Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Курсова робота

освітній ступінь – бакалавр

на тему: «**Розробка гри на фреймворку Unity з використанням алгоритмів процедурної генерації**»

Виконав: студент 4-го року навчання,

Спеціальності

122 комп’ютерні науки

Макаренко Б. П.

Керівник Борозенний С. О.

Старший викладач

«____» _____ 20____ р.

Київ – 2022

Київ-2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. Кафедри мультимедійних систем,
доцент, к.ф-м.н.
_____ О.П. Жежерун

(підпис)
“____” _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

Студенту Макаренко Богдану Петровичу факультету інформатики 4 курсу

ТЕМА Розробка гри на фреймворку Unity з використанням алгоритмів процедурної генерації

Зміст ТЧ до курсової роботи:

Календарний план виконання курсової роботи

Вступ

Аналіз предметної області. Постановка завдання курсової роботи

Дослідження та аналіз алгоритмів процедурної генерації в існуючих іграх

Опис реалізації власної гри на Unity

Висновки

Список використаних джерел

Додатки

Дата видачі “____” _____ 2022 р. Керівник _____

(підпис)

Завдання отримав _____

Календарний план виконання курсової роботи

Тема: Розробка віртуального асистента для месенджеру

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Аналіз теми курсової робити	Вересень - грудень 2021р.	
2.	Дослідження підходів процедурної генерації в існуючих іграх	Січень-березень 2022р.	
4.	Розробка власної гри на Unity	Березень-квітень 2022р.	
5.	Написання текстової частини	Квітень 2022р.	
6.	Перегляд праці науковим керівником	Травень 2022р.	
7.	Підготовка до презентації роботи	Травень 2022р.	
8.	Презентація курсової роботи	Травень 2022р.	

Студент Макаренко Б.П _____

Керівник Борозенний С. О. _____

“_____” _____ 2022 р.

ЗМІСТ

ЗМІСТ 4

Анотація 6

Вступ 7

Розділ 1. Аналіз існуючих ігор. Дослідження підходів процедурної генерації. 8

1.1 Diablo 1 8

1.1.1 Загальний опис Diablo 1 8

1.1.2 Кроки генерації рівня 8

1.1.3 Генерація рівнів в етапі Cathedral 9

1.1.4 Алгоритм marching squares 9

1.1.4 Інші етапи в diablo 11

1.2 Minecraft 11

1.2.1 Загальний опис гри Minecraft 11

1.2.2 Основа алгоритму генерації світу 11

1.2.3 Шум Перліна 12

1.2.4 Використання октав для покращення результатів 16

1.2.5 Випадковість результатів 17

1.2.6 Генерація біомів та ландшафту 18

1.3 No Man’s Sky 19

1.3.1 Загальний опис гри No Man’s Sky 19

1.3.2 Використання суперформули 19

1.3.4 Інші використані техніки 20

Розділ 2. Реалізація гри з процедурною генерацією з використанням фреймворку Unity 21

2.1 Тематика та особливості гри 21

2.1 Структура ігрового світу 21

2.2 Модуль для генерації шуму Перліна 22

2.3 Генератор світу 23

2.4 Заготовка тайлу 23

2.5 Зміна дня та ночі, шейдери 26

Висновки 29

Список використаних джерел 31

Додатки 33

Анотація

Мета цієї роботи – створення гри на фреймворку Unity з використанням процедурної генерації.

Проведено аналіз існуючих ігор та визначені методи і підходи до генерації ігрових елементів в реальному часі програми. Досліджено можливості сучасного ігрового фреймворку Unity.

Після аналізу було розроблено гру на даному фреймворку.

Вступ

Завдяки сучасним технічним та програмним можливостям розробникам ігор не обов’язкову створювати все з нуля. З ростом ігрової індустрії почали з’являтись програми які спрощують розробку та дають можливість використати готовий базовий функціонал – такі програми називаються ігровими фреймворками. Існують такі фреймворки як: Cocos2D, Unreal engine, Unity та багато інших. В рамках курсової роботи використовується саме Unity.

За типом ігрового світу, ігри можна поділити на 2 типи – ігри в яких рівні створюються заздалегідь гейм-дизайнерами та ігри в яких створення рівня відбувається динамічно, прямо під час роботи застосунку. В даній роботі будуть розглядатись ігри саме з динамічним типом створення рівнів.

Першим важливим етапом є дослідження існуючих ігор, аналіз підходів до створення рівнів за допомогою процедурної генерації.

Наступним етапом є реалізація власної гри на фреймворку Unity використовуючи досліджені підходи до генерації рівнів.

Використовується таке програмне забезпечення:

- Середовища розробки: Visual Studio 2019, Unity 2019.4.10f1 (64-bit)
- Мови програмування: C#
- Система керування версіями: Git
- Операційні системи: Windows 10

Робота складається з вступу, двох частин, висновків, списку використаних джерел та додатків.

Розділ 1. Аналіз існуючих ігор. Дослідження підходів процедурної генерації.

1.1 Diablo 1

1.1.1 Загальний опис Diablo 1

Diablo 1 – гра мандрівного жанру випущена в 1996 році. Була однією з перших ігор такого типу в якій використали повноцінну графіку замість ASCII. Основна ідея – гравець повинен пройти 4 етапи (Cathedral, Catacombs, Caves і Hell) кожен етап містить 4 рівні які зростають по складності. Весь ігровий світ складається з ізометричних тайлів (рисунок 1.1) які формують певний рівень певного етапу гри.

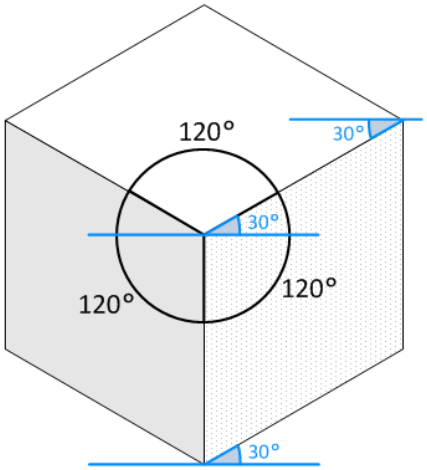

Рисунок 1.1 – Ізометричний тайл

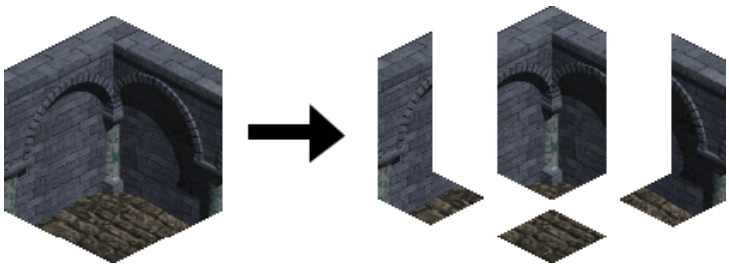

Рисунок 1.2 – Фрагменти ізометричного тайлу

1.1.2 Кроки генерації рівня

Кожен рівень починається з сітки 40 * 40 тайлів. Так як форма рівня – це повернутий на 45 градусів квадрат, то можна виділити такі дві осі: вісь X, яка напрямлена на південний захід та вісь Y - на південний схід. Після генерації схеми рівня кожен тайл розбивається на 4 фрагменти, які малюються на екрані (рисунок 1.2). Таке розбиття дозволяє визначити фрагменти по яким гравець може рухатись та забезпечує більш ефективне використання графічної пам’яті – тобто однакові фрагменти займають одну і ту ж область пам’яті.

Генерація відбувається у два етапи – в першому етапі створюється массив в якому помічаються через які тайли можна переміщуватись, в яких знаходяться

двері та інші специфічні ігрові елементи. На другому етапі вже генерується масив тайлів з урахуванням попередньої заготовки і вносяться необхідні зміни в самі тайли.

1.1.3 Генерація рівнів в етапі Cathedral

Перше що робить алгоритм – навмання обирає до трьох ділянок з сітки які мають розмір 10 * 10 – це і будуть початкові найбільші кімнати. Інші кімнати генеруються за допомогою рекурсивної функції L5roomGen яка використовує техніку «брунькування».

Функція L5roomGen отримує на вхід координати однієї з початкових кімнат та вісь(X або Y) по якій треба будувати нові кімнати. Також з вірогідністю 1/4 може змінитись передана вісь на протилежну. Далі по обраній осі (тобто з двох протилежних сторін кімнати) створюється нова кімната розміру 2, 4 або 6. Наступним кроком нова кімната центрується відносно попередньої, відбувається перевірка чи не перетинається кімната з іншими елементами та чи не виходить за межі рівня. Якщо все успішно - кімната малюється і для неї рекурсивно викликається функція L5roomGen. В параметри передається нова кімната та вісь протилежна попередній.

На наступному етапі потрібно згенерувати стіни на межах тайлів кімнат. Для цього використовується варіація алгоритму Marching squares, який буде описано в наступному розділі.

1.1.4 Алгоритм marching squares

Алгоритм marching squares використовується в комп’ютерній графіці для генерації ізоліній на двовимірній сітці. Також алгоритм використовують для малювання ізобар на географічних картах.

Нехай існує двовимірна сітка 7 * 7 та певна функція яка задає графік кола (рисунок 1.3). Наша задача визначити точки на сітці які будуть відображати коло та з’єднати їх правильним чином. Першим кроком помічаємо вершини які попали

в середину кола (помічені чорним кольором). Далі потрібно пробігти по кожній клітинці та відділити білі вершини від чорних за допомогою ліній які з'єднують середини сторін клітинок. Усі можливі комбінації можна побачити на рисунку 1.4.

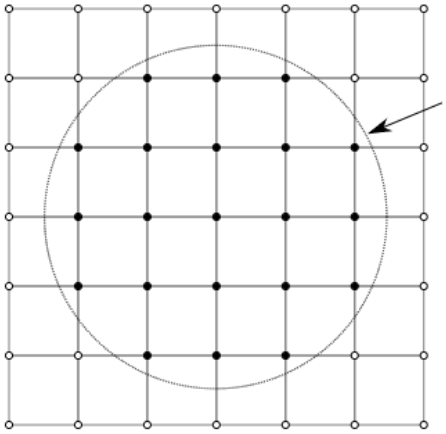

Рисунок 1.3 – Двовимірна сітка та коло

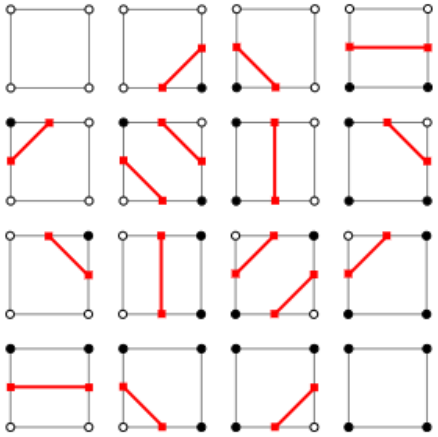

Рисунок 1.4 – Всі можливі комбінації відокремлення вершин

Пройшовши по всім клітинкам отримаємо результат (рисунок 1.5)

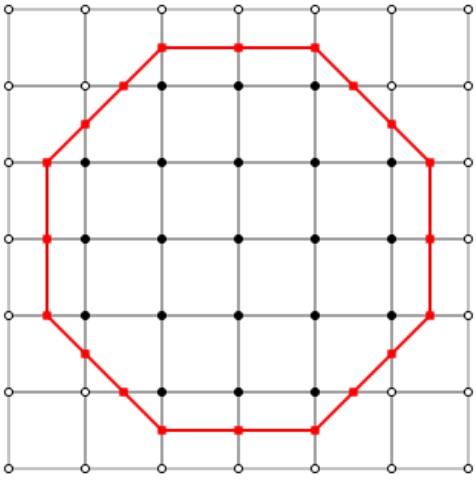

Рисунок 1.5 – Результат роботи алгоритму

Алгоритм так само працює в тривимірному просторі. В diablo використовується двомірний варіант і також пропускаються деякі комбінації з рисунку 1.4. Після розміщення основних стін за цим алгоритмом генератор також створює колони по куткам кімнат, в деяких місцях замінює стіни на арки та додає інші косметичні елементи.

Останнім кроком розміщаються сходи які переводять гравця на інший рівень. Але іноді буває що їх розміщення неможливе через відсутність вільних клітинок, в такому разі генерація починається заново.

1.1.4 Інші етапи в diablo

На інших етапах генерація рівнів відбувається схожим чином, але зі зміною деяких параметрів генерації (розміри і кількості кімнат, використовуються інші тайли відповідно до стилю етапу). Також в рівні додаються готові елементи, створені завчасно, вони необхідні для проходження квестів (битви з монстрами, босами і т.п.), в цьому випадку процедурна генерація не використовується.

1.2 Minecraft

1.2.1 Загальний опис гри Minecraft

Minecraft – гра жанру «пісочниця» випущена в 2011 році компанією Mojang. Світ гри – нескінченний та складається з кубів (вокселів) які дозволяють гравцю будувати будь-що та мандрувати світом з унікальним ландшафтом. Світ генерується по мірі просування гравця на певну відстань від нього – це дозволяє керувати навантаженням на комп’ютер користувача та запускати гру на девайсах майже будь-якої потужності.

1.2.2 Основа алгоритму генерації світу

Протягом існування гри алгоритм дещо змінювався, але розглядатись буде останній, найновіший варіант. Алгоритм генерації генерує світ частинами по 16*16*16 блоків, зберігає дані на диск і в подальшому, якщо гравець повернеться на цю ділянку, вона завантажиться з диску. Сама ж генерація відбувається у декілька етапів. В основі алгоритму лежить шум Перліна.

Шум Перліна – розроблений Кеном Перліном у 1983 році. Широко використовується в комп’ютерній графіці і належить до градієнтних шумів. В загальному випадку шум – випадкові дані або просто сміття яке не несе в собі ніякого сенсу. Приклад візуалізації у вигляді піксельного шуму представлений на рисунку 1.6. Кожен піксель у такому шумі незалежний від кольору інших. На відміну від нього

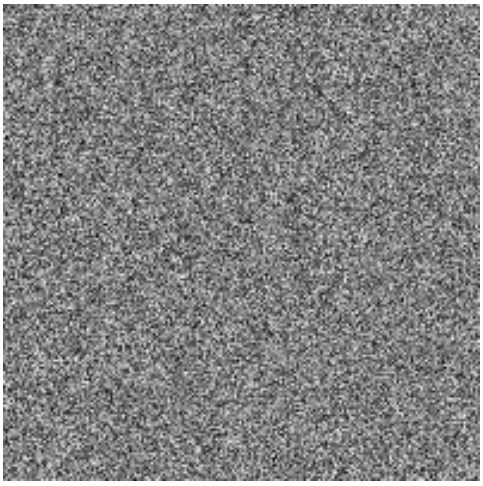

Рисунок 1.6 – Піксельний шум

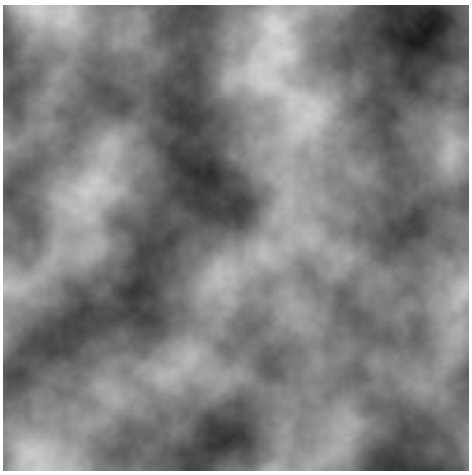

Рисунок 1.7 – Шум Перліна

шум Перліна (рисунок 1.7) має плавні (градієнтні) переходи що робить його корисним для створення текстур хмар, туману, диму. Також шум Перліна можна використовувати для генерації мап висот. На такій мапі чим темніший піксель – тим нижча ця точка по висоті, а чим світліша – тим вища.

1.2.3 Шум Перліна

Шум Перліна можна застосовувати на будь-якій кількості вимірів, але найзручніше – демонструвати на одному вимірі. Нехай існують деякі точки X, які представлені цілими числами, тобто x = 0, 1, 2, 3… Також для кожної такої точки виберемо випадкове дійсне число з діапазону [-1, 1]. Дійсне число – це тангенс кута під яким через нашу точку x проходить деяка пряма. Якщо зобразити це на графіку отримаємо рисунок 1.8. Далі потрібно добудувати кожну синю пряму до сусідніх іксів. Якщо виконаємо це отримаємо рисунок 1.9. Далі потрібно з’єднати наші точки на осі X плавними дугами орієнтуючись на добудовані точки прямих. Це робиться завдяки лінійній інтерполяції.

Лінійну інтерполяцію можна описати формулою: A + t(B-A), де A та B деякі точки зі значеннями, t – деяка координата на шляху від A в Б і приймає дійсне значення від 0 до 1 (при 0 ми знаходимось в точці A, при 1 в точні B). Таким чином, чим ближче ми до точки A тим сильніше вона впливає на результат, аналогічно і з точкою B.

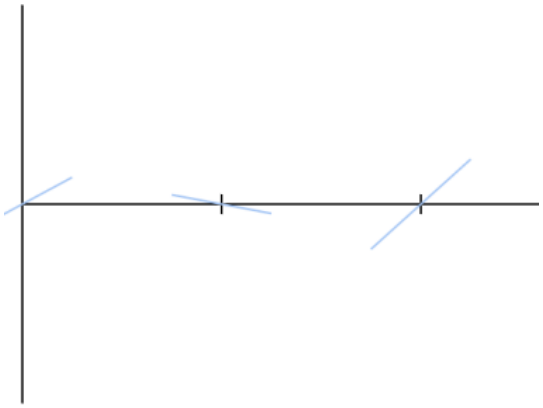

Рисунок 1.8 – Точки x та прямі які проходять через них(сині)

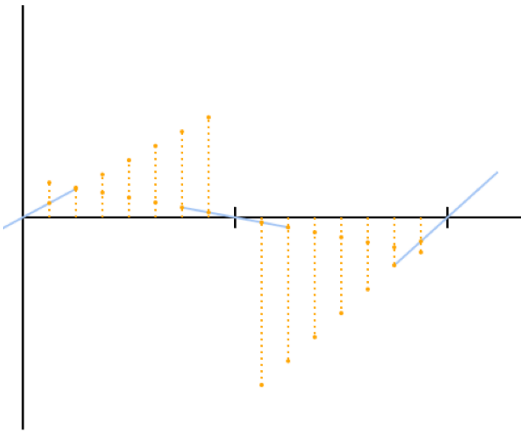

Рисунок 1.9 – Добудовані сині прямі по точкам(жовті точки)

В даному випадку лінійна інтерполяція буде використана щоб вирахувати точки на вертикальних жовтих відрізках, які з'єднують точки продовження сусідніх прямих. Значення t залежить від близькості до певного ікса, а A та B це y-координата відповідної жовтої точки. Помітивши результуючі точки червоним кольором отримаємо рисунок 1.10

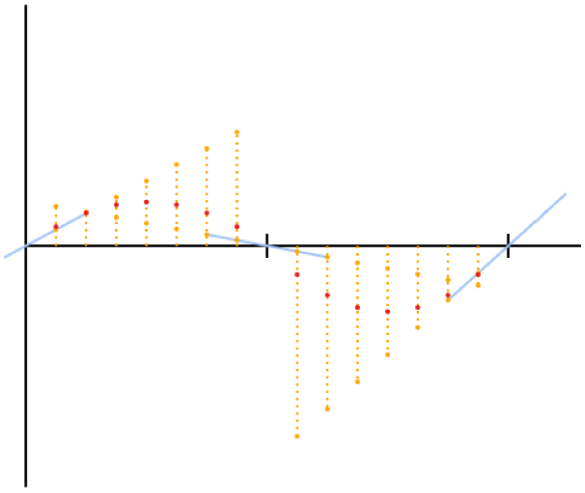

Рисунок 1.10 – Інтерпольовані точки (червоним)

Далі потрібно з'єднати точки, але з'єднавши точки відрізками ми отримаємо ламану лінію, яка не забезпечить плавного переходу між значеннями. Отже мета малювати параболи які проходять через червоні точки. Щоб їх побудувати використовуються криві Безьє. Для кривої Безьє нам потрібно мати хоча б три точки. 2 точки вже є – це початкові точки на осі X.

Третя точка будується наступним чином. Початкові прямі які проходять під кутом до точок на осі X дзеркально відображаються на кожній частині між двома точками (ці прямі помічені синім пунктиром на рисунку 1.11). Далі кут початкової прямої додається до кута дзеркальної і під цим кутом будується третя пряма

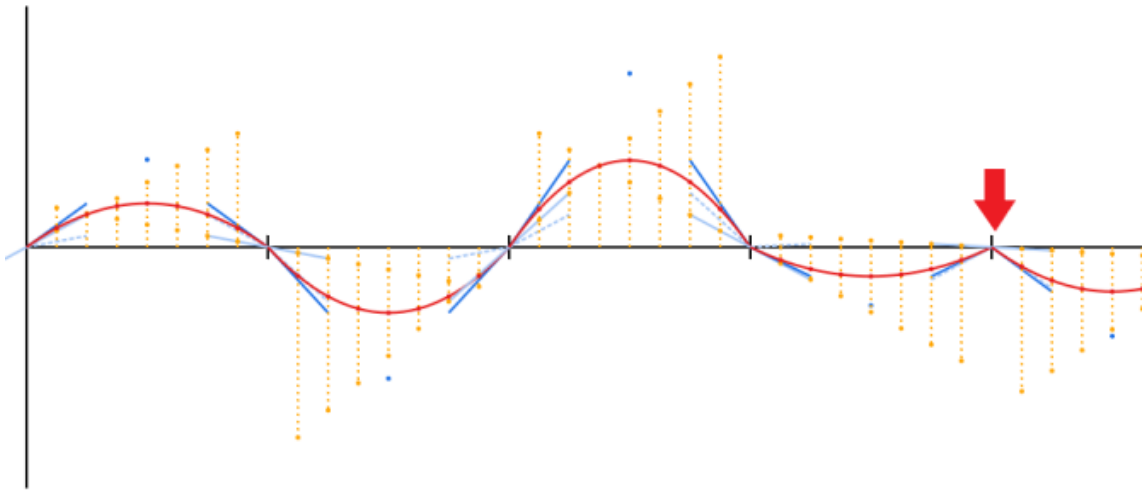

Рисунок 1.11 – Криві Безьє за трьома точками формують необхідні параболи

(темно синя). На перетині двох таких прямих ставимо точку(темно сині точки) – це і є 3 точка для кривої Безьє. Побудувавши їх отримуємо плавну червону лінію.

Також можливі проблеми з переходами в самих точках x, на рисунку помічено червоною стрілкою. Для уникнення таких дефектів перед лінійною інтерполяцією до параметра t застосовують функцію smoothstep. Вона схожа на звичайну діагональну пряму (y = x), але згладжену на кінцях. Застосувавши цю функцію ми отримаємо результат на рисунку 1.12.

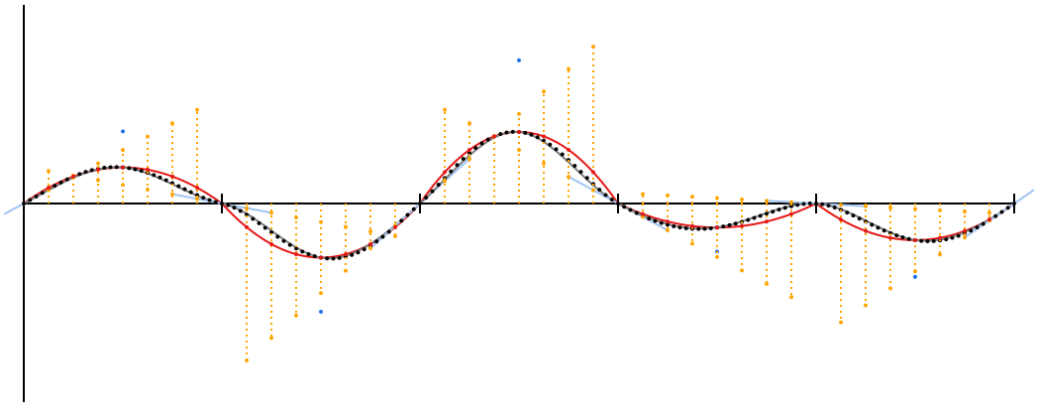

Рисунок 1.12 – Згладжений результат завдяки smothstep (сіра лінія з чорними точками)

Якщо на даному етапі шум Перліна одновимірний, тобто на вхід подаємо тільки значення іксів, а на виході отримуємо висоти, то виникає питання як використати шум Перліна в двовимірному просторі. В двовимірній інтерпретації шуму Перліна використовується сітка (рисунок 1.13), а в кожній точці перетину сітки розміщуються одиничний вектор з напрямом вибраним навмання (аналог тангенса кута з одновимірної інтерпретації).

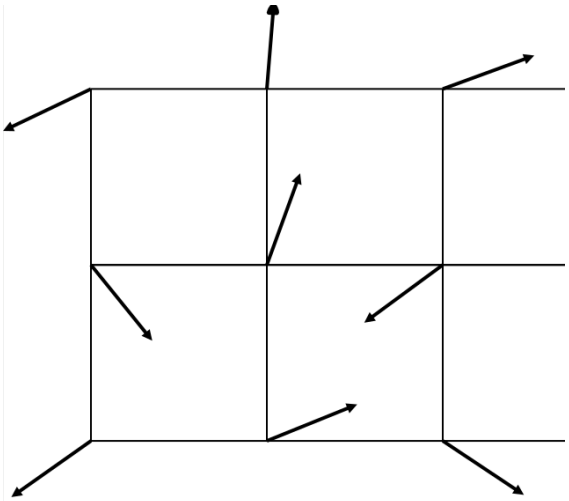

Рисунок 1.13 – Двовимірна сітка з одичними векторами на вузлах

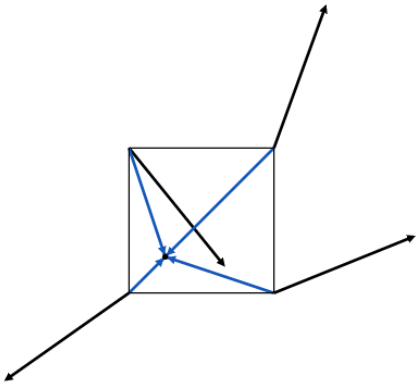

Рисунок 1.14 – Обрана точка на сітці (до неї прямують сині вектори)

Якщо в одновимірній інтерпретації кожна точка мала дві сусідніх на осі X, то тепер кожна точка має 4 сусідніх точки сітки. Оберемо деяку точку з сітки и визначимо як для неї отримати результуюче значення шуму, тобто висоту. До такої точки можна також провести вектори з кожної сусідньої точки перетину сітки (сині вектори на рисунку 1.14). Далі для кожної точки перетину сітки обчислимо скалярний добуток навмання вибраного вектора та вектора який прямує до обраної точки. Маючи чотири таких значення ми повинні провести інтерполяцію. Для цього першим кроком знайдемо інтерпольовані значення для одної пари точок і для другої пари. Другим кроком інтерполюємо по двом попередньо отриманим значенням. На виході отримаємо значення яке і буде нашою висотою для обраної точки.

Щоб отримати трьох вимірний шум Перліна достатньо замість двовимірної сітки використати трьохвимірну(можна уявити як куб, який складається з менших кубів) та зробити всі попередні процедури у цьому просторі.

1.2.4 Використання октав для покращення результатів

Проаналізувавши результат шуму Перліна на рисунку 1.12 можна зрозуміти, що одного шуму замало для імітації наприклад ландшафту. Річ у тім що результат все одно доволі одноманітний - на схилі або підйомі зміни графіку взагалі відсутні. Як же це змінити та покращити? Потрібно накласти декілька шумів Перліна з різними розширеннями – тобто різною відстанню між іксами на горизонтальній осі. Наприклад створимо ще один графік шуму Перліна, але тепер точки X це 0, 0.5, 1, 1.5. Будуємо новий графік шуму, ділимо результати навпіл та додаємо то попереднього (рисунок 1.15). Бачимо що результат істотно змінився в кращу сторону. Кожен такий окремий графік називають октавою. Оптимальними результати стають при кількості 4-5 октав (рисунок 1.16). Таким же чином покращуються і шуми більших вимірів. З точки зору математики це є фрактал, тобто з кожною ітерацією (додаванням нової октави) ми отримаємо більш деталізований результат. Сам Кен Перлін також запропонував використовувати різні математичні функції на кожну октаву шуму щоб отримати цікаві та особливі результати. Так наприклад застосувавши наступну функцію: sin(x + sum 1/f(|noise|)), можна отримати ефект полум’я (рисунок 1.17). Або хмари як на рисунку 1.18

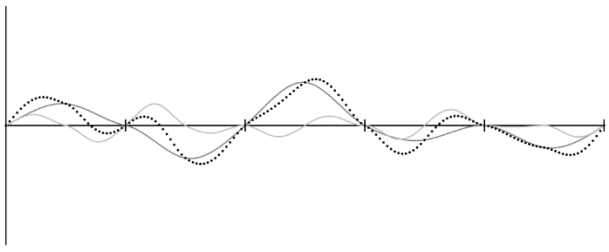

Рисунок 1.15 – Результат додавання двох шумів Перліна з різними розширеннями

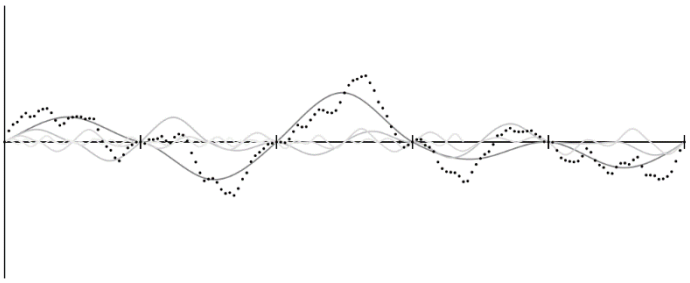

Рисунок 1.16 – Результат при 5 октавах

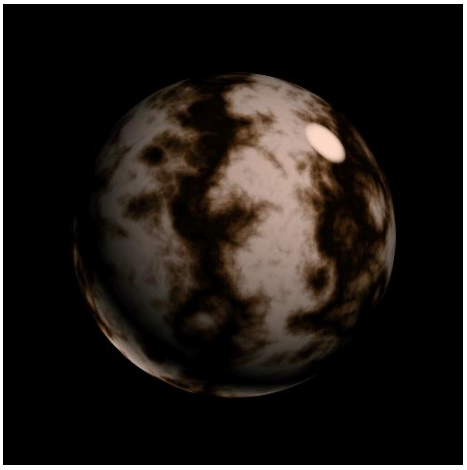

Рисунок 1.17 – Текстура полум’я отримана функцією sin(x + sum 1/f(|noise|))

Рисунок 1.18 – Текстура хмар отримана функцією 1/f(|noise|)

1.2.5 Випадковість результатів

Якщо розглядати шум Перліна абстрактно, як математичний інструмент, то навмання вибрані нахили наших прямих, у випадку одновимірного шуму, та напрями векторів, у двовимірному, забезпечують нам повну унікальність результатів. Але комп’ютерна реалізація алгоритму не використовує реально випадкові значення для цього. Існує певний алгоритм який просто встановлює ці значення, але при наступному запуску програми ми отримаємо такі ж самі результати. В контексті ігор це буде означати що згенерований світ буде повністю ідентичний попередньому і це зробить гру доволі нудною.

Чи може комп’ютер згенерувати по-справжньому випадкове значення? Насправді ні, але можна зробити значення таким, щоб людині здавалось що це так, тобто псевдовипадковим. Генератори таких значень зазвичай використовують таку інформацію як кількість тактів процесора, системний час, рух мишки і т.п. Чому ж це не вбудовують в реалізацію шуму Перліна? Тому що це дуже навантажить систему і зробить час виконання такої програми неприйнятним. Дієвим рішенням є згенерувати таке значення лише один раз і використовувати його як зміщення для шуму Перліна. Таке значення називають seed або зерно. Тоді шум стане дійсно випадковим в контексті ігрового світу комп’ютерної гри.

1.2.6 Генерація біомів та ландшафту

За допомогою шуму Перліна створюються мапи температури та вологості(результуюче значення шуму використовують як температуру та вологість відповідно з деякою зміною верхньої та нижньої межі).На основі двох цих значень обирається відповідний біом з графіку на рисунку . Також від температури залежать інтенсивність опадів. Річки генеруються на межах двох біомів.

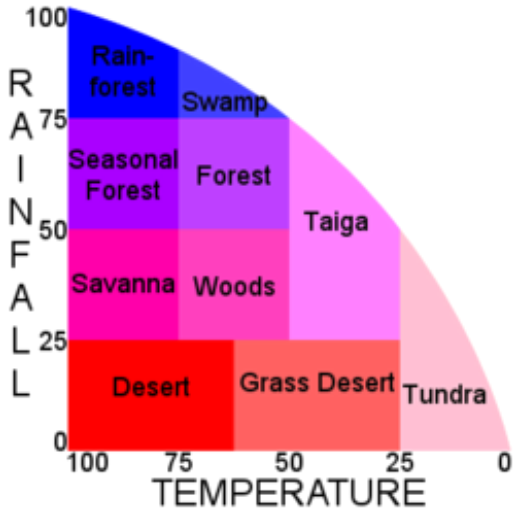

Рисунок 1.19 – Біоми в залежності від температури та опадів

Рисунок 1.20 – Ландшафт у грі minecraft

В перших версіях гри ландшафт генерувався за допомогою двовимірного шуму Перліна, але це робило результат доволі простим без багатьох деталей, таких як наприклад нависаючий ландшафт. Тому згодом генерація була змінена на трьохвимірну варіацію шуму. В цій варіації кожне вихідне значення шуму інтерпретується як щільність простору. Тобто якщо значення перевищує певний поріг – там вставляється блок, якщо ж ні – залишається пустий простір.

1.3 No Man’s Sky

1.3.1 Загальний опис гри No Man’s Sky

No Man’s Sky – гра в жанрі космічних пригод випущена студією “Hello Games” в 2016 році. Світ гри складається з планет в кількості 2^{64} і мета гравця виживати в такому світі, мандрувати різними планетами та досліджувати унікальні місця. При створенні ігрового світу використовується процедурна генерація. Гра займає приблизно 600 тисяч стрічок коду та займає 6gb пам’яті на жорсткому диску, що доволі небагато для такого роду проекту (для порівнянні Minecraft написаний п’ятьма мільярдами стрічок коду). Яким чином був досягнутий такий результат? Все завдяки широкому використанню математичних формул виведених відносно недавно.

1.3.2 Використання суперформули

Під час розробки гри програміст та засновник даної студії Сін Мюррей дійшов висновку що задати унікальні та неповторні ландшафти, елементи флори та інші деталі для кожної планети буде неможливо стандартними підходами процедурної генерації. Тому було вирішено використати так звану суперформулу.

Суперформула(рисунок 1.21) – математична формула виведена бельгійським вченим Йоханом Джилісом в 2003 році. Йохан припустив що ця формула може бути використана для опису складних природніх форм, кривих та інших елементів. Формула приймає три параметри (n1, n2, n3) які визначають саму форму та межі в яких вона задається (a і b).

$$r(\varphi) = \left[\left| \frac{\cos\left(\frac{m\varphi}{4}\right)}{a} \right|^{n_2} + \left| \frac{\sin\left(\frac{m\varphi}{4}\right)}{b} \right|^{n_3} \right]^{-\frac{1}{n_1}}$$

Рисунок 1.21 – Суперформула

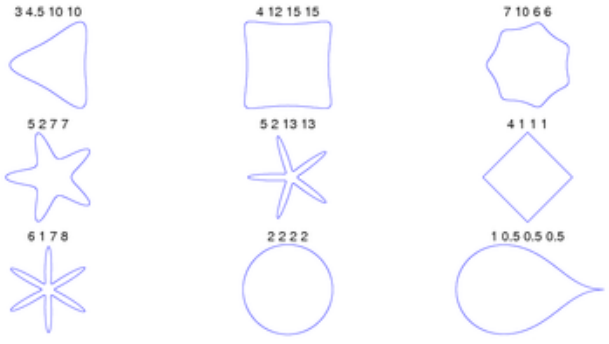

Рисунок 1.22 – Результати формули з різними вхідними параметрами

Завдяки формулі в грі генеруються ландшафти планет, тварини і рослини. Також через такий підхід у грі повністю відсутні очікування завантаження, світ генерується по мірі просування гравця.

1.3.4 Інші використані техніки

Звичайно у цій грі крім суперформули застосовується також велика кількість інших технік процедурної генерації. Наприклад для генерації листків дерев та рослин використовуються фрактали. А для генерації фауни був створений цілий двигун який дозволяє збирати істот з окремих частин, а сама комбінація залежить від набору параметрів (відстань до найближчої зірки, маса планети, атмосфера і т.д.) якими володіє певна планета. Така комбінація підходів дозволила створити дійсно унікальну гру, в якій при подорожі не зустрінеш одну і ту ж рослину, тварину, елемент ландшафту двічі.

Також уваги заслуговує звуковий двигун гри, він теж реалізує процедурний підхід. В даних самої гри є бібліотека звуків (семплів) та музики. Під час самої гри двигун обирає відповідні частини звуків та програє їх. Все залежить від обставин в яких знаходиться гравець. Коли ви безтурботно прогулюєтесь планетою – грає спокійна мелодична музика, як тільки починається буря – музикальний супровід стає суворим та напруженим. Схожий підхід використали і в грі Doom.

Рисунок 1.23 – Ландшафт у No Man’s sky

Розділ 2. Реалізація гри з процедурною генерацією з використанням фреймворку Unity

2.1 Тематика та особливості гри

Фреймворк Unity дозволить використовувати такі готові речі як сітки об’єктів, фізику зіткнень та падіння об’єктів, текстурування, освітлення і тіні, стандартні шейдери, анімації. Важливо що це також дає нам повний контроль над виглядом та формою об’єктів зі скриптів, які пишуться на мові C#. Завдяки Unity можна не вдаватись в низькорівневі особливості реалізації графіки і сконцентруватись переважно на алгоритмах процедурної генерації.

Перш за все потрібно вирішити тематику гри. Це буде гра з відкритим нескінченним світом по якому гравець зможе мандрувати та виконувати певні завдання. На відміну від Minecraft, ця гра не буде використовувати вокселі. Тільки повноцінна багатополігональна графіка може передати відчуття реальності та присутності в ігровому світі. Завдяки детальному дослідженню шуму Перліна – цей шум буде використовуватись для створення ландшафту та інших деталей гри. Моделі та текстури за необхідності будуть взяті з безкоштовного розділу Unity asset store.

2.1 Структура ігрового світу

Нескінченний ігровий світ не може генеруватись суцільно, він повинен бути розділений на структурні одиниці які дозволять провантажувати світ поступово. Зручною формою такого тайлу є квадрат або прямокутник – його координати та параметри легко сприймати та обробляти в алгоритмах.

Наступне – це вибір самого об’єкту. Такий тайл можна створювати з нуля, вручну, використовуючи скрипти, або ж взяти готове рішення від Unity. За замовчуванням в Unity э ігрові об’єкти продемонстровані на рисунку 2.1. Потенційними кандидатами для використання в якості тайлу є куб(cube), площина (plane) та місцевість (terrain).

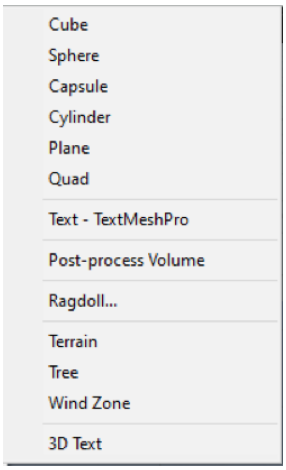

Рисунок 2.1 – Готові об’єкти в Unity

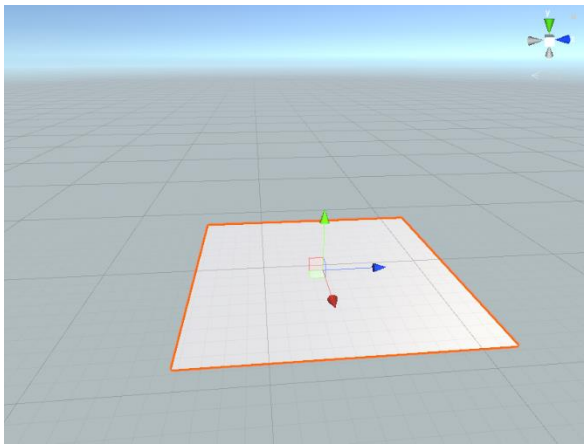

Рисунок 2.2 – Ігровий об’єкт - площина

Основна задача terrain – створення дуже великих природніх ланшафтів, також в Unity вбудований редактор який дозволяє формувати поверхню та текстуру такого об’єкту. Цей варіант одразу можна відкинути адже нам потрібен для маленьких ділянок. Куб – має всього лише два трикутники на одну сторону, що робить його занадто маленьким для формування одного тайлу, крім того висота куба в принципі не потрібна для створення ландшафту. Отже залишається площина, яка має 10 полігонів по висоті і ширині, можливість масштабування, трансформації та встановлення шейдерів. Це ідеальний об’єкт в якості тайлу.

2.2 Модуль для генерації шуму Перліна

Так як в грі буде використовуватись шум Перліна потрібно створити скрипт для зручного повторного використання в будь-якій частині проекту. Для цього створений клас NoiceMapGenerator, в імплементований метод GenerateNoiceMap. Метод приймає довжину тайла в полігонах, абсолютний розмір в пікселях, зміщення по осям x та z. Також створений додатковий клас Wave, який представляє параметри однієї октави – сід, частота та амплітуда. В самому методі проходить ітерація по всім точкам полігональної сітки тайла і для кожного генерується значення на основі певної кількості октав шуму Перліна. Отримавши такий результат можна змінити висоту вузла сітки тайла відповідно до виданого значення і отримати необхідний ландшафт.

2.3 Генератор світу

Генератор світу буде керувати генерацією усіх тайлів в рівні. Також його задача слідкувати за користувачем та завантажувати або видаляти необхідні тайли. Це необхідно для економії пам’яті адже нескінченно створювати та зберігати в пам’яті кожен тайл неможливо. Для визначення які тайли необхідно завантажити можна використати рівняння коло. Таким чином навколо гравця на певній відстані від нього будуть завжди присутній ландшафт. Наступний код створює список актуальних тайлів у вигляді (x, y), з центром світу у координатах (0, 0)

```
List<(int, int)> ps = new List<(int, int)>();
int r = visibleRange; // radius
int ox = playerX / TileData.tileWidth, oy = playerY / TileData.tileDepth;

for (int x = -r; x < r; x++)
{
    int height = (int)Math.Sqrt(r * r - x * x);

    for (int y = -height; y < height; y++)
    ps.Add((x + ox, y + oy));
}
return ps;
```

2.4 Заготовка тайлу

В Unity можливо використовувати так звані заготовки або prefabs. Вони дозволяють зберегти ігровий об’єкт з усіма його налаштуваннями у пам’ять. В подальшому можливо створювати клони таких заготовок у самій грі за допомогою скриптів. За основу заготовки тайлу взята площина(plane). На заготовку додані генератор шуму Перліна(NoiceMapGenerator) та генератор одного тайлу (TileGenerator). Задача генератору тайла прийняти параметри від генератора світу та внести необхідні зміни в модель тайлу. В першу чергу на основі значень з шуму Перліна змінюється сітка моделі тайлу.

Важливим елементом є накладання текстури. У грі текстура залежить від біому та висоти. Заздалегідь завантажені текстури комбінуються прямо при генерації тайлу. Кожен тайл умовно поділяється на 100 маленьких квадратів, кожен такий

квадрат відповідає двом полігонам тайлу. З текстур трави, каміння, піску, землі зчитуються квадрати такого самого розміру і вставляються в нову текстуру.

Для гарного результату використовується також стандартний шейдер Unity. Шейдер – це програма яка виконується на відеокарті комп'ютера і полягає в операціях з графікою, текстурами. Стандартний шейдер дозволяє вказати основну текстуру, карту нормалей та висот(роблять виступи на нерівності на об'єкті), карту оклюзій (впливає на освітлення). Всі ці додаткові елементи додаються до шейдеру під час генерації.

Істотною проблемою є стики на межах різних частинок текстур. Ці стики є абсолютні і відсутні плавні переходи від одної текстури до іншої. На рисунку 2.3 продемонстрована дана проблема. Вирішенням є плавний перехід від однієї текстури до іншої. Це можна зробити за допомогою лінійної інтерполяції, яка була розглянута в розділі з шумом Перліна. Проходимось по стику текстур на певній ширині та використовуємо функцію Color.Lerp, яка приймає два значення кольорів та ступінь близькості до однієї зі сторін від 0 до 1. На рисунку 2.4 можна побачити згладжений варіант стиків.

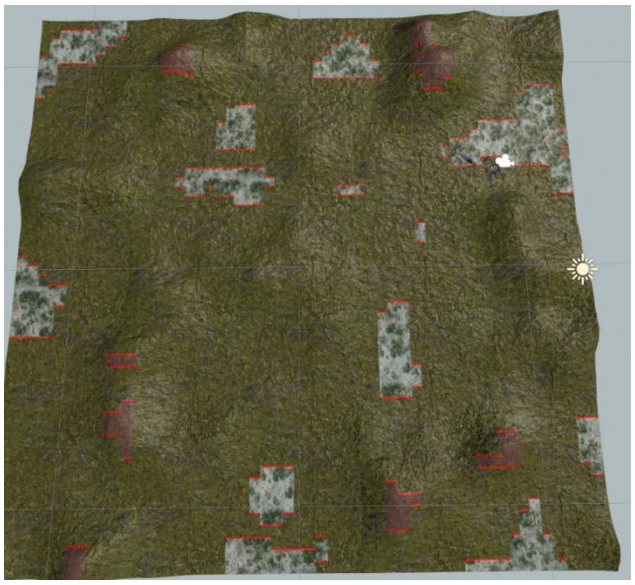

Рисунок 2.3 – Стики між згенерованими частинами текстур (горизонтальні стики додатково виділені червоним кольором)

Рисунок 2.4 – Згладжування переходу між різними частинами текстур

Серйозною проблемою є об’єми оперативної пам’яті які займають такі динамічні текстури, адже якщо потрібна хоч невелика зміна в тайлі – це буде нова ділянка пам’яті з інформацією про пікселі всієї текстури. Саме тому світ провантажується з рухом користувача и підчищається у ділянках де його немає. В цьому допомагає метод Resources.UnloadUnusedAssets(), який очищає пам’ять від текстур які не більше використовується.

В просунутих іграх водойми або річки можуть бути на різних рівнях висоти, але в рамках курсової роботи достатньо додати воду на одному рівні висоти. Це автоматично зробить глибокі западини водоймами та річками. Маніпуляції з октавами дозволяють змінювати ландшафт від рівнинного до гірського (рисунок 2.5 та 2.6).

Також в Unity можна задавати функції в візуальному редакторі. Клас AnimationCurve надає функціонал для роботи з такими функціями і одне з корисних застосувань для тайлу – розрівнювання дна западин. Тобто можна визначити функцію яка при значеннях менших деякого порогу буде видавати 0 , але при більшому залишати незмінним.

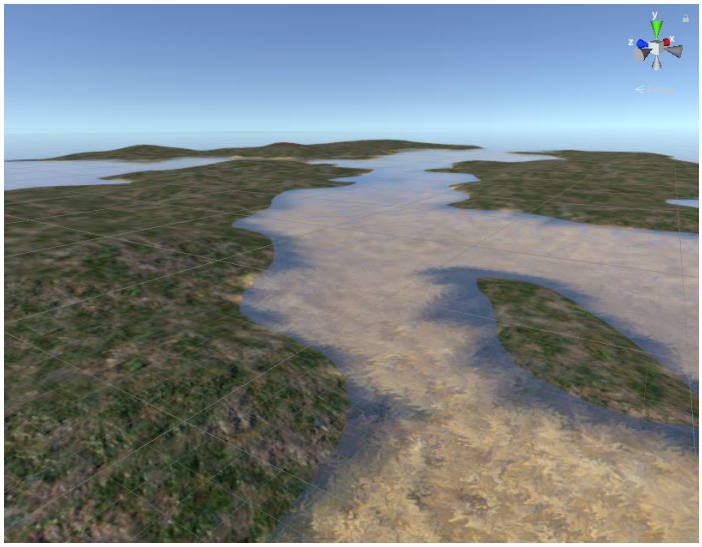

Рисунок 2.5 – Річка з піщаним дном

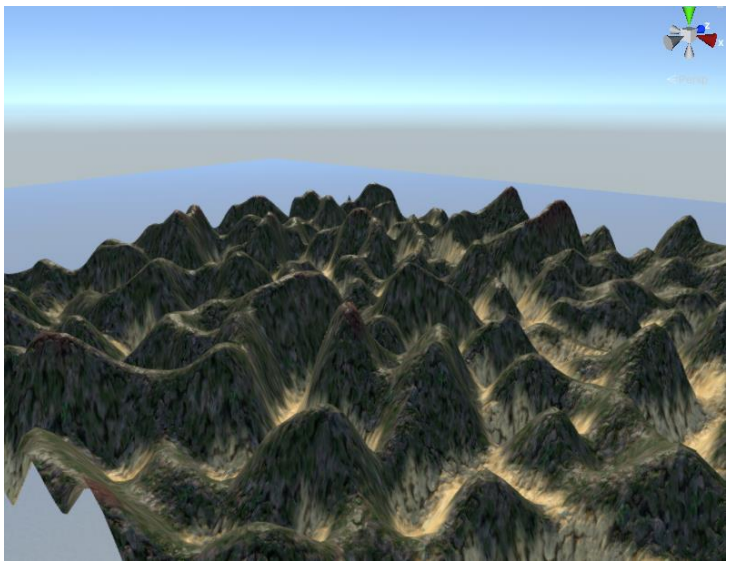

Рисунок 2.5 – Гірські хребти

2.5 Зміна дня та ночі, шейдери

Важливою складовою гри зміна дня та ночі, це надає грі більшої реалістичності. Для освітлення найкраще використовувати напрямлене світло (direction light) – воно створює паралельні промені світла по всьому ігровому світлу. Також воно може бути динамічним, тобто змінювати своє положення і при цьому буде змінюватись саме освітлення. Також існує декілька інших параметрів, такі як інтенсивність, колір, помножувач непрямого освітлення, силі та інші. Погравшись з налаштуваннями можна створити два джерела світла – для денного та нічного освітлення. Наступним кроком можна розмістити їх на 180 градусів один від одного і поступово повертати по колу що і буде зміною освітлення дня і ночі. Схожим чином створюються моделі сонця та місяця і на постійній відстані від користувача обертаються по колу змінюючи один одного.

Крім небесних тіл потрібно також міняти вигляд всього неба. В Unity для відображення неба використовується skybox, або небесна коробока. По суті камера гравця завжди поміщена в куб на який натягнуті текстури неба. Кожен кадр скайбокс промальовується в першу чергу, а потім всі інші елементи графіки. Розгортка скайбоксу представлена на рисунку 2.6. Налаштувати такий скайбокс дуже просто, достатньо лише скачати 6 текстур скайбоксу та імпортувати їх в Unity.

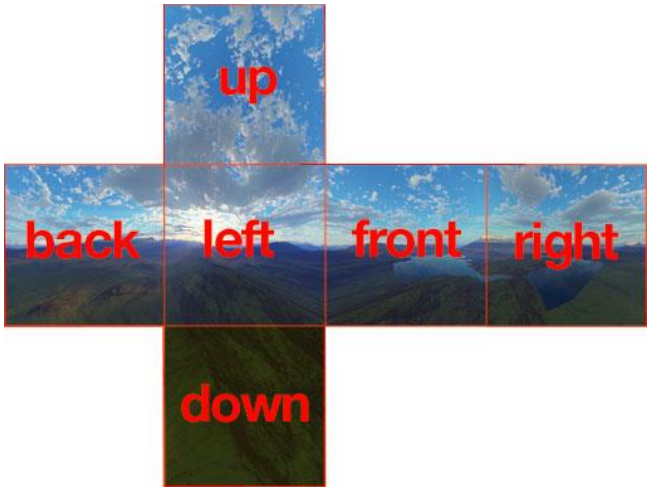

Рисунок 2.5 – Розгортка SkyBox

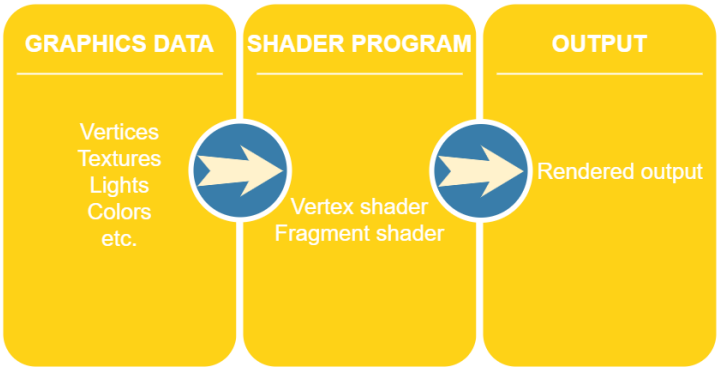

Рисунок 2.6 – Схема роботи шейдера

Складною задачею є реалізація плавного переходу від денного скайбоксу до нічного. Загалом це можливо реалізувати на процесорі (звичайним скриптом на C#), але дуже неефективно і гра стане гальмувати та глючити. Ефективно з такою задачею впорається відеокарта, а програму для неї можна написати шейдерною мовою. В Unity частіше всього використовується мова ShaderLab. Ця мова має простий с-подібний синтаксис та відносно проста у вивченні. Основною ідеєю для переходу буде знову лінійна інтерполяція, а відстань від одного скайбоксу до іншого буде встановлюватись з C# коду. Код шейдеру приведений в додатку А. Залишається лише на початку дня або ночі за певний проміжок часу виконувати таку команду RenderSettings.skybox.SetFloat("_Blend", i);. Це і є передача ступеня близькості від одного скайбоксу до іншого в середину шейдера. Змінна “i” в даному випадку може бути від 0 до 1.

Рисунок 2.7 – Небо посередині дня

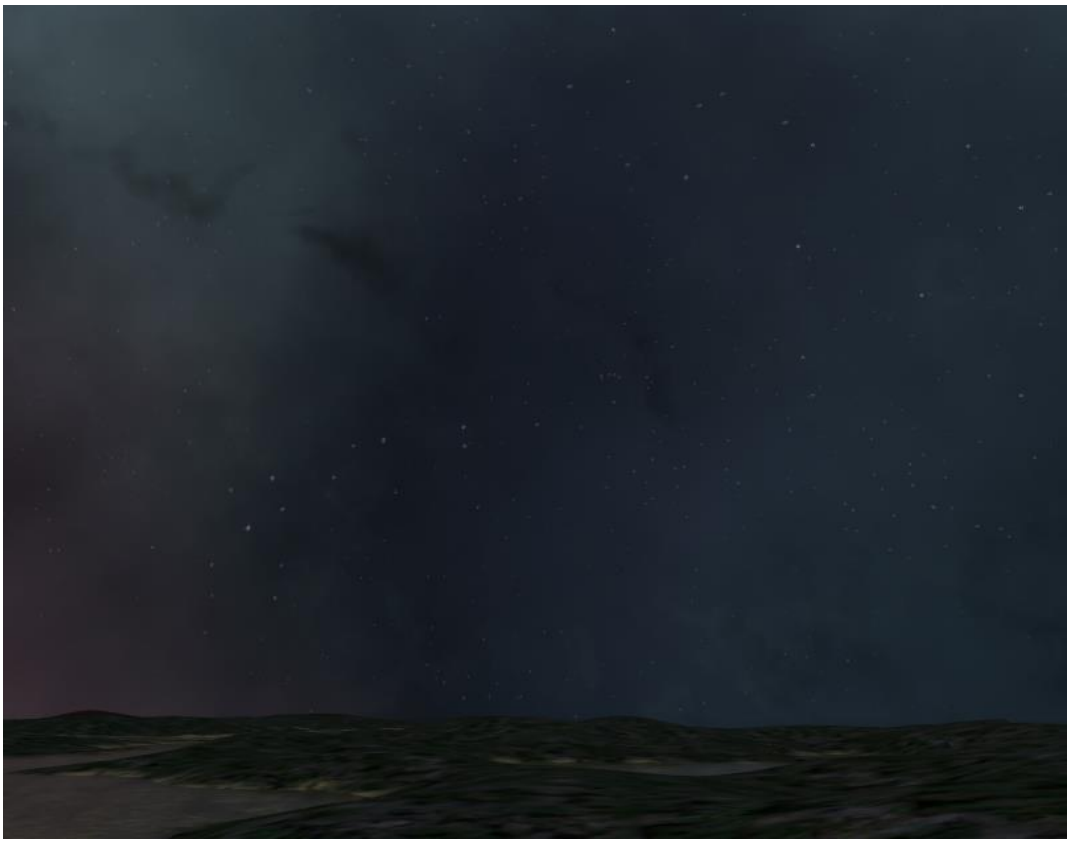

Рисунок 2.7 – Нічне небо

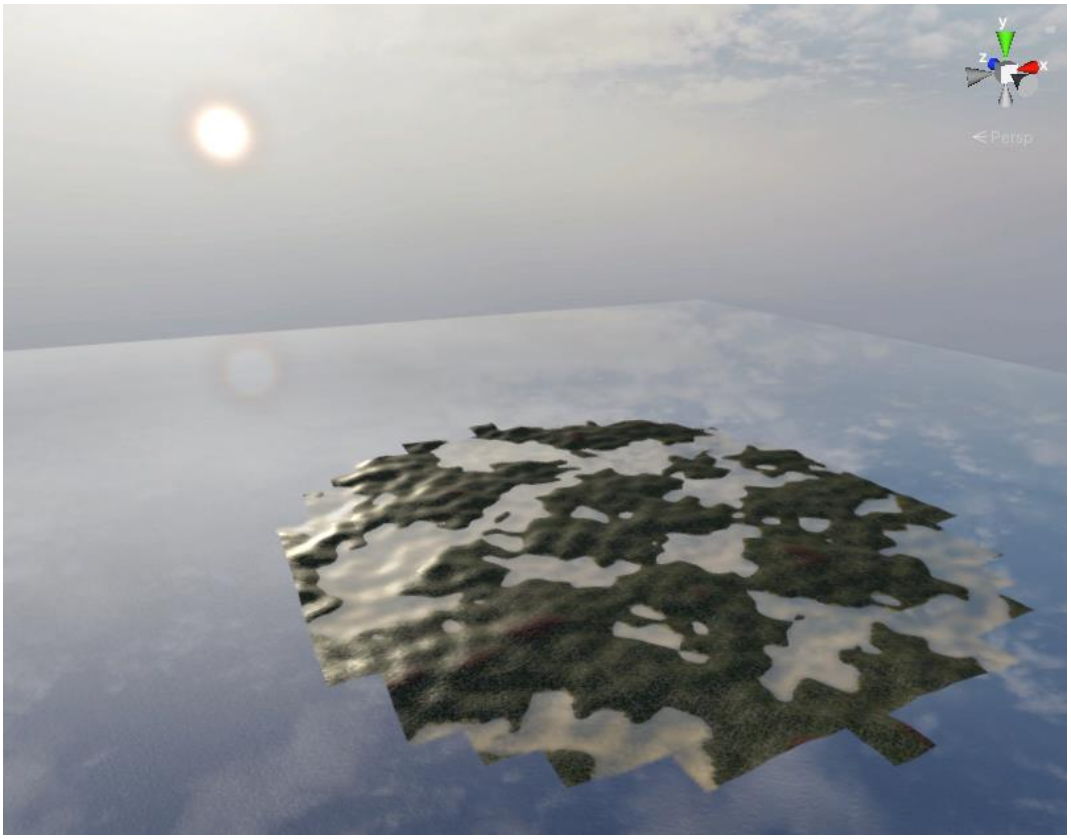

Рисунок 2.8 – Ігровий світ, вигляд зверху

Висновки

Завдяки алгоритмам процедруної генерації, потужному та сучасному залізу і ігровим фреймворкам типу Unity стало можливим створення нескінченних, унікальних світів. Алгоритми процедурної генерації – ключ до розв'язку задач такого типу. В іграх все починалось з простих алгоритмів на зразок крокуючих квадратів – їх застосування в перших графічних іграх типу diablo дало поштовх як використанню процедурної генерації в них так і ігровій індустрії в цілому. Вагомий вклад в розвиток вніс Кен Перлін створивши свій широковідомий шум Перліна. Саме алгоритмічні шуми можуть відтворити природню неповторність і зробити гру не просто послідовністю виконаних завдань, а цікавим всесвітом для подорожі. В останні роки розробники ігор ставлять планку все вище та вище – всесвіт з гри No Man's sky вражає уявлення своїми розмірами та формами. Використання суперформул та інших математичних інструментів може врешті решт призвести до створення ігрового світу ідентичного до нашого.

Сучасний фреймворк Unity – набір великої кількості інструментів, що значно спрощують та пришвидшують розробку ігор. Можливість не вдаватись в низькорівневі деталі гри розширює круг розробників, робить поріг входження в індустрію нижчим ніж це було десятки років тому.

Також розробка власної гри дозволила краще зрозуміти практичне використання алгоритмів процедурної генерації. Досліджено стандартний набір методів та функцій для роботи з ігровими об'єктами, створення текстур в runtime. Використання шейдерної мови ShaderLab дозволило покращити вигляд гри та зробити її графіку більш динамічною.

З подальшої розробки гри можна виділити такі пункти:

- Генерація флори та фауни
- Створення структур таких як замки, селища, інші рукотворні споруди
- Програмування взаємодії зі світом – підбір предметів в інвентар, виконання завдань, створення ворогів.

- Система крафтів інших предметів
- Багатокористувацький режим
- Збереження даних світу на диск
- Оптимізація використання пам’яті, покращення швидкодії

Список використаних джерел

1. Процедурна генерація в Unity [Електронний ресурс] / Renan Oliveira // gamedevacademy.org – 2019. - Режим доступу до ресурсу: https://gamedevacademy.org/complete-guide-to-procedural-level-generation-in-unity-part-1/
2. Процедурно генеровані карти в Unity [Електронний ресурс] / Jon Gallant// jgallant.com – 2016. - Режим доступу до ресурсу: http://www.jgallant.com/procedurally-generating-wrapping-world-maps-in-unity-csharp-part-4/
3. Процедурна генерація в diablo 1 [Електронний ресурс] / BorisTheBrave// boristhebrave.com – 2019. - Режим доступу до ресурсу: https://www.boristhebrave.com/2019/07/14/dungeon-generation-in-diablo-1/
4. Алгоритм marching squares [Електронний ресурс] / Roger Ngo // urbanspr1nter.github.io – 2015. - Режим доступу до ресурсу: https://urbanspr1nter.github.io/marchingsquares/
5. Шум Перліна [Електронний ресурс] / Lexy Munroe// eev.ee – 2016. - Режим доступу до ресурсу: https://eev.ee/blog/2016/05/29/perlin-noise/
6. Шум Перліна в грі Minecraft [Електронний ресурс] // minecraft.fandom.com – 2012. - Режим доступу до ресурсу: https://minecraft.fandom.com/wiki/Noise_generator
7. Октави та Шум Перліна [Електронний ресурс]/ Yvan Scher // medium.com – 2017. - Режим доступу до ресурсу: https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401
8. Суперформула [Електронний ресурс]/ Uwe Stohr // fkurth.de – 2016. - Режим доступу до ресурсу: http://web.archive.org/web/20160616215007/http://fkurth.de/uwest/usti/Superformel/SuperformulaU.pdf
9. Суперформула в грі No Man’s Sky [Електронний ресурс]/ JV Chamary// forbes.com – 2016. - Режим доступу до ресурсу:

https://www.forbes.com/sites/jvchamary/2016/07/27/no-mans-sky-superformula/?sh=39c3fe3a2247

10. Маніпуляції з сіткою в Unity [Електронний ресурс]/ Sean Duffy// raywenderlich.com – 2019. - Режим доступу до ресурсу: https://www.raywenderlich.com/3169311-runtime-mesh-manipulation-with-unity
11. Зміна параметрів матеріалу в Unity в runtime [Електронний ресурс]/ kristiel// blog.studica.com – 2016. - Режим доступу до ресурсу: https://blog.studica.com/unity-tutorial-code-material-properties
12. Шейдери в Unity [Електронний ресурс]/ Олексій Чернов// habr.com – 2021. - Режим доступу до ресурсу: https://habr.com/ru/post/582678/
13. Освітлення в Unity [Електронний ресурс]// learn.unity.com – 2021. - Режим доступу до ресурсу: https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering#
14. Освітлення в Unity [Електронний ресурс]// learn.unity.com – 2021. - Режим доступу до ресурсу: https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering#
15. Skybox [Електронний ресурс]/ Samarth Dhroov // medium.com – 2021. - Режим доступу до ресурсу: https://medium.com/nerd-for-tech/change-the-skybox-in-unity-in-just-3-simple-steps-afe537369c1a
16. Використання систему управління версіями на прикладі Unity [Електронний ресурс]/ Rick Reilly// thoughtbot.com – 2017. - Режим доступу до ресурсу: https://thoughtbot.com/blog/how-to-git-with-unity

Додатки

Додаток А

Шейдер плавного переходу скайбоксів на мові ShaderLab

```
Shader "Skybox/Blended" {
    Properties{
        _Tint("Tint Color", Color) = (.5, .5, .5, .5)
        _Blend("Blend", Range(0.0,1.0)) = 0.5
        _FrontTex("Front (+Z)", 2D) = "white" {}
        _BackTex("Back (-Z)", 2D) = "white" {}
        _LeftTex("Left (+X)", 2D) = "white" {}
        _RightTex("Right (-X)", 2D) = "white" {}
        _UpTex("Up (+Y)", 2D) = "white" {}
        _DownTex("Down (-Y)", 2D) = "white" {}
        _FrontTex2("2 Front (+Z)", 2D) = "white" {}
        _BackTex2("2 Back (-Z)", 2D) = "white" {}
        _LeftTex2("2 Left (+X)", 2D) = "white" {}
        _RightTex2("2 Right (-X)", 2D) = "white" {}
        _UpTex2("2 Up (+Y)", 2D) = "white" {}
        _DownTex2("2 Down (-Y)", 2D) = "white" {}
        _DownTex3("3 Down (-Y)", 2D) = "white" {}
    }

        SubShader{
            Tags { "Queue" = "Background" }
            Cull Off
            Fog { Mode Off }
            Lighting Off
            Color[_Tint]
            Pass {
                SetTexture[_FrontTex] {  combine texture  }
                SetTexture[_FrontTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous}
            }
            Pass {
                SetTexture[_BackTex] { combine texture }
                SetTexture[_BackTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous }
            }
            Pass {
                SetTexture[_LeftTex] { combine texture }
                SetTexture[_LeftTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous }
            }
            Pass {
                SetTexture[_RightTex] { combine texture }
                SetTexture[_RightTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous}
            }
            Pass {
                SetTexture[_UpTex] { combine texture }
                SetTexture[_UpTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous}
            }
            Pass {
                SetTexture[_DownTex] { combine texture }
                SetTexture[_DownTex2] { constantColor(0,0,0,[_Blend]) combine texture lerp(constant) previous}
            }
    }

        Fallback "Skybox/6 Sided", 1
}
```