Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

УДК 004.432.2 *На правах рукопису*

ВИПУСКНА КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

Програмне забезпечення процесуального генерування відкритого світу в ігрових додатках

Спеціальність 121 «Інженерія програмного забезпечення»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Студент
ІПЗ-41_____ / О.П.Мірошник /
(підпис, дата)
Науковий керівник
к.т.н., доц._____/ О.А.Курченко /
(підпис, дата)

Допускається до захисту з питань нормоконтролю
Завідувач кафедри
д.т.н., проф._____/О.С.Бичков/
(підпис, дата)

Київ - 2022

Рішенням Екзаменаційної комісії

випускна кваліфікаційна робота студента

Мірошника Олексія Павловича

захищена з оцінкою

_____

Голова Екзаменаційної комісії

Бондарчук Андрій

Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій
Освітньо-кваліфікаційний рівень магістр
Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ:
Завідувач кафедри програмних систем і технологій
_____ (О.С. Бичков)
“___”_____ 2022 р.

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Мірошнику Олексію Павловичу

1. Тема випускної кваліфікаційної магістерської роботи: “Програмне забезпечення процесуального генерування відкритого світу в ігрових додатках” затверджена наказом вищого навчального закладу від 29.11.2022 р. №

2. Строк подання студентом роботи: 12 червня 2022 р.

3. Вихідні дані до проекту (роботи): Програмне забезпечення процесуального генерування відкритого світу з можливістю модифікації та його дослідження.

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідити уже існуючі алгоритми оптимізації та генерування місцевості.
2. Провести порівняльний аналіз мов програмування для визначення найефективнішої.
3. Описати архітектуру програмного продукту у UML діаграмі.
4. Розробити програмний продукт.
5. Зробити імперичний аналіз програмному продукту, визначити швидкість роботи програми.

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

- Скріншот з гри Minecraft (рис. 1.2, ст. 10)
- UML діаграма головних класів (рис. 2.1, ст. 14)
- UML діаграма головних методів класів (рис. 2.2, ст. 15)
- Приклад нестандартних блоків у грі Minecraft (рис. 3.1, ст. 17)
- Текстурна розгортка блоку трави (рис. 3.2, ст. 20)
- Чанк (рис. 3.3, ст. 21)
- Ambient Occlusion (рис. 3.4, ст. 25)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
I.		Курченко О.А.	Мірошник О.П.
II.		Курченко О.А.	Мірошник О.П.
III.		Курченко О.А.	Мірошник О.П.

7. Дата видачі завдання 01 лютого 2022 р.

Керівник Курченко О.А. _____ (підпис, дата)

Завдання прийняв до виконання Мірошник О.П _____ (підпис, дата)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Затвердження теми роботи	15.12.2021	виконано
2	Аналіз концепцій та алгоритмів	15.01.2022	виконано
3	Вивчення категоризаційних концепцій в алгоритмах кластеризації даних	20.02.2022	виконано
4	Розробка алгоритмічної моделі	10.03.2022	виконано
5	Опис розробленого алгоритму	15.03.2022	виконано
6	Програмна реалізація методу категоризації	18.03.2022	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	19.05.2022	виконано
8	Передзахист	20.05.2022	виконано
9	Захист	25.05.2022	виконано

Студент-магістр _____ Мірошник О.П.

(підпис, дата)

Керівник роботи _____ Курченко О.А.

(підпис, дата)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

UML – Unified Modeling Language

МРТ – Магнітно-резонансна томографія

FIFO – First in first out

GPU – Graphics processing unit

PVS – Potentially Visible Sets

CPU – Central processing unit

СЗПК – Середнє значення пропуску кешу

OpenGL – Open Graphics Library

SSAO – Screen space ambient occlusion

GUI – Graphical user interface

VBO – Vertex buffer object

RAM – Random-access memory

VRAM – Video Random-Access Memory

NPC – Non-player character

GLSL – OpenGL Shading Language

АНОТАЦІЯ

Випускна кваліфікаційна магістерська робота: 65 с., 25 рис., 2 табл., 1 додаток, 18 джерел.

Тема: Програмне забезпечення процесуального генерування відкритого світу в ігрових додатках.

Об’єкт дослідження: 3D воксельні світи.

Мета роботи: створення програмного забезпечення для генерації різноманітного рельєфу у воксельних іграх.

Предмет дослідження: процесуальні алгоритми генерації рельєфу і рендерингу воксельних світів.

Результати дослідження: було створено програмне забезпечення, яке швидко генерує різноманітні воксельні ландшафти з можливістю змінювати та досліджувати їх.

Висновок: Були проаналізовані алгоритми генерації місцевості і алгоритми оптимізації рендерингу, були розроблені воксельної движок, алгоритм генерації місцевості і алгоритм оптимізації рендерингу, було розроблено програмне забезпечення, яке пропонує вирішення проблеми швидкої генерації місцевості за допомогою псевдовипадкових шумових функцій і дає можливість його досліджувати та модифікувати.

ANNOTATION

Final qualifying master's thesis: 65 pages, 25 images, 2 tables, 1 addition, 18 references.

Topic: Open world procedural generation software in game applications.

Object of research: 3D voxel worlds.

Purpose: creating software to generate a variety of relief in voxel games.

Subject of research: procedural algorithms of terrain generation and rendering of voxel worlds.

Research results: was created a software which quickly generates various voxel landscapes with the ability to modify and explore.

Conclusion: Terrain generation algorithms and rendering optimization algorithms were analyzed, voxel engine, terrain generation algorithm and rendering optimization algorithm were developed, was developed a program that offers a solution to the problem of fast terrain generation using pseudo-random noise functions and allows to explore it.

ВСТУП

Актуальність роботи

Поширена проблема серед воксельних ігор - це забезпечення гравця цікавим середовищем для дослідження. В таких іграх для генерації світу використовують шумові функції, але такі алгоритми погано підходять для створення чогось більш складного, чим локації з простим рельєфом. Тому вдосконалення генерації місцевості у воксельних світах є актуальною прикладною задачею.

Мета і задачі дослідження

Метою магістерської роботи є розробка програмного продукту для швидкої генерації місцевостей з можливістю їх модифікації і дослідження.

Досягнення мети включає розв'язання таких **задач**:

1. Дослідження вже існуючих алгоритмів оптимізації та генерування місцевості.
2. Проведення порівняльний аналіз мов програмування для визначення найефективнішої.
3. Опис архітектури програмного продукту за допомогою UML діаграм.
4. Розроблення програмного продукту.
5. Проведення імперичного аналізу програмному продукту, визначення швидкісті роботи програми.

Об’єктом дослідження є 3D воксельні світи.

Предметом дослідження є процесуальні алгоритми генерації рельєфу і рендерингу воксельних світів.

Методи дослідження

Під час виконання досліджень та прийнятті рішень використовувались такі **підходи та методи**: методи чисельних обчислень, методи когерентних шумових функцій, теорія оптимізації.

Наукова новизна отриманих результатів

При дослідженні було використано абсолютно новий алгоритм генерації ландшафту за допомогою псевдовипадкових чисел та було отримано новий алгоритм оптимізації, який був побудований на алгоритмах occlusion culling та frustum culling. Основним нововведенням було використання цих алгоритмів у програмному продукті для отримання більш швидких результатів.

Практичне значення одержаних результатів

Одержаний програмний продукт може бути використаний у сучасних проектах комп'ютерної графіки або іграх, які використовують технологію воксельного рендерингу.

Особистий внесок студента

Основним результатом є програмний продукт для генерації цікавого ландшафту з можливістю його модифікації. Основні частини вихідного коду можна знайти у Додатку А.

Публікації

Співавторство у Навчальному посібнику по інструментам командної розробки по темі: «Розподілений інструмент управління вихідним кодом Mercurial».

Структура та обсяг роботи

Робота викладена на 54 сторінках друкованого тексту, який складається із вступу, чотирьох розділів, висновків, списку використаних джерел (18 найменувань). Робота містить 2 таблиці, 24 рисунки та 1 додаток, обсягом 15 стор.

РОЗДІЛ 1
ВОКСЕЛЬНІ СВІТИ

Воксель - це тривимірний еквівалент пікселя. Починаючи з найдавніших днів досліджень комп'ютерної графіки, великі вимоги до пам'яті обмежували використання вокселів навіть у грубих сценах, тому вокселі історично знайшли лише обмежене застосування. Основне використання було в області медичної візуалізації, де представлення вокселів зазвичай використовувалось для результатів МРТ-сканувань, тощо. Вокселі також пробували у відеоіграх у різний час із неоднозначним успіхом. Одним із перших застосувань вокселів було відтворення місцевості у відеоігри 1992 року Comanche Maximum Overkill. Використовуючи підхід викидання променів (raycast) у двовимірну карту висот дозволяло тоді досягти набагато вищої графічної вірності місцевості, ніж використання воксельного рендерингу. Проте, поява апаратного прискорення, нових, та більш потужних відеокарт, означало, що растеризатор полігонів почне стрімко розвиватися, а інтерес до альтернативних методів рендерингу стрімко підріс у розробників.

Оскільки обсяг пам’яті комп'ютера та потужність обробки, доступних для програм, з роками зросли, воксельський підхід знову став більш привабливим. В пакетах комп'ютерної графіки для кіно і телебачення, таких як AutoDesk Maya ™, вокселі тепер інтегровані для різних робочих процесів з об'ємними ефектами, таких як моделювання хмар і рідини. Як такі вони використовувались для створення ефектів для великих бюджетних фільмів, таких як "Властелин кілець" та "Післязавтра" (Lord of the Rings, Day After Tomorrow).

Починаючи з відносно невідомого InfiniMiner і популяризуючись надзвичайно успішним Minecraft (див. рис. 1.1 Скріншот з гри Minecraft), вокселі тепер вдалося породити цілий новий жанр відеоігор, що називаються іграми "Воксельний Світ". У цих іграх весь світ побудований з кубиків вокселів. Ці воксели перетворюються в полігони для рендеринга на сучасних графічних картах, які оптимізовані для перетворення трикутників в лінію

сканування. Воксель у цьому випадку описує єдиний елемент у світі, так, як це робить об’єкт в 2D-грі, наприклад, Super Mario Bros. Хоча вокселі часто перетворюють безпосередньо в куби, вони можуть бути перетворені на інші форми (наприклад, як на факели або сходи у Minecraft).

Рисунок 1.1 Скріншот з гри Minecraft

Привабливість ігор на основі вокселів полягає в тому, що вони, на відміну від традиційних ігор, дозволяють здійснювати повні маніпуляції з боку гравця. Наявність повністю руйнівного світу вже давно є свого роду святим Граалем для комп'ютерних ігор. Був опробований ряд підходів, зазвичай путем просто розбиття деяких об'єктів і/або деформації карт висот. Всі ці методи були досить обмежені. Вокселі не тільки роблять весь світ руйнівним у связній манері; вони також роблять мир конструктивним. Тепер гравець може фактично побудувати все, що він хоче у своєму власному світі.

Інші ігри також намагалися надати конструктивний контроль гравцеві, але на основі сітки макет воксельного світу також має кілька великих переваг. По-перше, це ставить абсолютне обмеження на геометричну складність, яку

можна досягти в будь-якій даній області. На противагу цьому будь-який підхід, який дозволяє гравцеві розміщувати довільні моделі у світі, створює потенціал для створення сцен занадто складних, щоб відображати із задовільною швидкістю. По-друге, концепція укладання блоків один на одного дуже проста і легко зрозуміла. Таким чином, навіть маленькі діти не відчувають особливі труднощі в створенні складної архітектури в грі, як Minecraft, в той час як інші методи будівництва, як CSG (конструктивна стереометрія, тобто операції Boolean) навіть кваліфікованим дорослим людям часто важко правильно використовувати.

Гнучкість і простота приходять з ціною: графічний світ представлений в низькій роздільній здатності (пікселізований). Елементи розбиті на великі шматки, і велика частина світу будуються з кубиків, як правило, в масштабі близько 1м³ по відношенню до гравця. Такі куби зазвичай будуються з нормалізованими по обличчю нормалями, ще більше посилюючи різкий і блоковий зовнішній вигляд світу. Ця грубе поява може відсторонити деяких людей, але роздільна здатність і природа блоків цих воксельних світів є частиною того, що робить побудоване в них настільки доступним. Або це може бути просто існуючим питанням розміщення або видалення блоків.

Для того, щоб можна було надати гравцеві великий світ для дослідження, більшість таких ігор використовують процедурні методи для генерації світу. Насправді, створюючи нові частини світу на ходу, такі ігри, як Minecraft, здатні забезпечити гравця світами, які фактично безмежні.

Це дуже бажана особливість у таких іграх з кількох причин. По-перше, оскільки вони, як правило, пісочниці та зосереджені на будівництві, нескінченний світ означає нескінченні ресурси та необмежений простір для будівництва. Крім того, некерований характер гри означає, що індивідуальний гравець повинен знаходити сенс в ігровому просторі, і один з можливих підходів для гравця - це піти досліджувати світ. В цьому випадку більший світ виливається безпосередньо в більше годин гри, які можна отримати від цієї діяльності.

Звичайно, щоб зробити дослідження цікавим для гравця, світ повинен представити достатню різноманітність, щоб привернути його увагу. Як крайній приклад, просто використання шуму Перліна для створення карти висоти для світу, а потім його заповнення, створить світ з нескінченною різноманітністю. Однак у цьому випадку гравець помітить спрощену природу світового генератора і втратить інтерес до вивчення світу майже відразу. Тож важливо тоді наповнити світ різноманітним цікавим змістом. З цією метою такі ігри будуть використовувати різноманітні алгоритми для створення безлічі місцевостей, такі як гори, пагорби, тундри, пустелі, океани, тощо.

Метою цього проекту є створення воксельного двигуна, укомплектований процедурним генератором місцевості, подібним до того, який використовується у воксельних іграх. Генератор рельєфу місцевості, розроблений у цьому проекті, здатний створювати різноманітні правдоподібні, добре адаптовані до місцевості рельєфи.

РОЗДІЛ 2
ДИЗАЙН ПРОЕКТУ

2.1 Проблема

Поширена проблема серед воксельних ігор - забезпечення гравця цікавим середовищем для дослідження. Оскільки використовувані алгоритми генерації світу використовують псевдовипадкові процеси у вигляді когерентних шумових функцій, вони погано підходять для створення чогось більш складного, чим місцевостей з простою зовнішністю.

Для того, щоб отримати відчутний результат, проект потрібно було розділити на три етапи: основний двигун вокселів (див. Розділ 3.1 Основи воксельного світу), початкове генерування місцевості (див. Розділ 3.2 Генерація місцевості), та оптимізація (розглянуто у розділі 4 Оптимізація процесу рендерінгу).

Базовий воксельний двигун був розроблений з нуля, використовуючи мову C++, графічний інтерфейс OpenGL та декілько математичих бібліотек. Minecraft був використан як натхнення для багатьох деталей реалізації, таких як використання чанків (див. Розділ 3.1.3 Чанки) та засобів освітлення (детальніше див. Розділ 3.1.4 Освітлення).

Генерація рельєфу використовує ряд псевдовипадкових шумових функцій, об'єднаних разом для створення карти висот і заповнення світу блоками (обговорюється в розділі 3.2.1 Бібліотека Libnoise).

Процес оптимізації стадій рендерингу був зосереджений на прискоренні відтворення, підвищенні продуктивності та мінімізації обсягу обчислень.

2.2 Опис діаграми класів

На діаграмі класів, зображеній на рис. 2.1) основним класом є клас Engine, який є головним менеджером відтворення зображення. Від нього залежать такі класи, як Camera, Shader, ThreadPool, та WorldController.

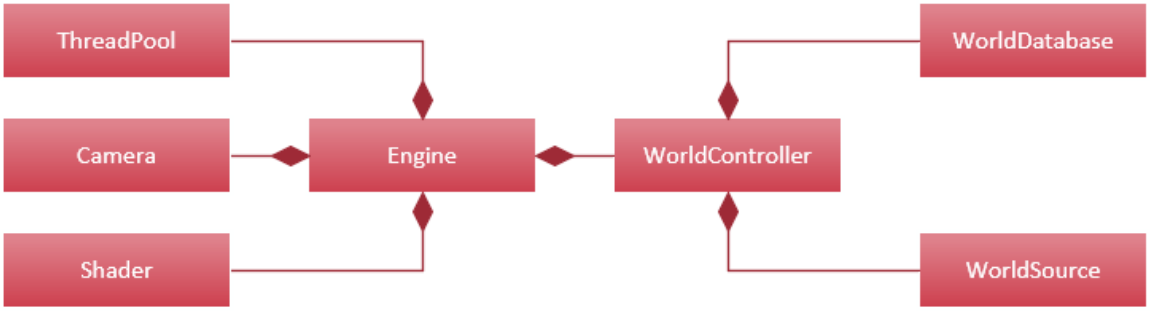

Рисунок 2.1 UML діаграма головних класів

Клас ThreadPool відповідає за багатопоточність виконання будь-яких завдань.

Клас Camera відповідає за приймання інформації о розтошуванні гравця у грі з классу Engine та відновлення позиції камери відповідно цього.

Клас Shader виконує ініціалізацію шейдерів інтерфейсу OpenGL для більш зручної роботи з ними і передачі інформації на відеокарту.

Клас WorldDatabase працює як локальна база даних для схову інформації про світ, змінені блоки в ньому та позиції гравців.

Клас WorldSource служить класом генерації світу. Він використовує псевдорандомні числа для формування карти висот, за допомогою якої і індексів блоків у неї, формується місцевість.

Клас WorldController відповідає за менеджмент чанків, секцій та блоків. При зміні якогось блоку у світі, чанк в якому він знаходиться, добавляється в чергу на оновлення.

Клас Engine відповідає за менеджмент івентів, налаштування, юзер інтерфейс та оновлення картинки на екрані користувача.

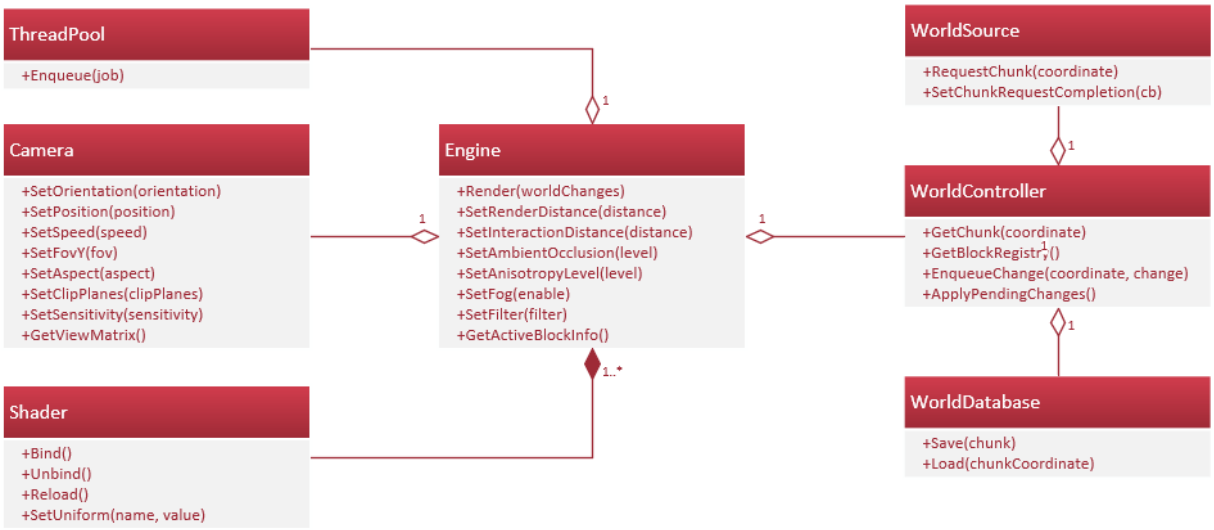

Рисунок 2.2 UML діаграма головних методів класів

РОЗДІЛ 3
ПРОЦЕСУАЛЬНЕ ГЕНЕРУВАННЯ СВІТУ

3.1 Вступ

Ця частина роботи докладно пояснює, як був створений двигун воксельного світу.

Двигуни воксельних світів, як правило, поділяють ряд елементів. В основі вони мають ряд різних типів блоків, які можна розмістити у світі у воксельних клітинках (елементи у звичайному 3-мірному масиві сітки). Сам масив зберігання вокселів зазвичай розбивається на більші елементи, які називаються чанками (див. розділ 3.1.3 Чанки), хоча можливі й інші способи зберігання, такі як воксельні дерева октантів.

Блоки зберігається у воксельній сітці, потім перетворюються в полігони[1] за допомогою деяких засобів для відображення. Перетворення полігону може бути таким же простим, як створення форми коробки розміром одного вокселя (це відноситься для більшості типів блоків у Minecraft, наприклад), створюючи більш складну сітку в розташуванні вокселя (наприклад, сходи або факел у Minecraft) див. рис. 3.1.

Для того, щоб заселити світ блоками, цікавий гравцю, використовується процедурні алгоритми. Хоча специфіка генерації світу відрізняється від воксельних ігор, вони зазвичай використовують якусь когерентну функцію шуму як основу.

[1] Полігони або багатокутники застосовуються у комп'ютерній графіці для побудови зображень, які мають виглядати як тривимірні об'єкти. Трикутники та полігони зазвичай (але не завжди) виникають при побудові поверхонь, коли для заданих вершин рендериться каркасна модель. Ця побудова зазвичай передує моделі з затіненнями і є етапом побудови зображення у комп'ютерній анімації. Кількість полігонів є характеристикою того, скільки полігонів потрібно рендерити на один кадр.

Рисунок 3.1 Приклад нестандартних блоків у грі Minecraft

3.2.1 Воксельний двигун

У цьому розділі розглядається основна архітектура воксельного двигуна. Для світового масштабу було вирішено дотримуватися розміру 1 світової одиниці, що дорівнює 1 метру. Що стосується шкали воксельної сітки, я вирішив використовувати ту саму шкалу, що і Minecraft: 1 воксельна клітина має розмір 1м³. Це було зроблено з двох причин: по-перше, це звичайний вибір у воксельних іграх (використовується в Minecraft, FortressCraft, Minetest, серед інших), і таким чином результат проекту можна було б відразу порівняти з цими популярними існуючими іграми. Друга причина - це просто те, що саме розмір забезпечує хороший баланс між світовою вірністю та продуктивністю. Половина розміру осередка до 0,5 м³ означає 8-кратне збільшення кількості вокселів, необхідних для заповнення заданої області, що означає значне збільшення використання пам'яті, обробки процесора освітлення та використання відеокарти, щоб зробити отримані сітки (все приблизно лінійно зі збільшенням кількості вокселів).

Для того, щоб користувальницький інтерфейс був чуйним під час обробки системи (наприклад, коли вона генерує місцевість під час ініціалізації),

використовувався пул потоків (див Розділ 4.4 Багатопоточне обчислення). Потік переднього плану для задач, які необхідно виконати на головному потоці і фоновий потік для обробки важкої (таких як генерація місцевості, розрахунки освітлення, оптимізація і більшість генерації сітки). Завдання можуть бути заплановані або на передньому потоці, або у фоновому, у міру необхідності. Планування завдань використовує буфер простої черги (FIFO[2]).

3D-система координат, що використовується в двигуні, така ж, як і система координат Unity3D за замовчуванням. Тобто вісь y йде вгору і вниз, а горизонтальна площина складається з осі x і z. Деякі двигуни, такі як Unreal Engine 4, використовують вісь z як вертикальну вісь, а вісь x і y описують горизонтальну площину. Це незначна деталь і вибір є лише питанням уподобань, але важливо це знати, коли деякі деталі реалізації будуть обговорені далі в цьому розділі.

3.2.2 Блоки

Для блоків було використано модель Flyweight[3] (Nystrom, 2014). Кожен блок зберігається як вказівник на екземпляр класу Block, який містить всю інформацію про блок. Менеджер блоків зберігає всі доступні екземпляри блоків (по одному для кожного типу блоку, доступного для використання в цьому світі вокселів, наприклад, бруд або камінь) і дозволяє шукати їх за іменем або ідентифікаційним номером.

Оскільки блоки будуть мати різні текстури залежно від типу, усі відповідні текстури завантажуються, коли програма запускається. Початкова програма-прототип використовувала для цього вдосконалену функцію Direct3D API, що називається текстурними масивами, і як випливає з назви, дозволяє вказати масив текстур для використання з об’єктом полігональної сітки, де індекс до масиву включений до кожної вершини.

---

[2] FIFO – First in First out, метод черги для управління пам’яттю

[3] Flyweight pattern - це модель дизайну програмного забезпечення, яка мінімізує використання пам'яті, обмінюючись якомога більше даних між подібними об'єктами. Використання посилань на спільний екземпляр об'єкта резервного зберігання даних в пам'яті зводиться до мінімуму.

Клас Block - це просто збір інформації про кожен тип блоку в двигуні, який має такі властивості:

- BlockGlobalCoord = glm::ivec3;
- BlockLocalCoord = glm::vec<3, uint8_t, glm::defaultp>;
- BlockId = unsigned int;
- Token name;
- uint32_t sideTextures[kNumSides];
- uint8_t opacity;
- uint8_t emissive;
- bool isTransparent();
- bool isOpaque();

Name, і BlockId надають способи ідентифікації блоків. Далі в цьому документі, коли потрібно посилатися на певний тип блоку, назва блоку записується в лапки. Наприклад: "Воздух", "Трава" або "Бруд".

BlockGlobalCoord та BlockLocalCoord - світові та локальні координати блока.

Opacity використовуються під час розрахунку освітлення (див. розділ 3.2.4 Освітлення), і природно, якщо встановлено значення true, це запобігає поширенню світла через цей блок.

Шість значень змінної sideTextures (TopFace, BottomFace тощо) кодують положення атласу текстури, яке потрібно використовувати для кожної сторони блоку. Для більшості блоків ці значення є ідентичними, але, наприклад, блоки "Трава" мають текстуру трави зверху, бруд на дні та перехід від трави до бруду з боків. На малюнку 3.1 показана текстурна розгортка трав'яного. Як видно, верхня сторона повністю зелена, тоді як бічні грані показують перехід від зеленої трави до коричневого бруду.

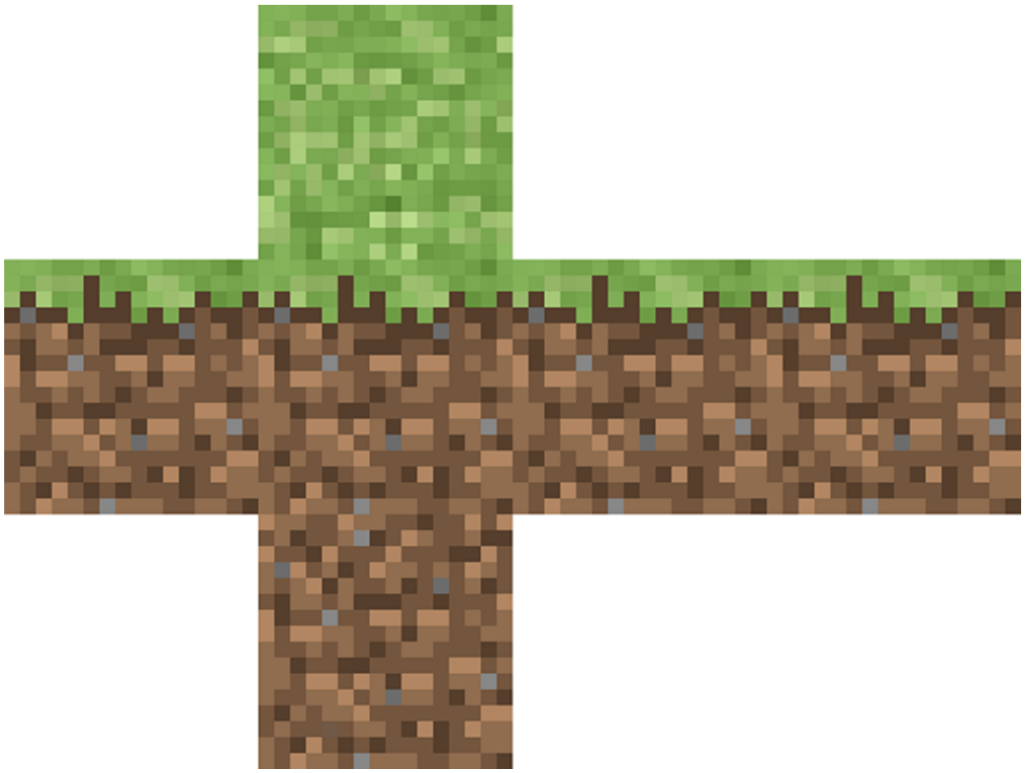

Рисунок 3.2 Текстурна розгортка блоку трави

Значення Emissive використовується для блоків, які випромінюють власне світло. Це може бути використано для факелів, наприклад. Хоча жоден блок в цьому проекті не випромінює велику кількість світла, всі вони мають низьке, але ненульове значення, встановлене для цього, щоб додати світу до навколишнього середовища (див. Розділ 3.1.4 Освітлення для отримання додаткової інформації про обчислення освітлення.).

3.2.3 Чанки

Воксельні ігри зазвичай поділяють світ на одиниці, які називаються «чанки». Розмір чанків змінюється між реалізаціями, хоча загальними є розміри 16^3 або 32^3. Minecraft використовує незначну модифікацію в цій системі. Він використовує фіксовану висоту світу в 256 блоків, при цьому один чанк - 16 × 256 × 16 блоків (див. рис. 3.3 Чанк). Кожен з чанків далі розділений на 16 секцій. Секція – шматок розміром в 16 x 16 x 16, тобто 16^3 блоків для візуалізації.

Рисунок 3.3 Чанк

Є кілька причин для поділу світу на чанки, а не розглядати його як єдине монолітне ціле. Перший полягає в тому, що він дозволяє динамічно завантажувати / вивантажувати частини світу під час руху гравця, що дозволяє отримати світи, куди більші, ніж можна намалювати одразу. Ще одна перевага полягає в тому, що при змінах у світі потрібно оновлювати лише змінені чанки. Оскільки відтворення даних освітлення та полігональної сітці - це дорога операція, це дуже бажано. Однак найбільша вигода полягає в рендерингу. Розбивши сітку місцевості на чанки, будь-який з тих, які не видно користувачеві, тобто не в режимі перегляду камери (див. Розділ 4.2. Frustum Culling), можна не відмальовувати зайві чанки до рендерингу, економлячи потужність GPU та збільшуючи частоту кадрів. Як вже згадувалося раніше, це також дозволяє нам укладати чанки по мірі необхідності вертикально. Це корисно при створенні великих гір.

Тому розмір 16^3 чанків у рендерингу світу є оптимальним варіантом, який використовується в Minecraft, і є звичайним вибором для інших воксельних ігор.

Деякі воксельні двигуни відокремлюють сховище блоків від фрагментів дисплея, наприклад, вирішуючи зберігати блоки у великому плоскому

циклічному буфері, а не як багатовимірні масиви як частину фрагментів. Це має деякі переваги від продуктивності (узгодженість кешу, менша фрагментація пам'яті), але ціною додаткової складності.

Єдине, що є спільним для всіх впроваджень у воксельному світі - це те, що положення вокселів завжди неявні. Жодні позиційні дані ніколи не зберігаються з окремими вокселями; скоріше розташування в масиві зберігання позначає їх положення у світі (тобто розташування вокселя неявне). Щоб отримати доступ до вокселя, менеджер світу спочатку обчислює позицію чанка, поділивши значення позиції на константу ChunkSize (16 у даному випадку). Потім це використовується для пошуку вірного чанку. Самі чанки зберігаються у C++ векторі (std::vector).

3.2.4 Освітлення

Освітлення в цьому проекті - це поєднання динамічного та статичного освітлення. Динамічне освітлення - це просто низьке навколишнє освітлення і спрямоване, тіньове світло для імітації сонця. Статичне освітлення здійснюється за допомогою грубої глобальної апроксимації освітлення і запису в чанк сітки вершин. Система освітлення, яка використовується аналогічна тій, яка використовується в Minecraft.

Спочатку створюється масив для світлих даних. Такого же розміру, як блок-масив для кожного фрагменту. Тому кожний воксель доступний для запису тільки один раз. Потім для початкового проходу простежується сонячне світло з верхівки світу вниз, поки не буде встановлений блок, який зупиняє світ. Це приблизно імітує пряме освітлення Сонця, припускаючи, що воно знаходиться безпосередньо над головою (тобто положення 12 годин). Найголовніше, що це дуже швидко, тому що це простий лінійний хід вниз по масиву, що зберігає значення освітленості. Більш складний алгоритм може використовувати трасування шляху для імітації різних позицій сонця, але це потребує додаткових розрахунків.

Наступним кроком є розповсюдження світла для імітації непрямого освітлення. Тут використовується простий багатопрохідний алгоритм, адаптований в Minecraft. Освітлення обмежено 17 відтінками освітленості (від 0 для областей, де немає світла, до 16 для областей при повному освітленні). Функція, яка обчислює петлі поширення світла на тривимірному масиві, що містить значення світла 16 разів, обробляє значення світла від 16 до 1. Під час кожного проходу запис світла вивчається для кожної воксельної комірки. Для кожного сусіда, який не блокує світло (тобто властивість блоку BlocksLight не встановлено), значення світла сусіда порівнюється зі значенням вокселя, який обробляється. Якщо сусідня клітинка має значення світла, нижче, ніж у поточної комірки вокселя мінус одиниця, вона має своє значення, встановлене значенням поточної комірки мінус одиницю.

Це лише дуже грубе наближення глобального освітлення, але воно дає дивовижно гарні результати, при цьому дуже швидко (11,79 секунди для розрахунку 5214 чанків або ~ 2,26 мс на чанк).

Заключна частина розрахунку освітлення полягає в застосуванні значень освітлення до вершин багатокутної сітки, що генерується з даних вокселів. Ми не можемо застосувати значення світла, обчислене для воксельної комірки, до вершин, згенерованих для цього вокселя, оскільки більшість вокселів блокує світло і так буде мати значення нуля. Натомість те, що ми хочемо знати, - це кількість світла, яке потрапляє на сторону вокселя з навколишніх воксельних осередків. Щоб отримати плавні переходи від світлого до темного, ми беремо в середньому чотири значення світла в клітинах вокселя, що оточують вершину, у напрямку до грані вокселя.

Ще один алгоритм, завдяки якому світ вокселів виглядає більш естетично, це ambient occlusion (див. рис. 3.4).

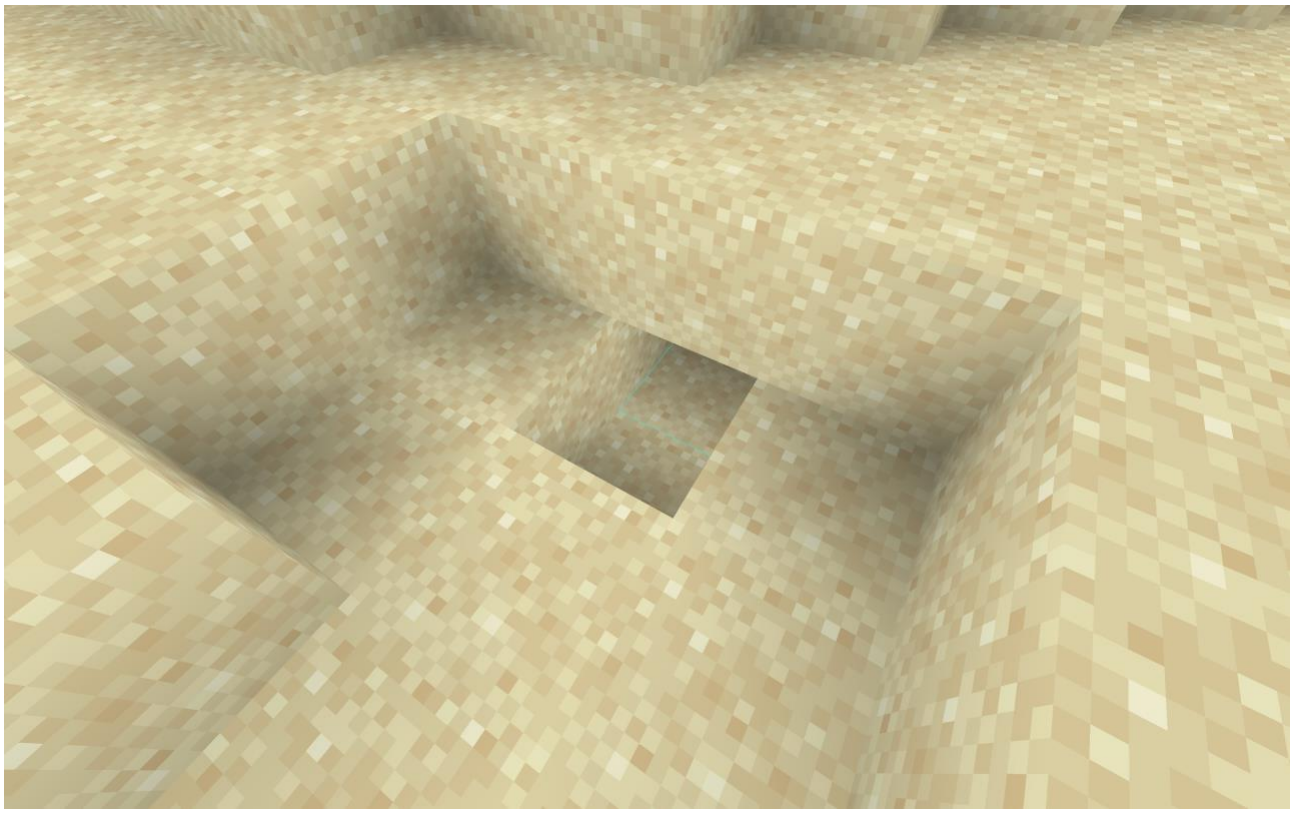

Рисунок 3.4 Ambient Occlusion

Оклюзія навколишнього середовища - це проста і ефективна методика підвищення якості освітлення у віртуальних середовищах. Основна ідея полягає в наближенні кількості світла, яке поширюється через сцену в бік точки від віддалених відбитків. В основі цієї ідеї лежить евристичний або емпіричний аргумент, і його можна обчислити, знайшовши величину площі поверхні на півкулі, яка видно з заданої точки (рис. 3.5) :

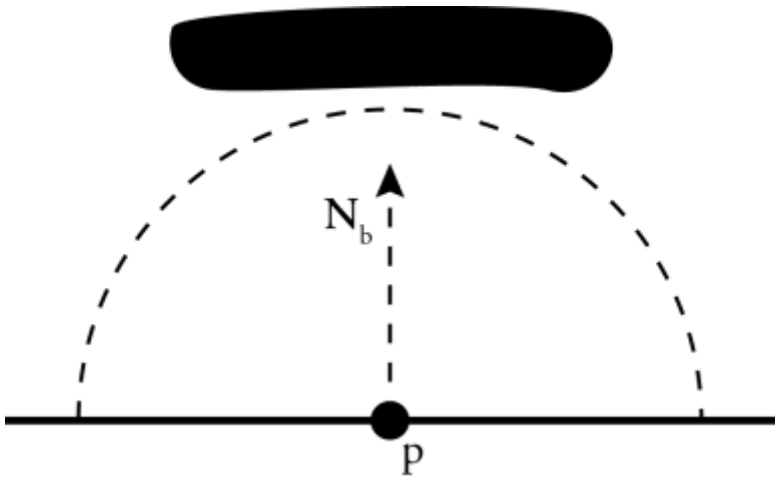

Рисунок 3.5 Оклюзія навколишнього середовища

Ambient occlusion найчастіше обчислюється шляхом побудови променів, що виходять з точки поверхні у всіх напрямках, з подальшою їх перевіркою на перетин з іншими об'єктами. Промені, що досягли фону або «неба», збільшують яскравість поверхні, в той час як промені, які перетинають інші об'єкти, не додають яскравості. В результаті точки, оточені великою кількістю геометрії, промальовується як більш темні, а точки з малою кількістю геометрії у видимій півсфері — світлими.

Додавання фактору оклюзії навколишнього середовища до сцени може значно покращити візуальну вірність, і тому багато думок перейшли до методів обчислення та наближення до оклюзії навколишнього середовища. Загалом, існує два загальних підходу до обчислення доступності:

1. Статичні алгоритми, які намагаються прорахувати оклюзію навколишнього середовища для геометрії вперед.
2. Динамічні алгоритми, які намагаються обчислити доступність із зміни або динамічних даних.

На щастя, є спосіб реалізувати оклюзію навколишнього середовища, яка є не тільки швидшою, але й незалежною. Загальна ідея полягає в обчисленні оклюзії для кожної вершини, використовуючи лише інформацію з прилеглих до неї кубів. Враховуючи це, існує симетричність 4 можливих значень оклюзії навколишнього середовища для вершини (рис. 3.6):

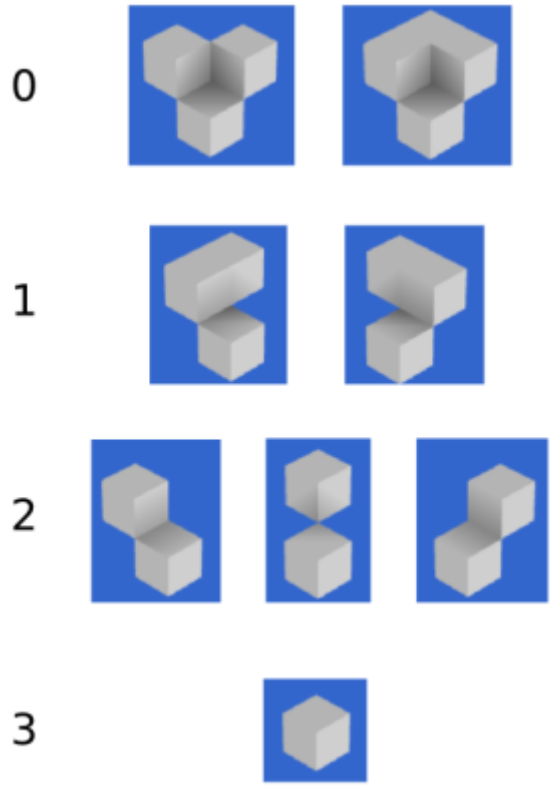

Рисунок 3.6 Чотири різні випадки оклюзії вокселя навколишнього середовища для однієї вершини.

За допомогою цієї діаграми ми можемо вивести схему. Нехай side1 і side2 дорівнюють 0/1 залежно від наявності бічних вокселів, а кут - непрозорий стан кутового вокселя. Тоді ми можемо обчислити навколишню оклюзію вершини за допомогою наступної функції:

```
function vertexAO(side1, side2, corner) {
  if (side1 && side2) {
    return 0
  }
  return 3 - (side1 + side2 + corner)
}
```

Інтегрувати вищезазначений алгоритм оклюзії навколишнього середовища насправді досить просто в систему, яка використовує жадібну мережу. Ключова ідея полягає в тому, що нам просто потрібно об'єднати грані, які мають однакове значення оклюзії навколишнього середовища в кожній з їх вершин. Це працює, тому що уздовж кожного з жадібних ребер, довжина яких перевищує 1 воксель, значення оклюзії навколишнього середовища жадної

сітки будуть постійними (вправа для читача: доведіть це). Отже, тут майже нічого робити, окрім зміни коду, який перевіряє, чи слід об’єднувати два вокселі.

Додавання навколишньої оклюзії до гри з вокселями надзвичайно просто і не вимагає великих витрат, крім незначного збільшення часу на створення сітки. Це також значно покращує візуальну якість результатів, і тому це одна з таких особливостей, що не потребують створення.

3.2.5 Перетворення на полігони

Щоб відобразити наш світ вокселів за допомогою традиційного GPU, ми повинні перетворити дані вокселів у багатокутну сітку. Наївний підхід генеруватиме дані багатокутної сітки для кожної не порожньої воксельної комірки. Це швидко створить більше полігонів, ніж може бути відображено навіть потужним графічним процесором. Натомість ми створюємо сітчасті дані лише для зовнішніх граней. На малюнку 3.7 показано два чанка, що переміщуються трохи один від одного, показуючи, як не створюються внутрішні грані.

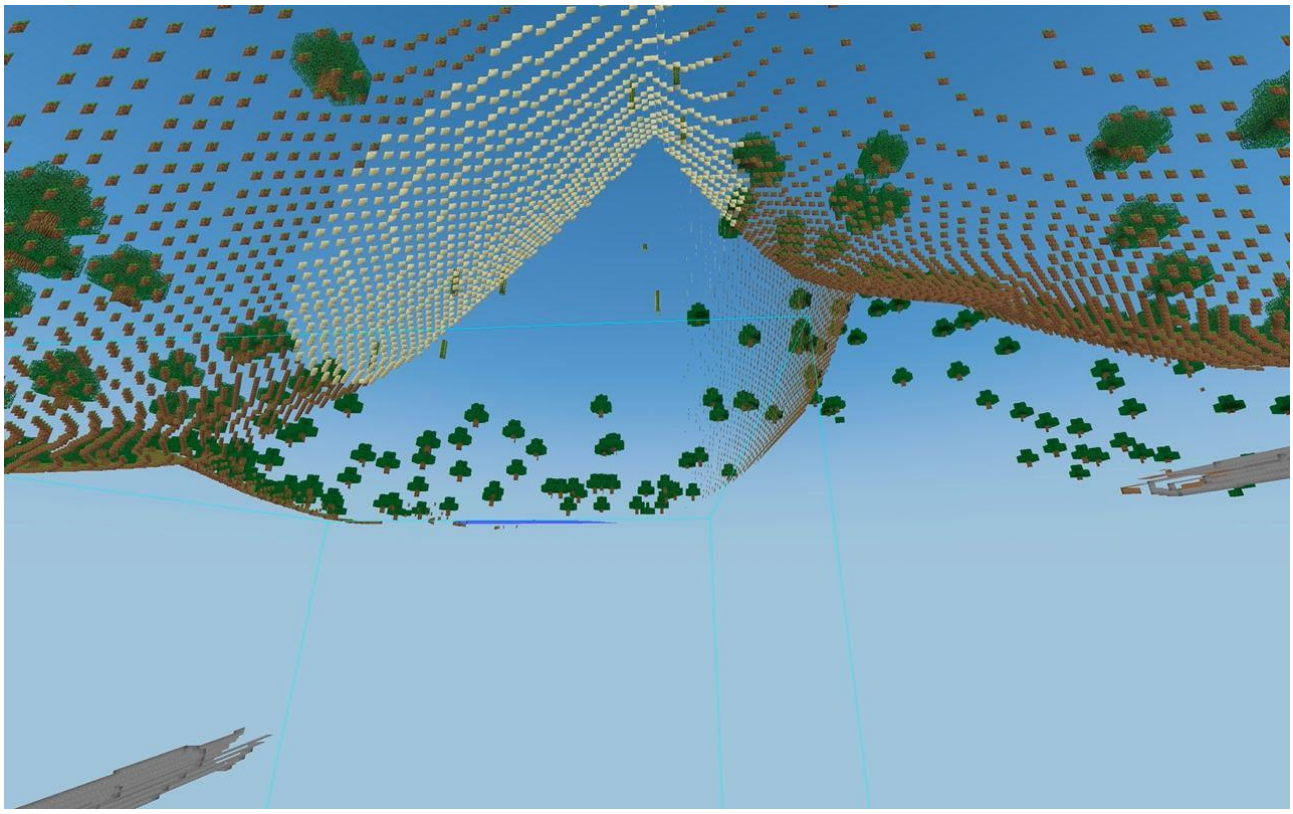

Рисунок 3.7 Демонстрація схованих граней

3.3.1 Генерація місцевості

Загальний підхід до випадкового генерування будь-якого типу місцевості полягає у використанні однієї або декількох когерентних функцій шуму (для отримання додаткової інформації про когерентний шум див. Розділ 3.2.1 Perlin Noise). Це також підхід, який двигун використовує для створення своєї місцевості. Він використовує кілька різних шумових функцій, пов'язаних між собою, щоб створити невелику різноманітність місцевості. Кожна з використаних шумових функцій описана в цьому розділі з подальшим описом того, як вони поєднуються для досягнення кінцевого результату.

Генерація місцевості складається з трьох основних етапів. Спочатку створюється карта висоти для місцевості, поєднуючи кілька функцій шуму. По-друге, комбінація карти висоти та декількох функцій шуму використовується для обчислення типу блоку для кожного блоку у світі. Нарешті, місцевість обробляється для додавання додаткових деталей, таких як дерева, вода та згладжування місцевості.

Цей проект використовує бібліотеку Libnoise, бібліотеку, яка забезпечує ряд різних когерентних функцій шуму. Перша частина цього розділу стосується Libnoise і того, що таке когерентний шум. Далі випливає опис кожної із використовуваних шумових функцій та приклад виведення з кожної. Про те, як ці функції шуму поєднуються для створення карти висоти, яка використовується в генераторі місцевості, описано далі в розділі 3.2.2 Створення карти висоти. Розділ генерації рельєфу закінчується описом того, як обчислюються блоки та описом завершальних кроків, що обмежують генерацію місцевості (додавання дерев, води та згладжування місцевості).

3.3.2 Бібліотека Libnoise

Замість того, щоб реалізувати кожну з необхідних шумових функцій та систему їх комбінування, була використана бібліотека Libnoise, розроблена спеціально для цього завдання. Libnoise поставляється з різноманітними шумовими функціями та модулями для модифікації та комбінування результатів цих функцій. Це ідеально підходить для створення такої мережі генерації шуму, необхідної для створення цікавого рельєфу. Libnoise містить більше функцій шуму, ніж потрібно для цього проекту; однак ті, що були використані, описані більш докладно нижче.

Libnoise створює "когерентний" шум, тобто створює тип псевдовипадкового шуму, де сусідні значення певним чином пов'язані, на відміну від некогерентного шуму, коли всі значення не залежать одне від одного. Когерентний шум при одному і тому ж вході (тобто, коли місце вибірки відбирається) буде генерувати одне і те ж значення виходу, невелика зміна входу призведе до невеликої зміни на виході, а велика зміна введення призведе до випадкового виходу.

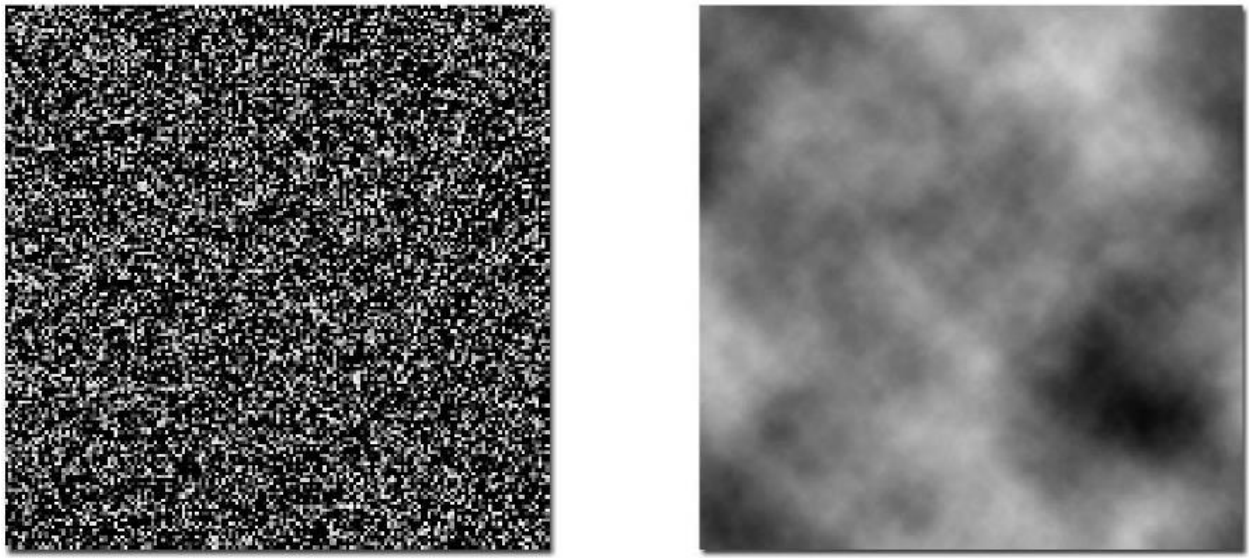

Рисунок 3.8 Випадковий шум проти когерентного шуму

На малюнку 3.8 вище показана різниця між абсолютно випадковим і когерентним шумом. Плавні градієнти, які створює когерентний шум, добре підходять для створення карти висоти для створення місцевості.

Всі модулі генератора Libnoise за замовчуванням виводять значення в діапазоні від -1 до 1, хоча цим можна керувати додатковими модулями (охопленими після шумових модулів у цьому розділі).

3.3.3 Perlin Noise

Один з найвідоміших шумових функцій, шум Перліна - це градієнтний генератор шуму, здатний генерувати різні види, такі як хмари, скло, вода та інше (Perlin, 1984) (Perlin, 1985) (Perlin & Hoffert, 1989). По своїй суті він функціонує, створюючи випадкові значення, а потім інтерполює між ними для створення градієнтів (отже, це називається градієнтним шумом). Приклад шуму Перліна показаний на малюнку 3.9 нижче.

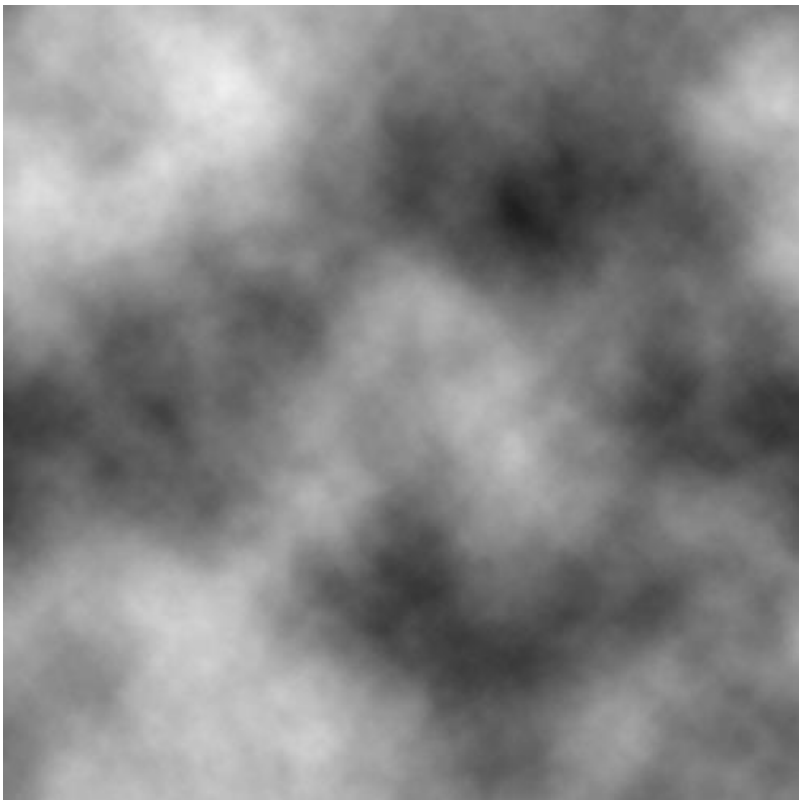

Рисунок 3.9 Perlin Noise

Шум Перліна можна керувати двома ключовими змінними: частотою та кількістю октав. Частота впливає на просторову частоту функції шуму. Це змінює швидкість змін у будь-якій частині функціонального простору. Цей ефект можна побачити на малюнку 3.10 нижче, показуючи ту саму функцію шуму Перліна з кількістю октав 2, 4, 8, 16 та 32.

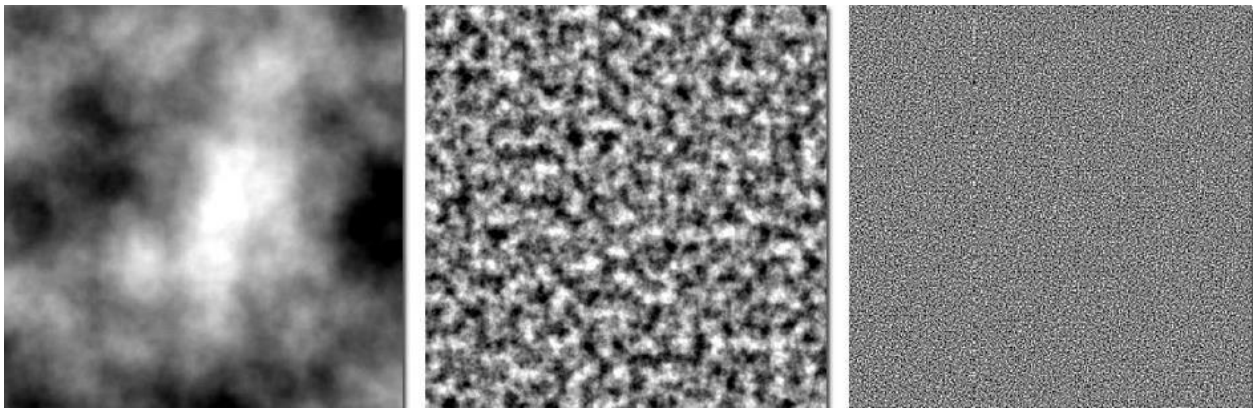

Рисунок 3.10 Вплив зміни частоти на шум Перліна

Кількість октав змінює кількість деталей, створених функцією шуму Перліна. Шум Перліна працює, додаючи разом декілька проходів над шумовою функцією на послідовно більш високих частотах, щоб генерувати додаткові

деталі. Кількість виконаних проходів контролюється октавним номером функції. Важливо зазначити, що збільшення кількості октав збільшує обчислювальну складність функції (тобто знадобиться більше часу для обчислення значень шуму). На малюнку 3.11 нижче показаний результат запуску шуму Перліна зі значеннями октави 2, 4, 8, 16 та 32. Зверніть увагу на збільшення складності зображення, оскільки кількість октав збільшується.

Рисунок 3.11 Вплив зміни кількості октав на шум Перліна

Шум Перліна використовується для більшості процесів генерації місцевості, включаючи вибір того, який тип місцевості повинен створюватися в будь-якій точці. Горбиста місцевість використовує дещо модифіковану версію шуму Перліна, який Лібнойз називає сильним шумом. Згідно з документацією на Лібнойз, це ідентично шуму Перліна, за винятком того, що кожна октава змінюється функцією абсолютного значення (Bevins, 2005a). Це створює вибагливий вигляд, придатний для хмар або у випадку цього проекту мапу висоти для пагорбів.

3.3.4 Створення карти висоти

Перший крок полягає в об'єднанні декількох модулів шуму, щоб створити карту висоти, яку може використовувати генератор місцевості, і що створює ділянки рівнин, пагорбів і гір. Цей процес додавання і масштабування шумових функцій також відомий як мультифрактальна побудова. Потужною особливістю такого підходу є те, що окремі бали можна оцінювати без посилання один на одного. Безконтекстний характер цього методу - це те, що дає змогу генерувати нові шматки світу під час руху та дає змогу воксельного світу майже безмежно рости під час дослідження гравця.

За допомогою карти висоти ви зберігаєте лише висотний компонент для кожної вершини (зазвичай у вигляді двовимірної текстури) та надаєте положення та роздільну здатність лише один раз для всього квадратика. Геометрія ландшафту генерується для кожного кадру, використовуючи шейдер геометрії або апаратну теселяцію. Карти висот - це найшвидший спосіб зберігати ландшафтні дані для виявлення зіткнень.

3.3.5 Розміщення блоків

Карта висоти створила перший головний крок у створенні місцевості. Наступним кроком є обчислення блоків місцевості за допомогою комбінації карти висоти та декількох додаткових модулів шуму.

Окрім карти висоти, у процесі генерації рельєфу місцевості використовуються ще три шумових модулі Перліна, один з частотою 0,005, 0,05 та 0,5. Це дає нам функцію шуму низької частоти для масштабних функцій, функцію середньої частоти для середніх масштабів та модуль шуму високої частоти, який слід використовувати для дрібних деталей масштабу. Вони будуть називатися модулями шуму низької, середньої або високої частоти, що залишилися в розділі. Крім того, мультифракційний модуль шуму використовується в процесі генерації місцевості (окремо від того, який використовується для створення мапи висоти).

Вихід від кожного з модулів шуму кешується достроково, так що доступ до них кілька разів не призводить до перерахунку. Це робиться тому, що обчислення функції шуму є дорогим, а деякі значення будуть використовуватися кілька разів. Це не було потрібно під час створення карти висоти, оскільки в цьому випадку кожне значення запитується лише один раз.

Весь фактичний розрахунок рельєфу місцевості виконується у функції генератора місцевості під назвою CalculateBlock(). Ця функція просто приймає позицію вокселя як значення x, y і z і повертає тип блоку. У цьому процесі використовується кілька цифр, як правило, порогових значень щодо функції шуму. Здебільшого до них доходили шляхом спроб та помилок для досягнення

бажаного вигляду місцевості. Де застосовано будь-які міркування за значеннями. Важливо пам’ятати, що значення шумових функцій генеруються в діапазоні від -1 до 1, тому, наприклад, якщо поріг встановлено більше 0, це означає, що це буде правдою приблизно половину часу. Якщо встановити більше 0.5, це було б приблизно чверті часу. При обчисленні блоку функція негайно повертається з цим значенням. Це означає, що обчислення блоків, які відбудуться далі в процесі, стають все менше ймовірними.

Алгоритм цієї функції полягає в наступному:

1. Якщо у-значення блоку менше -50, повертається блок «Камінь». Це являє собою найнижчу частину світу, про яку ми дбаємо. Все, що нижче, просто стає каменем.
2. Якщо значення у вище карти висоти та над рівнем моря (див. 4.2.7 Вода), поверніть блок типу "Повітря". Якщо воно вище значення мапи висоти, але нижче рівня моря, натомість поверніть блок "Вода".
3. Створення печер. Це обчислюється шляхом отримання значення від мультифракційного генератора шуму та повернення "Повітря", якщо значення вище певного порогу. Цей поріг встановлюється на 0.8, якщо значення у для поточного блоку вище 0, в іншому випадку воно встановлюється на 0.4. Значення генеруються в діапазоні від -1 до 1. Це робить печери досить рідкісними вище 0 і більш численними нижче 0 у світі.
4. Обчислення значення глибини. Це просто значення з мапи висоти мінус у-значення обчислюваного блоку.
5. Далі додаються пустелі. Якщо значення глибини, обчислене на останньому кроці, менше 20, а значення мапи висоти для цього блоку нижче 15, а значення модуля шуму Перлін низької частоти більше 0.42, блок кваліфікується як блок пустелі. Це означає, що пустельні райони генеруються лише на глибині 20 і лише в низьколегованих районах (гори не зроблені з піску). Якщо глибина менше 6, повертається блок «Пісок»,

інакше повертається блок «Пісковик». Тож усі пустельні райони - це просто шар піску, а за ним шар пісковика.

6. Тепер обчислюються кам’яні блоки. Для цього використовується мультифракційний модуль шуму. Якщо він повертає значення вище 0,66, повертається блок "Камінь". Це створює смуги каменю по всьому світу, щоб розбити бруд.
7. Далі обчислюється верхній шар залишку. Це або трава, або стовбур дерева, які використовуються для згодом генерування дерев (детальніше див. Розділ 4.2.8 Дерева). Спочатку ми перевіряємо, чи це верхній блок (якщо значення у дорівнює значенню висоти для цієї позиції). Якщо це так, то є шанс, що буде створений стовбур. Породжується стовбур, чи ні, визначається генератором випадкових чисел у поєднанні з низькочастотним генератором шуму Перліна. Це створює ліси, що стоншуються до країв. Точну формулу для цього можна показати у вигляді таблиці:

Поріг значення шуму	Шанс породження стовбура
>0.7	1 з 32
0.5-0.7	1 з 128
0.3-0.5	1 з 256
<0.3	1 з 1024

Таблиця 3.1 Значення шансу генерації дерев

8. Передостанній крок - обчислення гравійних блоків. Для цього використовується модуль багатофункціонального шуму, що повертає блок "Гравій", якщо значення вище 0.77. Це може здатися високим (майже 1 з 8), але пам’ятайте, що це досягається лише в тому випадку, якщо жодна з інших умов досі не була виконана.
9. Нарешті, якщо жодна з інших умов не була виконана, блок "Бруд" повертається.

Для прикладу того, як це все виглядає, поєднуючись у кінцевій місцевості, див. малюнок 3.12 нижче:

Рисунок 3.12 Приклад місцевості

3.3.6 Генерування дерев

Дерева генеруються шляхом розміщення стовбурів дерева випадковим чином замість блоків "Трава", як пояснено у розділі 3.1.2 Блоки. Самі дерева генеруються, передаючи всі шматки після завершення генерації місцевості та знаходячи всі стовбури (блоки "Дерево"), розміщені у світі, а потім генерує дерево довільної висоти (5-10 блоків) над ним. Цей процес просто передбачає розміщення більше блоків "Дерево" на блоці до потрібної висоти, а потім розкладку листяних блоків навколо верхівки дерева у приблизно півсферичній формі. Приклад того, як виглядають ці дерева, можна побачити на малюнку 3.13. Для цього потрібно обережно обробити чанки зверху вниз.

Рисунок 3.13 Дерево, створене генератором місцевості

3.3.7 Генерування води

Для створення води існує константа рівня моря, вказана у світовому генераторі. Для цього проекту спочатку було встановлено 0, але було переміщено до -5, щоб запобігти затопленню деяких низинних ділянок. Воду додають у світ у два кроки. По-перше, під час створення світу вода створюється у воксельних осередках, які знаходяться нижче рівня моря, але вище мапи висоти місцевості. Це відбувається до того кроку, який додає печер до світу, щоб печери нижче рівня моря не були затоплені автоматично.

Однак це означає, що можливо там існують райони, де печерна система перетинає рівень ґрунту в місці з водою. Після того, як завершиться початкова генерація місцевості у світі, всі шматки скануються на наявність водних блоків і застосовується рекурсивний алгоритм для заповнення води до порожніх блоків навколо та під водою. Це гарантує, що будь-які ділянки, де перетинаються водяні блоки та печерні системи, будуть затоплені правильно.

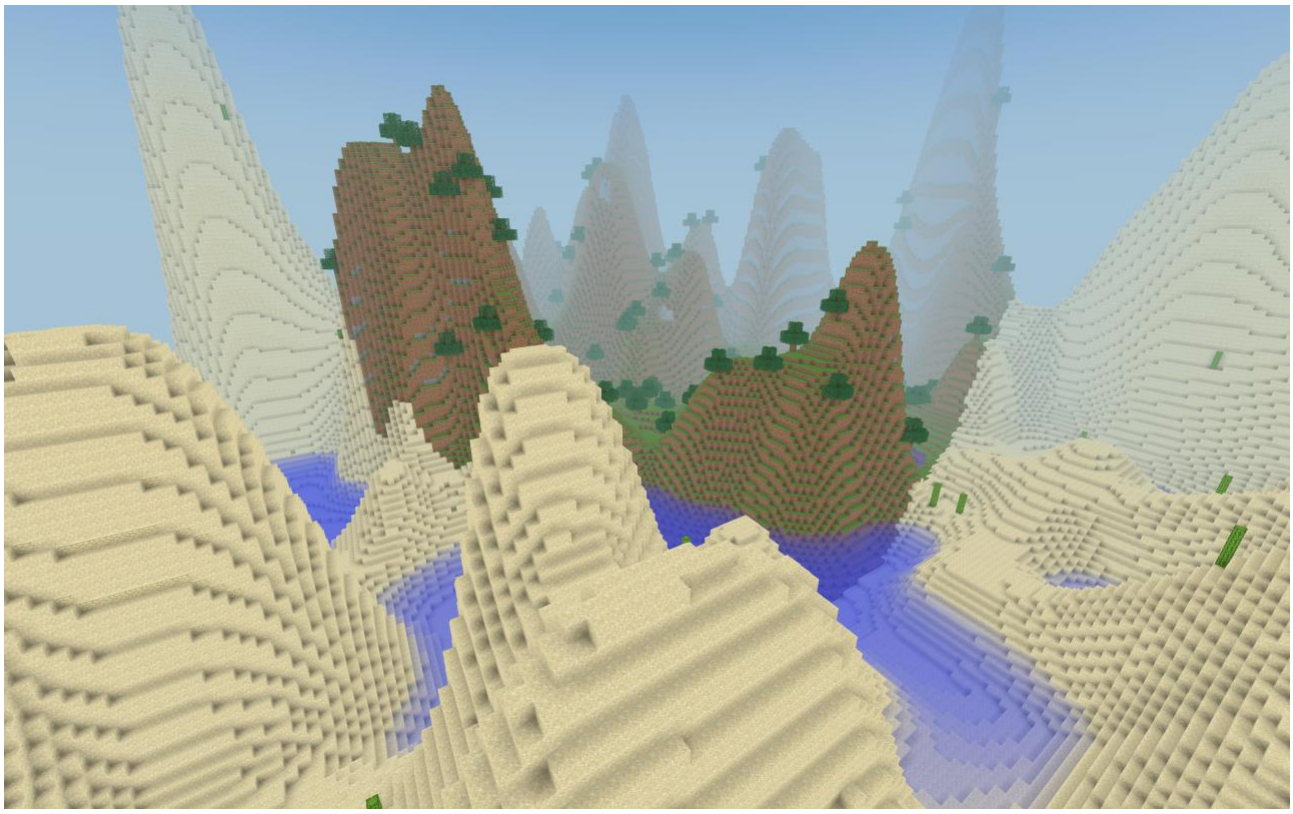

Рисунок 3.14 Приклад згенерованої води

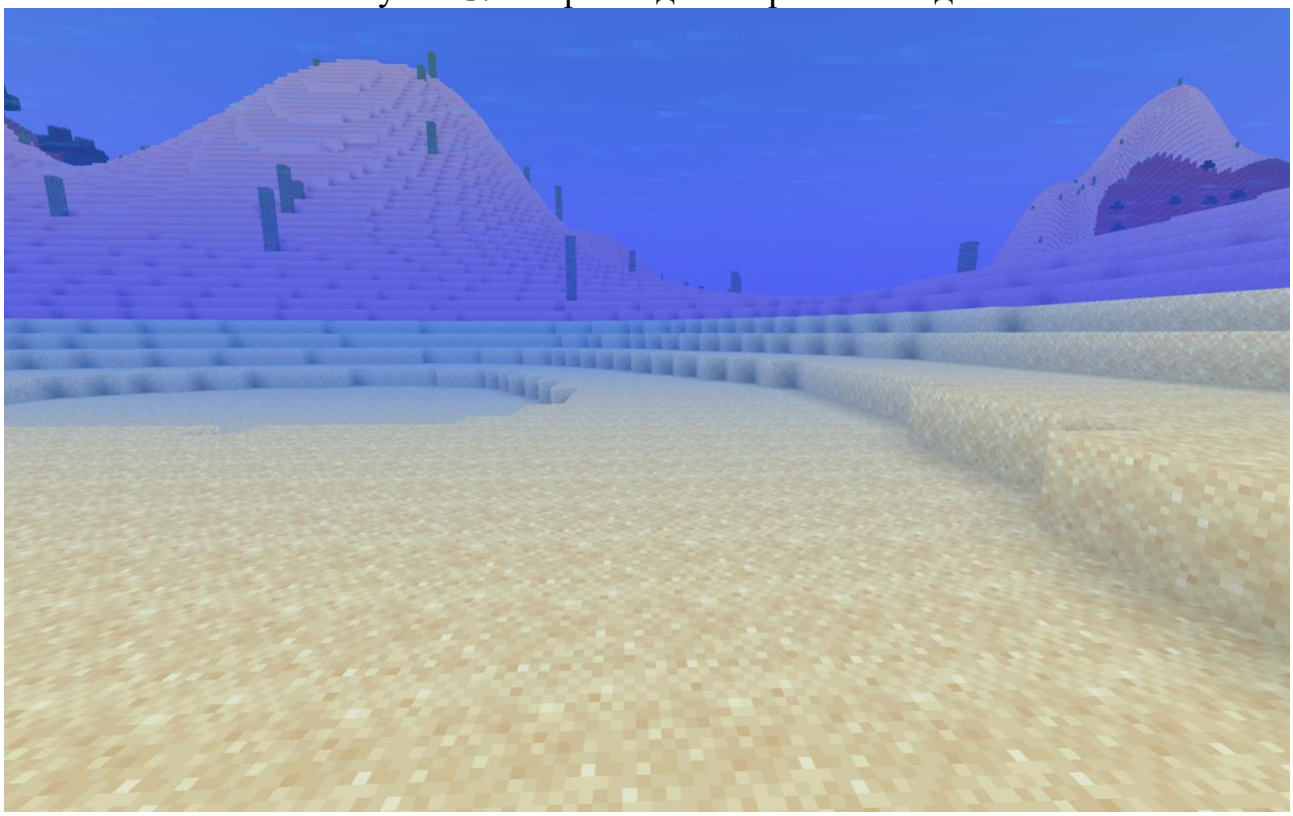

Рисунок 3.15 Вид з під води

РОЗДІЛ 4
ОПТИМІЗАЦІЯ ПРОЦЕСУ РЕНДЕРІНГУ

4.1 Вступ

Розрахунок видимості для тривимірних сцен суттєво складний. Такі поняття, як графіка аспектів, комплекс видимості або його більш проста версія, скелет видимості, є важливими для дослідження тривимірної наочності і можуть бути застосовані до сцен із кілька полігонів. Проблеми масштабування та пов'язані з чисельністю надійності створюють велику кількість спеціалізованих алгоритмів. Крім того, для систем візуалізації в реальному часі тимчасові обмеження є важливим фактором, оскільки така система не може наділіть багато часу для розрахунку видимості. Щоб впоратися зі складністю видимості, більшість алгоритмів використовують досить сильні допущення і спрощення, які відповідають вимогам лише декількох додатків. Тому ретельний вибір припущень і спрощень є важливою частиною в розробці алгоритму видимості.

Ідея сучасного алгоритму видимості - обчислити швидку оцінку тих частин сцени, які, безумовно, невидимі. Остаточне видалення прихованої поверхні проводиться за допомогою апаратного забезпечення, як правило, z-буфера. Оцінка видимість повинна бути консервативною, тобто, алгоритм ніколи не класифікує видимий об'єкт невидимим. Проте, невидимий об'єкт, все ще може бути класифікований як видимі, так як Z-буфер буде визначати його як невидимі в останній стадії рендеринга. Для великих моделей також загальна для розрахунку видимості на рівні кожного об'єкта, а не на рівні полігонів, так як:

- це економить значну кількість часу при виконанні тестів на оклюзію, що важливо для онлайн-розрахунків.
- споживання пам'яті було б проблемою для офлайн-обчислень.
- трикутники зазвичай надсилаються до графічного обладнання у трикутникові смужки. Класифікація видимості на багатокутник передбачає дорогу перебудову структур даних під час виконання.

4.2. Frustum culling

Піраміда огляду (рис 4.1) - є простим і загальним методом відбраковування, який можна застосувати практично до будь-якої моделі. Кожен вузол в сцені-графі має обмежувальний обсяг, наприклад, обмежувальний ящик, вирівняний по осі, або обмежувальна сфера. При обході сцени-графіка порівнюється граничний обсяг цих вузлів проти піраміди огляду. Якщо обмежена гучність повністю виходить за рамки піраміди , то вузол і його діти невидимі.

Перегляд фрустумового відсіву - це методика визначення об'єктів, які знаходяться поза плодової точки зору. Об'єкти поза зоною перегляду не видно (можливо, за винятком відображень) і, отже, не є необхідними для візуалізації.

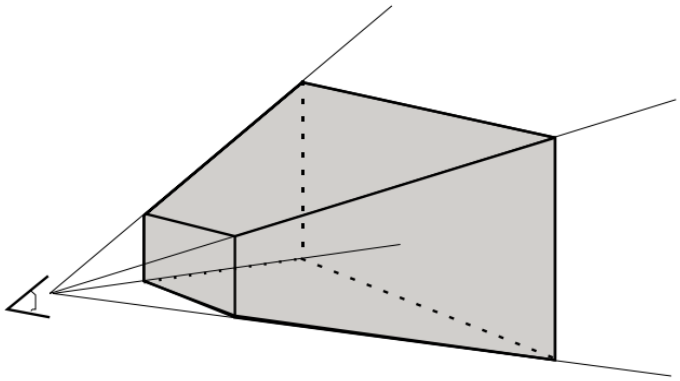

Рисунок 4.1 Піраміда огляду - область сірого кольору перед оком або камерою. Він визначається розширеннями вікон, полем зору та площинами ближніх та дальніх.

Основним вузьким місцем алгоритму зменшення видимості за допомогою регулярних сіток є велика кількість вокселів, до яких потрібно отримати доступ та оновлювати кожен кадр, оскільки ієрархії недоступні для ігнорування великих підмножин вокселів під час обходу фрустуму.

Визначення видимості фактично зроблено в обході перегряді фрустуму. Він осягає обходу вокселей, які охоплюють вид-усічений з метою виявлення окклюдером і потенційно видимих об'єктів.

Обхід вокселів виду-фрустума виконується в режимі "перегляд назад" від глядача, тому алгоритм не витрачає часу на обробку прихованих окклюдерів і може поступово визначати PVS (Potentially Visible Sets) в одному траверсі.

Для того, щоб ефективно обчислити відстань від точки спостереження до кожного воксель і, отже, виконати обхід передньої до задньої частини, ми пропонуємо використовувати метрику шахової дошки. Окрім уникнення дорогих корінних операцій, що вимагають евклідової метрики, метрика шахової дошки викликає швидке обхід правильних вокселів у напрямку, орієнтованих на осі. Оскільки лінія огляду завжди знаходиться всередині огляду-фрустума, можна поступово дискретизувати його з точки зору, використовуючи алгоритм малювання 3D-рядків, а з кожного вокселя, який містить дискретизовану зору огляду, пройти площину вокселей, які мають один і той же шахівниці відстань до воксель, що містить точку зору.

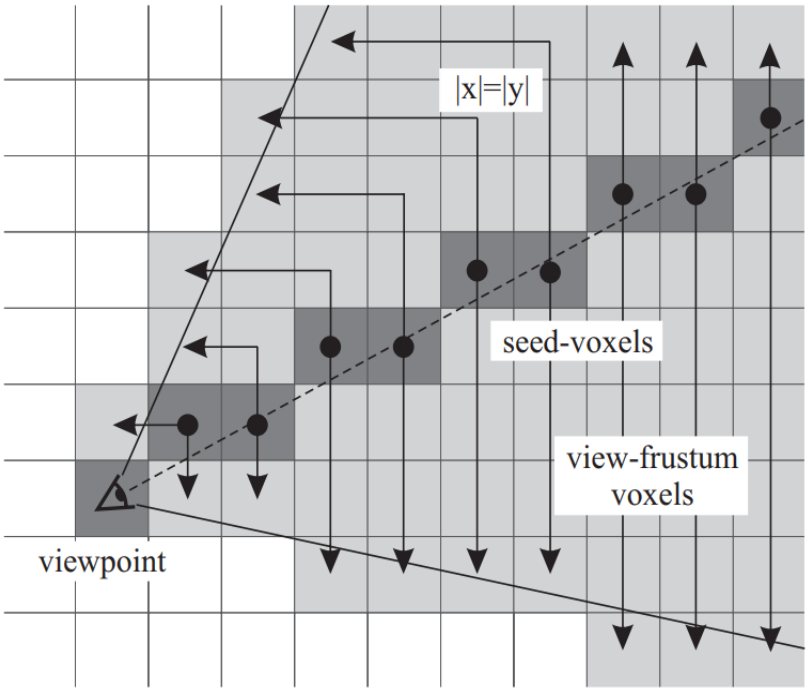

Рисунок 4.2 Перегляд фрустуму в площині xy.

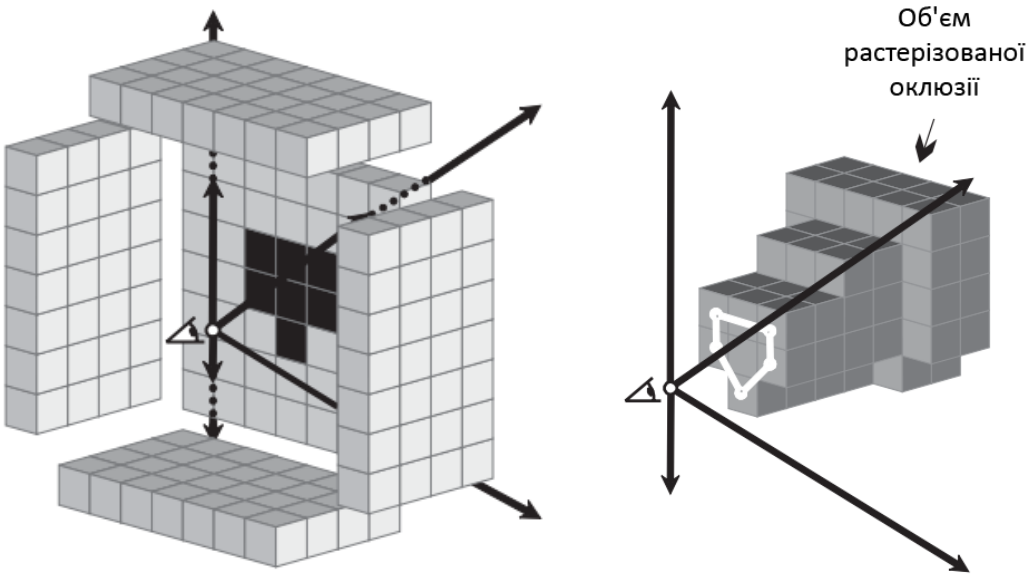

Рисунок 4.3 Зліва - розділення набору вокселів на площини вокселів, справа - піраміда огляду у воксельному вигляді.

4.3 Occlusion culling

Відключення оклюзії - це процес видалення об'єктів, які приховані іншими об’єктами з точки зору камери. Відкидання оклюзії є "глобальним", оскільки передбачає взаємозв'язок між багатокутниками і, таким чином, є набагато складнішим, ніж відключення оклюзії заднім шляхом. Історично необхідні дорогі етапи попередньої обробки, що значною мірою запобігали динамічним сценам.

Прийоми відключення оклюзії пов'язані з поняттям видимості в навколишньому середовищі і відповідають еволюції перших алгоритмів прихованого видалення поверхні. З основного алгоритму видалення прихованої поверхні (наприклад, алгоритму художника з сортуванням об'єктів відповідно до їх відстані від переглядача та z-буфера) алгоритм z-буфера є сьогоднішнім стандартом усіх графічних апаратних засобів через його прямої реалізації та зниження вартості бортової пам’яті.

Ці ранні алгоритми вирішують проблему видимості "локально" на противагу методикам відключення оклюзії, що поєднують інформацію про позицію глядача з відносним положенням ряду об'єктів навколишнього середовища, щоб відхилити візуалізацію для повних об'єктів або навіть ієрархій

сцени. Перший приклад техніки відключення оклюзії - це просте вилучення фрустуму виду, де середовище є ієрархічно розділеним і кожен вузол, який знаходиться поза полем зору спостерігача, відхиляється разом із його нащадками.

Графічні процесори, такі як сімейство PowerVR, знайдені на пристроях Apple, можуть виконувати дуже ефективне видалення прихованої поверхні, але ціною відстеження відсортованого списку пікселів екрану. Це дуже добре працює в простих сценах, але складність сцени стає пропорційною кількості фрагментів на піксель.

Виробник Notch вирішував проблему перемалювання на ПК, використовуючи вдосконалену тоді функцію OpenGL під назвою Hardware Occlusion Queries: вона малює кубічний корпус кожного куба 16x16x16, а потім перевіряє результат, чи були видимі якісь пікселі корпусу. Якщо так, то всі частини були визнані видимими та винесеними. Таким чином, виведення цих корпусів і зчитування результату в тому ж кадрі може затримати GPU, змушуючи його зупинитися і чекати процесора.

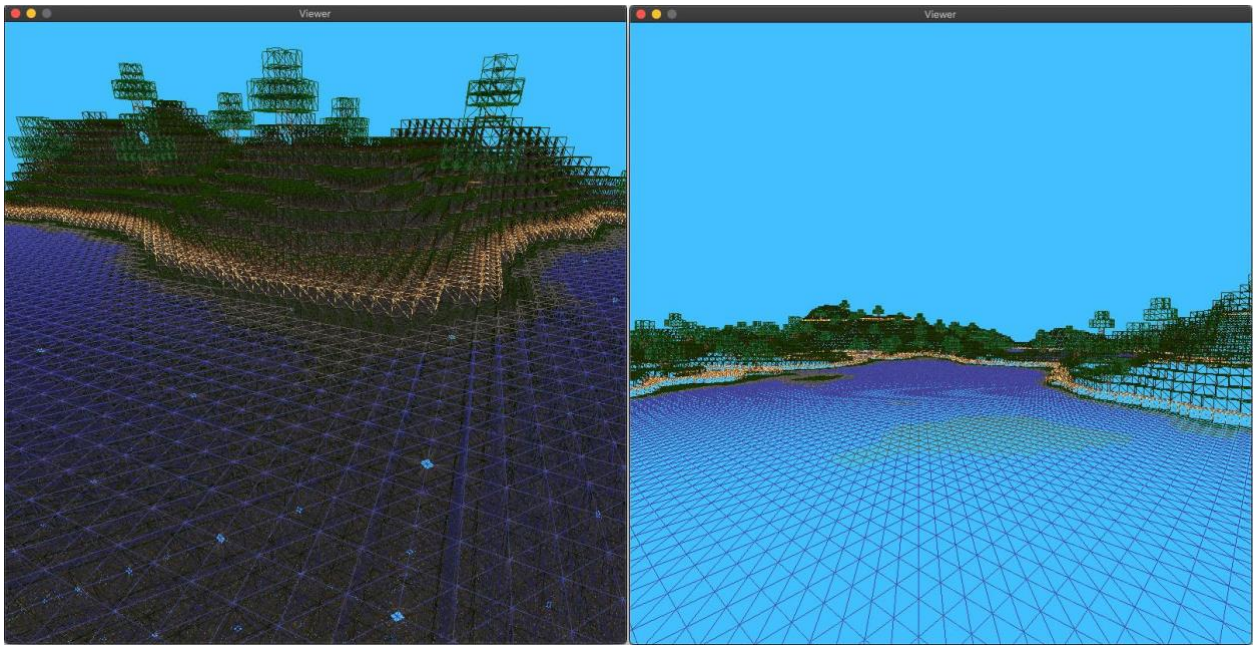

Рисунок 4.4 Зліва - програма без використання алгоритму відключення оклюції, справа - з використанням.

4.4 Багатопоточне обчислення

Thread pool - це модель дизайну програмного забезпечення для досягнення одночасності виконання в комп'ютерній програмі. Часто також його називають тиражованою моделлю робітників або екіпажу робітників, пул потоків підтримує декілька потоків, очікуючи, щоб завдання, призначені для одночасного виконання контролюючою програмою. Підтримуючи пул потоків, продуктивність моделі збільшується і дозволяє уникнути затримок у виконанні через часте створення і знищення потоків для короткоживучих завдань. Кількість доступних потоків налаштовується на доступні для програми обчислювальні ресурси, наприклад паралельну чергу завдань після завершення виконання.

Розмір пулу потоків - це кількість потоків, що зберігаються в запасі для виконання завдань. Зазвичай це налаштований параметр програми, налаштований для оптимізації продуктивності програми. Вибір оптимального розміру пулу потоків має вирішальне значення для оптимізації продуктивності.

Одна перевага пулу потоків над створенням нового потоку для кожної задачі полягає в тому, що створення потоку та знищення потоку обмежується початковим створенням пулу, що може призвести до кращої продуктивності та кращої стабільності системи. Створення та знищення потока та пов’язаних з нею ресурсів може бути дорогим процесом у часі. Однак надмірна кількість потоків, що знаходяться в резерві, витрачає пам’ять, а переключення контексту між потоками, що можна виконати, викликає штрафні показники.

Кількість потоків може бути динамічно скоригована протягом життя програми на основі кількості завдань, що очікують. Алгоритм, який використовується для визначення, коли створювати або знищувати потоки, впливає на загальну продуктивність:

- Створення занадто багато потоків витрачає ресурси і час на створення невикористаних потоків.
- Знищення занадто багато потоків вимагає більше часу пізніше, коли їх знову треба буде створювати.

- Занадто повільне створення потоків може призвести до низької продуктивності клієнта (тривалий час очікування).
- Руйнувати потоків занадто повільно може виснажувати інші процеси.

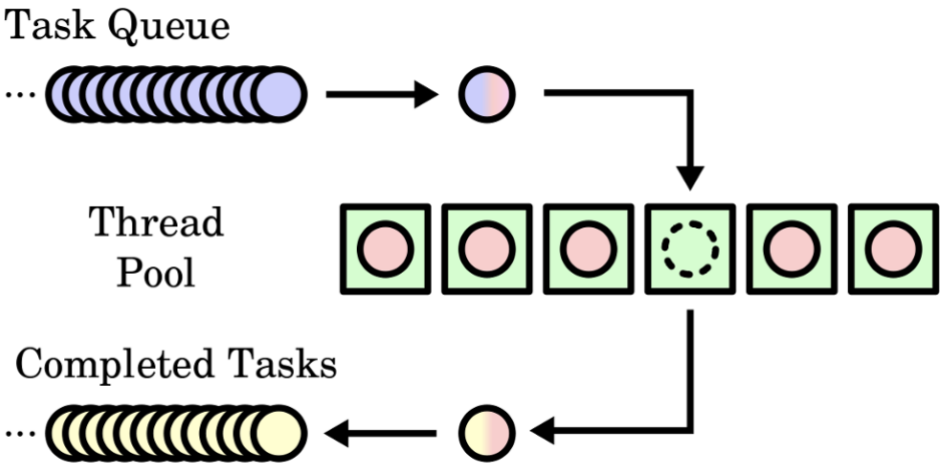

Рисунок 4.5 Зразок пулу потоків із завданнями очікування та виконаними завданнями.

4.5.1 Кешування данних

Візуалізація в режимі обчислювальної техніки стає все більш привабливою для нестандартних програм візуалізації, завдяки високій гнучкості виконання обчислювального режиму. Ці щойно розроблені конвеєри часто включають етапи обробки вершини та геометрії. У типових трикутникових сітках однакові перетворені вершини в середньому визиваються шість разів під час візуалізації. Щоб уникнути зайвих обчислень, традиційно пропонується кеш після перетворення, що дозволяє повторно використовувати результати обробки вершин. Однак традиційне кешування що не масштабується так добре, як апаратне забезпечення стає більш паралельним, і не може бути ефективно реалізовано при розробці програмного забезпечення. Цей алгоритм досліджує альтернативні стратегії повторного використання результатів затінення вершин на льоту для масової паралельної обробки геометрії програмного забезпечення.

Індексировані примітиви замінили смужки (GL_TRIANGLE_STRIP) та вентилятори (GL_TRIANGLE_FAN) як найпоширеніший візуалізацію примітиву у графічному обладнання сьогодні. Під час візуалізації кожного трикутника вершини, які він використовує, повинні бути оброблені (наприклад, перетворені на простір екрану та засвічені різними способами) перед виведенням. Індексований апарат для візуалізації зазвичай використовує невеликий кеш вершин, щоб уникнути повторної обробки вершин, які спільно використовуються між нещодавно виведеними трикутниками. Використання кешу середнього розміру (десь між 12 і 24 вершинами є загальним) зменшує навантаження на обчислювальний конвеєр вершин набагато більше, ніж у старих неіндексованих смугах і вентиляторах - часто вдвічі зменшується кількість роботи, необхідної на трикутник.

Однак кеш вершин спирається на впорядкування трикутників, оптимізованих під його характеристики. Якщо трикутники відправляються у випадковому порядку, вершини майже завжди пропустять кеш, і виходить, щл немає ніякої користі від цього. Якщо трикутники відправляються в кластерах, так що кожен трикутник близький до раніше наданих трикутників, то більш ймовірно, що більше його вершин вже знаходяться в кеші.

Таким чином, пошук правильного порядку виведення трикутників є важливим для зменшення роботи конвеєра обробки вершин. Фундаментальна проблема полягає в тому, що сітки зазвичай є двовимірними поверхнями, а кеші є найбільш ефективними при передачі одновимірних даних. Таким чином, ідеальне впорядкування далеко не очевидно.

Звичайний спосіб посилатися на ефективність кешу вершин - поділити кількість пропусків вершини кешу на кількість трикутників у сітці. Це дає середнє значення пропуску кешу (СЗПК) - середню кількість вершин, які потрібно обробити на трикутник.

4.5.2 Структура кешування

Кожній вершині присвоюються різні "бали" залежно від її положення в кеш-пам'яті (тобто того, як це нещодавно використовується). Трикутникам також присвоюється оцінка, яка є сумою балів вершин, які вона використовує. При розгляді того, який трикутник у сітці слід додати до наступної послідовності, додається трикутник з найбільшою кількістю балів. Три вершини для трикутника потім переміщуються або додаються до верхньої частини кешу, а інші вершини переміщуються вниз. Алгоритм жадібний і ніколи не відступає і не дивиться вперед на свій вибір трикутника.

Як описано, алгоритм - це жадібний алгоритм, який не вимагає пошуку або зворотного відстеження. Після того, як він вирішив додати трикутник до послідовності візуалізації, він зберігає це рішення. Хоча вибір трикутника впливає на майбутній вибір через вершини та бали трикутника, поточний трикутник не вибирається для того, щоб впливати на ці майбутні варіанти. Таким чином алгоритм залишається дуже швидким і працює не тільки в лінійний час, пропорційний кількості трикутників в сітці, але і лінійний час дуже швидкий. Ця швидкість є надзвичайно корисною для створення авторських конвєєрів, де користувачам потрібно швидко та точно проглядати свої моделі.

4.5.3 Оцінка вершин сітки

Оцінка даної вершини вказує, наскільки ймовірно, що трикутник, який використовує її, буде доданий до списку наданих нижче. Високі бали роблять це більш імовірним, низькі - менше. Оцінка залежить від ряду речей.

Перший полягає в тому, що чим недавно використовувалася вершина, тим вище її оцінка. Оцінка обчислюється, приймаючи позицію вершини в кеш-пам'яті як дробове число між нулем і одиницею, з нулем в останній позиції і однією в першій позиції. Потім це масштабоване положення піднімається до потужності, більшої від одиниці. Значення потужності було вибрано багаторазовим моделюванням, а значення 1,5, здається, дає хороші результати. Використання потужності, а не простої лінійної оцінки, схоже, дає поведінку,

що не залежить від масштабів - бажану ознаку в алгоритмі, що намагається оптимізувати для невідомого розміру апаратного кешу.

Доданий до цього показника вершин є стимулом до вершини, якщо його залишкова валентність низька. Залишилася валентність вершини - це кількість трикутників, які її використовують, ще не намальовані. Вершина з лише одним трикутником, який потрібно намалювати, отримує найвищий приріст, а вершина з великою кількістю трикутників, які ще потрібно намалювати, отримує набагато менший приріст. Збільшення балів пропорційне кількості трикутників, які ще залишаються в ньому, піднімається до негативної дробової потужності (тестування показало, що -0,5 є найбільш ефективним), а потім масштабується відносно балу FIFO. Знову ж таки, тестування показало, що шкала 2,0 була найбільш ефективною. Ось таблиця вершин з розумними валентностями та їх оцінкою:

Valence	Score	Valence	Score
1	2.00	6	0.82
2	1.41	7	0.76
3	1.15	8	0.71
4	1.00	9	0.67
5	0.89	10	0.63

Таблиця 4.1 Таблиця вершин з розумними валентностями та їх оцінкою

4.5.4 Опис алгоритму та ефективність виконання

На відміну від багатьох традиційних алгоритмів топології сітки, реберні дані не зберігаються і не потрібні. Це дозволяє уникнути витрат на пошук крайових даних (як правило, дещо трудомістких), а також проблем, притаманних патологічним сіткам, де один край поділяється багатьма трикутниками.

Натомість кожна вершина містить такі дані:

- Його положення в модельованому кеші (-1, якщо його немає в кеші)
- Його поточний бал
- Загальна кількість трикутників, які його використовують
- Кількість ще не доданих трикутників, які використовують його
- Список індексів трикутників, які його використовують

Кожен трикутник у сітці також зберігає такі дані:

- Незалежно від того, додано він до списку розіграшів чи ні
- Оцінка трикутника (сума балів його вершин)
- Індекси до трьох вершин

Ініціалізація даних досить проста, з двома проходами над даними трикутника. Перший просто збільшує лічильник кількості трикутників, що використовують кожну вершину, другий виділяє списки трикутників для кожної вершини і заповнює їх. Після цього оцінка кожної вершини знаходить за допомогою наведеного вище коду, а потім оцінка кожного трикутника знаходимо шляхом підсумовування балів кожної вершини, яку використовує трикутник. Ці показники є просто кешованими значеннями обчислених балів і оновлюються при необхідності в міру виконання алгоритму.

Потім йде основний фрагмент алгоритму, який вибирає по одному трикутнику одночасно, щоб додати до списку намальованих трикутників, поки не залишиться більше трикутників для малювання. Зазвичай алгоритм знає з попередньої ітерації, який трикутник має найвищий бал, і просто вибирає його. У деяких випадках, наприклад, коли перший алгоритм працює, він вже не знає найкращого трикутника, або передбачуваний найкращий трикутник має незвично низький бал (мається на увазі, що в сітці можуть бути інші, які є кращими). У тих рідкісних обставинах він пробігає всі решти трикутників у сітці, шукаючи найкращий результат.

Потім найкращий трикутник додається до списку на отрисовку. Для кожної вершини, що використовується трикутником, валентність цієї вершини (кількість ще не намальованих трикутників, які використовують її)

зменшується на одиницю, а список індексів трикутників у вершині оновлюється відповідним чином.

Три вершини, використовувані трикутником, або переміщуються до голови модельованого кешу, або додаються до голови, якщо вони ще не були в ній. Кеш тимчасово збільшується в розмірі на три вершини, щоб включати всі вершини, які раніше були в кеші, і до трьох нових з цього трикутника. Потім нові позиції вершин у кеші оновлюються, їх відповідні оцінки знаходять за допомогою наведеного вище коду, а також оновлюються оцінки всіх їх ще доданих трикутників.

Нарешті, кеш скорочується до нормального розміру, з нього випадає до трьох вершин. Їх положення кешу вже оновлено відповідним чином. Алгоритм повторюється, поки не залишиться трикутників для додавання.

4.6 Висновки

Зменшення кількості шейдерного виклику під час рендеринга має важливе значення для забезпечення високої продуктивності. Традиційно надмірний виклик шейдерів для вершин можна обійти за допомогою кешу після перетворення, але його погана масштабованість робить кеш вершин поганим вибором у масово паралельних середовищах. Тому, перевага цього методу полягає в тому, що він є загальноприйнятим для широкого діапазону розмірів кешу та політики заміни, роботи в лінійному (і швидкому) часі та просторі, пропорційному кількості трикутників у сітці.

ВИСНОВКИ

Це дослідження мало на меті вивчити можливість процедурного генерування цікавих місцевостей та оптимізацію візуалізації у воксельних іграх. У розділі дизайну проекту було визначено три основні компоненти проекту: воксельний двигун, генерація місцевості та оптимізація рендерингу. Як створені ці компоненти, було роз'яснено у розділах реалізації.

Генерація місцевості була реалізована за допомогою шумових функцій, пов'язаних між собою для створення різноманітних місцевостей. Спочатку створюється карта висоти, яка використовується разом з додатковими шумовими функціями для обчислення блоків місцевості, розміщених у світі. Додаткові кроки в процесі генерації місцевості додають дерева та воду. Для ефективного розміщення та руйнування блоків у світі була використана техніка випромінювання променів.

Також було отримано новий алгоритм оптимізації, який був побудований на алгоритмах occlusion culling та frustum culling. Основним нововведенням було використання цих алгоритмів у програмному продукті для отримання більш швидкої генерації воксельних світів.

Практичне значення одержаних результатів полягає в тому, що програмний продукт може бути використаний у сучасних проектах комп'ютерної графіки або у сучасних іграх, які використовують технологію воксельного рендерингу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Occlusion Culling for Real-Time Rendering of Urban Environments, Peter Wonka
2. Dynamic Scene Visibility Culling using a Regular Grid, Shin-Ting
3. Dynamic Occlusion Culling, A. Julian Mayer
4. Occlusion Culling Algorithms: A Comprehensive Survey – [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/220061856_Occlusion_Culling_Algorithms_A_Comprehensive_Survey
5. The Advanced Cave Culling Algorithm™, or, making Minecraft faster , Tommos Blog – [Електронний ресурс]. Режим доступу: https://tomcc.github.io/2014/08/31/visibility-1.html
6. On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing
7. Linear-Speed Vertex Cache Optimisation – [Електронний ресурс]. Режим доступу: http://eelpi.gotdns.org/papers/fast_vert_cache_opt.html
8. Cepero, M. (2013, August 3). Procedural World: EverQuest Next – [Електронний ресурс]. Режим доступу: http://procworld.blogspot.co.nz/2013/08/everquest-next.html
9. Wikipedia, Thread pool – [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Thread_pool
10. Wikipedia, Polygon – [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Polygon_(computer_graphics)
11. Minecraft Wiki, Chunk – [Електронний ресурс]. Режим доступу: https://minecraft.gamepedia.com/Chunk
12. Minecraft Wiki, Light – [Електронний ресурс]. Режим доступу: https://minecraft.gamepedia.com/Light
13. Wikipedia, Ambient Occlusion – [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Ambient_occlusion
14. Ambient Occlusion for Minecraft-like worlds – [Електронний ресурс]. Режим доступу: https://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/
15. Perlin Noise File Reference – [Електронний ресурс]. Режим доступу: https://glm.g-truc.net/0.9.4/api/a00063.html
16. Libnoise – [Електронний ресурс]. Режим доступу: http://libnoise.sourceforge.net
17. Real-time voxel rendering algorithm based on Screen Space Billboard Voxel Buffer with Sparse Lookup Textures – [Електронний ресурс]. Режим доступу: https://dspace5.zcu.cz/bitstream/11025/29528/1/Jablonsky.pdf
18. The Advanced Cave Culling Algorithm, or making Minecraft faster – [Електронний ресурс]. Режим доступу: https://tomcc.github.io/2014/08/31/visibility-1.html

ДОДАТКИ

Додаток А – вихідний код класу Engine

```
#include "engine.h"
#include "camera.h"

#include "world/controller.h"
#include "world/util.h"

#include "base/logger.h"
#include "base/glDebug.h"
#include "base/threadPool.h"

#include <GL/glew.h>
#include <lodepng.h>
#include <glm/ext/matrix_transform.hpp>

#include <array>
#include <bitset>

namespace vox {

namespace {

const int kNumEdgeChunks = 1;
const int kNumNeighborsPerAxis = 2 * kNumEdgeChunks + 1;
const int kNumNeighbors = kNumNeighborsPerAxis * kNumNeighborsPerAxis;

int GetNeighborChunkIndex(int offsetX, int offsetZ) {
    return (offsetX + kNumEdgeChunks) + (offsetZ + kNumEdgeChunks) * kNumNeighborsPerAxis;
}

glm::vec4 NormalizePlane(glm::vec4 plane) {
    float len = std::sqrt(plane.x * plane.x + plane.y * plane.y + plane.z * plane.z);
    plane /= len;
    return plane;
}

} // namespace anonymous

Engine::Engine(WorldController* worldController, Camera* camera, std::string const& resourceDir)
    : m_worldController(worldController)
    , m_camera(camera)
    , m_maxAnisotropicLevel([]() {
          float maxAnisotropy = 0.0f;
          if (GLEW_EXT_texture_filter_anisotropic) {
              glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &maxAnisotropy);
          }

          int anisotropicLevel = 0;
          if (maxAnisotropy != 0.0f) {
              anisotropicLevel = std::log2(static_cast<int>(maxAnisotropy));
              if ((1 << anisotropicLevel) != int(maxAnisotropy)) {
                  LOG_INFO("Unusual max anisotropy: {}. Expected to be power of two.",
maxAnisotropy);
              }
          }

          return anisotropicLevel;
      }())
    , m_anisotropicLevel(m_maxAnisotropicLevel)
    , m_mainShader(resourceDir + "/shaders/main")
    , m_outlineShader(resourceDir + "/shaders/outline") {

    auto& texturePaths = m_worldController->GetBlockRegistry()->texturePaths;

    const int kImageDim = 16;
    const int kNumImagePixels = kImageDim * kImageDim;
    const int kImageSize = kNumImagePixels * 3;
    size_t dataSize = kImageSize * texturePaths.size();
    auto data = std::make_unique<uint8_t[]>(dataSize);

    size_t textureOffset = 0;
    for (auto& texturePath : texturePaths) {
        auto filename = resourceDir + texturePath;
        unsigned char* out;
        unsigned int w;
        unsigned int h;
```

```
        auto status = lodepng_decode24_file(&out, &w, &h, filename.c_str());
        if (status != 0 || w != kImageDim || h != kImageDim) {
            if (status == 0) { free(out); }

            // Set to pink
            for (int i = 0; i < kNumImagePixels; ++i) {
                size_t pixelOffset = i * 3 + textureOffset;
                data[pixelOffset + 0] = 255;
                data[pixelOffset + 1] = 0;
                data[pixelOffset + 2] = 255;
            }
        } else {
            std::memcpy(data.get() + textureOffset, out, kImageSize);
            free(out);
        }

        textureOffset += kImageSize;
    }

    GL_CALL(glGenTextures(1, &m_textureArray));
    GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, m_textureArray));
    GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
    GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
    GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S, GL_REPEAT));
    GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T, GL_REPEAT));
    GL_CALL(glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGB8, kImageDim, kImageDim, texturePaths.size(),
0, GL_RGB, GL_UNSIGNED_BYTE, data.get()));
    GL_CALL(glGenerateMipmap(GL_TEXTURE_2D_ARRAY));
    GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, 0));

    GL_CALL(glGenVertexArrays(1, &m_outlineVao));
    GL_CALL(glBindVertexArray(m_outlineVao));

    std::vector<glm::vec3> cubeWireframe;
    cubeWireframe.reserve(24);
    auto fillXZ = [&cubeWireframe](float y) {
        cubeWireframe.emplace_back(0.0f, y, 1.0f);
        cubeWireframe.emplace_back(1.0f, y, 1.0f);

        cubeWireframe.emplace_back(1.0f, y, 1.0f);
        cubeWireframe.emplace_back(1.0f, y, 0.0f);

        cubeWireframe.emplace_back(1.0f, y, 0.0f);
        cubeWireframe.emplace_back(0.0f, y, 0.0f);

        cubeWireframe.emplace_back(0.0f, y, 0.0f);
        cubeWireframe.emplace_back(0.0f, y, 1.0f);
    };
    fillXZ(0.0f);
    fillXZ(1.0f);

    cubeWireframe.emplace_back(0.0f, 0.0f, 0.0f);
    cubeWireframe.emplace_back(0.0f, 1.0f, 0.0f);

    cubeWireframe.emplace_back(1.0f, 0.0f, 0.0f);
    cubeWireframe.emplace_back(1.0f, 1.0f, 0.0f);

    cubeWireframe.emplace_back(0.0f, 0.0f, 1.0f);
    cubeWireframe.emplace_back(0.0f, 1.0f, 1.0f);

    cubeWireframe.emplace_back(1.0f, 0.0f, 1.0f);
    cubeWireframe.emplace_back(1.0f, 1.0f, 1.0f);

    GL_CALL(glGenBuffers(1, &m_outlineVbo));
    GL_CALL(glBindBuffer(GL_ARRAY_BUFFER, m_outlineVbo));
    GL_CALL(glBufferData(GL_ARRAY_BUFFER, sizeof(cubeWireframe[0]) * cubeWireframe.size(),
cubeWireframe.data(), GL_STATIC_DRAW));

    GL_CALL(glEnableVertexAttribArray(0));
    GL_CALL(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0));

    GL_CALL(glBindVertexArray(0));
    GL_CALL(glBindBuffer(GL_ARRAY_BUFFER, 0));
}

Engine::~Engine() {
    if (m_textureArray) {
        glDeleteTextures(1, &m_textureArray);
    }
```

```
    if (m_outlineVao) {
        glDeleteVertexArrays(1, &m_outlineVao);
    }

    if (m_outlineVbo) {
        glDeleteBuffers(1, &m_outlineVbo);
    }
}

Engine::SectionMesh::~SectionMesh() {
    Release();
}

void Engine::SectionMesh::Release() {
    if (vbo) {
        GL_CALL(glDeleteBuffers(1, &vbo));
        vbo = 0;
    }
    if (ebo) {
        GL_CALL(glDeleteBuffers(1, &ebo));
        ebo = 0;
    }
    if (vao) {
        GL_CALL(glDeleteVertexArrays(1, &vao));
        vao = 0;
    }
}

void Engine::SetRenderDistance(uint8_t renderDistance) {
    if (m_renderDistance != renderDistance) {
        m_renderDistance = renderDistance;
        m_dirtyBits |= kDirtyRenderDistance;
    }
}

void Engine::SetInteractionDistance(float interactionDistance) {
    if (interactionDistance > 0.0f) {
        m_interactionDistance = interactionDistance;
    }
}

void Engine::SetActiveBlockOutlineColor(glm::vec3 const& color) {
    m_outlineShader.SetUniform("outlineColor", color.x, color.y, color.z, 1.0f);
}

void Engine::SetFog(bool isFogEnabled) {
    m_mainShader.SetUniform("isFogEnabled", isFogEnabled);
}

void Engine::SetAmbientOcclusion(float level) {
    m_mainShader.SetUniform("AOLevel", level);
}

void Engine::SetFilter(Filter newFilter) {
    if (m_filter == newFilter) {
        return;
    }

    if (m_filter == Filter::Anisotropic && m_maxAnisotropicLevel == 0) {
        LOG_ERROR("Anisotropic filter is unavailable");
        return;
    }

    GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, m_textureArray));

    if (m_filter == Filter::Anisotropic) {
        GL_CALL(glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAX_ANISOTROPY_EXT, 1.0f));
    } else if (newFilter == Filter::Anisotropic) {
        GL_CALL(glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAX_ANISOTROPY_EXT, pow(2.0,
m_anisotropicLevel)));
    }

    bool mipMapEnabled = false;
    if (m_filter == Filter::Anisotropic || m_filter == Filter::Mipmap) {
        mipMapEnabled = true;
    }

    if ((newFilter == Filter::Mipmap || newFilter == Filter::Anisotropic) && !mipMapEnabled) {
```

```
        GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER,
GL_NEAREST_MIPMAP_NEAREST));
    } else if (newFilter == Filter::None && mipMapEnabled) {
        GL_CALL(glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
    }

    m_filter = newFilter;

    GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, 0));
}

void Engine::SetAnisotropyLevel(int level) {
    if (level < 0 && level > m_maxAnisotropicLevel) {
        return;
    }

    if (m_anisotropicLevel != level) {
        m_anisotropicLevel = level;

        if (m_filter == Filter::Anisotropic) {
            GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, m_textureArray));
            GL_CALL(glTexParameterf(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAX_ANISOTROPY_EXT, pow(2.0,
m_anisotropicLevel)));
            GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, 0));
        }
    }
}

void Engine::Render(WorldController::ChangesSummary const& changes) {
    Update(changes);

    if (m_activeBlockInfo.exists) {
        auto activeBlockGlobalCoord = GetBlockGlobalCoord(m_activeBlockInfo.coord);
        glm::mat4 outlineTransform = glm::translate(glm::mat4(1.0f),
glm::vec3(activeBlockGlobalCoord));
        glm::mat4 viewProjection = m_camera->GetProjectionMatrix() * m_camera->GetViewMatrix() *
outlineTransform;

        glDisable(GL_POLYGON_OFFSET_FILL);

        m_outlineShader.Bind();
        GL_CALL(glBindVertexArray(m_outlineVao));
        m_outlineShader.SetUniform("viewProjection", viewProjection);
        GL_CALL(glDrawArrays(GL_LINES, 0, 24));
        GL_CALL(glBindVertexArray(0));
        m_outlineShader.Unbind();

        glEnable(GL_POLYGON_OFFSET_FILL);
        glPolygonOffset(1.0, 1.0);
    }

    GL_CALL(glBindTexture(GL_TEXTURE_2D_ARRAY, m_textureArray));

    m_mainShader.Bind();
    for (auto& chunkEntry : m_syncedChunks) {
        if (IsChunkInFrustum(chunkEntry.first)) {
            for (auto& mesh : chunkEntry.second.meshes) {
                if (mesh.vao) {
                    GL_CALL(glBindVertexArray(mesh.vao));
                    GL_CALL(glDrawElements(GL_TRIANGLES, mesh.numIndices, GL_UNSIGNED_INT, 0));
                }
            }
        }
    }
    GL_CALL(glBindVertexArray(0));
    m_mainShader.Unbind();

    if (m_activeBlockInfo.exists) {
        glPolygonOffset(0.0, 0.0);
    }
}

void Engine::Update(WorldController::ChangesSummary const& changes) {
    UpdateCamera();
    UpdateChunks(changes);
    UpdateActiveBlock();

    m_camera->m_engineDirtyFlags = Camera::kAllClean;
    m_dirtyBits = ChangeTracker::kAllClean;
```

```
}

void Engine::UpdateCamera() {
    if (m_camera->m_engineDirtyFlags & Camera::kDirtyPosition) {
        auto& cameraPosition = m_camera->GetPosition();
        auto newEyeChunkCoord = WorldChunkCoord(
            static_cast<int>(std::floor(cameraPosition[0] / WorldSection::kWidth)),
            static_cast<int>(std::floor(cameraPosition[2] / WorldSection::kDepth)));
        if (m_eyeChunkCoord != newEyeChunkCoord) {
            printf("{%.3f %.3f} -> {%d %d}\n",
                cameraPosition[0], cameraPosition[2],
                newEyeChunkCoord[0], newEyeChunkCoord[1]);
            m_eyeChunkCoord = newEyeChunkCoord;
            m_dirtyBits |= kDirtyEyeChunkCoord;
        }
    }

    if (m_dirtyBits & kDirtyRenderDistance) {
        // float farDistance = std::max(int(m_renderDistance), 1) * float(sqrt(WorldSection::kWidth
* WorldSection::kWidth + WorldSection::kDepth * WorldSection::kDepth));
        float farDistance = 1000.0f;
        m_camera->SetClipPlanes(0.1f, farDistance);
    }

    if (m_camera->m_engineDirtyFlags & (Camera::kDirtyProjection | Camera::kDirtyViewMatrix)) {
        auto viewProjection = m_camera->GetProjectionMatrix() * m_camera->GetViewMatrix();
        m_mainShader.SetUniform("viewProjection", viewProjection);

        if (updateFrustum) {
            auto m = glm::transpose(viewProjection);
            m_frustumPlanes[0] = NormalizePlane(m[3] + m[0]);
            m_frustumPlanes[1] = NormalizePlane(m[3] - m[0]);
            m_frustumPlanes[2] = NormalizePlane(m[3] - m[1]);
            m_frustumPlanes[3] = NormalizePlane(m[3] + m[1]);
            m_frustumPlanes[4] = NormalizePlane(m[3] + m[2]);
            m_frustumPlanes[5] = NormalizePlane(m[3] - m[2]);
        }
    }
}

void Engine::UpdateChunks(WorldController::ChangesSummary const& changes) {
    auto oldCachedChunksRange = m_cachedChunksRange;

    if (((m_dirtyBits & kDirtyEyeChunkCoord) ||
        (m_dirtyBits & kDirtyRenderDistance)) && updateChunkCache) {
        // Remove all cached chunks that are out of scope now
        m_cachedChunksRange.min = m_eyeChunkCoord - glm::ivec2(m_renderDistance + kNumEdgeChunks);
        m_cachedChunksRange.max = m_eyeChunkCoord + glm::ivec2(m_renderDistance + kNumEdgeChunks);

        if (!oldCachedChunksRange.InsideOf(m_cachedChunksRange)) {
            // Remove all no more needed chunks
            for (auto it = m_chunkCache.begin(); it != m_chunkCache.end();) {
                if (!m_cachedChunksRange.Contains(it->first)) {
                    it = m_chunkCache.erase(it);
                } else {
                    ++it;
                }
            }

            // Remove all synced chunks that are not visible anymore
            for (auto it = m_syncedChunks.begin(); it != m_syncedChunks.end();) {
                if (!IsChunkVisible(it->first)) {
                    it = m_syncedChunks.erase(it);
                } else {
                    ++it;
                }
            }
        }
    }

    for (auto& entry : changes.changedChunks) {
        // Sync only visible chunks
        if (IsChunkVisible(entry.first)) {
            bool chunkIsCached = m_chunkCache.count(entry.first) != 0;
            assert(chunkIsCached);
            if (chunkIsCached) {
                m_chunksToSync[entry.first] |= entry.second;
            }
        }
```

```
    }

    for (auto& chunk : changes.newChunks) {
        if (m_cachedChunksRange.Contains(chunk->coord)) {
            CacheChunk(chunk);

            // Sync only visible chunks
            if (IsChunkVisible(chunk->coord)) {
                m_chunksToSync[chunk->coord] |= WorldChunk::kSectionsAll;
            }
        }
    }

    if (((m_dirtyBits & kDirtyEyeChunkCoord) ||
        (m_dirtyBits & kDirtyRenderDistance)) && updateChunkCache) {
        // Add all left chunks that were not cached in the previous steps

        for (auto x = m_cachedChunksRange.min.x; x <= m_cachedChunksRange.max.x; ++x) {
            for (auto z = m_cachedChunksRange.min.y; z <= m_cachedChunksRange.max.y; ++z) {
                WorldChunkCoord coord{x, z};

                // If chunk was not cached previously
                if (!oldCachedChunksRange.Contains(coord)) {
                    // And if it was not cached in the previous steps, request it from world
controller
                    if (m_chunkCache.count(coord) == 0) {
                        if (auto chunk = m_worldController->GetChunk(coord)) {
                            CacheChunk(std::move(chunk));

                            // Sync only visible chunks
                            if (IsChunkVisible(coord)) {
                                m_chunksToSync[coord] |= WorldChunk::kSectionsAll;
                            }
                        }
                    }
                } else {
                    // Sync only visible chunks that was not synced before
                    if (IsChunkVisible(coord) &&
                        m_chunkCache.count(coord) &&
                        m_syncedChunks.count(coord) == 0 &&
                        m_syncingChunks.count(coord) == 0) {
                        m_chunksToSync[coord] |= WorldChunk::kSectionsAll;
                    }
                }
            }
        }
    }

    for (auto it = m_chunksToSync.begin(); it != m_chunksToSync.end();) {
        if (IsChunkVisible(it->first)) {
            if (EnqueueChunkSync(it->second, m_chunkCache.at(it->first))) {
                it = m_chunksToSync.erase(it);
            } else {
                ++it;
            }
        } else {
            it = m_chunksToSync.erase(it);
        }
    }

    if (!m_syncCompletions.empty()) {
        std::lock_guard<std::mutex> lock(m_syncMutex);
        for (auto& completion : m_syncCompletions) {
            completion();
        }
        m_syncCompletions.clear();
    }
}

void Engine::UpdateActiveBlock() {
    m_activeBlockInfo.exists = false;

    auto& origin = m_camera->GetPosition();
    auto& direction = m_camera->GetDirection();

    auto endPoint = origin + direction * m_interactionDistance;

    glm::ivec3 currentVoxel(
        static_cast<int>(std::floor(origin.x)),
```

```
    static_cast<int>(std::floor(origin.y)),
    static_cast<int>(std::floor(origin.z)));
glm::ivec3 endVoxel(
    static_cast<int>(std::floor(endPoint.x)),
    static_cast<int>(std::floor(endPoint.y)),
    static_cast<int>(std::floor(endPoint.z)));

glm::ivec3 step(
    (direction.x >= 0.0f) ? 1 : -1,
    (direction.y >= 0.0f) ? 1 : -1,
    (direction.z >= 0.0f) ? 1 : -1);

glm::vec3 tMax;
glm::vec3 tDelta;
for (int i = 0; i < 3; ++i) {
    if (direction[i] == 0.0f) {
        tMax[i] = std::numeric_limits<float>::max();
        tDelta[i] = std::numeric_limits<float>::max();
    } else {
        int nextVoxelBoundary = currentVoxel[i] + (step[i] + 1) / 2;
        tMax[i] = (nextVoxelBoundary - origin[i]) / direction[i];
        tDelta[i] = step[i] / direction[i];
    }
}

// Cache chunk
auto chunkCoord = GetChunkWorldCoord(currentVoxel.x, currentVoxel.z);
auto chunk = GetCachedChunk(chunkCoord);

if (!chunk) {
    return;
}

enum {
    kAxisX,
    kAxisY,
    kAxisZ,
} stepAxis;

while (endVoxel != currentVoxel) {
    if (tMax.x < tMax.y) {
        if (tMax.x < tMax.z) {
            currentVoxel.x += step.x;
            tMax.x += tDelta.x;
            stepAxis = kAxisX;
        } else {
            currentVoxel.z += step.z;
            tMax.z += tDelta.z;
            stepAxis = kAxisZ;
        }
    } else {
        if (tMax.y < tMax.z) {
            currentVoxel.y += step.y;
            tMax.y += tDelta.y;
            stepAxis = kAxisY;
        } else {
            currentVoxel.z += step.z;
            tMax.z += tDelta.z;
            stepAxis = kAxisZ;
        }
    }

    if (currentVoxel.y < 0 || currentVoxel.y >= WorldChunk::kHeight * WorldSection::kHeight) {
        break;
    }

    auto currentChunkCoord = GetChunkWorldCoord(currentVoxel.x, currentVoxel.z);
    if (currentChunkCoord != chunkCoord) {
        chunkCoord = currentChunkCoord;
        chunk = GetCachedChunk(chunkCoord);
        if (!chunk) {
            break;
        }
    }

    int sectionIndex = currentVoxel.y / WorldSection::kHeight;
    if (chunk->sections[sectionIndex].blocks) {
        auto blockCoord = GetBlockLocalCoord(currentVoxel);
        if (chunk->sections[sectionIndex].block(blockCoord)) {
```

```
                m_activeBlockInfo.exists = true;

                m_activeBlockInfo.coord.chunk = chunkCoord;
                m_activeBlockInfo.coord.section = sectionIndex;
                m_activeBlockInfo.coord.block = blockCoord;

                m_activeBlockInfo.neighbourOffset = glm::ivec3(0);
                if (stepAxis == kAxisX) {
                    m_activeBlockInfo.neighbourOffset.x = -step.x;
                } else if (stepAxis == kAxisY) {
                    m_activeBlockInfo.neighbourOffset.y = -step.y;
                } else {
                    m_activeBlockInfo.neighbourOffset.z = -step.z;
                }
                break;
            }
        }
    }
}

bool Engine::IsChunkVisible(WorldChunkCoord const& coord) const {
    return coord.x >= (m_cachedChunksRange.min.x + kNumEdgeChunks) && coord.x <=
(m_cachedChunksRange.max.x - kNumEdgeChunks) &&
           coord.y >= (m_cachedChunksRange.min.y + kNumEdgeChunks) && coord.y <=
(m_cachedChunksRange.max.y - kNumEdgeChunks);
}

bool Engine::IsChunkInFrustum(WorldChunkCoord const& coord) const {
    auto min = glm::vec3(coord.x * WorldSection::kWidth, 0.0f, coord.y * WorldSection::kDepth);
    auto max = min + glm::vec3(WorldSection::kWidth, WorldSection::kHeight * WorldChunk::kHeight,
WorldSection::kDepth);

    for (auto& plane : m_frustumPlanes) {
        // pick closest point to plane and check if it behind the plane
        // if yes - object outside frustum
        float d =
            std::max(min.x * plane.x, max.x * plane.x) +
            std::max(min.y * plane.y, max.y * plane.y) +
            std::max(min.z * plane.z, max.z * plane.z) + plane.w;
        if (d <= 0) {
            return false;
        }
    }
    return true;
}

void Engine::CacheChunk(std::shared_ptr<WorldChunk> chunk) {
    auto st = m_chunkCache.emplace(
        std::piecewise_construct,
        std::forward_as_tuple(chunk->coord),
        std::forward_as_tuple(std::move(chunk)));
    assert(st.second);
}

WorldChunk* Engine::GetCachedChunk(WorldChunkCoord const& coord) {
    auto it = m_chunkCache.find(coord);
    if (it == m_chunkCache.end()) {
        return nullptr;
    }
    return it->second.get();
}

bool Engine::EnqueueChunkSync(
    WorldChunk::SectionsMask changedSections,
    std::shared_ptr<WorldChunk> chunk) {
    if (m_syncingChunks.count(chunk->coord)) {
        return false;
    }

    // Retain chunks to prevent its deallocation while syncing
    std::array<std::shared_ptr<WorldChunk>, kNumNeighbors> retainedNeighbors;

    for (int dx = -1; dx <= 1; ++dx) {
        for (int dy = -1; dy <= 1; ++dy) {
            if (dx == 0 && dy == 0) {
                continue;
            }
            auto chunkIt = m_chunkCache.find(chunk->coord + glm::ivec2(dx, dy));
            if (chunkIt != m_chunkCache.end()) {
```

```
                retainedNeighbors[GetNeighborChunkIndex(dx, dy)] = chunkIt->second;
            } else {
                // We can't sync chunk if not all neighbors are present yet
                return false;
            }
        }
    }

    // If only one section is changed, resync it on the main thread for better responsiveness
    if (std::bitset<WorldChunk::kHeight>(changedSections).count() == 1) {
        SyncChunk(changedSections, chunk.get(), retainedNeighbors.data());
    } else {
        GetThreadPool().enqueue(
            [this](WorldChunk::SectionsMask changedSections,
               std::shared_ptr<WorldChunk> chunk,
               std::array<std::shared_ptr<WorldChunk>, kNumNeighbors> retainedNeighbors) {
            SyncChunk(changedSections, chunk.get(), retainedNeighbors.data());
        }, changedSections, chunk, std::move(retainedNeighbors));
        m_syncingChunks.insert(chunk->coord);
    }
    return true;
}

void Engine::SyncChunk(WorldChunk::SectionsMask changedSections,
    WorldChunk const* chunk,
    std::shared_ptr<WorldChunk> const* neighborChunks) {
    // Generate chunk mesh

    auto& blockDescs = m_worldController->GetBlockRegistry()->blockDescs;

    static const std::vector<glm::vec3> kCubePoints = {
        {1, 1, 1}, {1, 1, 0}, {1, 0, 0}, {1, 0, 1},
        {0, 1, 0}, {0, 1, 1}, {0, 0, 1}, {0, 0, 0},
        {0, 1, 1}, {0, 1, 0}, {1, 1, 0}, {1, 1, 1},
        {0, 0, 1}, {1, 0, 1}, {1, 0, 0}, {0, 0, 0},
        {0, 1, 1}, {1, 1, 1}, {1, 0, 1}, {0, 0, 1},
        {1, 1, 0}, {0, 1, 0}, {0, 0, 0}, {1, 0, 0},
    };
    static const std::vector<glm::vec3> kCubeNormals = {
        {1, 0, 0}, {1, 0, 0}, {1, 0, 0}, {1, 0, 0},
        {-1, 0, 0}, {-1, 0, 0}, {-1, 0, 0}, {-1, 0, 0},
        {0, 1, 0}, {0, 1, 0}, {0, 1, 0}, {0, 1, 0},
        {0, -1, 0}, {0, -1, 0}, {0, -1, 0}, {0, -1, 0},
        {0, 0, 1}, {0, 0, 1}, {0, 0, 1}, {0, 0, 1},
        {0, 0, -1}, {0, 0, -1}, {0, 0, -1}, {0, 0, -1},
    };
    static const std::vector<int> kCubeIndices = {
        0, 1, 2, 2, 3, 0, // +X
        4, 5, 6, 6, 7, 4, // -X
        8, 9, 10, 10, 11, 8, // +Y
        12, 13, 14, 14, 15, 12, // -Y
        16, 17, 18, 18, 19, 16, // +Z
        20, 21, 22, 22, 23, 20 // -Z
    };
    static const std::vector<glm::vec2> kCubeUvs = {
        {0, 0}, {1, 0}, {1, 1}, {0, 1},
        {0, 0}, {1, 0}, {1, 1}, {0, 1},
        {0, 1}, {0, 0}, {1, 0}, {1, 1},
        {0, 0}, {1, 0}, {1, 1}, {0, 1},
        {0, 0}, {1, 0}, {1, 1}, {0, 1},
        {0, 0}, {1, 0}, {1, 1}, {0, 1},
    };

    struct InterleavedVertex {
        glm::vec3 point;
        glm::vec2 uv;
        uint32_t data;
    };

    struct Mesh {
        std::vector<InterleavedVertex> vertices;
        std::vector<int> indices;
        int iSection;

        Mesh(int iSection) : iSection(iSection) {}
    };

    struct CompletionData {
        std::vector<Mesh> meshes;
```

```
        WorldChunk::SectionsMask changedSections;
    };
    auto completionData = new CompletionData;
    completionData->changedSections = changedSections;

    // Culled: skip faces that are hidden between two opaque blocks
    for (int iSection = 0; iSection < WorldChunk::kHeight; ++iSection) {
        if ((changedSections & (1 << iSection)) == 0) {
            continue;
        }

        auto& section = chunk->sections[iSection];
        if (!section.blocks) {
            continue;
        }

        glm::ivec3 sectionWorldCoordinate(chunk->coord.x * WorldSection::kWidth, iSection *
WorldSection::kHeight, chunk->coord.y * WorldSection::kDepth);

        // Create new entry if there was no entries before or previous one is not empty
        if (completionData->meshes.empty() ||
            !completionData->meshes.back().vertices.empty()) {
            completionData->meshes.emplace_back(iSection);
        } else if (completionData->meshes.back().vertices.empty()) {
            // If previous entry is empty we can reuse it but we need to override its section index
            completionData->meshes.back().iSection = iSection;
        }
        auto& vertices = completionData->meshes.back().vertices;
        auto& indices = completionData->meshes.back().indices;

        const int kNumCubeVerticesPerSide = kCubePoints.size() / WorldBlockDesc::kNumSides;
        struct CubeSide {
            glm::ivec3 neighbourOffset;
        };
        std::array<CubeSide, WorldBlockDesc::kNumSides> cubeSides;
        for (int i = 0; i < WorldBlockDesc::kNumSides; ++i) {
            auto& normal = kCubeNormals[i * kNumCubeVerticesPerSide];
            cubeSides[i].neighbourOffset = glm::ivec3(int(normal.x), int(normal.y), int(normal.z));
        }

        auto getNeighbour = [iSection, chunk, neighborChunks](int x, int y, int z, glm::ivec3 const&
neighbourOffset) -> WorldBlockId {
            int yy = y + neighbourOffset.y;
            int neighborSectionIdx;
            BlockLocalCoord neighborBlockCoord;
            if (yy == -1) {
                neighborSectionIdx = iSection - 1;
                neighborBlockCoord.y = WorldSection::kHeight - 1;
            } else if (yy == WorldSection::kHeight) {
                neighborSectionIdx = iSection + 1;
                neighborBlockCoord.y = 0;
            } else {
                neighborSectionIdx = iSection;
                neighborBlockCoord.y = yy;
            }

            // Fast return if out of the world
            if (neighborSectionIdx == WorldChunk::kHeight ||
                neighborSectionIdx == -1) {
                return kAirId;
            }

            int xx = x + neighbourOffset.x;
            int zz = z + neighbourOffset.z;
            glm::ivec2 chunkOffset;
            if (xx == -1) {
                chunkOffset.x = -1;
                neighborBlockCoord.x = WorldSection::kWidth - 1;
            } else if (xx == WorldSection::kWidth) {
                chunkOffset.x = 1;
                neighborBlockCoord.x = 0;
            } else {
                chunkOffset.x = 0;
                neighborBlockCoord.x = xx;
            }

            if (zz == -1) {
                chunkOffset.y = -1;
                neighborBlockCoord.z = WorldSection::kDepth - 1;
```

```
            } else if (zz == WorldSection::kDepth) {
                chunkOffset.y = 1;
                neighborBlockCoord.z = 0;
            } else {
                chunkOffset.y = 0;
                neighborBlockCoord.z = zz;
            }

            WorldChunk const* neighborChunk;
            if (chunkOffset.x == 0 && chunkOffset.y == 0) {
                neighborChunk = chunk;
            } else {
                neighborChunk = neighborChunks[GetNeighborChunkIndex(chunkOffset.x,
chunkOffset.y)].get();
            }

            auto& neighborSection = neighborChunk->sections[neighborSectionIdx];
            if (!neighborSection.blocks) {
                return kAirId;
            }

            return neighborSection.block(neighborBlockCoord);
        };

        for (int y = 0; y < WorldSection::kHeight; ++y) {
            int yOffset = y * WorldSection::kSliceSize;
            for (int z = 0; z < WorldSection::kDepth; ++z) {
                int zOffset = z * WorldSection::kDepth + yOffset;
                for (int x = 0; x < WorldSection::kWidth; ++x) {
                    auto blockId = section.blocks[x + zOffset];
                    if (blockId == kAirId) {
                        continue;
                    }

                    auto positionOffset = glm::vec3(x, y, z) + sectionWorldCoordinate;

                    for (int iSide = 0; iSide < WorldBlockDesc::kNumSides; ++iSide) {
                        // Skip bedrock down side
                        if (iSection == 0 && y == 0 && iSide == WorldBlockDesc::kSideBottom) {
                            continue;
                        }

                        auto neighbour = getNeighbour(x, y, z, cubeSides[iSide].neighbourOffset);
                        if (neighbour == kAirId || blockDescs[neighbour].isTransparent()) {
                            auto textureIndex = blockDescs[blockId].sideTextures[iSide];
                            auto indicesOffset = int(vertices.size());
                            int ao[4];

                            auto verticesOffset = iSide * kNumCubeVerticesPerSide;
                            vertices.reserve(vertices.size() + kNumCubeVerticesPerSide);
                            for (int iVertex = 0; iVertex < kNumCubeVerticesPerSide; ++iVertex) {
                                const int idx = iVertex + verticesOffset;
                                auto vertexPosition = kCubePoints[idx];
                                auto vertexNormal = cubeSides[iSide].neighbourOffset;

                                // Ambient occlusion
                                // Every vertex has 4 neighbor blocks that can occlude it -
                                // one block above the current face and 3 blocks adjacent to the
former.
                                // Obviously, if the block above the current face would exist,
                                // we would not reach this code.
                                // So, effectively only 3 blocks can occlude the current vertex.
                                // Normal indicates the front axis, all other axes are side ones.
                                // To calculate ambient occlusion we are checking for 3 neighbor
                                // blocks that lies on side axes.
                                // For example, if normal is (0 1 0) then side axises is X and Z.
                                int sideAxisIndex0;
                                int sideAxisIndex1;
                                bool firstZero = true;
                                for (int iAxis = 0; iAxis < 3; ++iAxis) {
                                    if (vertexNormal[iAxis] == 0) {
                                        if (firstZero) {
                                            firstZero = false;
                                            sideAxisIndex0 = iAxis;
                                        } else {
                                            sideAxisIndex1 = iAxis;
                                        }
                                    }
                                }
```

```
                                // Also, we need to know on which side of these side axes lies our
vertex
                                auto neighborDir = vertexPosition - glm::vec3(0.5f);
                                // dir > 0 indicates that the neighbor lies on the positive side of
the side axis
                                // Continuing my example, if neighborDir.x > 0 and neighborDir.z < 0
then
                                // we gonna check neighbor blocks with such offsets:
                                // (normal + (1 0 0)), (normal + (0 0 -1)), (normal + (1 0 -1))

                                auto offset = vertexNormal;
                                offset[sideAxisIndex0] = neighborDir[sideAxisIndex0] > 0 ? 1 : -1;
                                int side1 = getNeighbour(x, y, z, offset) != kAirId;

                                offset = vertexNormal;
                                offset[sideAxisIndex1] = neighborDir[sideAxisIndex1] > 0 ? 1 : -1;
                                int side2 = getNeighbour(x, y, z, offset) != kAirId;

                                if (side1 && side2) {
                                    ao[iVertex] = 0;
                                } else {
                                    offset[sideAxisIndex0] = neighborDir[sideAxisIndex0] > 0 ? 1 : -
1;
                                    int corner = getNeighbour(x, y, z, offset) != kAirId;

                                    ao[iVertex] = 3 - (side1 + side2 + corner);
                                }

                                static constexpr uint32_t kNumTextureBits = 16;
                                static constexpr uint32_t kTextureBitsOffset = 0;
                                static constexpr uint32_t kNumAOBits = 2;
                                static constexpr uint32_t kAOBitsOffset = kNumTextureBits;

                                static constexpr uint32_t kTextureMask = (1 << kNumTextureBits) - 1;
                                static constexpr uint32_t kAOMask = (1 << kNumAOBits) - 1;

                                uint32_t vertexData = 0;
                                vertexData |= (textureIndex & kTextureMask) << kTextureBitsOffset;
                                vertexData |= (ao[iVertex] & kAOMask) << kAOBitsOffset;

                                vertices.push_back({
                                    vertexPosition + positionOffset,
                                    kCubeUvs[idx],
                                    vertexData
                                });
                            }

                            indices.reserve(indices.size() + 6);
                            bool flip = (ao[1] + ao[3]) > (ao[0] + ao[2]);
                            if (flip) {
                                indices.push_back(indicesOffset + 0);
                                indices.push_back(indicesOffset + 1);
                                indices.push_back(indicesOffset + 3);
                                indices.push_back(indicesOffset + 1);
                                indices.push_back(indicesOffset + 2);
                                indices.push_back(indicesOffset + 3);
                            } else {
                                indices.push_back(indicesOffset + 0);
                                indices.push_back(indicesOffset + 1);
                                indices.push_back(indicesOffset + 2);
                                indices.push_back(indicesOffset + 0);
                                indices.push_back(indicesOffset + 2);
                                indices.push_back(indicesOffset + 3);
                            }
                        }
                    }
                }
            }
        }

        if (vertices.empty()) {
            continue;
        }
    }

    std::lock_guard<std::mutex> lock(m_syncMutex);
    m_syncCompletions.push_back([this, completionData, coord = chunk->coord]() {
        m_syncingChunks.erase(coord);
```

```
        if (!IsChunkVisible(coord)) {
            // Outdated, discard
            delete completionData;
            return;
        }

        auto& syncedChunk = m_syncedChunks[coord];
        for (int i = 0; i < WorldSection::kHeight; ++i) {
            if (completionData->changedSections & (1 << i)) {
                syncedChunk.meshes[i].Release();
            }
        }

        for (auto& mesh : completionData->meshes) {
            if (mesh.vertices.empty()) {
                continue;
            }

            auto& vertices = mesh.vertices;
            auto& indices = mesh.indices;

            auto& glMesh = syncedChunk.meshes[mesh.iSection];

            GL_CALL(glGenVertexArrays(1, &glMesh.vao));
            GL_CALL(glBindVertexArray(glMesh.vao));

            GL_CALL(glGenBuffers(1, &glMesh.vbo));
            GL_CALL(glBindBuffer(GL_ARRAY_BUFFER, glMesh.vbo));
            GL_CALL(glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertices[0]),
vertices.data(), GL_STATIC_DRAW));

            GL_CALL(glEnableVertexAttribArray(0));
            GL_CALL(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertices[0]), 0));

            GL_CALL(glEnableVertexAttribArray(1));
            GL_CALL(glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(vertices[0]),
reinterpret_cast<void*>(sizeof(glm::vec3))));

            GL_CALL(glEnableVertexAttribArray(2));
            GL_CALL(glVertexAttribIPointer(2, 1, GL_UNSIGNED_INT, sizeof(vertices[0]),
reinterpret_cast<void*>(sizeof(glm::vec3) + sizeof(glm::vec2))));

            GL_CALL(glGenBuffers(1, &glMesh.ebo));
            GL_CALL(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, glMesh.ebo));
            GL_CALL(glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(indices[0]),
indices.data(), GL_STATIC_DRAW));

            glMesh.numIndices = static_cast<int>(indices.size());
        }

        delete completionData;
    });
}

} // namespace vox
```