Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Інститут інформатики та радіоелектроніки,
Факультет комп'ютерних наук і технологій
(повне найменування інституту, факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр
(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ЛАНДШАФТУ ДЛЯ 3D ІГОР
RESEARCH OF PROCEDURAL TERRAIN GENERATION FOR 3D GAMES

Виконав: студент 2 курсу, групи КНТ-110м
Спеціальності 121 Інженерія програмного забезпечення
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Інженерія програмного забезпечення

Загоняйко А.А.
(прізвище та ініціали)
Керівник Дубровін В.І.
(прізвище та ініціали)
Рецензент Касьян М.М.
(прізвище та ініціали)

2021

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Інститут, факультет ІІРЕ, ФКНТ___________________________________
Кафедра програмних засобів_______________________________________
Ступінь вищої освіти___магістр____________________________________
Спеціальність 121 Інженерія програмного забезпечення_________________
(код і найменування)

Освітня програма (спеціалізація) Інженерія програмного забезпечення______
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
________________С.О. Субботін
“____” _______________2021 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ СТУДЕНТА

____________________Загоняйка Артура Андрійовича__________________
(прізвище, ім’я, по батькові)

1. Тема проєкту (роботи) Дослідження процедурної генерації ландшафту для 3D ігор.____
Research of Procedural Terrain Generation for 3D games.
керівник проєкту (роботи)_Дубровін Валерій Іванович, к.т.н., професор
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від “5”_листопада_2021_року № _435__

2. Строк подання студентом проєкту (роботи)_ грудень 2021 року________

3. Вихідні дані до проєкту (роботи)__рекомендована література, технічне___ завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1.Дослідження проблеми процедурної генерації ландшафтів. 2. Дослідження засобів процедурної генерації ландшафту та необхідних елементів. 3. Проєкт програмної системи генерування 3D ландшафтів. 4. Аналіз створенної процедурної генерації 3D ландшафту.

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

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	Дубровін В.І., професор		
Нормоконтроль	Липовець М.В.		

7. Дата видачі завдання __3 вересня 2021 р.______ __________________________

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Дослідження проблеми процедурної генерації ландшафтів	2–3 тижні	Розділ 1
3	Дослідження засобів процедурної генерації ландшафту та необхідних елементів	4-5 тиждень	Розділ 2
4	Проєкт програмної системи системи генерування 3D ландшафтів	6–9 тижні	Розділ 3
5	Аналіз створенної процедурної генерації 3D ландшафту	10-11 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї. Нормоконтроль та рецензування	12-15 тиждень	Додатки
7	Захист роботи.	16 тиждень	

Студент __________ ___Загоняйко А.А______
(підпис) (прізвище та ініціали)

Керівник проєкту _____________ ____Дубровін В.І.______
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра: 132 с., 85 рис., 2 дод., 20 джерел.

ПРОЦЕДУРНЕ ГЕНЕРУВАННЯ, UNITY3D, C#, TERRAIN, HEIGHT MAP, MESH.

Об'єкт дослідження – програмні засоби генерування ландшафту.

Предмет дослідження – комп'ютерні ігри з повноцінним або частковим застосуванням алгоритмів процедурної генерації.

Мета роботи – створення програмної системи для процедурного генерування ландшафту з налаштуванням параметрів генерування самого ландшафту, а також різного роду рослин та природних об'єктів.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, середа розробки Unity Engine 3D 2019 та Visual Studio 2019, мова програмування C#, персональний комп'ютер з процессором AMD Ryzen 5 1400 під управлінням операційної системи Microsoft Windows 10.

Результати. Створено програмний Windows-застосунок, керований клавіатурою та мишкою, з яким взаємодіє користувач для налаштування параметрів генерування.

Висновки. Розроблено програмну систему яка за допомогою алгоритмів процедурної генерації створює елементи ландшафту які потенційно можливо використовувати у майбутніх відео іграх або симуляціях.

Галузь використання – процедурне створення ландшафту для різного роду ігр про виживання або стратегій, або для створення базовогу ландшафту для гри з подальшою корекцію за допомогою Unity3D. А також створення красивих природних ландшафтів для демонстрацій.

Економічна ефективність. Термін розробки ландшафту суттєво зменшується, відповідно до цього зменшуються також і витрати на сам проєкт.

ABSTRACT

Explanatory note to the qualifying work of the master: 132 pages, 85 figures, 2 appendixes, 20 sources.

PROCEDURAL GENERATION, UNITY3D, C #, TERRAIN, HEIGHT MAP, MESH.

The object of research - software for landscape generation.

Subject of research - computer games with full or partial usage of procedural generation algorithms.

The purpose of the work is to create a software system for procedural landscape generation with the adjusting of the landscape generation parameters itself, as well as various plants and natural objects.

Materials, methods and technical means: structural and object-oriented programming, development environment Unity Engine 3D 2019 and Visual Studio 2019, C # programming language, personal computer with AMD Ryzen 5 1400 processor running Microsoft Windows 10 operating system.

Results. A Windows-based keyboard-and-mouse-controlled software application has been created that the user interacts with to configure generation settings.

Conclusions. A software system has been developed that uses procedural generation algorithms to create landscape elements that can potentially be used in future video games or simulations.

Scope - procedurally creating a landscape for different kinds of survival games or strategies, or to create a basic landscape for the game with subsequent correction with the help of Unity3D. As well as the creation of beautiful natural landscapes for demonstrations.

Economic efficiency. The term of landscape development is significantly reduced, accordingly, the costs of the project itself are also reduced.

ЗМІСТ

С.

Перелік скорочень та умовних познак 9
Голосарій 10
Вступ 12
1 Дослідження проблеми процедурної генерації ландшафтів 15
1.1 Загальні відомості про процедурне генерування 15
1.2 Приклади задач процедурної генерації 15
1.3 Використання процедурної генерації в розробці гри The Witcher 3 22
1.4 Шари для процедурних біомів у грі Horizon Zero Dawn 26
1.5 Процедурна генерація світів у грі No Man's Sky 28
1.6 Цільові властивості генерації 33
2 Дослідження засобів процедурної генерації ландшафту та необхідних елементів 35
2.1 Основні методи процедурної генерації приміщень 35
2.1.1 Метод генерування базового ландшафту – Perlin Noise 35
2.1.2 Метод генерування областей на ландшафті – Voronoi Diagram 42
2.1.3 Метод генерування ландшафту – Midpoint Displacement 40
3 Проєкт програмної системи генерування 3D ландшафтів 62
3.1 Архітектура системи 62
3.1.1 Архітектура алгоритмів генерування ландшафту 62
3.1.2 Архітектура алгоритмів покращення зовнішнього вигляду ландшафту ... 64
3.2 Засоби реалізації 66
3.2.1 Середовище розробки Microsoft Visual Studio 2019 Enterprise 66
3.2.2 Мова програмування C# 66
3.2.3 Ігровий двигун Unity3D версії 2019.3.15 67
3.3 Модулі та алгоритми 68
3.3.1 Обмеження програмної реалізації 68
3.3.2 Побудова 3D ландшафту 68

3.3.2.1 Алгоритм класичного шуму Перліна………………………….68

3.3.2.2 Алгоритм діаграми Вороного…………………………………...72

3.3.2.3 Алгоритм зміщення середньої точки…………………………...80

3.3.2.4 Алгоритм згладжування………………………………………...85

3.3.3 Процедурне покращення зовнішнього виду 3D ландшафту 87

3.3.3.1 Алгоритм текстурування террейну…………………………….87

4 Аналіз створенної процедурної генерації 3D ландшафту 91

4.1 Проблема оптимізації рішення 91

Висновки 94

Перелік джерел посилання 95

Додаток А Текст програми 97

Додаток Б Слайди презентації 124

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

AAA – triple-A game;

C# – c sharp language;

PC – personal computer;

PCG – procedural generation.

ГОЛОСАРІЙ

Процедурна генерація (англ. Procedural generation, скор. PCG) - автоматичне створення ігрового контенту за допомогою алгоритмів. PCG є програмне забезпечення, яке може створювати ігровий контент самостійно, або спільно при взаємодії з гравцями або геймдизайнерами [1].

Unity - крос-платформова середа розробки комп'ютерних ігор, розроблена американською компанією Unity Technologies. Unity дозволяє створювати додатки, що працюють на більш ніж 25 різних платформах, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-додатки та інші [2].

AAA («три A», англійською вимовляється як «triple-A») — умовна підмножина відеоігор, що створюються й розповсюджуються середніми й великими видавництвами, що зазвичай мають багато коштів на розробку й рекламу. Чітких мірил належності певної гри до підмножини AAA немає, тож зазвичай ідеться про клас передових ігор з великим бюджетом, розробка яких пов'язана зі значним економічним ризиком і потребою високих показників збуту задля забезпечення прибутковості [3].

Heightmap - двовимірний масив, кожен елемент якого інтерпретується як висота. Часто використовуються в програмах створення ландшафтів (Terrain), щоб зберігати інформацію про висоту кожної точки місцевості. Використовуються також у технології бамп-мапінгу [4].

Terrain – Unity Engine включає вбудований набір функцій ландшафту, які дозволяють додавати ландшафти до вашої гри. У редакторі можна створити кілька плиток ландшафту, налаштувати висоту за допомогою heightmap або зовнішній вигляд вашого ландшафту і додати до нього дерева або траву. Під час виконання Unity оптимізує вбудований рендеринг ландшафту для підвищення ефективності [5].

Mesh - це сукупність вершин, ребер та граней, які визначають форму багатогранного об'єкта у тривимірній комп'ютерній графіці та об'ємному

моделюванні. Гранями зазвичай є трикутники, чотирикутники або інші прості опуклі багатокутники (полігони), так як це спрощує рендеринг, але сітки можуть також складатися з найбільш загальних увігнутих багатокутників, або багатокутників з отворами [6].

ВСТУП

У даний період часу в нашому сучасному світі ігрова індустрія займає не останнє місце в переліку розваг. З розвитком ігрової індустрії виникло все більше та більше жанрів, але також почали виникати і проблеми з розробкою та придумуванням чогось нового та інноваційного.

На даний момент існує дві основні проблеми. Перша – необхідність в створенні ігрового контенту за маленький проміжок часу. Тобто необхідність за мінімально можливий проміжок часу створити програмний продукт, який добре працює, в цьому нам і допомагає автоматизація процесів створення. Друга проблема – це якість обробки ігрових об’єктів, а саме відображення всіх об’єктів максимально реалістично або у відповідному стилі. Але поки що з цією проблемою може найкраще справитися лише людина, не дивлячись на спроби створити нейронні мережі які б могли генерувати такий контент.

Щорічно ігрова індустрія в світі збільшує свої обороти, застосовуючи нові технології, збільшуються бюджети ігор та намагаючись зробити ігри якумога більш схожими на наш реальний світ. Але нажаль щоб створити чудову гру потрібно мати великі ресурси, як грошові так и людські. Лиш одиниці здатні зробити дещо інноваційне з невеликими витратами. І це є основною перепоною для розробників ігор з невеликими бюджетами. На даний момент ми можемо виділити три основні підходи до створення відео ігор. Перший – ігри створені повністю людською працею, а саме художники та дизайнери працюють над створенням різного роду моделей та текстур для гри, завдяки стандартним програмам для створення ресурсів цього роду. Другий – напівавтоматичний, коли для створення контенту використовуються додаткові інструменти, які можуть прискорити час створення різноманітних моделей та текстур, та видалити з процесу рутинну частину роботи. Третій – це майже повністю, а інколи і повністю автоматичний спосіб створення. Такий метод включає в себе або складні алгоритми генерування, або нейронні мережі, але нажаль такий спосіб не

менш легкий у використанні, та часто не задовольняє очікуваний результат. Усі ці варіанти авжеж мають свої недоліки та переваги. Перший варіант надає найкращий результат, адже весь процес контролюється людиною и виключає більшу частину похибок. Але найбільшим його недоліком є дороговизна, як стосовно витраченого людського ресурсу, так і стосовно коштів. Тому такий підхід найчастіше використовують великі ігрові студії, які мають на це ресурси, та називаються ААА[3]. Другий варіант виступає у якості балансу між загальним затраченим часом, та людино-годинами, але результат вже виходить не таким якісним, як у першому варіанті. Третій варіант не є найпростішим, як вже говорилося раніше, та результати його виконання не завжди задовольняють якості, але не дивлячись на це його потенціал доволі великий. Основна перевага процедурного генерування над звичайною працею людини у тому, за допомогою складних автоматизованих алгоритмів може генеруватися великий обсяг ігрового контенту, від персонажів, та текстур до цілих світів, і все це без участі людини. Такий підхід може значним чином розширити варіативність ігри та завжди підтримувати інтерес гравця. Але написана дипломна робота орієнтована на використання другого методу. Адже саме таким чином можливо отримати найкращий та оптимальний результат генерування, який буде одночасно швидким і якісним завдяки співпраці людини та машини.

В даній кваліфікаційній роботі була розібрана проблема процедурної генерації, а саме процедурної генерації ландшафту. Основна ідея в тому, що завдяки можливості змінювати параметри рослин, каміння, типів текстуруваня поверхні, критеріїв алгоритмів створення висот, а також базової води та берегів, програма зможе створювати натуральні ландшафти. Це відкриває можливість у майбутньому створювати різні правила генерування, та модифікувати програму таким чином, щоб створювались повноцінні ігрові світи.

Метою дослідження є вивчення методів процедурної генерації ландшафтів автоматизованим чином, а також розробка власної системи для створення ландшафтів в реальному часі.

Об’єктом дослідження є природні правила створення ландшафтів та їх складових, задля досягнення максимальної схожості.

Предметом дослідження є автоматизоване генерування ландшафтів.

Для розв’язання представлених завдань використовуються такі методи дослідження:

– аналіз джерел про процедурне генерування ландшафтів і способи вирішення цієї проблеми;

– проведення аналогії серед існуючих на ринку ігор з використанням систем генерування ландшафту;

– обробка отриманих результатів;

– порівняльний аналіз програмних продуктів.

Одержані результати є наочним відображенням переваг та недоліків автоматизованих систем створення контенту. Було проведено аналіз предметної області та на основі цього було розроблено програмну систему здатну генерувати ландшафти за заданими критеріями генерування. Виходячи з цього ми маємо можливість до створення ігор на основі цієї генерації, або базової структури ландшафту для гри, яка може бути відредагована в майбутньому.

Практичне значення одержаних результатів таке:

– готовий програмний продукт, який генерує ландшафти у 3D;

– проведений аналіз методів оптимізації процедурного генерування;

– проведений аналіз існуючих рішень.

1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ЛАНДШАФТІВ

1.1 Загальні відомості про процедурне генерування

В цьому розділі буде йти мова про процедурне генерування контенту та його різновиди. Під контентом розуміється створення рівнів гри, карти ігрового світу, правил гри, текстур, сюжетів, предметів, квестів, музики, зброї, транспортних засобів, персонажів і інше. В рамках цієї роботи іграми вважаються комп'ютерні відео ігри, настільні ігри, карткові ігри, головоломки та інші. Головним критерієм при створюванні такого роду контенту є те, що він має бути повністю іграбельним – під іграбельністю мається на увазі що гравець повинен бути в змозі пройти створений рівень, мати можливість використовувати згенеровані предмети, взаємодіяти з процедурно згенерованими персонажами тощо інакше згенерований контент неможна вважати успішно згенерованим [7].

Терміни «процедурна генерація» відноситься до комп'ютерів, адже процедурна генерація являється програмою та має бути запущено у вигляді процедури на комп'ютері, після виконання якої ми отримуємо якийсь контент для подальшого використання у грі. При цьому з такої процедура може складатися цілком уся гра або вона може бути лише інструменту який використовує гейм дизайнер для досягнення бажаного результату під час розробки гри [7].

1.2 Приклади задач процедурної генерації

Можно навести декілька конктретних прикладів використання процедурної генерації: один з найкращих прикладів генерації без участі людини - це генерація підземель в пригодницьких іграх наприклад таких як серія ігор The Legend of Zelda, в іграх такого роду ігровий світ створюється

при кожному запуску. Також процедурну генерацію можна використовувати з метою створення нових версій озброєння в іграх жанру RPG, така система може генерувати нові типи та варіації озброєння базуясь на деяких правилах, наприклад залежно від локації, або різних статистик ігрока. Ще одним застосуванням для цієї технології є генерування цілих головоломок, але це надзвичайно складний процес, який потребує великих зусиль та складних розрахунків аби згенерована головоломка була успішно розв’язуємою. Загальний алгоритм процедурної генерації працює таким чином внутрішня процедура ігрового двигуну швидко генерує на ігровій мапі рослини, ландшафт, заздалегідь задані або згенеровані у процесі додаткові об’єкти; такий інструмент, дає можливість створювати мапи для стратегічних ігор або ігор виживачів, та в наступних етапах за допомогою різноманітних запитів і заданих змінах перераховує мапу для її поліпшення та згідності з бажаним результатом, а також пропонує варіанти, що дозволяють зробити мапу більш збалансованою, цікавою та реіграбельною що є основними критеріями успішної гри. У той же час, простий редактор карт, штучний інтелект для настільної гри або інструмент інтеграції створеного контенту не відносяться до процедурної генерації, адже вони керуються людиною, або замість цілковито нового контенту генерують передбачуваний та заздалегідь заданий розробником.

Метою використання процедурної генерації може бути створення ігрового контенту без участі людини (що може бути як менш затратно, так і допомагати геймдизайнер у вирішенні їхніх завдань), розробка інших типів ігор (поліпшення показників різноманітності і реіграбельності), адаптація ігор під гравця «на льоту», поліпшення контенту за допомогою алгоритмічних рішень, а також формалізація геймдизайну як широку наукову задачу [7].

Перші широко відомі застосування процедурної генерації відносяться до початку 1980-х років, коли при обмежених ресурсах комп'ютерів неможливо було створювати великі і різноманітні світи які би були наповнені

згенерованим контентом, тому перші застосування були пов’язані з 2D іграми, та такі операції не були навантажені великою кількістю обчислень та правил генерування. Характерними прикладами таких ігор є Rogue і Elite [7], скріншоти яких зображено на рисунках 1.1, 1.2.

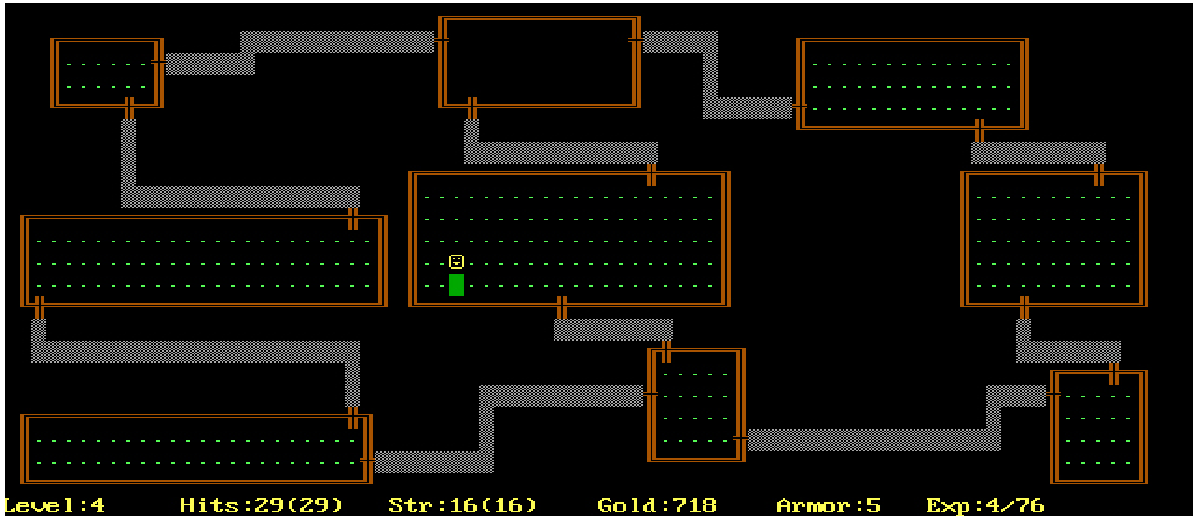


Рисунок 1.1 - Згенероване підземелля гри Rogue

Рисунок 1.2 - Генерування випадкової системи зірок в грі Elite

З часом комп’ютери розвивалися та їх ресурс ставав все більшим. Це відкрило двері для використовування процедурної генерації в великих комерційних іграх які здобули великої популярності завдяки цьому. Адже інтерес до гри збільшується якщо кожен гравець отримує якийсь новий ігровий досвід від проходження та може декілька разів перепройти свою улюблену гру отримуючи нові емоції та події. Серед таких ігор був рольовий бойовик Diablo від компанії Blizzard. В цій грі процедурна генерація застосовується для створення карт, числа монстрів та предметів. Процедурна генерація тут не створювала цілковита нові предмети або рівні. Але застосовувалася як допоміжна функція, яка автоматично вираховувала нові характеристики для озброєння або редагувала деякі елементи ігрових рівнів.[7], на рисунку 1.3 зображено процедурно згенероване підземелля.

Рисунок 1.3 - Гра Діабло

Серед усіх ігор того часу найбільш виділилася гра Spore. В ній розробники застосовували увесь потенціал процедурної генерації наскільки це було можливо в той час. В Spore процедурно генерувалося майже усе

навколо. Було сгенеровано тисячі світів, які у формі сферичних планет були розкидані по галактиці в системах. Завдяки інструменту процедурної генерації розробники створили доволі багато стартових інопланетних істот а однією з головних особливостей гри було генерування випадкової анімації згенерованих істот [7]. Приклад процедурного генерування зображено на рисунку 1.4.

Рисунок 1.4 - Гра Spore з випадково згенерованими ворогами

Серед ігор стратегій першою за своїм масштабом виділилась серії ігор Civilization. В ній процедурна генерація була основним елементом геймплея, адже для кожної нової та унікальної партії потрібно було генерувати нову мапу. Мапа складалася с тайлів, які спочатку були у формі чотирикутників, а пізніше виросли до шестикутників з приходом 3D. Підхід до такого генерування потребувал створення різноманітних сетів правил за якими будуть генеруватися тайли базуючись на сусідних тайлах. Приклад такої генерації зображено на рисунку 1.5.

Рисунок 1.5 - Редактор карт в грі Civilization 1

В грі Minecraft процедурна генерація використовується для створення ігрового світу [7]. Однією з основних особливостей при створенні гри був алгоритм створення системи печер, які повинні бути різноманітними, органічно поєднуватися з іншими печерами та мати свої входи, які мають вбудовуватися як в звичайний ландшафт так і в гористий. Іншою проблемою була оптимізація, оскільки згенерований світ надзвичайно великий, до границь якого неможливо дійти навіть за декілька місяців. Тому була застосована технологія підгрузки мапи під часи гри тобто у ран таймі за допомогою чанків (спеціальних зон невеликого розміру). Приклад згенерованого відкритого світу ми можемо бачити на рисунку 1.6. На ньому зображено фрагмент мапи з видом згори, на якому ми можемо побачити якими цікавими та різноманітними патернами згенерувалася ця частина світу.

Рисунок 1.6 - Карта світу в Minecraft

Також існує аналогічна майнкрафту гра яка називається Terraria. Вона також базуєтся на процедурній генерації мапи кожен раз при запуску нової гри. В основі лежить також шум, але оскільки це 2D гра, то шум використовується як у генеруванні по вісі X, для генерування біномів, так і для генерування по вісі Y для генерування печер, та шарів різних порід під землею. Приклад згенеравонного світу представленний на рисунку 1.7.

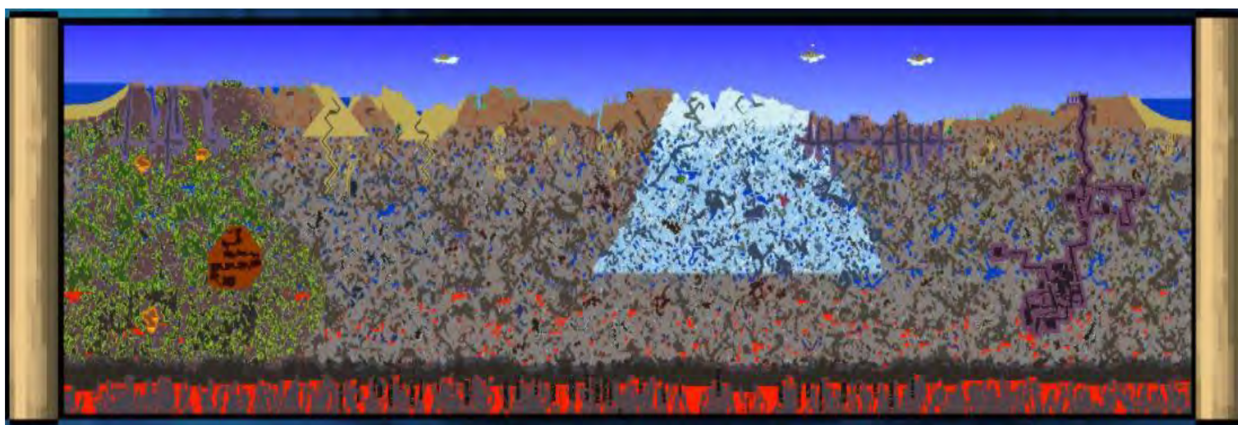


Рисунок 1.7 - Карта світу в Terraria

1.3 Використання процедурної генерації в розробці The Witcher 3

Один із сучасних прикладів процедурної генерації можна побачити у докладі зробленому розробниками компанії CD Project Red, які створили одну з найпопулярніших ігор The Witcher 3 [8]. Значною проблемою при розробці гри було створення великих відкритих просторів. У компанії CD Project Red на той час було недостатньо художників та дизайнерів, для створенням відкритих просторів, адже карта гри надзвичайно велика та для повного її створення руками було б витрачено надзвичайно велика кількість часу, або була б потрібна дуже велика команда дизайнерів. Така проблема примусила розробників шукати рішення для пришвидшення цього процесу. Одним із запропонованих рішень було створення інструментів для генерування ландшафтів та рослинності. Таке рішення стало найкращим з точки зору затраченого на нього часу та ресурсів, та отриманого результату.

Програмісти розробили спеціальний алгоритм, він міг розпізнавати два типи поверхонь: природні та рукотворні. До природних відносилися ґрунт, пісок, каменисті елементи, а до рукотворних – цеглу, бруківку, дерев’яні поверхні. Це дозволило по-різному накладати ефекти на будь-які типи матеріалів а алгоритм в свою чергу підгоняв їх за заданими параметрами. Наприклад, сніг міг виглядати порізно ми залежно від того на якій поверхні він знаходився, на звичайній стіні або на грунті. За допомогою цього розробники програмного продукту змогли отримати різноманітні красиві т натуральні ефекти, наприклад такі як плавні переходи між текстурою бруківки та природною текстурою землі, так щоб можна було розгледіти контури деяких булижників, рисунок 1.8.

Використання процедурної генерації дозволило розставити по поверхні текстури невеликі об’єкти, такі як камені та траву. Малювати та налаштовувати такі об’єкти під кожен тип поверхні або локацію надзвичайно затратно.

Рисунок 1.8 - Перехід між текстурою землі та дороги у грі The Witcher 3

Отже завдяки процедурній генерації художникам довелось намалювати лише десять видів трави, а сам алгоритм вже змінював їх колір під колір поверхні на якій вони знаходилися. Для цього левел-артисти використовували пігментну карту – текстуру локації з видом зверху в низькій роздільній здатності. Колір конкретного поля чи лугу накладався на траву градієнтом, таким чином аби досягти натурального результату, а саме щоб ближче до кореня рослинність була кольору землі, в той самий час як завдяки градієнту чим ближче до верхівки рослин тим більш вона зберігала свій оригінальний колір і не порушувала відтінки плодів та суцвіть.

У грі «Відьмак» процедурно згенерована трава не перекривала вхідні текстури, а доповнювала їх. Для цього програму навчили виділяти ті місця, на яких може щось рости, наприклад якщо на старій дорозі є невелика кількість землі то в цьому місті може вирости трава. В процесі генерування художники мали змогу налаштовувати густину трави. Такий підхід додавав схожості з реальним світом, коли трава проростає серед бруківки. Цей параметр працював на двох рівнях. Спочатку алгоритм визначав наскільки плоска текстура трави перекривала землю, та чи може на ній рости трава

залежно від градусу нахилу. Потім він налаштовував саму кількість, плотність та висоту рослинності. В залежності від цих параметрів художники могли створити як голу скалу з маленькими вкрапленнями трави, так і заповнений травою холом. А також швидко налаштувати яка трава де повинна рости залежно від градусу нахилу поверхні. Приклад такого генерування трави наведено на рисунку 1.9.

Рисунок 1.9 - Генерування трави в грі The Witcher 3

Для створення дерев та кущів художники компанії CD Project Red використовували інший інструмент – пензлики з набором рослинності, рисунок 1.10. Їх можна було налаштовувати під конкретну локацію, вказуючи насиченість ґрунту водою, кількість сонячних променів і навіть напрям розповсюдження насіння.

Алгоритми пензлів розпізнавали особливості рельєфу. Наприклад вони визначали, в яких місцях вода під час дощу стікає по схилах долин – там ліс виростав більш густим. В процесі система порівнювала рельєф з рельєфом сусідніх локацій, щоб визначити траєкторію сонця і наскільки освітлена територія.

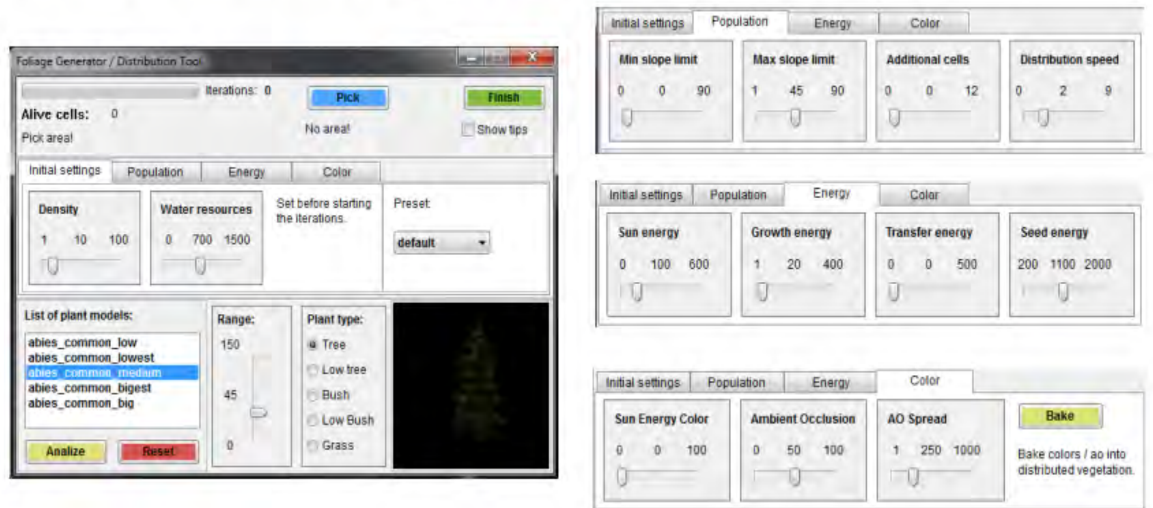


Рисунок 1.10 - Інструмент для створення рослинності у грі Witcher 3

Усі зібрані дані порівнювались з типами рослинності, які художники закладали до пензлів Набір дерев в пензлях південного лісу сильно відрізняється від пензлів для світлих гаїв на півдні, а ті в свою чергу, від рослинності на гірській місцевості.

Якщо для конкретного дерева не вистачало води або сонця, то пензель виключав цей тип із локації. Але також якщо ресурсів було достатньо, об’єкти все одно порівнювалися з заданими «ідеальними умовами».

Також ці пензлі давали різні результати при повторному використанні. Якщо художник використовує цю кість на одній ділянці один раз, у нього виходить легкий підлісок: чагарник, одне стояче дерево. Але якщо повторити використання пензля, ефект стає більш вираженим – чагарники розростаються, а дерев стає більше та їх висота зростає. На рисунку 1.11 наведено реальний скріншо з гри. Як можна побачити з цього скріншоту розробникам вдалося використати потенціал процедурної генерації не для самої генерації нового контенту під час гри, а для створення

спеціальних інструментів які допомогли значно пришвидшити процес розробки та отримати надзвичайно красиві та живописні пейзажі.

Рисунок 1.11 - Скріншот гри The Witcher 3

1.4 Шари для процедурних біомів у грі Horizon Zero Dawn

Працюючи над грою Horizon Zero Dawn, програмісти компанії Guerrilla Games також використовували процедурну генерацію [9]. Оскільки світ гри також доволі великий довелося розробляти алгоритм який би допоміг команді ландшафт-дизайнерів. Отже команді програмістів довелося також розробити алгоритм для створювання поверхонь. Вони змогли розробити такий алгоритм, який створював поверхні на основі шарів.

Головним шаром була карта рослинності, яка виглядала як карта висот у чорно-білому варіанті. На ціому шарі художники відмічали зони де міг рости ліс, також за допомогою цього визначалася густота лісу. Білим кольором позначалися самі ділянки та таким чином визначалося гущина дерев, чим біліший колір, тим більш густо ростуть дерева, рисунок 1.12.

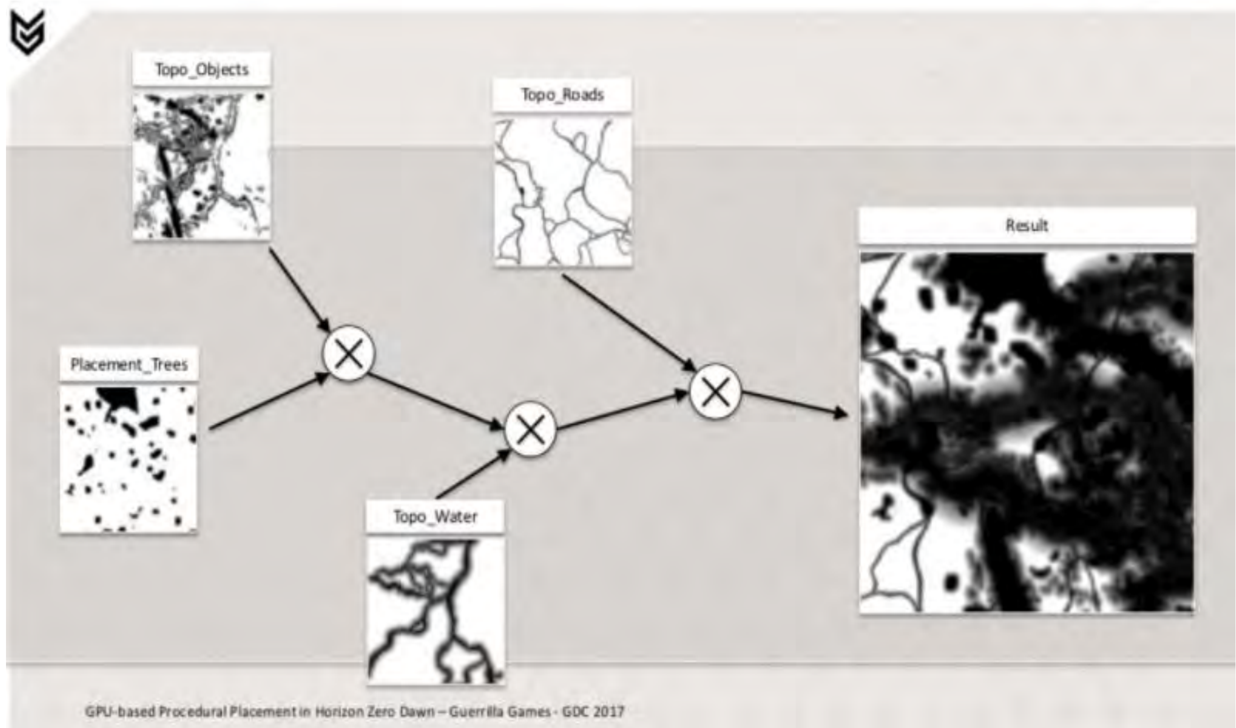


Рисунок 1.12 - Карта побудови ландшафту у грі Horizon Zero Down

Також було потрібно аби згенеровані локації виглядали доволі різноманітно, а не тільки просто працювали. Для цього художникам в студії довелося створити доволі багато різноманітних шаблонів для рослинностей різних типів. Такий процес однозначно займав деякий час, але це значно швидше ніж вручну малювати та моделювати усю мапу, та такий підхід дозволяє розпаралелити працю між художниками та дизайнерами. Усього три людини в Guerrila Games займалися непосредньо рослинами, та в результаті намалювали 500 типів рослин, а в цей самий час шаблонами займався лише один технічний художник, таким чином було зекономлено доволі багато часу та людських ресурсів.

Налаштувавши параметри генерації один раз, художники студії могли легко створювати визначені поєднання рослинності в різних частинах локацій. Алгоритм однаково успішно працював і з густими лісами, і з полями з чагарниками та травою.

Щоб моделі рослинності не заважали один одному, для кожної вказувався параметр footpoint, який задавав мінімальну дистанцію між

об'єктами одного типу. Для того щоб результат не виглядав штучним, художники налаштовували параметри які хаотично змінювали деякі характеристики та логіку розміщення.

В застосунку Horizon: Zero Dawn гравець досліджує багато різних біомів, - від середньоєвропейських лісів до засніжених тундр. Самі розробники назвали їх «екосистемами»: з точки зору інструменту це були просто набори налаштувань для створення локації.

Параметри біомів зачіпали не лише типи асетів, а й їх розповсюдження, а також погоду, візуальні ефекти і звукові ефекти. Тому результат генерування двох різних типів екосистем навіть на одній локації виглядав по-різному - джунглі на деякій горі сильно відрізнялися від звичайного лісу в тій же локації і за типами рослинності, а також по їх розповсюдженню. На рисунку 1.13 надано реальний скріншот з гри.

Рисунок 1.13 - Скріншот гри Horizon: Zero Dawn

1.5 Процедурна генерація світів у грі No Man's Sky

У 2017 році на ігровій конференції виступав розробник гри No Man's Sky та розповів усі подробиці генерування контенту у грі [10]. No Man's Sky

можно назвати покращеною версією Spore з точки зору застосування алгоритмів процедурної генерації. В цій грі фактично весь контент генерується процедурно, за виключенням перших п'яти галактик в яких розпочинає гравець. Тобто всі інші галактики, планети, кораблі, істоти які населяють планети та рослини генеруються процедурно. А сама гра нараховує сотні тисяч планет та супутників.

Гра були занадто велика та без процедурної генерації просто неможливо було обійтися. Авжеж з такої ідеєю одразу виникає ряд проблем навіть з застосуванням процедурної генерації. Основною метою було створення різноманітних та іграбельних світів. Для цього спочатку було обрано підхід який використовувся у майнкрафті, для генерування основного ландшафту планет, а саме алгоритм трилінійного фільтрування шума Перліна, рисунок 1.14.

Рисунок 1.14 - Скріншот гри Minecraft з не дуже натуральною генерацією гір

Але такий підхід швидко відпав, адже він добре підходив лише для генерування світів з кубиків, тобто не дуже реалістичних. Адже автор бачив планети саме в такому реалістично стилі замість не дуже природно натурального як у майнкрафті рисунок 1.15.

Рисунок 1.15 - Ландшафт якого прагнув отримати Шон Мюррей

Відкинувши варіацію шуму Перлина яка використовувалася в майнкрафті команда розробників продовжила пошуки. Другий варіант який вони вирішили використовувати були поля які складалися з шуму Перліна високої щільності, який використовувався в ігровому двигуні Space Engine. Результат такого шуму зображений на рисунку 1.16, але і цей підхід був відкинутий адже виглядав занадто одноманітно, деталі ландшафту часто повторювалися, а у схилів були однакові градієнти кольору.

Рисунок 1.16 - Ландшафт отриманий в Terrengine

Після цього проаналізувавши DEM Data Trainer – данні про висоту нашого реального ландшафту, команда розробників намагалася задати більше змінних на основі цих даних, для корегуванню шуму Перліна. Це дало більш наближений результат який зображено на рисунку 1.17, але і цей підхід не дуже задовільнив їх. Адже такий ландшафт виглядав не дуже цікавим у рамках ігри з сотнями тисяч планет.

Рисунок 1.17 - Шум Перліна із закономірностями розподілу, взятими з реального ландшафту

Але всі ці дослідження та тести не стали напрасними, завдяки цим даним програмісти зрозуміли, що шум в реальному світі розповсюджується експоненциально. Зібравши всі дослідження команді вдалося зроботи новий алгоритм який вони назвали Uber Noise. Він об’єднував у собі всі варіації шумів, які розробники збирали та тестували протягом декількох років. Алгоритм працює наступним образом за допомогою деформації області значень генерує ерозію схилів та висот, гірські хребти, плато, тераси, ланцюги пагорбів та різноманітні деталі. Після цього ще повинна залишатися похідна аналітичної функції, на основі якої можна генерувати наступний шар шуму. Таким чином декілька різних шумів накладуються шарами один поверх одного, а параметри для розрахунків беруться с результатів попереднього шару. На рисунках 1.18 та 1.19 представлені два скріншота з самої гри, які демонструють різноманітність ландшафтів генеруємих розробленим алгоритмом процедурної генерації.

Рисунок 1.18 - Скріншот з гри демонструючий генерування арок на ландшафті

Рисунок 1.19 - Скріншот демонструючий генерування різноманітних каньйонів

1.6 Цільові властивості генерації

Процедурно сгенерований контент повинен вирішувати ряд відповідних проблем, а також задовільна ти певним умовам щоб вважитися успішно згенрованим [7]. На практиці зазвичай виділяють та розглядають наступні властивості процедурної генерації:

– швидкість: в залежності від завдання вимоги до цього параметру можуть дуже сильно відрізнятися. Але в загальному випадку для генерування ігрового контенту, цей процесс повинен протікати якомога швидше та бути добре оптимізованим для задоволення потреб ігрового процесу;

– надійність: при створенні контенту генерація повинна виконуватися завжди зі 100% точністю. Під цим розуміється, що контент має мати можливість бути використоннами, або іншими слова бути іграбельним. Якщо в результаті генерації ми отримуємо лабіринт, який неможливо пройти, печеру в яку неможливо потрапити, або предмет який гравець ніколи не

зможе використати через погану генерації властивостей, тоді така генераціє не має права вважатися успішною та надійною;

– контролепригідність: це здатність контролювати контент який був згенерований системою виходячи з певної ситуації. Також це може досягатися за допомогою надання геймдизайнерам відповідної волі для впливання на згенерованний контент, але така властивість потрібна далеко не завжди, це скоріше як опціональна функція;

– різноманітність: такий генератор контенту який зможе створювати якомога більше різноманітного контенту, та він при цьому буду не схожий на інші генерації. Така властивість авжеж не безкінечна, але потрібно прагнути досягти максимально можливого числа комбінацій елементів контенту. Такий підхід додає ту саму реіграбельність та дарує гравцю відчуття новизни з кожним запуском, та допомагає затримати та зацікавити гравця;

– креативність і правдоподібність: мета генерації такого контенту, який виглядав би так, ніби його створила людина, а не генератор.

2 ДОСЛІДЖЕННЯ ЗАСОБІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ЛАНДШАФТУ ТА НЕОБХІДНИХ ЕЛЕМЕНТІВ

2.1 Основні методи процедурної генерації приміщень

В цьому розділі будуть описані найбільш підходящі рішення до проблем процедурної генерації терейну, а також більш глибоко розібрані їх основні особливості. А також розібрані які недоліки та переваги мають всі ці методи.

2.1.1 Метод генерування базового ландшафту – Perlin Noise

Перший метод який хотілось би розглянути детальніше – це Perlin Noise, або шум Перліна українскьою [11]. Цей алгоритм здобув надзвичайну популярність в багатьох сферах генерування, а не тільки в генеруванні ландшафту. Та можна сказати, що він являється базовим та основним алгоритмом для початкової генерації ландшафту та використовується всюди, без нього просто неможливо обійтись при процедурному генеруванні візуального контенту. За допомогою шуму Перліна можливо створювати доволі цікаві патерни ландшафту, та інших графічних об’єктів.

Шум Перліна був створений в 1983 році науковцем Перліном Кеном, та названий в його честь, також іноді його називають класичним шумом Перліна, адже після нього було створено ще декілька його різновидів [11].

Перейдемо до того як влаштований класичний шум Перліна. Цей алгоритм являється математичним алгоритмом, який в першу чергу був призначений до генерування процедурної текстури. Приклад генеруваня такої текстури ми можемо побачити на рисунок 2.1.

Рисунок 2.1 - Двомірний зріз тривимірного різновиду класичного шуму Перліна

Процес генерування шуму відбувається за допомогою генерування псевдо-рандомних чисел. Генерування псевдо-рандомних чисел це певний алгоритм, в результаті роботи якого ми отримуємо послідовність чисел які являються майже не залежними одне від одного, а також відповідають заданому розподіленню, частіше за все це рівномірне розподілення.

Класичний шум Перліна – це так званий градієнтний шум, який складається з набору псевдо-рандомних одиничних векторів, які знаходяться у певних точках в пространстві, та використовується для підвищення реалізму та графічної складності на поверхнях різноманітних геометричних об’єктів [11]. На рисунку 2.2 ми можемо спостерігати перший етап класичного шуму Перліна, а саме генерацію псевдо-рандомних одиничних векторів, які представлені в якості точок на графіку [12].

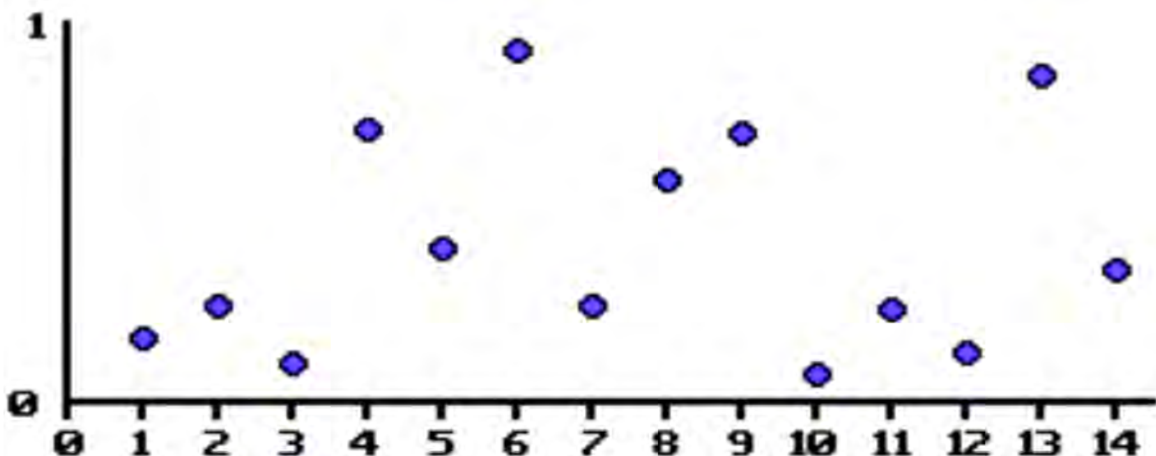


Рисунок 2.2 - Генерація псевдорандомних векторів класичного шуму Перліна

Наступний етап це інтерполяція отриманих значені за допомогою функції сглажуваня по цим точкам на графіці. Для того щоб згенерувати класичний шум Перліна в одномірному просторі ми повинні взяти кожну згенеровану точку та вирахувати для неї значення шумової хвилі, в цьому нам допоможе напрямок градієнта за яким була побудована точка [11]. На рисунку 2.3 ми можемо спостерігати результат такої інтерполяції побудованої за даним рисунку 2.2 [12].

Рисунок 2.3 - Інтерполяція згенерованних данних класичного шуму Перліна

Для подальшого генерування класичного шуму Перліна використовуються такі базові математичні поняття для хвиль на графіку як амплітуда, довжина хвилі та її частота. Довжиною хвилі називають відстань від однієї вершини до наступної сусідньої з нею. Амплітуда в свою чергу це висота хвилі на рисунку 2.4 [13]. На рисунку 2.5 представлено зображення амплітуди та довжини хвилі на класичному шумі Перліна, а червоні точки вказують на псевдорандомно згенеровані вектори. А частота вираховується за формулою (2.1) як відношення одиниці до довжини хвилі [12].

Рисунок 2.4 - Наглядна ілюстрація амплітуди та довжини хвилі звичайного графіку

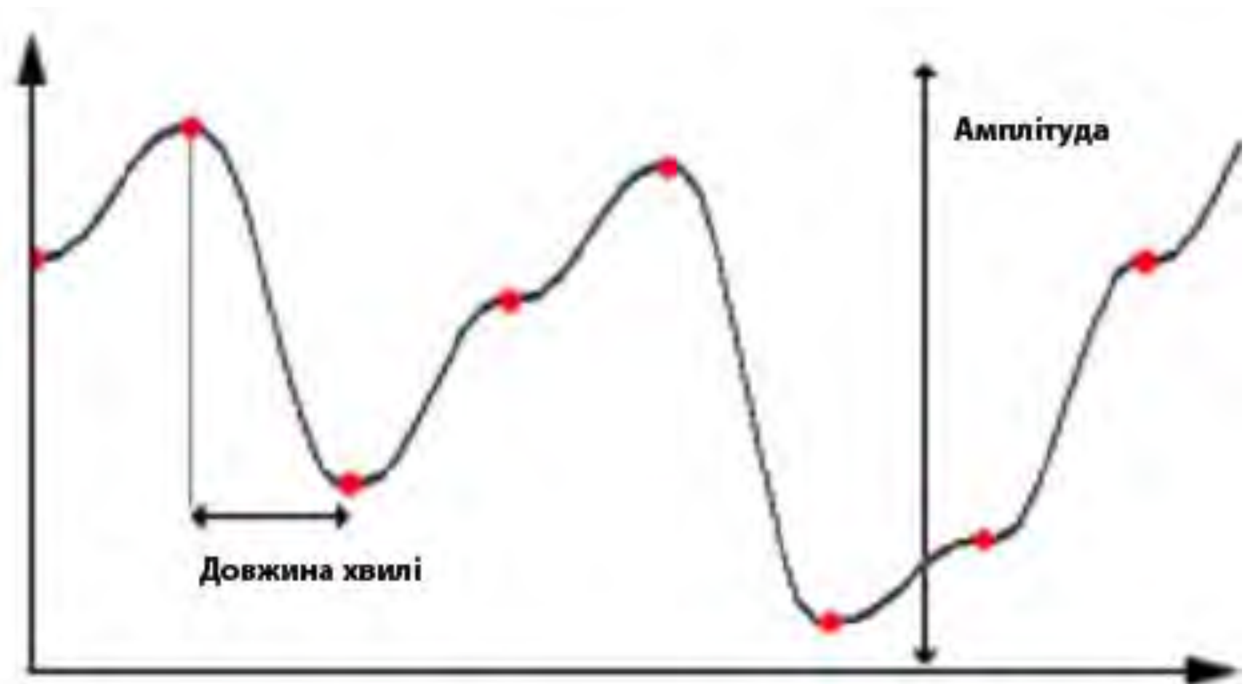


Рисунок 2.5 - Наглядна ілюстрація амплітуди та довжини хвилі на графіку побудованому за класичним шумом Перліна

$$frequency = \frac{1}{wavelength}, \quad (2.1)$$

де *frequency* – частота;

wavelength – довжина хвилі.

Наступним етапом для створення незвичайних та більш різноманітних є створення декількох шумів з різними амплітудою та частотою. Після того як такі шуми готові, ми можемо об’єднати їх в один шум для отримання надзвичайного результату. На рисунках 2.6-2.11 представлені різні класичні шуми Перліна в необ’єднаному вигляді.

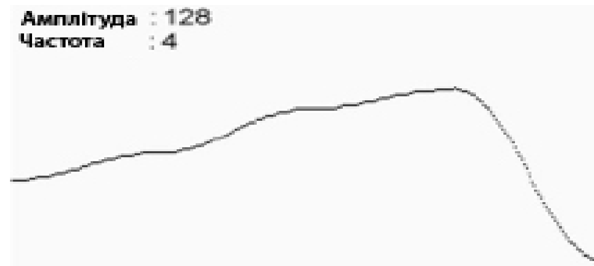


Рисунок 2.6 - Одиничний шум за амплітудою 128 та частотою 4

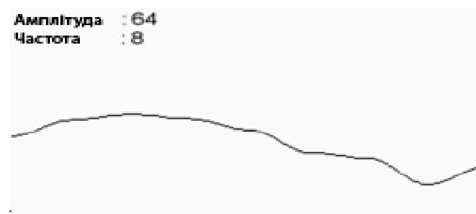


Рисунок 2.7 - Одиничний шум за амплітудою 164 та частотою 8

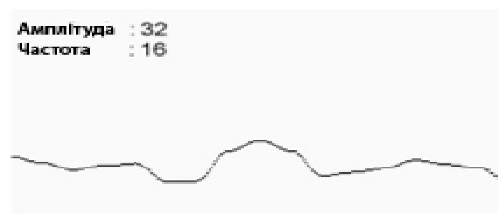


Рисунок 2.8 - Одиничний шум за амплітудою 32 та частотою 16

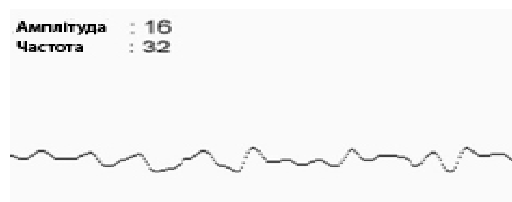


Рисунок 2.9 - Одиничний шум за амплітудою 16 та частотою 32

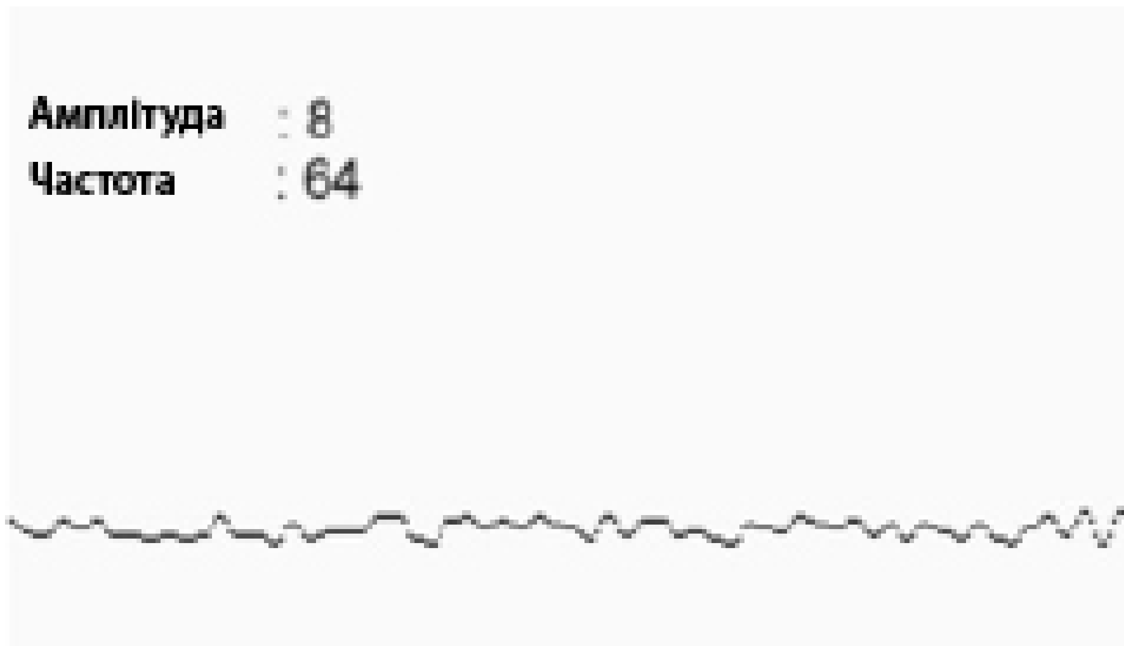


Рисунок 2.10 - Одиничний шум за амплітудою 8 та частотою 64

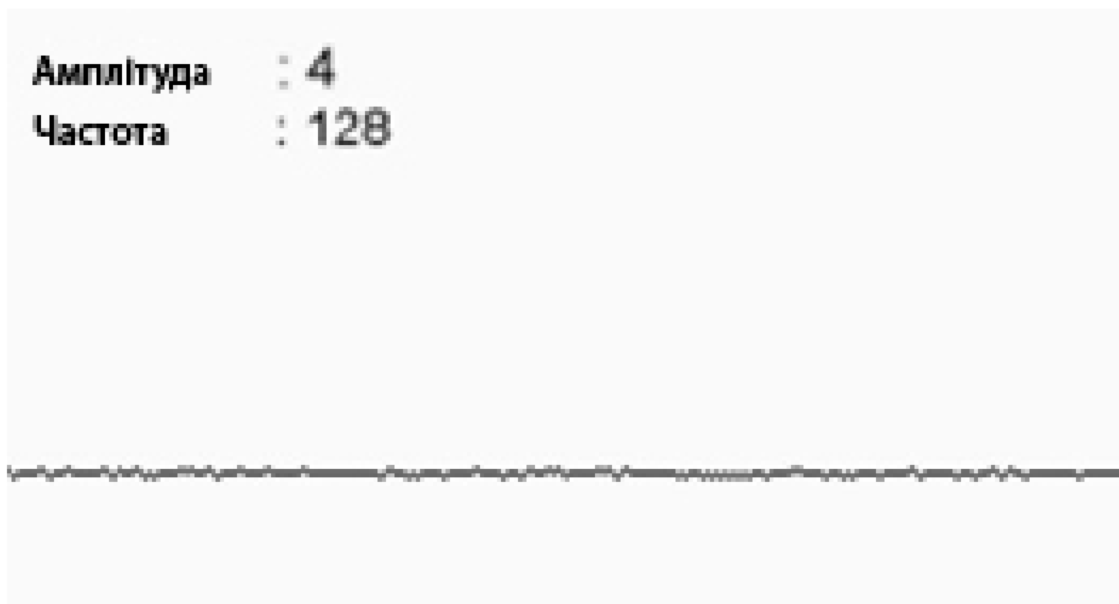


Рисунок 2.11 - Одиничний шум за амплітудою 4 та частотою 18

Вже на рисунку 2.12 ми можемо побачити який класичний шум Перліна у нас вийшов в результаті об’єднання.

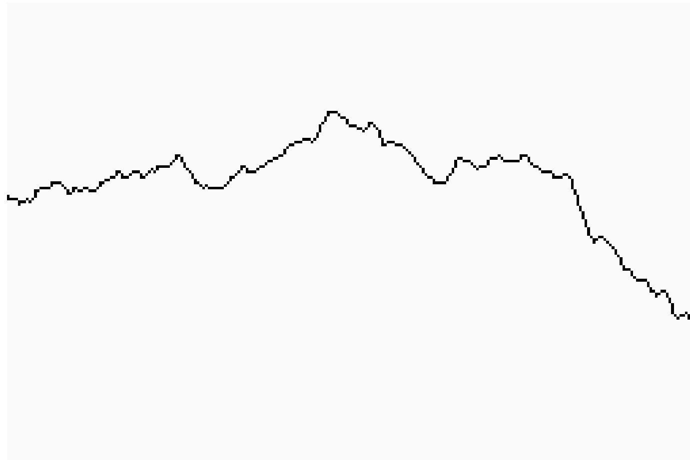


Рисунок 2.12 - Класичний шум Перліна отриманий в результаті об’єднання одиничних шумів

Отже як саме виходить об’єднання наших шумів в один класичний шум Перліна. Для цього виконується операція складання всіх отриманих класичних шумів Перліна. Але при цьому виникає складність у розрахунку частоти та амплітуди. Для таких розрахунків була введена нова величина яка називається стійкість, або Persistence англійскою [13]. Ця величина вираховується як різниця у зміні розміру, нашого поточного шуму та наступного. Також змінюються розрахунки частоти та амплітуди за формулами (2.2-2.3):

$$frequency = 2^i \,, \tag{2.2}$$

де i – i-та додана функція шума;

frequency – частота.

$$amplitude = persistence^i, \tag{2.3}$$

де i – i-та додана функція шума;

amplitude – амплітуда;

persistence – стійкість.

Що стосується окремих шумів які складаються в один шум, такі шуми називаються октавами. Така назва пішла з музики, адже так саме як і там кожен наступний шум має частоту вдвічі більше того, що йшов перед ним [11]. Ми можемо додавати скільки завгодно октав до нашого класичного шуму Перліна, але є декілька нюансів які слід враховувати при цих діях. По перше бувають такі моменти коли скомбінований шум досягає дуже великої частоти, через це в нас просто не вистачить пікселів на екрані для точного відображення всього шуму, а саме його дрібних деталей. По друге бажано вже не додавати багато шумових хвиль, амплітуда яких занадто мала, такі октави просто вже не зможуть оказувати достатнього впливу на кінечний результат.

Отже як висновки до класичного шуму Перліна ми можемо винести такі переваги та недоліки даного підходу. Являючись одним з основних алгоритмів генерування ландшафту він може слугувати дуже непоганою базою для первинного ландшафту. Також класичний шум Перліна можливо доволі просто контролювати та експериментувати для отримання різноманітних результатів які б задовольнили поставлену мету. З мінусів ми можемо винести, що без подальшої обробки в нас не вийде створити гарний ландшафт, а саме гористу місцевість з натуральними схилами та піками [12].

2.1.2 Метод генерування областей на ландшафті – Voronoi Diagram

Під час побудови ландшафту нам може знадобитися розбити згенерований ландшафт на зони, які умовно складаються с різних, лісів, полів або холмів. Або ж зробити декілька гористих піків віддалених на певну відстань, та горні схили яких зустрічаються в декількох точках. Для таких елементів нам ідеально підійде Діаграма Вороного.

Діаграма Вороного була винайдена у 1908 році Георгієм Феодосовичем Вороним, після того як він дослідив загальний n-мерний випадок цього

явища. Також цей алгоритм відомий під такими назвами як Мозаіка Вороного або Розбиття Вороного [14].

Діаграма Вороного складається с певної множини точок, ця множина є кінцевою, а точки розміщенні на поверхні у дозвільному порядку та місцях (рис. 2.13).

Рисунок 2.13 - Дозвільно розміщені точки до розбиття на Діаграму Вороного

Наступним кором є розбиття цієї області на зони навколо кожною з точок. Таке розбиття відбувається таким чином щоб кожна область створювала множну точок, які в свою чергу будуть більш близькими до одного з елементів множини наших основних точок на поверхні, ніж до будь якого іншого елементу цієї самої множини [14]. Виходячи з цього ми можемо бачити на рисунку 2.14 результат розбиття множини точок на області.

Існують декілька способів обчислення діаграми виходячи з побажань стосовно результуючою діаграми.

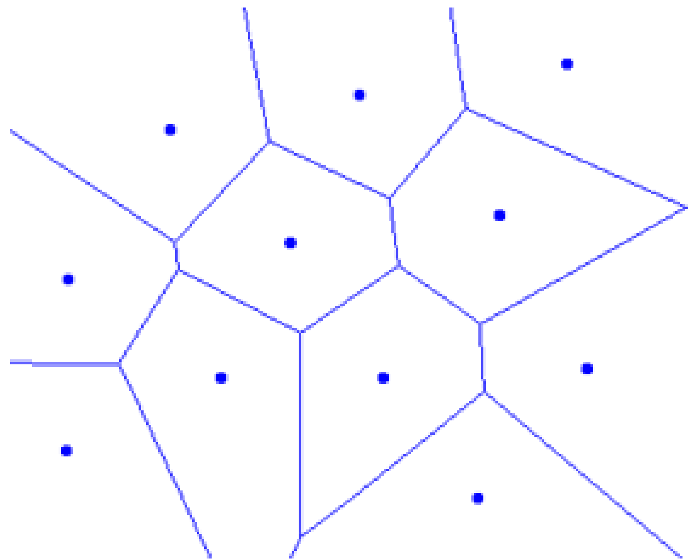


Рисунок 2.14 - Розбиття множини точок та отримання Діаграми Вороного

У найпростішому випадку, який ми продемонстрували на першому малюнку, нам дано множину точок {p1, ..., pn} яка є скінченною та знаходиться на евклідовій площині. У цьому випадку кожна ділянка pk є просто звичайною точкою, а її відповідна клітинка Вороного Rk розраховується та складається з кожної точки евклідової площини на якій вона знаходиться, відстань від якої до pk менша або дорівнює її ж відстані до будь-якого іншого pk на цій площині. Кожна така область виходить з розрахунку перетину півпросторів, а отже, виходячи з цього ми отримуємо опуклий багатогранник. Границі областей називають відрізки діаграми Вороного. До складу відрізків входять всі точки на площині, які рівновіддалені від двох найближчих точок. Точки які ми отримуємо на перетині відрізків Вороного називають вершинами або вузлами Вороного. До таких точок відносяться такі точки, які рівновіддалені від трьох (або більше) областей на площині [14].

Застосовуючи простий алгоритм ми повинні використовувати серединний перпендикуляр відрізка, який з’єднує певну пару точок p та q. Серединним перпендикуляром вважається пряма яка проходить через середину даного відрізку, а також у той самий час перпендикулярна цьому

відрізку. При застосуванні такого методу цей перпендикуляр розбиває нашу площину на дві нові на півплощини Hpq та Hqp. При цьому розбитті область Вороного точки p повністю належить до однієї з отриманих напівплощин, а область Вороного точки q повністю належить до протилежної напівплощини. Результат цього розбиття ми можемо бачити на рисунку 2.15 за допомогою перпендикуляру [14].

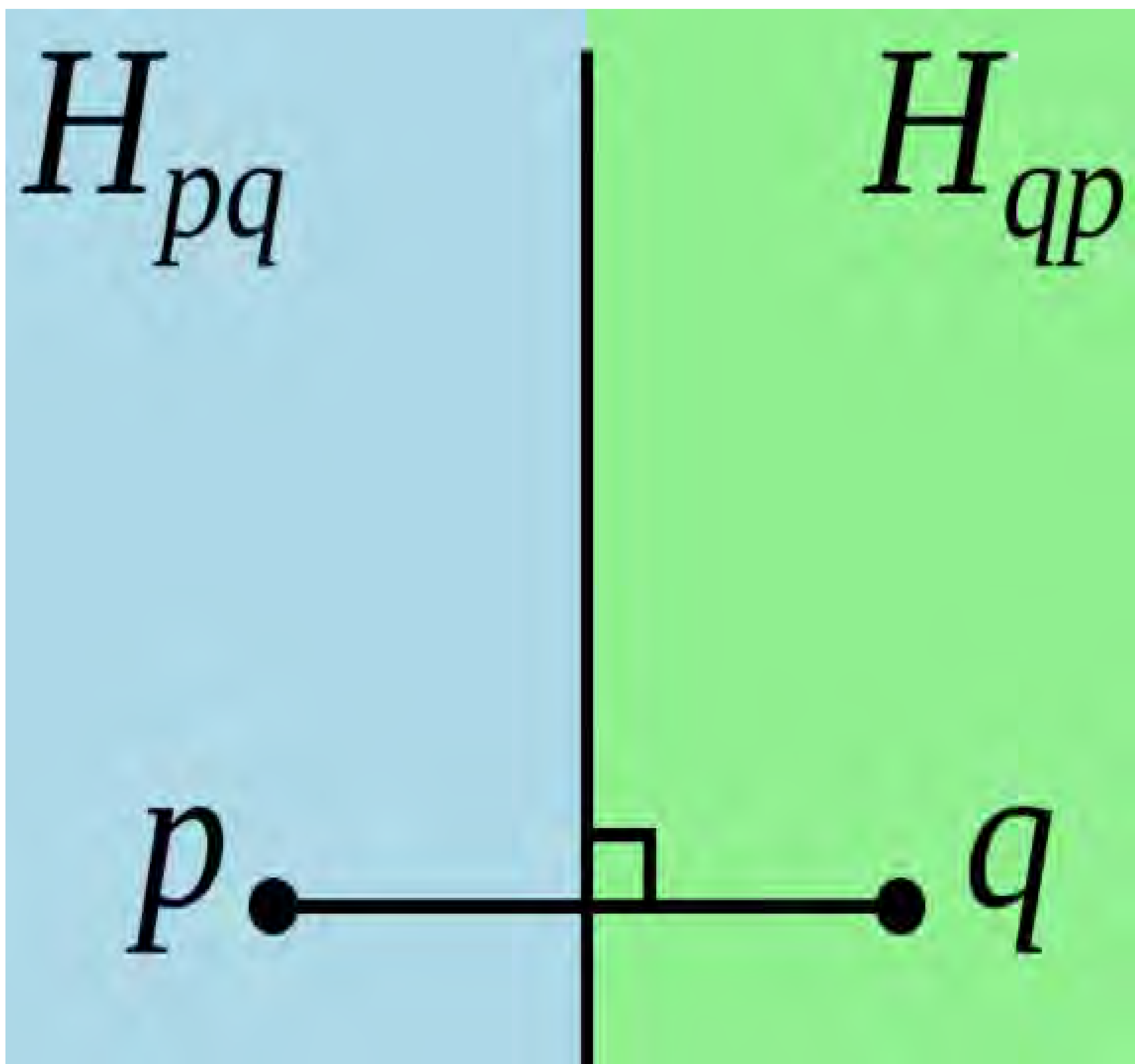


Рисунок 2.15 - Розбиття площини на дві напівплощини Hpq та Hqp за допомогою перпендикуляру та точок p і q

Таким чином область вороного V у точці p буде співпадати з перетинами усіх таких отриманих на півплощин Hpq. Та як результат

рішення зводиться до обчислення такого самого перетину для кожної точки p у скінченій множині [14]. Формула (2.4) представляє розрахунок методом перпендикуляру:

$$v_p = \bigcap_{q \in \frac{s}{\{p\}}} H_{pq}, \quad (2.4)$$

де V – область Вороного;

p та q – точки з кінцевої множини;

S – кінцева множина точок;

H – нова напівплощина.

Виходячи з цих розрахунків ми отримуємо обчислювальну складність для цього алгоритму яка становить $O(n^4)$.

Наступний алгоритм – це рекурсивний алгоритм. Його основна ідея полягає у використанні метода динамічного програмування. Суть цього метода полягає у розбитті комплексної задачі на декілька під задач для більш зручних обчислень. Метод динамічного програмування чудово підходить для задач які мають оптимальну підструктуру, тобто вони мають вигляд набору перекриваючих під задач, та складність таких під задач трохи менше за початкову основну задачу.

У нашому випадку ми маємо множину точок S на площині. Ця множина точок розбивається на дві підмножини S1 та S2. Для кожної з підмножин будується свої діаграма Вороного. Після побудови окремих діаграм вони знову об’єднуються в одну. Таке розбиття на підмножини здійснюється за допомогою прямої. Ця пряма в свою чергу розбиває нашу площина на дві на півплощини. Розбиття має робитися таким чином, щоб у кожній з отриманих напівплощин ми мали приблизно однакову кількість точок. Об’єднання двух діаграм Вороного складаючихся з множин S1 та S2 може бути виконано за час $O(n)$. Виходячи з цього ми можемо отримати

обчислювальну складність алгоритму, яка в свою чергу становитиме $O(n \log n)$ [14].

Також отриманний результат, а саме область Вороного описуємо відрізками залежить від обраної системи підрахунку відстані між точками на площині. Існує два типи відстані які ми можемо застосовувати для визначення областей Вороного. Перший це евклідова відстань. У математиці евклідовою відстанню може називатися відстань між двома точками в евклідовому просторі. Довжина відрізка прямої між двома точками становить евклідову відстан [15]. Цю відстань можна обчислити з декартової системи координат, за допомогою теореми Піфагора. Отже ми маємо розрахувати її як корень квадратний сум різниць у другій степені відповідних координат за формулою (2.5):

$$l_2 = d[(a_1, a_2), (b_1, b_2)] = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}, \qquad (2.5)$$

де l – відстань між двома точками;

d – відрізок між двома точками;

a та b – точки на площині.

Після розрахунку та отримання евклідових відстаней для усіх областе й Вороного ми отримуємо такий результат, схожий з тими що були представлені раніше (рис. 2.16).

Інший же спосіб порахувати відстань для розбиття на області Вороного це Манхетеньска відстань. Манхетеньска відстань була названа так завдяки структурі за якою побудавоний реальний Манхетен. Адже він розбит на квартали таким чином, що відстань від одного кварталу до сусіднього завжди однакова [16]. Це дає певні переваги перед евклідовою системою розрахунку відстані. Розберемо на прикладі з рисунку 2.17.

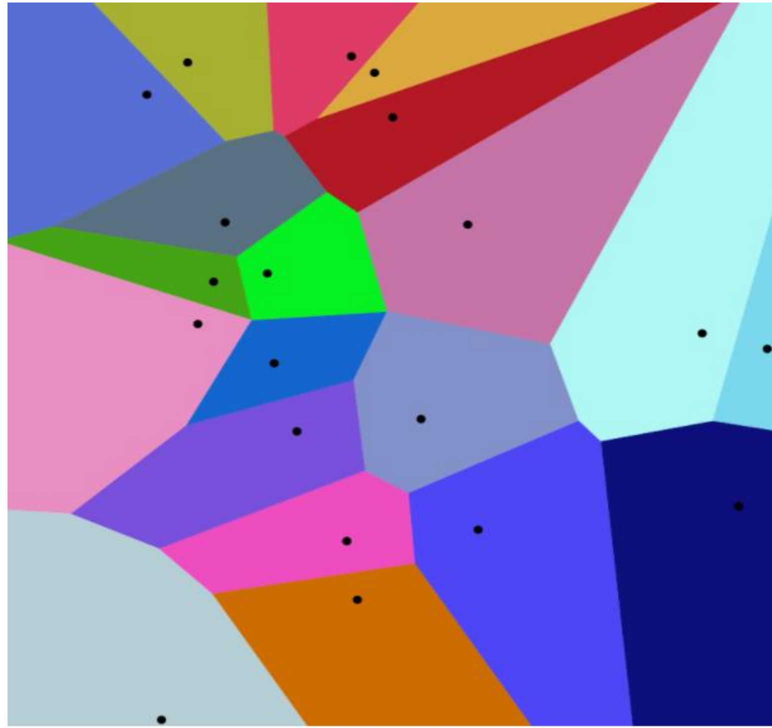


Рисунок 2.16 - Розбиття на площини на області Вороного за допомогою евклідової відстані

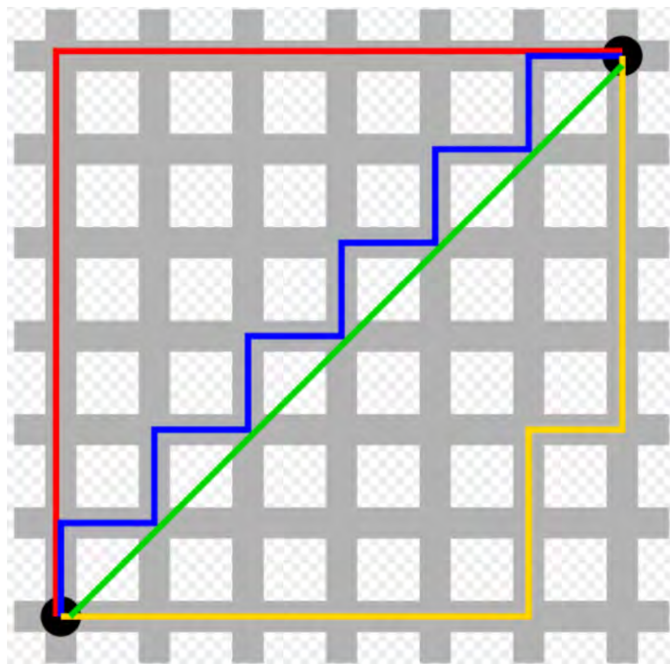


Рисунок 2.17 - Порівняльний приклад евклідової відстані та манхетеновської. Чорні точки – початок та кінець; зелена лінія – евклідова відстань; синя, червона та жовта лінії – манхетенська відстань

Отже з приклада ми можемо бачити, що манхетенська відстань дає нам набагато більше варіантів для найкоротшого шляху за точки А в точку В. Відстань між точками за жовтою, синьою та червоною лініями дорівнює 12 відрізків які довелося б пройти. В той час як за евклідовською відстанню ми не маємо жодного вибору и хочу шлях і виявився меншим $6\sqrt{2} \approx 8.5$ але це відстані напряму, без урахування можливих перепон у вигляді домів. Манхетеньска відстань розраховується як сума модулів різниць відповідних координаті двох точок за формулою формула (2.6) [16]:

$$d[(a_1, a_2), (b_1, b_2)] = |a_1 - b_1| + |a_2 - b_2|, \quad (2.6)$$

де d – відрізок між двома точками;

a та b – дві точки на площині.

Після розділення площини на області вороного Вороного за допомогою манхетеновської відстані ми маємо дещо інший результат відрізків областей. З рисунку 2.18 можна побачити, що границі областей Вороного дещо змінилися, отримавши більше відрізків які преломлюються у різних напрямках.

Виходячи з отриманного результату розбиття площини на області Вороного методом манхетенської відстані ми можемо зробити висновок, що такий підхід може на виході дати інакші патерни розбиття того ж самого скінченного набору точок.

Роблячи висновки виходячи з усіх описаних алгоритмів та властивостей алгоритму діаграми Вороного можна виділити такі переваги його застосування для генерування поверхонь.

Завдяки різноманітним алгоритмам розбиття на області Вороного ми можемо отримати доволі великий спектр для вибору методу розрахунків.

Рисунок 2.18 - Розбиття на площини на області Вороного за допомогою манхетеновської відстані

Через це ми можемо підібрати такий алгоритм, який найбільше підійде до розбиття на області які ми бажаємо побачити на нашому ландшафті. Але так само як класичний шум Перліна, діаграма Вороного не підходить для самостійного використання у формуванні ландшафту, адже навіть згенеровані таким чином гори будуть виглядати не натуральними.

2.1.3 Метод генерування ландшафту – Midpoint Displacement

Ми вже розглянули два важливих метода для процедурного генерування, а саме класичний шум Перліна та діаграму Вороного. Але як ми

вже вияснили самі по собі ці алгоритми не можуть видати задовільний результат для генерування реалістичних поверхонь, або потребують дуже точного налаштування. Через був винайдений один дуже ефективний алгоритм який називається Midpoint Displacement – або алгоритм зміщення середньої точки.

Алгоритм зміщення середньої точки вперше був використаним у 1982 році під час конференції siggraph, яка спеціалізується на обчислювальних методах комп'ютерної графіки, вченими Фурн'є, Фусселєм та Карпентером. Також такий алгоритм ще називають алгоритмом Diamond-Square через його особливості обчислювання [17].

Алгоритм зміщення середньої точки застосовується для змін значення мапи висот, з якої власно і складається ландшафт в ігрових двигунах. Мапа висот представляє собою двомірну матрицю яка визначає власне висоту певних точок нашого ландшафту. Беручи до уваги цю інформацію алгоритм зміщення середньої точки працює з двомірною сіткою.

Першим кроком ми маємо вибрати чотири початкові значення з нашої мапи висот, найчастіше беруться углові точки мапи висот, якщо ми хочемо охопити усю нашу мапу для процедурного генерування (рис. 2.19). Після того як ми обрали точки їм призначається початкова висота яка генерується випадковим чином [18].

Далі робота алгоритму зміщення середньої точки ділиться на два основні етапи, з застосуванням двох алгоритмів:

– метод діамантового кроку;
– метод квадратного кроку.

Власне через ці дві алгоритми алгоритм зміщення середньої точки і називають по іншому алгоритм Diamond-Square. Завдяки комбінації цих двох алгоритмів відкривається можливість для дуже успішної генерації різноманітних ландшафтів [17].

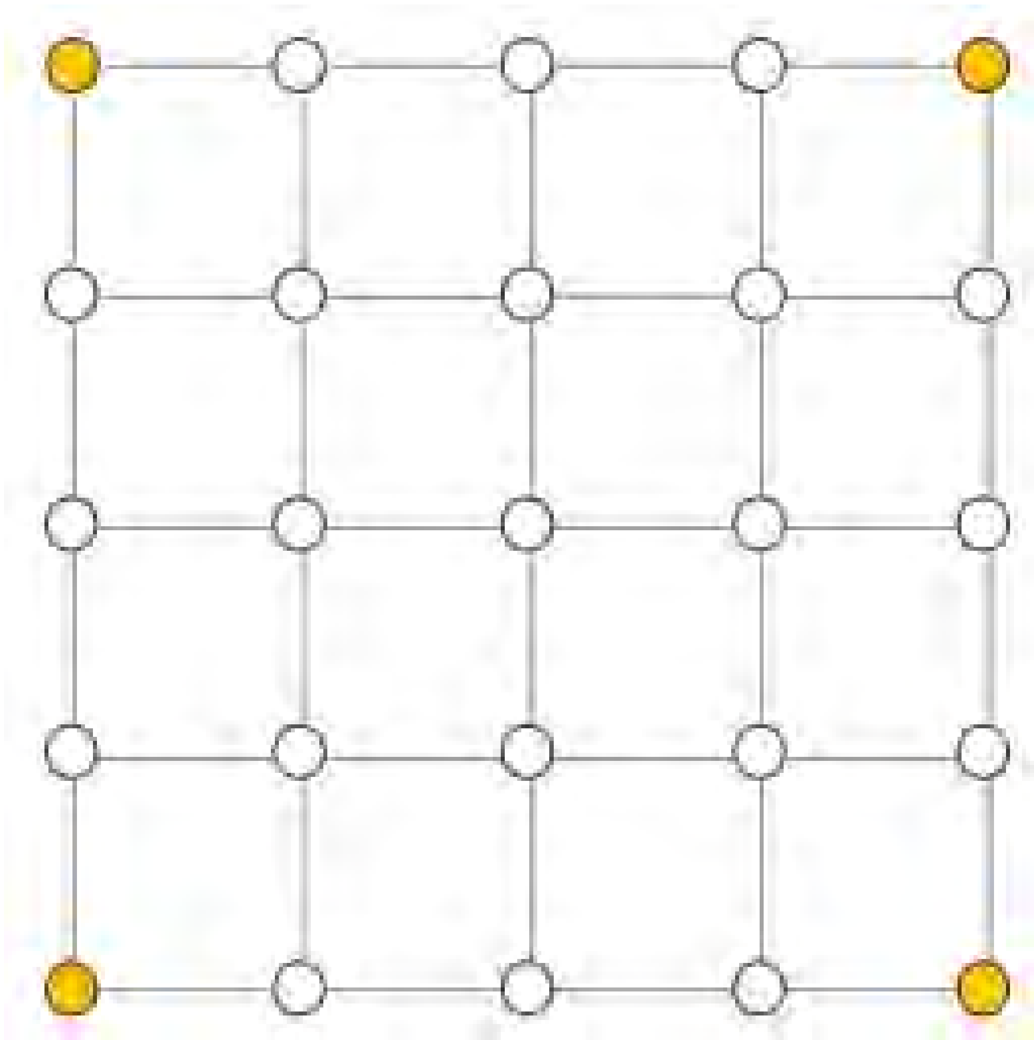


Рисунок 2.19 - Перший крок, обирання кутових точок

Наступним кроком після обирання кутових точок являється застосування одного з двох алгоритмів, а саме алгоритмау діамантового кроку. Суть застосування такого алгоритму полягає у знаходженні серединної точки, яка знаходиться між обраними чотирма. Такі розрахунки полягає на розрахунках середнього від кутових точок. Після цього серединній точці присвоюється випадково згенероване значення, яке визначає її висоту. Поглянувши на рисунок 2.20 можна побачити чому саме цей алгоритм називається діамантовим кроком, якщо ми намалюємо стрілочки, від кутових точок до їх серединної точки, то отримаємо картинку яка віддалено нагадує діамант [18].

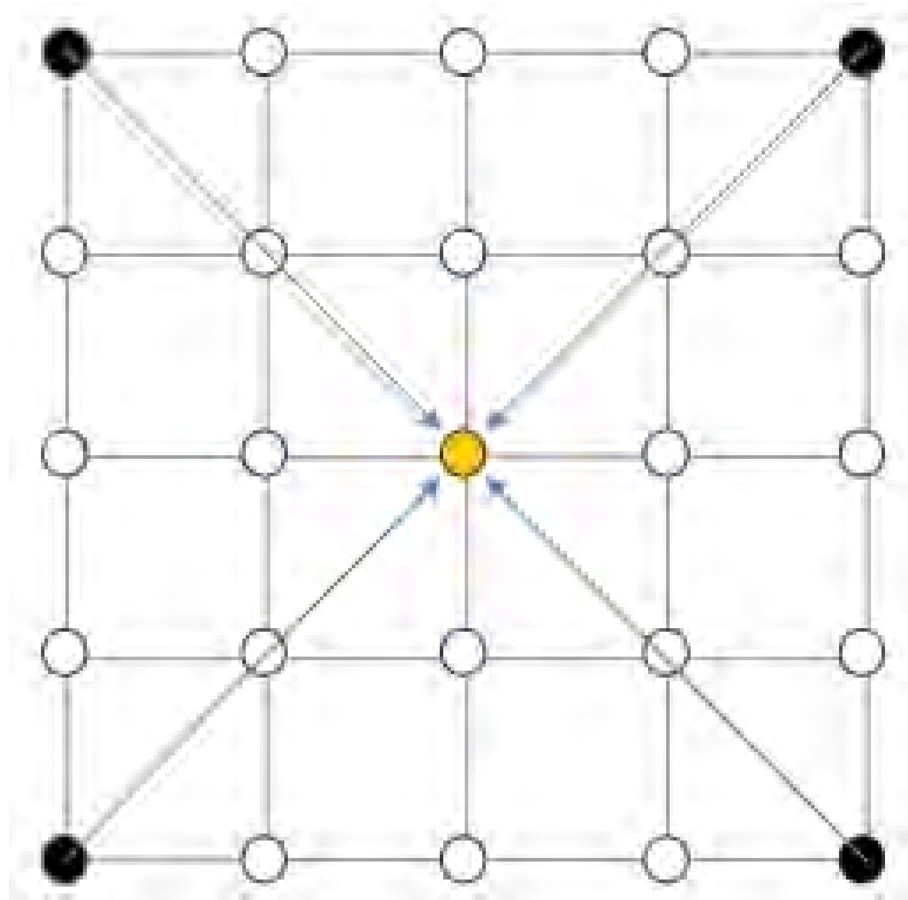


Рисунок 2.20 - Зображення виконання діамантового кроку, де чорні точки це кутові точки для яких розраховується серединна жовта точка

Після розрахунку серединної точки ми умовно отримали на нашій мапі висот один великий квадрат, який охоплює усю мапу, та складається с чотирьох кутових точок та однієї серединної. Тепер ми повинні застосувати метод квадратного кроку. Суть цього метода полягає у подальшому розбитті нашої мапи висот на менші квадрати для більш детальної генерації ландшафту. Для цього ми беремо нашу отриману серединну точку, та знаходимо нові серединні точки для кожної пари кутових точок. У цьому випадку ми уявляємо що кожна пара кутових точок створює діамант з серединною точку, а четверта точка такого діаманта є уявною та знаходиться за межами нашої мапи висот. Для знаходження нових серединних точок в квадратному кроці ми застосовуємо стандартний алгоритм, який використовували при розрахунках діамантового кроку [18]. Після визначення

нових серединних точок ми додаємо до них випадково згенеровану величину яка відповідає їх висоті. З рисунка 2.21 ми можемо побачити як за допомогою умовних стрілочок проходить розбиття на нові квадрати.

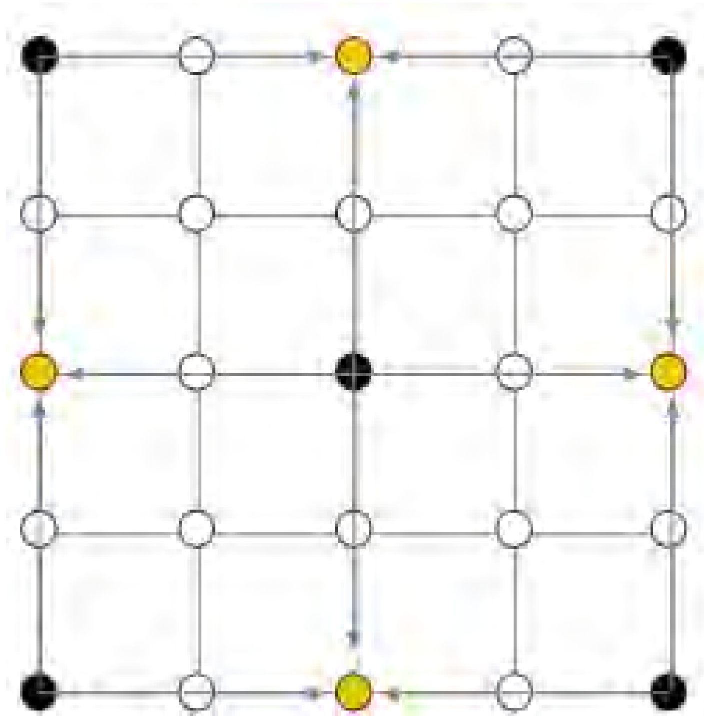


Рисунок 2.21 - Зображення роботи методу квадратного кроку, де чорними точками зображені точки отримані у результаті діамантового кроку, а жовтими – нові серединні точки які розбивають на квадрати.

Результатом виконання методу квадратного кроку ми отримуємо мапу висот яка розбита вже на менші квадрати. Якщо на етапі діамантового кроку ми мали один великий квадрат, то на етапі квадратного кроку ми вже отримуємо чотири квадрати, які додають більшої деталізації [18].

Алгоритм продовжує свою роботи поки не розіб’є всю мапу висот на менші квадрати, але таке розбиття може регулювати з метою додати більше, або меншої точності до генерації з метою оптимізації, або натуральності сгенерованої мапи. Результат пошагової роботи розбивання мапи, яка розглядалася раніше ми можемо спостерігати на рисунках 2.22 та 2.23.

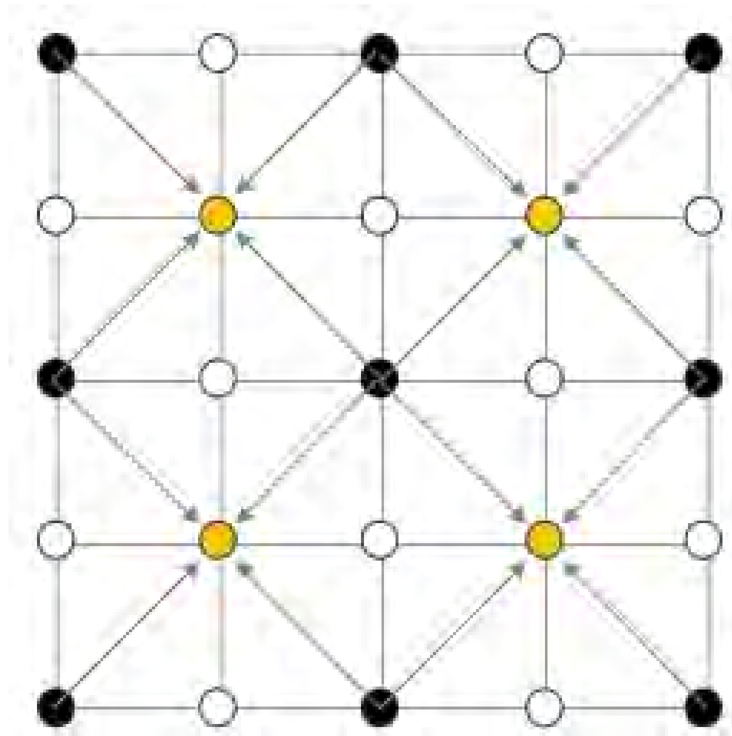


Рисунок 2.22 - Зображення розрахунку нових серединних точок для чотирьох отриманих квадратів під час квадратного кроку

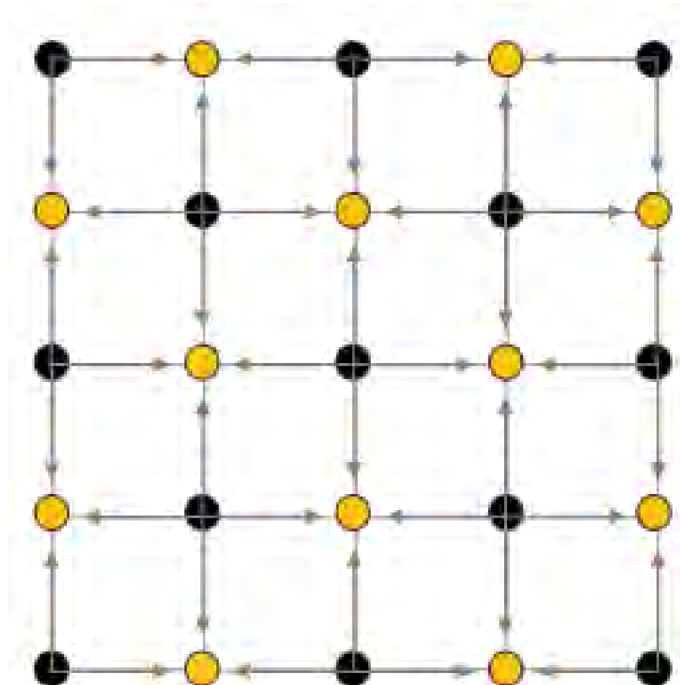


Рисунок 2.23 - Розбиття залишків мапи висот на квадрати методом квадратного кроку

Таким чином ми бачимо що кількість квадратів які будуть участувати під час процедурної генерації збільшилася в чотири рази, тобто ми отримали шістнадцять квадратів. З цього можна зробити висновок, що ми можемо вивести кількість ітерацій алгоритму за формулою (2.7):

$$n = \frac{squares\ count}{4}, \tag{2.7}$$

де n – кількість ітерацій алгоритму;

squarescount – кількість квадратів.

Далі я би хотів представити наскільки сильно змінюється ландшафт від кількості ітерацій та детальніше розглянути основні параметри які впливають на подальше генерування висот. На рисунках 2.24-2.26 представлені емуляції сгенерованного ландшафту на різних ітераціях алгоритму [17].

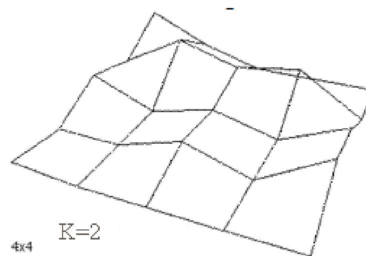


Рисунок 2.24 - Генерація висот результатом якого є ландшафт розмірністю 4x4 квадрати, та ітерацією К

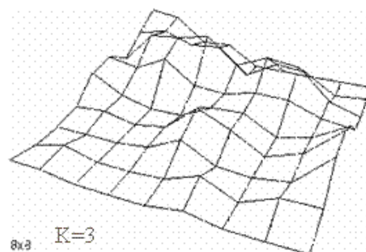


Рисунок 2.25 - Генерація висот результатом якого є ландшафт розмірністю 8x8 квадрати, та ітерацією К

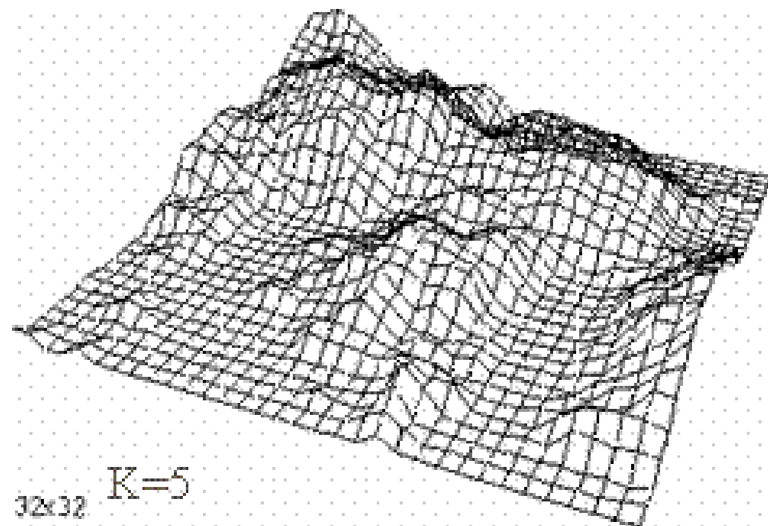


Рисунок 2.26 - Генерація висот результатом якого є ландшафт розмірністю 32x32 квадрати, та ітерацією K

З цього ми можемо побачити, як кількість ітерацій впливає на деталізацію та унікальність ландшафту, чим вище ітерація, тим більш детальний ландшафт ми отримаємо. Або іншими словами ми отримаємо більш високі піки с крутими схилами, а долини становляться глибшими, через нестачу визначених вершин на мапі висот [17]. Ми можемо порахувати зміщення на кожній ітерації алгоритму за формулою (2.8):

$$Displacement\ amount = average + \left(\frac{1}{2}\right)^{H} * random, \tag{2.8}$$

де *Displacement amount* – параметр зміщення;

average – середнє значення прилягаючих кутів;

random – рандомне число яке визначає висоту;

H – значення нерівності ландшафту.

Однією з основних величин які впливають на генерацію висот є параметр нерівності. Параметр нерівності це фактор за допомогою якого визначається з якою силою буде зменшуватися збурення на кожній ітерації алгоритму. Найчастіше стандартним значенням для нерівності

використовують двійку, але це значення можна змінювати як забажає розробник. Отже як саме воно впливає на зменшення збурення. Чим вище ми задаємо значення нерівності, тим більш рівним та гладким буде наш згенерований ландшафт. Отже чим менше буде значення величини нерівності тим більш нерівним то гористим буде наш згенерований ландшафт. За допомогою нерівності розраховується той самий масштаб який корегує висоту обраної точки за формулою (2.9):

$$scale = \frac{1}{2}^{roughness}, \tag{2.9}$$

де *scale* – значення коефіцієнту висоти;

roughness – нерівність.

Виходячи з цього ми маємо що чим більше масштаб тим менше ми маємо заданий параметр нерівності, отже наш ландшафт більш нерівний. Та це працює і у діаметрально протилежному випадку.

Алгоритм зміщення середньої точки також має похідний алгоритм, який дещо інакше генерує висоти отриманих піків. Цей алгоритм розраховує положення піків на мапі висот таким самим чином як і стандартний алгоритм, але для розрахунку висоти піка використовується не проста рандомне значення, але ще й кореговане за допомогою фрактального методу.

Фрактали – це певна множина яка має свійство само подібності, іншими словами об’єкт який приближено або в точності співпадає з частиною самого себе. Одними з найрозвповсюдженних прикладів фрактальних об’єктів є множина Мандельборта, рисунок 2.27, та фрактальна капуста романеско, рисунок 2.28 [19].

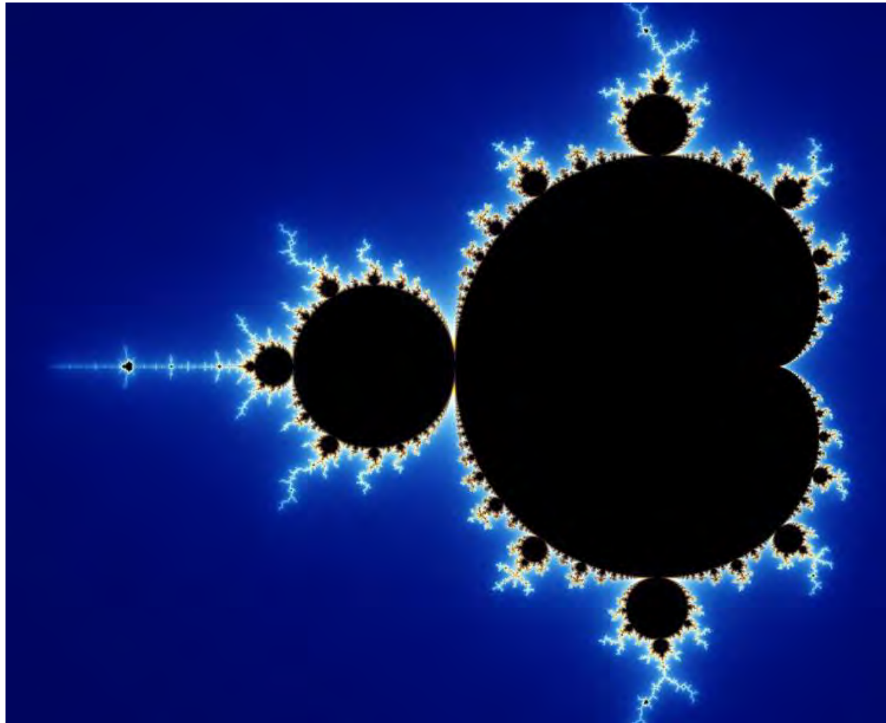


Рисунок 2.27 - Множина Мандельборта

Рисунок 2.28 - Фрактальна капуста романеско

Таке явище не обійшло й математику, в математиці фракталом вважається множина точок в евклідовому просторі, які мають метричну

розмірність. Одним з найвідоміших математичних прикладів серед фракталів є крива Коха. На рисунку 2.29 детально зображена ітерована крива Коха [20].

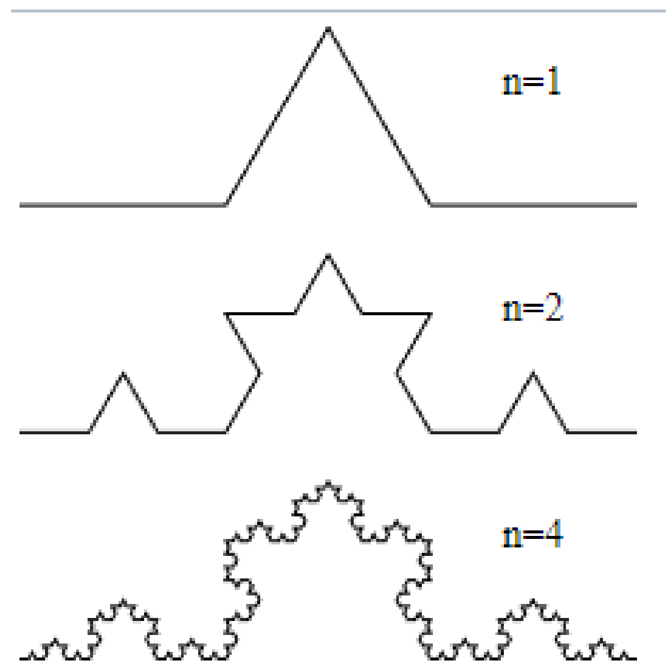


Рисунок 2.29 - Ітерована демонстрація кривої Коха

Ландшафт згенерований таким чином хоча і має схожі частини, але на загальному фоні виглядає натурально та цікаво. Особливість цієї варіації алгоритму зміщення середньої точки полягає у тому, що замість того, щоб масштаб зменшувався на кожному етапі генерування зміщення, він залишається незмінним з ростучою кількістю ітерацій. Наступною відмінність є знак зміщення, який визначається рандомно. В свою чергу висота піків вираховується за допомогою середнього значення сусідніх піків помноженого на рандомне значення та рандомний знак зміщення.

Також для кожного парного ряду значень на мапі висот ми маємо просто скопіювати цей існуючий рядок та додати до неї нові точки. Стосовно не парних рядків мапи висот ми маємо додати цілковито новий вирахуваний рядок між двома старими рядками, рисунок 2.30. Саме завдяки цим двом принципам досягається фрактальність ландшафту, та в той самий час

зберігається достатня індивідуальність багатьох частих цього ландшафту [18].

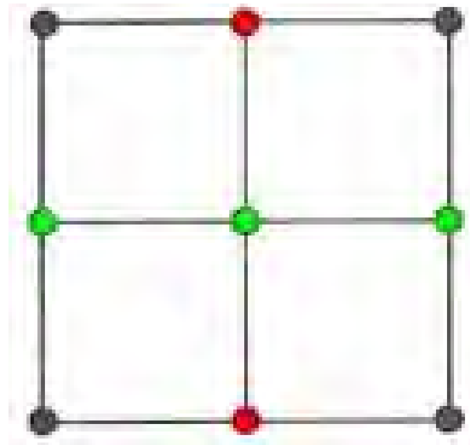


Рисунок 2.30 - Червоним помічено парні рядки, а зеленим непарний

Роблячи висновок з усього дослідження цього алгоритму та його різновиду ми можемо винести декілька переваг та недоліків. До переваг можна віднести, що алгоритм майже ідеально впорується з поставленою задачею, та генерує насправді виразні та натуральні ландшафти, а процес налаштування є дуже швидким и простим. Також за допомогою його фрактального різновиду спектр отримуваних ландшафтів значно збільшується та набуває нових більш привичних виглядів. Що стосується недоліків алгоритму, він генерує складки на краях ландшафту, але це можливо легко обійти під час написання алгоритму.

3 ПРОЄКТ ПРОГРАМНОЇ СИСТЕМИ ГЕНЕРУВАННЯ 3D ЛАНДШАФТІВ

3.1 Архітектура системи

Програмну реалізації систему процедурної генерації ми можемо поділити на два основних модуля. Перший з них - програмна реалізація алгоритмів генерування самого 3D ландшафту яка включає в себе наступний список алгоритмів:

– генерація класичного шуму Перліна;
– генерація діаграми Вороного;
– генерування зміщення середньої точки, а саме фрактального варіанту цього алгоритму;
– алгоритм згладжування ландшафту.

Другий модуль це - програмна реалізація алгоритмів покращення зовнішнього виду ландшафту:

– алгоритм процедурного текстурування усього ландшафту;
– алгоритм розсаджування дерев та кущів;
– алгоритм створення каміння та трави;
– алгоритм створення базової води та імітація берегу.

3.1.1 Архітектура алгоритмів генерування ландшафту

Моя архітектура програмної реалізації процедурного генерування 3D ландшафту базується на нескладній ієрархії класів, яка представлена на рисунку 3.1.

Аби не перевантажувати всіма функціями один клас та робити з нього суперклас була обрана така модель ієрархії. В цій моделі ми можемо бачити, що кожен метод генерал має свій окремий клас, за допомогою якого керується генерування саме цим методом.

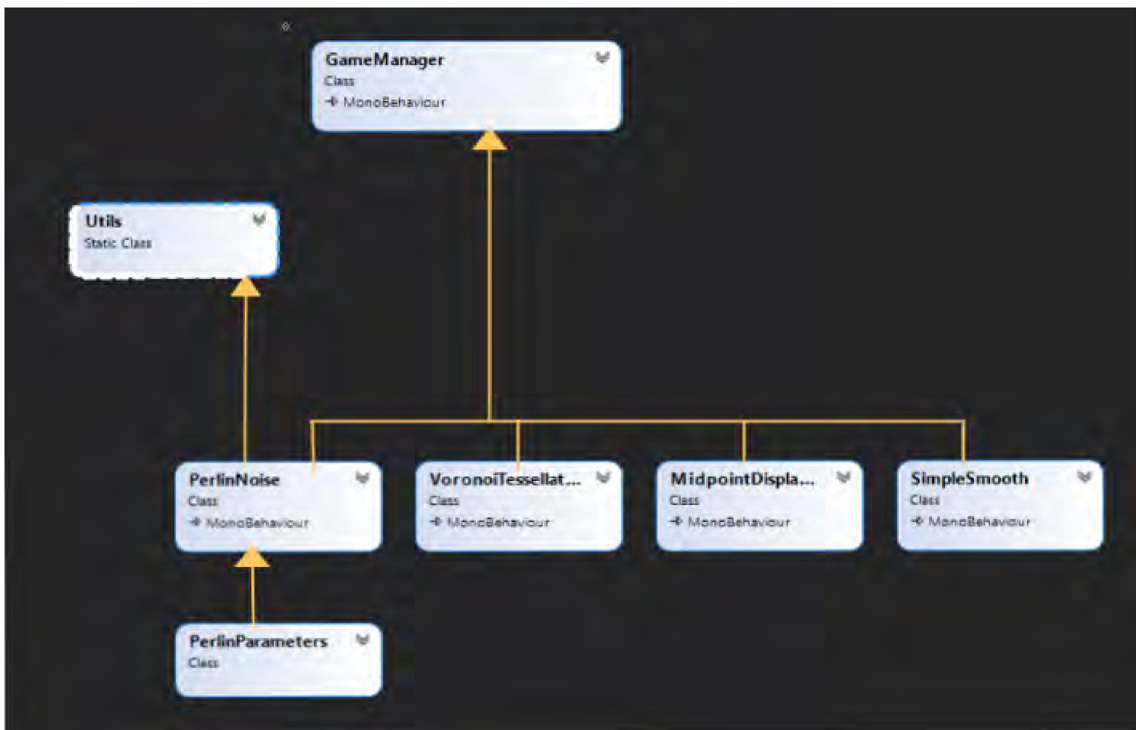


Рисунок 3.1 - Ієрархія класів алгоритмів генерування ландшафту

Така моделі дозволяє робити швидкий доступ до будь якого класу, та поліпшує навігацію під час налаштування параметрів генерації.

Оскільки класичний шум Перліна, який представлений класом PerlinNoise реалізован у якості багаторівневого фрактального шуму, він має допоміжний клас який називається PerlinParameters. Такий підхід допомагає без будь яких проблем створювати будь яку кількість октав з різними налаштовуємими параметрами. Також PerlinNoise використовує допоміжний клас який називається Utils, цей клас реалізує функцію фрактального Броунівського руху, яка може бути доступна з будь якого місця у програмі.

Як ми можемо бачити усі класи процедурної генерації використовують глобальний клас GameManager. Цей клас містить у собі інформацію про наш терейн, на якому ми виконуємо генерацію, а також ряд допоміжних функцій, які використовуються майже усіма алгоритмами генерації. Такий підхід спрощує доступ до загальних даних, та викреслює необхідність передавати або реалізовувати одні й ті ж самі данні та функції в класах з безпосередньо алгоритмами генерації.

3.1.2 Архітектура алгоритмів покращення зовнішнього вигляду ландшафту

Архітектура алгоритмів покращення зовнішнього вигляду схожа на архітектура базових методів генерацію, схема архітектури представлена на рисунку 3.2.

Рисунок 3.2 - Ієрархія класів алгоритмів покращення зовнішнього вигляду ландшафту

Підхід до побудови архітектуру цих класів був обраний такий ж самий як і з основними алгоритмами генерування ландшафту. Така архітектура

дуже добре показала себе під час розробки основних алгоритмів, саме тому було прийняте рішення використовувати її і надалі.

Також на цій ієрархії ми можемо бачити усі поля допоміжних класів, які використовуються для множинного налаштування елементів алгоритму. Для обробки таких полей, та передачі всіх необхідних значень від юзера до алгоритмів процедурної генерації була застосована архітектура зображена на рисунку 3.3.

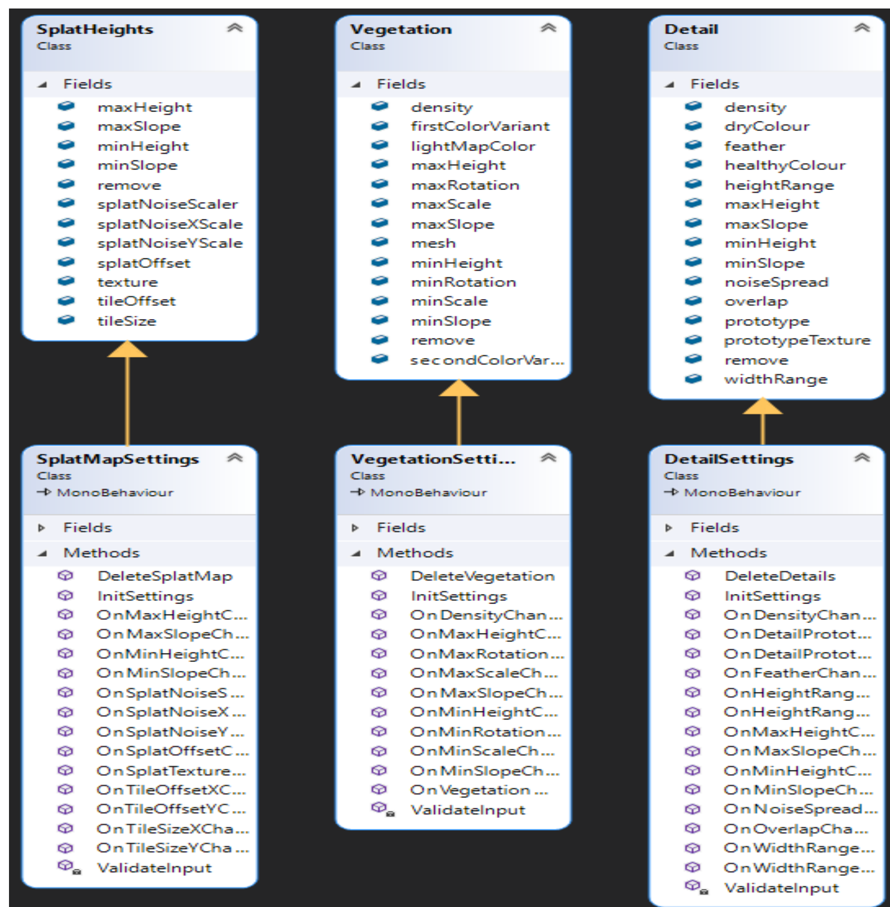


Рисунок 3.3 - Ієрархія допоміжниї UI класів

У цій архітектурі використовується підхід схожий на івентову систему. У кожного с класів який може бути представлений нескількими варіаціями одночасно, існує допоміжний UI клас. Такий клас в свою чергу є множинним представленям та зв’язаний з кожним унікальним елементом налаштуваня, який юзер додає динамічно. Після того як юзер виконує якийсь

ввід даних у поля вводу, Unity реагує на це та викликає відповідний метод з допоміжного класу, який обробляє дані за допомогою простої версії валідації, та після цього оновлює їх в реальному програмному представлені до якого відноситься цей параметр.

Така система допомагає легко маніпулювати даними за допомогою UI елементів та легко розширюєма, саме через це вона і була обрана для представлення множинного налаштування алгоритмів.

3.2 Засоби реалізації

Для реалізації процедурного генерування ландшафту були засновані наступні засоби реалізації:

– засоби .Net а саме мова програмування C#;
– середовище розробки Microsoft Visual Studio 2019 Enterprise;
– ігровий двигун Unity3D 2019.3.15.

3.2.1 Середовище розробки Microsoft Visual Studio 2019 Enterprise

Під час розробки програмного продукта була використана Microsoft Visual Studio 2019 Enterprise для написання коду скриптів. Це середовище підходить для роботи з ігровим двигуном Unity3D, адже воно має пакет інструментів для роботи з вбудованими типами, класами та функціями Unity. Також Visual Studio підтримує Debug режим за допомогою якого можливо продебажити код за брейкпоінтами та виявити несправності.

3.2.2 Мова програмування C#

Раніше ігровий двигун Unity підтримував декілька мов програмування, а саме JavaScript та C#. Але з часом в одній з нових версій

такий підхід був змінений, то мову JavaScript цілком видалили з Unity. Отже через це для розробки була застосована мова C#.

C# є об'єктно-орієнтованою мовою програмування яка була розроблена для платформи .NET.

Основною перевагою данної мови є строга типізація данних, а також об'єктно-орієнтований підхід добре підходить для структури написання та роботи в Unity.

3.2.3 Ігровий двигун Unity3D версії 2019.3.15

Unity – це кросплатформенне середовище для розробки комп'ютерних ігор, а також різного роду аплікацій, реліз якого відбувся в 2005 році, та кожен рік виходить нова версія з покращенням вже існуючого функціонали, та доданням нового. Unity дозволяє створювати ігри та аплікації для більш ніж 25-ти різних платформ, серед них числяться різноманітні консолі, телефони на операційних системах Android та IOS, фреймворки для сайтів, а також звісно ж комп'ютери.

Для створення процедурної генерації був обранн саме цей двигун адже він є одним з найдоступнішів та найзручніших у цілях генеруванні ландшафту завдяки вбудованому інструменту який називається Terrain. Цей інструмент містить у собі всі необхідні параметри для генерації починаючи з мапи висот та закінчуючи генерації рослин та натуральних об'єктів. А система скриптів допомагає швидко розширювати вже написану систему, та доповнювати її новим функціоналом.

Була обрана версії 2019 року, адже вона є однією з найстабільніших, та містить у собі усі необхідні поліпшення інструментів для роботи з тереном.

3.3 Модулі та алгоритми

Під час розробки було створено багато скриптів для певних алгоритмів, та фактично програма складається з одного модуля, але умовно ми можемо поділити її на 3 модулі:

– модуль процедурного генерування 3D ландшафту;
– модуль генерування рослинності та текстурування ландшафту, або іншими словами, покращення візуальної складової;
– ui модуль для взаємодії юзера з параметрами генерування.

3.3.1 Обмеження програмної реалізації

Программа має не дуже багато обмежень. По перше користувач повинен чітко підібрати граничні данні при редагуванні параметрів, адже результати з занадто великими або малими значеннями параметрів налаштування можуть видати зовсім неочікувані результати. Зрозуміти які саме значення є граничними можливо лише методом спроб та невдач. По друге що стосується текстурування, текстури мають подаватися у вигляді квадратів, тобто кількість пікселів по ширині та висоті мають співпадати, також вони мають бути безшовними, для збереження цілісності ландшафту при текстурованні. Якщо цього не зробити на етапі текстурування ми можемо побачити результат який дуже далекий від бажаного, та натурального варіанту.

3.3.2 Побудова 3D ландшафту

3.3.2.1 Алгоритм класичного шуму Перліна

Базовим алгоритмом який лежить в основі процедурної генерації був обраний алгоритм класичного шуму Перліна. Алгоритм працює з мапою висот, яка реалізована в інструменті ландшафту terrain в ігрового двигуну

Unity. Спочатку розглянему вхідні данні яки використовуються для роботи алгоритму, та можуть бути налаштованими користувачем за допомогою UI системи в рантаймі (рис. 3.4).

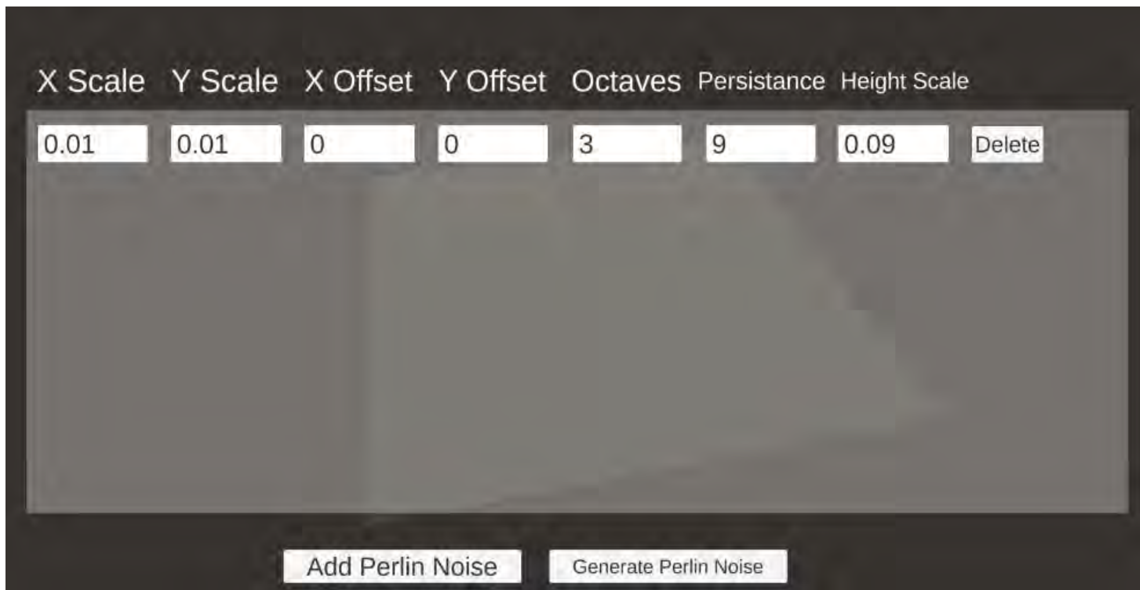


Рисунок 3.4 - UI для редагування параметрів класичного шуму Перліна

Отже для редагування параметрів ми маємо такий вид UI в якому за допомогою полів вводу можливо змінювати данні, за допомогою кнопки Delete видаляти додаткові шуми Перліна з алгоритму, або додавати нову за допомогою кнопки Add Pelrin Noise. Кнопка Generate Perlin Noise запускає алгоритм процедурної генерації.

Стосовно даних для редагування ми маємо Scale, Offset, Octaves, Persistance та height Scale. Параметр Scale відповідає за ширину хвиль за вісями X та Y, або іншими словами ми приближуємо конкретну частину нашої згенерованої кривої Перліна, чим більше значення тим далі ми віддаляємося та бачимо все більше точок кривої. Параметр Offset відповідає за зміщення по згенеровані алгоритмом кривій класичного шуму Перліна. Тобто, якщо Scale працює як зум для певної частини, то Offset в свою чергу зміщує місце до якого приміняється цей зум, та точки беруться з цього відрізку кривої. Octaves це кількість октав в нашому шумі Перліна, тобто як

говорилося раніше, це кількість згенерованих класичних шумів Перлін для одного представлення алгоритму. Параметр Persistance визначає наскільки сильно амплітуда настопнуї функції класичного шуму Перліна буде вищою за попреденю. Height Scale в свою чергу працює як деякий коефіцієнт, який допомагає мати більше контролю над висотою згенерованного шуму. Чим менший параметр скейлу висоти ми задаємо, тим більше сглажений терен ми отримаємо, тим самим зменшуючи реальну висоту згенерованного класичного шуму Перліна. Приклади різних параметрів скейлу та скейлу висоти наведено в рисунках 3.5 та 3.6.

Рисунок 3.5 - Генерація класичного шуму Перліна з низькими параметрами Scale та Height Scale

Це всі данні які може контролювати користувач. Інші змінні такі як частота та амплітуда вираховуються виключно під час роботи алгоритму.

Сама робота алгоритму спрощена за допомогою методів Unity. В Unity існує стандартна вбудована бібліотека Mathf яка э аналогом імплементації такої бібліотеки у чистій мові C#.

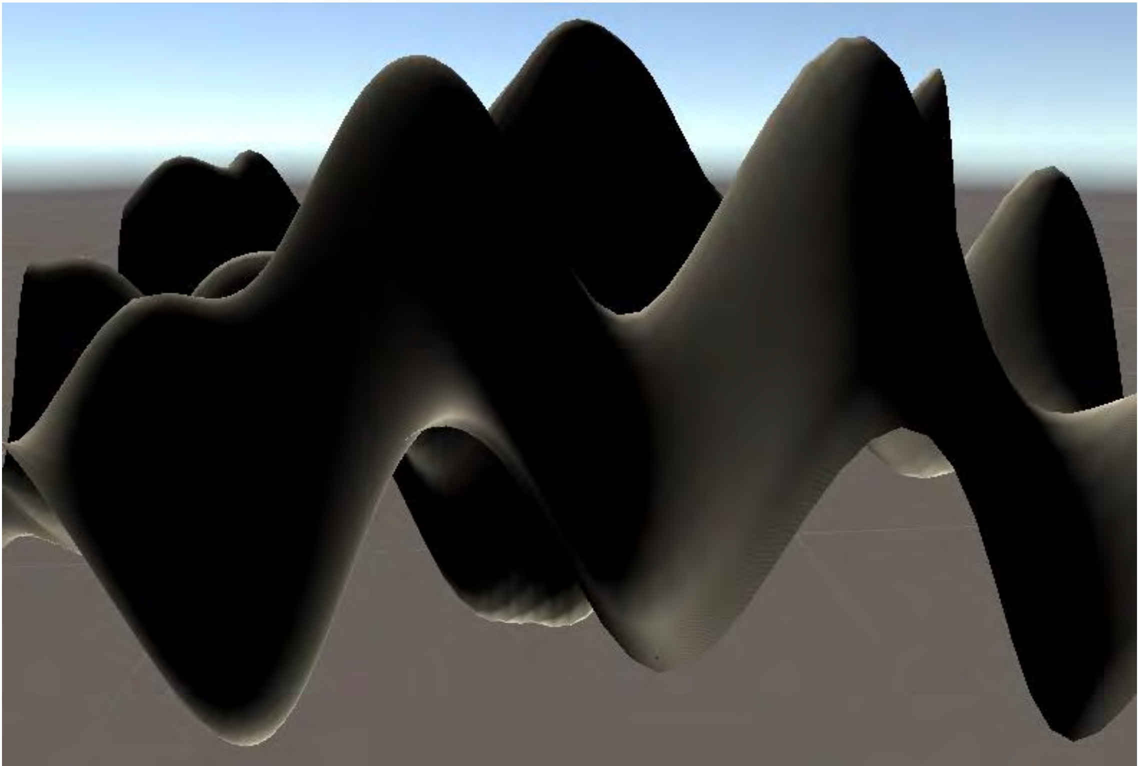


Рисунок 3.6 - Генерація класичного шуму Перліна за великими параметрами Scale та Height Scale

Ця бібліотека має свою власну імплементацію алгоритму класичного шуму Перліна. Це допомогає виключати зайві витрати на реалізацію власної функції, але замість цього ми реалізуємо підхід за допомогою фрактального Броуновського руху. За допомогою такого підходу ми можемо мати повторювані елементі на нашому терейні, та мати трохи більше контролю над генерацією. Якщо ми підвищимо наш Scale, тим самим як би віддаливши наш погляд на функцію шуму, то побачимо, як елементи нашої функції шуму повторюються через деякий проміжок. Реалізацію алгоритму фрактального Броуновського руху ми можемо побачити у лістингу 1.

```
public static float fractalBM(float x, float y, int octaves, float pers)
{
   float final = 0;
   float frq = 1;
   float amp = 1;
   float maxV = 0;
```

```
  for (int i = 0; i < octaves; i++)
  {
    final += Mathf.PerlinNoise(x * frq, y * frq) * amp;
    maxV += amp;
    amp *= pers;
    frq *= 2;
  }

  return final / maxV;
}
```

Лістинг 1. Функція фрактального Броуновського руху для розрахунку класичного шуму Перліна

Сама ж функція процедурного генерування на ландшафті бере карту висот та за допомогою подвійного циклу на кожній висоті застосовує розрахунки класичного шуму Перліна стільки разів, скільки зазначено різних шумів користувачем через UI.

Шумів не може бути менше ніж один, а до полів вводу застосовується базова валідація, для перевірки на пустий елемент вводу. Результатом відпрацювання алгоритму є відозмінений ландшафт зі зміненою картою висот, який являє собою базу для подальшого генерування ландшафту за допомогою інших двох алгоритмів.

3.3.2.2 Алгоритм діаграми Вороного

Наступним алгоритмом, за допомогою якого відбувається процедурне генерування, виступає алгоритм діаграми Вороного. Він був обран для визначення додаткових піків на нашому террейні, або задля глобального змінення висоти терейну, залежно від обраної функції генерування та налаштованих параметрів. Як і алгоритм класичного шуму Перліна, діагарма Вороного застосовується тільки для обробки мапи висот, хоча цей алгоритм має потенціал для застосування у визначенні наприклад границь біомів, у подальшому покращені роботи.

Спочатку давайте розглянемо данні які користувач програми може змінювати за допомогою UI. Оскільки діаграма Вороного охоплює усю мапу висот та не має алгоритмів з використанням декількох діаграм, як у класичного шуму Перліна, то UI представлений у вигляді одиничної таблиці налаштування без можливості додавання нових імплементацій алгоритму (рис. 3.7).

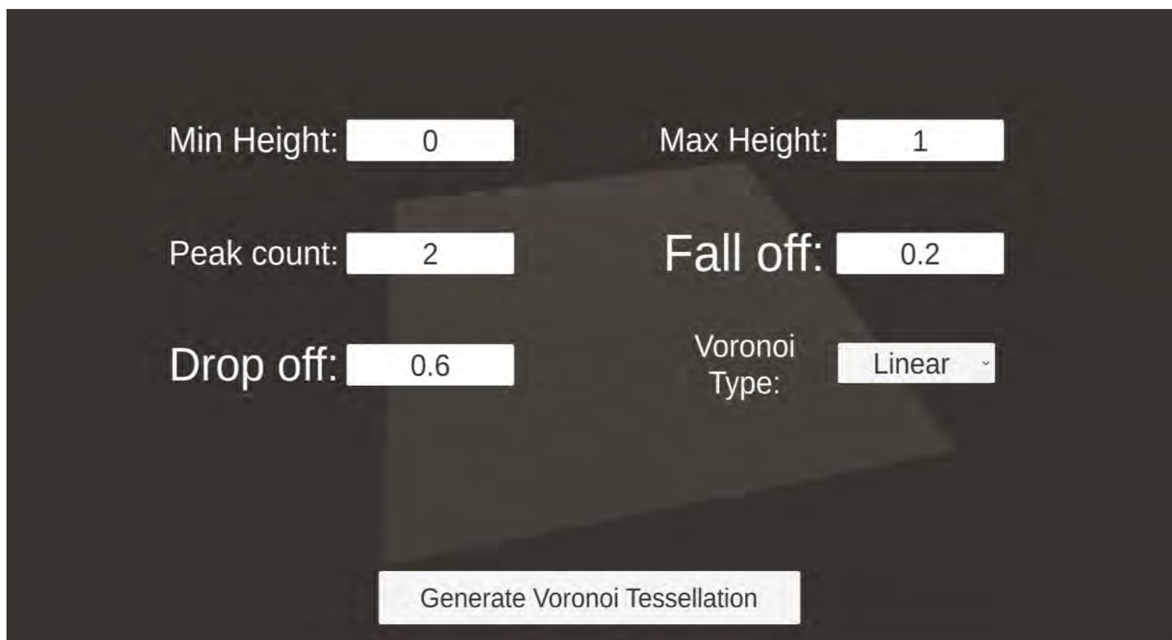


Рисунок 3.7 - UI для налаштування роботи алгоритму діаграми Вороного

Отже як ми можемо бачити для налаштування діаграми Вороного ми маємо менше параметрів, ніж для класичного шуму Перліна. Для налаштування ми маємо такі параметри Min Height, Max Height, Peak Count, Fall off, Drop off та Voronoi Type.

Аби зберегти елемент рандомності ми використовуємо параметри Min Height та Max Height. Ці два параметрі являються мінімумом та максимум при рандомному визначені висоти піку. Оскільки мінімальне значення висоти для терейну це нуль, а максимальне значення одиниця, ці два параметри повинні не виходити за ці границі які виставляє сам двигун Unity.

Наступним йде параметр Peak Count, це параметр типу int який визначає кількість піків, які будуть згенеровані. Місце знаходження піку на мапі висот визначається рандомно, береться вектор з координатами X та Y залежно від розширення террейну, яке задається через інспектор. Таким чином піки можуть бути дуже близько один до одного, та їх схили будуть пересікатися, та завдяки цьому ми побачему дію діаграми Вороного, де схили піків рівномірно визначають границі на місцях перетину. Також може бути случай, що пік був згенерований з висотою меншою ніж вже згенерований пік, або схил іншого піку, у цих самих координатах, у цьому випадку такий пік ігнорується та не створюється повторно.

Ще в нас присутній параметр який називається DropOff. Він відповідає за те як саме будуть виглядати наші схили, та загальна структура гори. За допомогою цього параметру ми можемо керувати формою нашої гори, чи буде вона випукла, на нагадувати пагорб, чи вона буде більше увігнута всередину, та представляти собою рівномірно звужаючийся гірський пік. Це допомагає створювати ландшафти різної форми швидким способом.

Також ми маємо такий параметр як Fall off. Він виступає свого роду коефіцієнтом висоти, яка буде між нашим піком та наступною точкою. Таке налаштування потрібне аби ми могли контролювати наскільки круті схили повинні бути у нашого схилу. Тобто, якщо ми хочемо, аби наш схил бук менш крутим та як результат сама гора займала більше місця, ми маємо виставити не ціле число менше за одиницю. Таким самим чином якщо ми хочемо бачити більш круті схили, ми виставляємо число більше за одиницю.

Останнім але не менш важливим є параметр Voronoi Type. Цей параметр цілковито визначає роботу алгоритму, а саме як само будуть генеруватися схили піків. Сюди ми можемо вставити будь який математичний вираз, який ми побажаємо для генерування схилів піків. Для демонстрації я обрав чотири різних вирази, а саме Linear, Power, Combined та SinPow. Користувач може вибрать будь який з цих алгоритмів який більш за

все підходить для його цілей генерування. Розберемо кожну з цих функцій за допомогою лістингу 2.

```
float dist = Vector2.Distance(pTransformP, new  Vector2(x, y)) / maxDist;
float height = 0.0f;

        switch (vVariant)
        {
          case VoronoiVariant.Linear:
            height = peak.y - dist * fallOff;
            break;
          case VoronoiVariant.Combined:
            height = peak.y - dist * fallOff -
             Mathf.Pow(dist, dropOff);
            break;
          case VoronoiVariant.Power:
            height = peak.y - Mathf.Pow(dist, dropOff) *
             fallOff;
            break;
          case VoronoiVariant.SinPow:
            height = peak.y - Mathf.Pow(dist * 3, fallOff) –
             Mathf.Sin(dist * 2 * Mathf.PI) / dropOff;
            break;
         }
```

Лістинг 2. Фрагмент основної частини алгоритму генерування схилів піків діаграми Вороного різними функціями

Спочатку перед основними розрахунками ми повинні розрахувати дистанцію до піку. Всі ці розрахунки протекають у подвійному циклі, який проходиться по всій мапі висот та беру значення X та Y на кожній ітерації. Дистанція до піку береться між точкою на мапі висот, на якій ми знаходимися в даний момент циклу, та між координатами самого піку, потім все це ділиться на максимальну відстань. Максимальна відстань представляє собою відстань між кутами террейну, тобто між нульовими координатами в одному куті, та координатами розширення террейну у протилежному. У ці

розрахунки не входить точка с координатами самого піку, адже його висоту ми вже визначали рандомно на ранньому етапі генерації.

Після цього ми можемо переходити до основних функцій серед яких йде вибір розрахунку діаграми. Першим йде Linear або лінійна функція, її суть полягає у тому що ми беремо висоту нашого піку, та віднімаємо від нього дистанцію помножену на falloff. Результатом такого підходу будуть доволі гладкі схили, це може бути корисно при генеруванні долини з великими, але плавними схилами. Результат такого генерування ми можемо побачити на рисунку 3.8, де виходить дещо схоже на схил пагорбу, який плавно йде донизу.

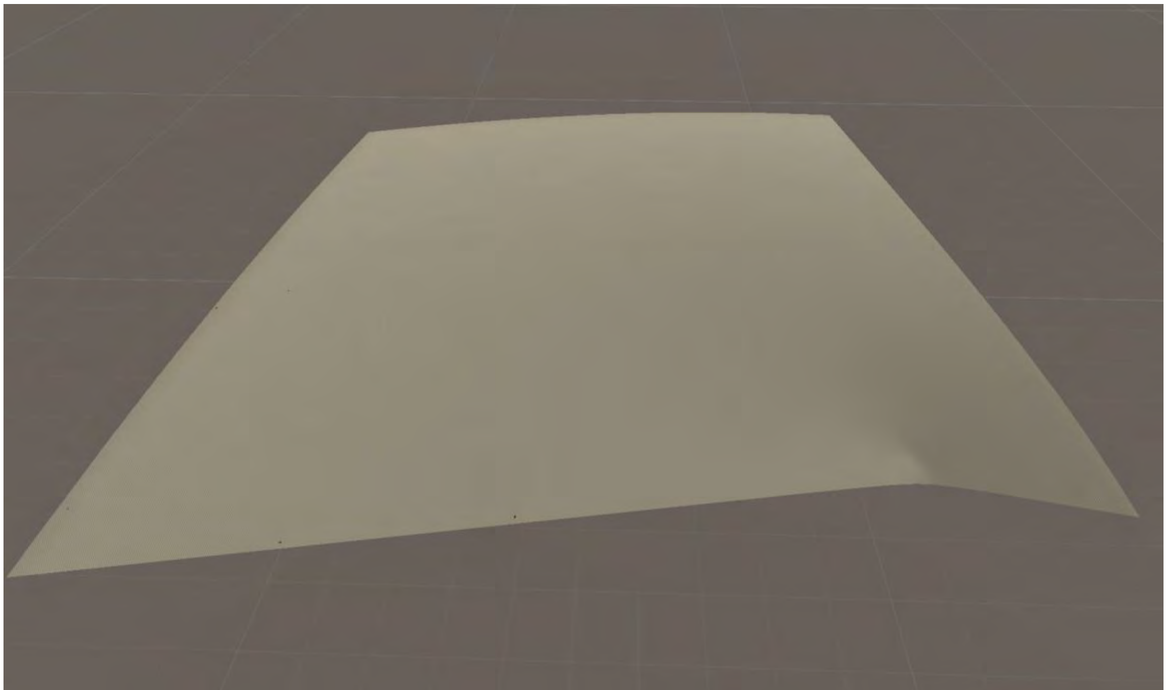


Рисунок 3.8 - Результат генерування діаграми Вороного лінійним методом

Наступним розглянемо метод функцію Power або функцію степені. Такий підхід вже добре працює для створення конусоподібних пагорбів. Для реалізації такої функції ми можемо використати вбудовану функції зведення у ступінь Mathf.Power, для цього ми передаємо до неї дистанції до піку, яка і буде зводитися у ступінь, а також dropOff який саме і виступає в якості ступені. Потім ми помножуємо цей результат на falloff, це буде вже впливати

на діаметр схилів, тобто чим менше значення тим більший простір в діаметрі будуть займати наші пагорби. На рисунку 3.9 ми можемо побачити результат такого виконання, яким є пагорби у формі конуса.

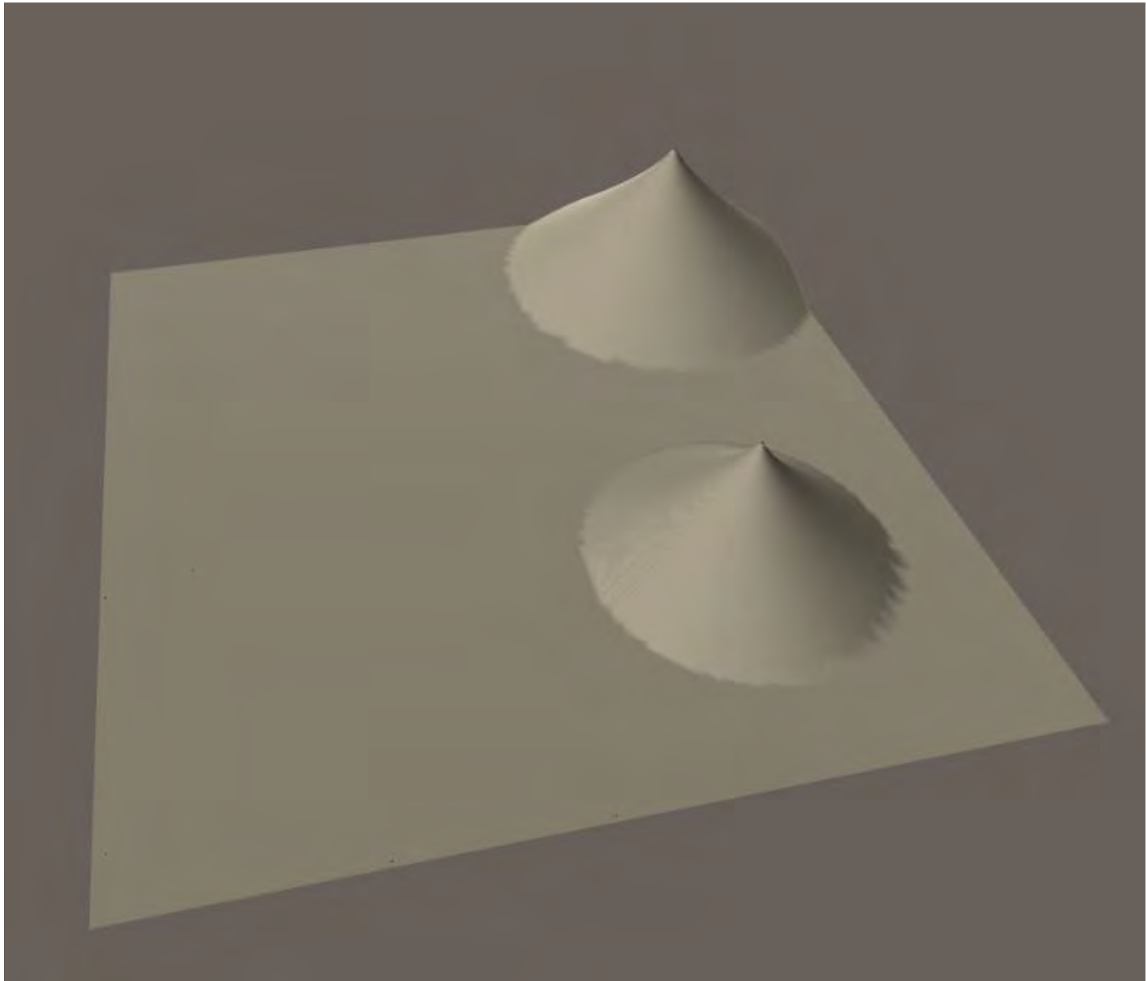


Рисунок 3.9 - Результат генерування діаграми Вороного за допомогою алгоритму зведення у ступінь

Третій метод полягає у комбінуванні двох попередніх, лінійного та зведення у ступінь, та називається Combined – комбінований. Він лише відрізняється тим, що лише частина, яка відповідає за лінійну функцію, помножується на falloff замість частини зведення у ступінь. Результатом такого комбінування вже є строгі конусоподібні піки, схили яких не мають плавного переходу, а йдуть чітко під одним кутом (рис. 3.10). Такий підхід не має практичного застосування сам по собі, але може бути добре

комбінований з іншими реалізованими алгоритмами процедурного генерування ландшафтів.

Рисунок 3.10 - Результат генерування діаграми Вороного за допомогою комбінованого алгоритму

Останній алгоритм, який називається SinPow використовує функції зведення у ступінь та взяття синусу кута. Взяття синусу реалізовано вбудованою функцію та приймає дистанцію до піку помножену на два та на число пі. Результат взяття синусу віднімається від результату зведення у ступінь дистанції до піку помноженої на три, у ступені falloff. Потім результат ділиться на dropOff. Ця функція є тестовою тільки для демонстрації можливості використання будь яких математичних функцій, для отримання результату. Множення на певні числа було підібрано вручну аби мати задовільний результат, та не несуть якоїсь інформативної цінності. Результатом виконання такої функції з великими значеннями dropOff та

falloff параметрами є щось схоже на загостренні піки, та надалі, якщо підібрати значення можна отримати щось на кшталт дна печери с загостреними сталагмітами. Результат можна побачити на рисунку 3.11.

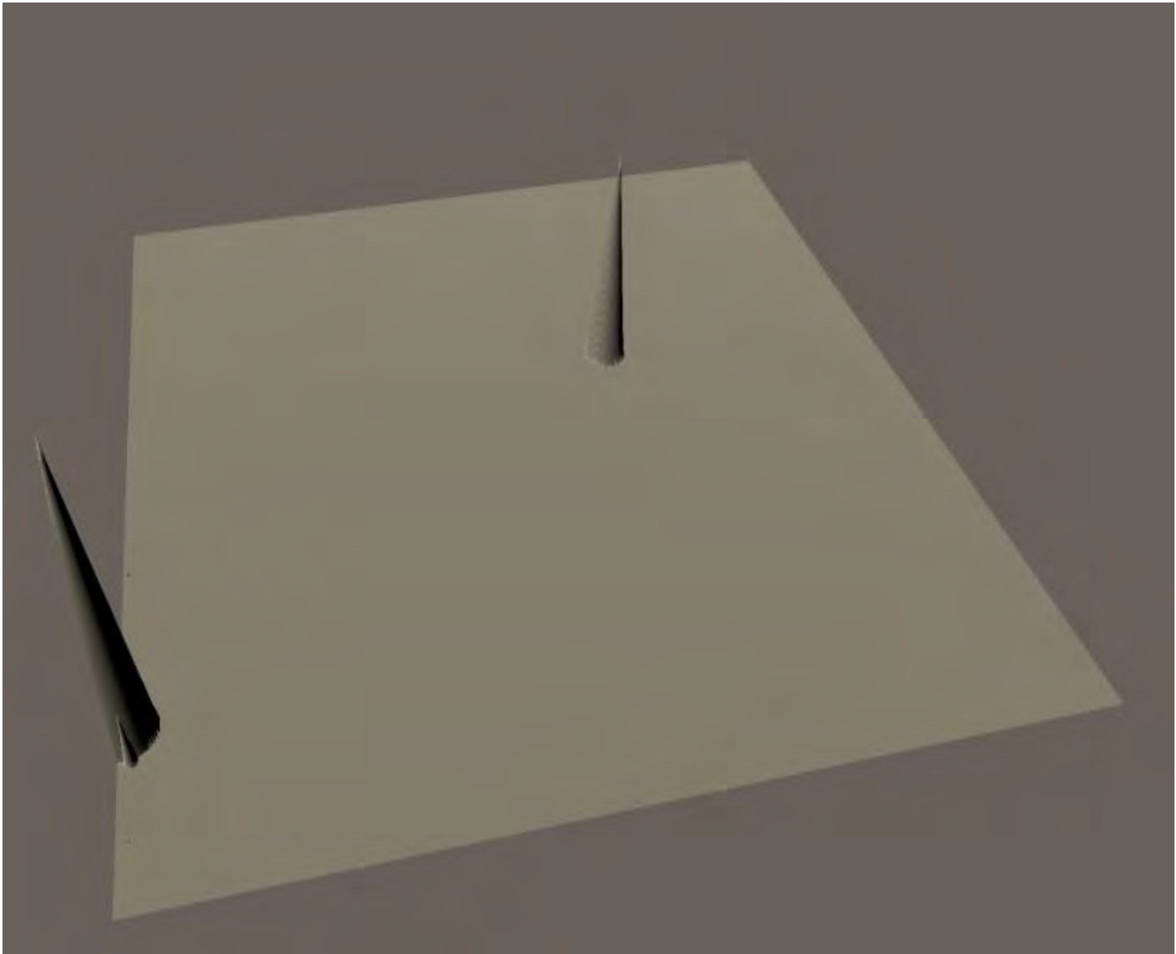


Рисунок 3.11 - Результат генерування діаграми Вороного за допомогою алгоритму взяття синусу

Отже як висновок цей алгоритм підтримує будь які математичні вирази, та якщо трохи попідбирати значення для синусів, косинусів та інших подібних функцій, можна отримати доволі цікаві та красиві результати, які можуть використовуватися надалі у процедурному генеруванні різного роду ландшафтів.

3.3.2.3 Алгоритм зміщення середньої точки

Останнім алгоритмом який використовується в процедурній генерації ландшафту є алгоритм зміщення середньої точки, або Midpoint Displacement. Цей алгоритм є найважливішим, адже саме за допомогою нього відбуваються основні перетворення ландшафту які надають йому схожості з нашим реальним світом. Для початку розглянемо налаштовуємі параметри генерації які представлені на рисунку 3.12.

Рисунок 3.12 - Параметри налаштування алгоритму зміщення середньої точки

Цей алгоритм не має дуже багато налаштувань. Параметри Min Height та Max Height визначають максимальну висоту на яку можуть піднятися гори. В даному випадку ми можемо поставити мінімальну висоту меншу за нуль, адже таким чином ми підвищимо вікно для рандому, аби наші гори розташовувалися не на всій площині террейну, а лише частинами. Максимальну висоту ми можемо виставляти навіть більшу за одиницю, яка

є максимальною висотою террейну, адже потім наша висота буде зменшена за допомогою наступної величини height dumper. Height Dumper допомогає робити наш террейн рівномірно зменшуваним, за допомогою чого ми отримуємо гори, а також виникає ефект фрактальності. Його величина застосовується в перерахованому вигляді, для цього ми беремо завдану величину та зводимо у ступінь мінус одиниці помноженої на Roughness. Параметр Roughness не приймає участі у подальших розрахунках, але він напряму впливає на кількість гострих піків на наших горах, під час початкового підрахунку Height Dumper параметру. Чим більше в нас задана roughness тим більш гладким в нас виходить террейн, та гори лишаються більшості своїх гострих піків.

Сам алгоритм як і його реалізація поділяються на два етапи. Ці етапи або іншими словами алгоритми називаються Діамантовий крок та Квадратний крок. Спочатку застосовується алгоритм діамантового кроку Diamond Step який зображений у лістингу 3.

```
for (int x = 0; x < w; x += sqSize)
        {
           for (int y = 0; y < w; y += sqSize)
           {
              midPointX = (int)(x + sqSize / 2.0f);
              midPointY = (int)(y + sqSize / 2.0f);
              cornerPointX = x + sqSize;
              cornerPointY = y + sqSize;
              var firstOperand = hMap[x, y] + hMap[cornerPointX, y]
                 + hMap[x, cornerPointY] + hMap[cornerPointX, cornerPointY];
          var secondOperand = 4.0f + Random.Range(hMin, hMax);
              hMap[midPointX, midPointY] =  firstOperand / secondOperand;
                }
            }
```

Лістинг 3. Алгоритм Diamond Step для алгоритму зміщення середньої точки

Алгоритм діамантового кроку проходиться по всьому нашому террейну та повинен визначити точки кутів які формують діамант, та його центральну

точку. Центральні координати так само як і координати кутів визначаються за допомогою squareSize. Ця змінна відповідає за визначення розміру нашого поточного квадрату на террейні над яким ми проводимо генерацію. Спочатку величина квадрату максимальна з центральною точкою в центрі нашого террейну, а після кожної глобальної ітерації вона зменшується вдвічі.

Далі визначається висота нашої середньої точки на мапі висот. Це відбувається методом додавання усіх кутів нашого діаманту, ділення цієї суми на чотири, адже ми маємо чотири кути, та додавання рандомної висоти яка береться у рамках заданих користувачем.

Результатом виконання цієї частини алгоритму є фрактальні піки, які формують діаманти якщо поглянути на них згори (рис. 3.13). Ці піки лягають в основу наступної частини алгоритму яка називається квадратний крок.

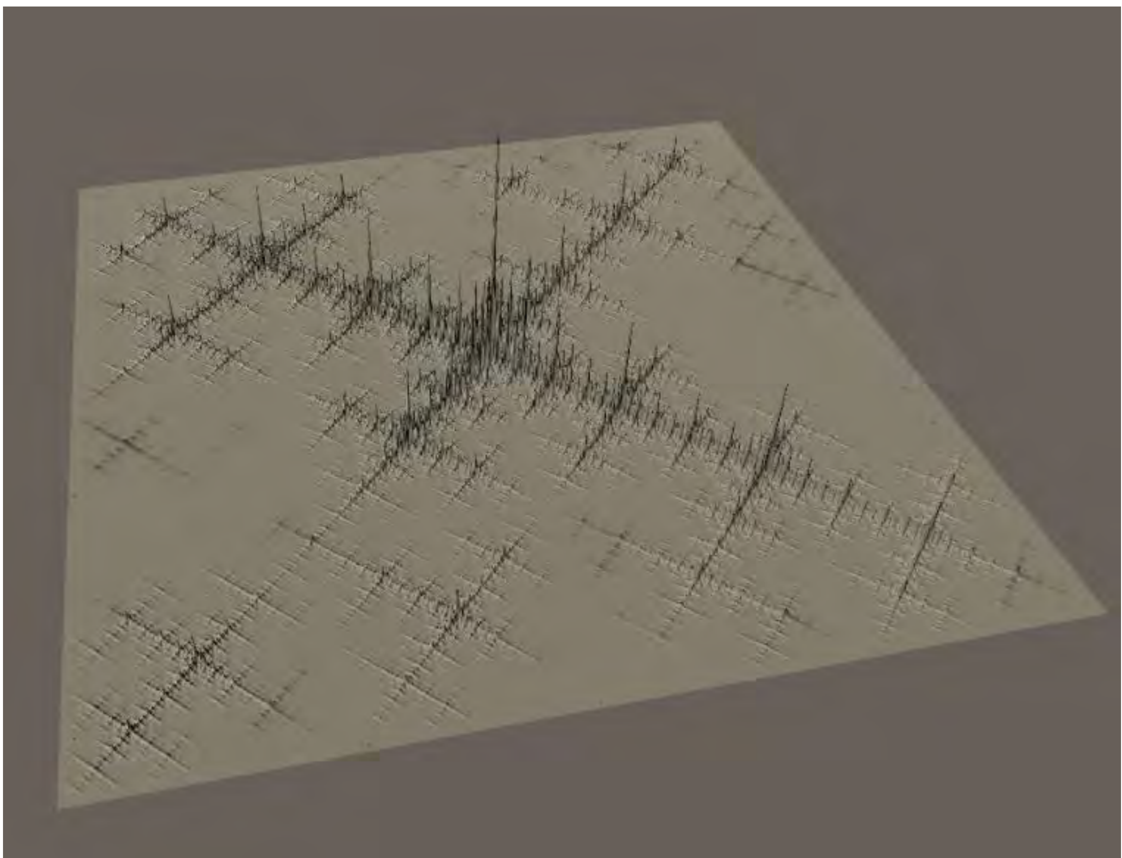


Рисунок 3.13 - Результат виконання алгоритму Diamond Step.

На основі сгенерованних висот піків ми тепер можемо застоувати другу частину алгоритму яка називається квадратний крок або Square Step. Цей

алгоритм в свою чергу генерує вже проміжкові значення які представляють схили наших піків. Розберемо код алгоритму представлений в лістингу 4.

```
for (int x = 0; x < w; x += sqSize)
        {
           for (int y = 0; y < w; y += sqSize)
           {
              cornerPointX = x + sqSize;
              cornerPointY = y + sqSize;

              midPointX = (int)(x + (sqSize / 2.0f));
              midPointY = (int)(y + (sqSize / 2.0f));

              pmidXR = (midPointX + sqSize);
              pmidYU = (midPointY + sqSize);
              pmidXL = (midPointX - sqSize);
              pmidYD = (midPointY - sqSize);

              if (pmidXL <= 0 || pmidYD < 0 || pmidXR >= w - 1
                || pmidYU >= w - 1)
                 continue;

                //Calculate the square value for the corners
              var firstOperand = hMap[midPointX, midPointY] + hMap[x, y] +
              hMap[midPointX, pmidYD] + hMap[cornerPointX, y];
              var secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
              hMap[midPointX, y] =  firstOperand / secondOperand;

              firstOperand = hMap[x, cornerPointY] + hMap[midPointX, midPointY]
        + hMap[cornerPointX, cornerPointY] +        hMap[midPointX, pmidYU];
               secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
                hMap[midPointX, cornerPointY] =  firstOperand / secondOperand;

               firstOperand = hMap[x, y] + hMap[pmidXL,midpointY] + hMap[x,
                cornerPointY] + hMap[midPointX, midPointY];
                secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
                hMap[x, midPointY] = firstOperand / secondOperand;

               firstOperand = hMap[midPointX, y] + hMap[midPointX,midPointY] +
                hMap[cornerPointX, cornerPointY] + hMap[pmidXR, midPointY];
```

```
            secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);

             hMap[x, midPointY] = firstOperand / secondOperand;
    }
  }
```

Лістинг 4. Реалізація алгоритму Square Step

Початкові розрахунки схожі на алгоритм Diamond Step, ми так само знаходимо кути та центральну точку. Після цього ми повинні знайти нові кутові точки, які будуть розбівати нашу мапу висот на нові квадрати. Далі ми перевірємо якщо отримані точки знаходяться на границях террейну, якщо це так, тоді ми пропускаємо їх, та переходимо на наступну ітерації алгоритму.

Наступним кроком ми повинні визначити висоти для нових отриманих точок. Цей процес робиться так само як і при Діамантовому кроці, за одним виключенням. Ми отримаємо висоту для кожної з чотирьох нових точок окремо, використовуючи відповідні до неї координати, які ми отримали на крок раніше.

Результат такого виконання вже значним чином перетворює наш ландшафт на щось схоже на гори та долини, залежно від налаштувань користувача (рис. 3.14).

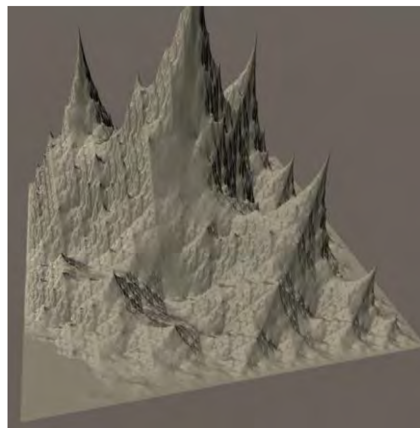


Рисунок 3.14 - Результат виконання алгоритму Square Step після алгоритму Diamond Step

3.3.2.4 Алгоритм згладжування

Дивлячись на отриманий ландшафт за допомогою трьох алгоритмів процедурної генерації ми можемо бачити, що ландшафт хоч і нагадує гори, але має занадто гострі піки та місцями навіть елементи на схилах. З метою покращення результату було реалізовано алгоритм простого згладжування. На рисунку 3.15 ми можемо розглянути UI складову цього алгоритму.

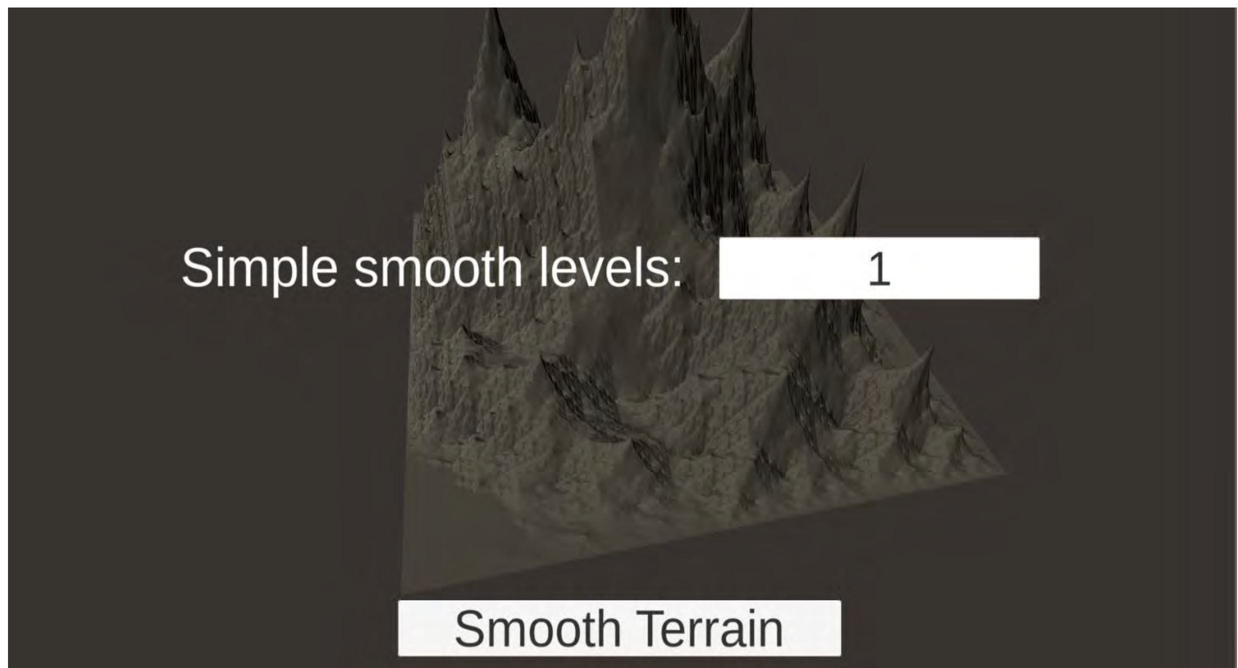


Рисунок 3.15 - UI алгоритму згладжування

Цей алгоритм не є занадто комплексний та не потребує детального налаштування з боку користувача. Він має лише один налаштовує мий параметр, який називається Simple smooth levels. Цей параметр відповідає за кількість так званих рівнів згладжування які будуть застосовані до нашого террейну. Або іншими словами, скільки разів буде застосований алгоритм. Це було зроблене для покращення користування програмою, замість натискання кнопки згладжування декілька разів, алгоритм відпрацює певну кількість самостійно. Розглянемо детальніше роботу алгоритму у лістингу 5.

```
for (int y = 0; y < height; y++)
  {
    for (int x = 0; x < width; x++)
    {
      float averageHeight = 0;
      List<Vector2> neighbours = gameManagerInstance.
         GenerateNeighbours(new Vector2(x, y), width, height);

      foreach (Vector2 n in neighbours)
      {
         averageHeight += heightMap[(int)n.x, (int)n.y];
      }

      heightMap[x, y] = averageHeight / ((float)neighbours.Count + 1);
    }
  }
```

Лістинг 5. Реалізація алгоритму згладжування

Алгоритм складається з двох кроків. Поперше ми знаходимо сусідні вершини на мапі висот для нашої поточної вершини. Для цього використовується функція GenerateNeighbours яка також була розроблена та полягає у знаходжені до чотирьох сусідніх вершин та додаванні їх у список. Після того як ми знайшли сусідні вершини, ми підсумовуємо їхню висоту, знаходимо середньо, та назначаємо нашій вершині. Алгоритм відпрацьовує для усіх вершин на нашому терпейні, та повторюється завдану кількість разів. Результат роботи згладжування ми можемо побачити на рисунку 3.16.

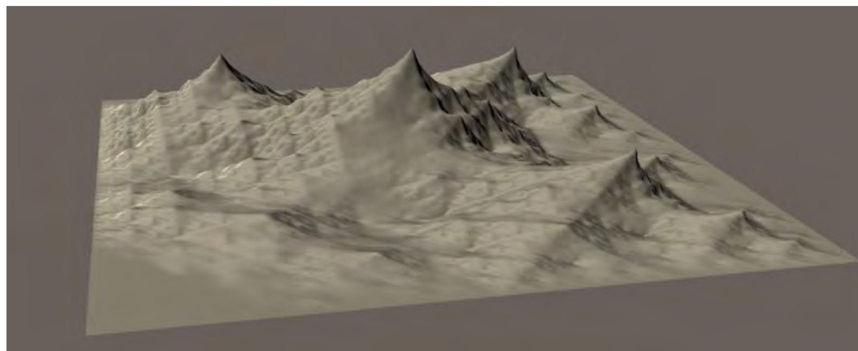


Рисунок 3.16 - Результат згладжування террейну представленного на рисунку 3.15, з завданним рівнем згладжування 3

Хоча цей алгоритм і зменшує загальну висоту сгенерованного ландшафту, але робить його більш натуральним на вигляд.

3.3.3 Процедурне покращення зовнішнього виду 3D ландшафту

3.3.3.1 Алгоритм текстурування террейну

Після генерування ландшафту нам потрібно придати йому привабливого вигляду, а саме додати до нього текстур, які зможуть визначати області з травою, камінням або снігом. Розглянемо UI який використовується для налаштування параметрів текстурування (рис. 3.17). Цей алгоритм має підтримку роботи з багатьма текстурами, та власне це і є наша основна мета. Для цього тут був обраний тип взаємодії з налаштуванням як і в випадку з класичним шумом Перліна. А саме була зроблена таблиця, в якій користувач може редагувати певні змінні, а також за допомогою певних кнопок видаляти шари текстур або додавати нові. Також в таблиці має залишатися хоча б один шар текстури та не може бути видаленим з неї.

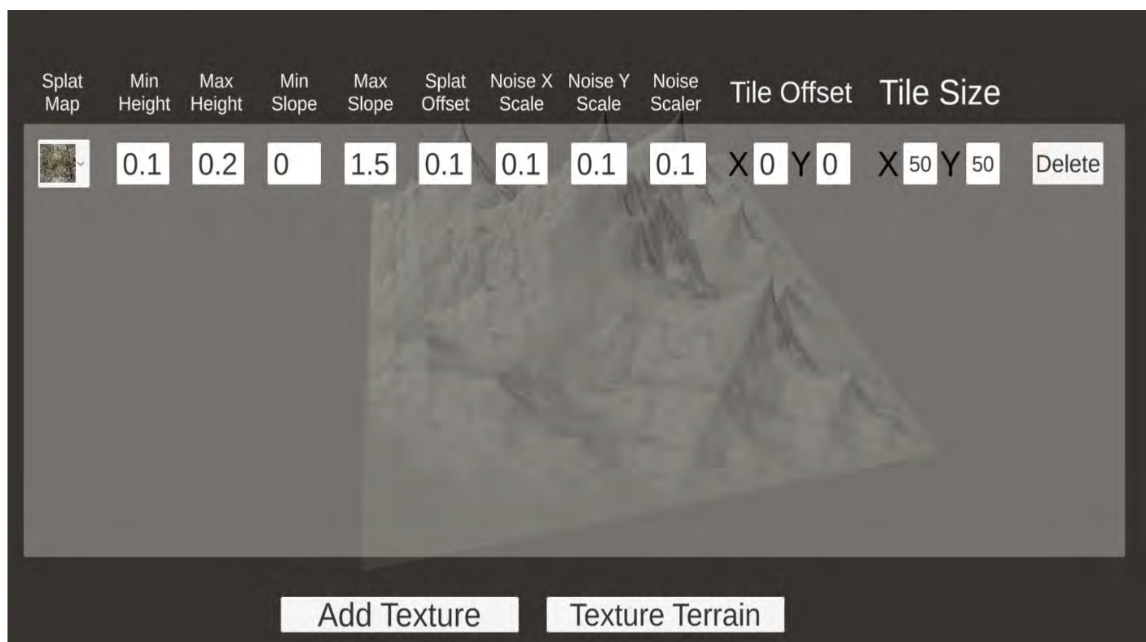


Рисунок 3.17 - UI для налаштування параметрів текстурування

Цей алгоритм має багато параметрів задля більш точного налаштування. Першим параметром є Splat Map, який є дропбоксом та містить у собі ряд заготовленних текстур, які можуть обиратися з цього списку та наноситися на поверхні у будь-яких комбінаціях.

Другими двома параметрами є Min Height та Max Height які відповідають за мінімальну та максимальну висоту на террейні на якій може розташовуватися текстура. Оскільки висота террейну вимірюється від нуля до одиниці, отже і тут ми можемо задавати значення у цьому діапазоні.

Іншим найважливішим параметром є Min та Max Slope. Цей параметр відповідає за кут нахилу на якому може розташовуватися наша текстура. Наприклад ми хочемо аби наша трава не росла на поверхнях з кутом нахилу більше тридцяти градусів, отже після завдання параметру Max Slope цього значення ми не наносити текстуру трави на на поверхні з кутом нахилу більше за завданий. Те ж саме стосується й мінімального кута. Кут нахилу визначається за допомогою перпендикуляру, отже ми можемо задавати значення від нуля до дев’яноста, інші кути просто не будуть мати значення.

Splat Offset відповідає за зміщення текстури відносно початкового варіанту за допомогою отриманого шуму. Це може бути потрібно якщо ми хочемо аби загальна картина текстурування нашого террейну змістилася в якийсь певний напрямок.

Задля покращення візуальної складової в цьому алгоритмі також використовується функція класичного шуму Перліна. Саме цьому ми маємо можливість налаштувати три параметра які використовуються для скейлінгу шуму, а саме Noise X Scale, Noise Y Scale та Noise Scaler. Перші два з перерахованих параметрів відповідають за скейл шуму перед його розрахунками по координатам ікс та ігрик. А останній параметр виступає в якості загального скейлеру, який застосовується в якості коефіцієнту після розрахунків шуму. За допомогою цього ми можемо налаштувати наскільки наш шум буде виходити за межі завдані заздалегідь.

Tile Offset та Tile Size представляють стандартні параметри текстури в Unity, а саме можливість визначати скільки в нас буде тайлів на текстурі, а також наскільки вона буде зміщеною за вісями X та Y. Ці параметрі представлені в якості двомірних векторів, але вводимо ми їх як окремі значення для обох координат.

Розглянемо детальніше реалізацію алгоритму яка представлена в лістингу 6.

```
float[] splatMaps = new float[gameManagerInstance.tData.alphamapLayers];
            for (int i = 0; i < splatH.Count; i++)
            {
             float perlinNoise = Mathf.PerlinNoise(x * splatH[i].noiseXScale, y *
          splatH [i].n oiseYScale) * splatH [i].noiseScaler;
             float offsetValue = splatH [i].offset + perlinNoise;
              float splatHStop = splatH [i].maxH + offsetValue;
              float splatHStart = splatH [i].minH - offsetValue;

            var steepnessX = y /(float)gameManagerInstance.tData.alphamapHeight;
              var steepnesY = x / (float)gameManagerInstance.tData.alphamapWidth;
            float steepness = gameManagerInstance.tData.GetSteepness(steepnessX
            , steepnesY);

              if ((hMap[x, y] >= splatHStart && hMap[x, y] <= splatHStop) &&
              (steepness >= splatH [i].minSlope && steepness <= splatH[i].maxSlope))
               {
                  splatMaps [i] = 1;
               }
            }
            NormalizeVector(splat);
            for (int j = 0; j < splatH.Count; j++)
            {
               splatmapData[x, y, j] = splatMaps [j];
            }
```

Лістинг 6. Реалізація алгоритму нанесення текстур

Спочатку ми створюємо наш масив для внесення в нього даних про наші шари. Система шарів текстур в Unity працює таким чином, що якщо

шари накладуються один на одного, то вираховується процент наповнення між текстурами, який вираховується між нулем та одиницею. Після цього першим кроком ми розраховуємо класичний шум Перліну, який буде робити текстурування нашого терейну більш натуральним, та позбавлятися від ненатуральних різких переходів між текстурами. Далі йдуть розрахунки офсету та висот базуючись на отриманих параметрах класичного шуму Перліна. Для розрахунку куту нахилу використовується вбудована функція GetSteepness. Далі йде перевірка чи задовольняє висота нашої обраної точки текстурування та кут нахилу у цій точці та завданих користувачем параметрів. Якщо точка пройшла перевірку, в массиві текстур цієї точки значення змінюється на одиницю. В протилежному випадку там залишається нуль. За допомогою цих значень Unity визначає чи повинна бути в цій точці текстура, або ні. Далі йде нормалізація вектору, у результаті якої ми отримуємо одиничний вектор, адже саме з такими векторами працює терен. Та на фінальному етапі назначаємо усі текстури на терейні в цій точці. Результат роботи цього алгоритму представлений на рисунку 3.18, для цього були підібрані текстури каміння, трави та снігу на піках.

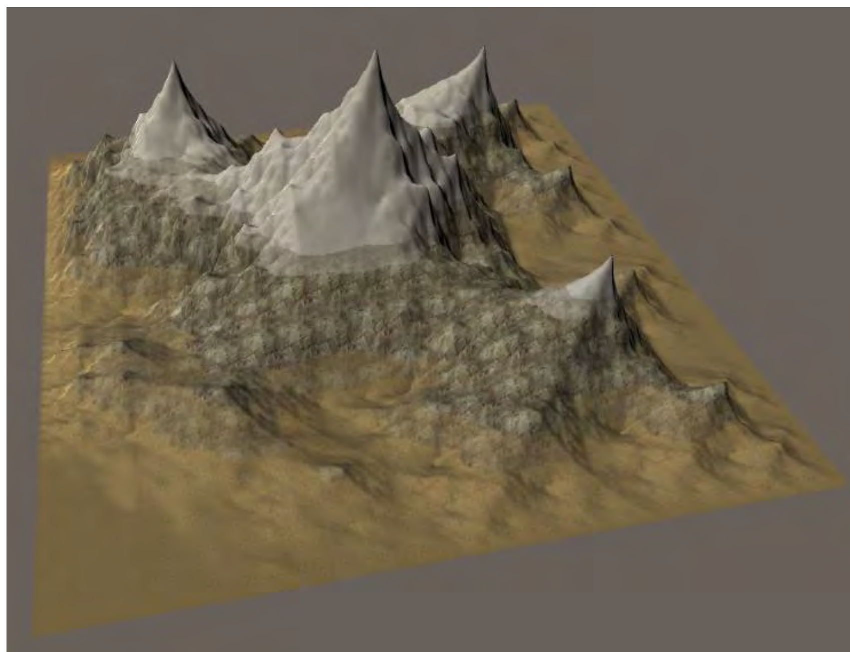


Рисунок 3.18 - Результат процедурного нанесення текстур

4 АНАЛІЗ СТВОРЕННОЇ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ 3D ЛАНДШАФТУ

4.1 Проблема оптимізації рішення

Реалізація процедурної генерації має суттєву проблему оптимізації з якою мені довелося зіштовхнутися під час чого процесу. Програма розроблювалася та тестувалася на PC з такими характеристиками:

– операцій система Windows 10 x64;
– відео карта Geforce GTX 2060ti;
– процесор AMD Ryzen 5 1400x Four-core processor 3.2 GHz;
– 16 Gb оперативної пам'яті.

Проблема виникла під час генерування берегової лінії для води. Щоб отримати більш натуральну берегову лінію, та анімувати хвилі, був реалізований алгоритм, який використовує квадрати з прив'язаною до них анімацією. Але тут виникає проблема, адже квадратів виходить занадто багато на кожній вершині нашої карти висот. Такий підхід був обраний спочатку адже ми маємо розташувати квадрати вздовж берегової лінії, яка генерується процедурно, отже ми не можемо заздалегідь створити 3D модель, яку б просто розміщували на террейні. Результат виявився успішним, але це призвело до значних втрат у фпсі, та як результат у неможливості використовувати такий ландшафт. На рисунку 4.1 ми можемо побачити що кількість фпсу доволі мала, та ніяк не підходить до комфортної гри на такомо згенерованому террейні.

Як ми можемо бачити, кількість фпсу дорівнює 9-10, що ніяк не може бути задовільним результатом для комфортної гри. Це відбувається через велику кількість окремих об'єктів які нам доводиться рендерити одночасно, та які виконують свої скрипти анімації. Все це потребує дуже великої кількості ресурсів, та не є оптимальним підходом.

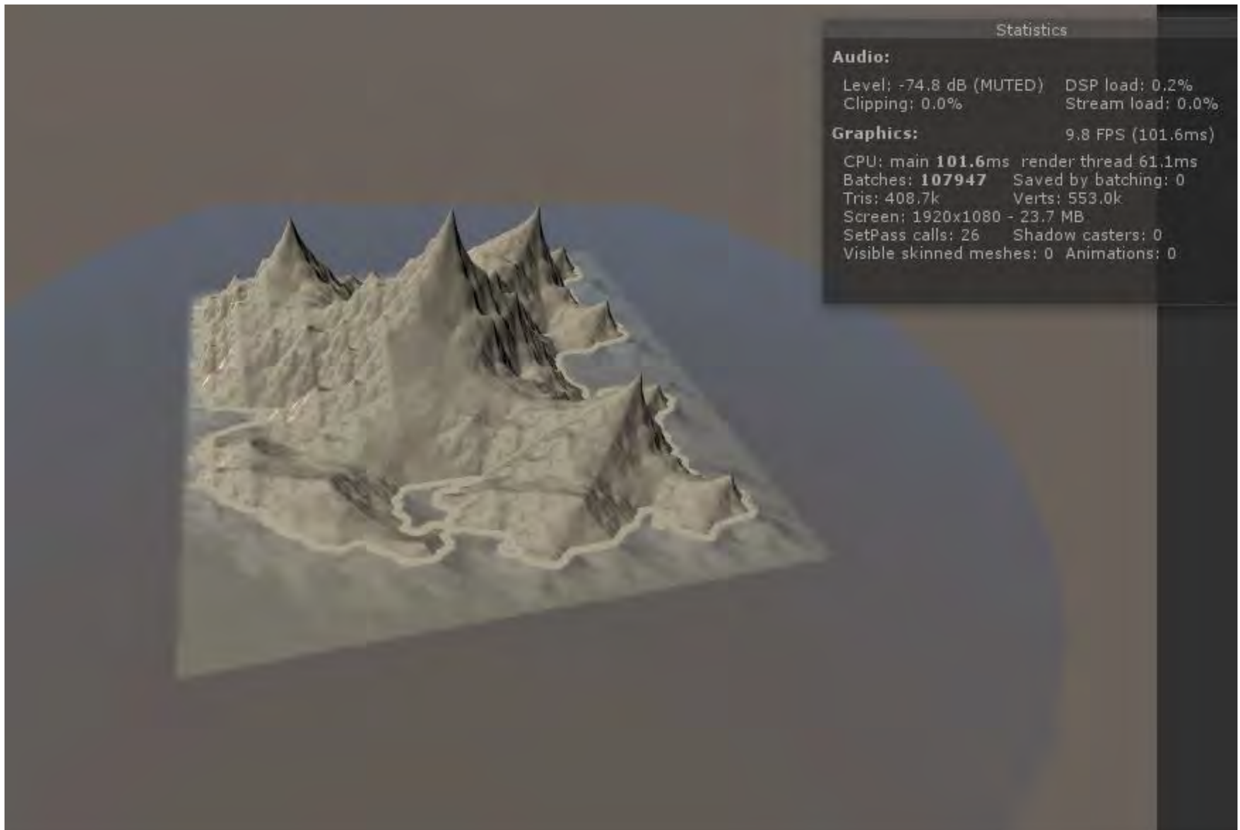


Рисунок 4.1 - Тестування продуктивності рішення з окремими квадратами

Аби покращити результат ми можемо об'єднати всі наші допоміжні меші на береговій лінії в один. Саме це й було зроблено, після генерації квадратів на всіх отриманих точках застосовується метод об'єднання мешів, який є вбудованим в Unity. Завдяки цьому Unity не має звертатися до кожного об'єкту під час його рендеру або взаємодії з ним, та замість працює з одним великим мешем. Як ми можемо побачити з рисунку 4.2, такий підхід в декілька разів підвищив продуктивність програми.

Як результат кількість фпсу зросла до 75-80. Мінусом такого підходу є трохи більша генерація самої берегової лінії, адже ще й затрачується час на об'єднання мешів. Але беручи до уваги що фпс виріс у вісім разів, ми можемо пожертвувати декількома додатковими секундами на генерацію, об отримати іграбельну версію програми.

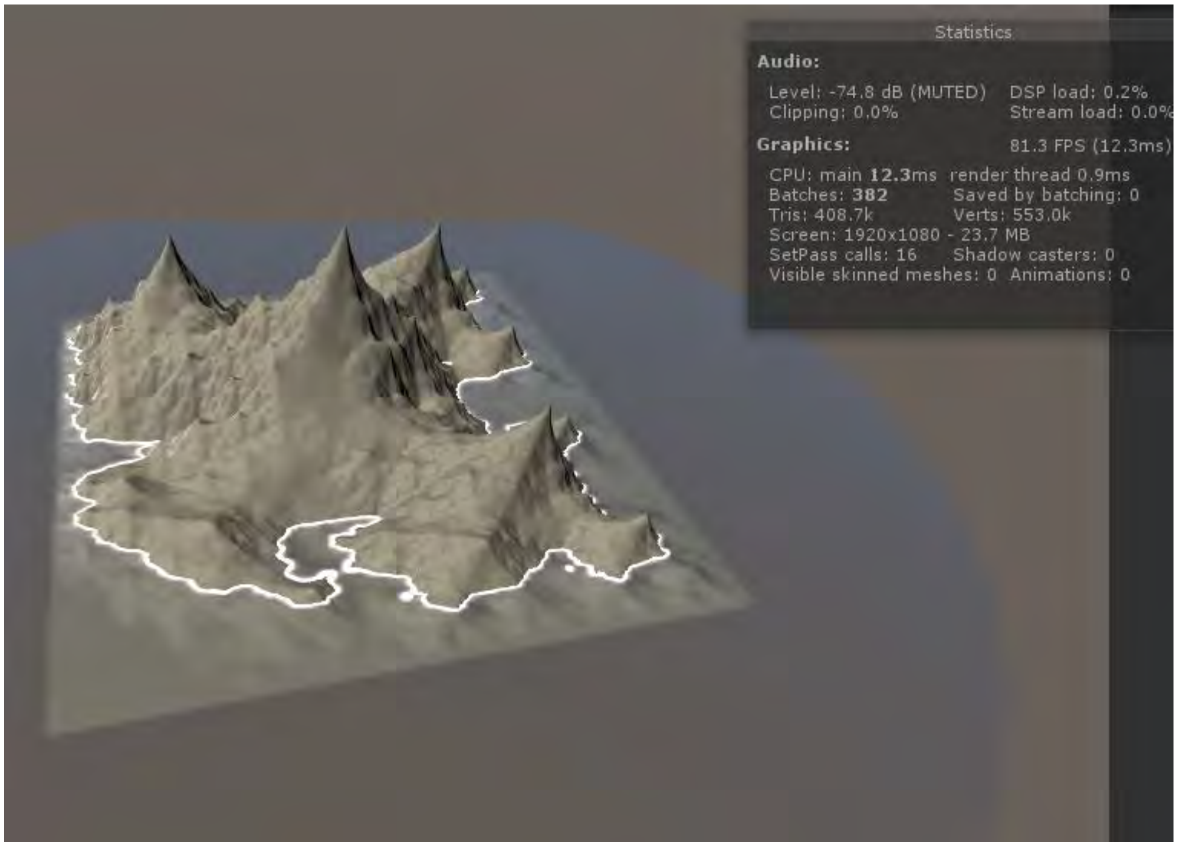


Рисунок 4.2 - Демонстрація підвищенної продуктивності з одним великим мешем

ВИСНОВКИ

В даній кваліфікаційній роботі був створений інструмент для процедурного генерування 3D ландшафту, та надання йому привабливого вигляду процедурними методами. Інструмент був реалізован базуючись вже на існуючих алгоритмах та реалізаціях. В розробленому ПЗ був продемонстрований спосіб створення процедурного ландшафту на основі трьох алгоритмів генерування класичного шуму Перліна, діаграми Вороного та алгоритму зміщення середньої точки. Також була спроба реалізувати базовий варіант налаштування води та генерації берегової лінії. Для покращення результату були також реалізовані текстурування та наповнення ландшафту рослинності. Була надана можливість налаштовувати параметри генерування, завдяки чому цей інструмент можна використовувати у подальшій розробці ігор. Всі текстури та об'єкти рослинності можуть бути зміненими за бажанням, при використанні інструменту у подальшій розробці.

Звісно ж процедурне генерування має ряд своїх недоліків та переваг. До переваг ми звісно ж можемо віднести зменшення затрат часу яке уходить для створення різного роду контенту. Також до переваг відноситься і можливість постійного генерування нового контенту для збільшення реіграбельності. До основного недоліку можна віднести певну неточність процедурного генерування, при застосуванні його у великих масштабах можуть траплятися непередбачувані ситуації, які будуть робити згенерований контент не таким привабливим на натуральним.

Роблячи остаточний висновок можна виділити, що техніка процедурного генерування є дуже потужним інструментом, який може видавати непогані результати за короткий проміжок часу, та в парі з розробником може значно зменшити затрати часу та ресурсів на створення різного роду контенту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Визначення терміну «Процедурна генерація» [Електрон. ресурс]. - Режим доступу: https://uk.wikipedia.org/wiki/Процедурна_генерація.

2. Официальный сайт разработчиков Unity3D [Электрон. ресурс]. – Режим доступа: https://unity.com/ru.

3. Визначення терміну «AAA» [Електрон. ресурс]. – Режим доступу : https://uk.wikipedia.org/wiki/%D0%90%D0%90%D0%90_(%D1%96%D0%BD%D0%B4%D1%83%D1%81%D1%82%D1%80%D1%96%D1%8F_%D0%B2%D1%96%D0%B4%D0%B5%D0%BE%D1%96%D0%B3%D0%BE%D1%80).

4. Heightmaps in Unity [Electronic resource]. – Access mode: https://docs.unity3d.com/Manual/terrain-Heightmaps.html.

5. Working with Terrain in Unity [Electronic resource]. – Access mode: https://docs.unity3d.com/Manual/terrain-Heightmaps.html.

6. Working with Meshes in unity [Electronic resource]. – Access mode: https://docs.unity3d.com/ScriptReference/Mesh.html.

7. Shaker, N. Procedural Content Generation in Games / N. Shaker, J. Togelius, M. J. Nelson. - Springer, 18.10.2016. - 218 p.

8. Development report about using procedural generation in The Witcher 3 [Electronic resource]. – Access mode: https://www.gdcvault.com/play/1020197/Landscape-Creation-and-Rendering-in.

9. Development report about using procedural generation in Horizon Zero Down [Electronic resource]. – Access mode: https://www.guerrilla-games.com/read/gpu-based-procedural-placement-in-horizon-zero-dawn.

10. Доклад разработчиков No Mans Sky о их реализации процедурной генерации [Электрон. ресурс]. – Режим доступа: https://dtf.ru/gamedev/6874-shum-perlina-matematika-i-generaciya-mira-no-man-s-sky.

11. Giffard, P. V. Perlin Noise / P.V. Giffard. - Tort, 2011. – 72 p.

12. Дополнительные данные про реализацию шума Перлина [Электрон. ресурс]. – Режим доступа: https://habr.com/ru/post/142592/.

13. Golossary for Perlin noise [Electronic resource]. – Access mode: http://libnoise.sourceforge.net/glossary/.

14. Sunil Arya, Sunil; Malamatos, Theocharis; Mount, David M. (2002). "Space-efficient approximate Voronoi diagrams". Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. P. 721–730.

15. Liberti, Leo; Lavor, Carlile (2017), Euclidean Distance Geometry: An Introduction, Springer Undergraduate Texts in Mathematics and Technology, Springer.

16. Krause, E. F. Taxicab Geometry [Electronic resource] / E. F. Krause. – Courier Corporation, 01.01.1986. – 88 p.

17. Theory of Midpoint Displacement algorithm [Electronic resource]. – Access mode: https://www.sfu.ca/~rpyke/335/projects/tsai/report1.htm.

18. Implementation of Midpoint Displacement [Electronic resource]. – Access mode: https://bitesofcode.wordpress.com/2016/12/23/landscape-generation-using-midpoint-displacement/.

19. Визначення терміну «Фрактал» [Електрон. ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Фрактал.

20. Peitgen, Heinz-Otto. The Science of fractal images. [Electronic resource] / Heinz-Otto Peitgen, M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, R. F. Voss. - New York: Springer-Verlag, 1988. – 312 p.

ДОДАТОК А

Текст програми

A.1 Текс файла GameManager

```
public class GameManager : MonoBehaviour
{
    public static GameManager Instance;

    public Terrain terrain;
    public TerrainData terrainData;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        terrainData = terrain.terrainData;
    }

    public float[,] GetHeightMap()
    {
        return terrainData.GetHeights(0, 0, terrainData.heightmapWidth,
terrainData.heightmapHeight);
    }

    public void ResetTerrainHeights()
    {
        float[,] heightMap = new float[terrainData.heightmapWidth,
terrainData.heightmapHeight];

        for (int x = 0; x < terrainData.heightmapWidth; x++)
        {
            for (int y = 0; y < terrainData.heightmapHeight; y++)
            {
                heightMap[x, y] = 0;
            }
        }

        terrainData.SetHeights(0, 0, heightMap);
    }

    public List<Vector2> GenerateNeighbours(Vector2 pos, int width,
    int height)
    {
        List<Vector2> neighbours = new List<Vector2>();
```

```
        for (int y = -1; y < 2; y++)
        {
            for (int x = -1; x < 2; x++)
            {
                if (!(x == 0 && y == 0))
                {
                    Vector2 nPos = new Vector2(Mathf.Clamp(pos.x + x, 0, width
                  - 1),  Mathf.Clamp(pos.y + y, 0, height - 1));
                    if (!neighbours.Contains(nPos))
                        neighbours.Add(nPos);
                }
            }
        }

        return neighbours;
    }
}
```

A.2 Текс файла Utils

```
public static class Utils
{
public static float fractalBM(float x, float y, int octaves, float pers)
    {
        float final = 0;
        float frq = 1;
        float amp = 1;
        float maxV = 0;

        for (int i = 0; i < octaves; i++)
        {
            final += Mathf.PerlinNoise(x * frq, y * frq) * amp;
            maxV += amp;
            amp *= pers;
            frq *= 2;
        }

        return final / maxV;
     }

    public static float Map(float value, float originalMin, float originalMax,
    float targetMin, float targetMax)
    {
```

```
        return (value - originalMin) * (targetMax - targetMin) /
            (originalMax - originalMin) + targetMin;
    }
}
```

A.3 Текс файла PerlinParameters

```
[System.Serializable]
public class PerlinParameters
{
    public float mPerlinXScale = 0.01f;
    public float mPerlinYScale = 0.01f;
    public int mPerlinOffsetX = 0;
    public int mPerlinOffsetY = 0;
    public int mPerlinOctaves = 3;
    public float mPerlinPersistance = 9;
    public float mPerlinHeightScale = 0.09f;
    public bool remove = false;
}
```

A.4 Текс файла PerlinNoise

```
public class PerlinNoise : MonoBehaviour
{
    public static PerlinNoise Instance { get; private set; }

    public GameObject scrollViewContent;
    public GameObject perlinSettingsPrefab;

    public List<PerlinParameters> perlinParameters = new
List<PerlinParameters>();

    private GameManager gameManagerInstance;

    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        gameManagerInstance = GameManager.Instance;
        AddNewPerlin();
```

```
        }

        public void MultiplePerlinTerrain()
        {
            float[,] heightMap = gameManagerInstance.GetHeightMap();

            for (int y = 0; y < gameManagerInstance
            .terrainData.heightmapHeight; y++)
            {
                for (int x = 0; x < gameManagerInstance.
                     terrainData.heightmapWidth; x++)
                {
                    foreach (PerlinParameters p in perlinParameters)
                    {
                        heightMap[x, y] += Utils.fBM((x + p.mPerlinOffsetX) *
p.mPerlinXScale,
                            (y + p.mPerlinOffsetY) * p.mPerlinYScale,
                            p.mPerlinOctaves,
                            p.mPerlinPersistance) * p.mPerlinHeightScale;
                    }
                }
            }

            gameManagerInstance.terrainData.SetHeights(0, 0, heightMap);
        }

        public void AddNewPerlin()
        {
            var perlinParameter = new PerlinParameters();
            perlinParameters.Add(perlinParameter);

    var newPerlinSettings = Instantiate(perlinSettingsPrefab,
scrollViewContent.transform).GetComponent<PerlinNoiseSettings>();
            newPerlinSettings.perlinParameters = perlinParameter;
            newPerlinSettings.InitSettings();
        }
    }
```

А.5 Текс файла VoronoiTessellation

```
    public class VoronoiTessellation : MonoBehaviour
    {
        private float minHeight = 0;
        private float maxHeight = 1;
```

```
private int voronoiPeakCount = 2;
private float fallOff = 0.2f;
private float dropOff = 0.6f;
private VoronoiVariant voronoiType = VoronoiVariant.Linear;

private GameManager gameManagerInstance;

public TMP_InputField minHeightTMP;
public TMP_InputField maxHeightTMP;
public TMP_InputField peakCountTMP;
public TMP_InputField fallOffTMP;
public TMP_InputField dropOffTMP;
public TMP_Dropdown typeDropDown;

void Start()
{
   gameManagerInstance = GameManager.Instance;
   InitSettings();
}

private void InitSettings()
{
   minHeightTMP.text = minHeight.ToString();
   maxHeightTMP.text = maxHeight.ToString();
   peakCountTMP.text = voronoiPeakCount.ToString();
   fallOffTMP.text = fallOff.ToString();
   dropOffTMP.text = dropOff.ToString();
   typeDropDown.value = (int)voronoiType;
}

private string ValidateInput(string input, TMP_InputField settingTMP)
{
   if (string.IsNullOrWhiteSpace(input))
   {
      settingTMP.text = "0";
      return "0";
   }

   return input;
}

public void OnMinHeightChange(string value)
{
 minHeight = float.Parse(ValidateInput(value, minHeightTMP));
```

```
}

public void OnMaxHeightChange(string value)
{
maxHeight = float.Parse(ValidateInput(value, maxHeightTMP));
}

public void OnPeakCountChange(string value)
{
    var peaksCount = Convert.ToInt32(ValidateInput(value,
   peakCountTMP));

   if(peaksCount < 0)
   {
      peakCountTMP.text = "0";
      peaksCount = 0;
   }

   voronoiPeakCount = peaksCount;
}

public void OnFallOffChange(string value)
{
   fallOff = float.Parse(ValidateInput(value, fallOffTMP));
}

public void OnDropOffChange(string value)
{
   dropOff = float.Parse(ValidateInput(value, dropOffTMP));
}

public void OnVoronoiTypeChange(Int32 value)
{
   voronoiVariant = (VoronoiType)typeDropDown.value;
   Debug.Log(voronoiType);
}

public void GenerateVoronoiTessellation()
{
   float[,] hMap = gameManagerInstance.GetHeightMap();

   for (int i = 0; i < voronoiPeakCount; i++)
   {
```

```
            Vector3 peak = new Vector3(Random.Range(0,
            gameManagerInstance.tData.heightmapWidth),
            Random.Range(voronoiMinHeight, voronoiMaxHeight),
            Random.Range(0,
            gameManagerInstance.tData.heightmapHeight));

        if (hMap[(int)peak.x, (int)peak.z] < peak.y)
        {
          hMap[(int)peak.x, (int)peak.z] = peak.y;
        }
        else
        {
          continue;
        }

        Vector2 pTransformP = new Vector2(peak.x, peak.z);
            float maxDist = Vector2.Distance(Vector2.zero, new
            Vector2(gameManagerInstance.tData.heightmapWidth,
             gameManagerInstance.tData.heightmapHeight));

        for (int y = 0; y < gameManagerInstance.
            terrainData.heightmapHeight; y++)
        {
          for (int x = 0; x < gameManagerInstance.
            terrainData.heightmapWidth; x++)
          {
            if (!(x == peak.x && y == peak.z))
            {
float dist = Vector2.Distance(pTransformP, new   Vector2(x, y)) / maxDist;
        float height = 0.0f;

              switch (vVariant)
              {
                case VoronoiVariant.Linear:
                  height = peak.y - dist * fallOff;
                  break;
                case VoronoiVariant.Combined:
                  height = peak.y - dist * fallOff -
                      Mathf.Pow(dist, dropOff);
                  break;
                case VoronoiVariant.Power:
                  height = peak.y - Mathf.Pow(dist, dropOff) *
                      fallOff;
                  break;
```

```
                case VoronoiVariant.SinPow:
                    height = peak.y - Mathf.Pow(dist * 3, fallOff) –
                        Mathf.Sin(dist * 2 * Mathf.PI) / dropOff;
                    break;
            }
                if (heightMap[x, y] < h)
                {
                    heightMap[x, y] = h;
                }
            }
        }
    }
}
gameManagerInstance.tData.SetHeights(0, 0, hMap);
    }
}
```

А.6 Текс файла MidpointDisplacement

```
public class MidpointDisplacement : MonoBehaviour
{
    private float mpdMinHeight = -2;
    private float mpdMaxHeight = 2;
    private float mpdHeightDamperPower = 1;
    private float mpdRoughness = 2.0f;
    private GameManager gameManagerInstance;

    public TMP_InputField minHeightTMP;
    public TMP_InputField maxHeightTMP;
    public TMP_InputField heightDamperPowerTMP;
    public TMP_InputField roughnessTMP;

    void Start()
    {
        gameManagerInstance = GameManager.Instance;
        InitSettings();
    }

    private void InitSettings()
    {
        minHeightTMP.text = mpdMinHeight.ToString();
        maxHeightTMP.text = mpdMaxHeight.ToString();
        heightDamperPowerTMP.text = mpdHeightDamperPower.ToString();
        roughnessTMP.text = mpdRoughness.ToString();
```

```
    }

    private string ValidateInput(string input, TMP_InputField settingTMP)
    {
        if (string.IsNullOrWhiteSpace(input))
        {
            settingTMP.text = "0";
            return "0";
        }

        return input;
    }

    public void OnMinHeightChange(string value)
    {
        mpdMinHeight = float.Parse(ValidateInput(value, minHeightTMP));
    }

    public void OnMaxHeightChange(string value)
    {
        mpdMaxHeight = float.Parse(ValidateInput(value, maxHeightTMP));
    }

    public void OnHeightDamperPowerChange(string value)
    {
         mpdHeightDamperPower = float.Parse(ValidateInput(value,
        heightDamperPowerTMP));
    }

    public void OnRoughnessChange(string value)
    {
        mpdRoughness = float.Parse(ValidateInput(value, roughnessTMP));
    }

    public void GenerateMidPointDisplacement()
    {
        float[,] hMap = gameManagerInstance.GetHeightMap();
        int w = gameManagerInstance.tData
        .heightmapResolution -1;
        int sqSize = w;
        float hMin = mpdMinHeight;
        float hMax = mpdMaxHeight;
        float heightDampener = (float)Mathf.Pow(mpdHeightDamperPower, -
  1 * mpdRoughness);
```

```
int cornerX, cornerY;
int midX, midY;
int pmidXL, pmidXR, pmidYU, pmidYD;

while (sqSize > 0)
{
   //Diamond Step
         for (int x = 0; x < w; x += sqSize)
   {
      for (int y = 0; y < w; y += sqSize)
      {
         midPointX = (int)(x + sqSize / 2.0f);
         midPointY = (int)(y + sqSize / 2.0f);
         cornerPointX = x + sqSize;
         cornerPointY = y + sqSize;
       var firstOperand = hMap[x, y] + hMap[cornerPointX, y]
        + hMap[x, cornerPointY] + hMap[cornerPointX, cornerPointY];

   var secondOperand = 4.0f + Random.Range(hMin, hMax);
    hMap[midPointX, midPointY] =  firstOperand / secondOperand;
        }
   }
   //The Square Step
for (int x = 0; x < w; x += sqSize)
        {
           for (int y = 0; y < w; y += sqSize)
           {
              cornerPointX = x + sqSize;
              cornerPointY = y + sqSize;

              midPointX = (int)(x + (sqSize / 2.0f));
              midPointY = (int)(y + (sqSize / 2.0f));

              pmidXR = (midPointX + sqSize);
              pmidYU = (midPointY + sqSize);
              pmidXL = (midPointX - sqSize);
              pmidYD = (midPointY - sqSize);

              if (pmidXL <= 0 || pmidYD < 0 || pmidXR >= w - 1
                || pmidYU >= w - 1)
                 continue;
```

```
        //Calculate the square value for the corners
        var firstOperand = hMap[midPointX, midPointY] + hMap[x, y] +
        hMap[midPointX, pmidYD] + hMap[cornerPointX, y];
        var secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
        hMap[midPointX, y] =  firstOperand / secondOperand;

        firstOperand = hMap[x, cornerPointY] + hMap[midPointX, midPointY]
    + hMap[cornerPointX, cornerPointY] +        hMap[midPointX, pmidYU];
         secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
          hMap[midPointX, cornerPointY] =  firstOperand / secondOperand;

          firstOperand = hMap[x, y] + hMap[pmidXL,midpointY] + hMap[x,
          cornerPointY] + hMap[midPointX, midPointY];
          secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
          hMap[x, midPointY] = firstOperand / secondOperand;

          firstOperand = hMap[midPointX, y] + hMap[midPointX,midPointY] +
          hMap[cornerPointX, cornerPointY] + hMap[pmidXR, midPointY];
          secondOperand = 4.0f + UnityEngine.Random.Range(hMin, hMax);
           hMap[x, midPointY] = firstOperand / secondOperand;
     }
    }
            sqSize = (int)(sqSize / 2.0f);
            hMin *= heightDampener;
            hMax *= heightDampener;
        }
        gameManagerInstance.tData.SetHeights(0, 0, hMap);
    }
}
```

А.7 Текс файла SimpleSmooth

```
public class SimpleSmooth : MonoBehaviour
{
    private int simpleSmoothLevels = 1;
    private GameManager gameManagerInstance;

    public TMP_InputField simpleSmoothLevelsTMP;

    void Start()
    {
        gameManagerInstance = GameManager.Instance;
```

```
    InitSettings();
}

private void InitSettings()
{
    simpleSmoothLevelsTMP.text = simpleSmoothLevels.ToString();
}

public void OnSimpleSmoothLevelsChanged(string value)
{
    if (string.IsNullOrEmpty(value))
    {
        value = "1";
    }
    var levels = Convert.ToInt32(value);
    if (levels < 0)
    {
        levels = 0;
    }

    simpleSmoothLevelsTMP.text = levels.ToString();
    simpleSmoothLevels = levels;
}

public void Smooth()
{
    float[,] heightMap = gameManagerInstance.GetHeightMap();
    int width = gameManagerInstance.terrainData.heightmapResolution;
    int height = gameManagerInstance.terrainData.heightmapResolution;
    int smoothLevels = simpleSmoothLevels;

    while (smoothLevels > 0)
    {
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                float averageHeight = 0;
                List<Vector2> neighbours = gameManagerInstance.
               GenerateNeighbours(new Vector2(x, y), width, height);

                foreach (Vector2 n in neighbours)
                {
                    averageHeight += heightMap[(int)n.x, (int)n.y];
```

```
                }

                heightMap[x, y] = averageHeight /
              ((float)neighbours.Count + 1);
            }
        }
        smoothLevels--;
    }
    gameManagerInstance.terrainData.SetHeights(0, 0, heightMap);
  }
}
```

А.8 Текс файла SplatHeights

```
[System.Serializable]
public class SplatHeights
{
   public Texture2D texture = null;
   public float minH = 0.1f;
   public float maxH = 0.2f;
   public float minSlope = 0;
   public float maxSlope = 1.5f;
   public float splatOffset = 0.1f;
   public float noiseXScale = 0.1f;
   public float noiseYScale = 0.1f;
   public float noiseScaler = 0.1f;
   public Vector2 offset = new Vector2(0, 0);
   public Vector2 tileSize = new Vector2(50, 50);
   public bool remove = false;
}
```

А.9 Текс файла SplatmapGenerator

```
public class SplatmapGenerator : MonoBehaviour
{
   public static SplatmapGenerator Instance { get; private set; }
   public List<Texture2D> splatTextures = new List<Texture2D>();
   public GameObject scrollViewContent;
   public GameObject splatmapSettingsPrefab;

   public List<SplatHeights> splatH = new List<SplatHeights>();

   private GameManager gameManagerInstance;
```

```
    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        gameManagerInstance = GameManager.Instance;
        AddNewSplatMap();
    }

    public void GenerateSplatMaps()
    {
        TerrainLayer[] newSplatPrototypes;
        newSplatPrototypes = new TerrainLayer[splatHeights.Count];
        int spindex = 0;

        foreach (SplatHeights sh in splatHeights)
        {
            newSplatPrototypes[spindex] = new TerrainLayer();
            newSplatPrototypes[spindex].diffuseTexture = sh.texture;
            newSplatPrototypes[spindex].tileOffset = sh.tileOffset;
            newSplatPrototypes[spindex].tileSize = sh.tileSize;
            newSplatPrototypes[spindex].diffuseTexture.Apply(true);
            spindex++;
            Selection.activeObject = gameManagerInstance.terrain.gameObject;
        }
gameManagerInstance.tData.terrainLayers = new SplatPrototypes;

        float[,] hMap = gameManagerInstance.tData.GetHeights(0, 0,
        gameManagerInstance.tData.heightmapResolution,
            gameManagerInstance.tData.heightmapResolution);

        float[,,] splatmapData = new float[gameManagerInstance.
        terrainData.alphamapWidth,
        gameManagerInstance.terrainData.alphamapHeight,
        gameManagerInstance.terrainData.alphamapLayers];

        for (int y = 0; y < gameManagerInstance.tData.alphamapHeight;
         y++)
        {
            for (int x = 0; x < gameManagerInstance.
        tData.alphamapWidth; x++)
            {
```

```
float[] splatMaps = new float[gameManagerInstance.tData.alphamapLayers];
          for (int i = 0; i < splatH.Count; i++)
          {
           float perlinNoise = Mathf.PerlinNoise(x * splatH[i].noiseXScale, y *
        splatH [i].noiseYScale) * splatH [i].noiseScaler;
           float offsetValue = splatH [i].offset + perlinNoise;
            float splatHStop = splatH [i].maxH + offsetValue;
            float splatHStart = splatH [i].minH - offsetValue;

          var steepnessX = y /(float)gameManagerInstance.tData.alphamapHeight;
            var steepnesY = x / (float)gameManagerInstance.tData.alphamapWidth;
          float steepness = gameManagerInstance.tData.GetSteepness(steepnessX
          , steepnesY);

            if ((hMap[x, y] >= splatHStart && hMap[x, y] <= splatHStop) &&
            (steepness >= splatH [i].minSlope && steepness <= splatH[i].maxSlope))
             {
                splatMaps [i] = 1;
             }
          }
          NormalizeVector(splat);
          for (int j = 0; j < splatH.Count; j++)
          {
             splatmapData[x, y, j] = splatMaps [j];
          }
                }
             }
             gameManagerInstance.terrainData.SetAlphamaps(0, 0, splatmapData);
           }

           public void AddNewSplatMap()
           {
             var splatHeight = new SplatHeights();
             splatHeight.texture = splatTextures[0];
             splatHeights.Add(splatHeight);

             var newSplatMapSettings = Instantiate(splatmapSettingsPrefab,
             scrollViewContent.transform).GetComponent<SplatMapSettings>();
             newSplatMapSettings.splatHeights = splatHeight;
             newSplatMapSettings.InitSettings();
           }

           private void NormalizeVector(float[] data)
```

```
    {
        float total = 0;
        for (int i = 0; i < data.Length; i++)
        {
            total += data[i];
        }

        for (int i = 0; i < data.Length; i++)
        {
            data[i] /= total;
        }
    }
}
```

А.10 Текс файла Vegetation

```
[System.Serializable]
public class Vegetation
{
    public GameObject mesh;
    public float minSlope = 0;
    public float maxSlope = 90;

    public float minHeight = 0.1f;
    public float maxHeight = 0.2f;
    public float minScale = 0.5f;
    public float maxScale = 0.95f;
    public float minRotation = 0;
    public float maxRotation = 360;
    public float density = 0.5f;

    public Color firstColorVariant = Color.yellow;
    public Color secondColorVariant = Color.green;
    public Color lightMapColor = Color.white;

    public bool remove = false;
}
```

А.11 Текс файла VegetationGenerator

```
public class VegetationGenerator : MonoBehaviour
```

```
{
    private const int TERRAIN_LAYER = 9;

    public List<GameObject> treesGO = new List<GameObject>();
    public GameObject scrollViewContent;
    public GameObject vegetationSettingsPrefab;

    public List<Vegetation> vegetation = new List<Vegetation>();
    public int treeSpacing = 2;
    public int maxTrees = 10000;

    private GameManager gameManagerInstance;
    public static VegetationGenerator Instance { get; private set; }

    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        gameManagerInstance = GameManager.Instance;
        AddNewVegetation();
    }

    public void AddNewVegetation()
    {
        var vegetationParams = new Vegetation();
        vegetationParams.mesh = treesGO[0];
        vegetation.Add(vegetationParams);

        var newVegetationSetttings = Instantiate(vegetationSettingsPrefab,
scrollViewContent.transform).GetComponent<VegetationSettings>();
        newVegetationSetttings.vegetationParams = vegetationParams;
        newVegetationSetttings.InitSettings();
    }

    public void PlantVegetation()
    {
        TreePrototype[] newTreePrototypes;
        newTreePrototypes = new TreePrototype[vegetation.Count];
        int tindex = 0;
        foreach (Vegetation veg in vegetation)
        {
            newTreePrototypes[tindex] = new TreePrototype();
```

```
        newTreePrototypes[tindex].prefab = veg.mesh;
        tindex++;
    }
    gameManagerInstance.terrainData.treePrototypes =
    newTreePrototypes;

    List<TreeInstance> allVegetation = new List<TreeInstance>();
     for (int z = 0; z < gameManagerInstance.terrainData.size.z; z +=
    treeSpacing)
    {
      for (int x = 0; x < gameManagerInstance.terrainData.size.x; x +=
    treeSpacing)
        {
            for (int tp = 0; tp < gameManagerInstance
              .terrainData.treePrototypes.Length; tp++)
            {
                if (Random.Range(0.0f, 1.0f) > vegetation[tp].density)
                    break;

                float thisHeight = gameManagerInstance.terrainData
                .GetHeight(x, z) / gameManagerInstance.terrainData.size.y;
                float thisHeightStart = vegetation[tp].minHeight;
                float thisHeightEnd = vegetation[tp].maxHeight;

float steepness = gameManagerInstance.terrainData.GetSteepness(x /
gameManagerInstance.terrainData.size.x, z / gameManagerInstance
.terrainData.size.z);

                if (thisHeight >= thisHeightStart && thisHeight <=
thisHeightEnd &&
                    (steepness >= vegetation[tp].minSlope && steepness <=
vegetation[tp].maxSlope))
                {
                    TreeInstance instance = new TreeInstance();
                 instance.position = new Vector3((x + Random.Range(-25.0f,
                25.0f)) / gameManagerInstance.terrainData.size.x, thisHeight, (z
                + Random.Range(-25.0f, 25.0f)) /
                gameManagerInstance.terrainData.size.z);

                    Vector3 treeWorldPos = new Vector3(instance.position.x *
                gameManagerInstance.terrainData.size.x, instance.position.y *
                gameManagerInstance.terrainData.size.y, instance.position.z *
                gameManagerInstance.terrainData.size.z)
                gameManagerInstance.terrain.transform.position;
```

```
                RaycastHit hit;
                int layerMask = 1 << TERRAIN_LAYER;
                if (Physics.Raycast(treeWorldPos + new Vector3(0, 10, 0),
            Vector3.down, out hit, 200, layerMask) ||
            Physics.Raycast(treeWorldPos - new Vector3(0, 10, 0),
            Vector3.up, out hit, 200, layerMask))
                {
          float treeHeight = (hit.point.y - gameManagerInstance
          .terrain.transform.position.y) / gameManagerInstance
          .terrainData.size.y;
          instance.position = new Vector3(instance.position.x, treeHeight,
          instance.position.z);

          instance.position = new Vector3(instance.position.x *
          gameManagerInstance.terrainData.size.x / gameManagerInstance
          .terrainData.alphamapWidth, instance.position.y,
          instance.position.z * gameManagerInstance.terrainData.size.z /
          gameManagerInstance.terrainData.alphamapHeight);
          instance.rotation = Random.Range(vegetation[tp].minRotation,
          vegetation[tp].maxRotation);
                    instance.prototypeIndex = tp;
               instance.color = Color.Lerp(vegetation[tp].firstColorVariant,
            vegetation[tp].secondColorVariant, Random.Range(0.0f, 1.0f));
                   instance.lightmapColor = vegetation[tp].lightMapColor;
                    float scale =  Random.Range(vegetation[tp].minScale,
            vegetation[tp].maxScale);
                    instance.heightScale = scale;
                    instance.widthScale = scale;
                    allVegetation.Add(instance);
                }
              }
              if (allVegetation.Count >= maxTrees)
              {
            gameManagerInstance.terrainData.treeInstances = allVegetation
            .ToArray();
                return;
              }
          }
      }
    }
    gameManagerInstance.terrainData.treeInstances = allVegetation
    .ToArray();
}
```

```
}
```

А.12 Текс файла SimpleWater

```
public class SimpleWater : MonoBehaviour
{
    public GameObject waterGO;
    public Material shoreLineMaterial;
    public TMP_InputField waterHeightTMP;

    private float waterHeight = 0.03f;
    private GameManager gameManagerInstance;

    void Start()
    {
        gameManagerInstance = GameManager.Instance;
        InitSettings();
    }

    private void InitSettings()
    {
        waterHeightTMP.text = waterHeight.ToString();
    }

    public void OnWaterHeightChanged(string value)
    {
        if (string.IsNullOrEmpty(value))
        {
            value = "0";
        }
        var height = float.Parse(value);
        if (height < 0 || height > 1)
        {
            height = 0;
            waterHeightTMP.text = height.ToString();
        }
        waterHeight = height;
    }

    public void AddWater()
    {
        GameObject water = GameObject.Find("water");
        if (!water)
        {
```

```
    water = Instantiate(waterGO, gameManagerInstance.terrain.transform
    .position, gameManagerInstance.terrain.transform.rotation);
      water.name = "water";
    }
    water.transform.position = gameManagerInstance.terrain.transform
    .position +  Vector3(gameManagerInstance.terrainData.size.x / 2,
    waterHeight * gameManagerInstance.terrainData.size.y,
    gameManagerInstance.terrainData.size.z / 2);
     water.transform.localScale = new Vector3(gameManagerInstance
    .terrainData.size.x, 1, gameManagerInstance.terrainData.size.z);
}

public void DrawShoreline()
{
    float[,] heightMap = gameManagerInstance.terrainData.GetHeights(0,
    0, gameManagerInstance.terrainData.heightmapResolution,
    gameManagerInstance.terrainData.heightmapResolution);

    int quadCount = 0;
    for (int y = 0; y < gameManagerInstance.terrainData.
          heightmapResolution; y++)
    {
      for (int x = 0; x < gameManagerInstance.terrainData.
          heightmapResolution; x++)
      {
        //find spot on shore
        Vector2 thisLocation = new Vector2(x, y);
          List<Vector2> neighbours =
          gameManagerInstance.GenerateNeighbours(thisLocation,
          gameManagerInstance.terrainData.heightmapResolution,
          gameManagerInstance.terrainData.heightmapResolution);
        foreach (Vector2 n in neighbours)
        {
          if (heightMap[x, y] < waterHeight && heightMap[(int)n.x,
          (int)n.y] > waterHeight)
            {
              quadCount++;
              GameObject go = GameObject.
                  CreatePrimitive(PrimitiveType.Quad);
              go.transform.localScale *= 20.0f;

              go.transform.position = gameManagerInstance.terrain
              .transform.position + new Vector3(y /
          (float)gameManagerInstance.terrainData.heightmapResolution
```

```
        * gameManagerInstance.terrainData.size.z, waterHeight *
        gameManagerInstance.terrainData.size.y, x /
        (float)gameManagerInstance.terrainData.heightmapResolution
        * gameManagerInstance.terrainData.size.x);

            go.transform.LookAt(new Vector3(n.y /
        (float)gameManagerInstance.terrainData.heightmapResolution*
        gameManagerInstance.terrainData.size.z, waterHeight *
        gameManagerInstance.terrainData.size.y, n.x /
        (float)gameManagerInstance.terrainData.heightmapResolution
        * gameManagerInstance.terrainData.size.x));

            go.transform.Rotate(90, 0, 0);

            go.tag = "Shore";
          }
        }
      }
    }

GameObject[] shoreQuads =
GameObject.FindGameObjectsWithTag("Shore");
MeshFilter[] meshFilters = new MeshFilter[shoreQuads.Length];
for (int m = 0; m < shoreQuads.Length; m++)
{
   meshFilters[m] = shoreQuads[m].GetComponent<MeshFilter>();
}
        CombineInstance[] combine = new
CombineInstance[meshFilters.Length];
int i = 0;
while (i < meshFilters.Length)
{
   combine[i].mesh = meshFilters[i].sharedMesh;
   combine[i].transform = meshFilters[i].transform
        .localToWorldMatrix;
   meshFilters[i].gameObject.SetActive(false);
   i++;
}

GameObject currentShoreLine = GameObject.Find("ShoreLine");
if (currentShoreLine)
{
   DestroyImmediate(currentShoreLine);
}
```

```
        GameObject shoreLine = new GameObject();
        shoreLine.name = "ShoreLine";
        shoreLine.AddComponent<WaveAnimation>();
        shoreLine.transform.position = gameManagerInstance.terrain
        .transform.position;
        shoreLine.transform.rotation = gameManagerInstance.terrain
        .transform.rotation;
        MeshFilter thisMF = shoreLine.AddComponent<MeshFilter>();
        thisMF.mesh = new Mesh();

        shoreLine.GetComponent<MeshFilter>().sharedMesh.CombineMeshs
        (combine);

        MeshRenderer r = shoreLine.AddComponent<MeshRenderer>();
        r.sharedMaterial = shoreLineMaterial;

        for (int sQ = 0; sQ < shoreQuads.Length; sQ++)
            DestroyImmediate(shoreQuads[sQ]);
    }
}
```

A.13 Текс файла DetailGenerator

```
public class DetailGenerator : MonoBehaviour
{
    public List<GameObject> detailsObjects = new List<GameObject>();
    public List<Texture2D> detailsTextures = new List<Texture2D>();
    public GameObject scrollViewContent;
    public GameObject detailSettingsPrefab;

    public List<Detail> details = new List<Detail>();
    public int detailSpacing = 2;
    public int maxDetails = 10000;

    private GameManager gameManagerInstance;
    public static DetailGenerator Instance { get; private set; }

    private void Start()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        gameManagerInstance = GameManager.Instance;
```

```
        AddNewVegetation();
    }

    public void AddDetails()
    {
        DetailPrototype[] newDetailPrototypes;
        newDetailPrototypes = new DetailPrototype[details.Count];
        int dIndex = 0;
        float[,] heightMap = gameManagerInstance.terrainData
        .GetHeights(0, 0, gameManagerInstance.terrainData
        .heightmapResolution,gameManagerInstance.terrainData.heightmapR
        esolution);

        foreach (Detail d in details)
        {
            newDetailPrototypes[dIndex] = new DetailPrototype();
            newDetailPrototypes[dIndex].prototype = d.prototype;
        newDetailPrototypes[dIndex].prototypeTexture = d.prototypeTexture;
            newDetailPrototypes[dIndex].healthyColor = d.healthyColour;
            newDetailPrototypes[dIndex].dryColor = d.dryColour;
            newDetailPrototypes[dIndex].minHeight = d.heightRange.x;
            newDetailPrototypes[dIndex].maxHeight = d.heightRange.y;
            newDetailPrototypes[dIndex].minWidth = d.widthRange.x;
            newDetailPrototypes[dIndex].maxWidth = d.widthRange.y;
            newDetailPrototypes[dIndex].noiseSpread = d.noiseSpread;

            if (newDetailPrototypes[dIndex].prototype)
            {
                newDetailPrototypes[dIndex].usePrototypeMesh = true;
                  newDetailPrototypes[dIndex].renderMode = DetailRenderMode
                  .VertexLit;
            }
            else
            {
                newDetailPrototypes[dIndex].usePrototypeMesh = false;
                newDetailPrototypes[dIndex].renderMode = DetailRenderMode
                  .GrassBillboard;
            }
            dIndex++;
        }
    gameManagerInstance.terrainData.detailPrototypes = newDetailPrototypes;

        for (int i = 0; i < gameManagerInstance.terrainData.detailPrototypes
        .Length; ++i)
```

```
{
        int[,] detailMap = new int[gameManagerInstance.terrainData
        .detailWidth, gameManagerInstance.terrainData.detailHeight];
        for (int y = 0; y < gameManagerInstance.terrainData
        .detailHeight; y += detailSpacing)
         {
           for (int x = 0; x < gameManagerInstance.terrainData
        .detailWidth; x += detailSpacing)
             {
               if (UnityEngine.Random.Range(0.0f, 1.0f) >
        details[i].density)
          continue;

         int xHM = (int)(x / (float)gameManagerInstance.terrainData
         .detailWidth * gameManagerInstance.terrainData
         .heightmapResolution);
         int yHM = (int)(y / (float)gameManagerInstance
         .terrainData.detailHeight * gameManagerInstance.terrainData
         .heightmapResolution);

         float thisNoise = Utils.Map(Mathf.PerlinNoise(x *
        details[i].feather, y * details[i].feather), 0, 1, 0.5f, 1);
         float thisHeightStart = details[i].minHeight * thisNoise -
                        details[i].overlap * thisNoise;
         float nextHeightStart = details[i].maxHeight * thisNoise +
                        details[i].overlap * thisNoise;

         float thisHeight = heightMap[yHM, xHM];
        float steepness = gameManagerInstance.terrainData
        .GetSteepness(xHM / (float)gameManagerInstance
        .terrainData.size.x, yHM / (float)gameManagerInstance
        .terrainData.size.z);
         if ((thisHeight >= thisHeightStart && thisHeight <=
        nextHeightStart) && (steepness >= details[i].minSlope
         && steepness <= details[i].maxSlope))
          {
             detailMap[y, x] = 1;
          }
       }
    }
  gameManagerInstance.terrainData.SetDetailLayer(0, 0, i, detailMap);
  }
}
```

```
    public void AddNewVegetation()
    {
        var detailsParams = new Detail();
        detailsParams.prototypeTexture = detailsTextures[0];
        details.Add(detailsParams);

        var newVegetationSetttings = Instantiate(detailSettingsPrefab,
        scrollViewContent.transform).GetComponent<DetailSettings>();
        newVegetationSetttings.detailParams = detailsParams;
        newVegetationSetttings.InitSettings();
    }
}
```

ДОДАТОК Б

Слайди презентації

ДОСЛІДЖЕННЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ЛАНДШАФТУ ДЛЯ 3D ІГОР

Виконав: студент 2 курсу групи КНТ-110м
Загоняйко А.А.
Керівник: професор кафедри ПЗ Дубровін В.І.

1

Рисунок Б.1 - Слайд 1

- Об'єкт дослідження – процес генерування 3D ландшафту.
- Предмет дослідження – комп'ютерні ігри з повноцінним або частковим застосуванням алгоритмів процедурної генерації.
- Мета роботи – створення програмної системи для процедурного генерування ландшафту з налаштуванням параметрів генерування самого ландшафту, а також різного роду рослин та природних об'єктів.

2

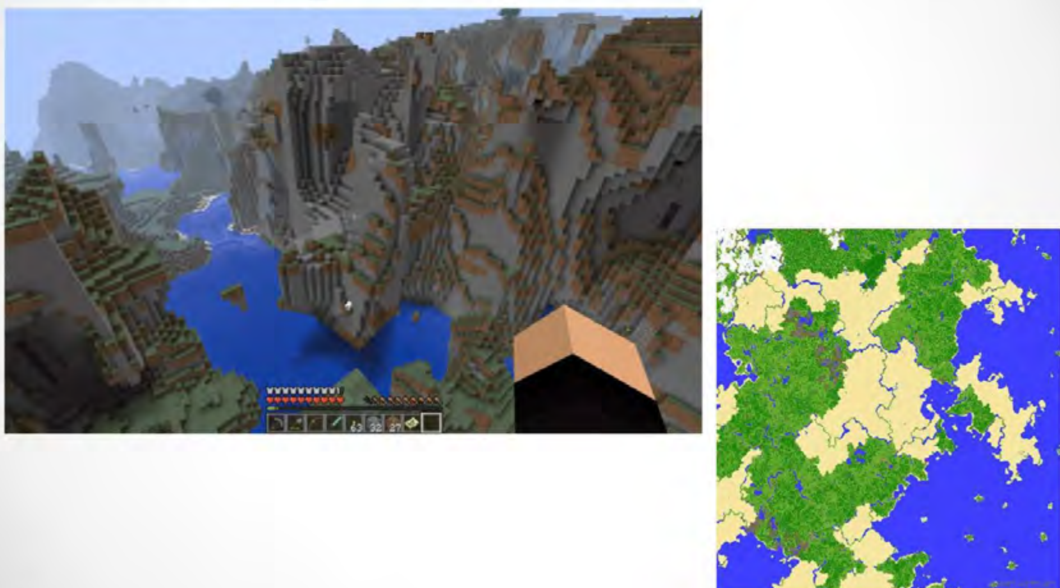
Рисунок Б.2 - Слайд 2

Процедурна генерація світів у грі No Man’s Sky

3

Рисунок Б.3 - Слайд 3

Генерація ландшафту в грі Minecraft



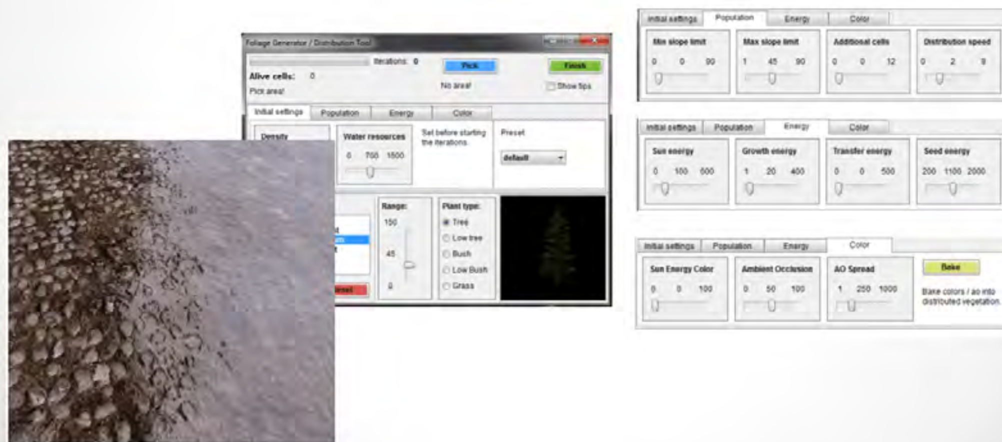
4

Рисунок Б.4 - Слайд 4

Процедурна генерація в грі The Witcher 3

GAME DEVELOPERS CONFERENCE® 2014 MARCH 17–21, 2014 GDCONF.CO

Vegetation generator tool



5

Рисунок Б.5 - Слайд 5

ВИКОРИСТАНІ МЕТОДИ

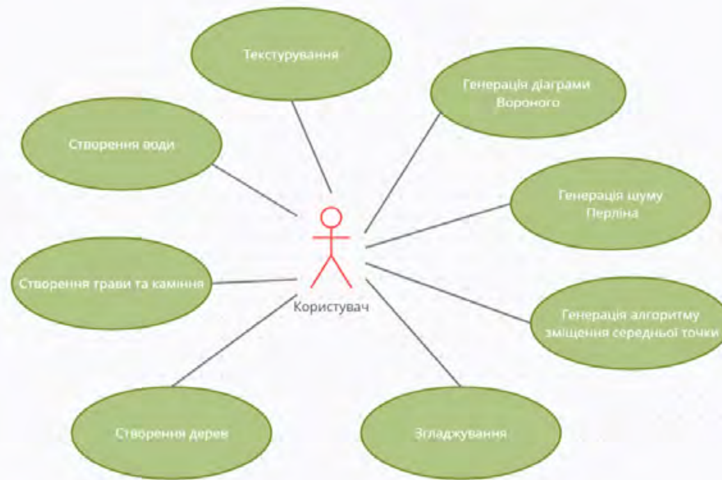
При розробці програми для процедурного генерування 3D ландшафтів були використані наступні методи:

1. Шум Перліна.
2. Побудова діаграми Вороного.
3. Алгоритм зміщення середньої точки.

6

Рисунок Б.6 - Слайд 6

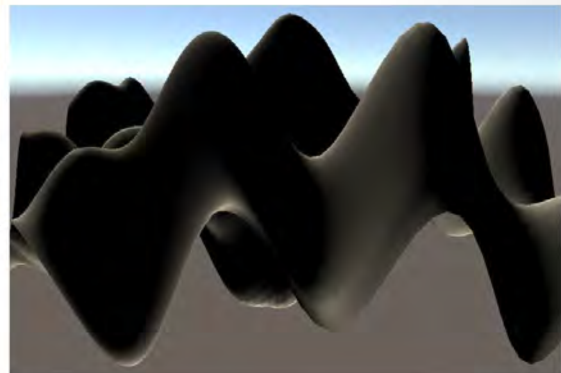
Use case діаграма



7

Рисунок Б.7 - Слайд 7

Шум Перліна



8

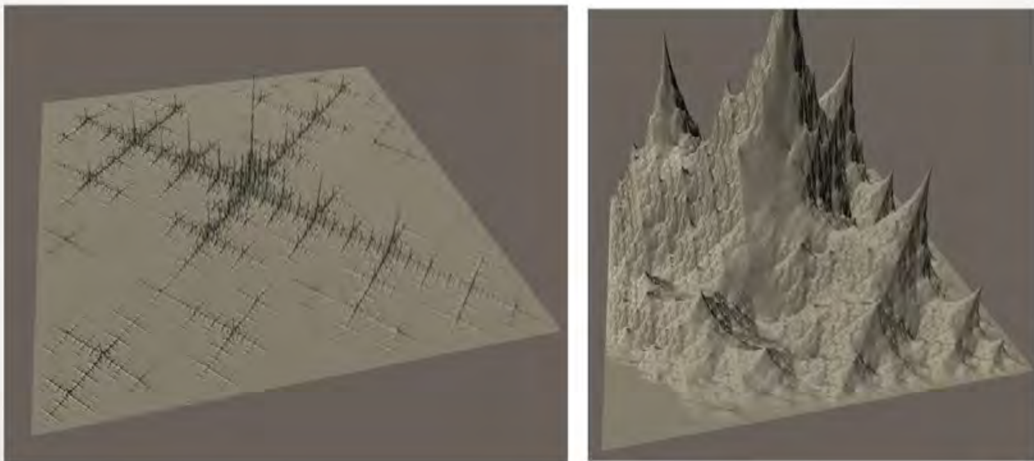
Рисунок Б.8 - Слайд 8

Діаграма Вороного

9

Рисунок Б.9 - Слайд 9

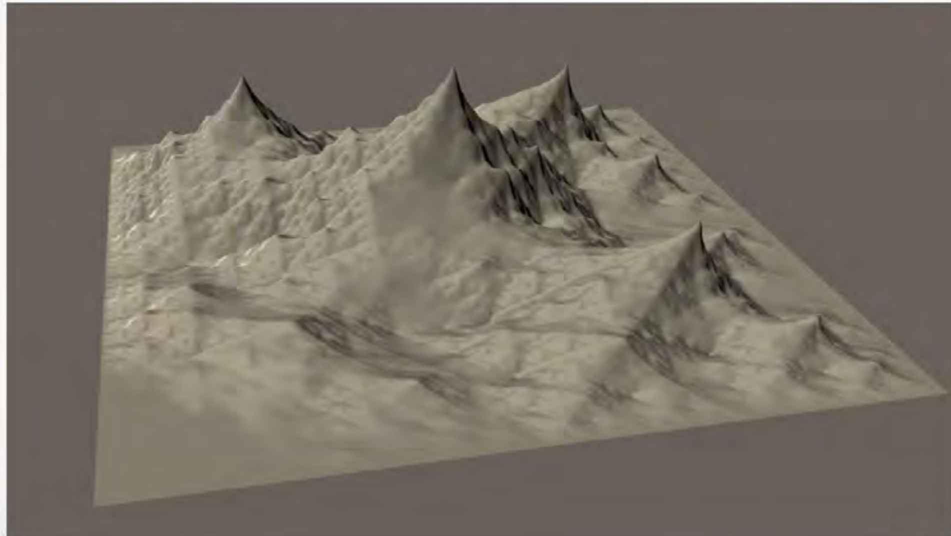
Алгоритм зміщення середньої точки



10

Рисунок Б.10 - Слайд 10

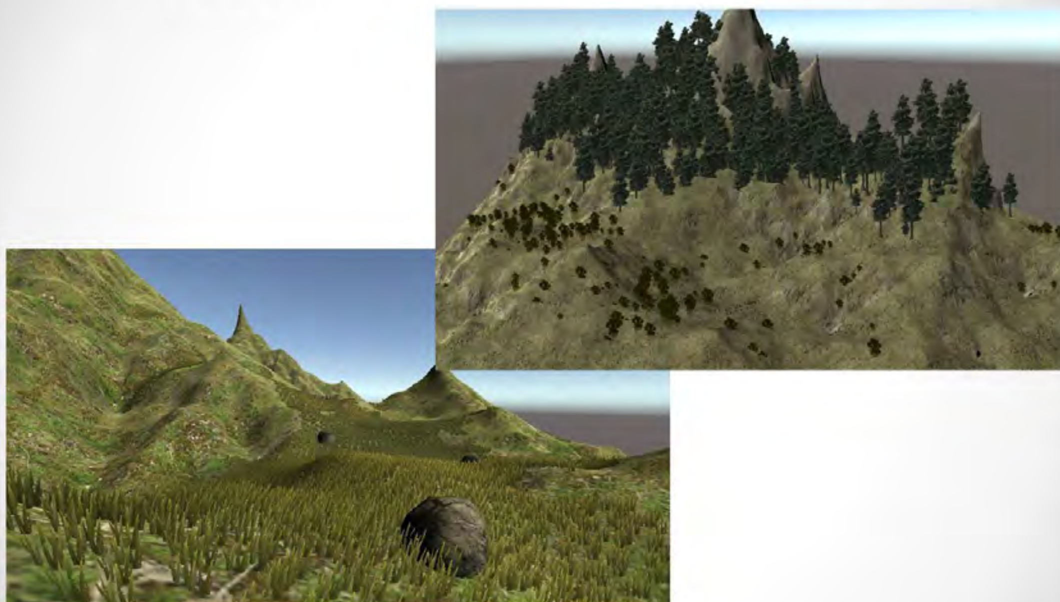
Згладжування



11

Рисунок Б.11 - Слайд 11

Текстурування, дерева та деталізація



12

Рисунок Б.12 - Слайд 12

Вода та берегова лінія

13

Рисунок Б.13 - Слайд 13

Інструменти розробки

14

Рисунок Б.14 - Слайд 14

Вибір ігрового двигуну для розробки

Характеристика	Unity	Unreal Engine
Набір необхідних інструментів для генерування	+	+
Простота використання	+	+\-
Підходить для створення невеликих проєктів	+	-
Кросплатформеність	+	+

15

Рисунок Б.15 - Слайд 15

Висновки

Роблячі висновки можна виділити що техніка процедурного генерування є дуже потужним інструментом, який може видавати непогані результати за короткий проміжок часу, та в парі з розробником може значно зменшити затрати часу та ресурсів на створення різного роду контенту.

16

Рисунок Б.16 - Слайд 16