ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ОЛЕСЯ ГОНЧАРА

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

Кваліфікаційна робота

перший (бакалаврський) рівень вищої освіти

спеціальність 113 Прикладна математика

освітня програма: Комп’ютерне моделювання та технології програмування

РОЗРОБКА СИСТЕМИ ГЕНЕРАЦІЇ СКЛАДНИХ ПОВЕРХОНЬ З ВИКОРИСТАННЯМ ПСЕВДОВИПАДКОВИХ ШУМІВ

Виконавець

студент групи ПА-19-2

\_Ільяшенко Єгор Віталійович\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(П.І.Б.) (підпис)

Керівник

доц., канд. фіз.-мат. наук

\_Степанова  Наталія Іванівна\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(П.І.Б.) (підпис)

Т.в.о. завідувача кафедри

комп’ютерних технологій

канд. фіз.-мат. наук, доц,

Зайцев В.Г . \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

2023 р.

**Дніпровський національний університет імені Олеся Гончара**

Факультет прикладної математики

Кафедра комп’ютерних технологій

Рівень перший (бакалаврський)

Спеціальність 113 Прикладна математика

Освітня програма Комп’ютерне моделювання та технології програмування

**ЗАТВЕРДЖУЮ**

**В.о. завідувача кафедри**

комп’ютерних технологій

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Зайцев В.Г.

**(підпис) (П.І.Б.)**

« 30 » березня 2023 року

**З А В Д А Н Н Я**

**НА КВАЛІФІКАЦІЙУ РОБОТУ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ільяшенко Єгор Віталійович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я, по батькові)

1. Тема роботи \_розробка системи генерації складних поверхонь з\_\_\_\_\_\_\_\_\_\_ використанням псевдовипадкових шумів\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Керівник роботи \_Степанова Наталія Іванівна\_\_\_ доц., канд. фіз.-мат. наук,

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по Університету від « 29 » березня 2023 року № 248с

2. Строк подання роботи 06 червня 2023 року

3. Вхідні дані до роботи \_Платформа Unity3D версії 2022.3, IDE Visual Studio для маніпулювання скриптами, Universal render pipeline.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)\_\_розробка системи хешування тривимірної координати, тривимірного шуму Перліна, системи генерації сферичного ландшафту за допомогою шуму Перліна, розробка системи оптимізації ландшафту за допомогою структури даних Quad-tree \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

21 зображення, що включають демонстрацію роботи програмного застосунку; зображень, які демонструють принцип роботи методів та вигляд інтерфейсу взаємодії з програмою.

6. Консультанти розділів роботи

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада  консультанта | Підпис, дата | |
| завдання видав | завдання  прийняв |
| Всі розділи | Степанова Н.І.,  доц., канд. фіз.-мат. наук | 30.03.23 | 08.06.23 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

7. Дата видачі завдання 30 березня 2023 р.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів роботи | Строк виконання етапів роботи | Примітка |
| 1. | Аналітичний огляд літературних джерел | 30.03.23 - 15.04.23 | Виконано |
| 2. | Реалізація тривимірного варіанту шуму Перліна | 16.04.23 – 22.04.23 | Виконано |
| 3. | Огляд існуючих способів та побудова оптимального алгоритму розподілення точок на сфері | 23.04.23 – 01.05.23 | Виконано |
| 4. | Оптимізація алгоритму генерації точок по шуму Перліна для одночасного пошуку багатьох точок за один перегляд | 02.05.23 – 08.05.23 | Виконано |
| 5. | Огляд та реалізація структури даних Quad-tree | 09.05.23 – 12.05.23 | Виконано |
| 6. | Побудова алгоритму генерації карти висот, використовуючи шум Перліна | 13.05.23 – 18.05.23 | Виконано |
| 7. | Визначення критеріїв для поділу поверхні та застосування Quad-tree алгоритму на планетарній сфері | 19.05.23 – 22.05.23 | Виконано |
| 8. | Розробка та застосування фрагментного шейдеру для розмалювання поверхні планети | 23.05.23 – 27.05.23 | Виконано |
| 9. | Складання та оформлення звіту | 28.05.23 – 08.06.23 | Виконано |

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_ (П.І.Б.)

( підпис )

Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_ (П.І.Б.) ( підпис )

**РЕФЕРАТ**

*Кваліфікаційна робота складається з* 39 с., 22 рис., 3 додатків.

*Об'єктом дослідження даної роботи є* система генерації складних поверхонь.

*Мета роботи*: розробити математичну модель, алгоритм побудови системи генерації складних поверхонь.

*Методика дослідження:* хешування віртуального простору, створення алгоритму генерації поверхні, створення та застосування алгоритмів оптимізації поверхі.

*Одержані висновки та їх новизна*: За результатами проведеного аналізу існуючих рішень, було побудовано систему генерації поверхні планети. За рішення генерації псевдовипадкових значень було вибрано шум Перліна та побудовано тривимірного варіанту шуму. Для додавання можливості генерації високої деталізації було обрано структуру Quad-tree, що дозволило генерувати високу деталізацію поряд із камерою у той час, як віддалені ділянки залишалися з малою деталізацією.

*Результати досліджень можуть бути застосовані при* проектуванні ландшафту при виготовленні ігор, які потребуюсь планетарного ландшафту, при побудові симуляцій, або зйомки фільмів.

*Перелік ключових слів:* ШУМ ПЕРЛІНА, QUAD-TREE, ЛАНДШАФТ, ПЛАНЕТА, ПСЕВДОВИПАДКОВІСТЬ, ПОВЕРХНЯ СФЕРИ, КАРТА ВИСОТ, UNITY.

ANNOTATION

The graduation research of the 4th year student Ilyashenko Egor

(DNU, Applied Math. Department of Computer Technology) is devoted to the development of a system for generating planetary meshes using pseudo-random noises. The objective of this research was to create an efficient method for generating realistic planetary surfaces by leveraging the power of pseudo-random noises and mesh generation techniques. The proposed system provides a framework for generating planetary terrains, enabling the creation of virtual environments for various applications. The work is interesting for game and simulation developers, filmmakers, 3d graphics engineers. The system was optimized for using at least mesh quality as possible by reducing quality of height map at further places, while maintaining maximum details near camera.

Bibliography - 8, pictures - 22, supplement - 3.

# **ЗМІСТ**

[ЗМІСТ 6](#_Toc137149570)

[ВСТУП 8](#_Toc137149571)

[ПОСТАНОВКА ЗАДАЧІ 12](#_Toc137149572)

[1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ 13](#_Toc137149573)

[1.1 Псевдо-випадкові послідовності 13](#_Toc137149574)

[1.2 Хеш функції 14](#_Toc137149575)

[1.3 Псевдовипадкові шуми 15](#_Toc137149576)

[1.4 Шум Перліна 16](#_Toc137149577)

[1.5 Шум Симплекс 17](#_Toc137149578)

[1.6 Шум Вороний 19](#_Toc137149579)

[1.7 Побудова сфери 20](#_Toc137149580)

[1.8 Quad-tree структура 22](#_Toc137149581)

[2 МАТЕМАТИЧНА МОДЕЛЬ 24](#_Toc137149582)

[2.1 Шум Перліна 24](#_Toc137149583)

[2.2 Кубо-сфера 25](#_Toc137149584)

[3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 27](#_Toc137149585)

[3.1 Платформа Unity 27](#_Toc137149586)

[3.2 Render pipeline 28](#_Toc137149587)

[3.3 Shader graph 29](#_Toc137149588)

[4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 31](#_Toc137149589)

[ВИСНОВКИ 37](#_Toc137149590)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 38](#_Toc137149591)

[ДОДАТКИ 39](#_Toc137149592)

[Додаток А 39](#_Toc137149593)

[Додаток Б 43](#_Toc137149594)

[Додаток В 47](#_Toc137149595)

# ВСТУП

У сучасному світі тривимірна комп'ютерна графіка використовується в багатьох сферах життя. Прикладами можуть стати:

* Ігрова індустрія;
* Виготовлення фільмів;
* Віртуальна реальність;
* Різноманітні симуляції.

Одним із ключових завдань тривимірного графічного дизайну - це створення реалістичних і деталізованих поверхонь. Поверхні бувають як у 3Д моделей, які складаються з набору тривимірних координат і з'єднань між ними, так і поверхня ландшафту. Такі поверхні найчастіше складаються з двовимірного масиву значень, які відображають висоту ландшафту в координаті, в якій це значення записано.

Класичний спосіб створення ландшафтної поверхні - це вибрати розмірність двовимірного масиву і заповнити всі значення власноруч. Зараз існує безліч інструментів, які спрощують процес створення таких карт висот. Вони зводять процес заповнення значень до заповнення наборами за допомогою пензлів. Такі карти висот потім зберігаються в пам'яті і можуть бути використаними в будь-який момент, шляхом завантаження цих значень із файлу і переведення значень у висоти на площині.

Але такий підхід має безліч недоліків. Кожну поверхню, яку ми захочемо використати, спочатку потрібно намалювати вручну, що вимагає дуже багато роботи, якщо ми хочемо отримати хоч якусь варіативність ландшафту. Більш того, алгоритм зчитування таких даних має квадратичну складність, бо поверхня має дві осі координат, кількість точок по кожній з яких - підвищується при збільшенні деталізації. За малої деталізації поверхні, або малого її розміру - це не відіграє великої ролі, але якщо ми хочемо створити хоч скількись правдоподібний ландшафт, або більше того, захопити справді велику ділянку - це стає великою проблемою. Створення таких поверхонь стає дуже ресурсовитратним або зовсім неможливим.

Наприклад, для створення поверхні планети, яку можна порівняти із Землею, з точністю хоча б одна точка висоти на квадратний кілометр поверхні - знадобиться 510065700 точок висот, які займуть 11.4 Гб пам'яті, з розрахунку, що одна точка містить 3 координати, кожна з яких містить 8 байт інформації.

Альтернативний варіант створення ландшафту, який розв'язує всі перераховані вище проблеми - це процедурна генерація. Це набагато складніший процес, ніж намалювати статичну карту висот, і вимагає багато роботи і планування для того, щоб отримати хоч якийсь прийнятний результат. Але в підсумку ми отримаємо систему генерації ландшафту - яку можна налаштувати за бажанням і яка має дуже багато переваг перед класичним варіантом побудови поверхонь, серед яких найосновніші - це:

* Замість зберігання всіх точок поверхні - ми зберігаємо одне значення розміром у кілька байт - seed планети;
* Можливість генерації ландшафту будь-якої складності з приголомшливою деталізацією.;
* Можливість генерувати світ на льоту для того, щоб отримати зовсім інший ландшафт - досить поміняти seed;
* Можливість генерації близько-нескінченних світів.

Після того, як ми отримали seed планети, ми можемо почати генерувати карту висот у ділянці, яка має відображатися на екрані. Але, просто генерувати псевдовипадкові числа - недостатньо, нам потрібен спосіб інтерполяції висот. Для досягнення цих цілей доцільно використовувати псевдовипадкові шуми.

Псевдовипадковий шум - це послідовність чисел, які здаються випадковими, але насправді повністю детерміновані. Такі шуми беруть псевдовипадкові значення в деяких точках площини, решта ж значень заповнюються за алгоритмом. Є багато різних алгоритмів побудови псевдовипадкового шуму, найвідоміші з яких - це шум Перлина, шум Вороного, Simplex і OpenSimplex шум.

Псевдовипадкові шуми і маніпуляції над ними - це один із найважливіших аспектів розробки системи генерації складних поверхонь. Дуже важливо визначити оптимальний підхід обробки псевдовипадкового шуму. Існує дуже багато методів фільтрації, модифікації та об'єднання шумів, що дає змогу досягти необхідної якості та деталізації згенерованих поверхонь. Під час вибору такого підходу потрібно розглядати такі критерії, як швидкодія, реалістичність, ефективність використання ресурсів і можливість регулювання параметрів, для досягнення бажаного вигляду поверхні.

Але відображення великої площини за великої деталізації все одно використовуватиме багато ресурсів пристрою, тож деталізацію потрібно зменшувати в місцях, де вона не доцільна. Наприклад, якщо камера користувача перебуває на поверхні планети, то в найближчому оточенні користувач має бачити карту висот у максимальній деталізації, тоді як на горизонті достатньо використовувати меншу кількість точок навіть без можливості користувача помітити цю заміну. Генерувати точки зі зворотного боку планети і зовсім не обов'язково. Для реалізації подібної системи деталізації можна використовувати Quad-tree структуру, де площина може містити або свою карту висот, або 4 карти висот такої ж роздільної здатності, що дає змогу рекурсивно ділити площину в місцях, де це необхідно для коректного відображення поверхні. Максимальну кількість поділів можна обмежити поряд із відстанями від камери, на яких поверхні мають ділитися.

Ця система так само важлива, як і псевдовипадкові шуми, тому що вона дає змогу уникнути використання великої кількості ресурсів пристрою і додає можливість відобразити поверхню від горизонту до горизонту, де за правильного налаштування поділок, користувач не буде здатний помітити заміну деталізації, що додасть реалістичності ландшафту.

Метою даної роботи є розробка та реалізація ефективної системи генерації складних поверхонь з використанням псевдовипадкових шумів, яка забезпечує високу деталізацію та реалістичність результатів. У рамках цієї роботи будуть вивчені існуючі методи генерації псевдовипадкових шумів, проведено їх порівняльний аналіз та вибрано найбільш оптимальні підходи для розробки системи.

Основними завданнями цієї роботи будуть:

* Вивчення теоретичних основ та існуючих підходів до генерації складних поверхонь з використанням псевдовипадкових шумів.
* Розробка та реалізація системи генерації складних поверхонь з використанням псевдовипадкових шумів.
* Результати даної роботи можуть мати практичне застосування в галузі комп'ютерної графіки, допомагаючи створювати більш реалістичні та деталізовані графічні об'єкти. Крім того, отримані в рамках роботи знання та розроблені методи можуть бути використані для подальшого вдосконалення систем генерації поверхонь та створення ще більш вражаючих візуальних ефектів.

Да-----------------------------------------------

Кваліфікаційна робота складається з таких частин:

* ВСТУП, який обґрунтовує актуальність роботи; визначає цілі проведення наукового дослідження; галузь дослідження; методи дослідження або розрахунків;
* ПОСТАНОВКА ЗАДАЧІ;
* АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ;
* МАТЕМАТИЧНА МОДЕЛЬ;
* ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ;
* АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ;
* ВИСНОВКИ;
* СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ;
* ДОДАТКИ.

# ПОСТАНОВКА ЗАДАЧІ

Метою даної роботи є ознайомлення з принципами псевдовипадкових шумів, методів генерації поверхонь ландшафту, програмна реалізація системи процедурної генерації складних поверхонь з використанням псевдовипадкових шумів.

Для виконання поставлених задач необхідно:

1. Дослідити та провести аналіз предметної області, а саме: проаналізувати принципи функціонування псевдовипадкових шумів, систем генерації ландшафту;
2. Побудувати математичну модель системи процедурної генерації псевдовипадкових поверхонь;
3. Визначити функціональні вимоги до програмного забезпечення, обрати мову програмування та платформу для програмної реалізації;
4. Спроектувати та розробити систему процедурної генерації ландшафту;
5. Проаналізувати отримані результати;
6. Зробити висновки, визначити способи подальшого поліпшення програмної реалізації.

# 1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

## 1.1 Псевдо-випадкові послідовності

Псевдо-випадкові послідовності є послідовностями чисел, які здавалося би мають випадковий характер, але фактично генеруються за допомогою алгоритмів псевдовипадкових генераторів. Ці алгоритми базуються на математичних формулах та початкових значеннях, відомих як seed або насіння.

Використовуючи однакове насіння, псевдо-випадковий генератор завжди буде генерувати однакову послідовність чисел. Однак, якщо змінити насіння, буде отримана інша послідовність чисел. Це дозволяє контролювати генерацію псевдо-випадкових послідовностей і забезпечує відтворюваність результатів.

Псевдо-випадкові послідовності широко застосовуються у різних галузях. У криптографії, вони використовуються для генерації випадкових ключів і векторів ініціалізації, що забезпечує безпеку і криптографічну стійкість. В моделюванні випадкових явищ, псевдо-випадкові послідовності допомагають створити віртуальні експерименти та симуляції, що дозволяють досліджувати та передбачати поведінку систем.

У багатьох програмах та алгоритмах, які вимагають випадковості, псевдо-випадкові послідовності можуть бути прийнятним рішенням через свою ефективність та передбачуваність. Однак, в криптографічних задачах, де безпека є критичною, використання справжньо випадкових послідовностей є необхідним.

Застосування псевдо-випадкових послідовностей включають генерацію градієнт-векторів для побудови псевдовипадкових шумів, що використовуються у візуальних ефектах, комп'ютерній графіці та моделюванні природних явищ. Ці шуми можуть створити враження реалістичної текстури, нерівномірного освітлення або вібрацій віддалених об'єктів.

В цілому, псевдо-випадкові послідовності є корисним інструментом для великої кількості застосувань, де випадковість або недетермінованість є важливою властивістю. Розуміння їх принципів та обмежень допомагає забезпечити правильне використання і відповідну оцінку їхньої випадковості в конкретних ситуаціях.

## 1.2 Хеш функції

Хеш-функція - це функція, яка приймає на вхід послідовність даних будь-якої довжини і обчислює фіксований вихідний код, відомий як хеш-код. Одна з основних властивостей хеш-функцій - це важкість знаходження двох різних вхідних послідовностей, які мають один і той самий хеш-код. Це називається колізією і в хороших хеш-функціях вони виникають дуже рідко.

Хеш-функції мають широке застосування в різних галузях. Одним з найпоширеніших використань є зберігання паролів. Замість зберігання самих паролів, хеш-функція обчислює хеш-код пароля і зберігається лише цей код. При перевірці пароля, введені дані обчислюються хеш-функцією, і їх хеш-код порівнюється зі збереженим. Це дозволяє перевірити правильність пароля без зберігання самого пароля, що забезпечує безпеку в разі витоку бази даних.

Хеш-функції також використовуються для перевірки цілісності даних. Наприклад, при завантаженні файлу з Інтернету можна перевірити його цілісність, порівнявши хеш-код файлу, обчислений на локальному комп'ютері, з вказаним хеш-кодом на веб-сайті. Якщо хеш-коди співпадають, це означає, що файл не був пошкоджений під час передачі.

Хеш-функції також застосовуються для швидкого доступу до даних. Наприклад, в хеш-таблицях ключі перетворюються в хеш-коди, які використовуються для швидкого пошуку значень. Це особливо корисно для пошуку великих наборів даних.

У вашому випадку, для генерації градієнт-векторів хеш-функція використовується для перетворення тривимірних координат і додаткових даних (seed) у фіксований хеш-код. Цей хеш-код може служити як початкова точка для псевдо-випадкової послідовності, яка буде використовуватися для генерації псевдовипадкових шумів. Таким чином, ми отримуємо контрольований і передбачуваний спосіб генерації шумів залежно від вхідних даних.

## 1.3 Псевдовипадкові шуми

Псевдовипадкові шуми є шумами, що генеруються за допомогою псевдовипадкових алгоритмів. Ці шуми можуть бути будь якої вимірності і можуть бути використані для моделювання природних або випадкових явищ, таких як вітрові коливання, хвилі в океані або деталі пейзажу. Вони генеруються шляхом використання математичних формул та псевдовипадкових послідовностей, що надають їм властивості випадковості, але при цьому є детермінованими. Псевдовипадкові шуми є корисним інструментом у багатьох галузях, включаючи комп'ютерну графіку, візуальні ефекти, моделювання природних явищ, аудіо та багато інших.

Один з найпопулярніших видів псевдовипадкових шумів - це шум Перліна, який названий на честь Кена Перліна, який його вперше запропонував. Шум Перліна генерується за допомогою градієнтного вектора, що призводить до створення нерегулярних і плавних змін яскравості або висоти. Цей вид шуму часто використовується для створення текстур, рельєфів і природних ландшафтів.

Інший популярний вид псевдовипадкових шумів - це симплекс-шум (simplex noise), який є вдосконаленням шуму Перліна. Симплекс-шум забезпечує більш гладкі переходи і вищу якість результату. Він часто використовується у графічних програмах для створення природних ефектів, таких як хвилі, хмари і трава.

Ще один поширений вид псевдовипадкових шумів - це шум Вороного (Voronoi noise). Він базується на теорії Вороного і створює мозаїчні ефекти з утворенням клітинок, які заповнюють простір. Шум Вороного використовується для створення вигляду кристалів, каміння або інших нерегулярних структур.

Кожен вид псевдовипадкових шумів має свої особливості і застосування в конкретних областях. Вони дозволяють створити візуально привабливі і реалістичні ефекти, що додають деталізацію та реалізм до комп'ютерних сцен і моделей.

## 1.4 Шум Перліна

Шум Перліна (Perlin noise) - це вид псевдовипадкового шуму, що використовується для моделювання нерегулярних текстур, таких як поверхні, хмари, вогні та інше. Був винайдений Кеном Перліном, створює плавні переходи між значеннями, створюючи ілюзію природного шуму. Шум Перліна генерується за допомогою інтерполяції градієнтів у випадкових точках сітки. Шум Перліна може бути будь якої розмірності, але зазвичай реалізується, як двовимірна, або тривимірна функція. У моєму випадку я буду використовувати тривимірний шум Перліна, бо генерації поверхні буде виконуватися у тривимірному просторі і двовимірна площина буде мати артефакти при накладанні на сферу.

Реалізації шуму Перліна проводиться в декілька етапів:

1. Спочатку ми визначаємо n-вимірну сітку, де кожній координаті сітки присвоюється n-вимірний вектор (градієнт) [рис. 1.1](#Рисунок_1_1). Цей вектор рандомізують за допомогою псевдо-випадкової послідовності, використовуючи координату сітки – як seed. Оскільки я реалізовував тривимірний шум Перліна, у моєму випадку це тривимірний вектор. Надалі, цей вектор потрібно нормалізувати.
2. Другий крок – це визначити, до якої комірки у нашій сітці потрапляє точка, значення якої треба знайти.
3. Для кожного вузла комірки визначаємо вектор відстані між точкою та координатами вузла.
4. Надалі обчислюємо скалярні добутки векторів відстані та градієнтних векторів кожного вузла комірки.
5. Значення скалярних добутків інтерполюють і ми отримуємо фінальну точку у потрібній координаті.

A picture containing diagram, line, colorfulness, plot

Description automatically generated

Рисунок .1 - Градієнт-вектори у вершинах сітки та їх інтерполяція

Така процедура повторюється для кожної точки, яку нам потрібно визначити. В результаті виконання двовимірного варіанту алгоритму, ми отримуємо набір точок, який можна перевести у зображення [рис. 1.2](#Рисунок_1_2).

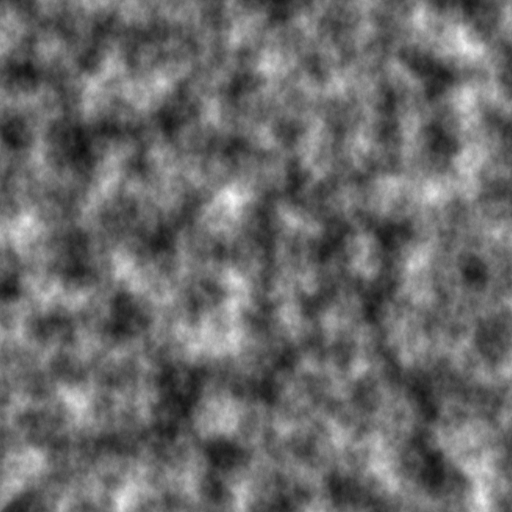


Рисунок 1.2 – Результат виконання алгоритму шуму Перліна

## 1.5 Шум Симплекс

Шум Симплекс (Simplex noise) є одним з популярних методів генерації псевдовипадкових шумів, який знаходить широке застосування у комп'ютерній графіці, симуляції природних явищ та інших областях [рис 1.3](#Рисунок_1_3).

У порівнянні з шумом Перліна, шум Симплекс пропонує поліпшення у якості та властивостях згладженості шумової поверхні. Цей метод використовує багатовимірні симплекси замість градієнтів на сітці, що дозволяє краще розподіляти значення шуму та уникати артефактів.

Шум Симплекс має деякі переваги порівняно з шумом Перліна, зокрема:

* Краща згладженість: Завдяки використанню симплексів замість градієнтів, шум Симплекс забезпечує більш плавні та згладжені переходи між значеннями шуму. Це дозволяє отримувати більш природні та реалістичні результати.
* Більша ефективність: Алгоритм генерації шуму Симплекс є більш ефективним, оскільки він використовує менше обчислювальних операцій для отримання значення шуму.
* Менше артефактів: Шум Симплекс має менше артефактів порівняно з шумом Перліна, таких як видимі сітчасті структури. Це дозволяє отримувати більш рівномірний та природній вигляд шумової поверхні.

Вибір між шумом Перліна та шумом Симплекс залежить від конкретної задачі і потреб у якості та ефективності. Обидва методи є корисними інструментами для моделювання природних та випадкових явищ, і вибір одного чи іншого може бути залежний від особливостей конкретного використання.

A close-up of a black and white speckled surface

Description automatically generated with low confidence

Рисунок 1.3 – Результат виконання шуму Сиплекс.

Хоч шум Симплекс і являється покращеною версією шума Перліна, при генеруванні поверхні вибір між цими основними шумами не грає великої ролі у кінцевому результаті.

## 1.6 Шум Вороний

Шум Вороного (Voronoi noise) є цікавим методом генерації псевдовипадкових шумів, який базується на діаграмі Вороного. Діаграма Вороного розбиває простір на області, використовуючи набір точок як базис. Кожна точка у шумі Вороного належить до своєї найближчої точки в цій діаграмі, і це надає шуму його характерний вигляд [рис. 1.4](#Рисунок_1_4).

Генерація шуму Вороного зазвичай включає наступні кроки:

* Вибір початкового набору точок: Генерується набір точок випадковим чином або за допомогою іншого псевдовипадкового алгоритму. Ці точки виступають як базис для діаграми Вороного.
* Обчислення найближчих точок: Кожній точці у просторі присвоюється значення, відповідне найближчій точці з початкового набору. Це може бути відстань або інше значення, яке відображає відношення до найближчої точки.
* Застосування інтерполяції: Для отримання плавних переходів між точками використовується інтерполяція значень шуму.
* Модифікація значень: Застосування додаткових операцій, таких як згладжування, масштабування або перетворення значень, може змінити вигляд та характеристики шуму Вороного.

Шум Вороного може бути використаний для створення випадкових паттернів, таких як плями, фрактальні структури або границі між областями. Його хаотичний та непередбачуваний вигляд робить його корисним інструментом для моделювання природних та випадкових явищ, а також для генерації художніх ефектів у комп'ютерній графіці. У нашому випадку, шум Вороного може використовуватися задля генерації варіацій у біомах поверхні, або для генерації тектонічних плит, якщо планета генерується принципом симуляції.

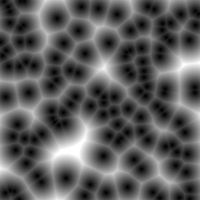


Рисунок 1.4 – Шум Вороного

## 1.7 Побудова сфери

Для того, щоб знайти карту висот на сфері, потрібно спочатку побудувати сітку вершин (mesh), які і будуть висотами. Є декілька способів побудови меша сфери, серед яких основні:

1. UV-сфера, яка найчастіше використовується для накладання UV розгортки текстур. Основний підхід полягає в теселяції сфери по мередіанам та паралелям [рис. 1.5](#Рисунок_1_5). Хоч цей підхід є дуже зручним для розгортки [рис. 1.6](#Рисунок_1_6) і накладання двовимірної текстури, для наших цілей цей метод не підходить, бо точність поверхні сильно залежить від частини сфери, в якій ми знаходимося. У полюсах спостерігається велике скупчення точок, коли як на екваторі точність недостатня.

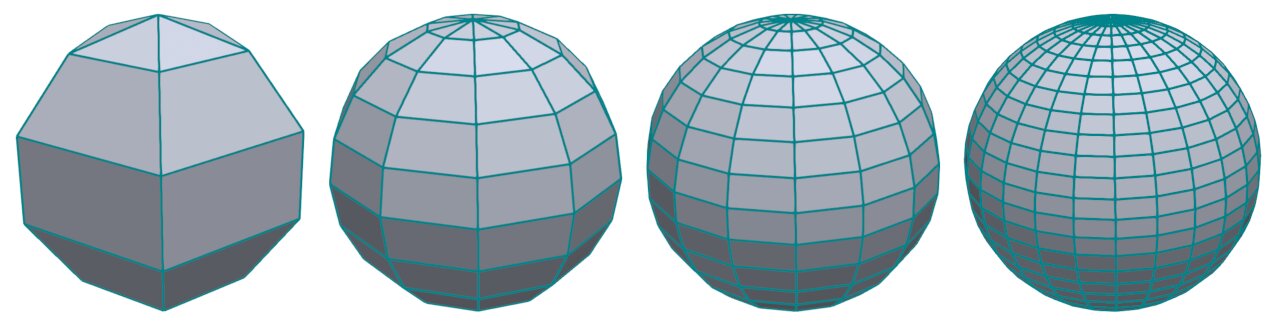


Рисунок 1.5 – Поділ поверхні у UV-сфері

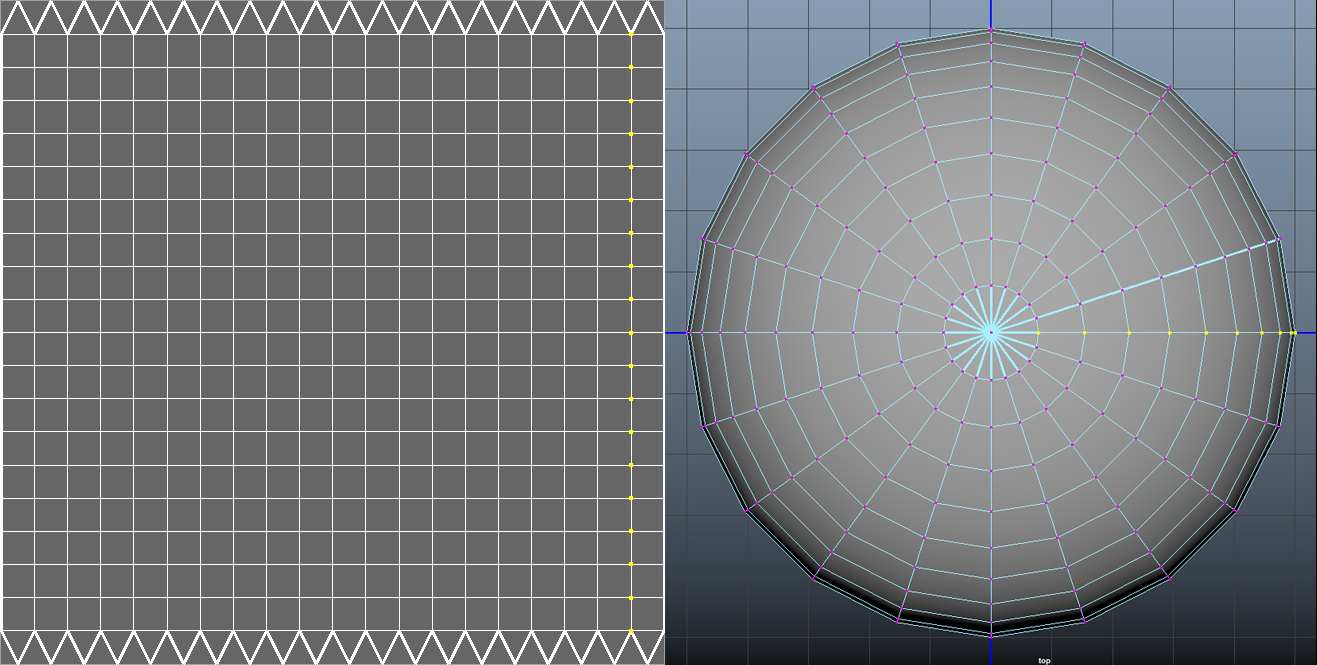


Рисунок 1.6 – UV-розгортка сфери

1. Наступний тип побудови поверхні сфери – це ікосфера. Назва пішла від ікосаедра (двадцятигранника), з якого і починається побудова сфери. Це найкращий спосіб рівномірно розподілити точки по сфері, але при цьому ми не маємо зручного підходу для процедурного поділу поверхні на менші елементи [рис. 1.7](#Рисунок_1_7).

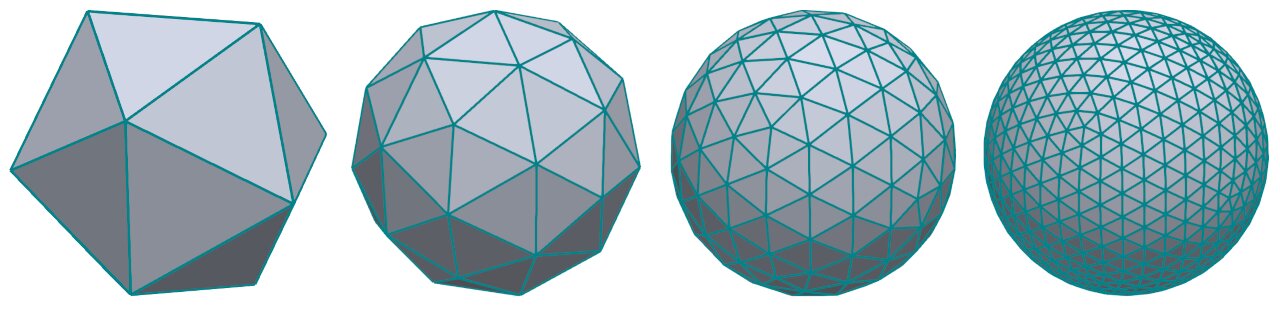


Рисунок 1.7 - Ікосфера

1. Останній спосіб – це кубо-сфера, яка з початку складається з 6 сторін куба, де довжина ребра куба – це діаметр сфери [рис. 1.8](#Рисунок_1_8). Хоч він розподіляє точки трохи гірше, ніж попередній метод, він є найбільш придатним способом для нашої ситуації, бо для розділу одного квадрату на більш деталізований варіант, нам потрібно взяти чотири квадрати, які рівно в 2 рази менші, ніж поточна сторона. Це і буде метод, який я використовував для побудови поверхні.

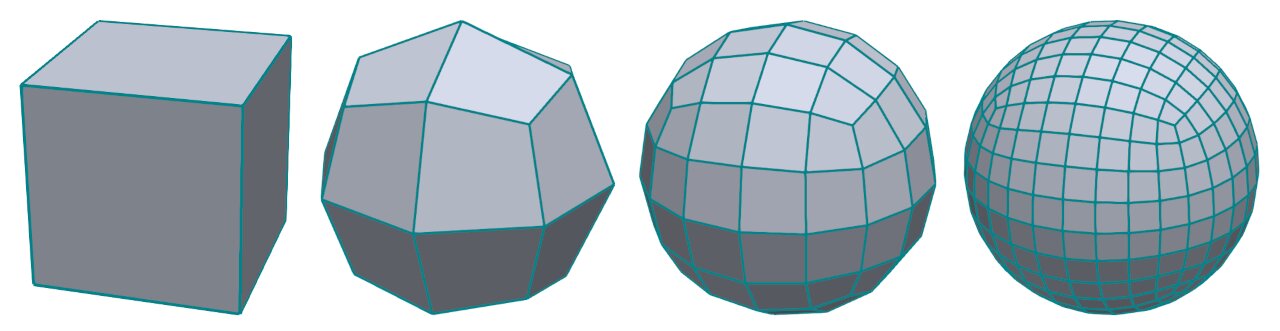


Рисунок 1.8 – Побудова кубосфери

## 1.8 Quad-tree структура

Quad-tree (квадродерево) є структурою даних, яка дозволяє поділити двовимірний простір на менші регіони шляхом рекурсивного розділення. Це дуже корисна структура для просторових задач, де необхідно швидко виконувати операції пошуку, вставки та видалення даних.

Процес побудови quad-tree починається з одного великого квадрата, який представляє весь простір. Потім цей квадрат розділяється на чотири рівні підрегіони, відомі як квадранти [рис. 1.9](#Рисунок_1_9). Кожен квадрант може бути поділений знову на чотири менші квадранти, якщо це необхідно, і цей процес може продовжуватися до досягнення певного критерію, наприклад, певного розміру або кількості об'єктів у кожному квадранті.

Один з основних переваг quad-tree полягає у тому, що він дозволяє ефективний пошук об'єктів у просторі. Замість перебору всіх об'єктів у просторі, можна швидко локалізувати потрібний регіон та виконувати операції тільки у ньому. Це дозволяє зменшити обсяг обчислень та покращити продуктивність.

У нашому випадку, кожна з шести сторін куба буде ділитися індивідуально одна від одної [рис. 1.10](#Рисунок_1_10) з використанням критерію, яким є відстань від камери порівняно з розміром поточної сторони куба. Це дозволяє ефективно керувати областями, які потребують подальшої обробки та дозволяє зосередитися на більш важливих ділянках простору.

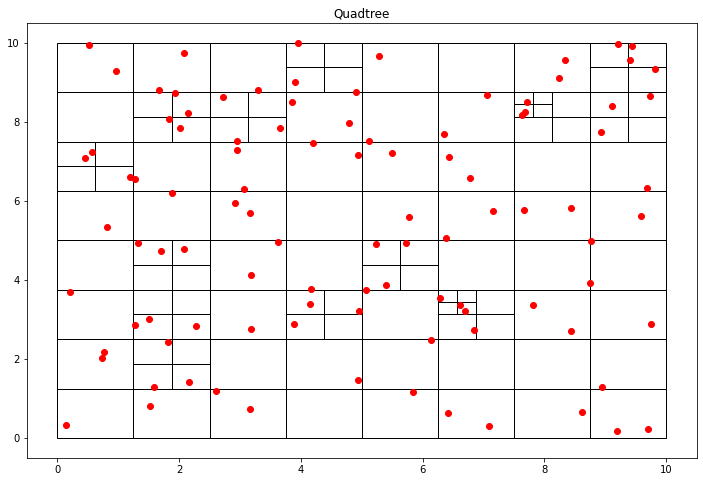


Рисунок 1.9 – Розподіл двовимірного простору, використовуючи структуру Quad-tree

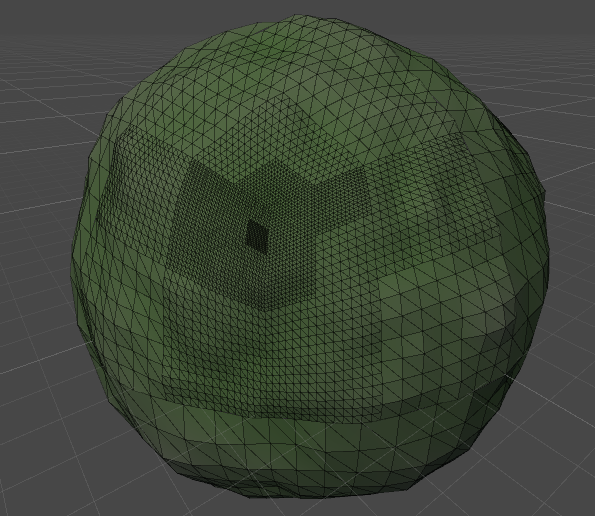


Рисунок 1.10 – Quad-tree структура на сфері

Quad-tree є потужним інструментом для ефективної організації та операцій з даними в просторових задачах, і його застосування широко поширене у таких галузях, як комп'ютерна графіка, комп'ютерне бачення, географічні інформаційні системи та інші.

# 2 МАТЕМАТИЧНА МОДЕЛЬ

## 2.1 Шум Перліна

Пошук точки у шумі Перліна зводиться до пошуку найближчих градієнт векторів до даної точки. У випадку тривимірного шуму Перліну наша точка попадає у так названу комірку, яка представляє собою куб. Полотно шуму складається з багатьох таких кубів, які являють собою сітку. У кожній вершині цієї комірки ми знаходимо значення градієнт вектора. Градієнт-вектор – це вектор однакової з шумом розмірності, де когне зі значень вектору рандомізується, використовуючи псевдо-випаднові послідовності. Сам вектор обов'язково повинен бути нормалізований перед його подальшим використанням. Одна комірка шуму Перліна визначається шляхом віднімання від поточної точки остачі від ділення на розмір комірки [(2.1)](#Формула_2_1), де мінімальна точка у комірці по всім осям, – точка, в якій нам потрібно знайти значення шуму, S – розмір комірки шуму, ‘%’ – операція остачі від ділення.

(2.1)

Остання точка комірки визначається шляхом додавання розміру комірки до першої точки [(2.2)](#Формула_2_2).

(2.2)

Таким чином ми отримали мінімальну та максимальну точку у кубі. Шляхом змішування координат цих двох точок, ми можемо знайти усі 8 вершин куба, оскільки сторони куба паралельні осям координат у просторі шуму. Наступна задача – це у кожній з цих точок знайти значення градієнт-вектора.

Градієнт-вектор у точці , де функція-рандомізатор, яка приймає координату точки, як seed, визначається за формулою [(2.3)](#Формула_2_3).

(2.3)

Після того, як ми отримали вісім градієнт-векторів, нам потрібно їх інтерполювати. Оскільки ми працюємо з векторами, доречно використовувати скалярний добуток. Робити скалярний добуток ми будемо між градієнт-вектором у цій точці та вектором від бажаної точки до вершини комірки.

Скалярне значення у точці після скалярного добутку між градієнт-вектором та відстанню між визначається за формулою [(2.4)](#Формула_2_4).

(2.4)

Після цієї операції в нас залишаються вісім скалярних значень. Значення інтерполюються між собою за законом , де V – результуюче значення шуму у точці .

Таким чином ми можемо заповнити текстуру будь-якого розміру, а саму комірну можна згенерувати лише один раз і змінювати тільки тоді, коли бажана точка виходить з цього проміжку.

## 2.2 Кубо-сфера

Щоб побудувати поверхню, нам спочатку потрібно знайти точки, в яких ми будемо знаходити точки поверхні. Оскільки ми намагаємося отримати поверхню сфери, доречно використовувати принцип побудови сферичної поверхні під назвою «кубосфера». Для кожної з шости сторін куба буде використовуватися одий і той самий алгоритм. Знаходження точок одніїє сторони куба за допомогою лінійної комбінації векторів виконується за формулою [(2.5)](#Формула_2_5), де - точка на стороні куба, одиничні вектори, розташовані увздовж локальних вісей сторони куба, центр сторони куба, r – радіус сфери, кількість точок уздовж однієї осі.

(2.5)

В результаті ми отримуємо 6 сторін куба, кожна з яких складається з n\*n точок. – це фінальна координата точки на сфері, яка знаходження шляхом нормалізації точок Кубо-сфери [(2.6)](#Формула_2_6). Після помноження на її радіус, ми отримуємо точки ідеальної сфери, які надалі можна використовувати для накладання октав шуму Перліна.

(2.6)

Для отримання фінальної точки поверхні, ми беремо значення шуму у кожній з цих точок. Оскільки ці точки представлені у вигляді векторів, ми помножаємо модифікований результат шуму на наш вектор і отримуємо фінальну точку поверхні. При використанні багатьох октав шуму, кожна наступна октава має вдвічі менший вплив на поверхню, ніж попередня.

# 3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 3.1 Платформа Unity

Для реалізації програми я обрав платформу Unity3D, яке є одним з найпопулярніших і потужних програмних забезпечень для розробки ігор, симуляцій і візуальних додатків [рис. 3.1](#Рисунок_3_1). Воно надає інтегроване середовище розробки, яке дозволяє створювати інтерактивні досвіди для різних платформ, включаючи комп'ютери, мобільні пристрої, віртуальну реальність та інші.

A black and white logo

Description automatically generated with medium confidence

Рисунок 3.1 – Логотип Unity3D

Основні функції Unity3D:

1. Графічний редактор: Unity3D має потужний графічний редактор, який дозволяє візуально створювати та редагувати об'єкти, сцени, ефекти та інші компоненти гри. Це дозволяє розробникам швидко створювати та налаштовувати візуальний вигляд своїх проектів.
2. Скриптування: Unity3D підтримує різні мови програмування, такі як C#, JavaScript і Boo. Це дає розробникам велику гнучкість у виборі мови програмування та створенні власної логіки гри або додатку.
3. Фізика та колізії: Unity3D має вбудовану систему фізики, яка дозволяє реалістично моделювати рух об'єктів, колізії та взаємодії між ними. Це робить гру більш живою та реалістичною.
4. Імпорт активів: Unity3D підтримує широкий спектр форматів активів, таких як зображення, звук, відео, 3D-моделі, анімації тощо. Це дозволяє розробникам використовувати свої власні ресурси або завантажувати готові активи з онлайн-бібліотек або ринків активів.
5. Мультиплатформена розробка: Unity3D підтримує розробку для різних платформ, включаючи Windows, macOS, iOS, Android, Xbox, PlayStation і багато інших. Це дозволяє розробникам створювати свої проекти для багатьох різних пристроїв безпосередньо з Unity3D.

Unity3D використовується як професійними розробниками, так і студентами для створення ігор, симуляцій, тренажерів, візуалізаційних додатків та багато іншого. Його потужність, гнучкість та активна спільнота розробників роблять Unity3D популярним вибором для будь-якої проектної потреби.

## 3.2 Render pipeline

Render pipeline - це технологія, яка визначає послідовність кроків, необхідних для відтворення графічного вмісту на екрані. Ця технологія виконує різні обчислення та операції для кожного кадру, включаючи обробку геометрії, освітлення, тіней, текстури та інших ефектів, що дозволяють створити реалістичне зображення.

У Unity, за замовчуванням використовується Build-in Render Pipeline, який є застарілим і менш гнучким. Проте, Unity також пропонує альтернативні варіанти рендер-пайплайнів для поліпшення графічної якості та продуктивності.

Одним з варіантів є High Definition Render Pipeline (HDRP), який спрямований на створення найреалістичніших графічних сцен. Він підтримує фізично-коректні матеріали, реалістичне освітлення, тіні, рефлексії та інші передові ефекти. HDRP використовує сучасні техніки рендерингу, такі як Global Illumination (GI) та Screen Space Reflections (SSR), для досягнення високої якості зображення.

Іншим варіантом є Universal Render Pipeline (URP), який покликаний оптимізувати продуктивність і підтримувати більш широкий спектр платформ. URP використовує сучасні техніки рендерингу, такі як Deferred Shading та Screen Space Ambient Occlusion (SSAO), а також підтримує можливість використання ShaderGraph для створення шейдерів за допомогою візуального редагування графів.

Universal Render Pipeline надає гнучкість та оптимізацію, дозволяючи розробникам досягти графічної якості, яка відповідає їх потребам, одночасно забезпечуючи хорошу продуктивність на різних платформах. Використання ShaderGraph у поєднанні з URP дозволяє створювати шейдери за допомогою візуального редагування графів, що спрощує процес розробки шейдерів і дозволяє швидко досягти бажаного візуального ефекту.

Вибір між HDRP і URP залежить від конкретних потреб проекту, де HDRP надає найбільшу графічну якість, а URP забезпечує кращу продуктивність і ширшу підтримку платформ.

Також він надає можливість написання шейдерів шляхом складання графу з різних методів за допомогою ShaderGraph.

## 3.3 Shader graph

Shader graph - це інструмент, який дозволяє візуально створювати і редагувати шейдерні графи. Він забезпечує графічне середовище, де ви можете складати графи з вузлів, що представляють шейдерні ефекти та операції. Це дає можливість програмувати візуально, без необхідності писати код вручну.

Shader graph надає велику кількість готових вузлів, що представляють різноманітні шейдерні операції, наприклад, вузли для обчислення освітлення, кольору, текстурного зчитування, альфа-змішування та багато інших. Ви можете з'єднувати ці вузли між собою, створюючи складні графи, які описують поведінку шейдера.

Після того, як ви скомпонували шейдерний граф, його необхідно компілювати в шейдерний код, що може бути використаний у програмі. Компілятор shader graph перетворює граф у два окремі файли: один для вершинного шейдера (vertex shader) інший для фрагментного шейдера (fragment shader). Вершинний шейдер обробляє кожен вершину геометрії, встановлюючи її позицію та інші властивості. Фрагментний шейдер працює на рівні фрагментів пікселів, обчислюючи кольори та інші атрибути, які відображаються на відображенні екрану.

Shader graph дозволяє швидко створювати та налаштовувати шейдерні ефекти без необхідності в глибокому розумінні шейдерного програмування. Це робить його корисним інструментом для розробки графічних програм, ігор, візуалізації даних та багатьох інших застосувань, де необхідно використовувати шейдери.

# 4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Результатом виконання цієї роботи став алгоритм побудови карти висот планети, з можливістю вибору величини октави шуму, накладання різних октав одне на одну для отримання більш реалістичних результатів та розмалюванням поверхні по висотам.

Алгоритм використовує структуру Quad-tree для збільшення резолюції поверхні поруч із спостерігачем [рис. 4.1](#Рисунок_4_1). Інша частина мешу, яка знаходиться далеко від спостерігача, залишається з більш простою поверхнею, що дозволяє зосередити ресурси комп'ютера на ділянці поверхні поруч із камерою спостерігача.

A screenshot of a video game

Description automatically generated with low confidence

Рисунок 4.1 – Карта висот на сфері із зображенням Quad-tree розподілу

Розмальовка поверхні виконується за критерієм дистанції точки поверхні від центра планети. Ми можемо виділити декілька точок висот, за допомогою яких буде виконуватись інтерполяція кольору. Наприклад, точка океану та точка континенту [рис 4.2](#Рисунок_4_2), [рис 4.3](#Рисунок_4_3).

A green and blue planet

Description automatically generated with low confidence

Рисунок 4.2 – Кольорове розділення поверхні на океан та континент.

A green and blue planet

Description automatically generated with low confidence

Рисунок 4.3 – Генерація поверхні окремими плоскостями, які не розділяють між собою вершини.

Управління генерацією планети виконується за допомогою компонента Sphere Builder [рис. 4.4](#Рисунок_4_4), у якому можна вибрати seed, найбільшу октаву шуму, кількість октав та інше.

A screenshot of a computer program

Description automatically generated with medium confidence

Рисунок 4.4 – Компонент для генерації планети

Згенерована планета з малою октавою шуму виглядає наступним чином. На [рис 4.5](#Рисунок_4_5) ми бачимо сам результат. Вибір кольору по висоті з двох кольорів: темно-зеленого, який відповідає за висоти близько рівня моря та сірого, який відповідає за скали. На [рис 4.6](#Рисунок_4_6) видно той самий ландшафт, але з демонстрацією Quad-tree.

A close-up of a ball

Description automatically generated with low confidence

Рисунок 4.5 – Ландшафт із шумом малої октави

A picture containing text, screenshot, pc game, 3d modeling

Description automatically generated

Рисунок 4.6 – Та сама поверхня, але з демонстрацією Quad-tree

Структура Quad-tree допомагає генерувати поверхню поступово, що дає можливість швидко генерувати більш детальні ділянки замість того, щоб генерувати усю поверхню зі збільшеною деталізацією. На [рис 4.7](#Рисунок_4_7) ми бачимо планету, згенеровану без використання Quad-tree, на [рис 4.8](#Рисунок_4_8) демонстрацію деталізації. Далі, на [рис 4.9](#Рисунок_4_9), ми бачимо відмінність поверхні при використанні Quad-tree структури. Чим темніша поверхня – тип щільніше один до одного генерувалися точки висоти. Тобто деталізація у таких ділянках найвища. Протиположна сторона планети генерується з мінімальною деталізацією, що видно з [рис 4.10](#Рисунок_4_10). Таке розподілення дозволяє генерувати дуже велику щільність висот поряд з камерою спостерігача.

A picture containing screenshot, 3d modeling, sphere

Description automatically generated

Рисунок 4.7 – Поверхня планети із однаковою деталізацією (без Quad-tree)

A picture containing screenshot

Description automatically generated

Рисунок 4.8 – Поверхня планети без Quad-tree з демонстрацією мешу

A picture containing screenshot, sphere

Description automatically generated

Рисунок 4.9 – Поверхня планети з використанням Quad-tree структури. Розмір однієї ділянки – 16х16 точок висоти

A picture containing sphere

Description automatically generated

Рисунок 4.10 – Протилежна до камери сторона планети.

Використовуючи цей підхід, ми можемо значно зменшити кількість точок висот у пам'яті, а зі збільшенням максимальної кількості поділів, різниця між статичною резолюцією та поділом по Quad-tree сильно збільшується. На [рис. 4.11](#Рисунок_4_11) ми бачимо два графіки, один з яких відповідає кількість точок при заповнені площини максимальною резолюцією, інший за використання Quad-tree. По горизонталі ми відміряємо кількість точок у одній площині максимальної деталізації.

A picture containing text, screenshot, line, plot

Description automatically generated

Рисунок 4.11 – Різниця між статичною резолюцією та Quad-tree

Із графіку видно, що для досягнення точності ділянки поверхні під камерою у 1024 точки, загальна кількість витрачених точок при використанні Quad-tree менше у 3.5 рази, ніж при рендеру усієї поверхні з однаковою точністю. При подальшому збільшенню точності поверхні, використання звичайного метода стає все більш неможливим, доки Quad-tree все ще зберігає робочий стан. Ефективність метода Quad-tree можна підвищити, змінивши критерій поділу поверхні. Але при цьому буде погіршуватись якість поділу.

# ВИСНОВКИ

За результатами виконаної роботи можна зробити такі висновки:

* Досліджено та проведено аналіз предметної області, а саме: проаналізовано принципи функціонування псевдовипадкових шумів, систем генерації ландшафту, та його оптимізації;
* Було побудовано математичну модель системи процедурної генерації псевдовипадкових поверхонь;
* Визначено функціональні вимоги до програмного забезпечення, обрано мову програмування та платформу для програмної реалізації;
* За постановленим завданням спроектовано та розроблено систему процедурної генерації ландшафту з використанням шуму Перліна;
* Проаналізовано отримані результати.

Результатом виконання цієї роботи став алгоритм побудови карти висот планети, з можливістю вибору величини октави шуму, накладання різних октав одне на одну для отримання більш реалістичних результатів. Алгоритм використовує структуру Quad-tree для збільшення резолюції поверхні поруч із спостерігачем. Інша частина мешу, яка знаходиться далеко від спостерігача, залишається з більш простою поверхнею, що дозволяє зосередити ресурси комп'ютера на ділянці поверхні поруч із камерою спостерігача.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційний сайт Міністерства освіти та науки України: <http://mon.gov.ua/>
2. Закон України про вище освіту [Електронний ресурс] // Верховна Рада України: [сайт]- про вищу освіту Верховна рада України; Закон від 01.07.2014№1556-VII.- Режим доступу: [http://zakon2.rada.gov.ua/ laws/show/1156-18](http://zakon2.rada.gov.ua/%20laws/show/1156-18%20(дата%20з) (Дата звернення 03.04.2020).
3. <http://www.dnu.dp.ua/docs/obgovorennya/Polozhennya_Antiplagiat_2016.doc>
4. Хеш-функції - Режим доступу: [https://uk.wikipedia.org/wiki/Hash-Function](https://uk.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F#:~:text=%D0%A5%D0%B5%D1%88%2D%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F%2C%20%D0%B0%D0%B1%D0%BE%20%D0%B3%D0%B5%D1%88%2D,%D1%80%D0%BE%D0%B7%D0%BC%D1%96%D1%80%D1%83%20%D0%B2%20%D0%B4%D0%B0%D0%BD%D1%96%20%D1%84%D1%96%D0%BA%D1%81%D0%BE%D0%B2%D0%B0%D0%BD%D0%BE%D0%B3%D0%BE%20%D1%80%D0%BE%D0%B7%D0%BC%D1%96%D1%80%D1%83) - Hash Function
5. Шум Перліна - Режим доступу: <https://en.wikipedia.org/wiki/Perlin_noise> - Perlin noise
6. Шум симплекс - Режим доступу: <https://en.wikipedia.org/wiki/Simplex_noise> - Simplex noise
7. Процедурна генерація ландшафту - Режим доступу: <https://www.simondev.io/> - SimonDev
8. Псевдовипадкові шуми - Режим доступу: <https://catlikecoding.com/unity/tutorials/pseudorandom-noise/> - Unity Pseudorandom Noise Tutorials

# ДОДАТКИ

## Додаток А

**Побудова початкових сторін сфери**

public class SphereBuilder : MonoBehaviour

{

[SerializeField, Range(1, 50)]

float radius = 1;

[SerializeField, Range(5, 320)]

int pointsPerAxis;

[SerializeField]

Material meshMaterial;

[SerializeField, Header("Create when entered play mode")]

bool createOnStart;

[SerializeField, Min(1)]

int noiseSpan = 1;

[SerializeField, Range(1, 7)]

int noiseOctaves = 1;

[SerializeField]

Transform quadTreeControllPoint;

List<SphereChunk> sphereChunks = new List<SphereChunk>(6);

[BurstCompile(CompileSynchronously = true, FloatMode = FloatMode.Fast, FloatPrecision = FloatPrecision.Low, OptimizeFor = OptimizeFor.Performance)]

struct TestJob : IJobFor

{

[WriteOnly]

public NativeArray<int> array;

public void Execute(int index)

{

array[index] = index;

}

}

void Start()

{

if (createOnStart)

CreateSphere();

}

private void Update()

{

QuadTreeLodUpdate();

}

void QuadTreeLodUpdate()

{

if (quadTreeControllPoint == null)

return;

if (sphereChunks.Count == 0)

return;

foreach (SphereChunk sphereChunk in sphereChunks)

sphereChunk.QuadTreeLodUpdate(quadTreeControllPoint);

}

public void CreateSphere()

{

RemoveChilds();

GameObject newSphere = new GameObject();

newSphere.name = "Procedural Planet";

Transform newSphereTransform = newSphere.transform;

newSphereTransform.parent = transform;

Vector2 сhunkSize = Vector2.one \* radius \* 2;

SphereChunk.ChunkParams[] chunkParams = new SphereChunk.ChunkParams[6]

{

new SphereChunk.ChunkParams(Vector3.forward, Vector3.up, Vector3.right \* radius, сhunkSize, radius), // Side Chunks

new SphereChunk.ChunkParams(Vector3.left, Vector3.up, Vector3.forward \* radius, сhunkSize, radius),

new SphereChunk.ChunkParams(Vector3.back, Vector3.up, Vector3.left \* radius, сhunkSize, radius),

new SphereChunk.ChunkParams(Vector3.right, Vector3.up, Vector3.back \* radius, сhunkSize, radius),

new SphereChunk.ChunkParams(Vector3.right, Vector3.forward, Vector3.up \* radius, сhunkSize, radius), // Top Plate

new SphereChunk.ChunkParams(Vector3.left, Vector3.forward, Vector3.down \* radius, сhunkSize, radius) // Bottom Plate

};

SphereChunk сhunkHolder;

foreach (SphereChunk.ChunkParams chunkParam in chunkParams)

{

сhunkHolder = InstantiateChunk(chunkParam);

сhunkHolder.transform.parent = newSphereTransform;

сhunkHolder.gameObject.SetActive(true);

sphereChunks.Add(сhunkHolder);

}

}

void RemoveChilds()

{

for (int i = 0; i < sphereChunks.Count; ++i)

SphereChunkObjectPool.PushChunk(sphereChunks[i]);

sphereChunks.Clear();

}

SphereChunk InstantiateChunk(SphereChunk.ChunkParams chunkParams)

{

SphereChunk sphereChunk = SphereChunkObjectPool.PopChunk();

MeshBuilder.MeshParams meshParams = new MeshBuilder.MeshParams()

{

noiseOctaves = noiseOctaves,

noiseSpan = noiseSpan,

pointsPerAxis = pointsPerAxis

};

sphereChunk.InstantiateChunk(chunkParams, meshParams);

return sphereChunk;

}

struct MeshPointsBuilderJob : IJobFor

{

[ReadOnly]

public NativeArray<int> vertexes;

[WriteOnly]

public NativeArray<int> normals;

public NativeArray<int> triangles;

public void Execute(int index)

{

}

}

}

## Додаток Б

**Класс, відповідальний за одну ділянку поверхні**

public class SphereChunk : MonoBehaviour

{

[SerializeField]

MeshFilter meshFilter;

[SerializeField]

MeshRenderer meshRenderer;

private void Awake()

{

meshRenderer = GetComponent<MeshRenderer>();

}

public enum SphereChunkMode

{

SingleChunk,

SubChunks

}

SphereChunkMode sphereChunkMode = SphereChunkMode.SingleChunk;

ChunkParams chunkParams;

List<SphereChunk> subChunks = new(4);

const int maxSplittings = 6;

int currentSplitting;

public void DestroyChunk()

{

if (sphereChunkMode == SphereChunkMode.SubChunks)

{

foreach (SphereChunk sphereChunk in subChunks)

sphereChunk.DestroyChunk();

//subChunks.Clear();

sphereChunkMode = SphereChunkMode.SingleChunk;

}

SphereChunkObjectPool.PushChunk(this);

}

MeshBuilder.MeshParams meshParamsCopy;

public void InstantiateChunk(ChunkParams chunkParams, MeshBuilder.MeshParams meshParams)

{

meshParamsCopy = meshParams;

this.chunkParams = chunkParams;

Mesh newMesh = MeshBuilder.BuildPlaneMesh(chunkParams, meshParams);

newMesh.RecalculateNormals();

newMesh.RecalculateTangents();

meshFilter.mesh = newMesh;

}

public void SplitChunk()

{

if (sphereChunkMode == SphereChunkMode.SubChunks)

return;

if (currentSplitting == maxSplittings)

return;

sphereChunkMode = SphereChunkMode.SubChunks;

ChunkParams[] subChunksParams = chunkParams.SplitChunk();

meshRenderer.enabled = false;

//gameObject.SetActive(false);

subChunks.Clear();

for (int i = 0; i < 4; ++i)

{

SphereChunk buffer = SphereChunkObjectPool.PopChunk();

buffer.InstantiateChunk(subChunksParams[i], meshParamsCopy);

buffer.transform.SetParent(transform);

buffer.gameObject.SetActive(true);

subChunks.Add(buffer);

}

}

public void MergeSubChunks()

{

if (sphereChunkMode == SphereChunkMode.SingleChunk)

return;

meshRenderer.enabled = true;

sphereChunkMode = SphereChunkMode.SingleChunk;

DestroySubChunks();

}

public void QuadTreeLodUpdate(Transform target)

{

Vector3 chunkCenterPosition = chunkParams.chunkCenter.normalized \* chunkParams.radius + transform.position;

float distanceToTarget = Vector3.Distance(target.position, chunkCenterPosition);

if (distanceToTarget < chunkParams.chunkSize.x)

{

if (sphereChunkMode == SphereChunkMode.SingleChunk)

SplitChunk();

else

foreach (SphereChunk sphereChunk in subChunks)

sphereChunk.QuadTreeLodUpdate(target);

}

else

{

if (sphereChunkMode == SphereChunkMode.SubChunks)

MergeSubChunks();

}

}

void DestroySubChunks()

{

foreach (SphereChunk sphereChunk in subChunks)

sphereChunk.DestroyChunk();

//subChunks.Clear();

sphereChunkMode = SphereChunkMode.SingleChunk;

}

public struct ChunkParams

{

public Vector3 xVector;

public Vector3 yVector;

public Vector3 chunkCenter;

public Vector2 chunkSize;

public float radius;

public ChunkParams(Vector3 xVector, Vector3 yVector, Vector3 chunkCenter, Vector2 chunkSize, float radius)

{

this.xVector = xVector;

this.yVector = yVector;

this.chunkCenter = chunkCenter;

this.chunkSize = chunkSize;

this.radius = radius;

}

public ChunkParams(ChunkParams chunkParams)

{

xVector = chunkParams.xVector;

yVector = chunkParams.yVector;

chunkCenter = chunkParams.chunkCenter;

chunkSize = chunkParams.chunkSize;

radius = chunkParams.radius;

}

public ChunkParams[] SplitChunk()

{

ChunkParams[] result = new ChunkParams[4];

for (int i = 0; i < 4; ++i)

result[i] = new ChunkParams

{

xVector = xVector,

yVector = yVector,

chunkSize = chunkSize / 2,

radius = radius

};

result[0].chunkCenter = chunkCenter - xVector \* chunkSize.x / 4 - yVector \* chunkSize.y / 4;

result[1].chunkCenter = chunkCenter + xVector \* chunkSize.x / 4 - yVector \* chunkSize.y / 4;

result[2].chunkCenter = chunkCenter - xVector \* chunkSize.x / 4 + yVector \* chunkSize.y / 4;

result[3].chunkCenter = chunkCenter + xVector \* chunkSize.x / 4 + yVector \* chunkSize.y / 4;

return result;

}

}

}

## Додаток В

**Реалізація шуму Перліна**

public static class Noise

{

struct NoiseChunk

{

public Vector3 basePointOffset;

public Vector3 basePoint;

public Vector3 endPoint;

public Vector3[] edgePoints;

public Vector3[] noisePoints;

public static float[] GeneratePoints(uint seed, int span, Vector3[] points)

{

float[] result = new float[points.Length];

Dictionary<Vector3, NoiseChunk> noiseChunks = new();

bool isBasePointExist = false;

void GenerateBasePoints(Vector3 basePointOffset, Vector3 basePoint, Vector3 point)

{

NoiseChunk noiseChunk = new NoiseChunk();

noiseChunk.basePointOffset = basePointOffset;

noiseChunk.basePoint = basePoint;

noiseChunk.endPoint = basePoint + Vector3.one \* span;

noiseChunk.edgePoints = new Vector3[8];//(8, Allocator.Persistent);

noiseChunk.edgePoints[0] = noiseChunk.basePoint;

noiseChunk.edgePoints[1] = new Vector3(noiseChunk.endPoint.x, noiseChunk.basePoint.y, noiseChunk.basePoint.z);

noiseChunk.edgePoints[2] = new Vector3(noiseChunk.basePoint.x, noiseChunk.endPoint.y, noiseChunk.basePoint.z);

noiseChunk.edgePoints[3] = new Vector3(noiseChunk.basePoint.x, noiseChunk.basePoint.y, noiseChunk.endPoint.z);

noiseChunk.edgePoints[4] = new Vector3(noiseChunk.endPoint.x, noiseChunk.endPoint.y, noiseChunk.basePoint.z);

noiseChunk.edgePoints[5] = new Vector3(noiseChunk.endPoint.x, noiseChunk.basePoint.y, noiseChunk.endPoint.z);

noiseChunk.edgePoints[6] = new Vector3(noiseChunk.basePoint.x, noiseChunk.endPoint.y, noiseChunk.endPoint.z);

noiseChunk.edgePoints[7] = noiseChunk.endPoint;

Vector3[] localNoisePoints = new Vector3[noiseChunk.edgePoints.Length];

for (int j = 0; j < noiseChunk.edgePoints.Length; ++j)

{

uint xh = noiseChunk.edgePoints[j].x.ConvertTo<uint>();

uint yh = noiseChunk.edgePoints[j].y.ConvertTo<uint>();

uint zh = noiseChunk.edgePoints[j].z.ConvertTo<uint>();

InitHash(xh + InitHash(yh + InitHash(zh)));

Vector3 gradientVector = HashVector;

localNoisePoints[j] = gradientVector;

}

noiseChunk.noisePoints = localNoisePoints;

noiseChunks.Add(basePoint, noiseChunk);

}

Vector3 currentPoint;

for (int i = 0; i < points.Length; ++i)

{

currentPoint = points[i] + Vector3.one \* 5000;

Vector3 basePointOffset = new Vector3(currentPoint.x % span, currentPoint.y % span, currentPoint.z % span);

Vector3 basePoint = currentPoint - basePointOffset;

if (!noiseChunks.ContainsKey(basePoint))

GenerateBasePoints(basePointOffset, basePoint, currentPoint);

NoiseChunk localNoiseChunk = noiseChunks[basePoint];

float[] dotProduct = new float[]

{

DotGradient(localNoiseChunk.noisePoints[0], localNoiseChunk.edgePoints[0], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[1], localNoiseChunk.edgePoints[1], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[2], localNoiseChunk.edgePoints[2], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[3], localNoiseChunk.edgePoints[3], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[4], localNoiseChunk.edgePoints[4], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[5], localNoiseChunk.edgePoints[5], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[6], localNoiseChunk.edgePoints[6], currentPoint, span),

DotGradient(localNoiseChunk.noisePoints[7], localNoiseChunk.edgePoints[7], currentPoint, span)

};

float xRelation = (currentPoint.x - localNoiseChunk.edgePoints[0].x) / span;

float yRelation = (currentPoint.y - localNoiseChunk.edgePoints[0].y) / span;

float zRelation = (currentPoint.z - localNoiseChunk.edgePoints[0].z) / span;

xRelation \*= xRelation;

yRelation \*= yRelation;

zRelation \*= zRelation;

float firstPlateInterpolation = Interpolate(

Interpolate(dotProduct[0], dotProduct[1], xRelation),

Interpolate(dotProduct[2], dotProduct[4], xRelation),

yRelation);

float secondPlateInterpolation = Interpolate(

Interpolate(dotProduct[3], dotProduct[5], xRelation),

Interpolate(dotProduct[6], dotProduct[7], xRelation),

yRelation);

result[i] = Interpolate(firstPlateInterpolation, secondPlateInterpolation, zRelation) / 2 + 0.5f;

}

return result;

}

public static float GeneratePoint(uint seed, int span, Vector3 point)

{

point += Vector3.one \* 5000;

Vector3 basePointOffset = new Vector3(point.x % span, point.y % span, point.z % span);

Vector3 basePoint = point - basePointOffset;

Vector3 endPoint = basePoint + Vector3.one \* span;

Vector3[] edgePoints = new Vector3[]

{

basePoint,

new Vector3(endPoint.x, basePoint.y, basePoint.z),

new Vector3(basePoint.x, endPoint.y, basePoint.z),

new Vector3(basePoint.x, basePoint.y, endPoint.z),

new Vector3(endPoint.x, endPoint.y, basePoint.z),

new Vector3(endPoint.x, basePoint.y, endPoint.z),

new Vector3(basePoint.x, endPoint.y, endPoint.z),

endPoint

};

Vector3[] noisePoints = new Vector3[edgePoints.Length];

float[] pointsRanges = new float[edgePoints.Length];

for (int i = 0; i < edgePoints.Length; ++i)

{

uint xh = edgePoints[i].x.ConvertTo<uint>();

uint yh = edgePoints[i].y.ConvertTo<uint>();

uint zh = edgePoints[i].z.ConvertTo<uint>();

InitHash(xh + InitHash(yh + InitHash(zh)));

Vector3 gradientVector = HashVector;

noisePoints[i] = gradientVector;

//pointsRanges[i] = 1 - (point - edgePoints[i]).magnitude / span;

//accumulate += noisePoints[i] \* pointsRanges[i];

}

float[] dotProduct = new float[]

{

DotGradient(noisePoints[0], edgePoints[0], point, span),

DotGradient(noisePoints[1], edgePoints[1], point, span),

DotGradient(noisePoints[2], edgePoints[2], point, span),

DotGradient(noisePoints[3], edgePoints[3], point, span),

DotGradient(noisePoints[4], edgePoints[4], point, span),

DotGradient(noisePoints[5], edgePoints[5], point, span),

DotGradient(noisePoints[6], edgePoints[6], point, span),

DotGradient(noisePoints[7], edgePoints[7], point, span)

};

float xRelation = (point.x - edgePoints[0].x) / span;

float yRelation = (point.y - edgePoints[0].y) / span;

float zRelation = (point.z - edgePoints[0].z) / span;

xRelation \*= xRelation;

yRelation \*= yRelation;

zRelation \*= zRelation;

float firstPlateInterpolation = Interpolate(

Interpolate(dotProduct[0], dotProduct[1], xRelation),

Interpolate(dotProduct[2], dotProduct[4], xRelation),

yRelation);

float secondPlateInterpolation = Interpolate(

Interpolate(dotProduct[3], dotProduct[5], xRelation),

Interpolate(dotProduct[6], dotProduct[7], xRelation),

yRelation);

float fullInterpolation = Interpolate(firstPlateInterpolation, secondPlateInterpolation, zRelation);

return fullInterpolation / 2 + 0.5f;

}

static float DotGradient(Vector3 noiseVector, Vector3 gridPoint, Vector3 point, float span)

{

Vector3 delta = point - gridPoint;

return Vector3.Dot(noiseVector, delta / span);

}

//static float Interpolate(float a, float b, float w) => a + (b - a) \* w;

static float Interpolate(float a, float b, float w) => (b - a) \* (Mathf.Sin(Mathf.PI \* (w - 0.5f)) / 2 + 0.5f) + a;

static Unity.Mathematics.Random random;

static uint InitHash(uint seed)

{

random.InitState(seed);

return random.NextUInt(0, 32769);

}

static Vector3 HashVector => random.NextFloat3(-1f, 1f);// new Vector3(Random.Range(-1f, 1f), Random.Range(-1f, 1f), Random.Range(-1f, 1f)).normalized;

}public struct MeshParams

{

public int pointsPerAxis;

public int noiseOctaves;

public int noiseSpan;

}

public static Mesh BuildPlaneMesh(SphereChunk.ChunkParams chunkParams, MeshParams meshParams) // With dublicating points // Optimized

{

Vector3 xVector = chunkParams.xVector.normalized;

Vector3 yVector = chunkParams.yVector.normalized;

Vector3 center = chunkParams.chunkCenter;

Vector2 size = chunkParams.chunkSize;

float radius = chunkParams.radius;

int pointsPerAxis = meshParams.pointsPerAxis;

int noiseOctaves = meshParams.noiseOctaves;

int noiseSpan = meshParams.noiseSpan;

Mesh newMesh = new Mesh();

int squarePerAxis = pointsPerAxis - 1;

int pointsOnPlane = pointsPerAxis \* pointsPerAxis;

int squaresOnPlane = squarePerAxis \* squarePerAxis;

int finalVertexCount = squaresOnPlane \* 4;

Vector3[] finalPoints = new Vector3[finalVertexCount];

Vector3[] normals = new Vector3[finalVertexCount];

int[] triangles = new int[squaresOnPlane \* 6]; // 6 = 2 triangles per square \* 3 vert per triangle

Vector3 basePoint = center - xVector \* size.x / 2 - yVector \* size.y / 2;

Vector3[] points = new Vector3[pointsOnPlane];

int pointIndex = 0;

Vector3 offsetX = size.x / (pointsPerAxis - 1) \* xVector;

Vector3 offsetY = size.y / (pointsPerAxis - 1) \* yVector;

for (int x = 0; x < pointsPerAxis; ++x)

for (int y = 0; y < pointsPerAxis; ++y, ++pointIndex)

{

points[pointIndex] = basePoint;

points[pointIndex] += x \* offsetX;

points[pointIndex] += y \* offsetY;

points[pointIndex] = points[pointIndex].normalized \* radius;

} // Every point on ideal sphere

float[][] noiseResults = new float[noiseOctaves][];

float finalKoef = 0.3f \* 0.5f;

for (int oct = 0; oct < noiseOctaves; ++oct)

{

noiseResults[oct] = Noise.GeneratePoints(0, noiseSpan / (oct + 1), points);

for (int i = 0; i < points.Length; ++i)

{

float modifiedNoiseValue = finalKoef / (oct + 1) \* noiseResults[oct][i];

points[i] += points[i] \* modifiedNoiseValue;

}

}

// Next step is just compose squares from those points

Vector3[,] points2D = new Vector3[pointsPerAxis, pointsPerAxis];

for (int i = 0; i < pointsPerAxis; ++i)

for (int j = 0; j < pointsPerAxis; ++j)

{

points2D[i, j] = points[i \* pointsPerAxis + j];

}

int finalPointsIter = 0;

int trianglesIter = 0;

for (int i = 1; i < pointsPerAxis; ++i)

for (int j = 1; j < pointsPerAxis; ++j)

{

normals[finalPointsIter] = points2D[i - 1, j - 1];

finalPoints[finalPointsIter++] = points2D[i - 1, j - 1];

normals[finalPointsIter] = points2D[i, j - 1];

finalPoints[finalPointsIter++] = points2D[i, j - 1];

normals[finalPointsIter] = points2D[i - 1, j];

finalPoints[finalPointsIter++] = points2D[i - 1, j];

normals[finalPointsIter] = points2D[i, j];

finalPoints[finalPointsIter++] = points2D[i, j];

triangles[trianglesIter++] = finalPointsIter - 4;

triangles[trianglesIter++] = finalPointsIter - 2;

triangles[trianglesIter++] = finalPointsIter - 3;

triangles[trianglesIter++] = finalPointsIter - 2;

triangles[trianglesIter++] = finalPointsIter - 1;

triangles[trianglesIter++] = finalPointsIter - 3;

}

newMesh.vertices = finalPoints;

newMesh.triangles = triangles;

newMesh.normals = normals;

//newMesh.RecalculateNormals();

//newMesh.RecalculateTangents();

return newMesh;

}

Нормоконтроль пройдено

\_\_\_\_\_\_\_\_\_ Ірина ШАПОВАЛ

(підпис)