**ДНІпровський національний університет**

**імені Олеся Гончара**

**ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ**

**КАФЕДРА КОМП’ЮТЕРНИХ ТЕХНОЛОГІЙ**

**КУРСОВА РОБОТА**

**ЗА СПЕЦІАЛЬНІСТЮ**

на тему: РОЗРОБКА СИСТЕМИ ПОЛІПШЕННЯ ЛАНДШАФТУ З ВИКОРИСТАННЯМ АЛГОРИТМІВ ЕРОЗІЇ ПОВЕРХОНЬ\_\_\_\_\_\_

## Освітньо–професійна програма Інформатика

## Спеціальність 113 Прикладна математика

## Галузь знань 11 Математика і статистика

## Рівень вищої освіти другий (магістерський)

Студента 1 курсу групи ПК-23м-1

\_\_\_\_\_Ільяшенко Є. В.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (прізвище та ініціали)

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

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Кількість балів \_\_\_\_\_\_\_\_\_\_\_

Національна шкала \_\_\_\_\_\_\_

Члени комісії:

Білозьоров В.Є.

(підпис) (прізвище та ініціали)

Зайцев В.Г.

(підпис) (прізвище та ініціали)

Золотько К.Є.

(підпис) (прізвище та ініціали)

м. Дніпро, 2024 р.

# **ЗМІСТ**

[ЗМІСТ 2](#_Toc168943541)

[ВСТУП 4](#_Toc168943542)

[ПОСТАНОВКА ЗАДАЧІ 7](#_Toc168943543)

[1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ 8](#_Toc168943544)

[1.1 Псевдо-випадкові послідовності 8](#_Toc168943545)

[1.2 Хеш функції 9](#_Toc168943546)

[1.3 Псевдовипадкові шуми 10](#_Toc168943547)

[1.4 Шум Перліна 11](#_Toc168943548)

[1.5 Шум Симплекс 12](#_Toc168943549)

[1.6 Шум Вороний 14](#_Toc168943550)

[1.7 Комбінація рівней шуму 15](#_Toc168943551)

[1.8 Принципи формування ландшафту 16](#_Toc168943552)

[1.9 Меш. Загальні поняття 17](#_Toc168943553)

[1.10 Нормалі мешу 18](#_Toc168943554)

[1.11 Ерозія поверхні 18](#_Toc168943555)

[2 МАТЕМАТИЧНА МОДЕЛЬ 22](#_Toc168943556)

[2.1 Генерація сітки вершин для мешу 22](#_Toc168943557)

[2.2 Генерація трикутників поверхні 23](#_Toc168943558)

[2.3 Визначення нормалі згідно з форматом поверхні 25](#_Toc168943559)

[2.4 Рух частинки 26](#_Toc168943560)

[2.5 Ерозія поверхні 27](#_Toc168943561)

[3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 29](#_Toc168943562)

[3.1 Платформа Unity 29](#_Toc168943563)

[3.2 Render pipeline 30](#_Toc168943564)

[3.3 Shader graph 31](#_Toc168943565)

[4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 33](#_Toc168943566)

[ВИСНОВКИ 38](#_Toc168943567)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39](#_Toc168943568)

[ДОДАТКИ 41](#_Toc168943569)

[Додаток А 41](#_Toc168943570)

[Додаток Б 44](#_Toc168943571)

[Додаток В 46](#_Toc168943572)

# 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 є створення реалістичних і деталізованих поверхонь. Ландшафтні поверхні найчастіше складаються з двовимірного масиву значень, які відображають висоту ландшафту в кожній точці координат. Створення таких поверхонь має велике значення для забезпечення реалістичності візуалізації у віртуальних світах.

Класичний спосіб створення ландшафтної поверхні передбачає вибір розмірності двовимірного масиву та заповнення всіх значень вручну або за допомогою спеціальних інструментів. Існує безліч програмних засобів, які спрощують цей процес, дозволяючи користувачам малювати карти висот за допомогою пензлів і інших інструментів. Такі карти висот потім зберігаються і можуть бути використані для візуалізації ландшафту.

Проте такий підхід має значні недоліки. Кожну поверхню необхідно створювати вручну, що потребує багато часу і зусиль. Крім того, для отримання високої деталізації і варіативності ландшафту потрібно генерувати велику кількість даних, що може бути ресурсоємним процесом. Зокрема, створення великих і деталізованих ландшафтів вимагає значних обчислювальних ресурсів для їх обробки та зберігання.

Альтернативний підхід до створення ландшафту, який вирішує багато з цих проблем, це процедурна генерація поверхонь. Процедурна генерація є інноваційним і ефективним методом, що дозволяє автоматично створювати складні і реалістичні ландшафти за допомогою спеціальних алгоритмів та математичних моделей.

Цей підхід ґрунтується на використанні різних алгоритмів генерації шуму, фрактальних алгоритмів та симуляцій фізичних процесів, що дозволяє досягти високого рівня деталізації та природності у створенні ландшафтів. Замість ручного моделювання кожного елемента ландшафту, процедурна генерація дозволяє створювати великі території з високою роздільною здатністю та варіативністю за допомогою визначених правил та параметрів.

Однією з головних переваг процедурної генерації є можливість збереження всього ландшафту у вигляді невеликого набору параметрів або seed, що значно зменшує вимоги до зберігання даних. Це також дозволяє швидко змінювати та відтворювати ландшафт, змінюючи лише початкові параметри або seed, що робить цей метод дуже гнучким та ефективним.

Процедурна генерація також забезпечує високу швидкість створення ландшафтів. Завдяки використанню ефективних алгоритмів, процес генерації може бути автоматизований та виконуватися в реальному часі, що особливо корисно для застосувань у ігровій індустрії та віртуальній реальності, де важлива динамічна зміна навколишнього середовища.

Окрім цього, процедурна генерація дозволяє створювати дуже великі та складні світи, які можуть бути практично нескінченними. Це особливо важливо для сучасних комп'ютерних ігор та симуляцій, де гравці або користувачі можуть досліджувати великі території з різноманітними біомами та географічними особливостями.

Таким чином, процедурна генерація поверхонь є потужним інструментом для створення реалістичних та деталізованих ландшафтів. Використання алгоритмів та математичних моделей дозволяє автоматизувати процес створення, забезпечуючи високу ефективність, гнучкість та якість кінцевого результату. Цей підхід значно спрощує роботу розробників та художників, дозволяючи їм зосередитися на творчих аспектах проектування, залишаючи технічні деталі генерації алгоритмам.

Але такі поверхні не завжди є реалістичними. Реальні ландшафти генеруються за допомогою фізичних процесів, таких як ерозія, зсув літосферних плит. Тож альтернативний шлях для генерації поверхонь – зробити симуляцію на основі фізичного процесу. Один із прикладів такого генерації - це симуляції ерозії на поверхні карти висот.

Алгоритми ерозії поверхонь імітують природні процеси ерозії, такі як вплив води, вітру і температури, на формування ландшафту. Вони дозволяють створювати реалістичні і динамічні поверхні, які виглядають природно і переконливо. Ерозія поверхонь додає деталізацію і складність до процедурно згенерованих ландшафтів, роблячи їх більш реалістичними.

Цей принцип генерації поверхонь має обмежену кількість сфер, де він може застосовуватися. Тоді як при генерації через випадкові шуми ми можемо пере генерувати будь-яку точку у будь-який час, при симуляції ми повинні мати готову базову карту висот, на які вже накладається симуляція. Оскільки час, необхідний на симуляцію, сильно залежить від розміру поверхні, та самої симуляції і, зазвичай, не виконується у межах 2мс, цей принцип не може використовуватися для миттєвої генерації поверхні і повинен бути підготовлений і збережений у пам'яті заздалегідь.

# ПОСТАНОВКА ЗАДАЧІ

Метою цієї роботи є розробка та реалізація ефективної системи поліпшення ландшафту з використанням алгоритмів ерозії поверхонь. У рамках цієї роботи будуть вивчені існуючі методи ерозії, проведено їх порівняльний аналіз та вибрано найбільш оптимальні підходи для розробки системи.

Основними завданнями цієї роботи є:

* Вивчення теоретичних основ і існуючих підходів до ерозії поверхонь.
* Розробка та реалізація системи поліпшення карти висот з використанням алгоритмів ерозії поверхонь.
* Реалізувати систему для побудови ландшафту у програмному забезпечені Unity та два шейдери для аналізу поверхні.
* Розробка системи для експортування зображень у вигляді карти висот.

Результати цієї роботи можуть мати практичне застосування в галузі комп'ютерної графіки, допомагаючи створювати більш реалістичні та деталізовані ландшафти для ігор, фільмів, віртуальної реальності та інших застосувань. Крім того, отримані знання і розроблені методи можуть бути використані для подальшого вдосконалення систем генерації поверхонь і створення ще більш вражаючих візуальних ефектів.

# 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 не детермінованість 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 1.1 – 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).

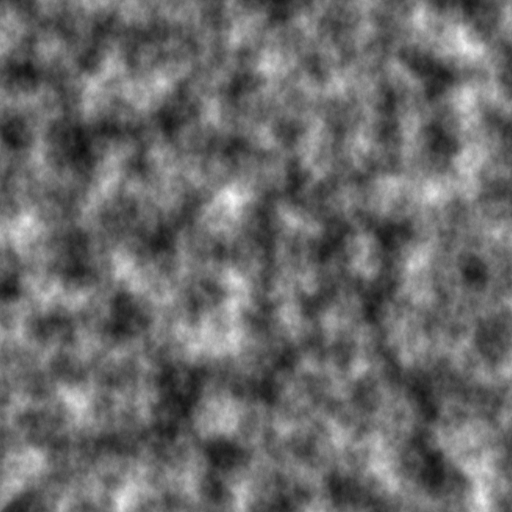


Рисунок 1.2 – Результат виконання алгоритму шуму Перліна

## 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.

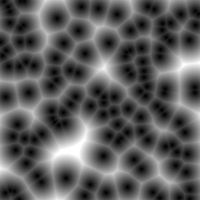


Рисунок 1.4 – Шум Вороного

## 1.7 Комбінація рівней шуму

Комбінація рівнів шуму є важливим прийомом у генерації складних текстур та ландшафтів [рис 1.5](#Рисунок_1_5). Зазвичай для цього використовують різні види псевдовипадкових шумів, такі як шум Перліна або шум Симплекс. Комбінуючи кілька рівнів шуму з різними частотами та амплітудами, можна створити складні, багатошарові текстури, які виглядають більш природно.

 Изображение выглядит как шаблон, дизайн, серый, ткань

Автоматически созданное описание Изображение выглядит как шаблон, ткань

Автоматически созданное описание

Рисунок 1.5 – Три рівні шуму Симплекс, окремо один від одного

Основний принцип полягає в додаванні шумів з різними параметрами. Наприклад, можна додати високочастотний шум для створення дрібних деталей і низькочастотний шум для великих структур [рис 1.6](#Рисунок_1_6). Це створює ефект фрактальності, коли дрібні деталі накладаються на великі форми, що імітує природні ландшафти.

Процедура комбінації шумів включає:

1. Генерацію базового шару низькочастотного шуму.
2. Додавання середньочастотного шуму з меншою амплітудою.
3. Накладення високочастотного шуму для деталізації.

Кінцевий результат – це багатошаровий ландшафт, що виглядає реалістично завдяки поєднанню різних рівнів шуму.



Рисунок 1.6 – Комбінований шум

## 1.8 Принципи формування ландшафту

Формування ландшафту – це процес створення реалістичних тривимірних моделей поверхні землі або іншої планети. Основні принципи формування ландшафту включають використання псевдовипадкових шумів, фрактальних алгоритмів та процедурного моделювання. Ці методи дозволяють створити складні та реалістичні ландшафти з мінімальними витратами ресурсів.

Основні етапи формування ландшафту:

1. Генерація висотної мапи: використання псевдовипадкових шумів для створення основної структури ландшафту.
2. Застосування ерозійних алгоритмів: симуляція водної та вітрової ерозії для створення природних форм.
3. Додавання деталей: використання додаткових шарів шуму для створення дрібних деталей, таких як скелі та западини.
4. Текстурування: накладання текстур на поверхню для створення реалістичного вигляду.

Ці принципи дозволяють створювати різноманітні ландшафти, від гірських хребтів до пустель та рівнин, забезпечуючи високу якість та реалістичність результату.

## 1.9 Меш. Загальні поняття

Меш (mesh) – це базовий елемент тривимірної графіки, що представляє поверхню об'єкта за допомогою вершин, ребер та граней. Кожен меш складається з набору вершин (точок у просторі), з'єднаних ребрами, які утворюють грані (трикутники або полігони) [рис 1.7](#Рисунок_1_7). Меші використовуються для моделювання будь-яких тривимірних об'єктів, від простих форм до складних моделей.

Основні компоненти мешу:

1. Вершини (vertices): точки в тривимірному просторі, що визначають форму мешу.
2. Ребра (edges): лінії, що з'єднують вершини.
3. Грані (faces): полігони, утворені з ребер. Найчастіше використовуються трикутники через їх простоту і стабільність у рендерингу.

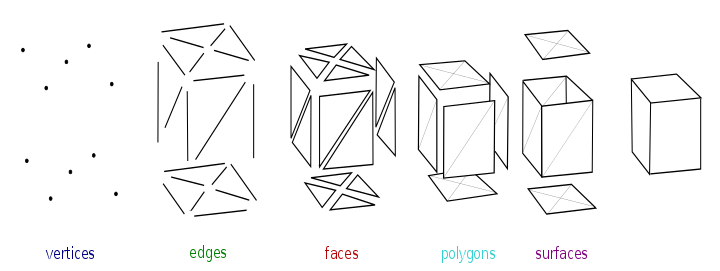


Рисунок 1.7 – Принципи побудови мешу

Меші можуть бути створені вручну або за допомогою процедурних алгоритмів. Вони є основою для текстурування, освітлення та анімації в тривимірних сценах.

## 1.10 Нормалі мешу

Нормалі мешу – це вектори, перпендикулярні до поверхні граней мешу. Вони використовуються для визначення освітлення та тіней на поверхні об'єкта. Кожна вершина та грань мешу має свою нормаль, яка впливає на те, як світло взаємодіє з поверхнею.

Основні аспекти нормалей:

1. Нормалі граней (face normals): перпендикулярні до кожної грані. Використовуються для плоского затінення.
2. Нормалі вершин (vertex normals): обчислюються як середнє значення нормалей суміжних граней. Використовуються для згладженого затінення.

Правильне визначення нормалей є критично важливим для реалістичного відображення освітлення на поверхні об'єкта. Вони допомагають створювати гладкі переходи між гранями, забезпечуючи високу якість візуалізації.

## 1.11 Ерозія поверхні

Ерозія поверхні – це ключовий процес у моделюванні природних ландшафтів, який додає реалістичні деталі та динаміку поверхні. Ерозія імітує вплив природних явищ, таких як дощ, вітер та текуча вода, на формування та зміну поверхні. Мій алгоритм, який генерує одну каплю за раз, є прикладом гідравлічної ерозії, що є однією з найпоширеніших методів ерозії у комп'ютерному моделюванні.

Результати виконання представлені на [рис 1.8](#Рисунок_1_8).



Рисунок 1.8 – Карта висот після виконання алгоритму ерозії

Гідравлічна ерозія моделює взаємодію води з поверхнею, включаючи процеси депонування та транспортування матеріалів.

Основні етапи гідравлічної ерозії включають:

1. Генерація каплі: Капля води падає на поверхню в певній точці.
2. Ерозія: Капля збирає частинки матеріалу з поверхні, що викликає ерозію в цій точці.
3. Транспортування: Капля рухається поверхнею під дією гравітації, збираючи та депонуючи матеріал на своєму шляху.
4. Депонування: Коли капля більше не може транспортувати матеріал, вона депонує його на новому місці.

Алгоритм гідравлічної ерозії виконується наступними кроками:

1. Генерація каплі: На початку кожної ітерації створюється капля в випадковій точці на поверхні.
2. Рух каплі:
   * Визначення напрямку руху каплі на основі градієнту висоти поверхні.
   * Рух каплі до наступної точки у напрямку максимального зниження висоти.
3. Ерозія матеріалу:
   * Капля збирає матеріал з поточної точки пропорційно до нахилу поверхні та кількості води в каплі.
   * Втрата матеріалу з поверхні моделюється шляхом зменшення висоти в точці.
4. Транспортування матеріалу:
   * Капля переносить зібраний матеріал до нової точки.
   * Транспортування залежить від швидкості руху каплі, яка зменшується зі збільшенням ваги каплі.
5. Депонування матеріалу:
   * Капля депонує матеріал у точках зниження швидкості або коли вода в каплі випаровується.
   * Збільшення висоти в точці депонування моделює накопичення матеріалу.
6. Повторення процесу: Процес повторюється для заданої кількості ітерацій або поки капля не втратить всю воду.

Основні параметри, що впливають на процес ерозії:

* Кількість капель: Визначає, скільки разів процес буде повторюватися.
* Інтенсивність ерозії: Визначає, скільки матеріалу капля може зібрати з поверхні.
* Швидкість випаровування: Визначає, як швидко вода випаровується з каплі.
* Максимальна кількість матеріалу: Обмежує кількість матеріалу, яку капля може транспортувати.

Генерація поверхні через гідравлічну ерозію відрізняється по принципу від генерації за допомогою шумів. Хоч вона і виправляє деякі недоліки генерацію через псевдовипадкові шуми, вона має свої недоліки:

З переваг можна виділити:

* + Реалістичність ­ Імітує природні процеси формування ландшафту.
  + Деталізація: Додає деталі та складність поверхні.

Недоліки:

* + Обчислювальна складність: Вимагає значних обчислювальних ресурсів.
  + Параметризація: Потребує точного налаштування параметрів для досягнення бажаних результатів.

Ерозія поверхні знаходить широке застосування в різних галузях:

* Ігрова індустрія: Використовується для створення реалістичних і динамічних ландшафтів.
* Анімація та візуальні ефекти: Створення природних текстур і динамічних середовищ.
* Наукові дослідження: Моделювання геологічних процесів та їх впливу на навколишнє середовище.

# 2 МАТЕМАТИЧНА МОДЕЛЬ

## 2.1 Генерація сітки вершин для мешу

Для генерації мешу на площині XOZ використовують прямокутну сітку вершин із двовимірними координатами.

Меш, який використовується у поточній симуляції, генерується на XOZ площині та є прямокутною сіткою, де двовимірні координати вершин визначаються індексами та [рис 2.1](#Рисунок_2_1). Кожна вершина має координати , де та – це індекси рядка та стовпця відповідно.­ Координата відповідає просторовій координаті , а – .

(2.1)

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Рисунок 2.1 – Представлення вершин у двовимірному масиві

Хоча ми і використовуємо двовимірний масив для маніпулювання вершинами мешу, зазвичай графічні фреймворки читають такі дані послідовно. Тож нам необхідно перевести наш двовимірний масив точок у одновимірний формат [рис 2.2](#Рисунок_2_2).

Изображение выглядит как текст, снимок экрана, число, диаграмма

Автоматически созданное описание

Рисунок 2.2 – Розташування вершин у пам'яті

## 2.2 Генерація трикутників поверхні

Оскільки поверхня буде деформуватися з плином часу, ми не зможемо знаходити нормалі поверхні математично, як це робилося-б при генерації за допомогою псевдовипадкових шумів. У випадку динамічної поверхні, ми будемо використовувати властивості поверхні, аби знайти нормаль поверхні у довільній точці. Тому, при побудові поверхні мешу, окрему увагу треба виділити принципу побудови окремих трикутників.

Оскільки , а поверхня являє собою квадрат, її можна поділити на множину під-квадратів кожна з яких використовує наступні вершини . Тобто, для довільно вибраного PpS, кількість під-квадратів sqr по кожній з осей координат дорівнює , а загальна кількість .

Наприклад, при меші розміром 5х5 [рис 2.3](#Рисунок_2_3), sqr(0, 0) буде включати у себе наступні точки у форматі двовимірного масиву: (0, 0), (0, 1), (1, 0), (1, 1). Ті ж самі точки у форматі одновимірного масиву: 0, 1, 5, 6.

Изображение выглядит как шаблон, диаграмма, линия, снимок экрана

Автоматически созданное описание

Рисунок 2.3 – Побудова граней у меші

Кожен під-квадрат складається з двох трикутників. Беручи окремий під-квадрат, трикутники у ньому будуються таким чином, аби прямі кути у трикутниках знаходилися на мінімальній та максимальній точці під-квадрату. Трикутники зберігаються, як послідовний одновимірний масив індексів точок, де кожні 3 точки – один трикутник. Точки обираються за годинниковою стрілкою, аби поверхня відрисовувалася зі сторони Y+.

|  |  |
| --- | --- |
|  | (2.3) |
|  | (2.4) |

Наприклад, беручи окремий під-квадрат [рис. 2.4](#Рисунок_2_4), трикутники запишуться наступним чином: .

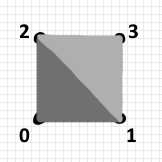


Рисунок 2.4 – Під-квадрат з мешу, розділений на 2 трикутники

## 2.3 Визначення нормалі згідно з форматом поверхні

Оскільки у рамках одного під-квадрату ми маємо два трикутника, якщо точка знаходиться у рамках одного з двох трикутників, нам достатньо знайти його нормаль.

Изображение выглядит как линия, треугольник

Автоматически созданное описание

Рисунок 2.5 – Перший випадок з нижнім трикутником

Якщо точка попадає до нижнього трикутника, ми будуємо два вектори та та знаходимо векторний добуток, використовуючи вектори проти годинникової стрілки [рис. 2.5](#Рисунок_2_5).

|  |  |
| --- | --- |
| *, де N* – нормаль у точці. | (2.5) |

Якщо точка попадає до верхнього трикутника [рис. 2.6](#Рисунок_2_6), ми також будуємо два вектори та та знаходимо векторний добуток, використовуючи вектори проти годинникової стрілки.

|  |  |
| --- | --- |
|  | (2.6) |

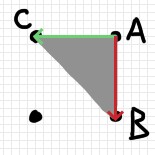


Рисунок 2.6 – Другий випадок з верхнім трикутником

## 2.4 Рух частинки

Рух частинки на поверхні моделюється за допомогою класичної механіки. Коротко, положення частинки змінюється швидкістю , яка змінюється прискоренням .

|  |  |
| --- | --- |
|  | (2.7) |
|  | (2.8) |

Відомо, що сила дорівнює масі, помноженій на прискорення:

|  |  |
| --- | --- |
|  | (2.9) |

Частинки відчувають прискорення вниз через гравітацію, але знаходяться на поверхні, що робить неможливим вертикальне прискорення. Натомість частинка відчуває силу вздовж поверхні, пропорційну нормалі поверхні.

Отже, можна сказати, що прискорення пропорційне вектору нормалі поверхні , поділеному на масу частинки:

|  |  |
| --- | --- |
|  | (2.10) |

Якщо маса дорівнює об’єму, помноженому на густину, ми отримаємо повну систему для руху частинок за допомогою класичної механіки.

## 2.5 Ерозія поверхні

Процес седиментації фізично відбувається через перенесення осаду з ґрунту в частинку і назад у будь-якій точці, де знаходиться частинка.

У хімічній інженерії зазвичай описують перенесення маси, тобто зміну маси/осаду в часі між ґрунтом і краплею за допомогою коефіцієнтів перенесення маси.

Перенесення маси пропорційне різниці між концентрацією та рівноважною концентрацією ​:

|  |  |
| --- | --- |
|  | (2.11) |

Ця різниця між рівноважною та фактичною концентрацією часто називається "рушійною силою".

Якщо рівноважна концентрація вища за поточну концентрацію, частинка поглинає осад. Якщо вона нижча, частинка втрачає осад. Якщо вони рівні, змін не відбувається.

Визначення нашої рівноважної концентрації повністю визначає нашу систему. Залежно від визначення, наша система буде демонструвати різні динаміки седиментації. Для моєї реалізації рівноважна концентрація вища, якщо ми рухаємося вниз по схилу і якщо ми рухаємося швидше, і пропорційна об’єму частинки.

# 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 Mono. 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 ландшафтної поверхні з використанням принципів ерозії поверхні. Ландшафт будується на квадратному меші розміром . Побудова поверхні відбувається через компонент «Terrain Chunk Generator» [рис. 4.1](#Рисунок_4_1).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 4.1 – Модуль генерації поверхні

Компонент знаходиться на об'єкті на сцені та є інтерфейсом програми. Він ділиться на частину з посиланнями на технічні класи, декілька інтерактивних частин, які і використовуються для налаштування поверхні та кнопки генерації поверхні.

Перша частина існує для налаштування стадії пре-генерації поверхні через псевдовипадкові шуми. Програма використовує шум Симплекс і має можливість налаштовувати кількість рівней шуму. Один рівень шуму складається з трьох параметрів:

* Scale – це величина, на яку домножається координата, аби отримати більш рівномірну поверхню. Визначається, як
* Displacement – це зсув координат на довільний вектор заради різноманітності рівней між собою
* Influence – вплив рівня шуму на фінальний результат. Визначається, як

Изображение выглядит как размытие, черно-белый, черный, белый

Автоматически созданное описание Изображение выглядит как шаблон, дизайн, серый, ткань

Автоматически созданное описание Изображение выглядит как шаблон, ткань

Автоматически созданное описание

Рисунок 4.2 – три рівні шуму з різними параметрами

Рівні шуму, продемонстровані на [рис. 4.2](#Рисунок_4_2) будувалися послідовно з рівней шуму з компоненту [рис. 4.1](#Рисунок_4_1) послідовно, результуючим шумом є [рис. 4.3](#Рисунок_4_3).



Рисунок 4.3 – шум, комбінований з трьох рівнів

Друга інтерактивна частина компоненту – налаштування властивостей поверхні, таких як розмір, кількість точок по кожній з осей, кількість ітерацій симуляції.

Також, при кожній генерації ландшафту, програма видає зображення карти висот з цього ландшафту. Керування цією системою відбувається через компонент «Height Texture Drawer», де ми можемо налаштувати колір для мінімальної та максимальної висоти [рис. 4.4](#Рисунок_4_4). Проміжні кольори лінійно інтерполюють між ними.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 4.4 – компонент для налаштування кольорів

Після накладання шуму, поверхня виглядає наступним чином [рис. 4.5](#Рисунок_4_5). Це одна і та сама поверхня, але з різним шейдером. Ліворуч накладений шейдер, який відображає два кольори: зелений та сірий, які залежать від кута між нормалью поверхні та вектором . Праворуч знаходиться шейдер, який розмальовує поверхню, лінійно інтерполюючи висоту між двома кольорами, один з яких відповідає найнижчій точці, інший найвишчій.

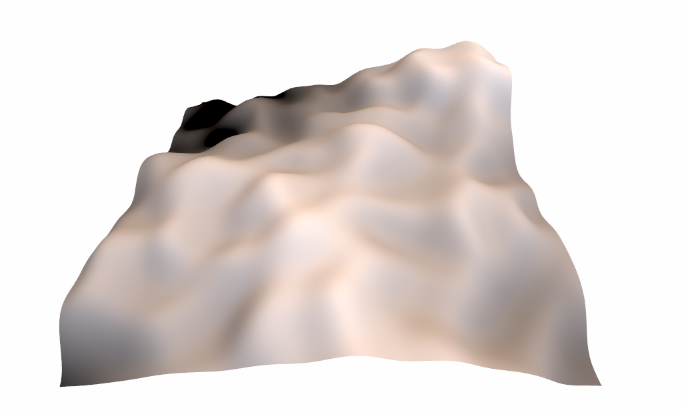
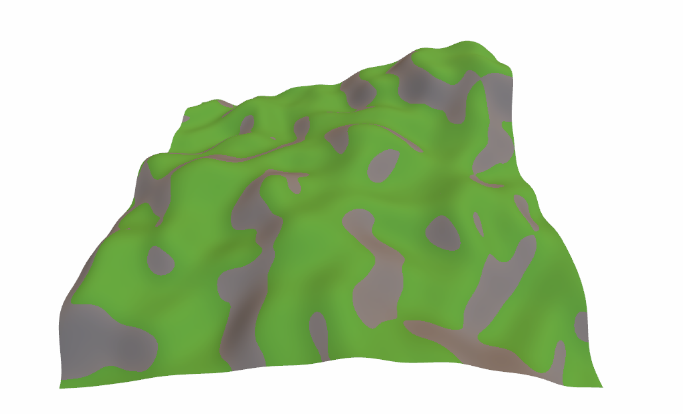


Рисунок 4.5 – Поверхня після накладання шуму

Ця поверхня використовується, як вхідні дані для симуляції ерозії. Вона буде динамічно змінюватись із кожною новою краплею. Після виконання симуляції на цій поверхні, вона змінилась наступним чином [рис. 4.6](#Рисунок_4_6).

Изображение выглядит как зеленый

Автоматически созданное описаниеИзображение выглядит как аксессуар, мешок

Автоматически созданное описание

Рисунок 4.6 – поверхня після виконання симуляції

Відображена у файлі карта висот виглядає наступним чином. Початкова поверхня знаходиться зліва, поверхня після симуляції знаходиться справа.

Рисунок 4.7 – Представлення у виді карти висот.

Далі представлено послідовність станів поверхні при ерозії поверхні з кроком у 5 ітерацій, де одна ітерація – 30000 крапель.

Изображение выглядит как снимок экрана, Симметрия, черно-белый, Красочность

Автоматически созданное описание

Рисунок 4.8 – Стани поверхні з різницею у 150000 ітерацій

# ВИСНОВКИ

За результатом виконання цієї роботи, була розроблена та реалізована система поліпшення ландшафту з використанням алгоритму ерозії поверхні. У рамках цієї роботи було вивчено існуючі методи ерозії, та вибрано метод симуляції гідравлічної ерозії по індивідуальним краплям.

За результатом виконання роботи було виконано наступні пункти:

* Вивчено теоретичні основи, існуючи підходи до ерозії поверхні.
* Розроблено та реалізовано систему поліпшення карти висот з використанням алгоритму ерозії поверхні.
* Реалізовано систему для побудови ландшафту у програмному забезпечені Unity та два різних шейдери для аналізу поверхні.
* Розроблено систему для експортування зображень у вигляді карти висот.

Результати цієї роботи можуть мати практичне застосування в галузі комп'ютерної графіки, допомагаючи створювати більш реалістичні та деталізовані ландшафти для ігор, фільмів, віртуальної реальності та інших застосувань. Отримані результати та знання можуть бути використані для подальшого вдосконалення систем генерації поверхі. Наступним шляхом для поліпшення може стати система генерації ерозії у реальному часі, завдяки одночасному моделюванню крапель для всієї сітки і переносу алгоритму на GPU.

# 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.
9. Проста гідравлічна ерозія на основі частинок - Режим доступу: <https://nickmcd.me/2020/04/10/simple-particle-based-hydraulic-erosion/> - Simple Particle-Based Hydraulic Erosion
10. Гідравлічна ерозія - Режим доступу: [https://www.youtube.com/watch?v=eaXk97ujbPQ](https://www.youtube.com/watch?v=eaXk97ujbPQ%20) – Coding Adventure: Hydraulic Erosion
11. Процедурна генерація через гідравлічну ерозію - Режим доступу: <https://www.reddit.com/r/Unity3D/comments/lamy3q/procedural_terrain_generation_tool_with_hydraulic> - Procedural Terrain Generation Tool With Hydraulic Erosion - Jobs System + Burst
12. Градієнт зображення - Режим доступу: [https://en.wikipedia.org/wiki/Image\_gradient](https://en.wikipedia.org/wiki/Image_gradient%20) - Image gradient

# ДОДАТКИ

## Додаток А

using System;  
using System.Collections.Generic;  
using System.Diagnostics;  
using DefaultNamespace;  
using NoiseTest;  
using Services.NoiseGeneration.Impls;  
using UnityEditor;  
using UnityEngine;  
using UnityEngine.Serialization;  
using Debug = UnityEngine.Debug;  
using Random = UnityEngine.Random;  
  
namespace Services.PlaneGeneration.Impls{  
 public class TerrainChunkGenerator : MonoBehaviour{  
 [SerializeField] private TerrainChunk planeAsset;  
 [SerializeField] private PerlinNoiseGenerator perlinNoiseGenerator;  
 [SerializeField] private ErosionCellSimulator erosionCellSimulator;  
 [SerializeField] private HeightTextureDrawer heightTextureDrawer;  
 [SerializeField] private GausianBlur gausianBlur;  
 [Header("Resimulation")]  
 public TerrainChunk SimulatableTerrainChunk = null;  
   
 [Serializable]  
 public struct NoiseLayer{  
 public Vector2 Scale;  
 public Vector2 Displacement;  
 public float Influence; }  
  
 [SerializeField] private List<NoiseLayer> noiseLayers;  
   
 [SerializeField] private bool useErrosion;  
 [SerializeField] private bool applyGaussianBlur;  
   
 [SerializeField] private float planeSize;  
 [SerializeField] private int resolution;  
 [SerializeField] private int iterationsCount;  
 [SerializeField] private MeshDataGenerator meshDataGenerator;  
 [SerializeField] private Transform terrainChunksHolder;  
 public TerrainChunk LastGenerated;  
   
 public void ApplyPerlin(ref Vector3[][] grid) {  
 var openSimplexNoise = new OpenSimplexNoise(3248);  
   
 for (var z = 0; z < resolution; ++z)  
 for (var x = 0; x < resolution; ++x) {  
 var point = grid[z][x];  
  
 double height = 0;  
 for (var i = 0; i < noiseLayers.Count; ++i) {  
 var noiseLayer = noiseLayers[i];  
 height += openSimplexNoise.Evaluate((point.x + noiseLayer.Displacement.x) / noiseLayer.Scale.x,  
 (point.z + noiseLayer.Displacement.y) / noiseLayer.Scale.y) \*  
 noiseLayer.Influence; }  
 point.y = (float)height;  
 grid[z][x] = point; }}  
  
 public TerrainChunk GeneratePlane(){  
 var stopwatch = new Stopwatch();  
 stopwatch.Start();  
   
 var meshData = meshDataGenerator.GenerateMesh(resolution, planeSize);  
   
 ApplyPerlin(ref meshData.Vertices);  
   
 var planeObject = Instantiate(planeAsset, terrainChunksHolder);  
 var newMesh = new Mesh();  
  
  
 if (useErrosion) {  
 planeObject.name += " Erroded";  
  
 erosionCellSimulator.SetupSimulator(meshData.Vertices);  
  
 for (var i = 0; i < iterationsCount; ++i) {  
 var position = new Vector2(Random.Range(0f, resolution - 1), Random.Range(0f, resolution - 1));  
 erosionCellSimulator.SimulateDroplet(position); }}  
 if (applyGaussianBlur)  
 gausianBlur.ApplyGaussianBlur(ref meshData.Vertices, meshData.Resolution);  
  
 heightTextureDrawer.GenerateTexture(meshData.Vertices, resolution);  
   
 var heightMapLinear = new Vector3[resolution \* resolution];  
  
 for (var z = 0; z < resolution; ++z)  
 for (var x = 0; x < resolution; ++x)  
 heightMapLinear[z \* resolution + x] = meshData.Vertices[z][x];  
  
 newMesh.vertices = heightMapLinear;  
 newMesh.triangles = meshData.Triangles;  
 newMesh.RecalculateNormals();  
 newMesh.RecalculateTangents();  
 newMesh.RecalculateBounds();  
 newMesh.RecalculateUVDistributionMetrics();  
   
 planeObject.MeshFilter.mesh = newMesh;  
 planeObject.MeshData = meshData;  
  
 LastGenerated = planeObject;  
  
 stopwatch.Stop();  
 Debug.Log($"Time: {Math.Round(stopwatch.ElapsedMilliseconds / 1000f, 2)} sec");  
   
 return planeObject; }  
   
 public TerrainChunk GeneratePlane(TerrainChunk simulatableTerrainChunk) {  
 if (!simulatableTerrainChunk)  
 return null;  
   
 var stopwatch = new Stopwatch();  
 stopwatch.Start();  
   
 var meshData = simulatableTerrainChunk.MeshData;  
 var newMesh = simulatableTerrainChunk.MeshFilter.mesh;  
   
 if (useErrosion) {  
 erosionCellSimulator.SetupSimulator(meshData.Vertices);  
  
 for (var i = 0; i < iterationsCount; ++i) {  
 var position = new Vector2(Random.Range(0f, resolution - 1), Random.Range(0f, resolution - 1));  
 erosionCellSimulator.SimulateDroplet(position); }}  
  
 heightTextureDrawer.GenerateTexture(meshData.Vertices, resolution);  
   
 var heightMapLinear = new Vector3[resolution \* resolution];  
  
 for (var z = 0; z < resolution; ++z)  
 for (var x = 0; x < resolution; ++x)  
 heightMapLinear[z \* resolution + x] = meshData.Vertices[z][x];  
  
 newMesh.vertices = heightMapLinear;  
 newMesh.triangles = meshData.Triangles;  
 newMesh.RecalculateNormals();  
 newMesh.RecalculateTangents();  
 newMesh.RecalculateBounds();  
 newMesh.RecalculateUVDistributionMetrics();  
   
 simulatableTerrainChunk.MeshFilter.mesh = newMesh;  
  
 stopwatch.Stop();  
 Debug.Log($"Time: {Math.Round(stopwatch.ElapsedMilliseconds / 1000f, 2)} sec");  
   
 return simulatableTerrainChunk; }}  
  
 [CustomEditor(typeof(TerrainChunkGenerator))]  
 public class PlaneGeneratorEditor : Editor{  
 public override void OnInspectorGUI(){  
 base.OnInspectorGUI();  
 if (GUILayout.Button("RegeneratePlane")){  
 var targetT = (target as TerrainChunkGenerator);  
  
 if (targetT.SimulatableTerrainChunk){  
 targetT.GeneratePlane(targetT.SimulatableTerrainChunk);  
 return; }  
   
 targetT.GeneratePlane();  
 }}}}

## Додаток Б

using Services.PlaneGeneration.Impls;

using UnityEditor;

using UnityEngine;

namespace DefaultNamespace{

public class TerrainChunk : MonoBehaviour{

public bool EnableDebugGizmos { get; set; }

public bool EnableVertexPreview { get; set; }

public bool EnableVertexOrderPreview { get; set; }

public float VertexPreviewSphereRadius { get; set; }

public float VertexOrderPreviewYOffset { get; set; }

[Header("Components")]

[SerializeField] private MeshFilter meshFilter;

public MeshFilter MeshFilter => meshFilter;

public MeshData MeshData { get; set; }

private void OnDrawGizmos(){

if (!EnableDebugGizmos)

return;

var mesh = meshFilter.mesh;

var vertices = mesh.vertices;

if (EnableVertexPreview) {

Gizmos.color = Color.red;

foreach (var vertex in vertices)

Gizmos.DrawSphere(vertex + transform.position, VertexPreviewSphereRadius); }

if (EnableVertexOrderPreview) {

var flattenMatrix = Matrix4x4.Scale(new Vector3(1, 0, 1));

var translateVector = new Vector4(0, VertexOrderPreviewYOffset, 0, 0);

for (var i = 0; i < vertices.Length - 1; ++i) {

Gizmos.color = Color.Lerp(Color.red, Color.blue, (float)i / vertices.Length);

var firstPoint = flattenMatrix \* vertices[i] + translateVector;

var secondPoint = flattenMatrix \* vertices[i + 1] + translateVector;

Gizmos.DrawRay(firstPoint, secondPoint - firstPoint); }}}

public void SetMesh(Mesh mesh) {

meshFilter.mesh = mesh; }}

[CustomEditor(typeof(TerrainChunk))]

public class TerrainChunkEditor : Editor{

private bool enableDebugGizmos;

private bool enableVertexPreview;

private float vertexPreviewRadius;

private bool enableVertexOrderPreview;

private float vertexOrderPreviewYOffset;

private TerrainChunk \_target;

private void Awake() =>

\_target = (TerrainChunk)target;

public override void OnInspectorGUI(){

base.OnInspectorGUI();

DebugGizmos();}

private void DebugGizmos(){

var result = GUILayout.Toggle(enableDebugGizmos, "Enable Debug Gizmos");

\_target.EnableDebugGizmos = result;

enableDebugGizmos = result;

if (!result)

return;

VertexPreview();

VertexOrderPreview();}

private void VertexPreview(){

var result = GUILayout.Toggle(enableVertexPreview, "Enable Vertex Preview");

enableVertexPreview = result;

\_target.EnableVertexPreview = result;

if(!result)

return;

var radius = GUILayout.HorizontalSlider(vertexPreviewRadius, 0, 2);

GUILayout.Space(15);

vertexPreviewRadius = radius;

\_target.VertexPreviewSphereRadius = vertexPreviewRadius; }

private void VertexOrderPreview(){

var result = GUILayout.Toggle(enableVertexOrderPreview, "Enable Vertex Order Preview");

enableVertexOrderPreview = result;

\_target.EnableVertexOrderPreview = result;

if(!result)

return;

var offset = GUILayout.HorizontalSlider(vertexOrderPreviewYOffset, 0, 10);

GUILayout.Space(15);

vertexOrderPreviewYOffset = offset;

\_target.VertexOrderPreviewYOffset = vertexOrderPreviewYOffset; }}}

## Додаток В

using System.Collections.Generic;  
using UnityEngine;  
  
namespace DefaultNamespace{  
 public class ErosionCellSimulator : MonoBehaviour{  
 [SerializeField] float waterDefaultVolume = 1f;  
 [SerializeField] float waterEvaporation = 0.01f;  
 [SerializeField] float gravity = 20f;  
 [SerializeField] float minSlope = -4f;  
 [SerializeField] float erosionSpeed = 0.9f;  
 [SerializeField] float depositionSpeed = 0.02f;  
 [SerializeField] int iterationsCount = 50;  
 [SerializeField] float sedimentCapacity = 0.5f;  
   
 private Vector3[][] \_heightMap;  
 private float[][] \_waterLevel;  
  
 private int \_resolution;  
  
 public Vector3[][] HeightMap => \_heightMap;  
   
 public ErosionCellSimulator(Vector3[][] heightMap){  
 \_heightMap = heightMap;  
  
 \_resolution = heightMap.Length;  
   
 \_waterLevel = new float[heightMap.GetLength(0)][];  
  
 for (var y = 0; y < \_resolution; ++y)  
 \_waterLevel[y] = new float[\_resolution];}  
  
 public void SetupParameters(){  
 SetupWaterLevel(10); }  
  
 public void SetupSimulator(Vector3[][] heightMap) {  
 \_heightMap = heightMap;  
 \_resolution = heightMap.Length;  
 \_waterLevel = new float[heightMap.GetLength(0)][];  
  
 for (var y = 0; y < \_resolution; ++y)  
 \_waterLevel[y] = new float[\_resolution]; }  
  
 public bool IsPointInBounds(Vector2Int point, float pointRadius, int resolution) {  
 var pointLeftX = point.x - pointRadius;  
 var pointRightX = point.x + pointRadius;  
 var pointTopY = point.y - pointRadius;  
 var pointBottomY = point.y + pointRadius;  
  
 return pointLeftX >= 0 &&  
 pointRightX < resolution &&  
 pointTopY >= 0 &&  
 pointBottomY < resolution; }  
  
 public Vector2Int[] GetPositionsInRadius(Vector2Int position, int pointRadius, int resolution) {  
 var squareVolume = pointRadius \* pointRadius \* 4;  
 var result = new List<Vector2Int>(squareVolume);  
  
 for (var x = position.x - pointRadius; x < position.x + pointRadius; ++x) {  
 for (var y = position.y - pointRadius; y < position.y + pointRadius; ++y) {  
 var positionInBound = new Vector2Int(x, y);  
  
 if (Vector2Int.Distance(position, positionInBound) <= pointRadius &&  
 IsPointInBounds(positionInBound, 1, resolution))  
 result.Add(positionInBound);} }  
   
 return result.ToArray(); }  
  
 public struct Droplet{  
 public Vector3 Position;  
 public Vector3 Speed;  
 public float WaterVolume;  
 public float SedimentConcentration;  
 public float SedimentCapacity; }  
  
 Vector3 GetSurfaceNormal(Vector2Int flooredPosition, Vector2 particlePosition) // x = x, y = z {  
 var squareGrid = new Vector3[][] {  
 new Vector3[] {  
 \_heightMap[flooredPosition.y][flooredPosition.x],  
 \_heightMap[flooredPosition.y + 1][flooredPosition.x] },  
 new Vector3[] {  
 \_heightMap[flooredPosition.y][flooredPosition.x + 1],  
 \_heightMap[flooredPosition.y + 1][flooredPosition.x + 1] } };  
  
 var distanceToFloor = Vector2.Distance(flooredPosition, particlePosition);  
 var distanceToCeil = Vector2.Distance(squareGrid[1][1], particlePosition);  
  
  
 if (distanceToFloor < distanceToCeil)  
 return Vector3.Cross(squareGrid[0][1] - squareGrid[0][0], squareGrid[1][0] - squareGrid[0][0])  
 .normalized;  
   
 if (distanceToFloor > distanceToCeil)  
 return Vector3.Cross(squareGrid[1][0] - squareGrid[1][1], squareGrid[0][1] - squareGrid[1][1])  
 .normalized;  
  
 var normal = ((squareGrid[1][0] + squareGrid[0][1] - squareGrid[0][0] - squareGrid[1][1]) / 2).normalized;  
  
 if (normal.y < 0)  
 normal \*= -1;  
  
 return normal; }  
  
 public void SimulateDroplet(Vector2 currentPosition) {  
 var iterations = iterationsCount;  
 var droplet = new Droplet() {  
 Position = new Vector3(currentPosition.x, 0, currentPosition.y),  
 Speed = Vector3.zero,  
 WaterVolume = waterDefaultVolume,  
 SedimentConcentration = 0 };  
  
 var friction = 0.1f;  
  
 while (iterations > 0 && droplet.WaterVolume > 0) {  
 var flooredPosition = Vector2Int.FloorToInt(new Vector2(droplet.Position.x, droplet.Position.z));  
   
 if (flooredPosition.x < 0 ||  
 flooredPosition.x >= \_resolution - 1 ||  
 flooredPosition.y < 0 ||  
 flooredPosition.y >= \_resolution - 1)  
 return;  
  
 var normal = GetSurfaceNormal(flooredPosition, new Vector2(droplet.Position.x, droplet.Position.z));  
   
 //Accelerate particle using newtonian mechanics using the surface normal.  
 var acceleration = normal;  
 droplet.Speed += acceleration; //F = ma, so a = F/m  
 droplet.Position += droplet.Speed;  
 droplet.Speed \*= 1.0f - friction; //Friction Factor  
  
 if (droplet.Position.x <= 0 ||  
 droplet.Position.x > \_resolution - 1 ||  
 droplet.Position.z <= 0 ||  
 droplet.Position.z > \_resolution - 1)  
 return;  
  
 //Compute sediment capacity difference  
 var heightDifference = \_heightMap[flooredPosition.y][flooredPosition.x].y -  
 \_heightMap[(int)droplet.Position.z][(int)droplet.Position.x].y;  
   
 var maxsediment = droplet.WaterVolume \* droplet.Speed.magnitude \* heightDifference;  
  
   
 if (maxsediment < 0.0)  
 maxsediment = 0;  
   
 var sdiff = maxsediment - droplet.SedimentConcentration;  
  
 droplet.SedimentConcentration += depositionSpeed \* sdiff;  
 \_heightMap[flooredPosition.y][flooredPosition.x] -= Vector3.up \* Mathf.Min(  
 droplet.WaterVolume \* depositionSpeed \* sdiff, heightDifference);  
  
 droplet.WaterVolume \*= (1.0f - waterEvaporation);  
  
 --iterations; } }  
  
 public void SimulateStep(){  
 var distance = 0.5f;  
 var newWaterLevel = new float[\_resolution][];  
 var newHeightMap = new float[\_resolution][];  
   
 for (var y = 0; y < \_resolution; ++y) {  
 newWaterLevel[y] = new float[\_resolution];  
 newHeightMap[y] = new float[\_resolution]; }  
  
 for (var y = 1; y < \_resolution - 1; ++y) {  
 for (var x = 1; x < \_resolution - 1; ++x) {  
 var currentHeight = \_heightMap[y][x];  
 var leftHeight = \_heightMap[y][x - 1];  
 var rightHeight = \_heightMap[y][x + 1];  
 var topHeight = \_heightMap[y - 1][x];  
 var bottomHeight = \_heightMap[y + 1][x];  
   
 var currentWaterLevel = \_waterLevel[y][x];  
 var leftWaterLevel = \_waterLevel[y][x - 1];  
 var rightWaterLevel = \_waterLevel[y][x + 1];  
 var topWaterLevel = \_waterLevel[y - 1][x];  
 var bottomWaterLevel = \_waterLevel[y + 1][x];  
  
 //var leftSlope = CalculateSlope(currentHeight, leftHeight, distance);  
 //var rightSlope = CalculateSlope(currentHeight, rightHeight, distance);  
 //var topSlope = CalculateSlope(currentHeight, topHeight, distance);  
 //var bottomSlope = CalculateSlope(currentHeight, bottomHeight, distance);  
  
 // var leftSlope = currentWaterLevel - leftWaterLevel;  
 // var rightSlope = currentWaterLevel - rightWaterLevel;  
 // var topSlope = currentWaterLevel - topWaterLevel;  
 // var bottomSlope = currentWaterLevel - bottomWaterLevel;  
  
 var outflow = 0f;  
 var inflow = 0f;  
  
 //if (leftSlope < 0)  
 // outflow += leftWaterLevel;  
 //  
 //if (rightSlope < 0)  
 // outflow += rightWaterLevel;  
 //  
 //if (topSlope < 0)  
 // outflow += topWaterLevel;  
 //  
 //if (bottomSlope < 0)  
 // outflow += bottomWaterLevel;  
 //  
 //if (leftSlope > 0)  
 // inflow += leftWaterLevel;  
 //  
 //if (rightSlope > 0)  
 // inflow += rightWaterLevel;  
 //  
 //if (topSlope > 0)  
 // inflow += topWaterLevel;  
 //  
 //if (bottomSlope > 0)  
 // inflow += bottomWaterLevel;  
  
 newWaterLevel[y][x] = currentWaterLevel - outflow + inflow;  
 //newHeightMap[y][x] = \_heightMap[y][x] - (Mathf.Abs(outflow) + Mathf.Abs(inflow));  
  
 if (newWaterLevel[y][x] <= 0)  
 newWaterLevel[y][x] = 0; } }  
  
 \_waterLevel = newWaterLevel;  
 //\_heightMap = newHeightMap; }  
  
 private float CalculateSlope(float currentCell, float neighbourCell, float distance) {  
 return (neighbourCell - currentCell) / distance; }  
  
 public void SetupWaterLevel(float constantLevel) {  
 for (var y = 0; y < \_resolution; ++y) {  
 for (var x = 0; x < \_resolution; ++x) {  
 \_waterLevel[y][x] = constantLevel;} } } }}