СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної випускної роботи

освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

на тему „Відеогра у жанрі rogue-lite з процедурною генерацією рівнів”

Виконав: студент групи ІПЗ-20д	_____ (підпис)	Є.О. Осінов (ініціали і прізвище)
Керівник	_____ (підпис)	В.Г. Іванов (ініціали і прізвище)
Завідувач кафедри	_____ (підпис)	О.І. Захожай (ініціали і прізвище)

Рецензент___В.О. Лифар_____

Київ – 2024

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Захожай О.І.
“___” _____ 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Осінов Євген Олексійович
(прізвище, ім’я, по батькові)

1. Тема роботи: Відеогра у жанрі rogue-lite з процедурною генерацією рівнів
керівник роботи Іванов Віталій Геннадійович, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від “06” травня 2024 року №171/15.15-С

2. Строк подання студентом роботи 13.06.2024р.
3. Вихідні дані до роботи: Об’єктом даної роботи є створення відеогри у жанрі rogue-lite з реалізацією алгоритмів процедурної генерації

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Вступ. Аналітичний огляд. Проектування відеогри. Процес розробки та його результат. Висновок. Перелік використаних джерел.

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 30.03.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	30.03.24	виконано
2	Укладання і погодження з керівником плану і етапів виконання роботи	07.04.24	виконано
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	20.04.24	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	01.05.24	виконано
5	Укладання та тестування програмного продукту	05.05.24	виконано
6	Укладання, оформлення та погодження пояснювальної записки з керівником	10.05.24	виконано
7	Здача готової пояснювальної записки на кафедру	13.06.24	виконано
8	Укладання доповіді і презентації	14.06.24	виконано

Студент _____ _____Є.О. Осінов_____
(підпис) (ініціали і прізвище)

Керівник роботи _____ _____В.Г. Іванов_____
(підпис) (ініціали і прізвище)

ЛИСТ ПОГОДЖЕННЯ І ОЦІНЮВАННЯ
дипломної роботи студента гр. ІПЗ-20д Осінов Є. О.

Науковий керівник

Доцент, к.т.н. _____ Іванов В.Г.

Оцінка наукового керівника: _____

Рецензент Лифар Володимир Олексійович, СНУ ім. В.Даля, проф. каф. ІТП
ПІБ, місто роботи, посада

Оцінка рецензента: _____

Кінцева оцінка за результатами захисту:
_____

Голова ЕК
Професор кафедри ІТП
д.т.н. _____ Меняйленко О.С.
підпис

РЕФЕРАТ

Робота містить: 50 сторінок основного тексту, 24 сторінок додатків, 41 рисунків, 19 використаних джерел.

Метою випускної кваліфікаційної роботи є вивчення та використання алгоритмів процедурної генерації ландшафтів та ігрових рівнів у відеоіграх жанру rogue-lite.

У роботі детально розглянуто основні принципи процедурної генерації, алгоритми, що використовуються для створення випадкових рівнів, та їх інтеграція у гру. Розглянуто існуючі ігрові рушії для розробки відеоігор, їх переваги та недоліки. Також було досліджено особливості жанру rogue-lite, зокрема такі елементи, як поступове покращення персонажів, випадкові події та високий рівень складності.

Реалізовано та описано всі етапи розробки, користувацький інтерфейс та ігровий процес з доданням відповідних знімків екрану та пояснення коду.

ЗМІСТ

ВСТУП

Актуальність досліджень. За декілька останніх десятиліть індустрія відеоігор дуже швидко розвивається і стала одним з найбільших джерел розваг для людей будь-якого віку. Популярність відеоігор призвела до розвитку нових технологій та платформ, таких як ігрові консолі, мобільні ігри, платформ трансляції відео, які в свою чергу розширили аудиторію зацікавлену у відеоіграх. Крім того, ігри вплинули на освіту та використовувалися як інструмент навчання в класах та інших навчальних закладах. Розвиток ігрової індустрії призвів до створення нових робочих місць і розвитку нових технологій і платформ.

Об'єкт дослідження: Відеогра з процедурною генерацією для ОС Windows

Предмет дослідження: Особливості відеоігор у жанрі rogue-lite. Відеоігри з процедурною генерацією. Ігрові рушії для розробки відеоігор.

Мета дослідження: Метою даної роботи є розробка відеогри з процедурною генерацією рівнів, що відповідає вимогам жанру rogue-lite.

Завдання дослідження:

1. Аналіз існуючих відеоігор та ігрових рушіїв.
2. Здійснити програмну реалізацію алгоритмів процедурної генерації.
3. Провести тестування відеогри, виправлення помилок та підтвердження працездатності.

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД.

1.1 Особливості відеоігор у жанрі rogue-lite

Жанр «rogue» отримав свою назву від класичного підземелля під назвою Rogue, який допоміг популяризувати деякі цікаві та незвичайні ідеї.

Передумови та сюжет гри Rogue, яка вийшла у 1980 році, були простими та мінімальними, як і її графіка, представлена за допомогою символів ASCII: у грі безіменний шукач пригод із символом «@» досліджує лабіринт підземелля в пошуках амулета, зустрічаючи злих монстрів, представлених літерами алфавіту («E» для Emu, «S» для Snake тощо). Якщо персонаж помирав, гравець повинен був почати гру повністю заново, не маючи жодної зброї чи інших покращень, які йому вдалося зібрати по дорозі. [1]

Більше того, після кожної смерті гравця рівні підземелля Rogue випадково генеруються заново, так само як і властивості предметів, таких як зброя, броня, зілля, жезли, посохи та свитки, змушуючи гравців знову вивчати гру кожного разу, коли вони починають нову гру. Популярність Rogue, ймовірно, пояснюється новизною та свіжістю, які створюють ці нові ігрові цикли, тоді як поширення численних похідних ігор можна пояснити відкритим кодом Rogue та текстовими візуалізаціями, що заохочує програмістів зосереджуватись на новому ігровому процесі, а не на новій графіці.

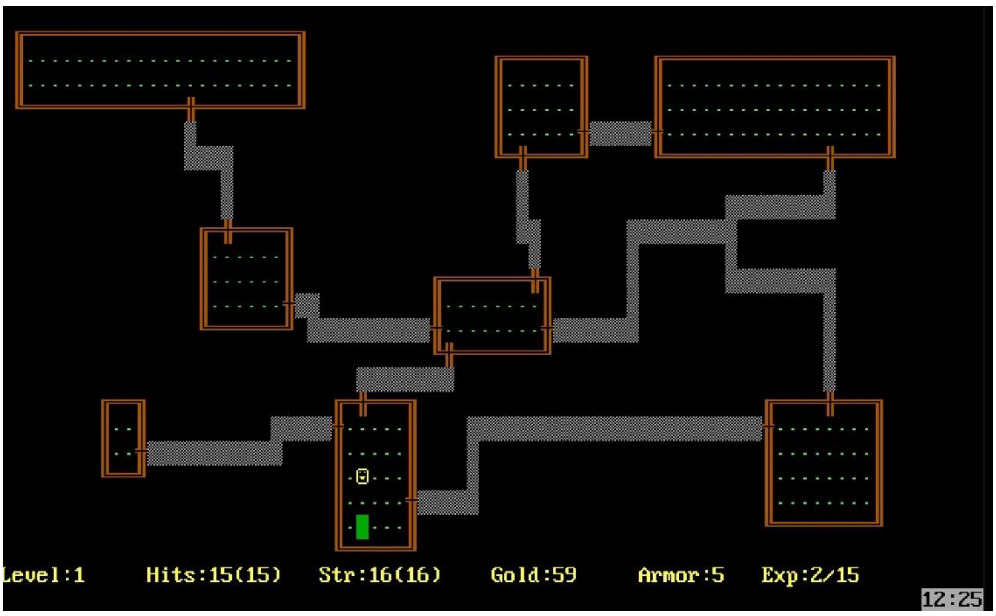


Рисунок 1.1. Кадр з гри Rogue.

Протягом багатьох років ігри запозичували елементи з Rogue, у результаті чого з’явився жанр roguelike. Але незважаючи на те, що roguelike став дещо загальним терміном, конференція з розробки 2008 року створила Берлінську інтерпретацію, повну версію якої доступна на RogueBasin, і визначила roguelike як такі, що мають вісім конкретних принципів дизайну [2]:

- Процедурна генерація - гра повинна мати процедурно згенеровані рівні, щоб кожне проходження було унікальним.
- Permadeath (Одне життя) - якщо гравець вмирає, він починає гру заново.
- Покроковий процес гри - гра повинна бути покроковою, де гравець і вороги діють по черзі.
- Рух на основі сітки - гра повинна бути тайловою, де простір розділений на дискретні клітинки сітки.

- Складність - гра повинна бути достатньо складною, щоб гравцю потрібно було ретельно планувати свої дії.
- Ресурсний менеджмент - гравець повинен управляти обмеженими ресурсами, такими як здоров'я, їжа, зілля тощо.
- Обмежена інформація - гравець не повинен бачити всю карту одразу, а тільки ту частину, яка знаходиться в його полі зору або була відкрита.
- Гравець проти світу - гра являє собою протистояння гравця і світу: немає відносин між монстрами (як ворожнечі або дипломатії).

Щоб гра була кваліфікована як roguelike на основі Берлінської інтерпретації, гра має відповідати всім основним критеріям.

Через суворі правила, викладені в Берлінській інтерпретації, більшість популярних roguelike ігор правильніше називати roguelite іграми. Ігри roguelite використовують деякі, але не всі, елементи дизайну rogue як основу свого ігрового процесу. Permadeath і процедурно згенеровані карти все ще мають вирішальне значення для дизайну roguelite, але багато ігор додали нові оберти до жанру.

1.2 Огляд існуючих відеоігор у жанрі rogue-lite

1.2.1 Enter the Gungeon

Enter the Gungeon - відеогра в жанрі bullet hell з елементами roguelike, розроблена компанією Dodge Roll і випущена компанією Devolver Digital. Гра розповідає про мандрівників які спустилися в Gungeon (від gun — рушниця і dungeon — підземелля) з однією метою, щоб знайти зброю, що вб'є їхнє минуле. [3]

Однак шлях до цілі є важким, і очікується, що гравці повторять спуск кілька разів, щоб досягти своєї мети. Хоча це може вказувати на нудний геймплей, Enter the Gungeon постійно представляє цікаві концепції зброї, які диктують його підхід до ігрового процесу. Від ручного мега бластера до пістолета мікротранзакцій, розробники ретельно створили арсенал зброї гри, щоб включити веселі та цікаві рушниці та віддати шану популярним концептам.

Окрім великої кількості включеної зброї, ігрові персонажі також впливають на геймплей Enter the Gungeon. Хоча загалом є вісім персонажів (дев'ять, якщо врахувати персонажа лише для кооперативу), вони значно відрізняються один від одного. Наприклад, персонаж «Мисливець» має собаку-компаньйона і вимагає від гравців знати скільки шкоди може завдати ворог, тоді як «Куля» перетворює гру на ближній бій. Хоча не всі персонажі розблоковуються з самого початку, чотири основних персонажі досить різноманітні для будь-якого геймера, якому подобається стиль гри.

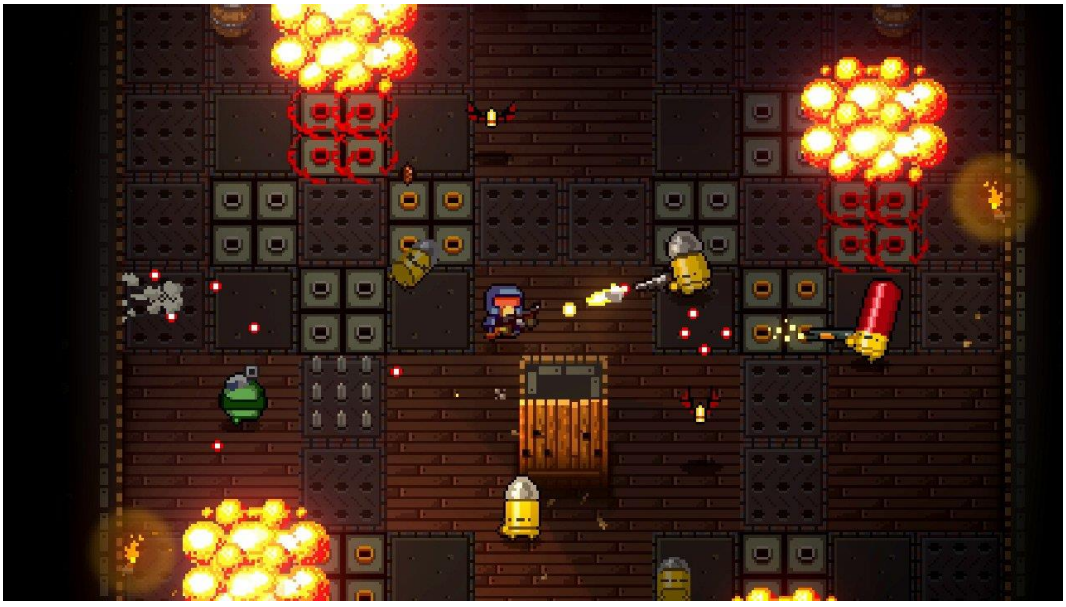


Рисунок 1.2. Процес гри у Enter the Gungeon

Enter the Gungeon підвищує складність і креативність за допомогою функції синергії. По суті, якщо певну зброю чи предмет підбирають разом із іншим предметом, який відповідає рецепту виготовлення, артефакти об’єднуються завдяки чому стають сильніше. Наприклад, якщо гравці отримують приціл і снайперську гвинтівку під час одного заходу, вони розблокують ефект «360 no scope», який надає їм додатковий урон, якщо вони обертаються на 360 градусів перед пострілом. Завдяки цій синергії Enter the Gungeon гарантує, що якщо гравці зустрінуть той самий предмет під час одного заходу, його можна буде використовувати по-різному. [4]

2. Noita — це roguelike гра в жанрах пісочниця і бойовик, розроблена та випущена фінською компанією Nolla Games. Гравці керують відьмою («noita» — це фінське слово, що означає «відьма»), яка може збирати та використовувати заклинання, щоб перемогти ворогів, названих на честь фінських міфологічних істот. [5]

Ви починаєте з двома чарівними паличками, перша стріляє деякими базовими снарядами, а друга запускає бомби, які поповнюються між рівнями. Ви можете знайти нові чарівні палички під час дослідження або купити їх у магазині між областями. Силу чарівних паличок можна змішувати та поєднувати: можна замінити снаряд на вогненну кулю або додати здатність, яка залишає кислотні сліди. У магазині є можливість вибрати між випадковими здібностями, такими як більше здоров’я, здатність заморожувати рідини або імунітет до вогню чи вибухів та багато іншого.

Розробники Noita описують гру як «світ, де кожен піксель фізично моделюється». Вугілля та дерево можна підпалити. Бруд можна розрити або знищити. Такі рідини, як кислота, вода та кров, накопичуються та переливаються, і вони також залишаються на вас, доки ви їх не змієте. Масло робить вас слизьким і легкозаймистим, а кислота з’їдає ваше здоров’я. Вода робить вас несприйнятливими до вогню, а також змиває інші ефекти.

Карта гри складається з різноманітних зон, гравець має змогу відвідати їх на його спуску до боса. Кожна зона має унікальну структуру завдяки процедурній генерації та матеріалам, які можна знайти лише на специфічних зонах.

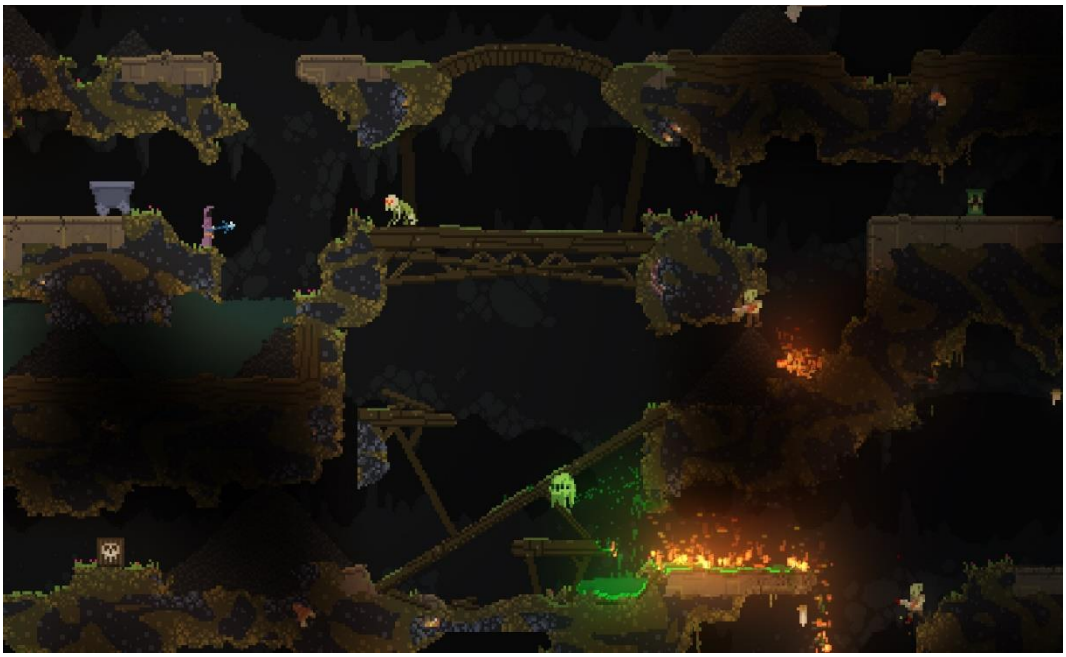


Рисунок 1.3. Перший рівень у грі має багато легкозаймистих матеріалів.

Все це призводить до великого хаосу, оскільки ваші дії мають послідовні, але часто непередбачувані наслідки. Горять легкозаймисті речі, і вогонь поширюється. Якщо вбити над собою монстра, що вивергає кислоту, ви виллєте кислоту на себе, і будете повільно помирати поки не знайдете воду. Якщо кинути бомбу надто близько до пороху, почнеться ланцюгова реакція вибухів, які будуть знищувати все навколо. [6]

3. Slay the Spire — roguelike гра зі створення колод карт, розроблена американською інді-студією Mega Crit і видана Humble Bundle. [7]

Мета гри полягає в тому, щоб пройти кілька рівнів шпиля, кожен рівень має кілька потенційних зіткнень, розподілених у розгалуженій структурі з

персонажем боса в кінці рівня. Зустрічі включають монстрів, які відрізняються за силою; елітні вороги, які пропонують підвищені винагороди; багаття для зцілення або оновлення карт до більш потужних версій; власники магазинів, у яких можна купувати карти, реліквії та зілля, а також прибирати карти з колоди; скрині з випадковою здобиччю; і зустрічі на основі випадкового вибору.

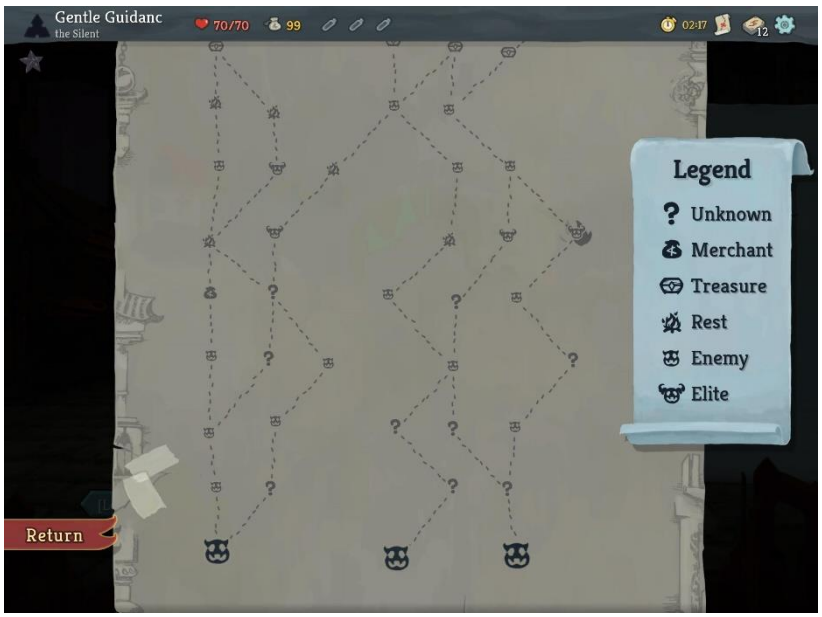


Рисунок 1.4. Карта одного з рівнів

На початку проходження гравець обирає одного з чотирьох попередньо визначених персонажів, який встановлює початкову кількість здоров’я, золота, початкову реліквію, яка забезпечує унікальну здатність для цього персонажа, і початкову колоду карт із базовою атакою та карти захисту, а також картки персонажів.

Бій грається по черзі. Кожен хід гравець отримує нову руку карт і три одиниці енергії. Гравець може зіграти будь-яку комбінацію карт, якщо у нього є достатньо енергії, щоб оплатити вартість кожної карти. Кожен ворог на полі бою буде повідомляти, який хід він зробить: чи атакуватиме та скільки шкоди

завдасть, чи блокуватиме, використає заклинання, щоб покращити себе, або послабити гравця.

Рисунок 1.5. Процес бою у грі

У Slay the Spire є аспекти прогресії. Завершені або невдалі забіги нараховують бали для розблокування нових персонажів або нових реліквій і карт, які будуть доступні для певного персонажа. До 20 рівнів складності підйому розблоковуються з кожним успішно завершеним забігом, кожен з яких додає сукупний негативний ефект, наприклад зниження здоров’я або сильніші атаки ворога.

Ознайомившись з популярними іграми у жанрі roguelite, можна виділити спільні аспекти ігор, а також їх унікальні елементи, які роблять ці ігри цікавими та захоплюючими:

З спільного можна виділити:

- Процедурна генерація контенту. У всіх іграх присутня процедурна генерація рівнів, предметів чи нагород. Випадковість такого підходу роблять кожне проходження цікавим та не схожим на попередні.
- Одне життя. Постійні програші стимулюють гравця вивчати гру та її особливостей зоб здобути кращий результат.

Особливості Enter the Gungeon:

- Здатність об’єднувати предмети для синергії та створення сильнішого предмета.
- Динамічна система бою, яка вимагає від гравця концентрації та швидкої реакції.
- Кожен ворог має унікальну поведінку.

Особливості Noita:

- Великий та відкритий світ, гравцю дозволяють досліджувати та змінювати його.
- Фізична симуляція кожного пікселя у грі, що робить світ динамічним та небезпечним.
- Здатність поєднувати закляття та застосовувати різноманітні модифікатори для них.

Особливості Slay the Spire:

- Створення своєї колоди карт з різноманітними діями та ефектами.
- Покроковий процес гри. Перед кожним кроком гравцю дають достатньо часу щоб продумати стратегію дій.

1.3 Огляд ігрових рушіїв

1. Unreal Engine - один із найпопулярніших ігрових рушіїв, який належить Epic Games. По суті, це мультиплатформенна система розробки ігор, призначена для підприємств будь-якого розміру, яка допомагає використовувати технології реального часу для перетворення ідей у привабливий візуальний вміст.

Рисунок 1.6. Візуальний інтерфейс Unreal Engine

Оригінальна версія була випущена ще в 1998 році, і через 26 років вона продовжує використовуватися в деяких з найбільших ігор. Сильною стороною Unreal Engine є гнучкість його архітектури, тому ігри можна перетворити на дуже унікальний досвід. Однак для цього потрібні кваліфіковані розробники з великим досвідом.

Крім того, Epic Games придбала Quixel, яка володіє величезною бібліотекою «фотограмметрії», 3д моделей з реалістичними текстурами, які можна використовувати для створення анімації та відеоігор. Користувачі

Unreal Engine зможуть безкоштовно використовувати надані інструменти Quixel (Bridge, Mixer) і всі ресурси бібліотеки Quixel Megascans. [8]

Мова програмування: C++, Blueprints

Вартість: Безкоштовно

Сильні сторони: масштабованість, безліч функцій, широкі можливості налаштування, 2D і 3D

Слабкі сторони: великий поріг входу, малочисельна спільнота, вимогливість рушія.

2. CryENGINE — це безкоштовна платформа, на якій ви отримуєте повний вихідний код двигуна та всі функції двигуна. Має чудові варіанти для придбання внутрішньо ігрових активів, які можна знайти на CryEngine Marketplace, що скорочує час виходу на ринок. [9]

Рушій розвивається дуже давно, володіє потужним інструментарієм та підтримкою кросплатформи. CryEngine також пропонує багато безкоштовних навчальних ресурсів - будь то підручники, форуми чи документація, необхідна для початку. Його функції включають високоякісні візуальні ефекти, ефективний набір інструментів, поєднання ШІ (штучного інтелекту) з анімацією тощо.

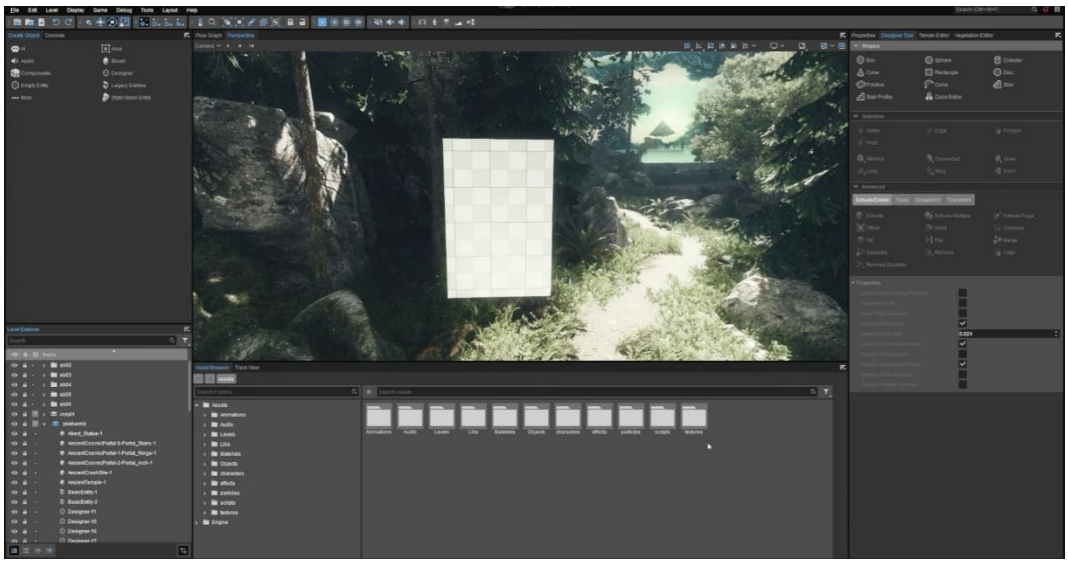


Рисунок 1.7. Візульниій інтерфейс CryEngine

Мова програмування: C#, C++, FlowGraph.

Вартість: Безкоштовно, але коли ви продаєте свій проект, стягується 5% після перших $5 тис. річного доходу.

Сильні сторони: Зручний інтерфейс, приголомшливі візуальні можливості, підтримка VR, гарна документація.

Слабкі сторони: для великих проектів потрібні серйозні знання в C++, дуже малочисельна спільнота.

3. Unity — це популярний ігровий движок для мобільних та настільних платформ, який дозволяє легко створювати інтерактивний 3D контент. Сьогодні його обирають багато великих організацій завдяки його відмінній функціональності, високоякісному контенту та можливості використовувати його для будь-якого типу гри. Він підтримує як 2D, так і 3D контент. [8]

Завдяки своєму універсальному редактору Unity сумісний із Windows, Mac, Linux, IOS, Android, Switch, Xbox, PS4, Tizen та іншими платформами. Зручний інтерфейс полегшує розробку та зменшує потребу в навчанні. Unity

Asset Store курує величезну колекцію інструментів і вмісту, які створюються щодня.

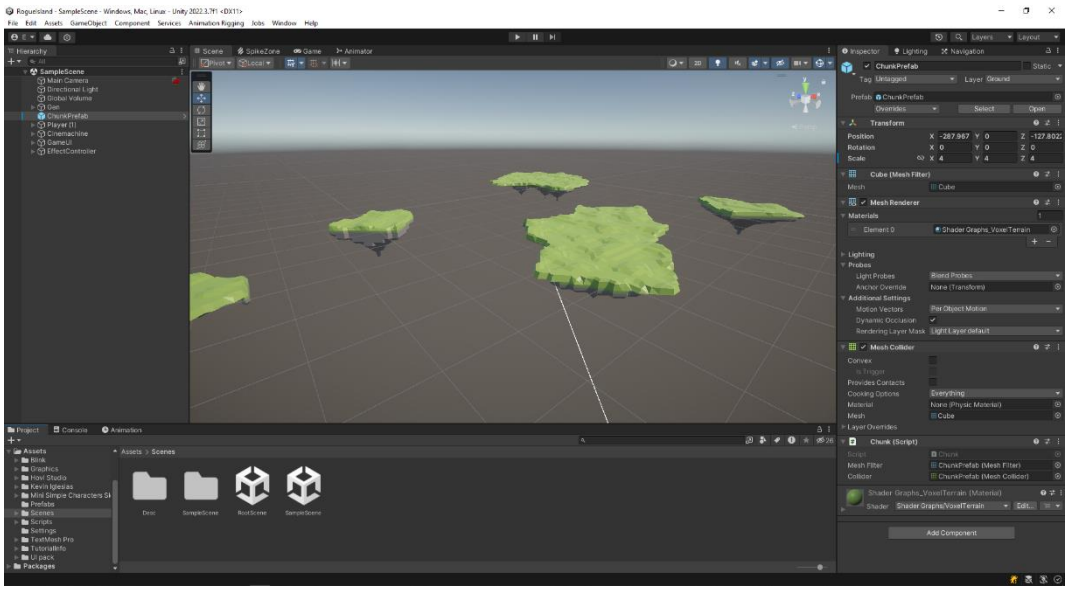


Рисунок 1.8. Графічний інтерфейс Unity

Мова програмування: C#, Visual Scripting (Bolt).

Вартість: Безкоштовно, до $100 тис. прибутку за рік. [10]

Сильні сторони: Зручний інтерфейс, велика спільнота, багато навчальних матеріалів, гарна документація, кросплатформність, гнучкість.

Слабкі сторони: великий розмір, тривалий час запуску редактора, відсутність відкритого коду,

4. Godot є безкоштовним для використання та з відкритим вихідним кодом через ліцензію MIT. Ніяких абонентських плат, ніяких прихованих умов. Рушій Godot чудово підходить для створення як 2D, так і 3D ігор. Також надає великий набір загальних інструментів, тому ви можете зосередитися на створенні своєї гри, не винаходячи колесо заново. [8]

Godot також має сильну спільноту, яка присвячує свої ресурси виправленню помилок та розробці нових функцій. Крім того, Godot має власну вбудовану скриптову мову, GDScript; це високорівнева мова програмування з динамічною типізацією, яка синтаксично схожа на Python.

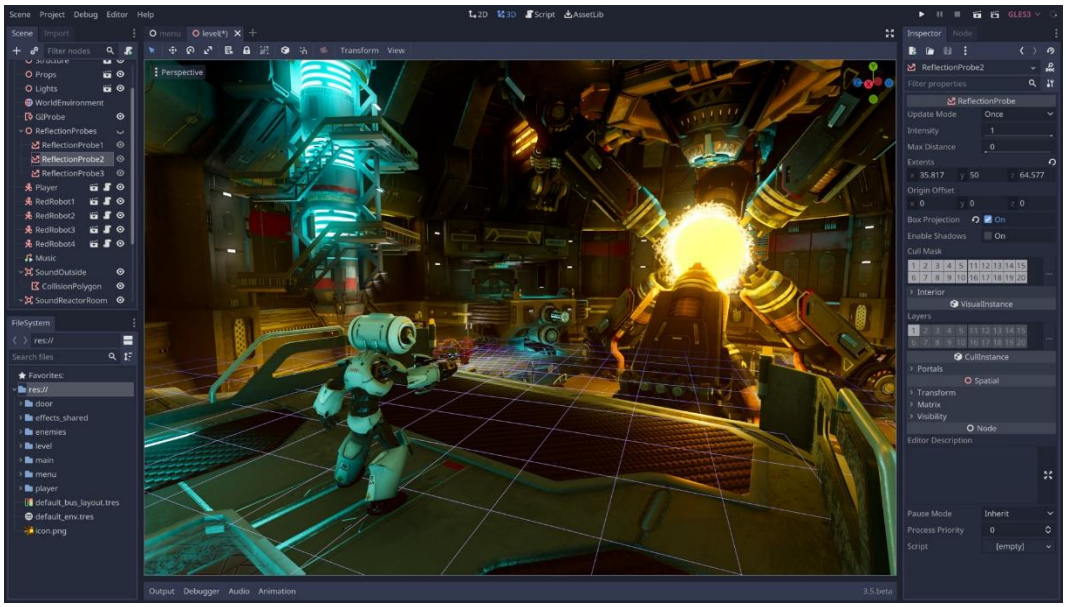


Рисунок 1.9. Графічний інтерфейс Godot

Мова програмування: GDScript, C#.

Вартість: Безкоштовно.

Сильні сторони: Зручний інтерфейс, багато навчальних матеріалів, гарна документація, невибагливий редактор, відкритий вихідний код.

Слабкі сторони: мала кількість функцій, невелика спільнота, є баги.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ВІДЕОГРИ.

2.1 Проектні рішення

Вибір ігрового рушія. Після ознайомлення з популярними рушіями для розробки відеоігор, було вирішено використовувати Unity через наступні переваги:

- Безкоштовне використання
- Велика кількість навчальних матеріалів
- Мова програмування C#
- Платформа Unity Asset Store з великою кількістю безкоштовних ресурсів

Вигляд камери. Вид зверху. Використання виду зверху при розробці roguelike гри має переваги над іншими варіантами розташування камери. При такому розташуванні камери гравець має змогу побачити більшу частину ігрового поля, що полегшує навігацію. Також завдяки цьому гравцю легше оцінювати розташування ворогів та вибирати шляхи для уникнення атак і підготовки для завдання зворотного удару.

Стиль графіки. Для розробки гри було обрано низькополігональний стиль графіки. Такий стиль характеризується відносно малою кількістю полігонів 3D моделей, що дозволяє скоротити час на розробку та надає грі унікального естетичного стилю.

Сетинг. Фентезі. Однією з основних рис фентезі є наявність магії, яка відіграє ключову роль у світах цього жанру, часто з чіткими правилами та законами. Події відбуваються у вигаданих світах де чудеса і вигадки нашого світу є реальністю. Міфологічні істоти, такі як дракони, ельфи, гноми, орки та

інші, часто з'являються у фентезі, наділені особливими здібностями та характерами.

Магія та інші надприродні явища не обов'язково повинні бути основними елементами сюжету, теми чи місця дії, але вони завжди присутні в творі як частина світу. Багато історій у цьому жанрі розгортаються у вигаданих світах, де магія є звичайною справою. Зазвичай, фентезі відрізняється від наукової фантастики та літератури жахів відсутністю (псевдо)наукових і макабричних тем, хоча всі три жанри мають багато спільного.

2.2 Функціональні вимоги

У відеогрі повинні бути реалізовані наступні елементи.

Режим «Меню». У цьому режимі у гравця повинна бути можливість почати\продовжити гру та вийти з гри.

Режим «Ігровий». У цей режим гравець потрапляє після обрання відповідного варіанту у режимі «меню». В цьому режимі відбуваються наступні процеси:

- Процедурна генерація рівня, поділеного на «кімнати» між якими є переходи. У згенерованому рівні завжди повинен бути шлях між початковою кімнатою та кінцем рівня.
- Розміщення гравця у початковій кімнаті.
- Створення ворогів у кімнатах.
- Після перемоги над ворогами, розблокуються переходи між кімнатами. Також у гравця з’являється можливість обрати одне з трьох випадкових посилень персонажа.

- У кінцевій кімнаті повинен бути бос рівня, якого потрібно перемогти для переходу на наступний рівень. Бос повинен бути значно сильнішим за звичайних ворогів та мати унікальні здібності.

Ігровий персонаж гравця повинен мати наступні можливості:

- Переміщення по землі у всіх напрямках.
- Стрибок
- Короткочасне прискорення, яке дає можливість уникнути ворожих атак.
- Атака у напрямку курсора.

2.3 Спосіб процедурної генерації рівня

Ідея процедурної генерації полягає в тому, щоб використовувати один або кілька алгоритмів для створення вмісту. Існує багато таких алгоритмів, які підходять для різних видів ігор. Ці алгоритми приймають певні вхідні дані, якими може бути випадкове число або набір параметрів (наприклад, складність), щоб створити рівень.

Сама генерація виконується за допомогою набору правил і функцій, які визначають, як буде створено рівень і як повинен виглядати бажаний рівень. В нашому випадку було вирішено що ігровий рівень повинене складатися з островів, які будуть з’єднані між собою переходами.

2.3.1 Генерація кімнат рівня

Для створення рівня розділимо мапу на окремі острови за допомогою модифікованого алгоритму створення діаграми вороного. У математиці діаграма Вороного — це розбиття площини на області, близькі до кожного з

даного набору об’єктів. Його також можна класифікувати як тесселяцію (заповнення простору однаковими формами). У найпростішому випадку ці об’єкти являють собою лише скінченну кількість точок на площині (так звані зерна або генератори). [11]

Для кожного зерна існує відповідна область, яка називається коміркою Вороного, що складається з усіх точок площини, ближчих до цього зерна, ніж до будь-якої іншої. Границі на діаграмі Вороного є всіма точками на площині, які рівновіддалені від двох найближчих вершин.

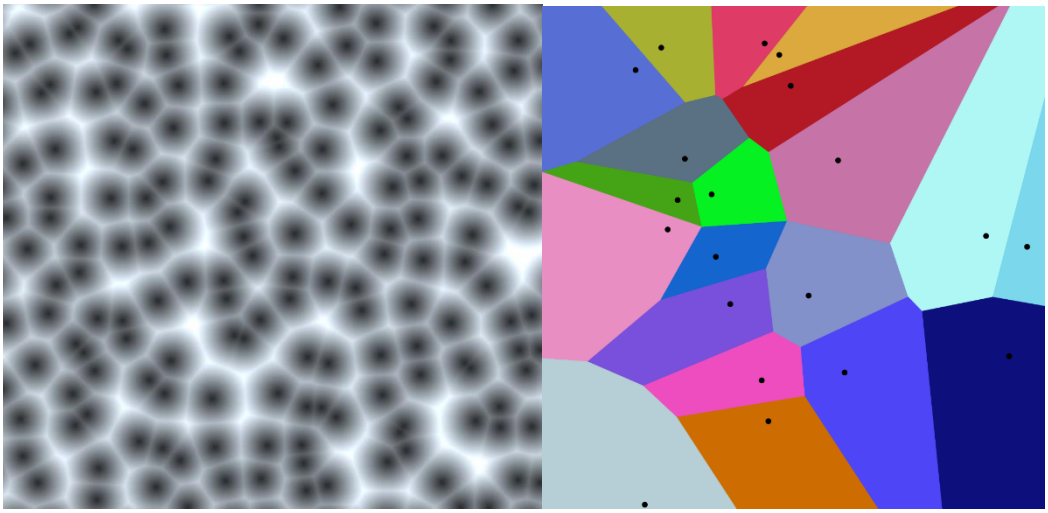


Рисунок 2.1. Ліворуч-шум Вороного, праворуч – діаграма вороного.

Для реалізації шуму Вороного потрібно розбити простір на сітку з клітинами. Після розбиття на сітку, всередині кожної клітини потрібно згенерувати випадкову позицію та обчислити відстань від точки до вхідного значення (точку яку ми перевіряємо). Також потрібно обчислювати відстані до всіх прилеглих точок на сітці з розмірами 3x3, та повернути найменшу відстань. [12]

Отримаємо сітку де кожна клітина якої має обчислену найближчу до неї вершину або зерно. Ці клітини можна вважати за одиницю площини що

належить до певної кімнати, таким чином можна розділити простір рівня на потрібні нам острови.

2.3.2 Створення мапи висот

Для генерації графічної моделі такого острова потрібно додатково мати дані про висоту та заповнення. Для висоти можна задати константне значення, але в такому випадку острів вийде плоским, щоб цього уникнути можна застосувати карту висот та алгоритм генерації шумів.

Існують різні типи та алгоритми, які генерують шум. Шум випадкового значення є найпростішим. Він створює сітку (у цьому випадку 2d), де кожна точка має випадкове плаваюче значення від 0 до 1. Проблема з використанням шуму випадкового значення полягає в тому, що, на відміну від природи, він не має послідовності, тобто кожна точка не залежить від інших. Щоб отримати більш природний рельєф, довільну сітку потрібно згладити. Такого ефекту можна досягти за допомогою лінійної інтерполяції, але результат буде далеким від ідеалу.

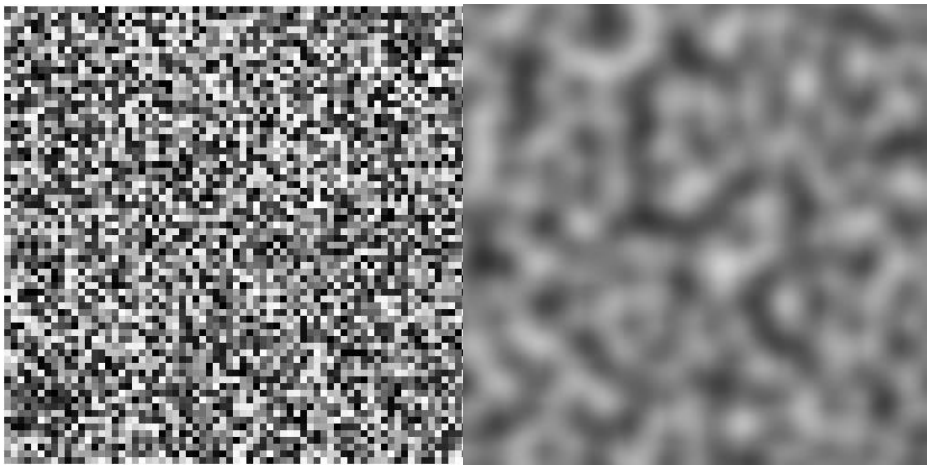


Рисунок 2.2. Візульне зображення шумів. Ліворуч – шум випадкового значення, праворуч – шум Перліна.

Шум Перліна (Perlin noise) – це тип шуму, який був розроблений у 1983 році Кеном Перліном з Mathematical Applications Group, inc, для безпосереднього використання в анімаційному науково-фантастичний фільмі Disney – Tron. Він був офіційно описаний у статті для SIGGRAPH у 1985 році під назвою «Синтезатор зображень». За цю роботу Кен Перлін отримав премію «Оскар» за технічні досягнення від Академії кінематографічних мистецтв і науки за внесок у CGI (комп'ютерне зображення)[13].

Перлін здійснив цю розробку, спонуканий своїм дискомфортом від результатів зображень CGI свого часу, оскільки вони були занадто грубими та неприродними. Створений ним алгоритм дозволяє створювати більш природні текстури для поверхонь, створених комп'ютерами. Perlin Noise надає набір псевдовипадкових значень із постійними відмінностями між ними, і з ними будує карту висот. Хоча його найбільш типові реалізації відповідають 2 або 3-вимірним просторам, цей алгоритм застосовується до n вимірів.

Процес генерації шуму Перліна:

2. Щоб створити двовимірний шум розміром ширина * висота, нам спочатку потрібно створити сітку такого ж розміру. Для цього прикладу припустимо, що ширина та висота рівні, тому створена сітка шуму має форму квадрата. Кожне значення в сітці має координати x і y. Під час роботи з шумом або текстурами зазвичай використовують u і v замість x і y, тому ми будемо використовувати цей тип позначення.

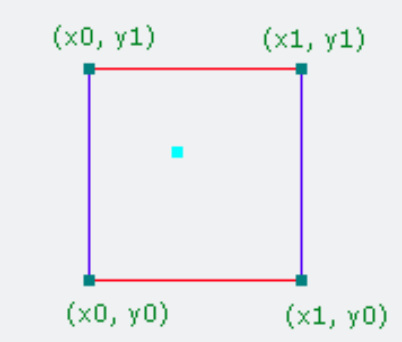


Рисунок 2.3. Клітина сітки координат з центром (x, y)

2. Тепер сітку можна розділити на менші підсітки, щоб масштабувати шум. Коефіцієнт ділення будемо називати масштабним коефіцієнтом шуму.

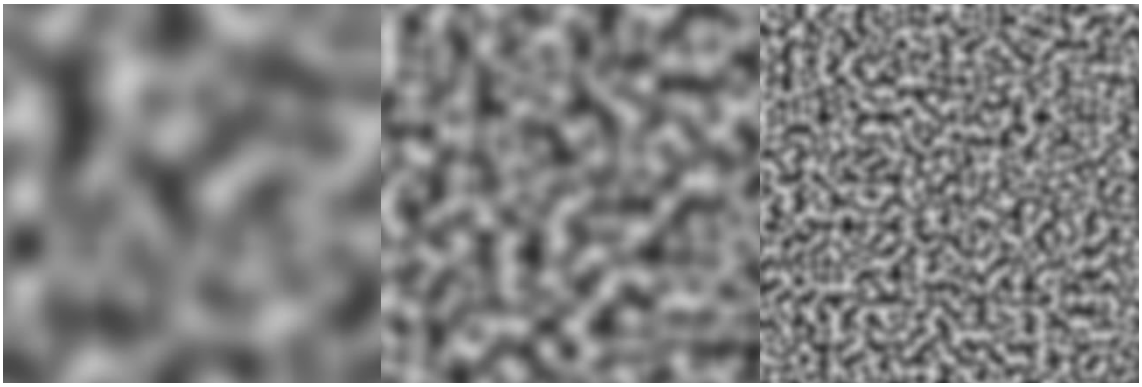


Рисунок 2.4. Вплив розміру клітин на результат. Розмір клітин для кожного зображення, починаючи зліва – 64, 32, 16

3. Використовуємо псевдовипадкову генерацію для створення векторів градієнта, спрямованих убік від квадрата сітки.

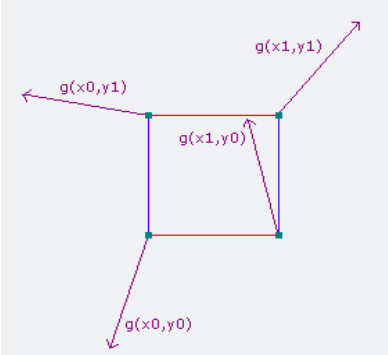


Рисунок 2.5. Випадкові градієнтні вектори

4. Обчислюємо скалярний добуток між вектором градієнта та вектором, що вказує на точку, у якій обчислюється значення шуму. Робимо це для всіх чотирьох вершин. Результатом є чотири скалярних добутку *d1, d2, d3, d4*

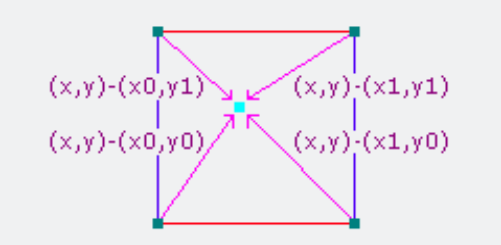


Рисунок 2.6. Відстань від кутів клітини до точки координат

5. Тепер обчислимо значення шуму в пікселі з координатами u і v:

x1 = lerp(d1, d2, u)

x2 = lerp(d3, d4, u)

noiseValue = lerp(x1, x2, v)

6. Лінійна інтерполяція дешева, однак вона неприродна, і результати будуть мати різкі переходи. Щоб зробити щойно згенерований шум кращим, ми можемо застосувати функцію полегшення. Він зробить перехід між плитками більш плавним і природним. Краща крива послаблення для цього типу шуму становить $6t^5 - 15t^4 + 10t^3$, як визначено Кеном Перліном [14]. Криву наносять на координати u і v.

2.3.3 Алгоритм створення тривимірної моделі рельєфу

Багато систем рельєфу використовують систему карти висот для визначення базової моделі рельєфу [15]. Для цього використовується двовимірна сітка зі значеннями, які визначають висоту місцевості в цій точці. Генерація сітки зазвичай досить швидка, оскільки сітка — це просто плоска сітка квадратів, де кожна вершина еквівалентна одному елементу даних на карті висот, з якої береться вертикальне положення вершини. Прикладами

цього є вбудований компонент Terrain Unity3D і функція Landscape Unreal Engine 4.

Однак такий підхід має свої недоліки, оскільки ми маємо справу з тривимірним світом, а мапа висот може мати лише одне значення для кожного вертикального стовпця простору. Це означає, що місцевість, заснована виключно на карті висот, не може підтримувати більш складні особливості місцевості, такі як системи печер, виступи та арки. Для цього нам потрібні тривимірні дані.

Воксель — це елемент, який представляє об'єм у тривимірному векторному просторі. Ландшафт можна визначити, розділивши простір на рівномірну тривимірну сітку вокселів, вирівняну по осі. Кожен воксель може містити значення, які описують рельєф у заданому обсязі, наприклад, значення може описувати тип рельєфу, який утворює більшість у цьому просторі. Для нашої мети нам також потрібне значення, яке відображає, скільки простору фактично займає рельєф. Це значення дозволить вирішити, де розташована поверхня між землею та повітрям, яку необхідно відобразити. [16]

За допомогою вокселів можна побудувати дуже детальну місцевість, використовуючи невеликі будівельні блоки. Це дозволяє запускати більш розширене моделювання на даних рельєфу, використовуючи складніші алгоритми генерації світу, яким більше не потрібно просто визначати висоту рельєфу та дозволяє змінювати рельєф на основі введення користувача з набагато більшою свободою. Ця свобода має свою ціну. Зберігання цих даних у тривимірній сітці значно збільшує споживання пам'яті, ніж двовимірні рішення.

Іншим аспектом, який значно ускладнюється, є візуалізація. Greeff [16] написав про деякі методи, які можна використовувати для відтворення об'ємних воксельних даних:

1. Об’ємне кидання променів (Volume ray casting) можна використовувати для створення променів на кожен піксель, щоб визначити, який воксель потрапив під промінь.
2. Розбризкування (Splatting) можна використовувати для зворотного проектування кожного вокселя на площину[10]. Ці спроектовані вокселі на площині потім об’єднуються, щоб сформувати відтворене зображення
3. Вилучення полігональної сітки дозволяє взяти набір вокселів і обробити його в сітку з трикутників. Цю сітку можна візуалізувати за допомогою стандартних методів, для яких створено більшість відеокарт.

У своїй роботі Greff також говорить про алгоритм вилучення воксельної сітки маршируючих кубів, коли зміни в окремому вокселі впливають лише на невелику навколишню область, тобто можна вносити локалізовані редагування, які не вимагають повторного вилучення всього набору даних.

Алгоритм маршируючих кубів (Marching Cubes), розроблений Лоренсеном та Кляйном [17], дозволяє створити тривимірну багатокутну сітку з сітки вокселів. Його автори описали це як алгоритм, який досліджує куб з воксельним елементом у кожному з його кутів. Для кожного кута визначається, чи знаходиться кут всередині чи зовні об’єкта. Оскільки в кубі вісім кутів і кожен може мати двійкове значення, загалом існує 256 можливих комбінацій. Для кожної з цих комбінацій існує заздалегідь визначений набір трикутників, які відображаються всередині цього куба. Для кожного з цих трикутників усі три кути трикутника знаходяться на окремих ребрах куба, тобто набір трикутників можна визначити масивом індексів ребер куба. Коли трикутники для одного куба побудовані, алгоритм переходить до наступного куба.

Оригінальна стаття Лоренсена та Клайна [17] представила 15 оригінальних конфігурацій (рис. 2.7), з яких інші 241 випадки можна було

переставити, використовуючи симетрію. Коли значення кутових вокселів куба інвертуються, структура трикутника всередині куба залишається незмінною, трикутники потрібно просто перевернути. Наприклад, інвертуючи значення вокселів конфігурації №8 (рис. 2.7), ми отримуємо куб, верхні вокселі якого розташовані всередині поверхні, а нижні — зовні, тобто поверхня залишається в тому самому положенні, але повинна дивитися вниз. . Це вдвічі зменшує необхідні тріангуляції. Решту комбінацій можна звести до 15 оригінальних випадків, використовуючи властивість обертальної симетрії кубів, тобто якщо дану комбінацію можна обертати для досягнення одного з оригінального випадка, тоді необхідну тріангульовану сітку можна досягти, змінивши обертання на сітці класу еквівалентності.

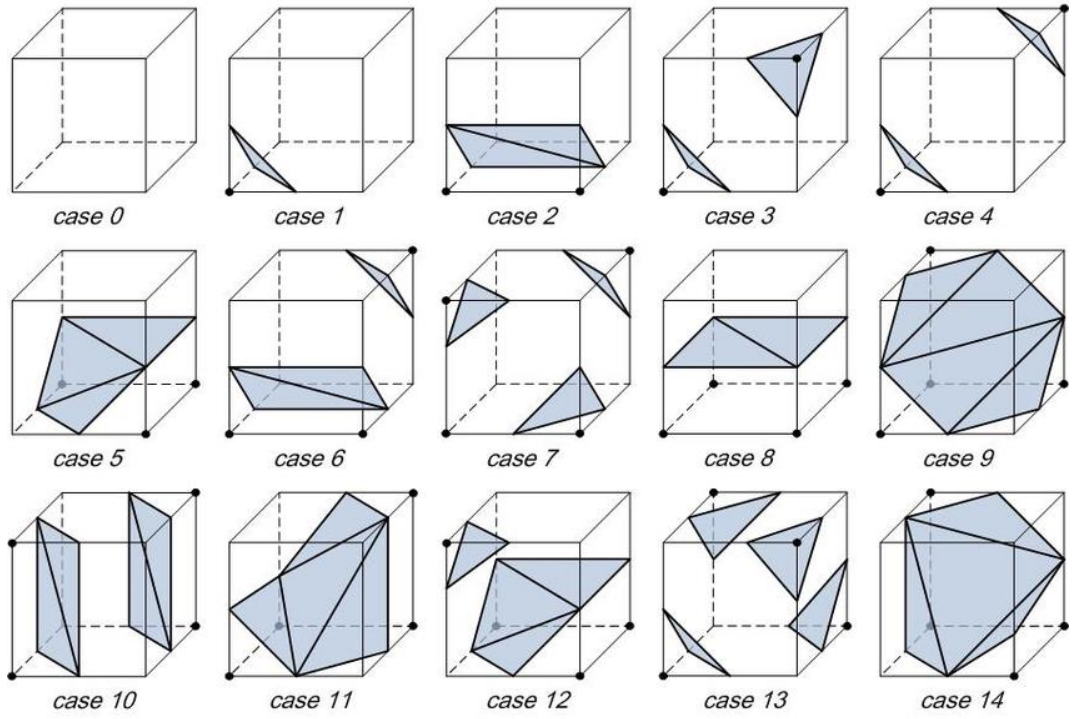


Рисунок 2.7. 15 оригінальних конфігурацій алгоритму крокуючих кубів. Чорна крапка позначає воксель всередині поверхні, а відсутність точки у куті позначає воксель поза поверхнею.

Можливість попереднього визначення трикутників для всіх комбінацій дає алгоритму крокуючих кубів значну перевагу в продуктивності, оскільки не потрібно виконувати складну тріангуляцію кожного циклу. Натомість 256 можливих комбінацій використовуються для створення індексу списків. Ці списки містять ребра куба, які потрібно з’єднати, щоб утворити трикутники необхідні для полігональної сітки.

Оскільки алгоритм крокуючих кубів переглядає лише вокселі у восьми кутах куба, редагування одного вокселя в наборі даних впливає лише на сітчасту структуру восьми кубів, які оточують воксель. Це означає, що редагування вокселів можна вносити, не вимагаючи повторного створення сітки всієї місцевості.

2.4 Проектування штучного інтелекту ворогів

Ігровий штучний інтелект (ШІ) — це набір програмних методик, що застосовуються у відеоіграх для створення ілюзії інтелектуальної поведінки персонажів, керованих комп'ютером. Окрім традиційних методів, він включає алгоритми з теорії керування, робототехніки, комп'ютерної графіки та інформатики загалом. [18]

Термін "ігровий ШІ" охоплює широкий набір алгоритмів, які використовуються у відеоіграх. Однак ці методики часто не відповідають стандартним критеріям "справжнього ШІ", оскільки не обов'язково сприяють комп'ютерному навчанню. Замість цього вони являють собою автоматизовані обчислення або заздалегідь визначений і обмежений набір відповідей на заздалегідь визначений і обмежений набір вхідних даних.

Реалізація штучного інтелекту (ШІ) значно впливає на ігровий процес, системні вимоги та бюджет гри. Розробники прагнуть збалансувати ці вимоги, створюючи цікавий і невибагливий ШІ за невеликі кошти. Тому підхід до

ігрового ШІ відрізняється від традиційного, широко застосовуючи спрощення, обмани й емуляції. Наприклад, в RPG-іграх персонажі, керовані ШІ, мають спрощені моделі поведінки для оптимізації роботи гри. Вони можуть використовувати прості алгоритми для пошуку шляху або ухвалення рішень у бою, щоб не робити складні та вибагливі розрахунки. Водночас це забезпечує видимість складної поведінки, створюючи відчуття глибокого занурення у світ гри.

Логіку роботи простого штучного інтелекту можна зобразити блок схемою, яка зображена на рис.

Рисунок 2.8. Блок-схема роботи простого штучного інтелекту ворогів.

Для роботи штучного інтелекту майбутньої гри треба виділити наступні зони:

- Зона пошуку цілей
- Зона початку атаки цілей
- Зона нанесення шкоди

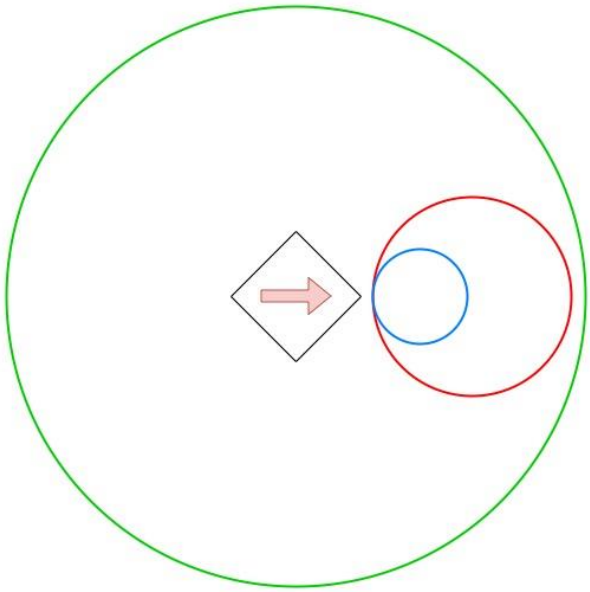


Рисунок 2.9. Приклад розподілу зон ворога, де: ромб з стрілкою – ворог та його напрямок руху, зелений – зона пошуку, червоний – зона нанесення шкоди, синій – зона початку атаки.

Розділення зони атаки на дві (початок атаки, нанесення шкоди) дозволяє створити більш цікаві моделі поведінки для ворогів та дасть гравцю змогу не отримати шкоду, ухилившись від атаки. Таким чином щоб ворог зміг атакувати гравця, він повинен наблизитись так, щоб гравець був у зоні початку атаки, а для того щоб нанести шкоду гравець повинен залишатись у зоні нанесення шкоди.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1 Візуальна складова

Графічні моделі та текстури ігрового персонажа, ворогів та ігрових об’єктів були взяті з онлайн-платформи Unity Asset Store. Для розробки були використані наступні набори: «Mini Simple Characters | Skeleton» - модель ворога (скелет), «FREE Stylized Bear - RPG Forest Animal» - модель ворога (медвідь), «Magic Effects FREE» - візуальні ефекти та текстури для ефектів, «Sleek essential UI pack» - текстури для графічного інтерфейсу.

Рисунок 3.1 – Графічна модель скелета

Оскільки для того щоб зробити бій з босом рівня цікавим потрібно використовувати унікальні механіки та ефекти, було створено унікальний візуальний ефект для бою з босом. Зазначений ефект створений за допомогою засобу «Visual Effect Graph» який є потужним інструментом для створення візуальних ефектів у реальному часі та використовує графічний редактор вузлів (node-based editor).

Він дозволяє художникам і розробникам створювати складні ефекти, такі як дим, вогонь, вибухи, магічні заклинання і багато інших, без необхідності написання коду.

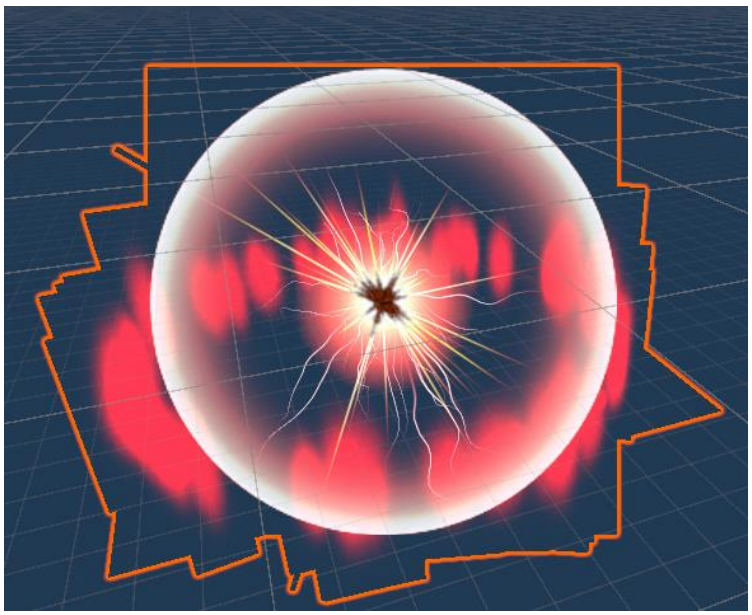
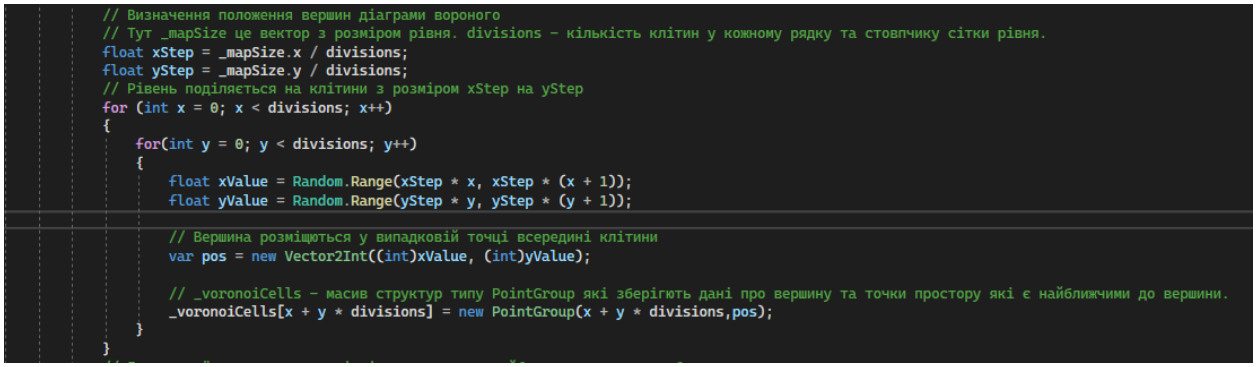


Рисунок 3.2 Візуальний ефект вибуху.

3.2 Генерація рівнів

1. Першим етапом генерації рівня є розділення простору рівня на відокремлені частини за алгоритмом створення діаграми Вороного. Реалізація цього алгоритму складається з двох частин. Першою частиною є визначення вершин діаграми, для цього умовний простір поділяється на клітини, розмір яких розраховується згідно кількості поділів простору, у кожній такій клітині визначається вершина з випадковою позицією (рис. 3.3).



```
// Визначення положення вершин діаграми вороного
// Тут _mapSize це вектор з розміром рівня. divisions - кількість клітин у кожному рядку та стовпчику сітки рівня.
float xStep = _mapSize.x / divisions;
float yStep = _mapSize.y / divisions;
// Рівень поділяється на клітини з розміром xStep на yStep
for (int x = 0; x < divisions; x++)
{
    for(int y = 0; y < divisions; y++)
    {
        float xValue = Random.Range(xStep * x, xStep * (x + 1));
        float yValue = Random.Range(yStep * y, yStep * (y + 1));

        // Вершина розміщються у випадковій точці всередині клітини
        var pos = new Vector2Int((int)xValue, (int)yValue);

        // _voronoiCells - масив структур типу PointGroup які зберігють дані про вершину та точки простору які є найближчими до вершини.
        _voronoiCells[x + y * divisions] = new PointGroup(x + y * divisions,pos);
    }
}
```

Рисунок 3.3. Реалізація першої частини алгоритму з створення вершин з випадковими позиціями.

Другим кроком є проходження по усім точкам простору для знаходження найближчої вершини, та збереження інформації про відношення до вершини для подальшого використання (рис 3.4).

```
// Для кожної точки у просторі рівня знаходимо найближчу вершину та зберегаємо
for (int x = 0; x < _mapSize.x; x++)
{
    for (int y = 0; y < _mapSize.y; y++)
    {
        int minIndex = 0;
        float min = float.MaxValue;
        var point = new Vector2Int(x, y);
        // Знаходмо координати клітини до якої відноситься точка
        var gridX = (int)((float)x / xStep);
        var gridY = (int)((float)y / yStep);

        // Перевіряємо яка з 9 вершин навколо точки є найближчою.
        foreach (var p in GetPointsAround(gridX,gridY))
        {
            var index = GetIndex(p.x, p.y);
            if (index < 0 || index >= _seedCount) continue;

            float dist = (point - _voronoiCells[index].Center).sqrMagnitude;
            if (min > dist) // Знаходження мінімальної відстані до вершини
            {
                min = dist;
                minIndex = index;
            }
        }
        _voronoiCells[minIndex].Add(point);
        _pointToIndex.Add(point, minIndex);
    }
}
```

Рисунок 3.4. Реалізація другої частини алгоритму з визначення найближчих вершин для точок простору рівня.

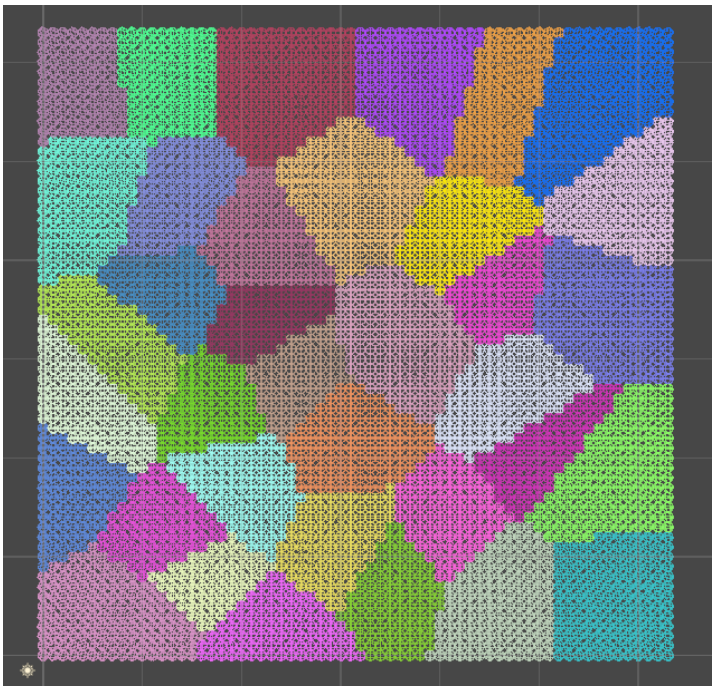


Рисунок 3.5 Результат роботи коду розділення простору на рівні з розміром 64x64, 6 поділів.

Отриманий результат можна використовувати для створення островів, але в такому вигляді вони будуть однотипні (рис 3.5). Для покращення зовнішнього вигляду островів згрупуємо ці частини між собою використовуючи алгоритм схожий на той що описано вище, але замість окремих точок простору будуть виступати вершини діаграми вороного. Границі карти зараз плоскі щоб цього уникнути додатково приберемо вершини, точки яких торкаються зовнішньої границі рівня.

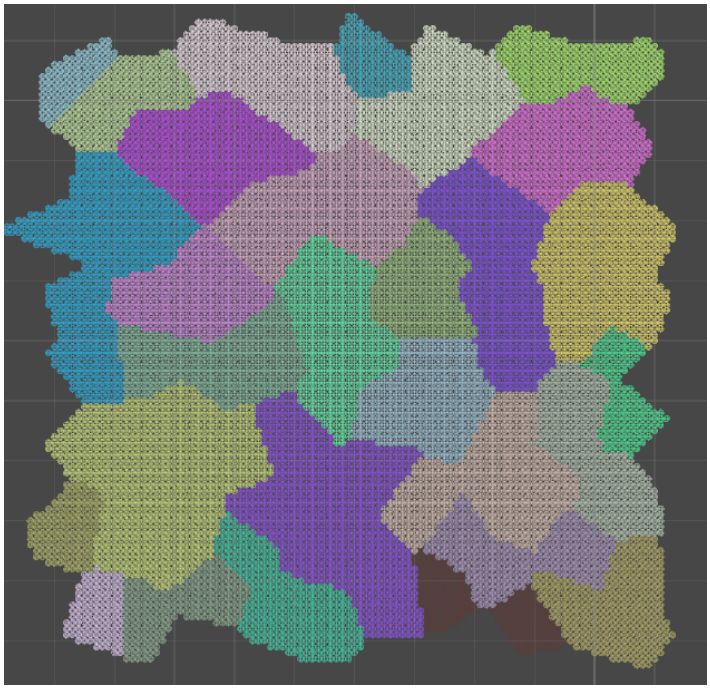


Рисунок 3.6 Результат групування вершин та видалення границь.

Групування вимагає збільшення мапи щоб досягти прийнятних результатів, що збільшує складність розрахунків додатково до того що додається розрахунок групування. Попри це, завдяки оптимізаціям та використанню кількох ядер процесора, різницю у швидкості генерації неможливо помітити не скориставшись додатковими інструментами.

2. Наступним етапом є застосування шумів для створення мапи висот та рельєфу. Також на цьому етапі згенеруємо нижню частину островів, хоча це і не обов’язково, але додає більш реалістичного вигляду. Створення мапи висот є нескладним, але для створення полігональної сітки тривимірної моделі потрібно зберегти дані у тривимірній сітці вокселів, що значно ускладнює розрахунки. Зменшити складність розрахунків для наступного етапу можна розділивши мапу на рівні сегменти (або «чанки») на цьому етапі.

Чанк (від англійського "chunk") — це концепція в розробці ігор, яка використовується для поділу великої ігрової мапи на менші, керовані частини

або сегменти. Такий підхід особливо корисний у процедурній генерації мап, де великі ігрові світи створюються динамічно під час гри.

Переваги використання чанків:

- Покращена продуктивність: Локальна генерація і обробка чанків значно покращує продуктивність гри.
- Ефективне управління пам'яттю: Завантаження тільки необхідних чанків зменшує використання пам'яті.
- Можливість створення великих світів: Розбиття мапи на чанки дозволяє створювати великі ігрові світи, які можуть безперервно розширюватися.

Таким чином, використання чанків є ключовою технікою для процедурної генерації мап, забезпечуючи ефективне управління ресурсами і підтримку великих ігрових світів.

Поділ на чанки виглядає наступним чином (рис 3.7): ітерація з визначенням простору який належить кожному фрагменту, створення задач з заповнення верхнього нижнього чанка, очікування завершення всіх задач та збереження даних. Результати виконання цього алгоритму можна побачити на рисунку 3.8.

```
// Розбиття острова на чанки. _chunkSize - розмір чанка. xSize,ySize - розміри острова які кратні розміру чанка.
for (int chunkXOffset = 0; chunkXOffset < xSize; chunkXOffset += _chunkSize)
{
    for (int chunkYOffset = 0; chunkYOffset < ySize; chunkYOffset += _chunkSize)
    {
        int chunkIndexTop = chunkIndex++;
        int chunkIndexBottom = chunkIndex++;
        var chunkTopOffset = new Vector3Int(chunkXOffset, 0, chunkYOffset);
        var chunkBottonOffset = new Vector3Int(chunkXOffset, -_chunkSize, chunkYOffset);
        int groupIndex = i;
        // Створення задач для паралеьного розрахунку заповнення чанку
        tasks.Add(Task.Run(() =>
        CalculateChunkData(chunkIndexTop, groupData, chunkTopOffset, groupIndex))
            );
        tasks.Add(Task.Run(() =>
        CalculateChunkData(chunkIndexBottom, field, chunkBottonOffset, groupIndex))
            );
    }
}
// Очікування виконання розрахунку на всіх чанках
Task.WaitAll(tasks.ToArray());

foreach (var task in tasks)
{
    (int chunkI, ChunkData chunkData,int values) = task.Result;
    _chunks.Add(chunkI, chunkData);
    if (values == 0) continue;

    // Збереження результатів до масиву
    _chunkGroups[chunkData.Group].Chunks.Add(chunkI);
}
tasks.Clear();
```

Рисунок 3.7. Алгоритм поділення островів на чанки.

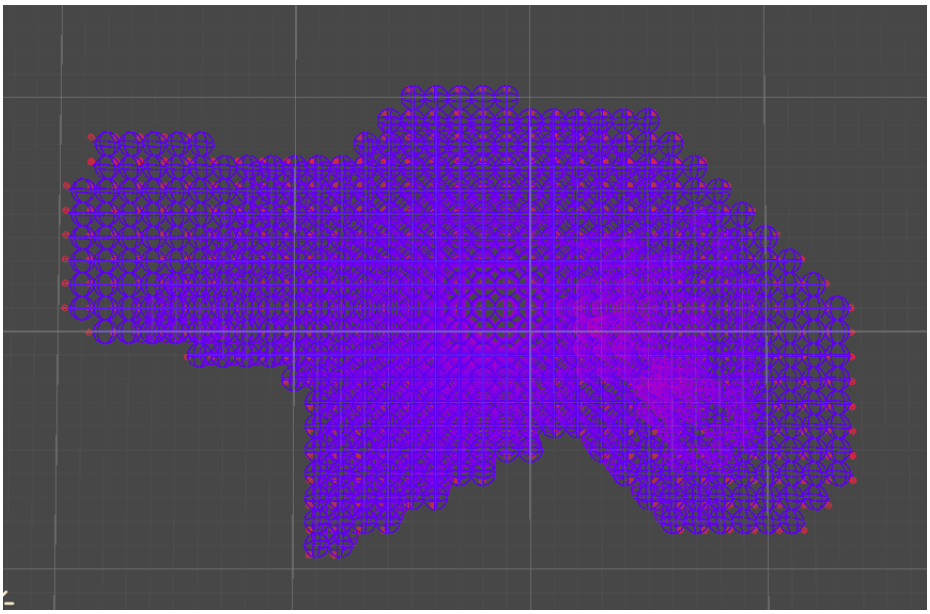


Рисунок 3.8 Результат генерації чанків одного з островів, вид зверху.

3. Наступним етапом є генерація полігональної сітки тривимірної моделі. Розглянутий раніше алгоритм крокуючих кубів добре підходить для цієї цілі,

але при реалізації на CPU час виконання занадто великий. Вирішення цієї проблеми можливе перенесенням виконання алгоритму на GPU, що дозволяє використовувати значно більшу кількість ядер графічного процесора.

Обчислення загального призначення на графічному процесорі (GPGPU) — це відносно нова техніка, яка дозволяє загальним програмам, які зазвичай виконуються на центральному процесорі, працювати на графічному процесорі. Завдяки введенню додаткового програмованого етапу шейдера та більш точної арифметики тепер можна використовувати потокову обробку даних, не пов’язаних із графікою.

Обчислювальний шейдер — це програмований етап шейдера, призначений для обчислень загального призначення на графічній карті. Він визначає програми, які називаються ядрами, які викликаються в потоці. Ці ядра мають бути відносно малими та незалежними для хорошого розпаралелювання. Метою використання обчислювальних шейдерів є можливість використовувати потужну паралельну природу графічного процесора, не маючи глибоких знань про конвеєр візуалізації. Це дозволяє спільно використовувати пам’ять і синхронізувати потоки, що було неможливо з іншими програмованими етапами шейдерів (вершини, геометрія, піксельні шейдери).

Використовуючи багато ядер GPU, можна пришвидшити час виконання шляхом розпаралелювання. Група потоків — це набір потоків, які працюють на одному ядрі GPU. Якщо існує дві групи потоків, кожна з них може працювати на окремому ядрі, і таким чином досягається паралельна робота. Під час написання обчислювального шейдера можна вказати, скільки груп потоків створити та скільки потоків у групі потрібно запустити. Як кількість груп потоків, так і потоків у групі вказується як тривимірний масив, а не просто як лінійний масив. Цей вид індексування корисний як геометрична візуалізація та стане дуже зручним при реалізації алгоритму, оскільки можна

зіставити ідентифікатор потоку безпосередньо з локальною позицією вокселів. Потім це можна було б легко перетворити на позицію світового простору, що є дуже корисним. Наприклад, якщо є об’єм із 4x4x10 вокселів, тоді можна створити 10 груп потоків із потоками 4x4 всередині групи. Тоді ідентифікатор потоку буде в діапазоні від (0,0,0) до (3,3,9). Ці ідентифікатори безпосередньо відображаються на унікальний воксель у вказаному обсязі. Ще одна чудова функція полягає в тому, що він може безпосередньо індексувати 3D-текстуру за допомогою цього 3-компонентного індексу, тому не потрібно виконувати перетворення в індекс одновимірного масиву.

Оскільки у вокселі є 8 кутів, можна створити максимум 28 = 256 можливих комбінацій знаків у кутах. Використовуючи один байт і де кожному куту призначається один біт і встановлюється значення 1, якщо воно має додатне значення, інакше 0. Це створює номер конфігурації в діапазоні від 0 до 255. Приклад показано на рисунку 3.9, де кути 3, 4 і 5 знаходяться в тому положенні, який створює конфігурацію 56. [19]

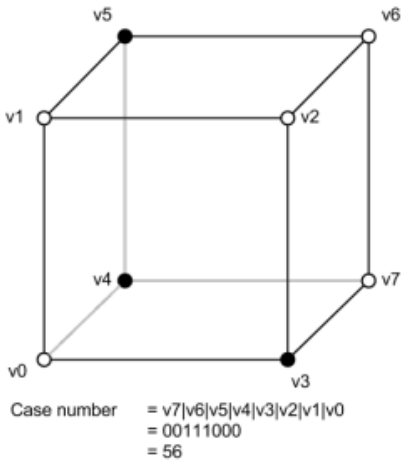


Рисунок 3.9. Воксельні вершини. Чорні кути мають додатні значення (внутрішній об’єм), білі кути мають від’ємні значення (зовнішній об’єм).

Номер кофігурації є індексом для двох таблицях пошуку. Перша таблиця пошуку приймає як вхід номер конфігурації та повертає кількість багатокутників, які мають бути згенеровані, а друга таблиця повертає список того, як трикутники з’єднані між краями вокселя. Правила іменування ребер показано на рисунку 3.10.

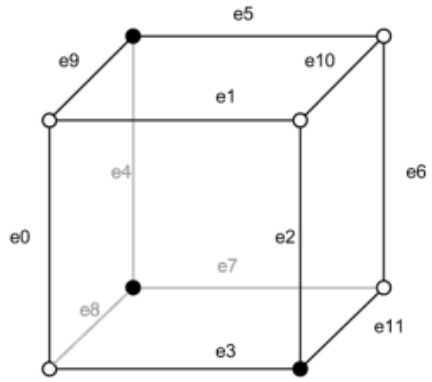


Рисунок 3.10. Найменування ребер вокселя

Наприклад, якщо номер випадку 56, то перша таблиця пошуку повертає 3, тобто кількість трикутників, які потрібно побудувати. Виконання пошуку в другій таблиці виведе такі дані.

```
edgeConnections [56][0] : 7 9 5
edgeConnections [56][1] : 7 8 9
edgeConnections [56][2] : 3 11 2
```

Це вся інформація, необхідна для побудови трикутників, але тепер потрібно згенерувати фактичні вершини. На рисунку 3.11 показана загальна конфігурація для прикладу.

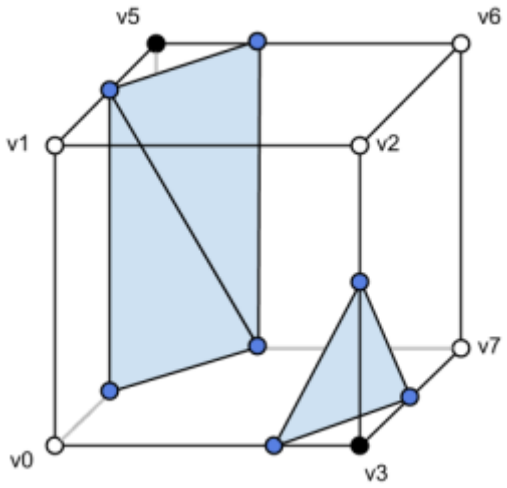


Рисунок 3.11. Трикутники для випадку 56. Сині точки показують ребра, на яких повинні бути вершини. Сині області показують створені трикутники.

Вершина повинна бути розміщена там, де значення щільності приблизно дорівнює 0, і це робиться шляхом лінійної інтерполяції. Наприклад, якщо ребро 11 потребує вершини на ньому, а значення щільності в v3 і v7 (вершини, які з'єднані вздовж ребра 11) становлять 0,2 і -0,8 відповідно, тоді остаточна вершина повинна бути розміщена на 25% від v3 уздовж ребра 11. Наступне рівняння можна використовувати для розрахунку положення вершини в 3D[17].

$$P = P_1 + (-D_1) * (P_2 - P_1)/(D_2 - D_1)$$

Де P1 і P2 – це положення вершин, з'єднаних ребром, а D1 і D2 – значення щільності в цих вершинах.

На цьому етапі список трикутників може бути створений із згенерованих вершин і надісланий до GPU для візуалізації. Три послідовні вершини в списку трикутників утворюють один трикутник, як це видно на рисунку 3.12. Цей процес потрібно виконати для кожного вокселя в об'ємному просторі. Ось чому це обчислювально дорогий алгоритм, хоча таблиці пошуку є швидкими.

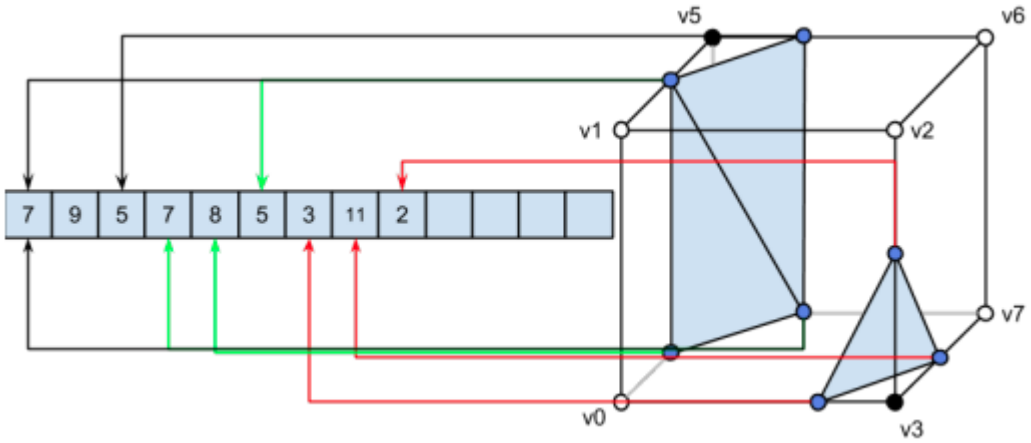


Рисунок 3.12. Приклад того, як вершини поміщаються в буфер вершин для випадку номер 56.

На рисунку 3.12 зображено як числа в буфері відносяться до ребра, на якому була створена вершина. Фактичні дані, що зберігаються в буфері, є визначеною користувачем структурою вершин.

Результат виконання алгоритму генерації полігональної сітки можна побачити нижче, на рисунку 3.13 та 3.14, на модель додатково застосовано шейдер який змінює колір відносно висоти на якій знаходиться фрагмент острову.

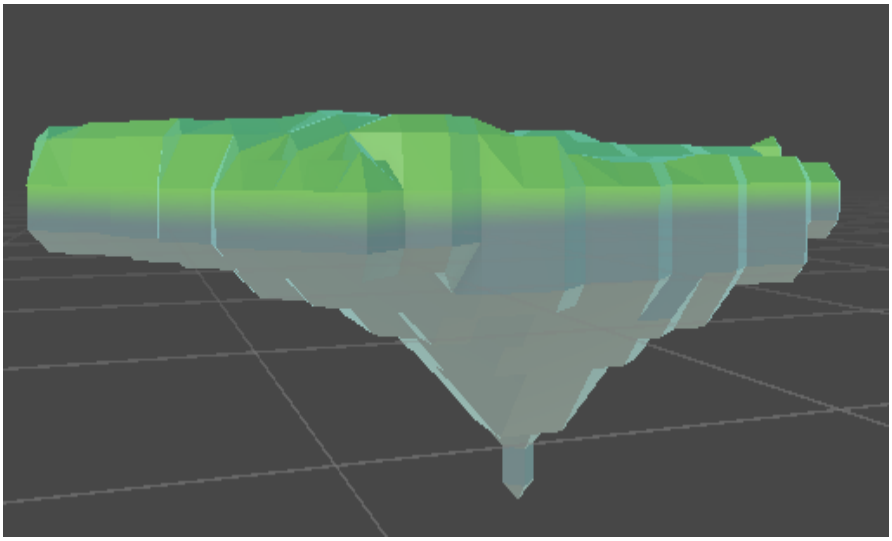


Рисунок 3.13 Острів створений алгоритмом крокуючих кубів, вид збоку.

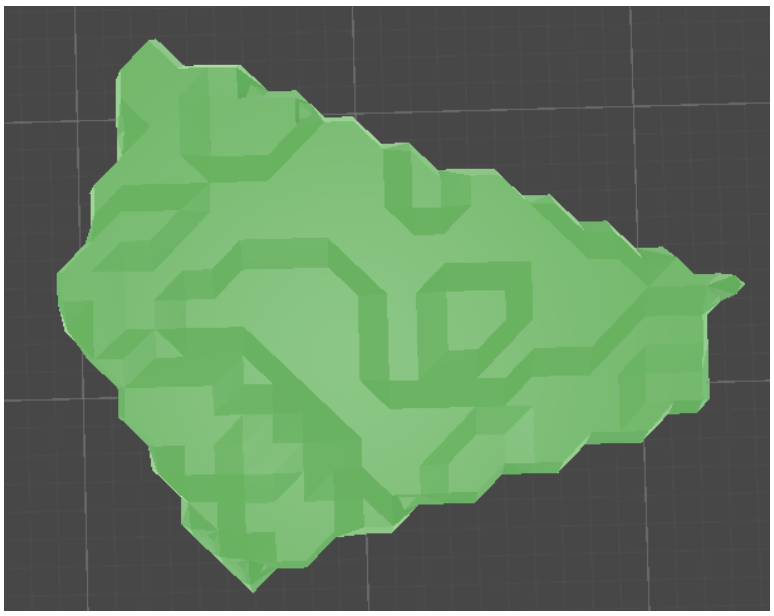


Рисунок 3.14 Острів створений алгоритмом крокуючих кубів, вид зверху.

4. Останнім етапом буде розміщення островів в просторі та створення переходів між ними. Для розміщення оптимальним способом буде використовувати початкові положення точок простору островів та зміщення від центру мапи на деяку відстань, код який це виконує показано на рисунку 3.15.

```
foreach (var islandIndex in map.Keys)
{
    var transform = _meshGen.GetGroupTransform(islandIndex);
    // Отримання початкової позиції острова та зміщення відносно центру мапи.
    var initPos = _meshGen.GetGroupInitialPosition(islandIndex) - new Vector3(Size.x / 2f, 0, Size.y / 2f);
    // Розсередження островів використовуючи параметр spacing
    transform.position = new Vector3(initPos.x * spacing, 0, initPos.z * spacing);
}
```

Рисунок 3.15. Розміщення островів на рівні.

Створення зв’язків між островами можна виконати великою кількістю алгоритмів, але в цій роботі було вирішено створити граф, де кожен острів є вершиною графа в той час як переходи між островами будуть ребрами цього графа.

Код який представлено на рисунку 3.16, виконує прохід по всім вершинам графа та створює зв’язки між трьома найближчими вершинами. Також задано умову при якій кожна додавати зв’язки лише якщо вершина має мати не більше трьох ребер. Результат виконання цього алгоритму показано на рисунку 3.17.

```
var visited = new HashSet<int>(); // Хеш таблиця яка містить індекси відвіданих вершин
// Словник який містить індекси вершин графу та відповідні вершини
_availableNodes = new Dictionary<int, GraphNode>() { [first]=_mapGraph };
var graphQueue = new Queue<GraphNode>(); // Черга з вершинами які повинні бути відвідані
graphQueue.Enqueue(_mapGraph);

var sort = new List<(int, float)>();
// Цикл виконується поки в черзі є вершини
while (graphQueue.Count > 0)
{
    var node = graphQueue.Dequeue(); // Дістаємо вершину з черги
    if (visited.Contains(node.Index)) continue; // Якщо вершина вже відвідана, пропускаємо
    var nodeCenter = _meshGen.GetGroupInitialPosition(node.Index);
    visited.Add(node.Index );
    sort.Clear();

    // Тут додаємо у список можливих з'єднань вершини, відстань між якими менша ніж rangeTres та ребро до якої ще не існує.
    foreach (var item in _voronoiGroups.Enumerate())
    {
        if (item.Size == 0) continue;
        if (node.HasConnection(item.Item1.Id) || visited.Contains(item.Item1.Id)) continue;
        float dist = (_meshGen.GetGroupInitialPosition(item.Item1.Id) - nodeCenter).magnitude;
        if (rangeTres < dist) continue;
        sort.Add((item.Item1.Id,dist));
    }
    // Сортуємо список можливих вершин за відстаню до вершини яку зараз розглядаємо.
    sort.Sort((item,item2) => item.Item2.CompareTo(item2.Item2));
    for (int i = 0; i < Mathf.Min(sort.Count,3); i++)
    {
        var item = sort[i];
        if (!_availableNodes.TryGetValue(item.Item1,out var nearNode))
        {
            nearNode = new GraphNode(item.Item1);
            _availableNodes.Add(nearNode.Index, nearNode);
        }
        // Додаємо зв'язки якщо кількість ребер вершин менша за 3
        if (node.Connections.Count < 3 && nearNode.Connections.Count < 3) node.AddConnection(nearNode);
        // Додаємо вершину до черги
        graphQueue.Enqueue(nearNode);
    }
}
```

Рисунок 3.16. Алгоритм створення зв’язків між островами.

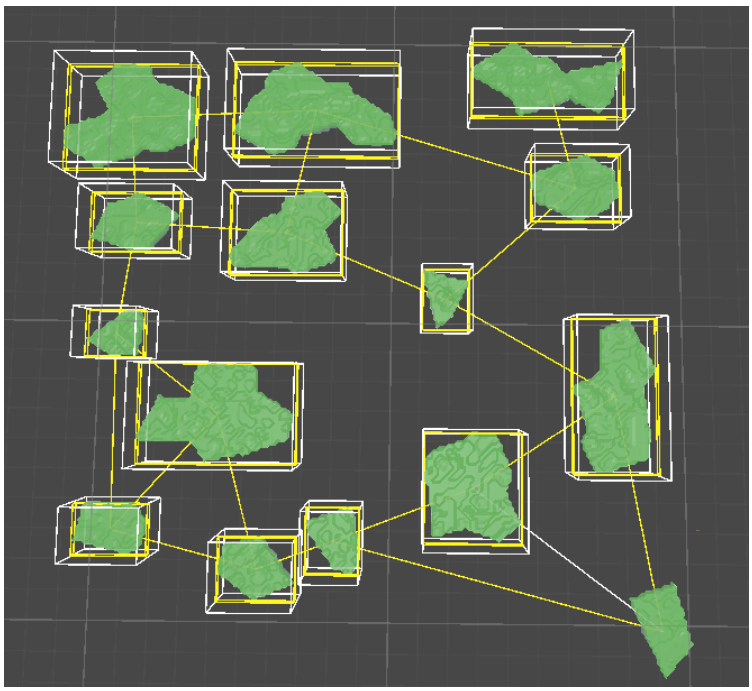
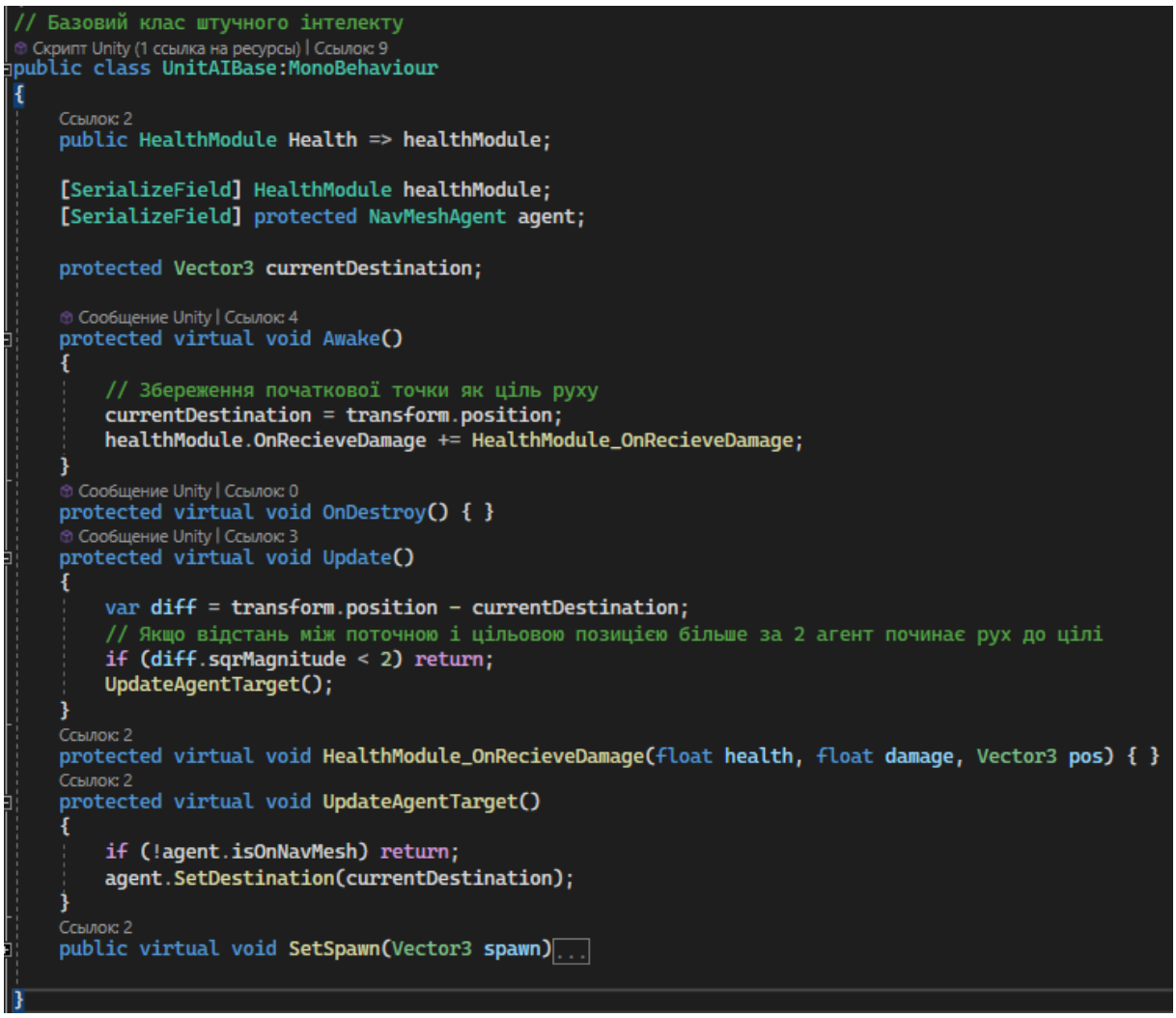


Рисунок 3.17. Створені зв’язки між островами позначено жовтими лініями.

3.3 Поведінка ворогів

Штучний інтелект розроблений з використанням компонента Unity – NavMeshAgent, який використовується для створення і керування агентами, що можуть переміщатися по навігаційній сітці (NavMesh). Навігаційна сітка визначає, де агенти можуть ходити, і використовується для навігації та пошуку шляхів у 3D-просторі. NavMeshAgent відповідає за обчислення та виконання цих шляхів, забезпечуючи плавне і реалістичне переміщення агентів.

Клас штучного інтелекту з назвою UnitAIBase є похідним для всіх інших видів штучного інтелекту та має базовий функціонал у вигляді руху до точки спавну та отримання урону від гравця, за що відповідає клас HealthModule. Код цього класу можна побачити на рисунку 3.18.



```
// Базовий клас штучного інтелекту
Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 9
public class UnitAIBase:MonoBehaviour
{
    Ссылок: 2
    public HealthModule Health => healthModule;

    [SerializeField] HealthModule healthModule;
    [SerializeField] protected NavMeshAgent agent;

    protected Vector3 currentDestination;

    Сообщение Unity | Ссылок: 4
    protected virtual void Awake()
    {
        // Збереження початкової точки як ціль руху
        currentDestination = transform.position;
        healthModule.OnRecieveDamage += HealthModule_OnRecieveDamage;
    }
    Сообщение Unity | Ссылок: 0
    protected virtual void OnDestroy() { }
    Сообщение Unity | Ссылок: 3
    protected virtual void Update()
    {
        var diff = transform.position - currentDestination;
        // Якщо відстань між поточною і цільовою позицією більше за 2 агент починає рух до цілі
        if (diff.sqrMagnitude < 2) return;
        UpdateAgentTarget();
    }
    Ссылок: 2
    protected virtual void HealthModule_OnRecieveDamage(float health, float damage, Vector3 pos) { }
    Ссылок: 2
    protected virtual void UpdateAgentTarget()
    {
        if (!agent.isOnNavMesh) return;
        agent.SetDestination(currentDestination);
    }
    Ссылок: 2
    public virtual void SetSpawn(Vector3 spawn)...

}
```

Рисунок 3.18. Базовий клас штучного інтелекту.

Для ворогів які переслідують гравця та атакують його було створено похідний клас з назвою MeleeAI, повний код з логікою штучного інтелекту завеликий, тому знаходиться у Додатку.

Коли гравець входить у радіус пошуку цілей, ворог починає рухатись до цілі, щоб та опинилась у зоні початку атаки. При досягненні цілі починається програвання анімації атаки, яка при успішному завершенні визиває код який відповідальний за перевірку знаходження цілі, якщо ціль у зоні нанесення урону, їй наноситься шкода яка дорівнює значенню атаки ворога. На рисунку 3.19 зображено зони атаки одного з ворогів.

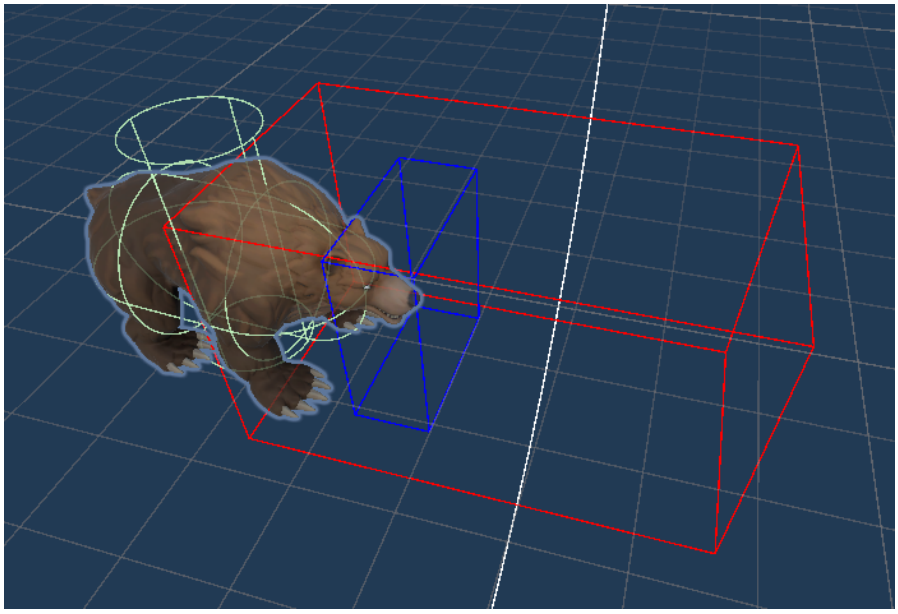


Рисунок 3.19. Зона атаки одного з ворогів, синім позначено зону початку атаки, а красним зону нанесення урону.

Вороги розміщуються на острові у випадкових позиціях та у кількості відповідній до розміру острова, приклад на рисунку 3.15. Розміщення починається коли гравець переходить до нового острову, вороги на якому ще не були переможені.

Рисунок 3.20. Розміщення та початок руху ворогів до гравця.

В процесі гри, коли ворог отримує шкоду яка опускає його здоров’я до нуля, програється анімація смерті та створюється відповідний ефект, приклад цього можна побачити на рисунку 3.21.

Рисунок 3.21. Ефект смерті ворога.

Поведінка боса рівня схожа на поведінку звичайного ворога, але додатково має можливість використовувати спеціальні здібності. Для боса першого рівня була створена здібність яка створює вибухи навколо гравця, які наносять шкоду якщо гравець буде знаходитись у радіусі вибуху, приклад цього можна побачити на рисунку 3.22.

Рисунок 3.22. Здібності боса рівня

3.4 Логіка роботи рівня гри

Гра починається з появи гравця на одному з островів рівня. Для першого рівня початком встановлюється один з кутових островів, в той час як кінцевим стає протилежний кут рівня. Бій з босом рівня, враховуючи спеціальні здібності, потребує достатньо простору для того щоб гравець мав змогу ухилитися від отримання удару, виходячи з цього кінцевий острів робиться найбільшим з усіх інших островів рівня.

Перемога на кожному острові супроводжується нагородою гравця у вигляді посилення персонажа. Сила ефектів залежить від складності острова та кількості ворогів на ньому, також вони розрізняється за силою ефекта.

Ось види ефектів які можна отримати за перемогу на острові:

- Збільшення швидкості бігу
- Збільшення сили атаки

- Зменшення часу перезарядження здібності ухилення
- Збільшення максимального здоров’я
- Відновлення здоров’я

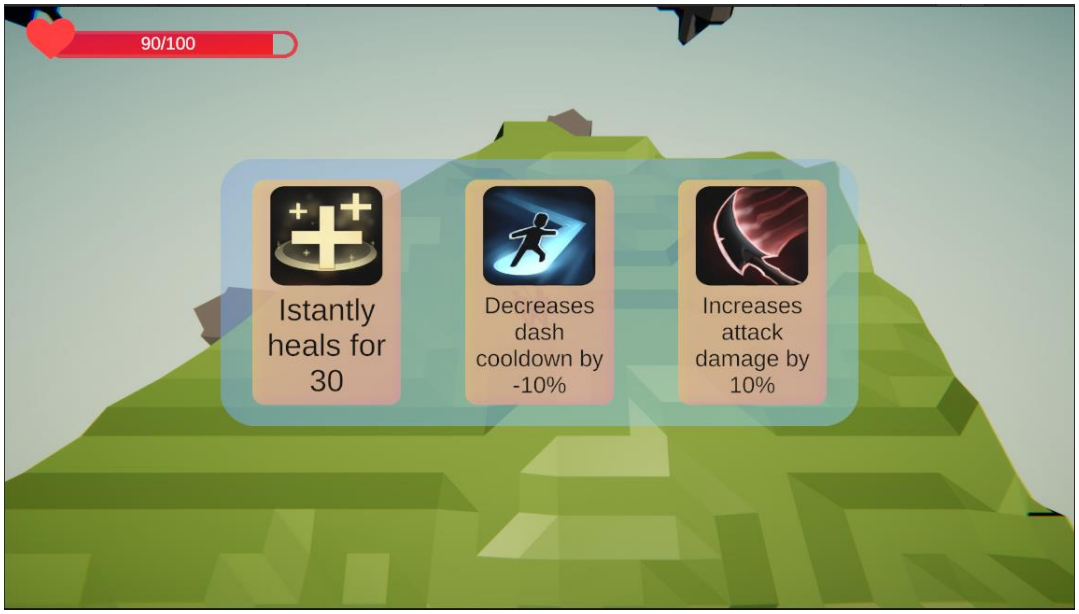


Рисунок 3.23 Вибір покращень після перемоги

Головною ціллю гравця є пошук кінцевого острова та перемога над босом рівня, після їх виконання гравець переходить на наступний рівень зберігаючи всі посилення.

ВИСНОВКИ

1. Розроблено повнофункціональну відеогру, яка реалізує елементи жанру rogue-lite з процедурною генерацією рівнів, що в свою чергу забезпечує різноманітність та непередбачуваність ігрового процесу.

2. Використання ігрового рушія Unity та мови програмування C# дозволило реалізувати основні механіки геймплею, забезпечуючи високу якість та ефективність розробки.

3. Розроблено систему процедурної генерації рівнів, яка створює унікальні ігрові рівні, які забезпечують непередбачуваність та високу повторюваність гри, що є ключовими елементами жанру rogue-lite.

4. Реалізовано штучний інтелект для ворогів, що забезпечує високу складність та інтерактивність боїв у грі.

5. Практична значимість результатів роботи полягає у створенні повнофункціональної відеогри, яка демонструє високий рівень технічної реалізації та творчого підходу. Отримані результати можуть бути використані як основа для подальшої розробки ігор у жанрі rogue-lite або інших жанрах, що використовують процедурну генерацію рівнів та системи штучного інтелекту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rogue (video game) [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Rogue_(video_game)
2. Berlin Interpretation [Електронний ресурс] – Режим доступу до ресурсу: https://roguebasin.com/index.php/Berlin_Interpretation.
3. Enter the Gungeon [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Enter_the_Gungeon
4. Official Enter the Gungeon Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://enterthegungeon.fandom.com/wiki/Enter_the_Gungeon_Wiki
5. Noita (video game) [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Noita_(video_game)
6. MacLeod R. Noita Is A Delightful Game Where You Can Destroy Every Pixel You See. *Kotaku*. URL: https://kotaku.com/noita-is-a-delightful-game-where-you-can-destroy-every-1838376682 (дата звернення: 20.05.2024).
7. Slay the Spire [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Slay_the_Spire
8. YoungWonks. Which are the leading game development engines in the market today? Our latest blog post brings you a comprehensive list. *YoungWonks*. URL: https://www.youngwonks.com/blog/Top-10-Game-Development-Engines-Today (дата звернення: 20.05.2024).
9. The Best Gaming Engines for 2024 - Incredibuild. *Incredibuild*. URL: https://www.incredibuild.com/blog/top-gaming-engines-you-should-consider (дата звернення: 07.06.2024).
10. Unity Plans and pricing [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/products?c=unity+engine&s=gaming

11. Voronoi diagram [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Voronoi_diagram.

12. Voronoi Noise [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.ronja-tutorials.com/post/028-voronoi-noise/.

13. Perlin noise [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Perlin_noise.

14. A. Lagae. A Survey of Procedural Noise Functions / A. Lagae, S. Lefebvre, R. Cook. // Computer Graphics Forum. – 2010. – С. 1–3.

15. David J. Field, What the statistics of natural images tell us about visual coding // Proceedings of SPIE 1077, Human Vision, Visual Processing, and Digital Display – 1989. - С. 269-277.

16. Gerrit Greeff. Interactive voxel terrain design using procedural techniques. // Master's thesis, Stellenbosch: University of Stellenbosch - 2009.

17. William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm.// SIGGRAPH Comput. Graph.– 1987 – С. 163-169.

18. Artificial intelligence in video games [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

19. Generating Complex Procedural Terrains Using the GPU [Електронний ресурс] – Режим доступу до ресурсу: https://developer.nvidia.cn/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu.

ДОДАТКИ

ПРОЦЕДУРНА ГЕНЕРАЦІЇЯ РІВНЯ

```
public class VoronoiCellMap : VoronoiMap
    {
        protected int divisions;
        protected int[,] cells;
        public VoronoiCellMap(Vector2Int size, int divisions)
        {
            //if (size.x % divisions != 0 || size.y % divisions != 0) throw
new ArgumentException($"Size of map {size} is not divisible by {divisions}");

            this.divisions = divisions;
            _mapSize = size;
            _seedCount = divisions * divisions;
            _voronoiCells = new PointGroup[_seedCount];
            _pointToIndex = new Dictionary<Vector2Int, int>((int)(size.x *
size.y));

        }
        public override void Calculate()
        {
            // Визначення положення вершин діаграми вороного
            // Тут _mapSize це вектор з розміром рівня. divisions - кількість
клітин у кожному рядку та стовпчику сітки рівня.
            float xStep = _mapSize.x / divisions;
            float yStep = _mapSize.y / divisions;
            // Рівень поділяється на клітини з розміром xStep на yStep
            for (int x = 0; x < divisions; x++)
            {
                for(int y = 0; y < divisions; y++)
                {
                    float xValue = Random.Range(xStep * x, xStep * (x + 1));
                    float yValue = Random.Range(yStep * y, yStep * (y + 1));

                    // Вершина розміщються у випадковій точці всередині
клітини
                    var pos = new Vector2Int((int)xValue, (int)yValue);

                    // _voronoiCells - масив структур типу PointGroup які
зберігють дані про вершину та точки простору які є найближчими до вершини.
                    _voronoiCells[x + y * divisions] = new PointGroup(x + y *
divisions,pos);
                }
            }
            // Для кожної точки у просторі рівня знаходимо найближчу вершину
та зберегаємо
            for (int x = 0; x < _mapSize.x; x++)
            {
                for (int y = 0; y < _mapSize.y; y++)
                {
                    int minIndex = 0;
                    float min = float.MaxValue;
                    var point = new Vector2Int(x, y);
                    // Знаходмо координати клітини до якої відноситься точка
                    var gridX = (int)((float)x / xStep);
                    var gridY = (int)((float)y / yStep);

                    // Перевіряємо яка з 9 вершин навколо точки є найближчою.
                    foreach (var p in GetPointsAround(gridX,gridY))
                    {
                        var index = GetIndex(p.x, p.y);
```

```
                        if (index < 0 || index >= _seedCount) continue;

                        float dist = (point -
_voronoiCells[index].Center).sqrMagnitude;
                        if (min > dist) // Знаходження мінімальної відстані
до вершини
                        {
                            min = dist;
                            minIndex = index;
                        }
                    }
                    _voronoiCells[minIndex].Add(point);
                    _pointToIndex.Add(point, minIndex);
                }
            }

        }
        private IEnumerable<Vector2Int> GetPointsAround(Vector2Int center)
        {
            return GetPointsAround(center);
        }
        private IEnumerable<Vector2Int> GetPointsAround(int centerX, int
centerY)
        {
            for (int y = -1; y < 2; y++)
                for (int x = -1; x < 2; x++)
                    yield return new Vector2Int(centerX + x, centerY + y);
        }
        public bool IsBorderCell(int index)
        {
            int y = index / divisions;
            int x = index % divisions;
            if (y == 0 || x == 0) return true;
            if (x == divisions - 1 || y == divisions - 1) return true;
            return false;
        }
        private int GetIndex(int x, int y) => x + y * divisions;
    }
public class VoronoiMap
    {

        public Vector2Int Size => _mapSize;
        public int SeedCount => _seedCount;
        protected Vector2Int _mapSize;
        protected int _seedCount;
        protected Dictionary<Vector2Int, int> _pointToIndex;

        protected PointGroup[] _voronoiCells;
        public VoronoiMap()
        {
            _voronoiCells = new PointGroup[0];
            _pointToIndex = new Dictionary<Vector2Int, int>();
        }
        public VoronoiMap(Vector2Int size, int count)
        {
            _mapSize = size;
            _seedCount = count;
            _voronoiCells = new PointGroup[count];

            _pointToIndex = new Dictionary<Vector2Int, int>((int)(size.x *
size.y));
        }
```

```
        public virtual void Calculate()
        {
            for (int i = 0; i < _seedCount; i++)
            {
                var pos = new Vector2Int(Random.Range(0, _mapSize.x),
Random.Range(0, _mapSize.y));
                _voronoiCells[i] = new PointGroup(i,pos);
            }
            for (int x = 0; x < _mapSize.x; x++)
            {
                for (int y = 0; y < _mapSize.y; y++)
                {
                    int minIndex = 0;
                    float min = float.MaxValue;
                    var point = new Vector2Int(x, y);

                    Parallel.For(0, _seedCount, (i) =>
                    {
                        float dist = (point -
_voronoiCells[i].Center).sqrMagnitude;
                        if (min > dist)
                        {
                            min = dist;
                            minIndex = i;
                        }
                    });
                    _voronoiCells[minIndex].Add(point);
                    _pointToIndex.Add(point, minIndex);
                }
            }
        }
        public List<Vector2Int> GetPoints(int seed)
        {
            return _voronoiCells[seed].Points;
        }
        public int GetIndex(Vector2Int point) => _pointToIndex[point];
        public IEnumerable<(Vector2Int point, int seed)> GetMap()
        {
            for (int i = 0; i < _seedCount; i++)
            {
                foreach (var item in _voronoiCells[i].Points)
                {
                    yield return (item,i);
                }
            }
        }
        public Vector2Int GetSeed(int index) => _voronoiCells[index].Center;
        public PointGroup GetGroup(int index) => _voronoiCells[index];
    }


public class VoronoiGroup
    {
        protected (PointGroup<int>,int Size)[] _groups;
        protected Vector3[] _avgCenters;
        protected VoronoiMap _map;
        public int GroupsCount { get; private set; }
        private int _divisions;
        public VoronoiGroup(VoronoiMap map, int divisions)
        {
            _map = map;
            GroupsCount = divisions * divisions;
```

```
            _divisions = divisions;
            _groups = new (PointGroup<int>,int)[divisions * divisions];
            _avgCenters = new Vector3[GroupsCount];
            float xStep = map.Size.x / divisions;
            float yStep = map.Size.y / divisions;
            for (int x = 0; x < divisions; x++)
            {
                for (int y = 0; y < divisions; y++)
                {
                    float xValue = Random.Range(xStep * x, xStep * (x + 1));
                    float yValue = Random.Range(yStep * y, yStep * (y + 1));

                    var pos = new Vector2Int((int)xValue, (int)yValue);
                    _groups[x + y * divisions] = (new PointGroup<int>(x + y *
divisions, pos),0);
                }
            }

            for (int i = 0; i < map.SeedCount; i++)
            {
                int nearestIndex = 0;
                float nearest = float.MaxValue;
                if (map is VoronoiCellMap cellMap && cellMap.IsBorderCell(i))
continue;
                for (int g = 0; g < GroupsCount; g++)
                {
                    var d = (_groups[g].Item1.Center -
map.GetSeed(i)).sqrMagnitude;
                    if (d < nearest)
                    {
                        nearest = d;
                        nearestIndex = g;
                    }
                }
                var group = _groups[nearestIndex];
                group.Size += map.GetGroup(i).Size;
                group.Item1.Add(i);
                _groups[nearestIndex] = group;
            }
            for (int i = 0; i < _groups.Length; i++)
            {
                var info = _groups[i].Item1.Id;
                var center = new Vector2();
                Parallel.ForEach(_groups[i].Item1.Points, (index) =>
                {
                    var voronoiCell = map.GetGroup(index);
                    var cellCenter = new Vector2();
                    for (int j = 0; j < voronoiCell.Points.Count; j++)
                    {
                        cellCenter += voronoiCell.Points[i];
                    }
                    cellCenter /= voronoiCell.Points.Count;
                    center += cellCenter;
                });
                center /= _groups[i].Item1.Points.Count;
                _avgCenters[i] = new Vector3(center.x, 0, center.y);
            }
        }
        public int GetGroupSize(int index) => _groups[index].Size;
        public PointGroup<int> GetGroup(int index) => _groups[index].Item1;
        public Vector2Int GetGroupCenter(int groupIndex) =>
_groups[groupIndex].Item1.Center;
```

```
        public List<int> GetGroupSeeds(int groupIndex) =>
_groups[groupIndex].Item1.Points;
        public Vector3 GetGroupAverageCenter(int index) =>
_avgCenters[index];

        public IEnumerable<(PointGroup<int>, int Size)> Enumerate()
        {
            return _groups;
        }
        public void GetAdjecentGroups(int index, List<int> buffer)
        {
            int y = index / _divisions;
            int x = index % _divisions;
            int r = (x + 1) + y * _divisions;
            int l = (x - 1) + y* _divisions;
            int t = x + (y + 1) * _divisions;
            int d = x + (y -1) * _divisions;
            if (r < _groups.Length && r >= 0)
                buffer.Add(r);
            if (l < _groups.Length && l >= 0)
                buffer.Add(l);
            if (t < _groups.Length && t >= 0)
                buffer.Add(t);
            if (d < _groups.Length && d >= 0)
                buffer.Add(d);
        }
        public void GetGroupsAround(int index, List<int> buffer)
        {
            int y = index / _divisions;
            int x = index % _divisions;

            for (int xShift = -1; xShift < 2; xShift++)
            {
                for (int yShift = -1; yShift < 2; yShift++)
                {
                    int shiftIndex = (x + xShift) + (y + yShift) *
_divisions;
                    if (shiftIndex == index || shiftIndex < 0 || shiftIndex >
GroupsCount) continue;

                    buffer.Add(shiftIndex);
                }
            }
        }
        public bool SwapGroupCenters(int index, int secondIndex)
        {
            if (index < 0 || secondIndex < 0 || index > _groups.Length ||
secondIndex > _groups.Length) return false;
            var first = _groups[index];
            var second = _groups[secondIndex];

            first.Item1.Id = second.Item1.Id;
            second.Item1.Id = _groups[index].Item1.Id;

            _groups[secondIndex] = first;
            _groups[index] = second;

            var tmp = _avgCenters[index];
            _avgCenters[index] = _avgCenters[secondIndex];
            _avgCenters[secondIndex] = tmp;
            return true;
```

```
            var pGroupF = first.Item1;
            var pGroupS = second.Item1;
            pGroupF.Center = second.Item1.Center;
            pGroupS.Center = first.Item1.Center;
            first.Item1 = pGroupF;
            second.Item1 = pGroupS;
            _groups[index] = second;
            _groups[secondIndex] = first;
            return true;
        }
    }

// Клас який відповідає за розділення мапи на воксельні чанки, які у
подальшому використовуються для генерації полігональної сітки
    public class WeightChunkGenerator
    {
        // Структура даних чанку
        public struct ChunkData
        {
            public Vector3Int Offset;
            public float[,,] Data;
            public float[] DataFlat;
            public int Group;
            public ChunkData(float[,,] data, float[] flatData, int group,
Vector3Int offset)
            {
                Data = data;
                Group = group;
                DataFlat = flatData;
                Offset = offset;
            }
        }
        // Групування чанків
        public struct ChunkGroupData
        {
            public List<int> Chunks;
            public float[,] GroupData;
            public BoundsInt Bounds;
            public ChunkGroupData(float[,] data, BoundsInt bounds)
            {
                Chunks = new List<int>();
                GroupData = data; Bounds = bounds;
            }
        }
        public int ChunkCount => _chunks.Count;
        private VoronoiMap _voronoiMap;
        public VoronoiGroup voronoiGroup;
        private Dictionary<int, ChunkData> _chunks;
        private Dictionary<int, ChunkGroupData> _chunkGroups;
        private int _chunkSize;
        private float[,] _heightMap;
        private int _roundDigits;
        private PerlinNoiseParams _noise;
        public WeightChunkGenerator(VoronoiMap map, VoronoiGroup group,
float[,] heightMap, int chunkSize, int roundDigits = 0, PerlinNoiseParams
noise = default)
        {
            _voronoiMap = map;
            voronoiGroup = group;
            _chunkSize = chunkSize;
            _heightMap = heightMap;
            _chunks = new Dictionary<int, ChunkData>();
```

```
            _chunkGroups = new Dictionary<int, ChunkGroupData>();
            _roundDigits = roundDigits;
            _noise = noise;
        }
        public void GenerateChunks()
        {
            int xStart = 0, xEnd = _chunkSize, yStart = 0, yEnd = _chunkSize;


            float maxX = float.MinValue, maxY = float.MinValue;
            Dictionary<int,Vector2> groupMin = new Dictionary<int,Vector2>();
            List<Vector2Int> groupPoints = new List<Vector2Int>();
            _chunks.Clear();
            _chunkGroups.Clear();
            int chunkIndex = 0;

            // Знаходження розміру групи вершин вороного
            var tasks = new List<Task<(int, ChunkData,int)>>();

            for (int i = 0; i < voronoiGroup.GroupsCount; i++)
            {
                groupPoints.Clear();
                var min = new Vector2Int(int.MaxValue, int.MaxValue);
                var max = new Vector2Int(int.MinValue, int.MinValue);
                foreach (var seedIndex in voronoiGroup.GetGroupSeeds(i))
                {
                    if (_voronoiMap is VoronoiCellMap cellMap &&
cellMap.IsBorderCell(seedIndex)) continue;
                    foreach (var point in _voronoiMap.GetPoints(seedIndex))
                    {
                        if (point.x < min.x) min.x = point.x;
                        if (point.y < min.y) min.y = point.y;
                        if (point.x > max.x) max.x = point.x;
                        if (point.y > max.y) max.y = point.y;
                        groupPoints.Add(point);
                    }
                }
                int xSize = (max.x - min.x) + 1, ySize = max.y - min.y + 1;
                xSize += (_chunkSize - xSize % _chunkSize); ySize +=
_chunkSize - ySize % _chunkSize;

                var groupData = new float[xSize, ySize];
                foreach (var point in groupPoints)
                {
                    var localPoint = point - min;
                    groupData[localPoint.x+1,localPoint.y+1] =
_heightMap[point.x,point.y];
                }
                if (max.x < 0 || max.y < 0) continue;

                _chunkGroups.Add(i, new ChunkGroupData(groupData, new
BoundsInt(min.x,min.y,0, max.x - min.x+1, max.y - min.y+1, 2)));
                var field = CreateGroupField(i);
                // Розбиття острова на чанки. _chunkSize - розмір чанка.
xSize,ySize - розміри острова які кратні розміру чанка.
                for (int chunkXOffset = 0; chunkXOffset < xSize; chunkXOffset
+= _chunkSize)
                {
                    for (int chunkYOffset = 0; chunkYOffset < ySize;
chunkYOffset += _chunkSize)
                    {
                        int chunkIndexTop = chunkIndex++;
```

```
                        int chunkIndexBottom = chunkIndex++;
                        var chunkTopOffset = new Vector3Int(chunkXOffset, 0,
chunkYOffset);
                        var chunkBottonOffset = new Vector3Int(chunkXOffset,
-_chunkSize, chunkYOffset);
                        int groupIndex = i;
                        // Створення задач для параленьного розрахунку
заповнення чанку
                        tasks.Add(Task.Run(() =>
                        CalculateChunkData(chunkIndexTop, groupData,
chunkTopOffset, groupIndex))
                            );
                        tasks.Add(Task.Run(() =>
                        CalculateChunkData(chunkIndexBottom, field,
chunkBottonOffset, groupIndex))
                            );
                    }
                }
                // Очікування виконання розрахунку на всіх чанках
                Task.WaitAll(tasks.ToArray());

                foreach (var task in tasks)
                {
                    (int chunkI, ChunkData chunkData,int values) =
task.Result;
                    _chunks.Add(chunkI, chunkData);
                    if (values == 0) continue;

                    // Збереження результатів до масиву
                    _chunkGroups[chunkData.Group].Chunks.Add(chunkI);
                }
                tasks.Clear();
            }


            (int chunkIndex, ChunkData data, int values)
CalculateChunkData(int chunkIndex, float[,] groupData, Vector3Int offset, int
groupIndex)
            {
                (float[,,] chunkData, float[] flatData) = (null,null);
                int values = 0;
                if (offset.y >= 0)
                    (chunkData, flatData) = PopulateChunkData(groupData,
offset.x, offset.z, out values);
                else
                    (chunkData, flatData) = PopulateBottomChunk(groupData,
offset.x, offset.z, out values);
                var data = new ChunkData(chunkData, flatData, groupIndex,
offset);
                return (chunkIndex, data, values);
            }


            (float[,,], float[]) PopulateChunkData(float[,] groupData, int
xOffset, int yOffset, out int values)
            {
                values = 0;

                int expSize = _chunkSize + 2;
                var flat = new float[expSize * expSize * expSize];
                var chunkData = new float[expSize, expSize, expSize];
```

```
                int xSize = expSize + xOffset > groupData.GetLength(0) ?
_chunkSize : expSize,
                    zSize = expSize + yOffset > groupData.GetLength(1) ?
_chunkSize : expSize;

                for (int x = 1; x < xSize; x++)
                {
                    for (int z = 1; z < zSize; z++)
                    {
                        for (int y = 0; y < _chunkSize; y++)
                        {
                            float height = groupData[x + xOffset, z +
yOffset];
                            int index = x + expSize * (y + expSize * z);
                            float cut = height - (int)height;
                            cut = (int)(cut * 100 / 25) * 25 / 100f;
                            if (y < height)
                            {
                                chunkData[x, y, z] = 1f;
                                flat[index] = 1f;
                                values++;
                            }else if ( y == Math.Ceiling(height) && cut != 0)
                            {
                                chunkData[x, y, z] = cut;
                                flat[index] = cut;
                            }
                        }
                    }
                }
                return (chunkData,flat);
            }
            (float[,,], float[]) PopulateBottomChunk(float[,] fieldData, int
xOffset, int yOffset, out int values)
            {
                values = 0;
                int expSize = _chunkSize + 2;
                var flat = new float[expSize * expSize * expSize];
                var chunkData = new float[expSize, expSize, expSize];
                int xSize = expSize + xOffset > fieldData.GetLength(0) ?
_chunkSize : expSize,
                    zSize = expSize + yOffset > fieldData.GetLength(1) ?
_chunkSize : expSize;

                for (int x = 1; x < xSize; x++)
                {
                    for (int z = 1; z < zSize; z++)
                    {
                        for (int y = _chunkSize; y >= 0; y--)
                        {
                            float height = fieldData[x + xOffset, z +
yOffset];
                            int index = x + expSize * (y + expSize * z);
                            float cut = (float)Math.Round(height -
(int)height, _roundDigits);

                            if (_chunkSize - y < height)
                            {
                                chunkData[x, y, z] = 1f;
                                flat[index] = 1f;
                                values++;
                            }
```

```
                            else if (_chunkSize - y == Math.Ceiling(height)
&& cut != 0)
                            {
                                chunkData[x, y, z] = cut;
                                flat[index] = cut;
                            }
                        }
                    }
                }
                return (chunkData, flat);
            }
        }

        public float[,,] GetChunk(int index) => _chunks[index].Data;
        public int GetChunkGroup(int index) => _chunks[index].Group;
        public float[] GetFlatChunk(int index) => _chunks[index].DataFlat;
        public Vector3Int GetChunkOffset(int index) => _chunks[index].Offset;
        public float[,] CreateGroupField(int index)
        {
            var groupData = _chunkGroups[index];

            var result = (float[,])groupData.GroupData.Clone();
            int minValue = _chunkSize/2, maxValue = _chunkSize-1;
            int count = 0; Vector2Int groupCenter = new Vector2Int();
            foreach (var seed in voronoiGroup.GetGroupSeeds(index))
            {
                if (_voronoiMap is VoronoiCellMap cellMap &&
cellMap.IsBorderCell(seed)) continue;
                var value = Random.Range(minValue, maxValue);

                NewFill(_voronoiMap.GetSeed(seed), value);
                count++;
                groupCenter += _voronoiMap.GetSeed(seed);
            }
            var groupVal = Random.Range(_chunkSize/2, _chunkSize);
            groupCenter /= count;
            NewFill(groupCenter, _chunkSize);

            return result;

            void Fill(Vector2Int point, int amount)
            {
                for (int x = point.x - amount; x < point.x + amount; x++)
                {
                    for (int y = point.y - amount; y < point.y + amount; y++)
                    {
                        float dist = (point - new Vector2Int(x,
y)).magnitude;
                        int value = (int)(amount - dist);

                        if (!groupData.Bounds.Contains(new Vector3Int(x, y,
0))) continue;
                        var index = new Vector3Int(x, y, 0) -
groupData.Bounds.min;
                        if (result[index.x, index.y] >= value ||
result[index.x, index.y] == 0f) continue;

                        result[index.x, index.y] = value + _noise.Apply(x,y)
* Math.Max(1,value);
                    }
                }
            }
```

```
            void NewFill(Vector2Int point, int amount)
            {
                float divider = 0.3f;
                for (int x = groupData.Bounds.min.x; x <
groupData.Bounds.max.x; x++)
                {
                    for (int y = groupData.Bounds.min.y; y <
groupData.Bounds.max.y; y++)
                    {
                        float dist = (point - new Vector2Int(x,
y)).magnitude;
                        int value = (int)(amount/ ((dist* divider) + 1.5f));

                        if (!groupData.Bounds.Contains(new Vector3Int(x, y,
0))) continue;
                        var index = new Vector3Int(x, y, 0) -
groupData.Bounds.min;
                        if (result[index.x, index.y] >= value ||
result[index.x, index.y] == 0f) continue;

                        result[index.x, index.y] = value + _noise.Apply(x, y)
- 2;
                    }
                }
            }
        }
        public bool TryGetChunkGroup(int index, out ChunkGroupData group) =>
_chunkGroups.TryGetValue(index, out group);

        public bool SwapGroupChunks(int index1, int index2)
        {
            if (!_chunkGroups.TryGetValue(index1, out var group1) ||
!_chunkGroups.TryGetValue(index2, out var group2)) return false;

            _chunkGroups[index2] = group1;
            _chunkGroups[index1] = group2;
            return true;
        }
    }



// Клас відповідальний за створення рівня який складається з роздільних
островів з використанням алгоритму створення діаграми вороного.
public class MapGeneration : MonoBehaviour
{
    public int GroupCount { get; private set; }
    public IReadOnlyDictionary<int, GraphNode> Nodes => _availableNodes;

    [SerializeField] private Vector2Int Size = new Vector2Int(16,16);
    [SerializeField] private Vector2 CellSize = new Vector2(1,1);
    [SerializeField] private float scale = 1f;
    [SerializeField] private float heightScale = 1f;
    [SerializeField] private Color minColor = new Color(0,1,0);
    [SerializeField] private Color maxColor = new Color(1, 0, 0);
    [SerializeField] private int seedCount = 8;

    [SerializeField] private int divisions = 1;

    [SerializeField] private int DrawChunk = 0;
    [SerializeField] private MapMeshGenerator _meshGen;
    [SerializeField] private PerlinNoiseParams _noiseParams;
```

```
    [SerializeField] private PerlinNoiseParams _bottomNoiseParams;
    [SerializeField] private float rangeTres = 30f;
    [SerializeField] private float spacing = 2f;

    private float[,] HeightMap= new float[0,0];
    private float min, max;
    private VoronoiMap _voronoi;
    private VoronoiGroup _voronoiGroups;
    private WeightChunkGenerator _chunkGen;
    private GraphNode _mapGraph;
    private int _chunkSize = 8;

    private int lastItemIndex = -1;
    private int maxSizeIndex = -1;
    private Dictionary<int,GraphNode> _availableNodes;
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
    // Метод сворює нову мапу
    [ContextMenu("Regenerate")]
    public void Generate()
    {
        seedCount = divisions * divisions;
        Profiler.BeginSample("Voronoi map Gen");
        _voronoi = new VoronoiCellMap(Size, divisions);
        _voronoi.Calculate();
        Profiler.EndSample();

        Profiler.BeginSample("Height map Gen");

        HeightMap = new float[Size.x,Size.y];
        min = float.MaxValue;
        max = float.MinValue;
        for (int x = 0; x < HeightMap.GetLength(1); x++)
        {
            for (int y = 0; y < HeightMap.GetLength(0); y++)
            {
                HeightMap[x,y] = _noiseParams.Apply(x,y);
                if (min > HeightMap[x,y]) min = HeightMap[x,y];
                if (max < HeightMap[x,y]) max = HeightMap[x,y];
            }
        }
        Profiler.EndSample();
        Profiler.BeginSample("VoronoiGroup Gen");
        _chunkSize = Size.x / 8;
        _voronoiGroups = new VoronoiGroup(_voronoi, divisions/2);
        Profiler.EndSample();

        Profiler.BeginSample("WeightChunkGenerator Gen");
        _chunkGen = new WeightChunkGenerator(_voronoi, _voronoiGroups,
HeightMap, _chunkSize, 0, _bottomNoiseParams);
        _chunkGen.GenerateChunks();
        Profiler.EndSample();

        Profiler.BeginSample("Mesh Gen");
```

```
        _meshGen.Setup(_chunkGen, _chunkSize);
        Profiler.EndSample();

        //_chunkGen.CreateGroupField(0);
        Profiler.BeginSample("Map Gen");
        CreateMap();
        Profiler.EndSample();
    }

    private void OnDrawGizmos()
    {
        if (_voronoi == null) return;

        if (_chunkGen == null || _chunkGen.ChunkCount == 0) return;
        if (DrawChunk > _chunkGen.ChunkCount) DrawChunk = 0;
        else if (DrawChunk < 0) DrawChunk = _chunkGen.ChunkCount-1;

        //var data = _chunkGen.GetChunk(DrawChunk);
        //Gizmos.color = UnityEngine.Color.white;
        //var size = new Vector3Int(data.GetLength(0), data.GetLength(1),
data.GetLength(2));
        //for (int x = 0; x < data.GetLength(0); x++)
        //{
        //    for (int y = 0; y < data.GetLength(1); y++)
        //    {
        //        for (int z = 0; z < data.GetLength(2); z++)
        //        {
        //            var pos = transform.position + new Vector3(x, y, z);
        //            var value = data[x, y, z];
        //            if (value == 0) continue;
        //            Gizmos.DrawWireCube(pos, Vector3.one);
        //        }
        //    }
        //}


        //if (_mapTree == null) return;
        //int lastIndex = _mapTree.Index;
        //_mapTree.Traverse(node =>
        //{
        //    if (node.Parent == null) return;
        //    var from =
_meshGen.GetGroupTransform(node.Parent.Index).position;
        //    var to = _meshGen.GetGroupTransform(node.Index).position;
        //    Gizmos.DrawLine(from, to);
        //});

        if (_mapGraph == null) return;
        _mapGraph.Traverse((node1, node2) =>
        {
            var from = _meshGen.GetGroupTransform(node1.Index).position;
            var to = _meshGen.GetGroupTransform(node2.Index).position;
            Gizmos.DrawLine(from, to);
            var bounds = _meshGen.GetMeshBounds(node2.Index);
            Gizmos.color = Color.white;
            Gizmos.DrawWireCube(bounds.center,bounds.size);
            var groupBounds = _meshGen.GetGroupBounds(node2.Index);
            Gizmos.color = Color.yellow;
            Gizmos.DrawWireCube(groupBounds.center, groupBounds.size);
        });
    }
```

```
    private void CreateMap()
    {
        Dictionary<int, Vector2> map = new Dictionary<int, Vector2>();
        int min = int.MaxValue, max = int.MinValue;
        int startPos = 0, first = -1;
        foreach (var item in _voronoiGroups.Enumerate())
        {
            if (item.Size == 0) continue;
            if (first == -1) first = item.Item1.Id;
            if (item.Size > max)
            {
                max = item.Size;
                maxSizeIndex = item.Item1.Id;
            }
            if (item.Size < min)
            {
                min = item.Size;
                startPos = item.Item1.Id;
            }
            var center = item.Item1.Center;
            var pos = new Vector2(center.x, center.y);
            map.Add(item.Item1.Id, pos);
            lastItemIndex = item.Item1.Id;
        }
        GroupCount = map.Count;
        _voronoiGroups.SwapGroupCenters(lastItemIndex, maxSizeIndex);
        _chunkGen.SwapGroupChunks(lastItemIndex, maxSizeIndex);
        _meshGen.SwamGroupPositions(lastItemIndex, maxSizeIndex);



        _mapGraph = new GraphNode(first);

        var path = Dijkstra(_mapGraph,
lastItemIndex,_voronoiGroups.GetGroupsAround, (f, s) =>
        {
            if (!_meshGen.TryGetGroupTransform(f, out var aTransform) ||
!_meshGen.TryGetGroupTransform(s, out var bTransform)) return float.MaxValue;
            return (aTransform.position - bTransform.position).magnitude;
        });

        foreach (var islandIndex in map.Keys)
        {
            var transform = _meshGen.GetGroupTransform(islandIndex);
            // Отримання початкової позиції острова та зміщення відносно
центру мапи.
            var initPos = _meshGen.GetGroupInitialPosition(islandIndex) - new
Vector3(Size.x / 2f, 0, Size.y / 2f);
            // Розсередження островів використовуючи параметр spacing
            transform.position = new Vector3(initPos.x * spacing, 0,
initPos.z * spacing);
        }
        //Debug.Log("Path: " + path[0]);
        for (int i = 1; i < path.Count; i++)
        {
            var pPos = _meshGen.GetGroupTransform(path[i-1]).position;
            var Pos = _meshGen.GetGroupTransform(path[i]).position;
            Debug.DrawLine(pPos, Pos, Color.red, 10f);
            //Debug.Log("Path: " + path[i]);
        }
```

```
        var visited = new HashSet<int>(); // Хеш таблиця яка містить індекси
відвіданих вершин
        _availableNodes = new Dictionary<int, GraphNode>() {
[first]=_mapGraph }; // Словник який містить індекси вершин графу та
відповідні вершини
        var graphQueue = new Queue<GraphNode>(); // Черга з вершинами які
повинні бути відвідані
        graphQueue.Enqueue(_mapGraph);

        var sort = new List<(int, float)>();
        // Цикл виконується поки в черзі є вершини
        while (graphQueue.Count > 0)
        {
            var node = graphQueue.Dequeue(); // Дістаємо вершину з черги
            if (visited.Contains(node.Index)) continue; // Якщо вершина вже
відвідана, пропускаємо
            var nodeCenter = _meshGen.GetGroupInitialPosition(node.Index);
            visited.Add(node.Index );
            sort.Clear();

            // Тут додаємо у список можливих з'єднань вершини, відстань між
якими менша ніж rangeTres та ребро до якої ще не існує.
            foreach (var item in _voronoiGroups.Enumerate())
            {
                if (item.Size == 0) continue;
                if (node.HasConnection(item.Item1.Id) ||
visited.Contains(item.Item1.Id)) continue;
                float dist = (_meshGen.GetGroupInitialPosition(item.Item1.Id)
- nodeCenter).magnitude;
                if (rangeTres < dist) continue;
                sort.Add((item.Item1.Id,dist));
            }
            // Сортуємо список можливих вершин за відстаню до вершини яку
зараз розглядаємо.
            sort.Sort((item,item2) => item.Item2.CompareTo(item2.Item2));
            for (int i = 0; i < Mathf.Min(sort.Count,3); i++)
            {
                var item = sort[i];
                if (!_availableNodes.TryGetValue(item.Item1,out var
nearNode))
                {
                    nearNode = new GraphNode(item.Item1);

                    _availableNodes.Add(nearNode.Index, nearNode);
                }
                // Додаємо зв'язки якщо кількість ребер вершин менша за 3
                if (node.Connections.Count < 3 && nearNode.Connections.Count
< 3)
                {
                    node.AddConnection(nearNode);
                }
                // Додаємо вершину до черги
                graphQueue.Enqueue(nearNode);
            }
        }
        // Виправлення накладень островів один на одного
        FixIntersections();
    }
    // Метод виконує усунення перетину островів
    [ContextMenu("Fix intersections")]
    private void FixIntersections()
    {
```

```
        var collisionList = new List<(int, int, float)>();
        FindCollisionsOnMap(collisionList);
        var iterations = 16;
        while (collisionList.Count > 0 && iterations-- > 0)
        {
            foreach (var collision in collisionList)
            {
                var aT = _meshGen.GetGroupTransform(collision.Item1);
                var bT = _meshGen.GetGroupTransform(collision.Item2);
                var aDiff = (aT.position - transform.position);
                var bDiff = (bT.position - transform.position);
                if (aDiff.sqrMagnitude < bDiff.sqrMagnitude)
                {
                    Debug.DrawLine(bT.position, bT.position +
bDiff.normalized * collision.Item3 * 2f, Color.red, 20f);
                    bT.position = bT.position + bDiff.normalized *
(collision.Item3 + 3f);
                    Debug.Log($"Shifted {collision.Item2} group
{collision.Item3}", bT);

                }
                else
                {
                    Debug.DrawLine(aT.position, aT.position +
aDiff.normalized * collision.Item3 * 2f, Color.red, 20f);

                    aT.position = aT.position + aDiff.normalized *
(collision.Item3 +3f);
                    Debug.Log($"Shifted {collision.Item1} group by
{collision.Item3}", aT);

                }
            }
            //Debug.Log($"Moved {collisionList.Count} groups");
            collisionList.Clear();
            FindCollisionsOnMap(collisionList);

        }
    }
    // Метод виконує пошук перетину островів та вносить дані у буфер
    private void FindCollisionsOnMap(List<(int, int, float)> buffer)
    {
        var aroundBuffer = new List<int>(9);
        var visited = new HashSet<int>();
        foreach (var item in _availableNodes.Keys)
        {

            var centerBounds = _meshGen.GetMeshBounds(item);
            aroundBuffer.Clear();
            _voronoiGroups.GetGroupsAround(item, aroundBuffer);

            foreach (var group in aroundBuffer)
            {
                if (!_availableNodes.ContainsKey(group)) continue;

                var bBounds = _meshGen.GetMeshBounds(group);
                var aPoint = centerBounds.ClosestPoint(bBounds.center);
                var bPoint = bBounds.ClosestPoint(centerBounds.center);

                if (!visited.Contains(group) && centerBounds.Contains(bPoint)
&& (aPoint - bPoint).magnitude > 0f)
                {
```

```
                    visited.Add(item);
                    visited.Add(group);
                    buffer.Add((item, group, (aPoint - bPoint).magnitude));

                }

            }
        }
    }
    private List<int> Dijkstra(GraphNode root, int
target,Action<int,List<int>> adjecentGetter, Func<int,int,float>
distanceFunc)
    {
        var visited = new HashSet<int>();
        var distDict = new Dictionary<int, (int from,float distance)>() {
[root.Index] = (0,0) };
        var nodeDict = new Dictionary<int, GraphNode>() { [root.Index] = root
};
        var path = new List<int>();

        var graphQueue = new Queue<GraphNode>();
        var adjecent = new List<int>(9);
        graphQueue.Enqueue(root);

        while (graphQueue.Count > 0)
        {
            var node = graphQueue.Dequeue();
            if (visited.Contains(node.Index)) continue;
            visited.Add(node.Index);
            adjecent.Clear();
            adjecentGetter(node.Index, adjecent);
            var distToNode = distDict[node.Index];

            foreach (var adj in adjecent)
            {

                if (!distDict.TryGetValue(adj, out var value))
                {
                    value = (-1, float.MaxValue);
                }
                float distNodeAdj = distanceFunc(node.Index, adj);
                if (distNodeAdj == float.MaxValue) continue;

                if (distToNode.distance + distNodeAdj < value.distance)
                {
                    distDict[adj] = (node.Index, distToNode.distance +
distNodeAdj);
                }
                if (visited.Contains(adj)) continue;
                if (!nodeDict.TryGetValue(adj, out var adjNode)){
                    adjNode = new GraphNode(adj);
                }

                //node.AddConnection(adjNode);
                graphQueue.Enqueue(adjNode);
            }
        }
        int from = target;
        while (from != root.Index)
        {
            var pair = distDict[from];
            path.Add(from);
```

```
            from = pair.from;
        }
        path.Add(root.Index);
        return path;
    }

    [ContextMenu("Move end group")]
    private void AdjustEndPosition()
    {
        int iterations = 6;
        var aroundBuffer = new List<int>(9);
        _voronoiGroups.GetGroupsAround(lastItemIndex, aroundBuffer);
        var lastTransform = _meshGen.GetGroupTransform(lastItemIndex);
        var fromCenterDir = (lastTransform.position -
transform.position).normalized;
        _chunkGen.TryGetChunkGroup(lastItemIndex, out var groupA);

        while (iterations-- > 0)
        {
            var aBounds = new Bounds(lastTransform.position, new
Vector3(groupA.Bounds.size.x*1.5f, groupA.Bounds.size.z, groupA.Bounds.size.y
* 1.5f));

            foreach (var group in aroundBuffer)
            {
                var aPos = lastTransform.position;
                if (!_meshGen.TryGetGroupTransform(group, out var
bTransform)) continue;
                var bPos = bTransform.position;

                _chunkGen.TryGetChunkGroup(group, out var groupB);
                var bBounds = new Bounds(bPos, new
Vector3(groupB.Bounds.size.x, groupB.Bounds.size.z, groupB.Bounds.size.y));
                Vector3 dif = aPos - bPos;
                float diffMag = dif.magnitude;
                var onDiffA = Vector3.Dot(dif, groupA.Bounds.size) / diffMag;
                var onDiffB = Vector3.Dot(dif, groupB.Bounds.size) / diffMag;
                if (aBounds.Intersects(bBounds))
                {
                    Debug.Log($"Intersection {group}");
                    float intersectionDist = ((aPos -
bBounds.ClosestPoint(aPos)) - (bPos - aBounds.ClosestPoint(bPos))).magnitude
+ 5f;
                    var shift = intersectionDist * fromCenterDir;
                    lastTransform.position = aPos + shift;
                    break;
                }

            }
        }
    }

    public Bounds GetBounds(int nodeIndex)
    {
        return _meshGen.GetMeshBounds(nodeIndex);
    }
    public Transform GetTransform(int nodeIndex)
    {
        return _meshGen.GetGroupTransform(nodeIndex);
    }
    public Vector3 GetAvgGroupCenter(int nodeIndex)
    {
```

```
        var groupCenter = _voronoiGroups.GetGroupCenter(nodeIndex);
        var offset = _meshGen.GetGroupTransform(nodeIndex).position + new
Vector3(groupCenter.x,0,groupCenter.y);
        return _voronoiGroups.GetGroupAverageCenter(nodeIndex) + offset;
    }
    public int GetSize(int nodeIndex)
    {
        return _voronoiGroups.GetGroupSize(nodeIndex);
    }
    public int GetLastIndex()
    {
        return lastItemIndex;
    }
}

// Клас який зв'язаний з обчислювальним шейдером та
    // відповідає за побудову полігональної сітки з даних вокселів
    public class MarchingCubes: IDisposable
    {
        public ComputeShader MarchingShader;

        ComputeBuffer _trianglesBuffer;
        ComputeBuffer _trianglesCountBuffer;
        ComputeBuffer _weightsBuffer;
        struct Triangle
        {
            public Vector3 a;
            public Vector3 b;
            public Vector3 c;

            public static int SizeOf => sizeof(float) * 3 * 3;
        }

        float[] _weights;
        public int[] _triCountByWeights;

        private int PointsPerChunk = 8;
        private int NumThreads = 2;
        public MarchingCubes(ComputeShader marchingShader, int chunkSize, int
threads)
        {
            MarchingShader = marchingShader;
            PointsPerChunk = chunkSize;
            NumThreads = threads;
            CreateBuffers();
        }
        public Mesh Construct(float[] values)
        {
            if (values.Length != Math.Pow(PointsPerChunk, 3)) return null;

            _weights = values;
            return ConstructMesh();
        }
        Mesh ConstructMesh()
        {
            // Задаємо буфери шейдера
            MarchingShader.SetBuffer(0, "_Triangles", _trianglesBuffer);
            MarchingShader.SetBuffer(0, "_Weights", _weightsBuffer);

            MarchingShader.SetInt("_ChunkSize", PointsPerChunk);
            MarchingShader.SetFloat("_IsoLevel", .5f);
```

```
            // Встановлюємо в буфер дані з вокселями
            _weightsBuffer.SetData(_weights);

            _trianglesBuffer.SetCounterValue(0);

            // Виконуємо програму шейдера
            MarchingShader.Dispatch(0, PointsPerChunk / NumThreads,
PointsPerChunk / NumThreads, PointsPerChunk / NumThreads);

            Triangle[] triangles = new Triangle[ReadTriangleCount()];
            _trianglesBuffer.GetData(triangles); // Отримання даних з
трикутниками

            return CreateMeshFromTriangles(triangles);
        }

        int ReadTriangleCount()
        {
            int[] triCount = { 0 };
            ComputeBuffer.CopyCount(_trianglesBuffer, _trianglesCountBuffer,
0);
            _trianglesCountBuffer.GetData(triCount);
            return triCount[0];
        }
        // Метод який відповідає за створення полігональної сітки з даних про
трикутники
        Mesh CreateMeshFromTriangles(Triangle[] triangles)
        {
            Vector3[] verts = new Vector3[triangles.Length * 3];
            int[] tris = new int[triangles.Length * 3];

            for (int i = 0; i < triangles.Length; i++)
            {
                int startIndex = i * 3;

                verts[startIndex] = triangles[i].a;
                verts[startIndex + 1] = triangles[i].b;
                verts[startIndex + 2] = triangles[i].c;

                tris[startIndex] = startIndex;
                tris[startIndex + 1] = startIndex + 1;
                tris[startIndex + 2] = startIndex + 2;
            }

            Mesh mesh = new Mesh();
            mesh.vertices = verts;
            mesh.triangles = tris;
            mesh.RecalculateNormals();
            return mesh;
        }

        void CreateBuffers()
        {
            _trianglesBuffer = new ComputeBuffer(5 * (PointsPerChunk *
PointsPerChunk * PointsPerChunk), Triangle.SizeOf, ComputeBufferType.Append);
            _trianglesCountBuffer = new ComputeBuffer(1, sizeof(int),
ComputeBufferType.Raw);
            _weightsBuffer = new ComputeBuffer(PointsPerChunk *
PointsPerChunk * PointsPerChunk, sizeof(float));
        }

        public void Dispose()
```

```
        {
            _trianglesBuffer.Release();
            _trianglesCountBuffer.Release();
            _weightsBuffer.Release();
        }
    }
```

ІГРОВА ЛОГІКА

```
// Базовий клас штучного інтелекту
public class UnitAIBase:MonoBehaviour
{
    public HealthModule Health => healthModule;

    [SerializeField] HealthModule healthModule;
    [SerializeField] protected NavMeshAgent agent;

    protected Vector3 currentDestination;

    protected virtual void Awake()
    {
        // Збереження початкової точки як ціль руху
        currentDestination = transform.position;
        healthModule.OnRecieveDamage += HealthModule_OnRecieveDamage;
    }
    protected virtual void OnDestroy() { }
    protected virtual void Update()
    {
        var diff = transform.position - currentDestination;
        // Якщо відстань між поточною і цільовою позицією більше за 2 агент
починає рух до цілі
        if (diff.sqrMagnitude < 2) return;
        UpdateAgentTarget();
    }
    protected virtual void HealthModule_OnRecieveDamage(float health, float
damage, Vector3 pos) { }
    protected virtual void UpdateAgentTarget()
    {
        if (!agent.isOnNavMesh) return;
        agent.SetDestination(currentDestination);
    }
    public virtual void SetSpawn(Vector3 spawn)
    {
        agent.enabled = false;
        currentDestination = spawn;
        transform.position = spawn;
        agent.enabled = true;
    }

}

// Клас штучного інтелекту який наближається до цілі для атаки
    public class MeleeAI:UnitAIBase
    {
        [SerializeField] LayerMask _searchLayer;
        [SerializeField] AttackModule attackModule;
        [SerializeField] protected AnimationController anim;
```

```
        private const float _targetUpdateInterval = 0.2f;
        private float _targetUpdateElapsed;

        private float _serachInterval = 1f;
        [SerializeField] private float _serachRadius = 15f;

        private float _searchElapsed;
        private Collider[] searchBuffer = new Collider[12];
        protected HealthModule _enemyHealth;
        private Quaternion? _targetRotation;
        private float _distanceForSnap = 4f;
        [SerializeField] private float _rotSpeed = 360f;
        [SerializeField] private float _maxRotSpeedMult = 2f;
         private float _rotSpeedMult = 1f;
        private bool _canAttack = true;
        private bool _canMove = true;

        private Vector3 _lastPosition = Vector3.zero;
        protected override void Awake()
        {
            base.Awake();
            attackModule.AttackPerformed += AttackModule_AttackPerformed;
        }

        private void AttackModule_AttackPerformed()
        {
            SetMovement(true);
        }

        protected override void Update()
        {
            if (!enabled) return;
            _searchElapsed += Time.deltaTime;
            _targetUpdateElapsed += Time.deltaTime;

            // Пошук ворогів
            if (_searchElapsed >= _serachInterval)
            {
                _searchElapsed = 0;
                SearchForEnemy();
            }
            // Рух до цілі
            if (_targetUpdateElapsed >= _targetUpdateInterval)
            {
                UpdateTarget();
                _targetUpdateElapsed = 0;
}
            if (_targetRotation.HasValue)
                transform.rotation =
Quaternion.RotateTowards(transform.rotation, _targetRotation.Value, _rotSpeed
* _rotSpeedMult * Time.deltaTime);

            var speed = (transform.position -
_lastPosition).magnitude/Time.deltaTime;
            _lastPosition = transform.position;
            anim.UpdateMovementSpeed(speed / agent.speed);
        }
        private void SearchForEnemy()
        {
            int overlaps = Physics.OverlapSphereNonAlloc(transform.position,
_serachRadius, searchBuffer, _searchLayer, QueryTriggerInteraction.Ignore);
```

```
            for (int i = 0; i < overlaps; i++)
            {
                if (!searchBuffer[i].TryGetComponent(out HealthModule
health)) continue;
                if
(!attackModule.TargetCategory.HasFlag(health.TargetCategory)) continue;
                _enemyHealth = health;
                break;
            }
        }
        private void UpdateTarget()
        {
            if (_enemyHealth == null) return;

            var enemyPos = _enemyHealth.transform.position;
            var dir = enemyPos - transform.position;

            // якщо ціль в зоні атаки, починаємо атаку
            if (attackModule.IsInRange(enemyPos, out var distance))
            {
                _rotSpeedMult = _maxRotSpeedMult;
                dir.y = 0;
                _targetRotation = Quaternion.LookRotation(dir);
                if (attackModule.CanAttack())
                {
                    SetMovement(false);

                    _canAttack = false;
                    if (attackModule is SyncronizedAttackModule)
                        attackModule.StartAttack();
                    else
                    anim.StartAttack(() =>
                    {
                        //Debug.Log("Attack performed", this);
                        attackModule.StartAttack();
                        SetMovement(true);
                    });
                    return;
                }
            }
            if (distance <= _distanceForSnap)
            {
                _rotSpeedMult = _maxRotSpeedMult * (1 - distance /
_distanceForSnap);
                dir.y = 0;
                _targetRotation = Quaternion.LookRotation(dir);
            }
            else _targetRotation = null;

            if (attackModule.IsInApproachRange(enemyPos, out distance))
return;

            currentDestination = transform.position + dir *
(distance/dir.magnitude);
            UpdateAgentTarget();
        }
        private void OnDrawGizmosSelected()
        {
            Gizmos.color = Color.white;
            Gizmos.DrawWireSphere(transform.position, _serachRadius);
        }
```

```
        protected override void HealthModule_OnRecieveDamage(float health,
float damage, Vector3 pos)
        {
            if (health > 0) return;
            enabled = false;
        }
        protected void SetMovement(bool enabled)
        {
            agent.isStopped = !enabled;
        }
    }

// Компонент який відповідає за здоров'я
    public class HealthModule:MonoBehaviour
    {
        public event Action<float> OnHealthChange;
        public event Action<float, float, Vector3> OnRecieveDamage;
        public event Action<float, float> OnRecieveHeal;
        public float Health => _health;
        public float MaxHealth => _maxHealth;
        public TargetCategory TargetCategory => _targetCategory;

        [Header("Health base")]
        [SerializeField] protected TargetCategory _targetCategory;
        [SerializeField] protected float _maxHealth;
        [SerializeField] protected float _health;
        protected virtual void Awake()
        {
            _health = _maxHealth;
        }

        public void GetHit(float damage, Vector3 pos)
        {
            if (!enabled || _health == 0) return;
            _health -= damage;
            if (_health < 0) _health = 0;
            OnRecieveDamage?.Invoke(_health,damage, pos);
            OnHealthChange?.Invoke(_health);
        }
        public void Heal(float value)
        {
            if (!enabled || _health == 0) return;

            _health += value;
            if (_health > _maxHealth) _health = _maxHealth;
            OnRecieveHeal?.Invoke(_health, value);
            OnHealthChange?.Invoke(_health);
        }
        public void ChangeMaxHealth(float maxHealth)
        {
            _maxHealth = maxHealth;
            OnHealthChange.Invoke(_health);
        }
    }


// Компонент який відповідає за здійснення атак та нанесення шкоди
    public abstract class AttackModule:MonoBehaviour
    {
        public virtual event Action AttackPerformed;
        public TargetCategory TargetCategory => targetCategory;
        public LayerMask HitMask => hitMask;
```

```
        [SerializeField] protected TargetCategory targetCategory;
        [SerializeField] protected float attackDamage;
        [SerializeField] protected float attackInterval;

        [SerializeField] protected LayerMask hitMask;

        protected float attackCooldown;

        private float _baseDamage;
        private float _baseInterval;
        // Метод початку атаки
        public virtual void StartAttack()
        {
            if (!CanAttack()) return;

            PerformAttack();
            attackCooldown = attackInterval;
        }
        protected abstract void PerformAttack();
        public bool CanAttack()
        {
            return attackCooldown <= 0;
        }
        protected virtual void Update()
        {
            attackCooldown -= Time.deltaTime;
        }
        // Метод який відповідає за нанесення шкоди цілям
        protected virtual bool TryApplyDamage(HealthModule health)
        {
            if (!targetCategory.HasFlag(health.TargetCategory)) return false;
            health.GetHit(attackDamage,GetAttackPosition());
            return true;
        }
        public abstract Vector3 GetAttackPosition();
        public abstract bool IsInRange(Vector3 targetPos, out float
distance);
        public virtual bool IsInApproachRange(Vector3 targetPos, out float
distance)
        {
            return IsInRange(targetPos, out distance);
        }
        public virtual void UpgradeValues()
        {
            if (_baseDamage == 0)
            {
                _baseDamage = attackDamage;
                _baseInterval = attackInterval;
            }
            attackDamage = PlayerData.GetBuffedValue(_baseDamage,
Effect.AttackDamage);
        }
    }

// Компонент ближньої атаки
    public class MeleeAttackModule : AttackModule
    {
        public override event Action AttackPerformed;
        [SerializeField] protected Vector3 attackOffset = Vector3.forward;
        [SerializeField] protected Vector3 hitBoxSize = Vector3.one;
        private Collider[] _attackBuffer = new Collider[6];
```

```
        protected override void PerformAttack()
        {
            var boxPosition = transform.position + transform.rotation *
attackOffset;
            int hits = Physics.OverlapBoxNonAlloc(boxPosition, hitBoxSize /
2, _attackBuffer, transform.rotation, hitMask,
QueryTriggerInteraction.Ignore);
            for (int i = 0; i < hits; i++)
            {
                if (!_attackBuffer[i].TryGetComponent(out HealthModule
module)) continue;
                TryApplyDamage(module);
            }
            AttackPerformed?.Invoke();
        }
        private void OnDrawGizmosSelected()
        {
            var boxPosition = transform.position + transform.rotation *
attackOffset;
            Gizmos.color = Color.red;
            //Gizmos.matrix = transform.localToWorldMatrix;
            Gizmos.DrawWireCube(boxPosition, hitBoxSize);
        }
        public override Vector3 GetAttackPosition()
        {
            return transform.position;
        }

        public override bool IsInRange(Vector3 targetPos, out float distance)
        {
            var diff = (transform.position + transform.rotation *
attackOffset) - targetPos;
            distance = diff.magnitude - hitBoxSize.z/2;
            return distance <= 1f;
        }
    }
```