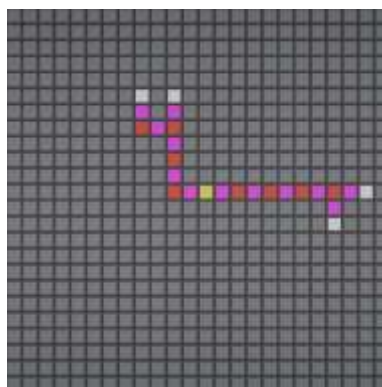


Рис. 2.6. Візуалізація полотна після масштабування із збереженням зв'язків без випадкового відхилення

Для додавання унікальності слід придати середньому елементу з'єднувачу певне відхилення (рис. 2.7).

```
Vector2Int ortho = child.position - pixel.position;  
ortho = new Vector2Int(ortho.y, ortho.x);  
if(ortho.x != 0)
```

```

        ortho.x = ortho.x / Mathf.Abs(ortho.x);
        if(ortho.y != 0)
            ortho.y = ortho.y / Mathf.Abs(ortho.y);
        int ran = Random.Range(0, 11);
        if(ran >= 9) connector.position += ortho;
        else if(ran >= 7) connector.position -= ortho;

```

Для цього спочатку вираховується вектор від батьківського елемента до дочірнього, та переставляються значення x та y, що в результаті створить перпендикулярний вектор. Для забезпечення того, що значення перебувають в діапазоні [-1; 1], координати діляться на свій модуль значення. Потім з ймовірністю 4 з 10 відхилення застосовується до пікселя.

```

        child.parent = connector;
        connector.children.Add(child);
        CreateConnections(child);
    }
    pixel.type = Pixel.PixelType.Old;
    pixel.position *= UPSCALE_FACTOR;
}

```

Після додавання відхилення елемент-з'єднувач призначається як батьківський до поточного дочірнього, та як дочірній до пікселя, що був надісланий в метод як вхідний параметр. В кінці знову визивається метод CreateConnection, але цього разу туди надсилається поточний дочірній елемент. Таким чином буде опрацьовано всі елементи списку **pixels** та оновлено масив **grid** (див. рис. 2.6).

Червоним кольором зображено старі пікселі, що були перенесені при масштабуванні на нове місце, рожевим – пікселі-з'єднувачі.

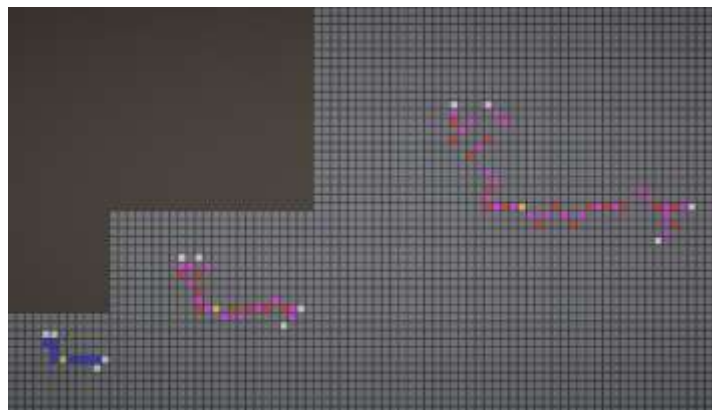


Рис. 2.7. Візуалізація результату операції масштабування полотна з випадковими відхиленням

Для натурального, та загалом кращого, вигляду висоти крайніх пікселів мають бути мінімальними, а висоти центральних, які були додані на перших ітераціях, мають бути максимальні. В даній реалізації визначення висот пікселів здійснюється в тілі методу CalculateValues.

```
static private void CalculateValues() {  
    foreach(var pixel in pixels) {  
        if(pixel.children.Count == 0) {  
            int value = 1;  
            pixel.type = Pixel.PixelType.Last;  
            pixel.value = value;  
            Pixel parent = pixel.parent;  
            while(parent != null) {  
                value++;  
                parent.value = value;  
                parent = parent.parent;  
            }  
        }  
    }  
}
```

Список **pixels** є статичним, тому метод по замовчуванню має на нього посилання.

Так як кожен піксель має лише один батьківський, алгоритм полягає в знаходженні кожного кінцевого елементу, тобто такого, в якого немає жодного дочірнього. Визначення висот пікселів відбувається під час проходження від такий кінцевих елементів до центрального, використовуючи для цього поле **parent**, на кожному кроці збільшуючи висоту наступного на 1. Результат визначення висот можна бачити на рисунку 2.8.

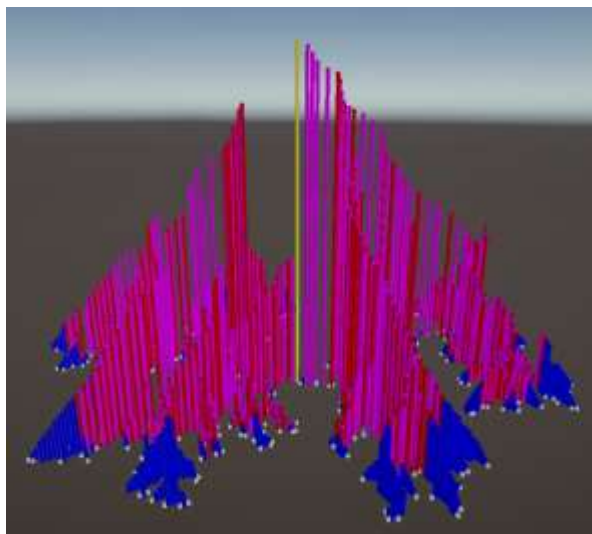


Рис. 2.8. Візуалізація висот пікселів після виклику методу CalculateValues

Після кожного масштабування полотна та додавання пікселів відбувається масштабування текстури, під час якого воно також розмивається, використовуючи розмиття Гаусса. Це поширений метод згладжування, який використовується в обробці зображень для зменшення шуму та виділення важливих деталей. Основою цього методу є Гауссове розподілення (нормальний розподіл) – функція, що має форму дзвона та характеризується середнім значенням і стандартним відхиленням. Результат його роботи з вхідними параметрами **start_size=12**, **upscale_count=5** можна бачити на рисунку 2.9.

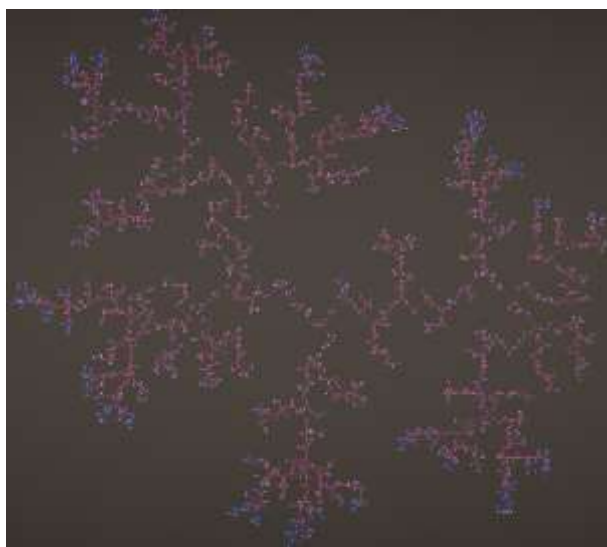


Рис. 2.9. Полотно із зв'язками пікселів

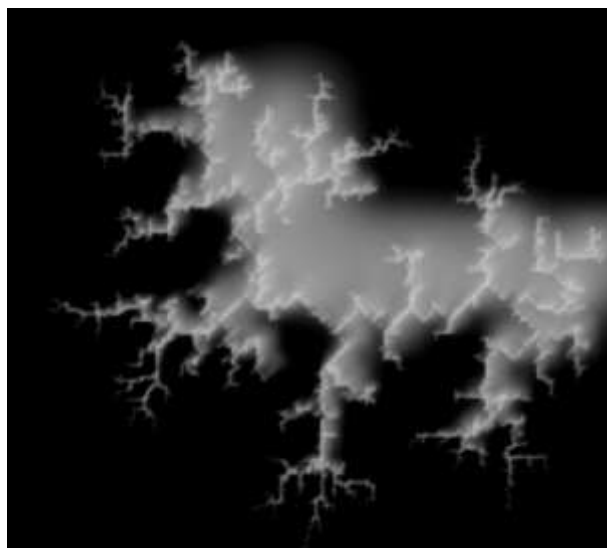


Рис. 2.10. Фінальна текстура

Ще одним алгоритмом, який використовує природне явище для покращення або генерації тривимірного ландшафту, є симуляція **Ерозії**. У випадку з даним методом до його роботи алгоритм вже має мати доступ до

згенерованої мапи висот. Загальний алгоритм його роботи можна бачити на рисунку 2.11.

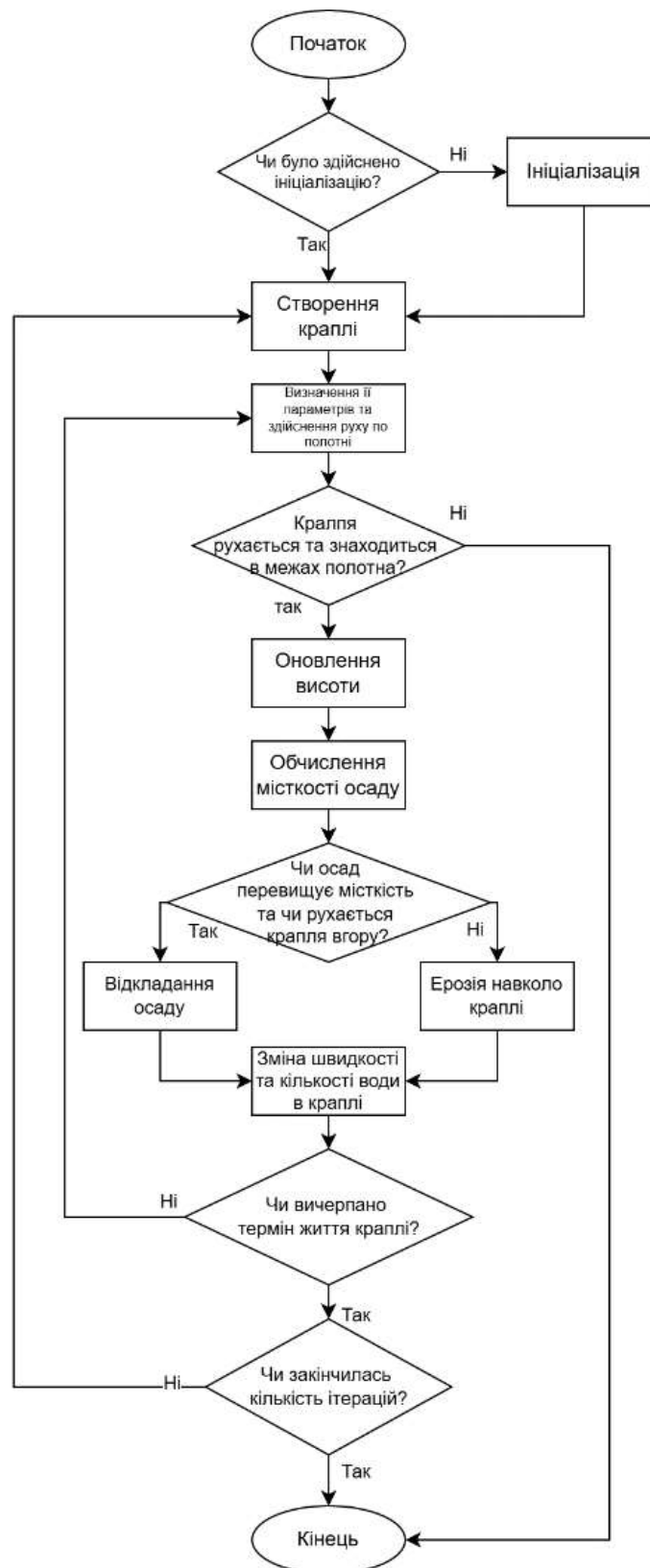


Рис. 2.11. Загальний алгоритм симуляції ерозії

Налаштування параметрів симуляції є дуже важливими в даній імплементації.

```

public int seed;
[Range (2, 8)] public int erosionRadius = 3;
[Range (0, 1)] public float inertia = .05f;
public float sedimentCapacityFactor = 4;
public float minSedimentCapacity = .01f;
[Range (0, 1)] public float erodeSpeed = .3f;
[Range (0, 1)] public float depositSpeed = .3f;
[Range (0, 1)] public float evaporateSpeed = .01f;
public float gravity = 4;
public int maxDropletLifetime = 30;
public float initialWaterVolume = 1;
public float initialSpeed = 1;

```

Цей блок оголошує змінні та налаштування для алгоритму ерозії:

- Параметри ерозії: `erosionRadius`, `inertia`, `sedimentCapacityFactor`, `erodeSpeed`, `depositSpeed`, тощо, контролюють поведінку ерозії;

```

int[][] erosionBrushIndices;
float[][] erosionBrushWeights;
System.Random prng;

```

- **`erosionBrushIndices`** і **`erosionBrushWeights`**: зберігають індекси та ваги для ерозійної кисті (області впливу ерозії);
- **`prng`**: для генерації випадкових чисел на основі `seed`;
- Початок симуляції починається з виклику методу `Erode`.

```

public void Erode(float[] map, int mapSize, int numIterations=1, bool resetSeed=false)
{
    Initialize (mapSize, resetSeed);
    for (int iteration = 0; iteration < numIterations; iteration++) {
        // Create water droplet at random point on map
        float posX = prng.Next (0, mapSize - 1);
        float posY = prng.Next (0, mapSize - 1);
        float dirX = 0;
        float dirY = 0;
        float speed = initialSpeed;
        float water = initialWaterVolume;
        float sediment = 0;
    }
}

```

Головними вхідними параметрами є сама мапа висот, на якій буде проводитися симуляція, та її розмір в одному вимірі.

Спочатку проводиться ініціалізація, під час якої оголошується випадковий генератор (**`prng`**) на основі `seed`, а також визначається область впливу ерозії. Якщо розмір карти чи радіус ерозії змінюються, функція перераховує ці значення.

Потім починається сам процес ерозії, створюючи "краплю води" на випадковій позиції карти. Вона починає рухатися під впливом гравітації, зберігаючи напрямок, швидкість, об'єм води та кількість осаду.

```
for (int lifetime = 0; lifetime < maxDropletLifetime; lifetime++) {
    int nodeX = (int) posX;
    int nodeY = (int) posY;
    int dropletIndex = nodeY * mapSize + nodeX;
    // розрахунок відхилення краплі всередині клітинки
    float cellOffsetX = posX - nodeX;
    float cellOffsetY = posY - nodeY;
    // розрахунок висоти та напрямку руху краплі за допомогою білінійної
    інтерполяції найближчих висот
    HeightAndGradient heightAndGradient = CalculateHeightAndGradient (map,
    mapSize, posX, posY);
    // оновлення напрямку та позиції краплі
    dirX = (dirX * inertia - heightAndGradient.gradientX * (1 - inertia));
    dirY = (dirY * inertia - heightAndGradient.gradientY * (1 - inertia));
    // нормалізація напрямку
    float len = Mathf.Sqrt (dirX * dirX + dirY * dirY);
    if (len != 0) {
        dirX /= len;
        dirY /= len;
    }
    posX += dirX;
    posY += dirY;
```

Цей цикл обчислює нову позицію краплі на основі напрямку і градієнту висоти. Функція **CalculateHeightAndGradient** обчислює висоту і напрямок течії в залежності від нахилу рельєфу. Напрямок краплі оновлюється з урахуванням інерції, а потім нормалізується. У даному контексті градієнт означає нахил або швидкість зміни висоти на рельєфі в певній точці. Він має дві головні складові— **gradientX** і **gradientY**, які показують зміну висоти в горизонтальному та вертикальному напрямках на сітці ландшафту.

Ці градієнти вказують, наскільки крутий нахил рельєфу в кожному напрямку. Чим більше значення градієнта, тим стрімкіший нахил, і тим швидше крапля води буде рухатися в напрямку цього нахилу; менші значення означають більш пологий нахил.

Крапля води зупиняється, якщо вона перестав рухатися або виходить за межі мапи.

```
if ((dirX == 0 && dirY == 0) || posX < 0 || posX >= mapSize - 1 || posY < 0 || posY
>= mapSize - 1) { break;}
```

Потім відбувається обчислення зміни висоти краплі (**deltaHeight**) та її **ємність для перенесення осаду**. Це залежить від швидкості краплі, кількості води та нахилу поверхні.

```
float newHeight = CalculateHeightAndGradient (map, mapSize, posX, posY).height;
float deltaHeight = newHeight - heightAndGradient.height;
// розрахунок ємності осаду краплі
float sedimentCapacity=Mathf.Max(-deltaHeight*speed*water*sedimentCapacityFactor,
minSedimentCapacity);
```

Якщо крапля зберігає в собі більше осаду, ніж може переносити, вона відкладає частину.

```
if (sediment > sedimentCapacity || deltaHeight > 0) {
// якщо рухається вгору (deltaHeight > 0), відкладаємо частину осаду
float amountToDeposit = (deltaHeight > 0) ? Mathf.Min (deltaHeight, sediment) :
(sediment - sedimentCapacity) * depositSpeed;
sediment -= amountToDeposit;
```

Після відкладання відбувається додавання осаду до чотирьох вузлів поточної клітинки. Відкладення не розподіляється по радіусу, як ерозія, щоб воно могло заповнити невеликі ями.

```
map[dropletIndex] += amountToDeposit * (1 - cellOffsetX) * (1 - cellOffsetY);
map[dropletIndex + 1] += amountToDeposit * cellOffsetX * (1 - cellOffsetY);
map[dropletIndex + mapSize] += amountToDeposit * (1 - cellOffsetX) * cellOffsetY;
map[dropletIndex + mapSize + 1] += amountToDeposit * cellOffsetX * cellOffsetY;
}
```

Якщо крапля ще може переносити більше осаду, вона еродує поверхню, забираючи матеріал із сусідніх клітин за допомогою ерозійної кисті.

```
else {
float amountToErode = Mathf.Min((sedimentCapacity - sediment) * erodeSpeed,
-deltaHeight);
```

Щоб уникнути артефакту, коли ерозія руйнує ландшафт занадто сильно, відбувається фіксація значення **amountToErode** за допомогою **Mathf.Min**. Якщо воно буде занадто великим, воно прийме значення **deltaHeight**. Потім цикл **for** проходить по всіх сусідніх вузлах, збережених у **erosionBrushIndices[dropletIndex]**. Ці вузли формують "пензель" ерозії навколо краплі, попередньо визначений у коді для ефективного розподілу ерозії в заданому радіусі.

```
for (int brushPointIndex = 0; brushPointIndex <
erosionBrushIndices[dropletIndex].Length; brushPointIndex++) {
int nodeIndex = erosionBrushIndices[dropletIndex][brushPointIndex];
```

```

        float weighedErodeAmount = amountToErode *
erosionBrushWeights[dropletIndex][brushPointIndex];
        float deltaSediment = (map[nodeIndex] < weighedErodeAmount) ? map[nodeIndex] :
weighedErodeAmount;
        map[nodeIndex] -= deltaSediment;
        sediment += deltaSediment;
    }
}

```

Цей блок виконує або осадження, або ерозію залежно від того, чи перевищує кількість осаду максимальну місткість осаду краплі. Після цього оновлюється швидкість краплі (з урахуванням гравітації та зміни висоти) і об'єм води (враховуючи випаровування).

```

        speed = Mathf.Sqrt (speed * speed + deltaHeight * gravity);
        water *= (1 - evaporateSpeed);

```

Важливими факторами тривалості роботи алгоритму є кількість ітерацій **numIterations**, протягом яких відбувається процес ерозії, і тривалість життя кожної краплі **maxDropletLifetime**, що визначає максимальну кількість її кроків. Додатково, розмір карти **mapSize** впливає на загальну кількість обчислень, оскільки кожен вузол може бути оброблений кілька разів у межах радіусу ерозії. Параметри, такі як інерція **inertia**, швидкість ерозії **erodeSpeed** та осадження **depositSpeed**, також впливають на поведінку крапель, змушуючи алгоритм здійснювати більше чи менше обчислень залежно від інтенсивності ерозійного процесу.

2.2. Генерація тривимірної сітки ландшафту

Для створення тривимірного ландшафту було реалізовано клас **MeshGenerator**, який відповідає за генерацію сітки, налаштування висот, нормалей тощо. Для зберігання даної інформації створено клас **MeshData**.



Рис. 2.12. Структура класу **MeshData**

Він зберігає всі дані про згенеровану сітку тривимірного ландшафту. Вона створюється за допомогою методу `GenerateTerrainMesh` класу `MeshGenerator`.

```
public static MeshData GenerateTerrainMesh(float[,] heightMap, float
height_multiplier, AnimationCurve _height_curve, int level_of_detail, bool
useFlatShading) {
```

Метод приймає наступний набір вхідних параметрів:

1. Двовимірний масив **heightMap**, який зберігає значення висот в діапазоні $[0; 1]$, де координати елементу – координати висоти в тривимірному ландшафті.
2. Значення **height_multiplier** визначає максимальне значення висоти у тривимірному просторі.
3. Параметр **_height_curve** надає контроль над висотами, додаючи можливість змінювати висоту залежно від графіку, який відповідає за швидкість зміни їх значення на проміжку $[0; 1]$. В редакторі Unity він може виглядати наступним чином:

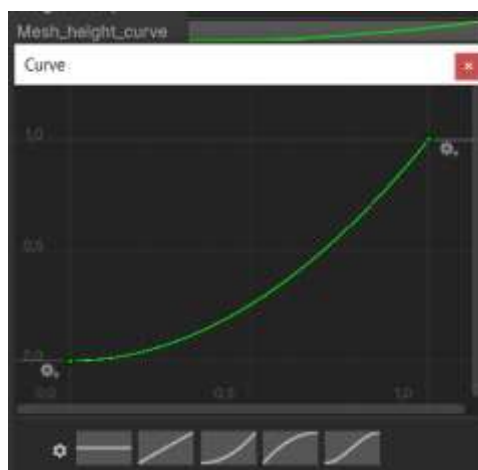


Рис. 2.13. Крива висот

4. Значення **level_of_detail** визначає кількість деталей тривимірної сітки. При збільшенні цього значення зменшується рівень деталей.
5. Параметр **useFlatShading** визначає, чи використовувати Flat Shading. Це техніка, при якій кожна полігональна поверхня об'єкта зафарбовується одним кольором. Для плавного затінення спільні вершини кількох поверхонь згладжуються, що створює поступовий перехід кольору між ними. Проте для Flat Shading кожна частина об'єкта отримує унікальні вершини, а нормалі орієнтовані перпендикулярно до їхньої площини.

Завдяки цьому кожен полігон відбиває світло рівномірно, що і створює ефект чітко окреслених поверхонь та надає об'єкту характерний "плоский" вигляд.

Спочатку метод створює локальну копію кривої висоти `height_curve`, щоб уникнути можливих багатопоточних проблем, а також визначає крок спрощення (`meshSimplificationIncrement`).

```
AnimationCurve height_curve = new AnimationCurve(_height_curve.keys);
int meshSimplificationIncrement = (level_of_detail == 0)? 1: level_of_detail*2;
```

Якщо рівень деталізації **level_of_detail** дорівнює нулю, крок встановлюється на 1, тобто немає спрощення, інакше він обчислюється як подвійний рівень деталізації. Потім визначається розмір сітки та позиції вершин.

```
int borderedSize = heightMap.GetLength (0);
int meshSize = borderedSize - 2 * meshSimplificationIncrement;
int meshSizeUnsimplified = borderedSize - 2;
float topLeftX = (meshSizeUnsimplified - 1) / -2f;
float topLeftZ = (meshSizeUnsimplified - 1) / 2f;
int verticesPerLine = (meshSize-1) / meshSimplificationIncrement + 1;
```

Змінна **borderedSize** — це розмір мапи висот, тоді як **meshSize** — це фактичний розмір сітки з урахуванням рівня деталізації. Також оголошуються координати верхнього лівого кута, **topLeftX** та **topLeftZ**, що дозволяє створити центровану сітку, незалежно від рівня деталізації.

Для коректного згладжування переходів меж сусідніх ландшафтів, відбувається індексування кожної вершини на мапі висот, де крайні отримують негативні індекси. Вони не будуть відображатися на фінальній сітці, проте використовуватимуться при розрахунку нормалей.

```
int[,] vertexIndicesMap = new int[borderedSize, borderedSize];
int meshVertexIndex = 0;
int borderVertexIndex = -1;
for (int y = 0; y < borderedSize; y += meshSimplificationIncrement) {
    for (int x = 0; x < borderedSize; x += meshSimplificationIncrement) {
        bool is_border_vertex = y == 0 || y == borderedSize-1 || x == 0 || x ==
borderedSize-1;
        if(is_border_vertex) {
            vertexIndicesMap[x,y] = borderVertexIndex;
            borderVertexIndex--;
        }else {
            vertexIndicesMap[x,y] = meshVertexIndex;
            meshVertexIndex++;
        }
    }
}
```

Цикл `for` проходить по всім елементам мапи висот з інкрементом поточного рівня деталізації, та перевіряє, чи лежить він на краю мапи. Якщо так – він вважається крайнім і його негативний індекс записується в масив **vertexIndicesMap**. Приклад заповнення індексів мапи 5x5 можна бачити на рисунку 2.14.

-1	-2	-3	-4	-5
-6	0	1	2	-7
-8	3	4	5	-9
-10	6	7	8	-11
-12	-13	-14	-15	-16

Рис. 2.14. Приклад заповнення індексів

Розраховуємо UV з урахуванням наявності країв та створюємо вершини сітки, використовуючи значення з **heightMap**, **height_curve**, та **height_multiplier**. Кожна вершина розташована у тривимірному просторі за допомогою координат **topLeftX** та **topLeftZ**, що центрує мапу висот у просторі.

```
for (int y = 0; y < borderedSize; y += meshSimplificationIncrement) {
    for (int x = 0; x < borderedSize; x += meshSimplificationIncrement) {
        int vertexIndex = vertexIndicesMap[x,y];
        Vector2 percent = new Vector2 ((x-meshSimplificationIncrement) /
(float)meshSize, (y-meshSimplificationIncrement) / (float)meshSize);
        float height = height_curve.Evaluate(heightMap[x,y])*height_multiplier;
        Vector3 vertex_position = new Vector3 (topLeftX +
percent.x*meshSizeUnsimplified, height, topLeftZ - percent.y*meshSizeUnsimplified);

        meshData.AddVertex(vertex_position, percent, vertexIndex);
    }
}
```

Після додавання вершини до об'єкту **meshData**, відбувається генерація трикутників для формування полігонів [40].

```
if (x < borderedSize - 1 && y < borderedSize - 1) {
    int a = vertexIndicesMap[x,y];
    int b = vertexIndicesMap[x+meshSimplificationIncrement, y];
    int c = vertexIndicesMap[x, y+meshSimplificationIncrement];
    int d = vertexIndicesMap[x+meshSimplificationIncrement,
y+meshSimplificationIncrement];
    meshData.AddTriangle (a,d,c);
    meshData.AddTriangle (d,a,b);
}
}
```

Кожна вершина, яка не є крайньою, використовується для створення двох трикутників (A-D-C і D-A-B). Це необхідно для визначення полігонів, які згодом обробляються для відображення текстури та форми ландшафту.

В кінці метод повертає об'єкт типу MeshData із всією згенерованою інформацією.

```
return meshData;
```

Приклад роботи методу для генерації тривимірного ландшафту з використанням фрактального шуму Перліна можна бачити на рисунку 2.15.

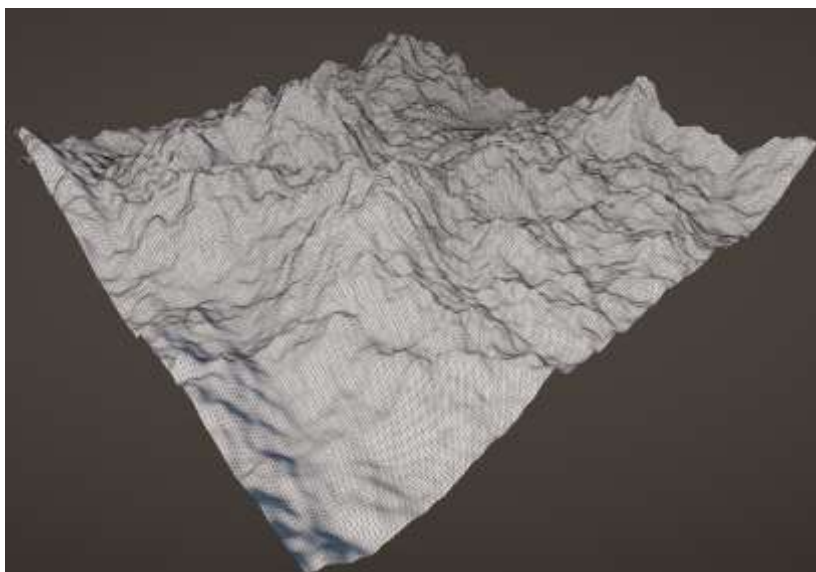


Рис. 2.15. Приклад генерації тривимірного ландшафту з використанням фрактального шуму Перліна

При роботі з налаштуванням параметрів процедурної генерації може з'явитися проблема з визначенням потрібних значень для необхідного результату. Це призводить до потреби використання певної структури даних, яка буде зберігати різні набори, або ассети (від англ. Asset). Ідеальним рішенням даної ситуації є використання Scriptable Objects.

В Unity об'єкти типу Scriptable Objects є спеціалізованими об'єктами, які використовуються як контейнери для зберігання даних. На відміну від C#-скриптів, які наслідують функціонал об'єкта Unity MonoBehaviour, об'єкти типу ScriptableObjects не потрібно прикріплювати до конкретного ігрового об'єкта в сцені. Це означає, що їх слід розглядати переважно як ресурс проекту, подібно до графіки чи інших ассетів, які використовуються в проекті. Хоча зберігати

ігрові дані можна в різних форматах, таких як XML, JSON або навіть .txt файли, та зчитувати їх звідти, зберігання тих самих даних у ScriptableObjects зменшує необхідність створення окремого функціоналу для зчитування або використання сторонніх інструментів для правильного читання даних із цих форматів. Unity автоматично серіалізує та десеріалізує ScriptableObjects, тому доступ до серіалізованих значень значно спрощується [41].

2.3. Створення шейдерів для ландшафтів

Шейдери відіграють ключову роль у створенні реалістичних і візуально привабливих тривимірних ландшафтів, визначаючи вигляд поверхонь об'єктів залежно від їхніх властивостей, умов освітлення тощо. У тривимірних ігрових світах шейдери забезпечують фінальну обробку їхнього вигляду. Вони дозволяють інтегрувати складні ефекти, такі як відображення води, зміна кольору залежно від кута нахилу поверхні чи висоти, детальне накладення текстур і багато іншого. У даному випадку проект створено на Unity Universal Render Pipeline (URP), де є можливість як створювати шейдери вручну за допомогою коду, так і використовувати інструмент Shader Graph, який забезпечує візуальний підхід до розробки.

Shader Graph є більш зручним інструментом у порівнянні з написанням шейдерів вручну через його наочність і простоту використання. Завдяки візуальному інтерфейсу, є можливість створювати складні матеріали за допомогою вузлів, не заглиблюючись у технічні деталі програмування шейдерів. Це дозволяє значно зекономити час і уникнути помилок, які часто виникають при написанні коду. Shader Graph також інтегрується безпосередньо в редактор Unity, що надає можливість миттєвого перегляду результатів змін і налаштувань, що є корисним при роботі з тривимірними ландшафтами, де важлива точний порядок кольорів та текстур. У той час як написання шейдерів вручну забезпечує повний контроль над кожним аспектом обчислень, Shader Graph дозволяє зосередитися на творчих аспектах і швидко створювати прототипи нових ідей [42].

Під час розробки даного проекту було створено два шейдери: для ландшафту та для води. Шейдер для ландшафту використовує 17 параметрів, які

є можливість налаштовувати в редакторі Unity. З найголовніших параметрів є текстури землі, схилу, гір, піску та інформація про їх висоту на ландшафті.

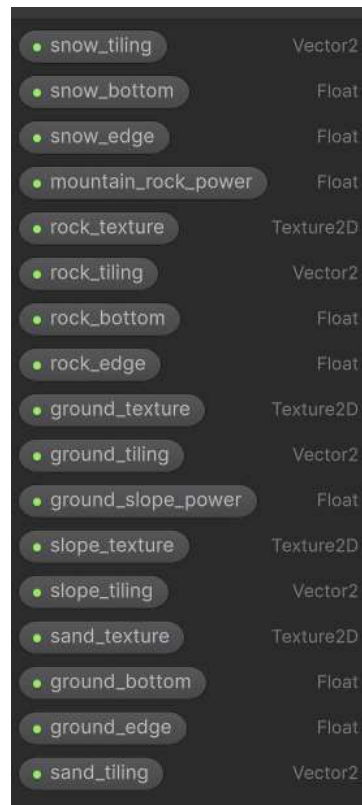


Рис. 2.16. Параметри шейдеру ландшафту

Найбільш головними текстурами в шейдері є текстури землі та схилу, адже вони займають найбільше місця на ландшафті. Для визначення розміщення текстури використовується скалярний добуток нормалі з вектором $(0, 1, 0)$, який є нормалізованим напрямком вгору в глобальному масштабі. Результатом цього скалярного добутку буде значення в діапазоні $[-1;1]$, де 1 означає, що нормаль спрямована в напрямку вгору, а -1 – спрямована в протилежному, тобто вниз. В шейдері результат скалярного добутку затискається, щоб воно не дорівнювало менше нуля, адже на ландшафті не передбачено генерацію полігонів, напрямлених вниз. Потім це значення використовується у вузлі Lerp як коефіцієнт лінійної інтерполяції, визначаючи, яку текстуру де відобразити. Як можна бачити на рисунку 2.17, текстурою, де значення дорівнюватиме 0, буде **dirt_texture**, тобто текстура землі, а текстурою з 1 – **grass_texture**, текстура трави, яка в даному випадку може рости лише на горизонтальних поверхнях. Параметр **ground_slope_power** визначає силу відображення нахилу.

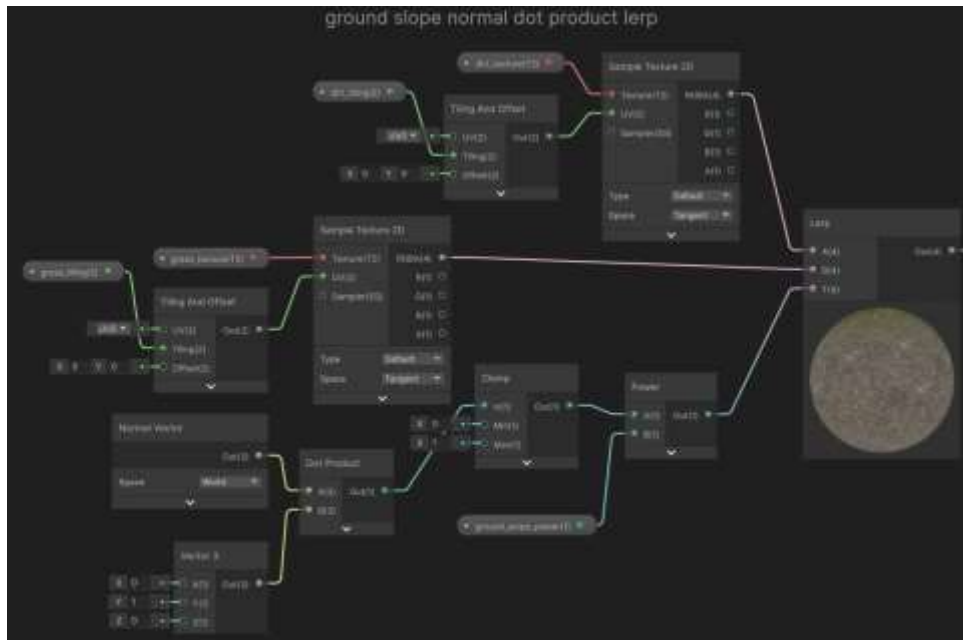


Рис. 2.17. Розміщення текстур трави та землі на схилах

Схожим чином відбувається обчислення текстур гір та снігу, останній який в даному випадку зображений шумом Simple Noise з малим масштабом. Після цього визначається висота, на якій сніг буде відображатись. Для цього було створено підграф ObjectPositionSmoothStep, який приймає інформацію про висоту на силу розмиття. Результатом є маска, яка використовується як коефіцієнт лінійної інтерполяції для визначення висоти нижньої межі початку снігу на ландшафті. Все що є нижче цієї межі приймає текстуру гір. Параметр **mountain_rock_power** визначає силу відображення гір на схилах снігу.

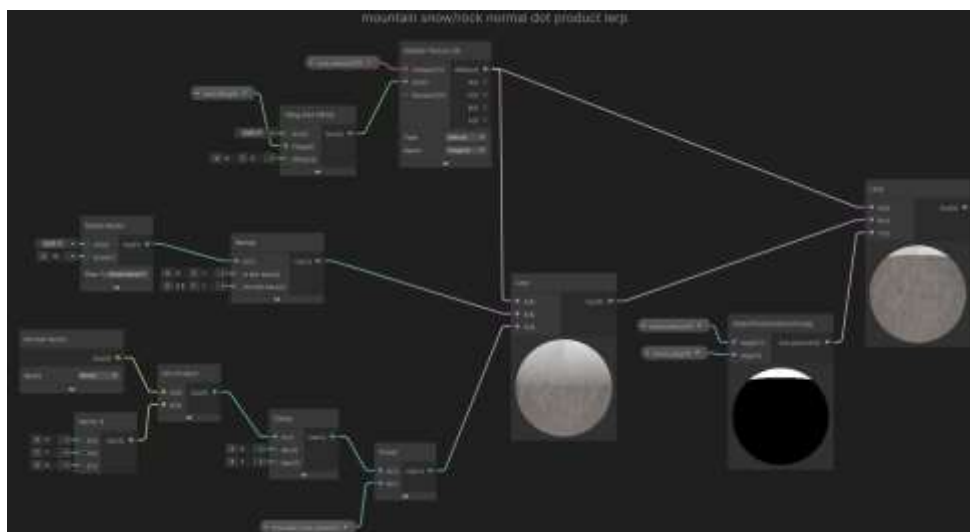


Рис. 2.18. Розміщення текстур снігу та гір на схилах

Останнім кроком є додавання піску, який відображується на темній частині маски, тобто знизу.

... який враховує такі факт

59

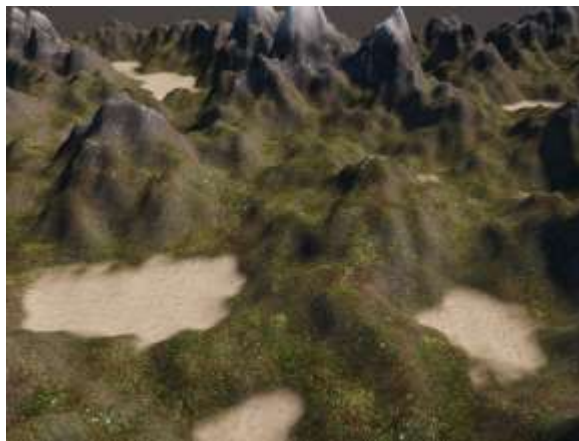


Рис. 2.21. Результат шейдеру

Шейдер для води складається з двох головних частин: обчислення глибини та рух нормалей для створення ілюзії хвиль.

Найголовнішим вузлом в частині обчислення глибини для шейдера води є SceneDepth, який надає доступ до поточного буфера глибини. Це спеціальна текстура, яка зберігає відстань кожного пікселя до камери. Окрім води її також часто використовують для створення шейдеру туману та лазерного поля (рис. 2.22).

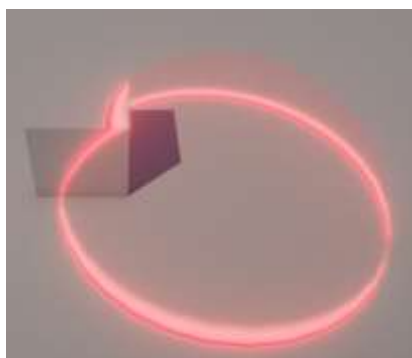


Рис. 2.22. Приклад використання буфера глибини для створення лазерного поля

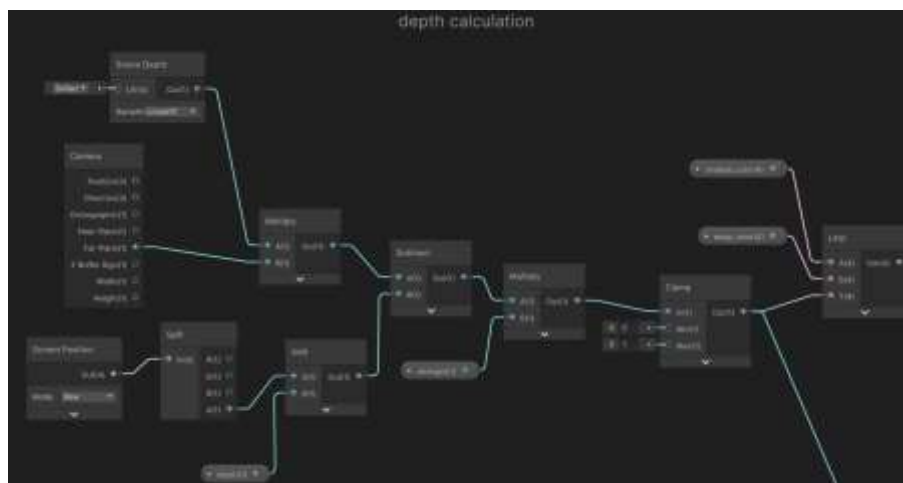


Рис. 2.23. Обчислення глибини води

В результаті використання вузлів SceneDepth, Camera, та ScreenPosition отримуємо градієнт, який змінює своє значення при близькому контакті з іншими ігровими об'єктами, в даному випадку з тривимірним ландшафтом. Його слід використати як коефіцієнт лінійної інтерполяції для зміни кольору води залежно від відстані до берега.



Рис. 2.24. Маска для зміни кольору води



Рис. 2.25. Градієнт води

Для створення ілюзії руху хвиль по воді було використано комбінацію з двох мап нормалей, які рухаються в протилежних напрямках. Рух здійснюється за допомогою вузла Time, який зміщує UV текстури.

відіграє ключову роль. В основі такої системи лежить динамічне створення та управління ділянками ландшафту, які завантажуються лише в області видимості гравця, а також видаляються, коли вони виходять за межі активної зони. Такий підхід дозволяє ефективно використовувати обчислювальні ресурси, адже створюється і обробляється лише те, що бачить гравець. Головною метою реалізації нескінченного ландшафту є забезпечення плавного та реалістичного переходу між зонами, підтримуючи враження безмежного світу. Для цього використовуються алгоритми розбиття карти на частини (чанки), динамічного оновлення даних і системи рівнів деталізації (LOD), які оптимізують продуктивність навіть на слабших пристроях.

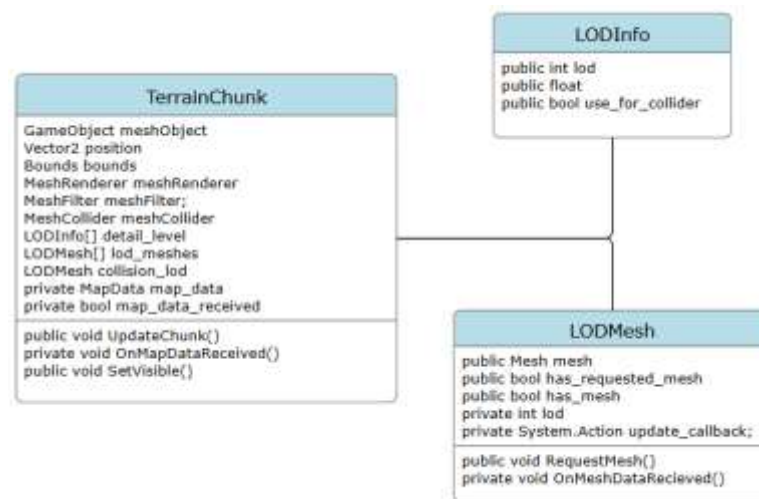


Рис. 2.28. Діаграма класів чанків

Для тестування в майбутньому було імплементовано три різних моди оновлення чанків:

1. Мод Square: створює та оновлює всі чанки навколо поточного розміщення гравця;
2. Мод View: створює та оновлює лише ті чанки, які знаходяться в зазначеному куті погляду гравця;
3. Мод PreloadView: створюються всі чанки навколо гравця, проте відображаються лише ті, на які він дивиться.

Загальний алгоритм оновлення чанків можна бачити на рисунку 2.29.

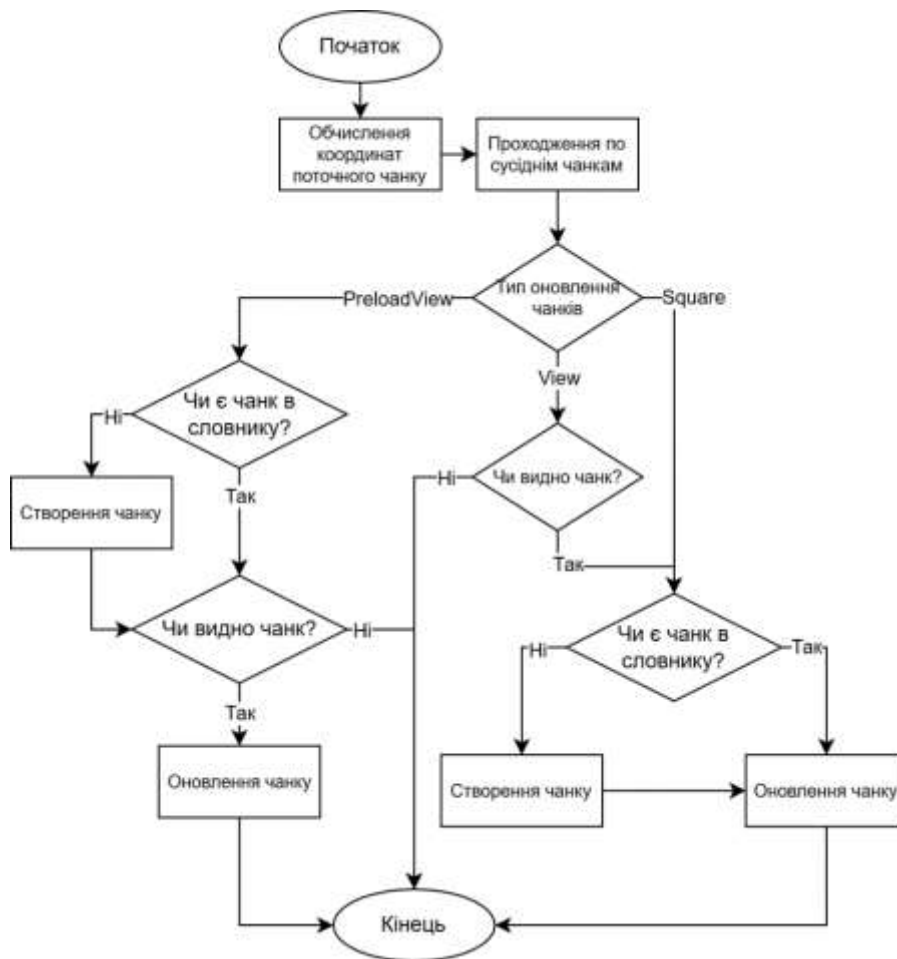


Рис. 2.29. Загальний алгоритм оновлення чанків

Перш за все відбувається обчислення координат поточного чанку.

```

public void UpdateVisibleChunks() {
    int current_chunk_coord_x = Mathf.RoundToInt(viewer_position.x/chunk_size);
    int current_chunk_coord_y = Mathf.RoundToInt(viewer_position.y/chunk_size);

```

Чанки зберігаються в словнику **terrain_chunk_dictionary**, який приймає координати типу **Vector2Int** як ключ та повертає об'єкт типу **TerrainChunk**, структуру якого зображено на Рис. Після цього здійснюється проходження по чанкам, які знаходяться найближче до гравця, залежно від значення **max_view_dist**, яке використовується для обчислення **chunkVisibleAmount**.

```

for(int offset_y = -chunkVisibleAmount; offset_y <= chunkVisibleAmount; offset_y++) {
    for(int offset_x = -chunkVisibleAmount; offset_x <= chunkVisibleAmount; offset_x++) {
        Vector2 chunk_coord = new Vector2(current_chunk_coord_x + offset_x,
            current_chunk_coord_y + offset_y);
        Vector3 chunk_position_on_scene = new Vector3(chunk_coord.x*chunk_size, 0,
            chunk_coord.y*chunk_size) * mapGenerator.terrain_data.uniform_scale;

```

В даному випадку координатами чанку є координати найближчого чанку, тобто на якому стоїть гравець, зміщені за допомогою **offset_x** та **offset_y**.

Незалежно від моду оновлення чанків, ландшафт на якому стоїть гравець завжди повинен оновлюватися.

```
if(offset_x == 0 && offset_y == 0) {  
    if(terrain_chunk_dictionary.ContainsKey(chunk_coord)) {  
        terrain_chunk_dictionary[chunk_coord].UpdateChunk();  
    }else {  
        terrain_chunk_dictionary.Add(chunk_coord, new TerrainChunk(chunk_coord,  
chunk_size, detaillevels, this.transform, mapMaterial));  
    }  
    continue;  
}
```

Гравець стоїть на чанку, якщо змінні зміщення **offset_x** та **offset_y** одночасно дорівнюють нулю. В такому випадку виконується перевірка наявності елемента в словнику за отриманими координатами. Якщо він вже створений – чанк оновлюється, якщо ні – створюється та додається в словник.

В іншому випадку далі відбувається оновлення чанків залежно від обраного моду. Якщо він є **Square**, відбувається просто перевірка наявності чанку в словнику та його створення.

```
switch (chunkUpdateMode) {  
    case ChunkUpdateMode.Square:  
        ..створення та оновлення чанку  
        break;
```

Чанки відображаються якщо вони знаходяться ближче за вказану максимальну відстань. Активні чанки під час даного моду можна бачити на рисунку 2.30.

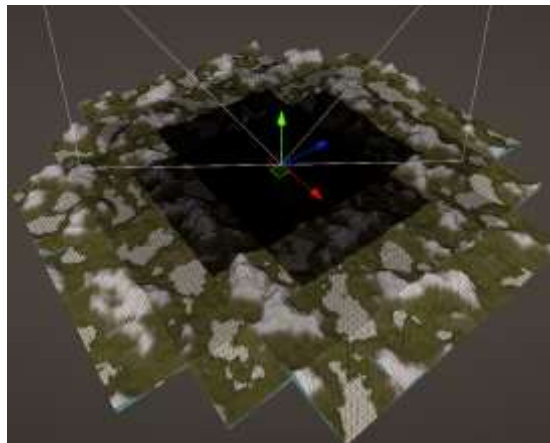


Рис. 2.30. Активні чанки при моді оновлення **Square**

Під час встановленого моду View, створення та оновлення чанків відбувається лише у випадку, коли гравець може його бачити.

```
case ChunkUpdateMode.OnlyView:
    if(IsObjectInView(main_camera.transform.position,
main_camera.transform.forward, chunk_position_on_scene, viewAngle)) {
        ... створення та оновлення чанку
    } break;
```

Активні чанки під час даного моду можна бачити на рисунку 2.31.

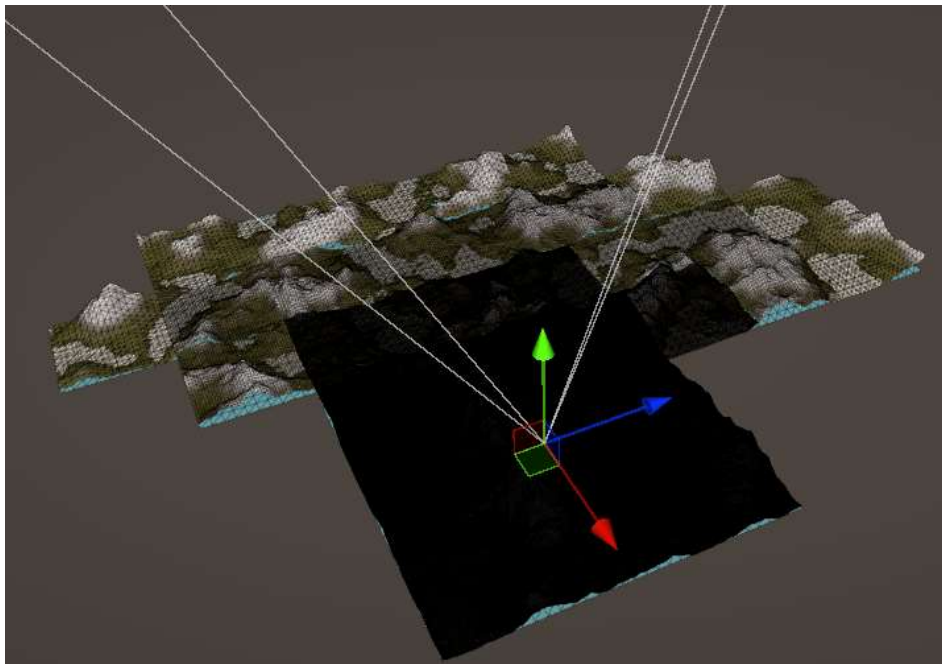


Рис. 2.31. Активні чанки при моді оновлення ViewOnly

У випадку PreloadView чанк створюється, якщо його немає, та оновлюється як при Square, проте його відображення відбувається лише коли гравець його може бачити.

```
case ChunkUpdateMode.ViewPreload:
    ... створення чанку
    var chunk = terrain_chunk_dictionary[chunk_coord];
    if(IsObjectInView(main_camera.transform.position,
main_camera.transform.forward, chunk_position_on_scene, viewAngle)) {
        chunk.UpdateChunk();
    }
    break;
```

Під час створення нового чанку відбувається багато дуже важливих операцій. Загальний алгоритм конструктору класу TerrainChunk можна бачити на рисунку 2.32.

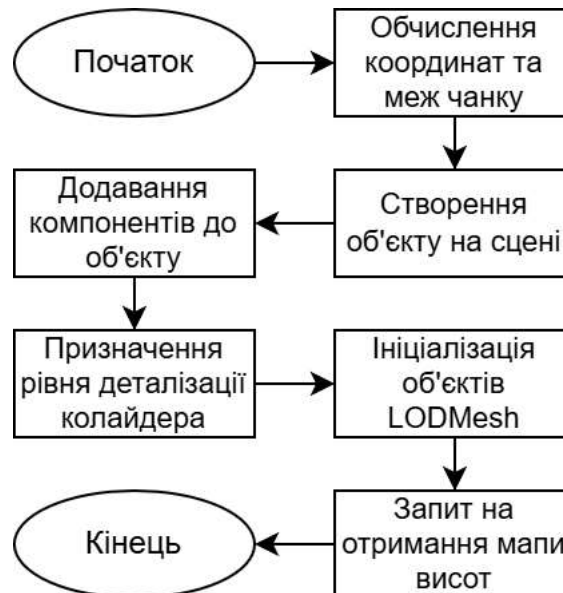


Рис. 2.32. Загальний алгоритм конструктору класу TerrainChunk

Конструктор приймає 5 обов'язкових вхідних параметрів та 1 необов'язковий.

```

public TerrainChunk(Vector2 coord, int size, LODInfo[] detail_level, Transform
parent, Material material, bool auto_update=true) {
    this.detail_level = detail_level;
  
```

Параметр **coord** визначає координати чанку в словнику. Значення **size** визначає розмір сторони одного чанку. Масив значень **detail_level** зберігає в собі інформацію про рівні деталізації, при якій відстані вони активні, та чи використовувати його для генерації колайдеру. Об'єкт **parent** визначає батьківський елемент для об'єкту чанку на сцені. Параметр **material** передає матеріал чанку, який буде призначений ландшафту. Необов'язковий параметр **auto_update** визначає, чи відображати ландшафт одразу при отриманні інформації про мапу висот та меш.

Першим кроком відбувається обчислення таких змінних, як позиція чанку на сцені та його межі. Для цього використовується об'єкт типу **Bounds**.

```

position = coord * size;
Vector3 position_on_scene = new Vector3(position.x,0,position.y);
bounds = new Bounds(position,Vector2.one*size);
  
```

В Unity тип **Bounds** – це структура, яка використовується для представлення обмежувального прямокутного паралелепіпеда у тривимірному просторі. Це об'єкт, який допомагає визначати розміри, центр і просторові межі об'єктів. Він використовуватиметься в методі оновлення чанку.

Потім створюється ігровий об'єкт чанку та призначається його шар для фізичних взаємодій.

```
meshObject = new GameObject("Terrain Chunk");
meshObject.layer = LayerMask.NameToLayer("Ground");
```

При розрахунку координат та розміру чанку на сцені враховується такий фактор, як масштаб **uniform_scale**, зазначений в **terrain_data**.

```
meshObject.transform.position = position_on_scene *
mapGenerator.terrain_data.uniform_scale;
meshObject.transform.parent = parent;
meshObject.transform.localScale = Vector3.one *
mapGenerator.terrain_data.uniform_scale;
```

Для відображення згенерованої сітки ігровий об'єкт повинен містити такі компоненти, як **MeshRenderer** та **MeshFilter**. Також потрібен **MeshCollider** для взаємодії з ландшафтом.

```
meshRenderer = meshObject.AddComponent<MeshRenderer>();
meshFilter = meshObject.AddComponent<MeshFilter>();
meshCollider = meshObject.AddComponent<MeshCollider>();
meshRenderer.material = material;
```

Після того як чанку призначається матеріал, об'єкт ховається зі сцени, адже поки ще не має мапи висот та згенерованої сітки.

```
SetVisible(false);
lod_meshes = new LODMesh[detail_level.Length];
```

Так як один чанк може бути під різними рівнями деталізації, створюється масив об'єктів **LODMesh**, елементи якого зберігатимуть меш різної кількості полігонів. Для створення колайдери

```
for(int i = 0; i < lod_meshes.Length; i++) {
    lod_meshes[i] = new LODMesh(detail_level[i].lod, ()=>
{UpdateChunk(auto_update);} );
    if(detail_level[i].use_for_collider) {
        collision_lod = lod_meshes[i];
    }
}
```

Значення **collision_lod** визначає рівень деталізації колайдери при його створенні. Це використовується для оптимізації.

В кінці здійснюється запит на отримання інформації про висоти точок ландшафту.

```
mapGenerator.RequestMapData(position, OnMapDataReceived);
}
```

Метод `OnMapDataReceived` використовується як callback, який буде визваний після отримання мапи висот.

```
private void OnMapDataReceived(MapData map_data) {  
    this.map_data = map_data;  
    map_data_received = true;  
    UpdateChunk();  
}
```

Після цього виконується оновлення чанку за допомогою методу `UpdateChunk`. Його загальний алгоритм можна бачити на рисунку 2.33.



Рис. 2.33. Загальний алгоритм методу `UpdateChunk`

Він приймає лише один вхідний параметр, який по замовчуванню дорівнює `true`, та визначає, чи відображувати чанк після його створення. Це потрібно для моду оновлення `PreloadView`, під час якого можливі випадки, що чанк створюється та не відображається.

```
public void UpdateChunk(bool show_after_loaded=true) {  
    if(map_data_received == false) {  
        return;  
    }  
}
```

Якщо ще не отримано інформацію про мапу висот – метод закінчує свою роботу. В іншому випадку, потім відбувається обчислення відстані від гравця до найближчого краю чанку, використовуючи квадратну відстань, `bounds.SqrtDistance`, для оптимізації. Потім порівнюється з максимальною видимою відстанню, **`max_view_dist`**, для визначення, чи повинен чанк бути видимим.

```
float distance_from_viewer_to_nearest_edge =  
Mathf.Sqrt(bounds.SqrtDistance(viewer_position) );  
bool visible = distance_from_viewer_to_nearest_edge <= max_view_dist;
```

Якщо чанк є видимим, відбувається певний набір дій. Перш за все обчислюється рівень деталізації ландшафту, залежно від відстані до гравця.

```
for(int i = 0; i < detail_level.Length-1; i++) {  
    if(distance_from_viewer_to_nearest_edge >  
detail_level[i].visible_distance_threshold) {  
        lod_index = i + 1;  
    }else {  
        break;  
    }  
}
```

У випадку, коли новий рівень деталізації відрізняється від попереднього, **`prev_lod_index`**, виконується оновлення меша:

- Якщо меш для цього LOD вже є, **`has_mesh`** дорівнює `true`, він застосовується до `meshFilter`.
- Якщо меш ще не створений, виконується запит на його генерацію за допомогою `RequestMesh`.

```
if(lod_index != prev_lod_index) {  
    LODMesh lod_mesh = lod_meshes[lod_index];  
    if(lod_mesh.has_mesh) {  
        meshFilter.mesh = lod_mesh.mesh;  
        prev_lod_index = lod_index;  
    }else if(lod_mesh.has_requested_mesh == false) {  
        lod_mesh.RequestMesh(map_data);  
    }  
}
```

Для найближчих чанків, з найвищим рівнем деталізації – LOD 0, здійснюється оновлення мешу, який використовується для фізичних взаємодій. Якщо меш вже є, він додається в `meshCollider`. Інакше запитується його генерація.

```

        if(lod_index == 0) {
            if(collision_lod.has_mesh) {
                meshCollider.sharedMesh = collision_lod.mesh;
            }else if(collision_lod.has_requested_mesh == false) {
                collision_lod.RequestMesh(map_data);
            }
        }
    }
}

```

Після цього чанк додається до списку видимих чанків, щоб зберегти інформацію для подальшої обробки.

```

        visible_chunk_last_update.Add(this);
    }
}

```

В кінці залежно від вхідного параметру **show_after_loaded** виконується зображення чанку.

```

if(show_after_loaded) {
    SetVisible(visible); }

```

Висновки до другого розділу

В результаті написання другого розділу було створено блок-схеми для опису структур важливих класів та блок-схеми для опису роботи реалізованих алгоритмів. Також було детально описано безпосередній код імплементованих методів для генерації мап висот та тривимірної сітки ландшафту.

Для коректного накладання текстур на рельєф було створено та описано шейдер за допомогою пакунку ShaderGraph.

Реалізовані алгоритми було використано для створення нескінченного ландшафту, який складається з чанків.

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ, ЙОГО ТЕСТУВАННЯ ТА АНАЛІЗ

3.1. Опис роботи з програмним додатком

При запуску створеного програмного додатку перед користувачем відкривається меню створення та редагування мап висот. Основний набір інструментів для взаємодії розміщений на правій стороні користувацького інтерфейсу (рис. 3.1).

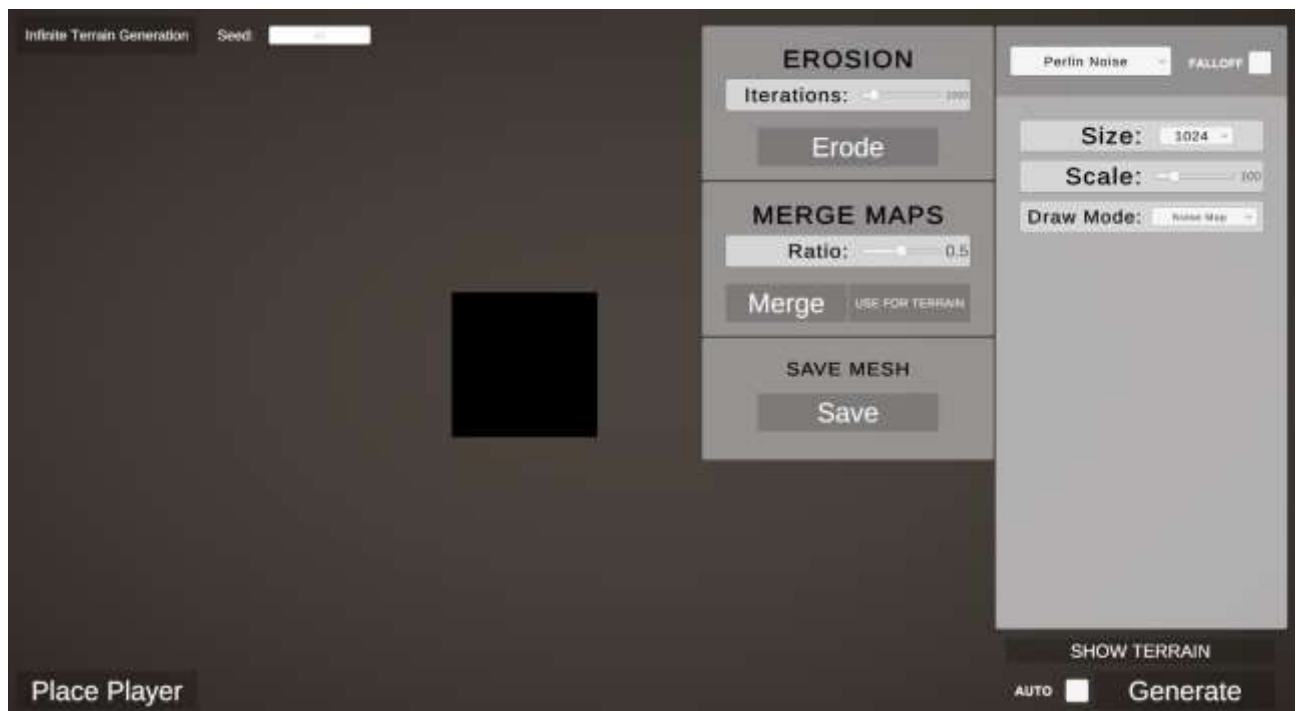


Рис. 3.1. Початковий інтерфейс користувача

При натисканні кнопки Generate за допомогою ЛКМ, яка розміщена у правому нижньому кутку екрану, відбудеться генерація мапи висот з обраними вище параметрами. Результат генерації шуму Перліна з розміром 512 та масштабом 100 можна бачити на рисунку 3.2.

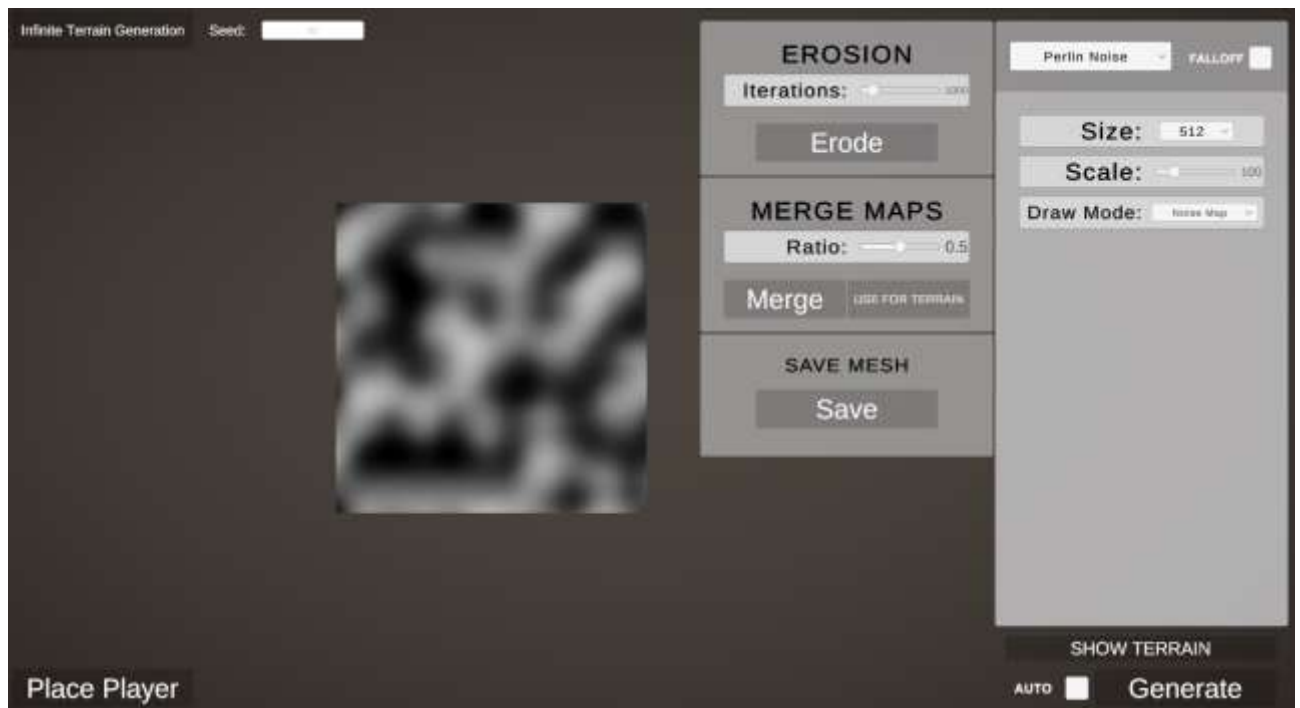


Рис. 3.2. Результат генерації шуму Перліна з обраними параметрами

Для того щоб змінити алгоритм генерації мапи висот слід скористатися меню, що випадає, яке розміщене зверху панелі параметрів поточного алгоритму. Результат генерації фрактального шуму Перліна з розміром 512, масштабом 100, кількість октав 4, можна бачити на рисунку 3.3.

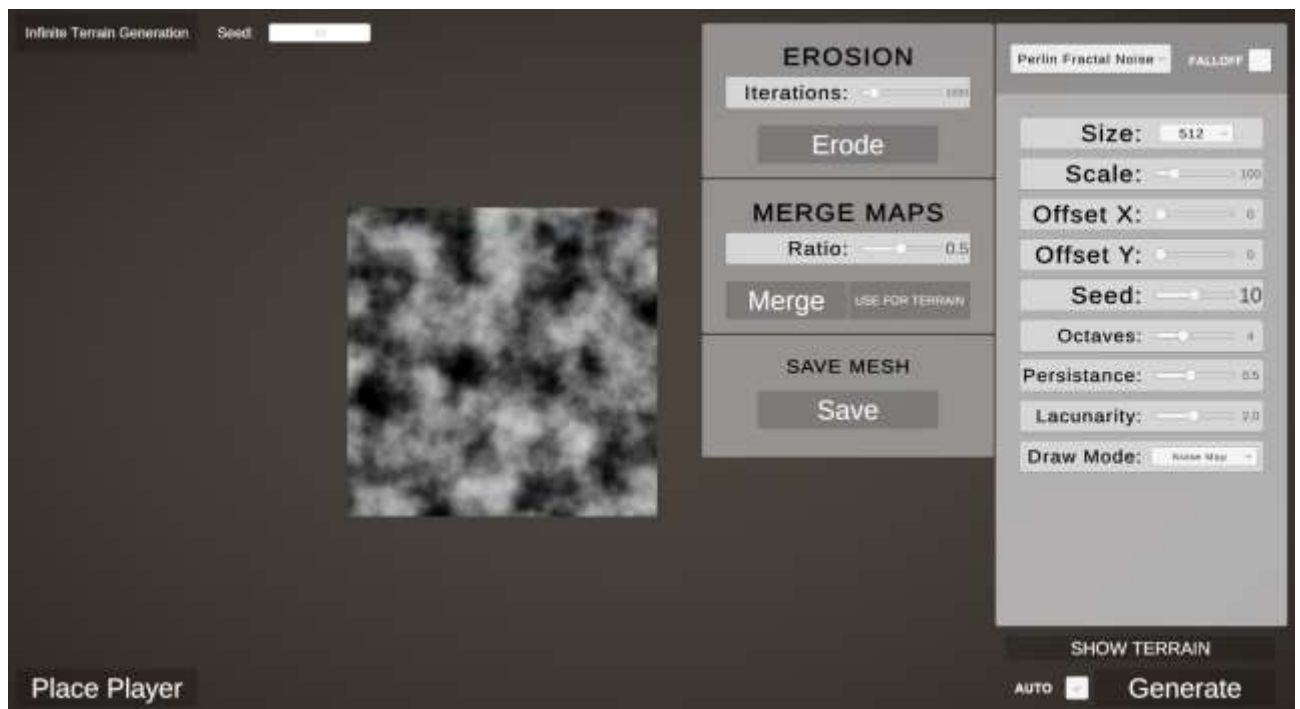


Рис. 3.3. Результат генерації фрактального шуму Перліна з обраними параметрами

Результат генерації мапи висот за допомогою Diamond-Square з розміром 512, який буде змінений на 513, щоб задовольняти умову $2^n + 1$, та різкістю 2, можна бачити на рисунку 3.4.



Рис. 3.4. Результат генерації мапи висот за допомогою Diamond-Square з обраними параметрами

Результат генерації діаграм Вороного з розміром 512, кількістю точок 30, можна бачити на рисунку 3.5.

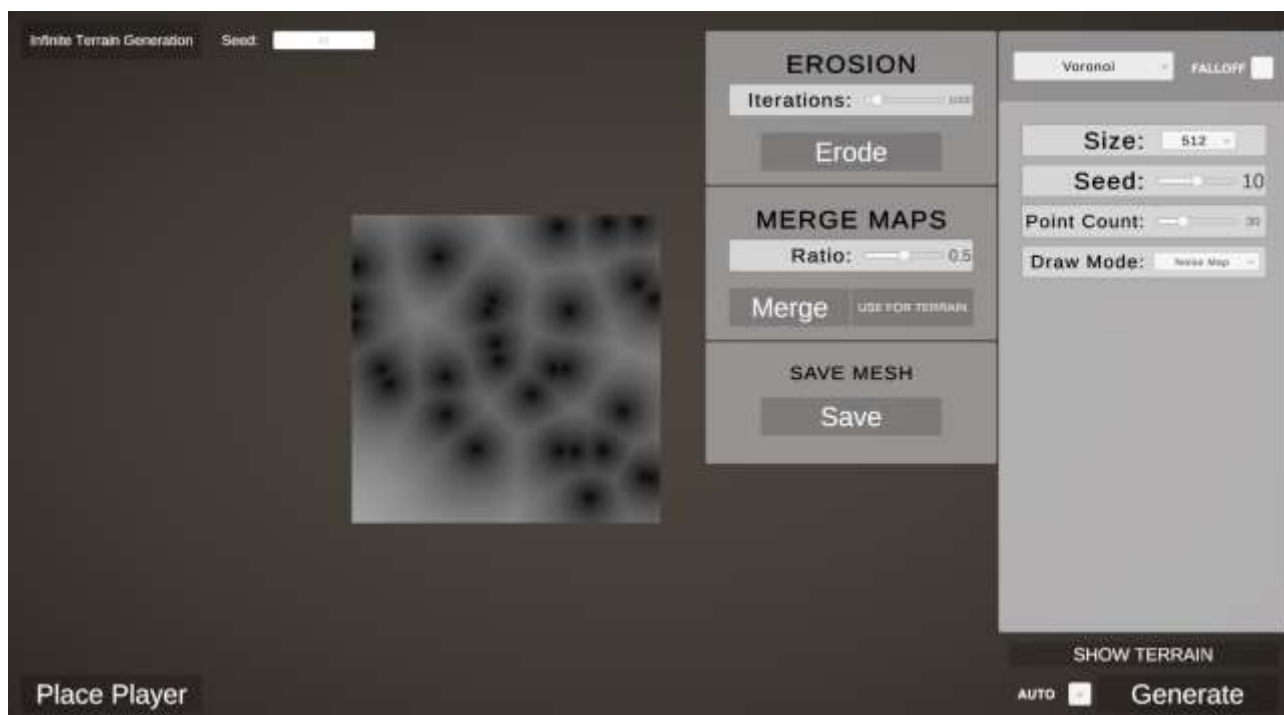


Рис. 3.5. Результат генерації діаграм Вороного з обраними параметрами

Так як алгоритм DLA не має параметру кінцевого розміру полотна, користувачу надається вибір початкового та кількості його збільшення. В даному прикладі початковим розміром є полотно 12x12, та кількістю масштабування – 4.

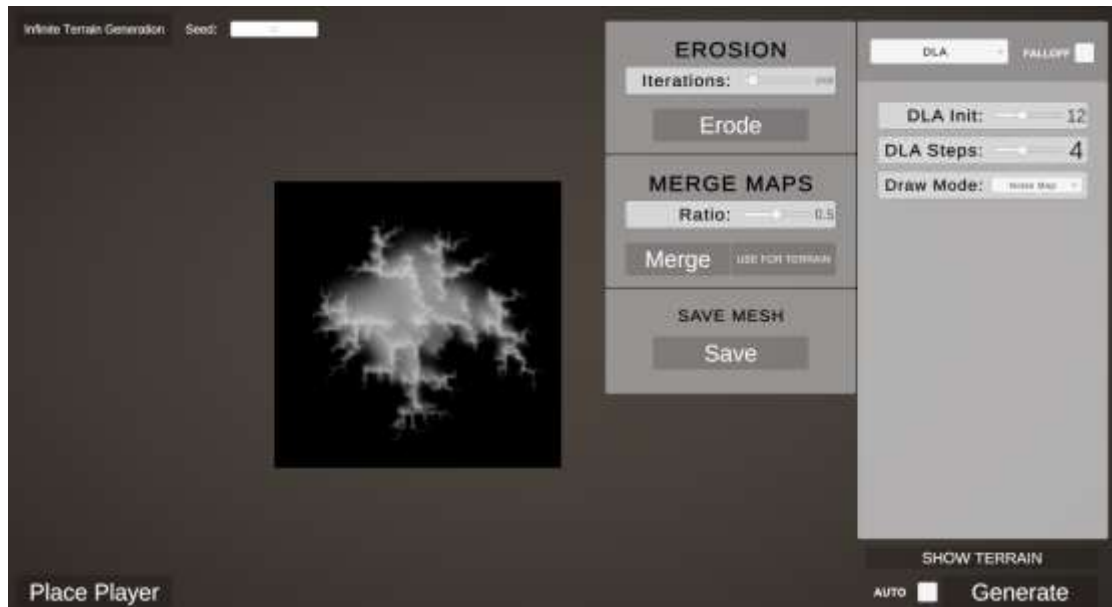


Рис. 3.6. Результат генерації мапи висот за допомогою DLA

Також реалізовано можливість поєднувати різні мапи висот, що в результаті може привести до цікавих рельєфів. В даному прикладі (рис. 3.7) було здійснено поєднання попереднього результату DLA та фрактального шуму Перліна із пропорціями ratio=0.5, тобто 1:1.



Рис. 3.7. Результат поєднання DLA та фрактального шуму Перліна із пропорціями 1:1

Для того щоб використати створену мапу висот для генерації тривимірного ландшафту можна натиснути кнопку «Use for terrain». Результат генерації зображено на рисунку 3.8.

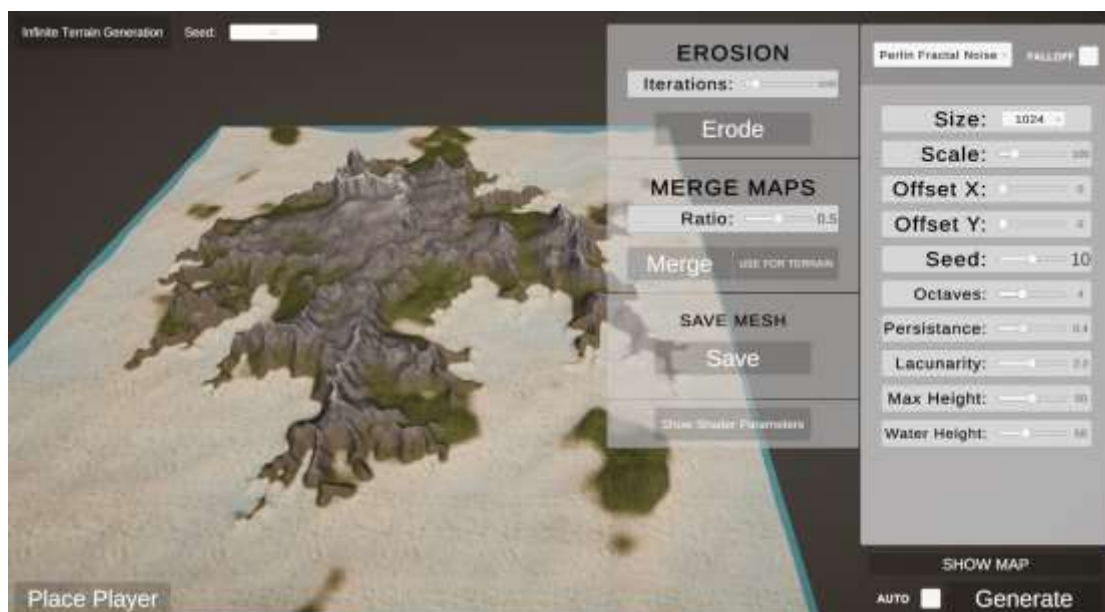


Рис. 3.8. Результат генерації тривимірного ландшафту за допомогою створеної мапи висот

У користувача є можливість змінити налаштування шейдеру. Для цього слід натиснути кнопку «Show Shader Parameters», яка покаже панель для зміни значень параметрів шейдеру.



Рис. 3.9. Зміна параметрів шейдеру

Також було реалізовано можливість піднести ландшафт під вплив ерозії. Для цього слід обрати кількість ітерацій та натиснути кнопку Erode. Різницю між до та після ерозії можна бачити на рисунку 3.10.

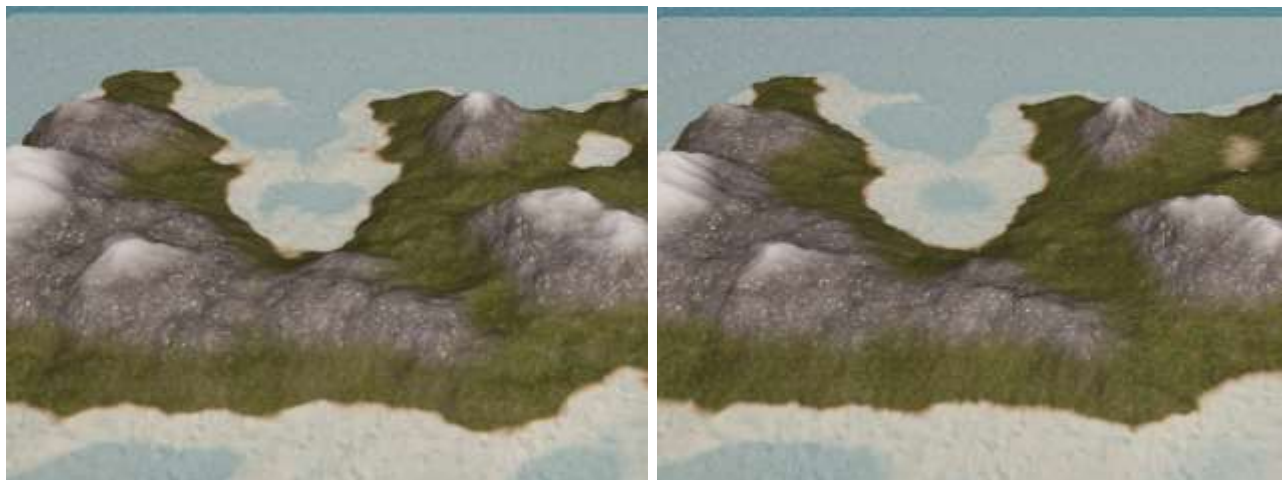


Рис. 3.10. Вплив симуляції ерозії на мапу висот ландшафту

У користувача є можливість побігати по створеному ландшафту ігровим персонажем. Для цього потрібно натиснути на кнопку “Place Player”, яка розміщена зліва знизу. Управління здійснюється за допомогою клавіш WASD або стрілками. Повороти – мишкою, стрибок – space. Знизу також є підказка як повернутися назад до меню редактора: натиснути кнопку E.



Рис. 3.11. Інтерфейс управління персонажем на згенерованому ландшафті

При натисканні кнопки «Infinite Terrain Generation» в меню редактора гравця перекине на іншу сцену (рис. 3.12), в якій відбувається динамічна генерація ландшафту із зазначеним початковим значенням випадкових чисел (seed).



Рис. 3.12. Інтерфейс управління персонажем на динамічному ландшафті

На екран виводиться така корисна інформація, як seed, обраний мод оновлення чанків, кількість кадрів в секунду, кількість завантажених чанків та кількість чанків, що відображаються.

На даній сцені у гравця наявно більше можливостей керування персонажем. Наприклад, є можливість перезавантажити сцену з іншим випадковим початковим значенням на клавішу R. Щоб побачити, як мод оновлення чанків, який можна змінити на клавіши 1-3, впливає на роботу додатку, можна натиснути на клавішу V, яка відобразить ландшафт зверху (рис. 3.13).

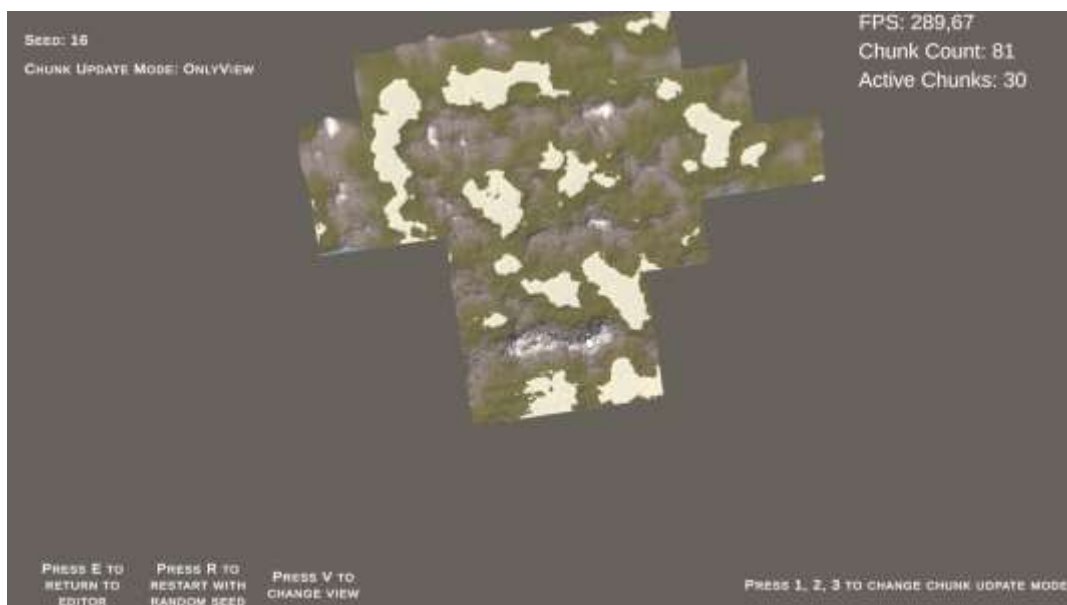


Рис. 3.13. Відображення ландшафту зверху при моді оновлення OnlyView

Згенерований в редакторі ландшафт можна зберегти для використання в інших програмних додатках. Для цього потрібно натиснути кнопку Save та обрати місце. Файл зберігається в форматі glTF.

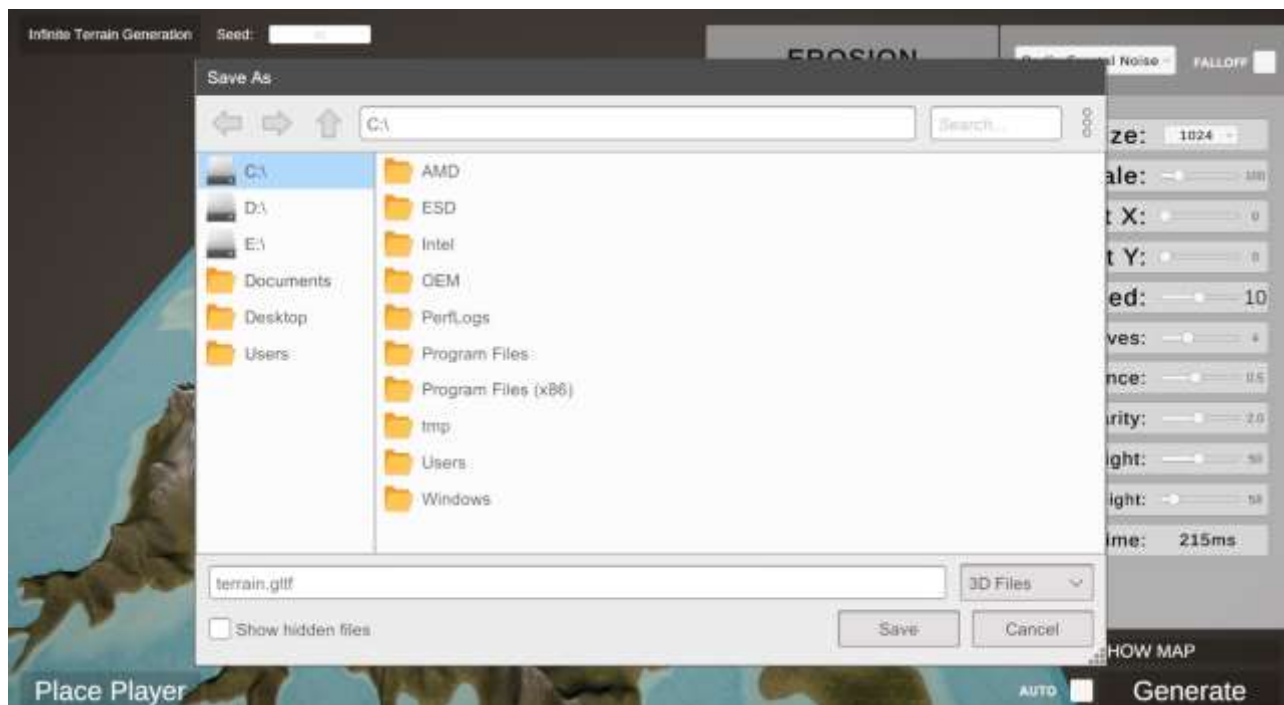


Рис. 3.14. Збереження моделі тривимірного ландшафту

3.2. Тестування роботи програмного забезпечення

Проведено тестування програмного додатку для виявлення та усунення дефектів та проблем.

Таблиця 3.1

Результати тестування

Сцена	Вхідні дані	Очікувані дані	Спостережені дані
Меню редагування мап висот	Зміна алгоритму у меню що випадає	Змінюється панель з налаштуваннями вхідних параметрів алгоритму	Змінюється панель з налаштуваннями вхідних даних алгоритму
	Натиск на кнопку «Generate»	Обраний алгоритм використовує вхідні параметри та відображує мапу висот на полотні	Обраний алгоритм використовує вхідні параметри та відображує мапу висот на полотні

	Натиск на кнопку «Erode»	Здійснюється симуляція ерозії мапи висот вказану кількість ітерацій. Результат відображається залежно від того, на що дивиться користувач.	Здійснюється симуляція ерозії мапи висот вказану кількість ітерацій. Результат відображається залежно від того, на що дивиться користувач.
	Натиск на кнопку «Merge» коли не була згенерована початкова мапа висот.	Результатом є лише згенерована мапа висот обраним алгоритмом.	Результатом є лише згенерована мапа висот обраним алгоритмом.
	Натиск на кнопку «Merge» коли була згенерована початкова мапа висот.	Результатом є співвідношення між згенерованою та початковою мапами висот.	Результатом є співвідношення між згенерованою та початковою мапами висот.
Меню редагування мап висот	Натиск на кнопку «Merge» коли було обрано інший розмір ніж в початковій мапі висот.	На початкову накладається згенерована масштабована мапа висот з обраним співвідношенням.	На початкову накладається згенерована масштабована мапа висот з обраним співвідношенням.
	Натиск на кнопку «Show Terrain»	Камера починає дивитись на тривимірний ландшафт	Камера починає дивитись на тривимірний ландшафт
Меню редагування ландшафту	Натиск на кнопку «Generate»	Обраний алгоритм використовує вхідні параметри та генерує ландшафт з мапи висот	Обраний алгоритм використовує вхідні параметри та генерує ландшафт з мапи висот
	Натиск на кнопку «Show Map»	Камера починає дивитись на мапу висот	Камера починає дивитись на мапу висот

	Натиск на кнопку «Show Shader Parameters»	Панель налаштування алгоритму приховується. Панель налаштування шейдеру відображається.	Панель налаштування алгоритму приховується. Панель налаштування шейдеру відображається.
	Натиск на кнопку «Show Input Parameters»	Панель налаштування шейдеру приховується. Панель налаштування алгоритму відображається.	Панель налаштування шейдеру приховується. Панель налаштування алгоритму відображається.
	Натиск на кнопку «Place Player»	Панель налаштування приховується. На тривимірний ландшафт розміщується персонаж, яким можна керувати.	Деякі параметри з панелі не приховується. Після розміщення персонажу на ландшафт, не можна здійснити вертикальний нахил камери.
	Використання клавіш управління персонажем на ландшафті	Персонаж реагує на дії користувача. Відбувається взаємодія між колайдерами ландшафту та персонажа.	Персонаж реагує на дії користувача. Колайдери ігнорують один одного.
	Натиск клавіші повернення з управління персонажем до редактора	Персонаж приховується. Камера повертається до мапи висот. Панель налаштування відображається.	Об'єкт персонажу приховується. Камера повертається до мапи висот. Панель налаштування алгоритму відображається.

	Спроба ввести не цифровий символ в поле Seed для нескінченої генерації світу	Будь-які не цифрові символи видаляються з поля seed.	Будь-які не цифрові символи видаляються з поля seed.
	Натиск на кнопку «Infinite Terrain Generation»	Завантажується сцена із нескінченною генерацією світу. Значення з поля seed приймається як зерно випадкових чисел.	Завантажується сцена із нескінченною генерацією світу. Значення з поля seed приймається як зерно випадкових чисел.

При тестуванні було виявлено ряд помилок. Наприклад, були певні дефекти в поведінці камери ігрового персонажа, через відсутність у нього компонентів пакунку Cinemachine, який використовується на камерах при редагуванні мап висот. Рішенням може бути відключення непотрібних камер при роботі з персонажем. Також було виправлено проблему відсутності взаємодії колайдерів персонажа та ландшафту, яка була зумовлена некоректним налаштуванням шарів.

3.3. Створення набору даних для аналізу

З ціллю створення набору даних для подальшого аналізу було реалізовано клас DataSetCreator, який відповідає за створення масиву структур з інформацією про згенеровані мапи висот та запис його в .csv файл.

Для збереження результатів роботи алгоритмів було створено структуру CSVHeightMapData.

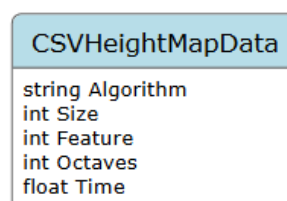


Рис. 3.15. Структура даних про мапи висот CSVHeightMapData

Структура має п'ять полів:

1. **Algorithm:** зберігає назву алгоритму.
2. **Size:** зберігає розмір полотна.
3. **Feature** може зберігати різну інформацію залежно від алгоритму.
Наприклад, в Diamond-Square це Roughness, у Voronoi – PointCount, у FractalPerlinNoise – Scale, DLA - StartPosition.
4. **Octaves:** визначає кількість накладання шуму один на одне або кількість масштабування полотна в алгоритмі DLA.
5. **Time:** час роботи алгоритму.

Метод CreateHeightMapCSV приймає назву файлу, який потрібно буде створити, та починається з ініціалізації списку **height_map_data**.

```
static public void CreateHeightMapCSV(string csv_file_name) {  
    List<CSVHeightMapData> height_map_data = new();
```

Відбувається проходження в циклі for по степеням двійки для визначення розміру полотна. Для алгоритму FractalPerlinNoise в даному випадку таким чином визначаються параметри scale та octaves.

```
    for(int pow_of_2 = 6; pow_of_2 <= 10; pow_of_2++) {  
        int size = (int)Mathf.Pow(2, pow_of_2);  
        for(int scale = 10; scale <= 100; scale+=10) {  
            for(int octave = 1; octave <= 10; octave++) {
```

Для вимірювання часу роботи алгоритмів в даній частині коду використовується клас System.Diagnostics.Stopwatch.

```
                System.Diagnostics.Stopwatch sw = new();  
                sw.Start();  
                float[,] height = FractalPerlinNoise.GenerateHeights(size, 0,  
scale, octave, 0.5f, 2f, Vector2.zero, FractalPerlinNoise.NormalizeMode.Local,  
FractalPerlinNoise.Noise.UnityPerlin);  
                sw.Stop();
```

Після роботи алгоритму отримуємо час його роботи, створюємо екземпляр структури CSVHeightMapData та додаємо його до раніше оголошеного списку **height_map_data**.

```
height_map_data.Add(new CSVHeightMapData  
{  
    Algorithm = "FractalPerlinNoise",  
    Size = size,  
    Feature = scale,
```

```

        Octaves = octave,
        Time = sw.ElapsedMilliseconds
    });
}
} ...

```

Схожим чином відбувається отримання інформації про інші алгоритми. Після сформованого списку дані записуються в файл, назва якого зберігається значенням **csv_file_name**, за допомогою методу WriteToCsv.

```

WriteToCsv(csv_file_name, height_map_data);
}

```

Результат запису даних про мапи висот зображено на рисунку 3.16

Algorithm	Size	Feature	Octaves	Time
FractalPer	64	10	1	0
FractalPer	64	10	2	0
FractalPer	64	10	3	0
FractalPer	64	10	4	1
FractalPer	64	10	5	1

Рис. 3.16. Вміст створеного файлу з даними про мапи висот

Набір даних про чанки має іншу структуру (рис. 3.17).

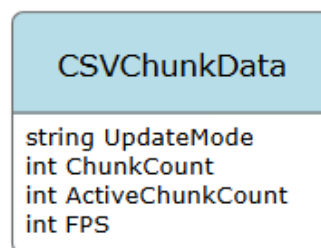


Рис. 3.17. Структура даних про оновлення чанків CSVChunkData

На відміну від мап висот створення набору даних про моди оновлення чанків відбувається під час симуляції руху персонажа по ігровій сцені. Тому в даному випадку найкращим випадком є використання корутинів.

```

private IEnumerator CreateChunkData(string csv_file_name, float delta_time, int
rows_count=1) {
    List<DatasetCreator.CSVChunkData> data=new List<DatasetCreator.CSVChunkData>();
}

```

Після оголошення списку елементів CSVChunkData відбувається обчислення середньої кількості кадрів за секунду.

```

int rows_written = 0;
while(rows_written < rows_count) {
    float timer=0;
    int fps_counter=0;
    while(timer < delta_time) {
        timer += Time.deltaTime;
    }
}

```

```

        fps_counter++;
        yield return null;
    }

```

Для отримання інформації про кількість активних та завантажених чанків використовується метод з класу `InfiniteTerrainGeneration`, який відповідає за нескінченну генерацію ландшафту та зберігає потрібні для аналізу дані.

```

infiniteTerrainGeneration.GetInfoAboutChunks(out int all, out int active);
data.Add(new DatasetCreator.CSVChunkData{
    UpdateMode = infiniteTerrainGeneration.UpdateMode,
    ChunkCount = all,
    ActiveChunkCount = active,
    FPS = fps_counter
});
rows_written++;

```

Після створення набору даних викликається метод `WriteToCsv` для запису їх в файл типу csv.

```

    }
    DatasetCreator.WriteToCsv(csv_file_name, data, append);
}

```

Результат запису даних про чанки зображено на рисунку 3.18.

UpdateMode	ChunkCount	ActiveChunkCount	FPS
Square	81	73	292
Square	81	73	351
Square	90	73	353
Square	90	73	359
Square	90	69	360
Square	90	73	361
Square	90	73	360

Рис. 3.18. Вміст створеного файлу з даними про чанки

3.4. Візуалізація та аналіз роботи алгоритмів генерації

Для здійснення аналізу та візуалізації результатів було використано Google Colaboratory. Це безкоштовна інтерактивна платформа для роботи з Jupyter Notebook в хмарі, розроблена Google. Вона дозволяє писати та виконувати код на Python, що особливо корисно для навчання, досліджень у галузі машинного навчання та аналізу даних. Colab надає доступ до потужних обчислювальних ресурсів, таких як GPU та TPU, що дозволяє виконувати складні обчислення без потреби в локальній високопродуктивній техніці.

Спершу було виведено вплив збільшення кількості октав в алгоритмі фрактального шуму Перліна (рис. 3.19).

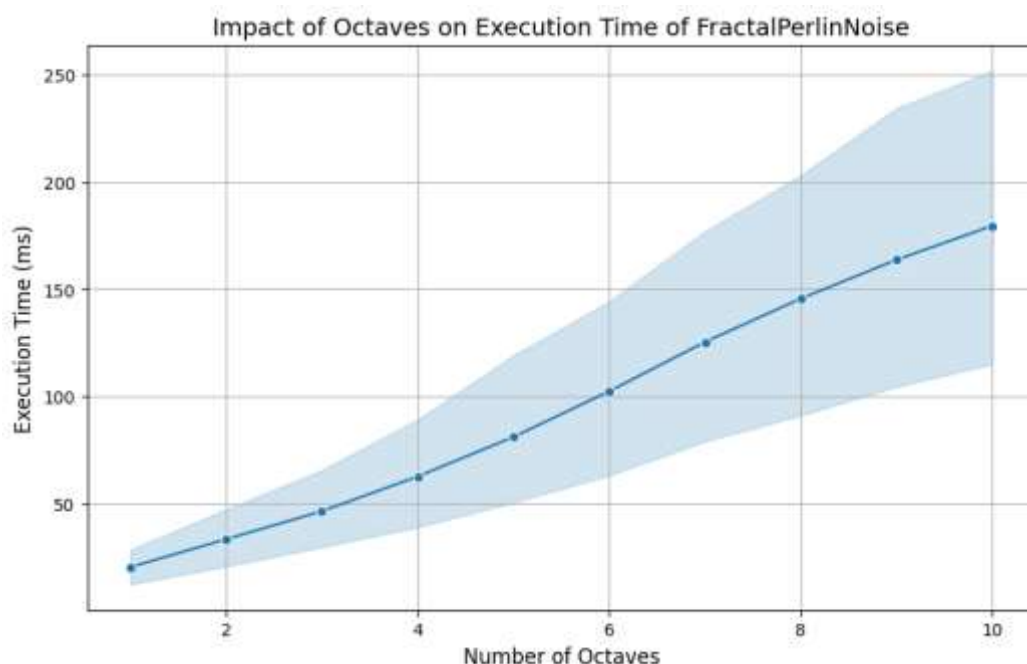


Рис. 3.19. Вплив кількості октав на час роботи фрактального шуму Перліна

Іншим популярним методом, який доволі часто використовують для генерації мап висот для тривимірного ландшафту є Diamond-Square. Топу цікаво було б порівняти швидкість його роботи з фрактальним шумом Перліна. Для досягнення конкурентного вигляду використовується чотири октави накладання шуму Перліна.

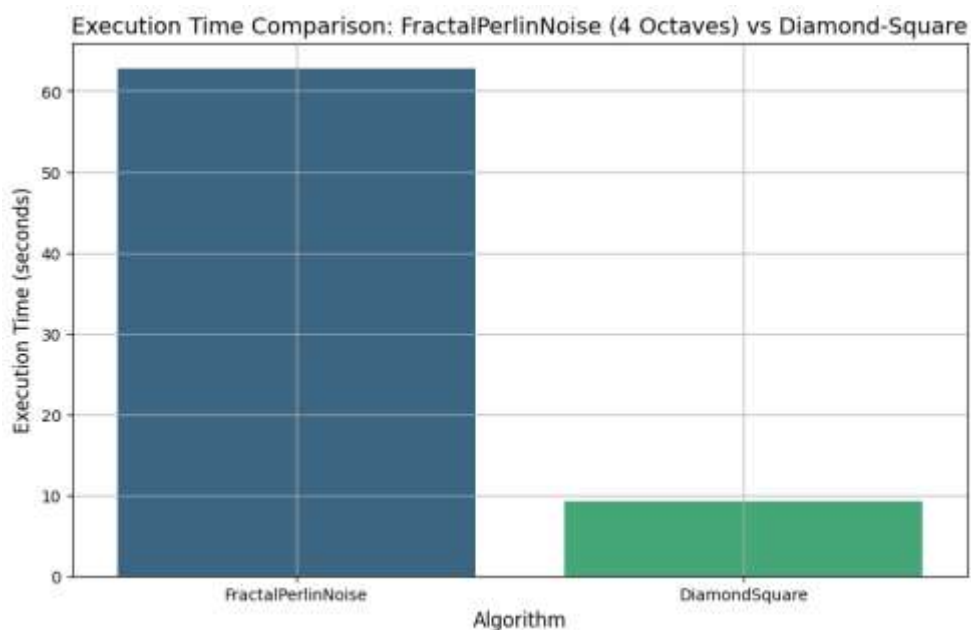


Рис. 3.20. Час роботи фрактального шуму Перліна та алгоритму Diamond-Square

Як можна бачити алгоритм Diamond-Square працює набагато швидше, та в результаті створює достатньо приємний шум (рис. 3.21).

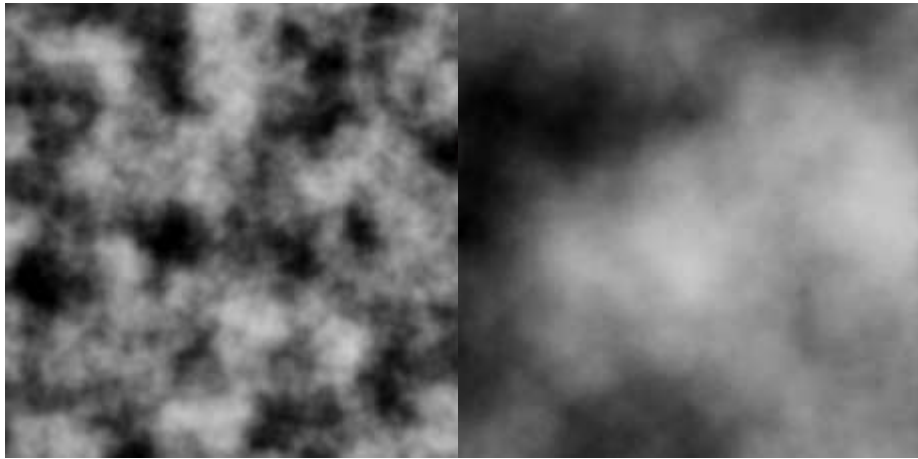


Рис. 3.21. Результати роботи фрактального шуму Перліна з октавою 4 (зліва) та алгоритму Diamond-Square (справа)

Для генерації мап висот нескінченного ландшафту було використано фрактальний шум Перліна через наступні властивості:

- Шум Перліна набагато зручніше масштабувати.
- Шум Перліна дозволяє легко регулювати рівень деталізації через параметри, такі як масштаб, амплітуда, кількість октав, що забезпечує більший контроль над виглядом ландшафту.
- Шум Перліна забезпечує плавні переходи між сусідніми чанками, що ідеально підходить для генерації нескінченного світу, коли Diamond-Square вимагає додаткових обчислень для вирівнювання меж між чанками.

Було також візуалізовано вплив розміру полотна на швидкість роботи вищезгаданих алгоритмів (рис. 3.22).

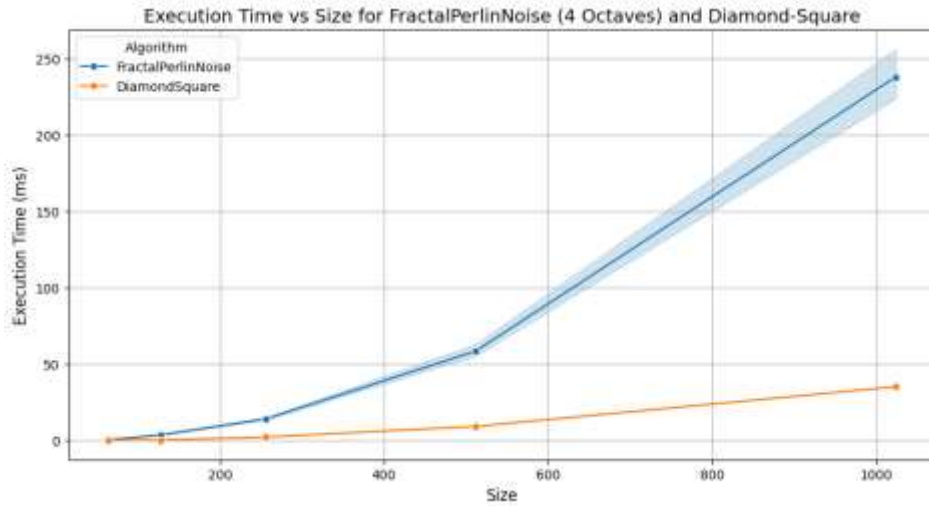


Рис. 3.22. Швидкість роботи фрактального шуму Перліна з октавою 4 та алгоритму Diamond-Square на різних розмірах полотна

Як можна бачити, алгоритм Diamond-Square швидше за фрактальний шум Перліна при будь-яких розмірах полотна [43].

На час роботи також впливає кількість точок в алгоритмі генерації діаграм Вороного (рис. 3.23).

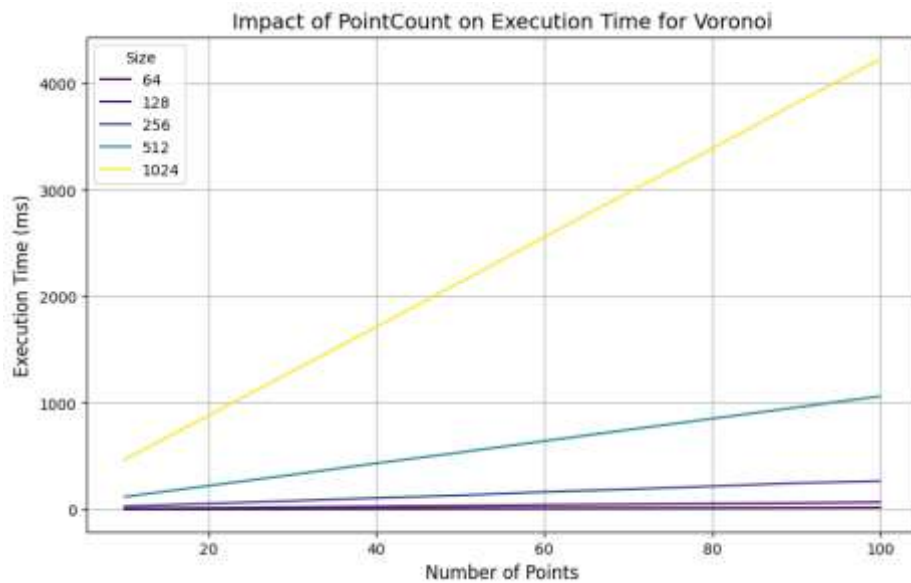


Рис. 3.23. Вплив кількості точок на час роботи генерації діаграм Вороного

Було здійснено візуалізацію впливу кількості масштабування на час роботи алгоритму DLA залежно від початкового розміру (рис. 3.24).

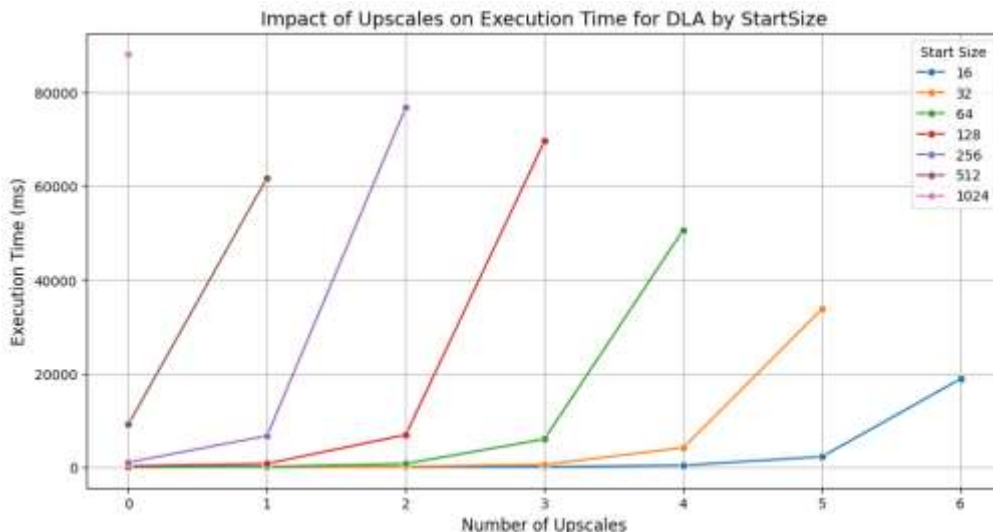


Рис. 3.24. Вплив кількості масштабування на час роботи алгоритму DLA залежно від початкового розміру

Як можна бачити, алгоритм працює найшвидше у випадку меншого початкового розміру полотна, майже з будь-якою кількістю масштабувань. Найповільнішим випадком роботи алгоритму є з початковим розміром 1024, який в даному випадку є максимальним. Це можна пояснити тим, що першим частинкам важко знайти початкову центральну на великому пустому просторі.

Також було проаналізовано дані про різні моди оновлення чанків. Вплив моду оновлення на кількість кадрів в секунду (рис. 3.25) не так сильно помітно як на кількість завантажених (рис. 3.26) та активних чанків (рис. 3.27).

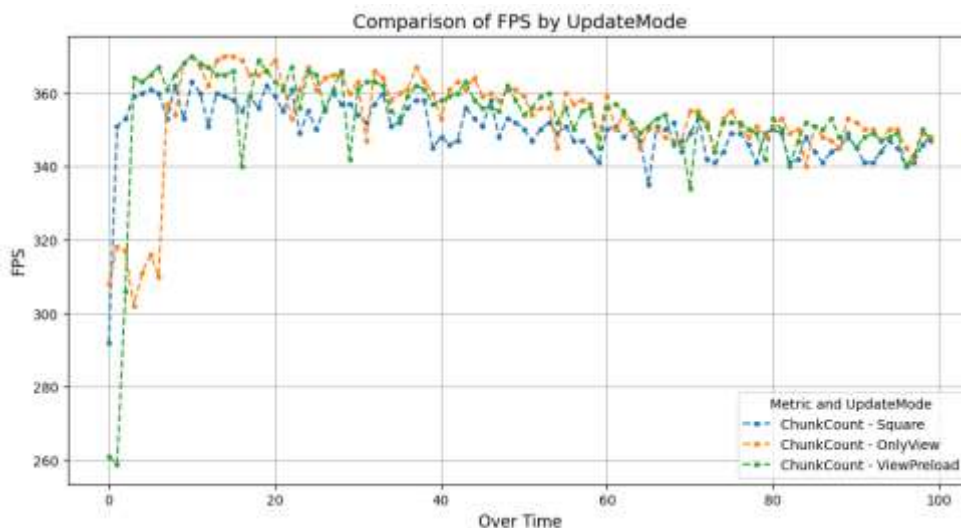


Рис. 3.25. Вплив моду оновлення чанків на кількість кадрів в секунду

Мод оновлення чанків **Square**, який завантажує та відображує всі чанки навколо гравця, призводить до трішки меншої кількості кадрів в секунду ніж два

інші моди, але різниця не відчутна. Він забезпечує стабільний FPS, але може споживати більше ресурсів.

OnlyView демонструє ефективне використання ресурсів, оновлюючи та відображаючи лише видимі чанки. Це призводить до стабільнішого FPS без різких провалів.

ViewPreload показує хорошу продуктивність, але періодичні просадки FPS можуть вказувати на додаткові витрати ресурсів під час попереднього завантаження чанків.

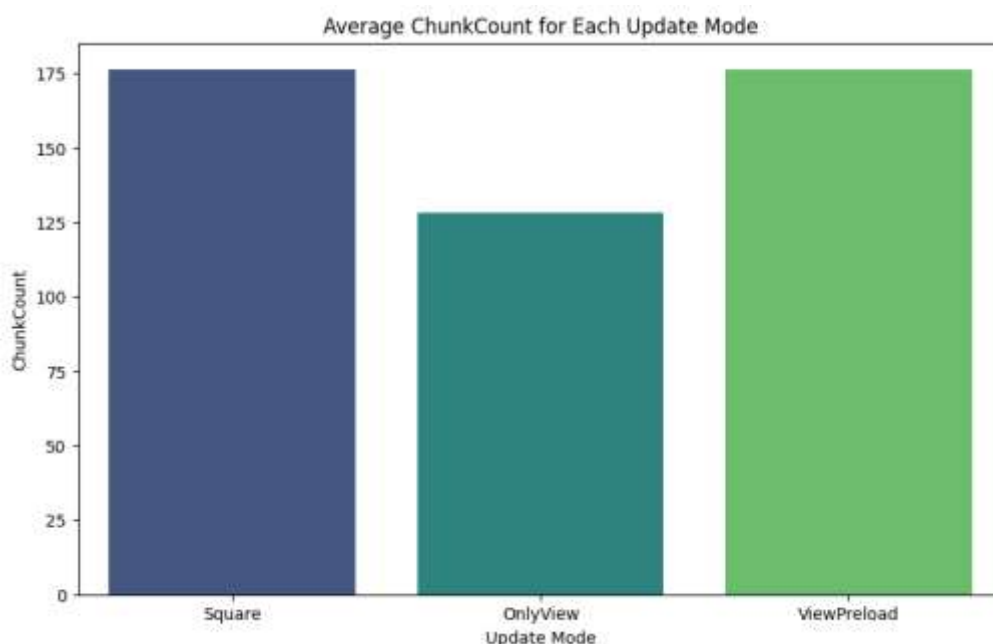


Рис. 3.26. Вплив моду оновлення чанків на кількість завантажених чанків

На рисунку 3.26 видно, що моди, які завантажують всі чанки навколо гравця, в сумі завантажують більше, ніж мод OnlyView, який завантажує лише ті, які гравець може бачити.

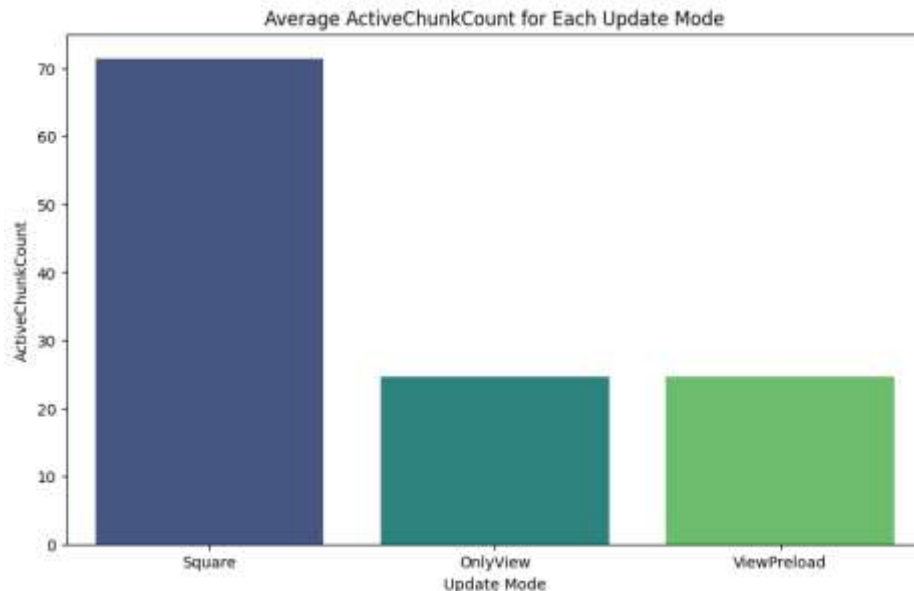


Рис. 3.27. Вплив моду оновлення чанків на кількість активних чанків

На відміну від завантажених чанків, кількість активних при моді ViewPreload дорівнює OnlyView.

Висновки до третього розділу

В результаті написання третього розділу було описано користувацький інтерфейс програмного додатку. Було наведено приклади результатів роботи алгоритмів генерації мап висот та тривимірного ландшафту. Також було описано процес тестування програмного додатку та його результати – було виявлено певний ряд дефектів та описано шляхи їх вирішення.

Було реалізовано клас, що відповідає за створення набору даних про результат генерації кожним алгоритмом з різними вхідними параметрами та інформацію про роботу додатку при різних модах оновлення чанків в нескінченній генерації тривимірного ландшафту. Ці дані потім було записано в файл .csv та завантажено у середовище Colaboratory.

За допомогою Google Colaboratory було проведено аналіз згенерованих даних та виконано візуалізацію побудованих графіків. В результаті аналізу було виявлено залежності часу роботи від кількості октав в алгоритмі Fractal Perlin Noise. Також було виявлено незначний вплив різних модів оновлення динамічної генерації ландшафту на кількість кадрів в секунду та значний на кількість завантажених чанків.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було досягнуто значних результатів у вивченні й застосуванні популярних підходів до генерації ландшафтів ігрових світів, зокрема методів, що базуються на шумових алгоритмах та імітаціях природних процесів.

В роботі було розглянуто основні існуючі методи генерації ландшафту. Серед них значна увага була приділена шуму Перліна, який дозволяють створювати реалістичні деталі поверхні. Також розглядались алгоритми фрактальної генерації, які надають можливість масштабувати ландшафт, зберігаючи деталі. Іншими важливими методами були симуляція ерозійних процесів для досягнення реалістичності форми ландшафтів та методи агрегації, такі як DLA, для створення нерівних і природних поверхонь.

Було представлено детальний опис розроблених алгоритмів, їх імплементація в Unity, а також пояснення принципів роботи методів для створення тривимірної сітки. Були побудовані блок-схеми і діаграми класів, які пояснюють основні етапи роботи алгоритмів і їх взаємодію. Також було реалізовано нескінченну генерацію ландшафту за допомогою використання чанків та рівня деталізації (LOD).

Також було детально описано користувацький інтерфейс програмного додатку та наведено приклади результатів роботи алгоритмів. Здійснено тестування, яке показало працездатність розробленого програмного додатку.

За допомогою візуалізації результатів алгоритмів з різними параметрами, було виявлено їх вплив на час роботи. Також, залежно від обраного моду оновлення чанків, можна помітити різну їх поведінку та її результат на працездатність додатку.

В результаті було отримано робочий програмний додаток, який надає користувачу всі важливі можливості редактора таких об'єктів, як мапа висот та тривимірний ландшафт.

В подальшому проект можна розвивати розширюючи інструментарій редагування мап висот та ландшафту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013
2. Galin, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M. P., Benes, B., & Gain, J. (2019, May). A review of digital terrain modeling. In *Computer Graphics Forum* (Vol. 38, No. 2, pp. 553-577).
3. Bryan Bergeron. *Developing Serious Games* (Game Development Series). Charles River Media, January 2006.
4. Tait, Emma R., and Ingrid L. Nelson. "Nonscalability and generating digital outer space natures in No Man's Sky." *Environment and Planning E: Nature and Space* 5.2 (2022): 694-718.
5. Patterson, Blake, and Michael Ward. "Adaptive Procedural Generation in Minecraft." (2022).
6. Fridh, Marcus, and Fredrik Sy. "Settlement generation in minecraft." (2020).
7. Liu, Yiming. "Research on Interactive Experience based on 3D Game Scene Design." *Transactions on Social Science, Education and Humanities Research* 1 (2023): 427-431.
8. Watkins, Ryan. *Procedural content generation for unity game development*. Packt Publishing Ltd, 2016.
9. Westerlund, Fredrik. "Evaluating the Usability of an Interactive Bridge Generator: Procedural Content Generation in Unreal Engine 4." (2022).
10. About Massive. Massive Software. URL: <https://www.massivesoftware.com/about.html> (date of access: 20.11.2024).
11. Buchholz, Henrik, et al. "Automated construction of urban terrain models." *Progress in Spatial Data Handling: 12th International Symposium on Spatial Data Handling*. Springer Berlin Heidelberg, 2006.
12. G enevaux, Jean-David, et al. "Terrain generation using procedural models based on hydrology." *ACM Transactions on Graphics (TOG)* 32.4 (2013): 1-13.

13. Grossmannová, Pavla, and Daniel Švábek. "PROCEDURAL TERRAIN GENERATION AND PATH SEARCHING IN EDUCATION."
14. Nikkola, Aapo. "Procedural Terrain Generation in VR Simulations." (2022).
15. Rose, Thomas J., and Anastasios G. Bakaoukas. "Algorithms and approaches for procedural terrain generation-a brief review of current techniques." *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*. IEEE, 2016.
16. Perlin, Ken. ["In the beginning: The Pixel Stream Editor"](#). Retrieved May 31, 2022.
17. Ken Perlin, Noise hardware. In Real-Time Shading SIGGRAPH Course Notes (2001), Olano M., (Ed.).
18. Fournier, Alain; Fussell, Don; Carpenter, Loren (June 1982). ["Computer rendering of stochastic models"](#). *Communications of the ACM*. 25 (6): 371–384. doi:[10.1145/358523.358553](#). ISSN [0001-0782](#).
19. Janert, P. (2022). The Diamond-Square Algorithm for Terrain Generation. *Janert Blog*. Retrieved November 30, 2024, from <https://janert.me/blog/2022/the-diamond-square-algorithm-for-terrain-generation/>
20. Shaker, Noor, Julian Togelius, and Mark J. Nelson. "Procedural content generation in games." (2016): 978-3.
21. Bulgakova, O. S., et al. "Algorithmic modifications in procedural generation systems." *Control Systems and Machines*. (2019).
22. David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin and Steven Worley: Texturing and Modeling: A Procedural Approach (Third Edition). Morgan Kaufmann Publishers, 2003.
23. Aurenhammer, Franz, and Rolf Klein. "Voronoi Diagrams." *Handbook of computational geometry* 5.10 (2000): 201-290.
24. Mitchell, Tom M. (1997). Machine Learning (International ed.). *McGraw-Hill*. p. [233](#). ISBN [978-0-07-042807-2](#).

25. Basic Information - Grand Canyon National Park (U.S. National Park Service). *NPS.gov Homepage* (U.S. National Park Service). URL: <https://www.nps.gov/grca/planyourvisit/basicinfo.htm> (date of access: 12.12.2024).
26. GitHub - henrikglass/erodr: An implementation of Hans Theobald Beyer's algorithm for simulated hydraulic erosion. *GitHub*. URL: <https://github.com/henrikglass/erodr> (date of access: 15.11.2024).
27. GitHub - SebLague/Hydraulic-Erosion. *GitHub*. URL: <https://github.com/SebLague/Hydraulic-Erosion> (date of access: 15.12.2024).
28. Beyer, H. T. (2015). *Implementation of a Method for Hydraulic Erosion*. Bachelor's Thesis in Informatics: Games Engineering, Technical University of Munich
29. O. Deussen and B. Lintermann. *Digital Design of Nature*. Springer, Berlin, Germany, 2005.
30. D. Ebert, K. Musgrave, P. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, San Francisco, USA, 2003
31. Halsey, Thomas C. "Diffusion-limited aggregation: A model for pattern formation." *Physics Today* 53.11 (2000): 36-41.
32. Sala, Nicoletta. "Fractals, computer science and beyond." *Complexity science, living systems, and reflexing interfaces: new models and perspectives*. IGI Global, 2013. 268-291.
33. Perlin, Ken. "Improving noise." *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002.
34. Perlin, Ken, and Eric M. Hoffert. "Hypertexture." *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. 1989.
35. Parberry, Ian. "Designer worlds: Procedural generation of infinite terrain from real-world elevation data." *Journal of Computer Graphics Techniques* 3.1 (2014).

36. Ken Perlin. Improved Noise reference implementation.
URL: <https://cs.nyu.edu/~perlin/noise/> (date of access: 19.11.2024).
37. Sanders, Andrew. *An introduction to Unreal engine 4*. AK Peters/CRC Press, 2016.
38. Buyuksalih, Ismail, et al. "3D modelling and visualization based on the unity game engine—advantages and challenges." *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 4 (2017): 161-166.
39. Holfeld, Julian. "On the relevance of the Godot Engine in the indie game development industry." *arXiv preprint arXiv:2401.01909* (2023).
40. Ahmed, Sameer, and Bipin Pandey. "Procedural Terrain Generation by Sampling a 2D Monochrom Perlin Noise Map in Unity." *Asian Journal of Research in Computer Science* 16.1 (2023): 37-42.
41. Gauffin, Heikki. "Additional Unity tool creation for game development using Unity Editor Scripting." (2022).
42. Ilett, Daniel. "Shader Graph." *Building Quality Shaders for Unity®: Using Shader Graphs and HLSL Shaders*. Berkeley, CA: Apress, 2022. 81-107.
43. Марчук, Г. В., Левківський, В. Л., Харченко, А. В., & Марчук, Д. К. (2024). Огляд і аналіз алгоритмів процедурної генерації ігрових світів. *Таврійський науковий вісник. Серія: Технічні науки*, (4), 101-110.
<https://doi.org/10.32782/tnv-tech.2024.4.9>

ДОДАТКИ

Файл DLA.cs

```

using UnityEngine;
using System.Collections.Generic;
public class DLA : MonoBehaviour{
    [SerializeField] private MapDisplay map_display;
    static GameObject cubes_parent = null;
    public const int UPSCALE_FACTOR = 2;    // Фактор збільшення розміру сітки
    static int step = 1; // how often this dla was upscaled already
    static readonly Vector2Int[] offsets = { new Vector2Int(0,1), new Vector2Int(1,0),
new Vector2Int(0,-1), new Vector2Int(-1,0) };
    static List<Pixel> pixels;
    static public Pixel mainPixel;
    static Pixel[,] grid;
    static float[] image;
    public static float[,] RunDLA(int start_size, int upscale_count) {
        step = 1;
        // parent for cubes that are used for pixels visualisation
        if(cubes_parent == null) {
            cubes_parent = GameObject.Find("cubes_parent");
            if(cubes_parent == null) {
                cubes_parent = new GameObject();
            }
        }else {
            foreach(Transform child in cubes_parent.transform) {
                GameObject.Destroy(child.gameObject);
            }
        }
        // Початкова генерація
        grid = new Pixel[start_size, start_size];
        image = new float[start_size*start_size];
        pixels = new List<Pixel>();
        // Початковий піксель в центрі
        Vector2Int start_pos = new Vector2Int(start_size / 2, start_size / 2);
        mainPixel = new Pixel(start_pos, null, Pixel.PixelType.MAIN);
        grid[start_pos.x, start_pos.y] = mainPixel;
        pixels.Add(mainPixel);
        System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
        sw.Start();
        AddPixels(PixelsPerStep(grid.GetLength(0), 0));
        UpscaleTexture();
        sw.Stop();
        // Генерація пікселів і зв'язків на кожному етапі
        for (int step = 0; step < upscale_count; step++) {
            sw.Restart();
            UpscaleGrid();
            AddPixels(PixelsPerStep(grid.GetLength(0), step+1));
            UpscaleTexture();
        }
    }
}

```

```

        sw.Stop();
    }
    int n = image.Length;
    int size = (int)Mathf.Sqrt(n);
    if (size * size != n) {
        Debug.Break();
        return new float[0,0]; // wtf is that
    }
    image = GaussBlur(image, size);
    // 1d -> 2d
    float[,] matrix = new float[size, size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i, j] = image[i * size + j];
        }
    }
    // finding min & max
    float min = int.MaxValue;
    float max = int.MinValue;
    for(int i = 0; i < matrix.GetLength(0); i++) {
        for(int j = 0; j < matrix.GetLength(1); j++) {
            if(matrix[i,j] > max) {
                max = matrix[i,j];
            }
            if(matrix[i,j] < min) {
                min = matrix[i,j];
            }
        }
    }
    // normilizing
    for(int i = 0; i < matrix.GetLength(0); i++) {
        for(int j = 0; j < matrix.GetLength(1); j++) {
            matrix[i,j] = Mathf.InverseLerp(min, max, matrix[i,j]);
        }
    }
    return matrix;
}

public static int PixelsPerStep(int size, int step) {
    return size * (step == 0 ? 1 : step);
}

static void AddPixels(int amount=1, bool on_edge=false) {
    if(amount < 1) {
        return;
    }
    Vector2Int randomPos;
    int width = grid.GetLength(0);
    int height = grid.GetLength(1);

    for(int i = 0; i < amount; i++) {

```

```

        if(on_edge) {
            // Randomly choose which edge to pick from: top, bottom, left, or
right
            int edge = Random.Range(0, 4);
            randomPos = edge switch
            {
                // Top edge
                0 => new Vector2Int(Random.Range(0, width), 0),
                // Bottom edge
                1 => new Vector2Int(Random.Range(0, width), height - 1),
                // Left edge
                2 => new Vector2Int(0, Random.Range(0, height)),
                // Right edge
                3 => new Vector2Int(width - 1, Random.Range(0, height)),
                _ => Vector2Int.zero, // Fallback
            };
        }
        else {
            randomPos = new Vector2Int(Random.Range(0, width), Random.Range(0,
height));
            while(grid[randomPos.x, randomPos.y] != null) { // change position
until grid element is null
                randomPos = new Vector2Int(Random.Range(0, width), Random.Range(0,
height));
            }
        }
        MovePixelToConnection(randomPos);
    }
}

static void MovePixelToConnection(Vector2Int pos)
{
    int size = grid.GetLength(0);
    while (grid[pos.x, pos.y] == null)
    {
        foreach(var offset in offsets) {
            if(pos.x + offset.x > 0 && pos.x + offset.x < size-1 && pos.y +
offset.y > 0 && pos.y + offset.y < size-1) {
                if(grid[pos.x + offset.x, pos.y + offset.y] != null)
                {
                    Pixel parent = grid[pos.x + offset.x, pos.y + offset.y];
                    Pixel new_pixel = new Pixel(pos, parent, Pixel.PixelType.New);
                    pixels.Add(new_pixel);
                    grid[pos.x, pos.y] = new_pixel;
                    break;
                }
            }
        }
    }

    if(grid[pos.x, pos.y] == null) {
        pos += offsets[Random.Range(0, 4)];
    }
}

```

```

        // Correct possible overshoots over the grid.
        pos.x = Mathf.Clamp(pos.x, 0, size-1);
        pos.y = Mathf.Clamp(pos.y, 0, size-1);
    }
}

static void UpscaleGrid() {
    int oldSize = grid.GetLength(0);
    int newSize = oldSize * UPSCALE_FACTOR;
    grid = new Pixel[newSize, newSize];
    CreateConnections(mainPixel);
    foreach(var pixel in pixels) {
        grid[pixel.position.x, pixel.position.y] = pixel;
    }
}

static private void CreateConnections(Pixel pixel) {
    Pixel[] children = pixel.children.ToArray();
    pixel.children.Clear();
    foreach(Pixel child in children) {
        Vector2Int connector_pos = pixel.position * UPSCALE_FACTOR +
(child.position - pixel.position);
        Pixel connector = new Pixel(connector_pos, pixel, Pixel.PixelType.Mid);
        pixels.Add(connector);
        // jiggle time !!!
        Vector2Int ortho = child.position - pixel.position;
        if(ortho.x > 1) {
            ortho.x = 1;
        }
        if(ortho.y > 1) {
            ortho.y = 1;
        }
        ortho = new Vector2Int(ortho.y, ortho.x); // swap x and y
        if(ortho.x != 0)
            ortho.x = ortho.x / Mathf.Abs(ortho.x);
        if(ortho.y != 0)
            ortho.y = ortho.y / Mathf.Abs(ortho.y);

        int ran = Random.Range(0, 11);
        if(ran >= 9) connector.position += ortho;
        else if(ran >= 7) connector.position -= ortho;
        child.parent = connector;
        connector.children.Add(child);
        CreateConnections(child);
    }
    pixel.type = Pixel.PixelType.Old;
    pixel.position *= UPSCALE_FACTOR;
}

private static float[] GaussBlur(float[] image, int image_size) {
    List<float> new_image_list = new List<float>();
    for(int x = 0; x < image_size; x++) {

```

```

        for(int y = 0; y < image_size; y++) {
            int mx = x * image_size;
            int bx = x > 0 ? (x-1) * image_size : mx;
            int ax = x < image_size-1 ? (x+1) * image_size : mx;
            int my = y;
            int by = y > 0 ? y-1 : my;
            int ay = y < image_size-1 ? y+1 : my;
            //      min              middle              max
            float bxby = image[bx + by]; float bxmy = image[bx + my]; float bxay =
image[bx + ay];
            float mxby = image[mx + by]; float mxmy = image[mx + my]; float mxay =
image[mx + ay];
            float axby = image[ax + by]; float axmy = image[ax + my]; float axay =
image[ax + ay];

            float minWeight = 4f * (Random.Range(1, 11) / 10f);
            float midWeight = 8f * (Random.Range(1, 11) / 10f);
            float maxWeight = 16f * (Random.Range(1, 11) / 10f);
            float v = 1f / (4f*minWeight + 4f*midWeight + maxWeight) * (
                minWeight*bxby + midWeight*bxmy + minWeight*bxay +
                midWeight*mxby + maxWeight*mxmy + midWeight*mxay +
                minWeight*axby + midWeight*axmy + minWeight*axay
            );
            new_image_list.Add(v);
        }
    }
    return new_image_list.ToArray();
}

static void UpscaleTexture() {
    CalculateValues();
    int image_size = grid.GetLength(0);
    // place current grid on the image
    for(int x=0; x<image_size; x++) {
        for(int y=0; y<image_size; y++) {

            float v = grid[x,y] != null ? 1f - 1f / (1f + 0.5f * grid[x,y].value) :
0f;

            int index = x * image_size + y;
            float fv = 1f - 1f / (1f + (1f / step) * v + 1.25f * image[index]);
            image[index + 0] = fv;
        }
    }
    ++step;
    // upscale the current image
    List<float> new_image_list = new List<float>();
    float multiplier = 1f / UPSCALE_FACTOR;

    for(int x = 0; x < image_size * UPSCALE_FACTOR; x++) {
        for(int y = 0; y < image_size * UPSCALE_FACTOR; y++) {
            int mx = (int)Mathf.Floor(x * multiplier) * image_size; // mx - middle
pixel.x

```

```

        int bx = x > 0 ? (int)Mathf.Floor((x - 1) * multiplier) * image_size :
mx; // bx - pixel.x on the left from middle.x one
        int my = (int)Mathf.Floor(y * multiplier); // same but with y
coordinate
        int by = y > 0 ? (int)Mathf.Floor((y - 1) * multiplier) : my;

        float v = 0.25f * image[bx + by] + 0.25f * image[bx + my] + 0.25f *
image[mx + by] + 0.25f * image[mx + my];
        new_image_list.Add(v);
    }
}
image = new_image_list.ToArray();
new_image_list.Clear();

image_size *= UPSCALE_FACTOR;
image = GaussBlur(image, image_size);
}
static private void CalculateValues() {
    foreach(var pixel in pixels) {
        if(pixel.children.Count == 0) {
            int value = 1;
            pixel.type = Pixel.PixelType.Last;
            pixel.value = value;
            Pixel parent = pixel.parent;
            while(parent != null) {
                value++;
                parent.value = value;
                parent = parent.parent;
            }
        }
    }
}
}
public class Pixel{
    public enum PixelType{
        None ,New, Mid, Old, Last, MAIN
    }
    public PixelType type; // for visualisation
    public Vector2Int position;
    public Pixel parent; // this pixel is connected to parent
    public List<Pixel> children; // pixels connected to this one
    public float value;
    public Pixel(Vector2Int position, Pixel parent, PixelType type) {
        this.position = position;
        this.type = type;

        this.parent = parent;
        children = new List<Pixel>();
        parent?.children.Add(this);
    }
}
}
}

```

Файл Erosion.cs:

```

using UnityEngine;
public class Erosion : MonoBehaviour {
    public int seed;
    [Range (2, 8)]
    public int erosionRadius = 3;
    [Range (0, 1)]
    public float inertia = .05f; // At zero, water will instantly change direction to
    flow downhill. At 1, water will never change direction.
    public float sedimentCapacityFactor = 4; // Multiplier for how much sediment a
    droplet can carry
    public float minSedimentCapacity = .01f; // Used to prevent carry capacity getting
    too close to zero on flatter terrain
    [Range (0, 1)]
    public float erodeSpeed = .3f;
    [Range (0, 1)]
    public float depositSpeed = .3f;
    [Range (0, 1)]
    public float evaporateSpeed = .01f;
    public float gravity = 4;
    public int maxDropletLifetime = 30;
    public float initialWaterVolume = 1;
    public float initialSpeed = 1;
    // Indices and weights of erosion brush precomputed for every node
    int[][] erosionBrushIndices;
    float[][] erosionBrushWeights;
    System.Random prng;
    int currentSeed;
    int currentErosionRadius;
    int currentMapSize;
    // Initialization creates a System.Random object and precomputes indices and
    weights of erosion brush
    void Initialize (int mapSize, bool resetSeed) {
        if (resetSeed || prng == null || currentSeed != seed) {
            prng = new System.Random (seed);
            currentSeed = seed;
        }
        if (erosionBrushIndices == null || currentErosionRadius != erosionRadius ||
        currentMapSize != mapSize) {
            InitializeBrushIndices (mapSize, erosionRadius);
            currentErosionRadius = erosionRadius;
            currentMapSize = mapSize;
        }
    }
    public void Erode (float[] map, int mapSize, int numIterations = 1, bool resetSeed
    = false) {
        Initialize (mapSize, resetSeed);
        for (int iteration = 0; iteration < numIterations; iteration++) {
            // Create water droplet at random point on map

```

```

float posX = prng.Next(0, mapSize - 1);
float posY = prng.Next(0, mapSize - 1);
float dirX = 0;
float dirY = 0;
float speed = initialSpeed;
float water = initialWaterVolume;
float sediment = 0;
for (int lifetime = 0; lifetime < maxDropletLifetime; lifetime++) {
    int nodeX = (int) posX;
    int nodeY = (int) posY;
    int dropletIndex = nodeY * mapSize + nodeX;
    // Calculate droplet's offset inside the cell (0,0) = at NW node, (1,1)
= at SE node
    float cellOffsetX = posX - nodeX;
    float cellOffsetY = posY - nodeY;
    // Calculate droplet's height and direction of flow with bilinear
interpolation of surrounding heights
    HeightAndGradient heightAndGradient = CalculateHeightAndGradient (map,
mapSize, posX, posY);
    // Update the droplet's direction and position (move position 1 unit
regardless of speed)
    dirX = (dirX * inertia - heightAndGradient.gradientX * (1 - inertia));
    dirY = (dirY * inertia - heightAndGradient.gradientY * (1 - inertia));
    // Normalize direction
    float len = Mathf.Sqrt (dirX * dirX + dirY * dirY);
    if (len != 0) {
        dirX /= len;
        dirY /= len;
    }
    posX += dirX;
    posY += dirY;
    // Stop simulating droplet if it's not moving or has flowed over edge
of map
    if ((dirX == 0 && dirY == 0) || posX < 0 || posX >= mapSize - 1 || posY
< 0 || posY >= mapSize - 1) {
        break;
    }
    // Find the droplet's new height and calculate the deltaHeight
    float newHeight = CalculateHeightAndGradient (map, mapSize, posX,
posY).height;
    float deltaHeight = newHeight - heightAndGradient.height;
    // Calculate the droplet's sediment capacity (higher when moving fast
down a slope and contains lots of water)
    float sedimentCapacity = Mathf.Max (-deltaHeight * speed * water *
sedimentCapacityFactor, minSedimentCapacity);
    // If carrying more sediment than capacity, or if flowing uphill:
    if (sediment > sedimentCapacity || deltaHeight > 0) {
        // If moving uphill (deltaHeight > 0) try fill up to the current
height, otherwise deposit a fraction of the excess sediment
        float amountToDeposit = (deltaHeight > 0) ? Mathf.Min (deltaHeight,
sediment) : (sediment - sedimentCapacity) * depositSpeed;

```

```

        sediment -= amountToDeposit;
        // Add the sediment to the four nodes of the current cell using
        bilinear interpolation
        // Deposition is not distributed over a radius (like erosion) so
        that it can fill small pits
        map[dropletIndex] += amountToDeposit * (1 - cellOffsetX) * (1 -
        cellOffsetY);
        map[dropletIndex + 1] += amountToDeposit * cellOffsetX * (1 -
        cellOffsetY);
        map[dropletIndex + mapSize] += amountToDeposit * (1 - cellOffsetX)
        * cellOffsetY;
        map[dropletIndex + mapSize + 1] += amountToDeposit * cellOffsetX *
        cellOffsetY;
    } else {
        // Erode a fraction of the droplet's current carry capacity.
        // Clamp the erosion to the change in height so that it doesn't dig
        a hole in the terrain behind the droplet
        float amountToErode = Mathf.Min ((sedimentCapacity - sediment) *
        erodeSpeed, -deltaHeight);
        // Use erosion brush to erode from all nodes inside the droplet's
        erosion radius
        for (int brushPointIndex = 0; brushPointIndex <
        erosionBrushIndices[dropletIndex].Length; brushPointIndex++) {
            int nodeIndex =
            erosionBrushIndices[dropletIndex][brushPointIndex];
            float weighedErodeAmount = amountToErode *
            erosionBrushWeights[dropletIndex][brushPointIndex];
            float deltaSediment = (map[nodeIndex] < weighedErodeAmount) ?
            map[nodeIndex] : weighedErodeAmount;
            map[nodeIndex] -= deltaSediment;
            sediment += deltaSediment;
        }
        // Update droplet's speed and water content
        speed = Mathf.Sqrt (speed * speed + deltaHeight * gravity);
        water *= (1 - evaporateSpeed);
    }
}

HeightAndGradient CalculateHeightAndGradient(float[] nodes, int mapSize, float
posX, float posY) {
    int coordX = (int)posX;
    int coordY = (int)posY;
    // Calculate droplet's offset inside the cell (0,0) = at NW node, (1,1) = at SE
    node
    float x = posX - coordX;
    float y = posY - coordY;
    // Calculate heights of the four nodes of the droplet's cell
    int nodeIndexNW = coordY * mapSize + coordX;
    float heightNW = nodes[nodeIndexNW];
    float heightNE = nodes[nodeIndexNW + 1];

```

```

        float heightSW = nodes[nodeIndexNW + mapSize];
        float heightSE = nodes[nodeIndexNW + mapSize + 1];
        // Calculate droplet's direction of flow with bilinear interpolation of height
        difference along the edges
        float gradientX = (heightNE - heightNW) * (1 - y) + (heightSE - heightSW) * y;
        float gradientY = (heightSW - heightNW) * (1 - x) + (heightSE - heightNE) * x;
        // Calculate height with bilinear interpolation of the heights of the nodes of
        the cell
        float height = heightNW * (1 - x) * (1 - y) + heightNE * x * (1 - y) + heightSW
        * (1 - x) * y + heightSE * x * y;
        return new HeightAndGradient() { height = height, gradientX = gradientX,
        gradientY = gradientY };
    }

    void InitializeBrushIndices (int mapSize, int radius) {
        erosionBrushIndices = new int[mapSize * mapSize][];
        erosionBrushWeights = new float[mapSize * mapSize][];
        int[] xOffsets = new int[radius * radius * 4];
        int[] yOffsets = new int[radius * radius * 4];
        float[] weights = new float[radius * radius * 4];
        float weightSum = 0;
        int addIndex = 0;
        for (int i = 0; i < erosionBrushIndices.GetLength(0); i++) {
            int centreX = i % mapSize;
            int centreY = i / mapSize;
            if (centreY <= radius || centreY >= mapSize - radius || centreX <= radius +
1 || centreX >= mapSize - radius) {
                weightSum = 0;
                addIndex = 0;
                for (int y = -radius; y <= radius; y++) {
                    for (int x = -radius; x <= radius; x++) {
                        float sqrDst = x * x + y * y;
                        if (sqrDst < radius * radius) {
                            int coordX = centreX + x;
                            int coordY = centreY + y;
                            if (coordX >= 0 && coordX < mapSize && coordY >= 0 &&
coordY < mapSize) {
                                float weight = 1 - Mathf.Sqrt(sqrDst) / radius;
                                weightSum += weight;
                                weights[addIndex] = weight;
                                xOffsets[addIndex] = x;
                                yOffsets[addIndex] = y;
                                addIndex++;
                            }
                        }
                    }
                }
            }
        }
        int numEntries = addIndex;
        erosionBrushIndices[i] = new int[numEntries];
        erosionBrushWeights[i] = new float[numEntries];
        for (int j = 0; j < numEntries; j++) {

```

```

        erosionBrushIndices[i][j] = (yOffsets[j] + centreY) * mapSize +
xOffsets[j] + centreX;
        erosionBrushWeights[i][j] = weights[j] / weightSum;
    }
}
}
struct HeightAndGradient {
    public float height;
    public float gradientX;
    public float gradientY;
}
}

```

Файл MeshGenerator.cs:

```

using System.Runtime.ConstrainedExecution;
using UnityEngine;
public static class MeshGenerator {

    public static MeshData GenerateTerrainMesh(float[,] heightMap, float
height_multiplier, AnimationCurve _height_curve, int level_of_detail, bool
useFlatShading) {
        // height_curve.Evaluate returns weird results when is used from different
threads at the same time
        // so i create new one
        AnimationCurve height_curve = new AnimationCurve(_height_curve.keys);
        int meshSimplificationIncrement = (level_of_detail == 0)? 1: level_of_detail*2;
        int borderedSize = heightMap.GetLength (0);
        int meshSize = borderedSize - 2 * meshSimplificationIncrement;
        int meshSizeUnsimplified = borderedSize - 2;
        float topLeftX = (meshSizeUnsimplified - 1) / -2f; // makes posititon not
dependent on LOD
        float topLeftZ = (meshSizeUnsimplified - 1) / 2f;
        int verticesPerLine = (meshSize-1) / meshSimplificationIncrement + 1;
        MeshData meshData = new MeshData (verticesPerLine, useFlatShading);
        int[,] vertexIndicesMap = new int[borderedSize, borderedSize];
        int meshVertexIndex = 0;
        int borderVertexIndex = -1;
        for (int y = 0; y < borderedSize; y += meshSimplificationIncrement) {
            for (int x = 0; x < borderedSize; x += meshSimplificationIncrement) {
                bool is_border_vertex = y == 0 || y == borderedSize-1 || x == 0 || x ==
borderedSize-1;
                if(is_border_vertex) {
                    vertexIndicesMap[x,y] = borderVertexIndex;
                    borderVertexIndex--;
                }else {
                    vertexIndicesMap[x,y] = meshVertexIndex;
                    meshVertexIndex++;
                }
            }
        }
        for (int y = 0; y < borderedSize; y += meshSimplificationIncrement) {
            for (int x = 0; x < borderedSize; x += meshSimplificationIncrement) {
                int vertexIndex = vertexIndicesMap[x,y];
                // віднімаємо meshSimplificationIncrement, тому що при 0 воно визначає
кордони, borders, а меш починається при 0+meshSimplificationIncrement
                Vector2 percent = new Vector2 ((x-meshSimplificationIncrement) /
(float)meshSize, (y-meshSimplificationIncrement) / (float)meshSize);
                float height = height_curve.Evaluate(heightMap[x,
y])*height_multiplier;
                Vector3 vertex_position = new Vector3 (topLeftX +
percent.x*meshSizeUnsimplified, height, topLeftZ - percent.y*meshSizeUnsimplified);

```

```

        meshData.AddVertex(vertex_position, percent, vertexIndex);
        if (x < borderedSize - 1 && y < borderedSize - 1) {
            int a = vertexIndicesMap[x,y];
            int b = vertexIndicesMap[x+meshSimplificationIncrement, y];
            int c = vertexIndicesMap[x, y+meshSimplificationIncrement];
            int d = vertexIndicesMap[x+meshSimplificationIncrement,
y+meshSimplificationIncrement];
            meshData.AddTriangle (a,d,c);
            meshData.AddTriangle (d,a,b);
        }
    }
    meshData.Finalise();
    return meshData;
}
}

public class MeshData {
    Vector3[] vertices;
    int[] triangles;
    Vector2[] uvs;
    Vector3[] baked_normals;
    Vector3[] border_vertices;
    int[] border_triangles;
    int triangleIndex;
    int borderTrianleIndex;
    bool use_flat_shading;
    public MeshData(int vertices_per_line, bool use_flat_shading) {
        this.use_flat_shading = use_flat_shading;
        vertices = new Vector3[vertices_per_line * vertices_per_line];
        uvs = new Vector2[vertices_per_line * vertices_per_line];
        triangles = new int[(vertices_per_line-1)*(vertices_per_line-1)*6];
        border_vertices = new Vector3[vertices_per_line*4+4]; // 4 lines + edges
        border_triangles = new int[24*vertices_per_line];
    }
    public void AddVertex(Vector3 vertex_position, Vector2 uv, int vertex_index) {
        if(vertex_index < 0) {
            border_vertices[-vertex_index-1] = vertex_position;
        }else {
            vertices[vertex_index] = vertex_position;
            uvs[vertex_index] = uv;
        }
    }
    public void AddTriangle(int a, int b, int c) {
        if(a < 0 || b < 0 || c < 0) {
            // then it triangle belonging to the border
            border_triangles[borderTrianleIndex] = a;
            border_triangles[borderTrianleIndex + 1] = b;
            border_triangles[borderTrianleIndex + 2] = c;
            borderTrianleIndex += 3;
        }else {
            triangles[triangleIndex] = a;

```

```

        triangles[triangleIndex + 1] = b;
        triangles[triangleIndex + 2] = c;
        triangleIndex += 3;
    }
}
// so the normal are consistant on the edges of chunks
Vector3[] CalculateNormals() {
    Vector3[] vertex_normals = new Vector3[vertices.Length];
    int traingle_count = triangles.Length/3;
    for(int i = 0; i < traingle_count; i++) {
        int normal_triangle_id = i*3;
        int vertex_id_A = triangles[normal_triangle_id];
        int vertex_id_B = triangles[normal_triangle_id+1];
        int vertex_id_C = triangles[normal_triangle_id+2];
        Vector3 triangle_normal = SurfaceNormalFromIndices(vertex_id_A,
vertex_id_B, vertex_id_C);
        vertex_normals[vertex_id_A] += triangle_normal;
        vertex_normals[vertex_id_B] += triangle_normal;
        vertex_normals[vertex_id_C] += triangle_normal;
    }
    int border_triangle_count = border_triangles.Length/3;
    for(int i = 0; i < border_triangle_count; i++) {
        int normal_triangle_id = i*3;
        int vertex_id_A = border_triangles[normal_triangle_id];
        int vertex_id_B = border_triangles[normal_triangle_id+1];
        int vertex_id_C = border_triangles[normal_triangle_id+2];
        Vector3 triangle_normal = SurfaceNormalFromIndices(vertex_id_A,
vertex_id_B, vertex_id_C);
        if(vertex_id_A >= 0) {
            vertex_normals[vertex_id_A] += triangle_normal;
        }
        if(vertex_id_B >= 0) {
            vertex_normals[vertex_id_B] += triangle_normal;
        }
        if(vertex_id_C >= 0) {
            vertex_normals[vertex_id_C] += triangle_normal;
        }
    }
    for(int i = 0; i < vertex_normals.Length; i++) {
        vertex_normals[i].Normalize();
    }
    return vertex_normals;
}
private Vector3 SurfaceNormalFromIndices(int id_A, int id_B, int id_C) {

    Vector3 point_A = (id_A < 0) ? border_vertices[-id_A-1] : vertices[id_A];
    Vector3 point_B = (id_B < 0) ? border_vertices[-id_B-1] : vertices[id_B];
    Vector3 point_C = (id_C < 0) ? border_vertices[-id_C-1] :
vertices[id_C];
    Vector3 side_AB = point_B - point_A;
    Vector3 side_AC = point_C - point_A;

```

```

        return Vector3.Cross(side_AB, side_AC);
    }
    public void Finalise() {
        if(use_flat_shading) {
            FlateShaded();
        }else {
            BakeNormals();
        }
    }
    private void BakeNormals() {
        baked_normals = CalculateNormals();
    }
    private void FlateShaded() {
        Vector3[] flat_shaded_vertices = new Vector3[triangles.Length];
        Vector2[] flat_shaded_uvs = new Vector2[triangles.Length];
        for(int i = 0 ; i < triangles.Length; i++) {
            flat_shaded_vertices[i] = vertices[triangles[i]];
            flat_shaded_uvs[i] = uvs[triangles[i]];
            triangles[i] = i;
        }
        vertices = flat_shaded_vertices;
        uvs = flat_shaded_uvs;
    }
    public Mesh CreateMesh(bool big_boy=false) {
        Mesh mesh = new Mesh ();
        if(big_boy) {
            mesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32;
        }
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.uv = uvs;
        if(use_flat_shading) {
            mesh.RecalculateNormals();
        }else {
            mesh.normals = baked_normals;
        }
        return mesh;
    }
}

```