Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Корнієнка Данила Андрійовича*
	(ПІБ)
академічної групи	*121м-22-2*
	(шифр)
спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«Комп'ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження та розробка високопродуктиного шейдеру для рендеру та анімації трави в ігрових застосунках*

___________________ *Корнієнко Д.А.*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		Рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	*Доц. Приходченко С.Д.*			
Рецензент				
Нормоконтролер	*Доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
Алексєєв М.О.
(підпис) (прізвище, ініціали)
« » 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студента *121м-22-2* *Корнієнка Данила Андрійовича*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження та розробка високопродуктивного шейдеру для рендеру та анімації трави в ігрових застосунках*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процеси обробки та відображення великої кількості об'єктів.

Предмет досліджень – методи оптимізації ренедеру об'єктів.

Мета роботи – розробити оптимізоване ришення для рендеру трави; з уникальним візуальним стилем.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна работи полягає у – створені шейдеру трави та послідовної оптимізації задля покращення ефективності рендеру та навантаження на графічний процессор.

Пракична цінність полягає у – створенні унікального візуального досвіду, яке дозволяє користувачам насолоджуватися естетичним досвідом ігрового застосунку. Рішення забезпечує рендеринг в реальному часі. Це важливо для забезпечення плавності та ьного досвіду та підвищеної ефективності створює оптимальні умови для задоволення потреб користувачів та покращення їхньої взаємодії з ігрою.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Розробка рішення для рендеру трави під рущій Unity, швидкодійність кінцевого рішення та унікальний візуальний стиль за допомогою графічного програмування звичайних та обчислювальних шейдерів.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2023 - 30.09.2023
Теоретична частина	01.10.2023 - 31.10.2023
Розробка та оптимізація рішення	01.11.2023 - 12.12.2023

Завдання видав: ____________ *Приходченко С.Д.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ____________ *Корнієнко Д.А.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *15.12.2023р.*

РЕФЕРАТ

Пояснювальна записка: 68 стор., 22 рис., 1 таблиця, 2 додатка, 35 джерел.

Об’єкт досліджень – процеси обробки та відображення великої кількості об’єктів.

Предмет досліджень – методи оптимізації ренедеру об’єктів.

Методи дослідження – Для розробки шейдера були використані методи та засоби програмування, орієнтовані на графічний процесор та створення шейдерів. Також були впроваджені методи оптимізації та підвищення ефективності рендерингу об'єктів.

Розроблено шейдер для візуалізації трави із виразним графічним стилем, а також розроблено засіб для оптимізації шейдера на рушії Unity.

Практична цінність – результатів полягає у тому, що в результаті проведеного дослідження було розроблено засіб для рендеру трави в унікальному графічному стилі, який створювати поля трави.

Область застосування – розроблене рішення можна застосовувати у розробці ігор для створення успішного досвіду взаємодії з грою.

Список ключових слів: ШЕЙДЕР, РЕНДЕР, ТРАВА, ВІЗУАЛІЗАЦІЯ, UNITY, ГРАФІКА, ІГРИ, ОПТИМІЗАЦІЯ.

ABSTRACT

Explanatory note: 68 pages, 22 figures, 1 table, 2 appendices, 35 sources.

The object of research is the processes of processing and displaying a large number of objects.

The subject of research is methods of optimizing the rendering of objects.

Research methods – To develop the shader, programming methods and tools focused on the graphics processor and the creation of shaders were used. Methods of optimization and increasing the efficiency of object rendering were also implemented.

A grass rendering shader with a distinct graphic style has been developed, and a tool to optimize the shader on the Unity engine has also been developed.

The practical value of the results is that, as a result of the research, a tool for rendering grass in a unique graphic style was developed to create grass fields.

Scope – the developed solution can be applied in game development to create a successful game interaction experience.

Keyword list: SHADER, RENDER, GRASS, VISUALIZATION, UNITY, GRAPHICS, GAMES, OPTIMIZATION.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

CG – computer graphic;

FPS – frames per second;

HLSL – high level shader language.

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ та постановка задачі досліджень 10
1.1. Актуальність теми дослідження та аналіз актуальності поставленої задачі 10
1.2. Призначення розробки та область застосування 11
1.3. Огляд термінології графічного програмування 12
1.4. Постановка задачі 16
РОЗДІЛ 2 ФОРМУВАННЯ ВИМОГ ЗА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ 17
2.1. Мета вирішення задачі 17
2.2. Оптимізація процесу 23
РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНИХ РІШЕНЬ 30
3.1. Розробка шейдера 30
3.2. Оптимізація рендеру 40
ВИСНОВКИ 52
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 53
Додаток А. Лістинг програми 57
ДОДАТОК Б. ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ 68

ВСТУП

Розробка відеоігор — це складний і трудомісткий творчий процес, що вимагає від розробників не тільки високого рівня експертизи, але й здатності ефективно використовувати ресурси. Сучасні ігри — це мистецтво створення вражаючих інтерактивних світів, де сплітаються найновітніші технології, захоплюючі механіки та передові графічні рішення.

У світі відеоігор, візуал виконує ключову роль у враженні гравців. Графічний стиль не лише задає естетичний тон грі, але й визначає її загальний характер та настрій. Вдало підібраний графічний стиль може стати справжньою визначальною рисою і допомогти грі вирізнятися серед конкурентів. Іноді саме він може стати тим фактором, що залишиться в пам'яті гравців найтриваліше.

Проте, несучи на собі завдання зробити гру візуально захопливою, розробники часто стикаються з проблемою збереження продуктивності. Велика кількість графічних ефектів та деталей може вплинути на швидкодію, що є критичним чинником для багатьох гравців. Тому виникає потреба в уважному балансуванні між вражаючим візуальним виконанням та оптимізацією продуктивності гри.

Кожен розробник прагне зробити свою гру унікальною, наділеною естетичністю. Але цей шлях не лише веде до творчих висновків, але і поставляє перед нами труднощі. Спроба додати таку банальність як траву стає невдячним завданням, оскільки це може призвести до проблем з продуктивністю. Рендерити поле з 10 000 об'єктів може бути викликом, але що, якщо їх 100 000 чи навіть 1 000 000? Здійснення оптимізацій — це наш шлях розв'язання цих труднощів. Тож давайте проведемо дослідження та впровадимо необхідні оптимізації для ефективної реалізації ігрового світу.

Вирішенням даної проблеми розглядається створення високоефективного шейдера для ігрового рушія Unity.

Кваліфікаційна робота розглядається в трьох ключових частинах.

У першому розділі проводиться аналіз предметної області, визначається актуальність поставленої задачі та призначення проекту. Окрім цього, робиться огляд базової термінології графічного програмування, особливостей мови шейдерів та виокремлюються вимоги до розробки.

Другий розділ присвячений проведенню досліджень. В ньому буде детально розглянуто, як працює шейдер, а також будуть розглянуті техніки оптимізації рендеру.

У третьому розділі надається інформація про сам процес розробки та оптимізації шейдера, описуються конкретні кроки та вирішені завдання.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ та постановка задачі досліджень
1.1. Актуальність теми дослідження та аналіз актуальності поставленої задачі

Історія розробки трави в іграх є важливим етапом у виникненні та розвитку графічного дизайну та програмування в ігровій індустрії. З самого початку існування відеоігор візуалізація природного ландшафту, включаючи траву, не здобувала належної уваги розробників. Проте, із зростанням можливостей технічного обладнання, програмісти стали активно працювати над створенням більш реалістичних та ефективних методів відтворення трави в ігрових світах.

На початкових етапах розвитку галузі програмісти використовували прості текстури трави, які виглядали статично та обмежено взаємодіяли з оточенням. Запровадження 3D-графіки внесло нові можливості у використанні тривимірних моделей трави, але це також викликало виклики в області забезпечення ефективності та оптимізації гри.

Бажання досягти реалістичності та поліпшення графічних ефектів підштовхнули розробників до різноманітних технологічних інновацій. Різноманіття підходів до відтворення трави варіювало від простих методів до більш сучасних, таких як використання шейдерів. Зокрема, виникла тенденція до використання шейдерів для створення реалістичних динамічних ефектів росту та руху трави.

Запровадження шейдерів відкрило нові можливості для створення деталізованих та динамічних ефектів трави. Використання шейдерів для симуляції хвильового руху трави під впливом вітру чи руху гравця стало популярним спрямуванням у розвитку графічного дизайну. Це сприяло появі реалістичних візуальних деталей, що збагачують іммерсивність ігрового досвіду.

Однією із значущих тенденцій у розвитку графіки трави стала впровадження технік procedural generation. Замість заздалегідь визначених текстур, застосування алгоритмів для генерації текстур трави дало можливість створювати непередбачувані та унікальні ландшафти, надаючи грі новий рівень реалізму. Такий підхід дозволяє створювати велику кількість варіацій, забезпечуючи гравцям унікальний ігровий світ.

Розробка ефективних алгоритмів для рендеринга трави є ключовим етапом в створенні реалістичних ігрових ландшафтів. Програмісти вивчають різні підходи до оптимізації відображення трави, враховуючи при цьому різні відстані та умови освітлення для досягнення оптимальної графічної якості. Це вимагає неперервної розробки та вдосконалення алгоритмів для відтворення природної краси трав'янистих покривів у різних умовах.

Шейдери грають визначальну роль у симуляції різних ефектів, зокрема хвильового руху трави. Вони дозволяють створювати реалістичні анімації, де кожна травинка рухається відповідно до внутрішніх логік і впливу навколишнього середовища. Використання шейдерів дозволяє досягти природнього та живого вигляду трави під час геймплею.

Оптимізація обчислювальних витрат є необхідною для підтримки високої продуктивності гри на різних платформах. Розробники вдосконалюють алгоритми та використовують різні техніки, такі як LOD (Level of Detail) та асинхронний рендеринг, щоб забезпечити плавну та ефективну роботу гри, незважаючи на об'єм обчислень, необхідних для відображення трави.

1.2. Призначення розробки та область застосування

Призначення розробки – задоволення потреб розробників ігор, що працюють над створенням вражаючих і унікальних віртуальних світів. Це

рішення спрямоване на поліпшення творчого процесу та розширення можливостей створення віртуальних фаун і флор в ігровому середовищі.

Однією з ключових особливостей розробленого рішення є його спроможність допомагати розробникам створювати унікальні світи з живою фауною. Завдяки передовим технологіям, що входять в це рішення, створення реалістичних та захоплюючих ігрових просторів стає більш доступним і ефективним завдяки широкому набору інструментів та ресурсів.

Рішення дозволяю розробникам швидше втілювати свої ідеї та концепції у віртуальних світах.

Розроблене рішення допомагає оптимізувати використання ресурсів розробників, забезпечуючи ефективні інструменти для створення ігрових об'єктів. Це робить процес розробки ігор більш економічним та швидким, дозволяючи фахівцям у сфері розваг зосередитися на творчості та концептуальному проектуванні.

1.3. Огляд термінології графічного програмування

Робиться огляд базової термінології графічного програмування та особливостей мови шейдерів
Розробка шейдерів для Unity – це таємничий світ, де майстерність зустрічається з творчістю, але є одна важлива визначна точка, яка робить цей шлях менш заплутаним.

На першому етапі створення шейдерів необхідно мати чітке з чим маємо справу.

Unity – це рушій для створення ігор та інтерактивних віртуальних середовищ. Воно є одним з найпопулярніших інструментів у галузі розробки від інді-гравців до великих студій. Unity підтримує багато

платформ, включаючи комп'ютери, консолі, мобільні пристрої та віртуальну реальність.

Unity надає зручні інструменти та інтерфейс для взаємодії та програмування шейдерів, щоб розробники могли створювати вражаючі графічні ефекти та вигляд об'єктів у своїх проектах.

Шейдер (від англ. – Shader) – це програма, яка використовується на одному з етапів графічного конвеєра в тривимірній графіці. Її завданням є визначення остаточних параметрів об'єкта чи зображення під час процесу рендерингу. Шейдери визначають вигляд та властивості графічних об'єктів, контролюють освітлення, кольори, текстури та інші аспекти візуального відображення.

Для розробки звичайних шейдерів в юніті використовують ShaderLab.

ShaderLab є мовою опису для написання шейдерів в Unity. Це власний мовний інтерфейс, розроблений Unity, який дозволяє програмістам і художникам визначати зовнішній вигляд об'єктів у грі. ShaderLab використовується для створення матеріалів, які управляють тим, як об'єкти відображають світло та кольори в грі. Сам по собі він є мостом між мовами написання шейдерів та юніті дозволяючи налаштовувати параметри через інспектор.

У Unity, матеріали визначають, як об'єкт буде виглядати, і вони часто містять шейдери. Матеріали визначають властивості, такі як колір, текстури, прозорість, блиск та інші параметри.

ShaderLab підтримує дві мови програмування шейдерів CG та HLSL.

CG (C for Graphics) – це мова програмування для написання шейдерів, яку підтримує Unity. Вона є високорівневим абстрактним шаром над мовами програмування шейдерів, такими як HLSL або GLSL. CG дозволяє розробникам створювати шейдери, які можуть бути використані в різних графічних API, таких як DirectX або OpenGL.

HLSL – це мова програмування шейдерів, яка використовується для написання шейдерів у середовищі розробки Unity. Вона є частиною DirectX API та служить для опису обчислювальних і графічних шейдерів. HLSL забезпечує високорівневий інтерфейс для визначення, як поведуться пікселі та вершини при обробці графіки.

Також, Юніті підтримує Compute Shader.

Compute Shader – це тип шейдера, який використовується для виконання обчислень на графічному процесорі, а не для відображення графіки. Вони часто використовуються для паралельних обчислень, таких як обробка даних великого обсягу або обчислення фізичних ефектів. Цей тип шейдеру відкриває нові перспективи для розробників, дозволяючи їм ефективно використовувати графічний процесор для широкого спектру завдань, які вимагають великої обчислювальної потужності. Він стає необхідним інструментом для створення складних та продуктивних графічних додатків.

Щоб шейдер працював належним чином, необхідні дані меша, які виступають в ролі вхідних даних.

Меш (від англ. – Mesh) у графіці та комп'ютерних іграх є основною структурою для визначення геометричної форми 3D-об'єкта. Він складається з ряду елементів, які визначають форму та вигляд об'єкта у тривимірному просторі.

Розглянемо, з чого саме складається меш:

- вершини (vertices): основними будівельними блоками меша є його вершини. Кожна вершина представляє точку в просторі і має координати (x, y, z), що визначають її положення;
- ребра (edges): ребра – це лінії, які з'єднують вершини і визначають структуру об'єкта. Вони визначають, як вершини пов'язані між собою;

– трикутники (faces): майже всі меші складаються з трикутників. Трикутники формуються за допомогою трьох вершин та їхніх з'єднаних ребер. Трикутники визначають поверхню об'єкта;

– нормалі (normals): кожен трикутник має вектор нормалі, який вказує на напрямок відповідної поверхні. Нормалі важливі для визначення, як світло взаємодіє з поверхнею та для розрахунку тіней;

– текстурні координати (texture coordinates): для накладання текстур на поверхню меша визначаються текстурні координати. Вони вказують, як текстура буде нанесена на поверхню трикутника;

– матеріал: інформація про матеріал, що використовується для відображення меша. Включає в себе параметри, такі як колір, прозорість, блиск тощо;

– UV-координати: використовуються для визначення, як текстура буде розміщена на поверхні меша, тобто вони визначають взаємозв'язок між текстурою та геометрією;

– скелет (skeleton): у складних моделях може бути використано скелетну анімацію, і меш може містити скелет (або арматуру), яка дозволяє моделі рухатися та анімуватися;

– анімаційні дані (animation data): якщо об'єкт анімований, меш може містити дані про анімацію, такі як ключові кадри для зміни форми або положення.

Узагальнюючи, меш визначає форму та структуру 3D-об'єкта, а його компоненти визначають різні аспекти, такі як геометрія, текстури, матеріали та інші характеристики.

1.4. Постановка задачі

Задача роботи полягає у досліджені як різні методи оптімізації впливають на рендер об’єктів для створення рішення для рендера та анімації трави в ігрових застосунках, для чого треба вирішити наступні завдання:

1. Провести аналіз предметної області.
2. Провести аналіз процесу створення шейдера та анімації трави. Розглянути етапи розробки та втілення анімаційного ефекту для трави.
3. Провести докладний огляд методів оптимізації, спрямованих на ефективне відображення значної кількості об'єктів.
4. Розробити шейдер для реалізації унікального візуального стилю відображення трави.
5. Розробити шейдер для процедурного генерування трави, що дозволить покривати велику площу травою.
6. Провести оптимізацію процесу відображення з метою ефективного використання ресурсів графічного процесору, забезпечуючи оптимальну продуктивність.

РОЗДІЛ 2
ФОРМУВАННЯ ВИМОГ ЗА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1. Мета вирішення задачі

Занурімося у світ шейдерів та вивчимо, як вони функціонують та як їх створювати.

Щоб розробити шейдер у ShaderLab, необхідно визначити його та надати йому ім'я, використовуючи відповідний блок:

Властивості шейдера, такі як кольори, текстури чи числові значення, визначаються в розділі Properties. Ці властивості дозволяють користувачу налаштовувати шейдер безпосередньо в інтерфейсі Unity.

У Unity кожен шейдер складається з переліку сабшейдерів. При відображенні мешу Unity вибирає відповідний шейдер та використовує перший підтримуваний сабшейдер для графічної карти користувача. У випадку, якщо певні платформи не підтримуються за замовчуванням, але вимагають підтримки, для кожної платформи може бути розроблений власний шейдер.

Кожен сабшейдер включає в себе прохід (Pass), що визначає конкретний етап рендерінгу геометрії.

Давайте докладніше розглянемо, що нам необхідно для нашого шейдера в межах цього проходу.

По-перше, важливо належним чином описати параметр Cull, щоб передати шейдеру інформацію про те, що ми бажаємо рендерити лише передню сторону об'єкта, не витрачаючи зайві ресурси на відрисовку пікселів, розташованих позаду.

1. Cull Front.

Цей параметр дозволяє використовувати ефективну оптимізацію, оскільки лише видима частина об'єкта буде відображена, що сприяє підвищенню продуктивності при обробці сцени.

Далі ми визначаємо мову програмування, обираючи між HLSL або CG. У даному випадку обидва варіанти є прийнятними.

Розглянемо різні типи шейдерів, з якими ми можемо працювати:

1. Surface Shader.

Представляє собою шейдер, спеціально призначений для створення шейдерів. Ця функція отримує невеликий обсяг даних про об'єкт та повертає інформацію, що містить дані про світло, випромінювання, гладкість, прозорість та інші параметри.

В цілому Surface shader підходить для швидкого створення шейдеру, це робить розробку шейдеру значно простіше, але ми не маємо повного контролю над процессом.

2. Vertex shader.

Цей шейдер відповідає за обробку кожного вершини (точки) у тривимірному об'єкті, який рендериться на екрані. Основна функція вершинного шейдера - трансформація координат вершин об'єкта в просторі, а також виконання різних операцій для обробки вершинної інформації перед її передачею на наступний етап графічного конвеєра.

Вершинний шейдер виконує операції над координатами вершин об'єкта для перетворення їх у просторі моделі, виду та проекції. Це дозволяє відобразити об'єкт на екрані та забезпечити перспективу.

В цьому шейдері для оптимізації краще виконувати більшисть операцій, так як кількість вершин зазвичай менше ніж кількість пікселів на екрані, що дозволить мінімізувати кількість викликів.

З цього шейдеру можна передавати інформацію до графічного шейдеру.

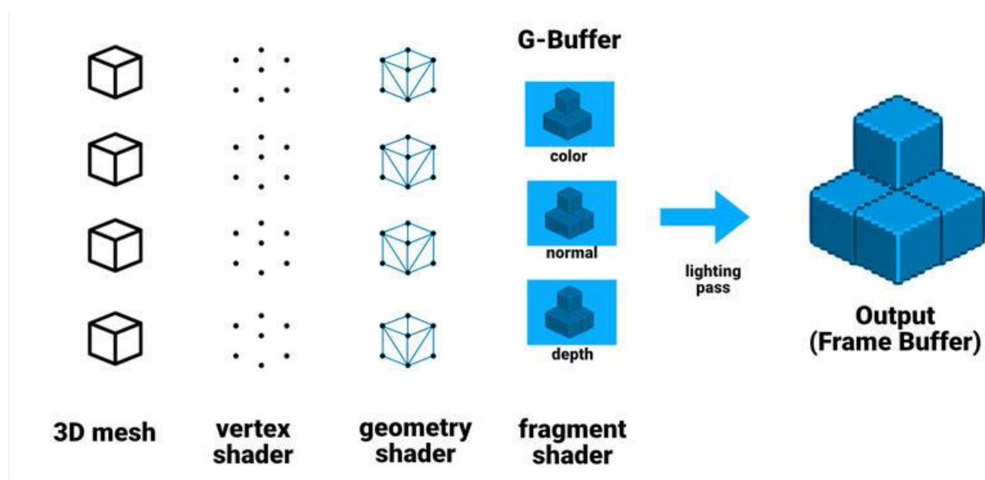

Рис. 2.1.Приклад графічного конвеїру шейдера

3. Fragment Shader.

Цей шейдер відповідає за обробку окремих фрагментів (пікселів) на екрані. Після того, як вершинний шейдер трансформує вершини об'єкта та передає дані на фрагментний шейдер, останній виконує операції, специфічні для кожного пікселя, для визначення його кольору та інших характеристик.

Після виконання всіх необхідних операцій фрагментний шейдер визначає кінцевий колір для кожного пікселя та передає цю інформацію для виведення на екран. Таким чином, вершинні та фрагментні шейдери співпрацюють для створення зображення на екрані.

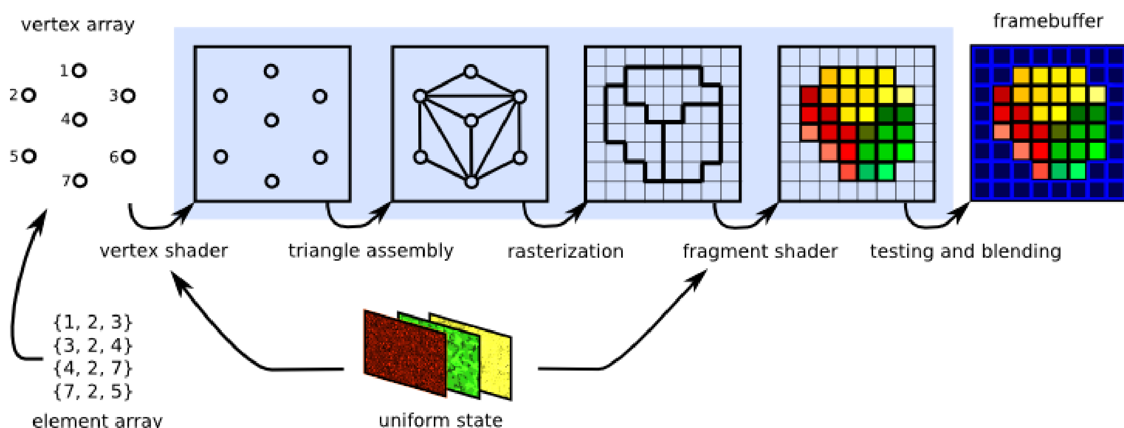

Рис. 2.2. Процесс растеризизаії мешу

4. Geometry shader.

Це тип шейдера у програмному забезпеченні для графічного рендерингу, який дозволяє виконувати операції на рівні геометрії об'єктів під час їх обробки графічним конвеєром.

Геометричний шейдер може генерувати нові геометричні примітиви, такі як точки, лінії або трикутники. Це робить його потужним інструментом для створення нових об'єктів або модифікації існуючих.

Шейдер може змінювати атрибути геометричних примітивів, такі як координати вершин, колір, текстурні координати тощо.

Геометричний шейдер може також взаємодіяти з примітивами, що надходять, і сходжувати їх на більш складні структури, або, навпаки, розділяти примітиви на менші.

Часто використовується для динамічної генерації геометрії, наприклад, для створення трави на поверхні землі, ефектів вогню або інших складних об'єктів.

Геометричний шейдер розташований між вершинним та фрагментним шейдерами у графічному конвеєрі. Це робить його потужним інструментом для обробки та створення складних геометричних об'єктів під час рендерингу.

Цей шейдер часто використовують для створення шейдеру трави, але його ефективність доволі низка, та не всі графічні карти та програмне забезпечення підтримують геометричні шейдери.

Наприклад використовуючи геометричний щейдер ми закриважмо собі можливість розробляти під Mac OS, так як Metal не підтримує геометрічні шейдери.

Виходячи з цієї інформації ми будемо використовувати тільки базові: Vertesx Shader та Fragment Shader.

Якщо ми не можемо використовувати Geometry shader розглянемо варіант який нам допомже.

Compute Shader – це тип шейдера в графічних програмах, який призначений для виконання обчислень на графічних процесорах (GPU). Він входить у склад шейдерів у програмному забезпеченні, що підтримує стандартні графічні API, такі як OpenGL або DirectX. Основним завданням Compute Shader є виконання обчислювальних задач на великій кількості даних паралельно, що дозволяє використовувати потужність паралельного обчислення GPU.

Compute Shader дозволяє виконувати обчислення на тисячах чи навіть мільйонах об'єктів паралельно. Це особливо ефективно для обчислень, які можна розділити на незалежні задачі.
Також має доступ до власної локальної та спільної пам'яті, що дозволяє ефективно використовувати пам'ять на відміну від вершинних та фрагментних шейдерів.

У відміну від стандартних шейдерів, Compute Shader не взаємодіє з процесом відображення графіки. Він призначений виключно для обчислення.

Compute Shader надає велику гнучкість для виконання різноманітних обчислювальних завдань, і він може бути використаний для великої кількості різних застосувань, що виходять за межі графічного рендерингу. Обчислювальна модель Compute Shader базується на робочих групах, які можуть бути організовані в локальні та глобальні групи. Це дозволяє керувати паралельністю та розподілом завдань.

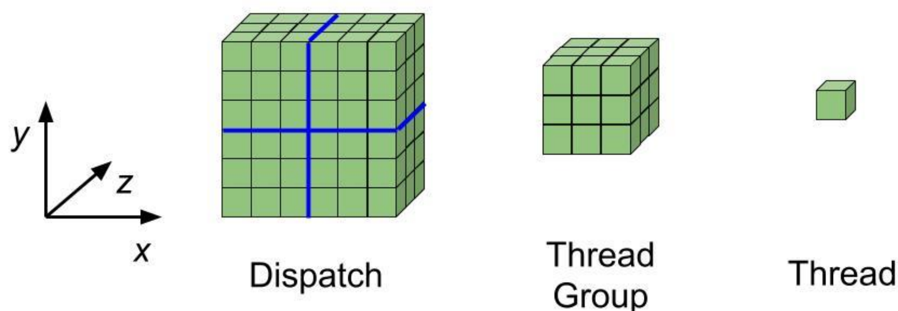

Рис. 2.3. Візуалізація груп Compute shader

Робочі групи (Work Groups) є концепцією, яка визначає організацію і взаємодію потоків в обчислювальних шейдерах, зокрема в Compute Shader. Коли ви використовуєте Compute Shader, ви маєте можливість визначати та керувати робочими групами, які відповідають за паралельне виконання завдань.

Це великі групи, які об'єднують потоки в одній робочій групі, охоплюючи всю область обчислень. Вони визначаються на глобальному рівні та покривають усю задану простір обчислень.

Кожна робоча група містить певну кількість потоків, які виконують обчислення паралельно.

Потоки в межах однієї робочої групи можуть спілкуватися між собою через локальну пам'ять, що дає можливість синхронізації та обміну даними.

Кожна робоча група має свій унікальний ідентифікатор, який може бути використаний для визначення, яка саме група виконує дану роботу.

Також, кожен потік у робочій групі має свій власний унікальний ідентифікатор, який може бути використаний для різних обчислень.

Робочі групи в Compute Shader надають можливість ефективно використовувати паралельні обчислення та управляти роботою потоків у великих обчислювальних завданнях. Ця концепція особливо корисна для задач, де можливо поділити обчислення на незалежні частини, які можуть бути виконані паралельно.

Отже, ми плануємо перемістити основні операції з обчислення позицій та оптимізації в Compute Shader. Це дозволить нам виконувати обчислення лише тоді, коли це необхідно. Отримані результати будемо передавати до шейдеру трави для подальшого використання. Цей підхід дозволяє ефективно використовувати потужність паралельних обчислень на GPU та забезпечує гнучкість у використанні обчислень для оптимізації позицій та інших аспектів.

Але, щоб це все запрацювало, потрібно дослідити, які техніки оптимізації знадоблятся.

2.2. Оптимізація процесу

Оптимізація у світі розробки ігор стає ключовим елементом, який визначає комфорт і задоволення від відтворення віртуальних світів. Важливість цього процесу не може бути недооцінена, оскільки від його успішності залежить здатність гравців насолоджуватися повністю розробленими ігровими універсумами.

Гра, яка швидко реагує та миттєво відгукується на команди гравця, стає основою для іммерсивного досвіду. Оптимізація направлена на те, щоб уникнути будь-яких затримок чи висновків, які можуть розчарувати користувача.

Мета оптимізації полягає не лише у вирішенні конкретних технічних проблем, але й у тому, щоб зробити процес якнайменш помітним для гравця. Оптимізація повинна стати невидимою рукою, яка полегшує гравцеві відчуття поглиблення у світ гри.

Оптимізацію трави в грі може включати в себе розробку технологій, що дозволяють створювати реалістичні та ефективні графічні елементи, не надто обтяжуючи систему гравця.

В процесі оптимізації графіки, розробники можуть шукати способи зменшення навантаження на обчислювальні ресурси, зберігаючи при цьому високий стандарт візуальної якості.

Коли оптимізація успішно впроваджена, гравець може поглиблюватися в ігровий світ, не переймаючись технічними аспектами. Це дозволяє гравцеві максимально відчувати атмосферу та насолоджуватися кожним моментом гри.

Оптимізація в геймінгу – це тонке мистецтво, яке вимагає збалансованості між технічною ефективністю та задоволенням від гри. Розробники, прагнучи до оптимальної продуктивності, створюють засади для того, щоб гравець міг насолоджуватися кожним моментом геймплею, не втрачаючи відчуття іммерсії та задоволення.

Дослідимо, як можна оптимізувати траву в нашому випадку:

1. Distance culling. Техніка оптимізації, яка використовується в комп'ютерній графіці для підвищення продуктивності обробки графічної інформації, особливо в контексті відображення тривимірних сцен. Основна ідея полягає в тому, щоб обчислювати та рендерити графічні об'єкти лише тоді, коли вони знаходяться визначеного відстані від точки огляду (камери) гравця. Об'єкти, які перебувають за межами цієї визначеної відстані, не рендеряться, що призводить до зменшення обсягу обчислень та підвищення продуктивності.

Зазвичай, використовуються формули відстані між камерою та об'єктом для визначення того, на якій відстані вони знаходяться один від одного.

В рендеринговому процесі реалізовано умову, яка перевіряє, чи об'єкт перебуває в межах відстані відсічення. Якщо об'єкт знаходиться за межами цієї відстані, його рендеринг може бути пропущений.

Це дозволяє зменшити кількість об'єктів, які потрібно рендерити, допомагає зменшити навантаження на графічний процесор та покращити продуктивність.

Дозволяє ефективніше використовувати ресурси графічної підсистеми.

Але також може викликати артефакти або неочікувані зникнення об'єктів, якщо великі об'єкти розташовані за межами відстані відсічення.

Вимагає додаткового обчислювального часу для перевірки відстані та визначення видимості об'єктів.

Техніка відсічення за відстанню використовується в багатьох 3D-графічних програмах та іграх для оптимізації відображення об'єктів і забезпечення оптимальної продуктивності при великій кількості графічних елементів.

2. Frustrum culling. Техніка оптимізації, яка використовується в комп'ютерній графіці для підвищення продуктивності шляхом відсічення графічних об'єктів, які не потрапляють у видимий об'єм (frustum) камери. Камера створює конусоподібну область, яку називають видимим об'ємом або frustrum, і всі об'єкти, які не перетинають цей об'єм, виключаються з обчислень рендерингу.

Frustrum — це область, яку бачить камера. У 3D-графіці вона зазвичай має форму піраміди чи конуса і охоплює всі об'єкти, які можливо побачити з певної точки огляду.

Розрахунок frustrum включає в себе обчислення його площі, орієнтації та розташування в просторі. Це може бути виконано на етапі підготовки сцени або під час рендерингу.

Кожен графічний об'єкт перевіряється на те, чи перетинає він видимий об'єм (frustum) камери. Об'єкти, які не потрапляють у frustrum, виключаються з процесу рендерингу.

Існують різні алгоритми для реалізації frustrum culling, такі як алгоритми перетину простору об'єкта з frustrum (наприклад, алгоритм об'єкта-видимого простору або алгоритм Міллера) або використання простих геометричних обмежень.

Це дозволяє значно зменшити кількість об'єктів, які рендеряться, сприяє підвищенню продуктивності графічної системи. Зменшує обсягу обчислень під час рендерингу, що важливо для оптимальної продуктивності.

Але це може призводити до проблем, коли об'єкти частково знаходяться в frustrum, і частина їх не рендериться. Також при збільшені кількості об’єктів збільшується кількість додаткових обчислень для перевірки видимості об'єктів.

Frustrum culling є важливою оптимізацією в розробці комп'ютерних ігор та додатків в галузі візуалізації, допомагаючи забезпечити швидке та ефективне відображення тільки видимих об'єктів.

3. Перенесення обчислень з Fragment Shader до Vertex shader.

Перенесення обчислень з фрагментного шейдера (Fragment Shader) до вершинного шейдера (Vertex Shader) є однією з технік оптимізації графічного рендерингу. Ця техніка спрямована на зменшення обчислювального навантаження на фрагментний шейдер, де обчислюються значення для кожного пікселя екрану, і переміщення частини обчислень до вершинного шейдера, де обчислюються значення для кожної вершини об'єкта.

Фрагментний шейдер виконується для кожного пікселя на екрані після растеризації. Вершинний шейдер виконується для кожної вершини об'єкта перед растеризацією.

Фрагментний шейдер часто використовується для обчислення освітленості, текстур, тіней, анімацій тощо. Оскільки він викликається для

кожного пікселя, може виникнути велика кількість обчислень, що може затримувати швидкодію.

Частина обчислень може бути перенесена до вершинного шейдера. Наприклад, якщо значення однакові для всіх пікселів, можна їх обчислити в вершинному шейдері і передати у фрагментний.

Для передачі обчислених значень від вершинного до фрагментного шейдера використовуються структура.

Це дозволяє ефективно використовувати результати обчислень вершинного шейдера у фрагментному. Зменшити кількість обчислень у фрагментному шейдері. Це допомагає підвищити продуктивність, особливо при великій кількості пікселів на екрані.

Перенос обчислень з фрагментного шейдера до вершинного є частиною стратегії оптимізації, яка включає в себе розсіювання обчислень на різні етапи графічного конвеєра з метою підвищення продуктивності та забезпечення оптимального використання ресурсів.

4. Chunking (розділення). Техніка оптимізації, яка використовується в ігрових двигунах та графічних програмах для поділу сцени або світу на менші частини, які називаються "чанками". Кожен чанк є окремим об'єктом, який рендериться та обробляється незалежно від інших, що може призвести до покращення продуктивності та ефективного використання ресурсів.

Чанк – це фрагмент сцени або світу, який обробляється та рендериться незалежно від інших чанків. Це може бути фрагмент землі, окремий об'єкт або навіть ціла зона гри.

Розділення рендерингу може допомогти розподілити завдання обчислень між процесором та графічною картою, покращуючи взаємодію між цими компонентами.

Чанки дозволяють краще управляти використанням графічним процессом. Лише необхідні чанки оброблются, зменшуючи обсяг работи який необхідно виконати за один кадр.

Розділення рендерингу на чанки є популярною стратегією оптимізації, особливо в ігровій розробці, де дозволяє підтримувати великі та відкриті світи, зменшуючи при цьому навантаження на процесор та графічну підсистему.

5. LOD (від англ. – Level of Details). Техніка оптимізації в графічному рендерингу, яка полягає в зміні рівня деталізації моделей або текстур в залежності від віддаленості глядача або камери. Основна ідея полягає в тому, щоб використовувати менш деталізовані версії об'єктів або текстур, коли вони знаходяться далеко від точки огляду, і використовувати більш деталізовані версії, коли об'єкт стає ближче.

Кожен об'єкт або текстура має кілька рівнів деталізації. Ці рівні можуть бути створені передчасно під час проектування або генеруватися автоматично на основі геометричних або текстурних аналізів.

Визначається відстань, на якій має відбуватися зміна рівня деталізації. Якщо об'єкт знаходиться за межами цієї відстані, використовується менш деталізована версія.

LOD зазвичай залежить від віддаленості об'єкта від камери чи точки огляду. Чим далі об'єкт, тим менш деталізована версія використовується.

Це дозволяє покращити продуктивності при рендерингу великих та складних сцен.

Техніка LOD допомагає оптимізувати рендеринг, особливо у великих ігрових світах або сценах, забезпечуючи баланс між деталізацією та продуктивністю.

6. Indirect Render. Використовується для рендерингу великої кількості екземплярів одного й того ж об'єкта. Ця техніка є частиною стратегії оптимізації, де низькорівневі операції графічного рендерингу

використовуються для забезпечення максимальної продуктивності при великих обсягах графічних об'єктів.

дозволяє рендерити багато екземплярів одного об'єкта, використовуючи тільки один виклик функції. Це значно покращує продуктивність в порівнянні із окремими викликами для кожного екземпляра.

Одним з ключових елементів є використання буфера, який містить дані про екземпляри, такі як їх позиції, обертання, розмір та інші атрибути. Цей буфер, часто визначений як буфер для рендерингу інстансів.

Замість великої кількості окремих викликів для кожного екземпляра графічної моделі, використовується один виклик, що полегшує управління та оптимізацію коду.

РОЗДІЛ 3
ПРАКТИЧНА РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНИХ РІШЕНЬ

3.1. Розробка шейдера

Розпочнемо з найпростішого візуального стилю, який буде використовуватися для відображення трави. Надамо цьому стилю назву, щоб чітко визначити його характер, і оголосимо необхідні властивості для графічного представлення.

Визначимо необхідні властивості, які будуть використовуватися в розробленому шейдері для створення реалістичного відображення трави. Ці властивості будуть ключовими елементами, які визначають зовнішній вигляд трави на графічному інтерфейсі.

Для цього необхідно використати такі кольори:

- _BottomColor
- _MainColorBottom
- _MainColorTop

Візуальний стиль трави буде представлений трьома основними кольорами. Перший колір визначається як нижній тон, який співпадає з кольором землі, на якій росте трава. Такий підхід допоможе уникнути конфлікту між візуальним відображенням трави та землі, забезпечуючи при цьому необхідний рівень контрастності.

Крім того, для досягнення естетичного вигляду трави, використовуватимуться ще два основних кольори, які будуть змішуватися між собою. Ця комбінація кольорів додасть візуальну глибину та різноманіття, роблячи вигляд трави більш привабливим та натуральним.

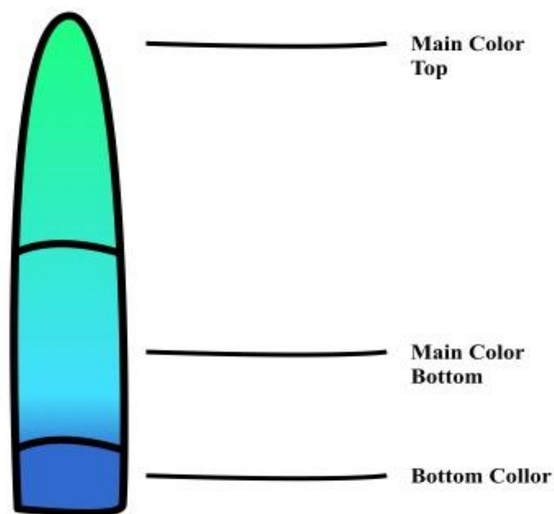

Рис. 3.1. Візуалізація шейдеру трави

Щоб визначити якого коліру буде піксель скористуємося UV кординатами мешу. Скористуємося формулою лінейної інтерполяції.

Ваша формула виглядає на лінійну інтерполяцію між a та b за допомогою параметра w. Я можу записати це у більш математичному вигляді:

$$result = a + w * (b - a)$$

де:

result – отримане значення (результат інтерполяції);

a – початкове значення;

b – кінцеве значення;

w – параметр ваги (від 0 до 1), який контролює ступінь інтерполяції між a та b.

Завдяки цій формулі порахуємо колір пікселю, якщо UV кординати по y, менше ніж 0.2 то зробимо інтерполяцію між _BottomColor та _MainColorBottom, якщо більше між _MainColorBottom та _MainColorTop.

Отриманий результат повернемо як вектор fixed4 фрагментної функції.

Якщо матеріал з цим шейдером додати мешу, отримаємо градієнт з 3-х кольорів. Тож, треба процедурно все розмістити в світі.

Доробимо Vertex функцію.

По-перше, треба додати щоб трава завжди дивилася до камери. Треба зарендрити плоский об’єкт в 3 вимірному просторі.

Щоб це зробити, необхідно скористатися основними матрицями Unity.

− UNITY_MATRIX_V: Це видова матриця, яка відповідає за перетворення координат зі світової системи координат до системи координат камери (виду).

− UNITY_MATRIX_P: Це проекційна матриця, яка відповідає за проекцію тривимірних об'єктів на двовимірний екран.

− UNITY_MATRIX_VP: Це добуток матриць виду та проекції (UNITY_MATRIX_V * UNITY_MATRIX_P). Використовується для перетворення координат зі світової системи до системи координат екрану. UNITY_MATRIX_MVP: Це добуток трьох основних матриць: модельної (Model), видової (View) та проекційної (Projection). Використовується для перетворення вершин від локальних координат моделі до екранних координат.

Якщо перемножити позицію позиції вертексі x та y на UNITY_MATRIX_V, отримаємо кординати які будуть разположені у системі кординат камер, це дозволить рендерити об’єкти повернутими до камери.

Анімація трави в графіці та ігровій розробці може бути виконана різними способами, але зазвичай вона базується на принципах симуляції природних рухів. Є кілька підходів до створення анімації трави:

Створимо анімацю використовуючи Simplex noise. Simplex noise — це одна з варіацій шумових функцій, яка була запропонована Кеном Перліном у 2001 році. Simplex noise є покращеною альтернативою класичному шуму Перліна, оскільки вона володіє кращими характеристиками, зокрема, вона працює швидше та має менше артефактів.

Основна відмінність між класичним шумом та Simplex noise полягає у використанні високоузгоджених симплексів.

Симплекс – це геометрична фігура в просторі вищих вимірів.

Simplex noise ефективніше проявляє себе у вищих вимірах, а також не має проблеми артефактів, які можуть виникнути при певних конфігураціях класичного шуму.

Simplex noise володіє великим періодом, низькими артефактами та вищою збалансованістю, що робить його ідеальним для графіки та симуляцій.

У порівнянні з класичним шумом, Simplex noise працює швидше, особливо в вищих вимірах.

Формула Simplex noise дещо складніша, ніж у класичного шуму, і вона використовує симплекс-структури для створення основи для шумової функції.

Вхідними даними для функції буде позиції кожного вертекса з доданим параметром _Time який репрезентує пройдений час.

Вихідними даними буде число яке перемножаємо на позицію вертекса, але тільки для верхньою частини меша.

Якщо зараз накласти матеріал з цим шейдером плейн, ми отримаємо лише одну травинку.

Далі необхідно розробити рішення для процедурної генерації трави. Для цього доцільно використати ComputeShader.

Процедурне генерування позицій знаходить широке використання в графічних застосунках, особливо при створенні реалістичних ігрових світів чи візуалізації складних сцен.

Один із потужних інструментів для цього – ComputeShader, який дозволяє виконувати обчислення на графічних процесорах.

У процесі використання ComputeShader для генерації позицій, спочатку визначаються параметри, які впливатимуть на результуючий об'єкт. Наприклад, це може бути кількість точок, їхній розподіл, або фактори, що впливають на їхню взаємодію.

Сам ComputeShader визначається алгоритмом, який враховує вказані параметри та генерує координати для кожної точки. Цей процес відбувається паралельно для кожної точки, використовуючи вбудовані функції графічного процесора.

Результатом є набір позицій, який можна використовувати для розміщення об'єктів у тривимірному просторі. Важливою перевагою використання ComputeShader є його висока продуктивність, оскільки він використовує графічний процесор для обчислень, розгружаючи центральний процесор від зайвого наватаження.

Визначимо вхідні дані:

- _Width – параметр який обозначая ширину зони у якій буде трава;
- _Height – параметр який будемо використовувати для перемноження з картою висоти для отримання наскільки треба підняти нашу траву;
- _Density – параметр який означає скільки в одному ряду буде трави;
- _HeightMap – чорно-біла текстура, яка зберігає інформацію про висоту у кольорі (0,0,0) – низина, (1,1,1) – висота.

Використаємо карту, що зображена на рис. 3.2.

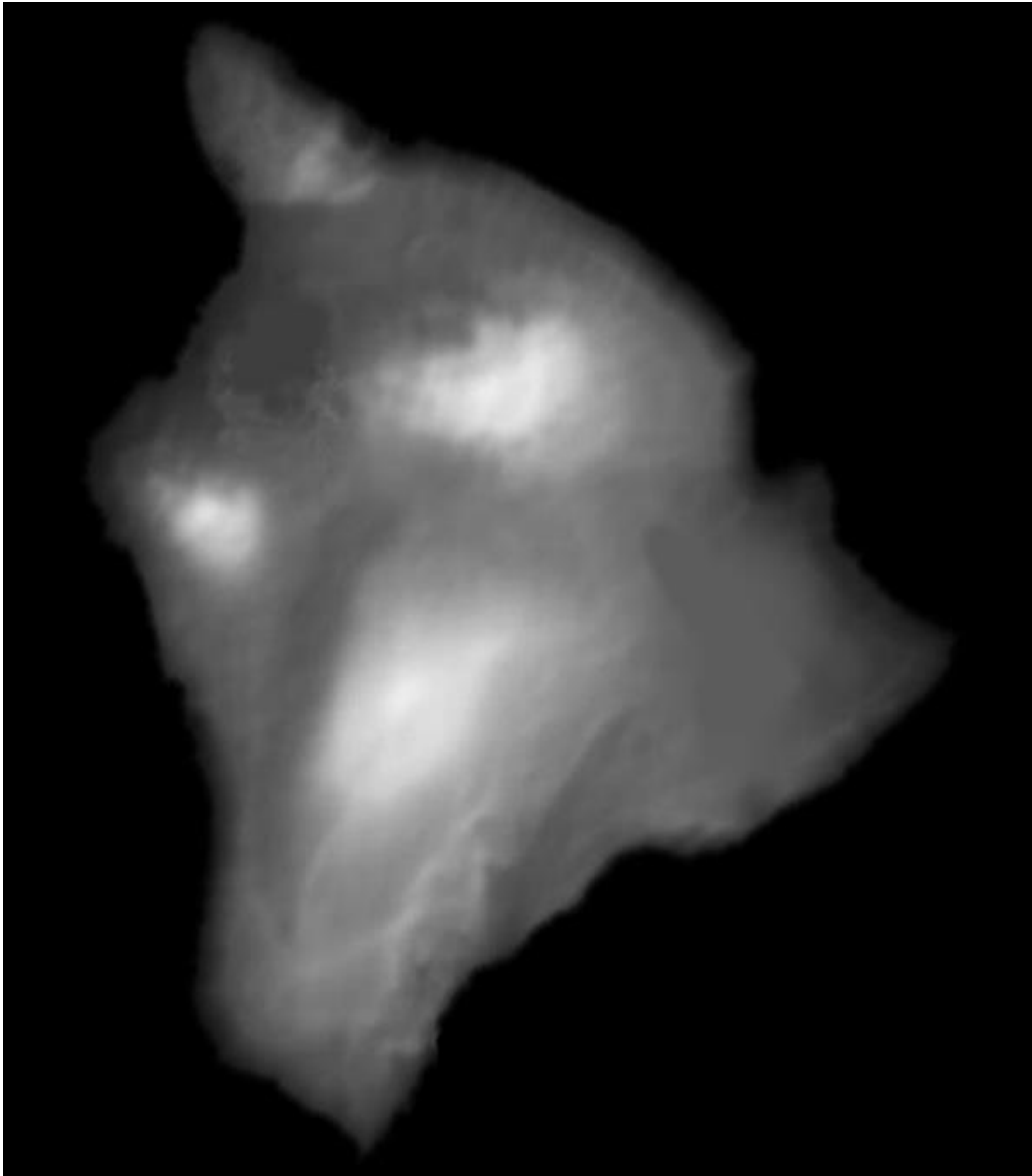

Рис. 3.2. Карта висот

Після цього необхідно порахувати точку для однієї травинки, завдяки обчислювальному шейдеру, ми можемо розбити работу на 8 потоків по X та 8 потоків по Y. Отже, 64 потоки.

Порахувавши позицію кожної травинки як ми ділимо її на ширину зону трави отримуємо нормалізовані значення які будемо використовувати для вирахування висоти за допомогою функції Texture2D.SampleLevel()

Основна мета функції – отримання значення текстури в конкретній точці.

Отримане значення додаємо до висоти позиції та записуємо в StructuredBuffer.

Структурований буфер (Structured Buffer) є одним із видів буферів, що використовуються в програмуванні шейдерів, зокрема в мовах програмування, таких як HLSL (High-Level Shading Language) у контексті DirectX.

Основна ідея структурованого буфера полягає в тому, що він містить елементи певної структури даних. На відміну від звичайних буферів (наприклад, Buffer<float4>), де кожен елемент є одним числом або вектором, структурований буфер може містити складні структури, які визначені користувачем.

Наразі трава виглядає як сітка, де кожен об’єкт рівновіддалений від іншого.

Щоб додати естетики, додамо випадковості. Скористувавшись функцією шума. В нашому випадку це буде Simplex Noise.

Додамо готове рішення яке створює Simplex Noise. Нам потрібна функція яка буде приймати float2 (позицію по осі xz), та перемножимо їх з рандомним значеням сідом для яких буде індитіфікатор потоку xy. Так як функції рандому в HLSL та CG немає, скористуємося шумом перліна.

Він значно простіше та його функцію можно описати формулою:

$$frac(sin(dot(uv, float2(12.9898, 78.233)) \cdot 43758.5453)$$

Отримавши зміщення по осі x та z, додамо їх до позиції траву, та отримаємо рандомну процедурну генерацію трави.

Також, одразу порахуємо Scale. Це потрібно для того, щоб трава виглядала неоднородною та живою.

Щоб відобразити землю для трави, створимо такий самий ComputeShader. Згенеруємо меш.

Першочергово визначається кількість вершин, враховуючи ширину та деталізацію об'єкта. Це вказує на високий рівень деталізації та точності при моделюванні.

Вибіркове створення кожної вершини здійснюється з урахуванням параметрів площини, утворюючи координати відповідних точок у тривимірному просторі. Впроваджений механізм визначення трикутників важливий для формування геометричної структури плейну, а їх розташування відображає високий рівень розуміння топології об'єкта.

Відзначається вправне встановлення нормалей для кожної вершини, вказуючи на спритну обробку аспектів освітлення. Досконало здійснюється також визначення текстурних координат, гарантуючи точне відображення вершин на текстурі.

Завершальним етапом є інтеграція отриманих даних у меш, який є втіленням конструюваного плейну. Цей код є виразом вищого рівня майстерності в галузі розробки тривимірних об'єктів та надає показовий приклад організованого та оптимізованого підходу до створення геометричних структур.

Отриманий меш передаємо в ще один ComputeShader, який також генерує позиції але вже для точок меша.

Для рендеру трави знадобится також сам меш, створимо його в середовищі Blender.

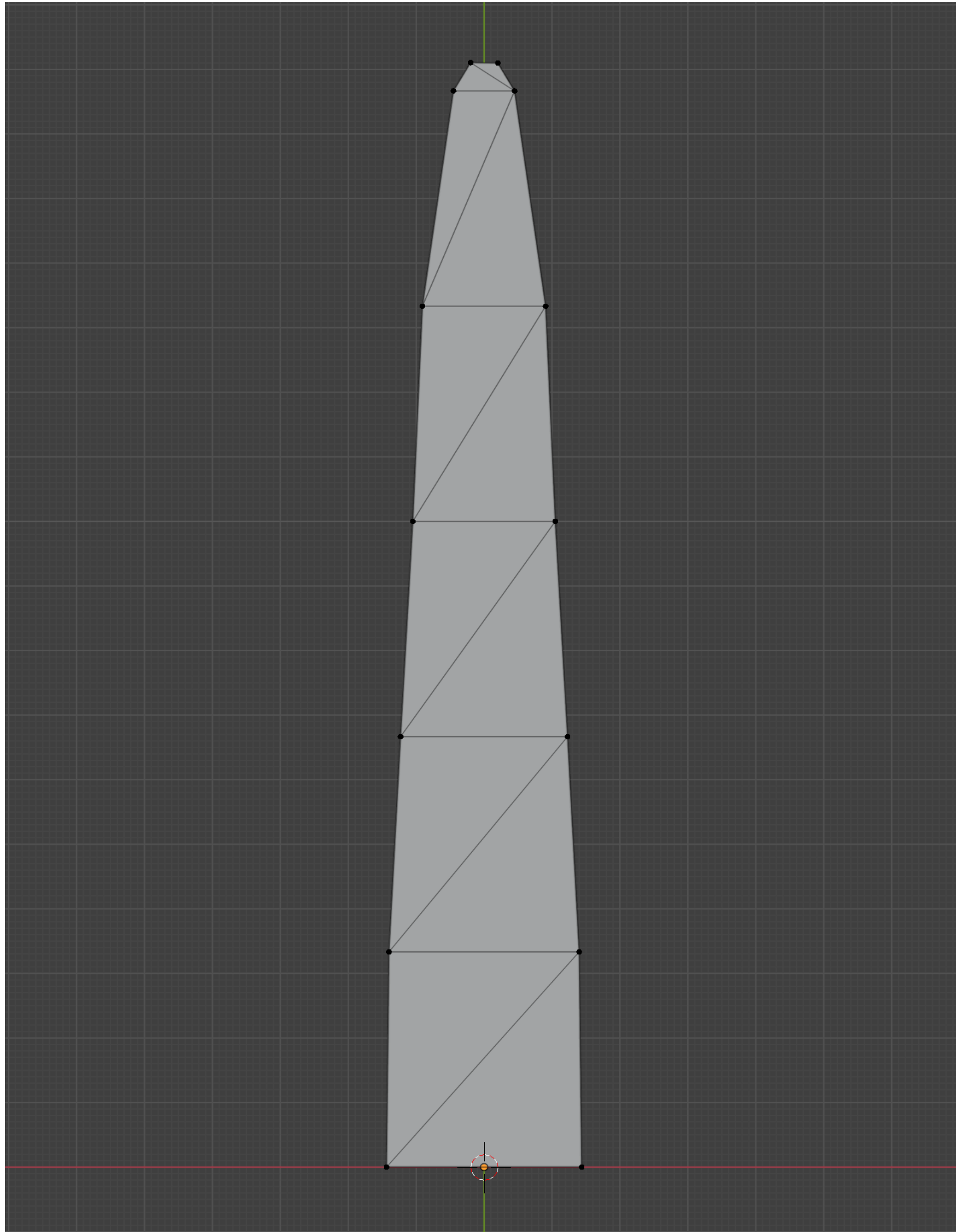

Рис. 3.3 Високополигональний меш трави

Був створений меш з 14 вертекалей – основний меш.

Заспавнимо графіку, як нам пропонує Unity через Instantiate та створення GameObject.

Далі. об’єднаємо все в одномо скритпі процессорі який буде займатися з’єднаням коду та тримаємо перший результат.

При 16384 об’єктах маємо в середньому лише 75 fps.

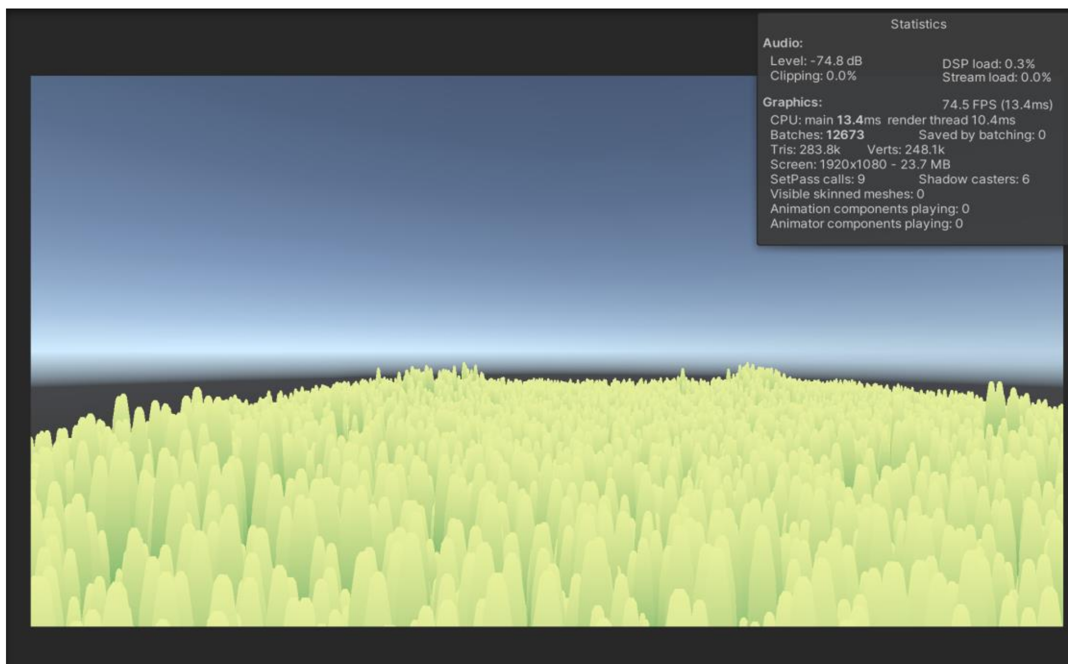

Рис. 3.4. Трава з ігрових об’єктів

Якщо збільшити кільсіть до 1048576, от отримаємо 0.5 fps.

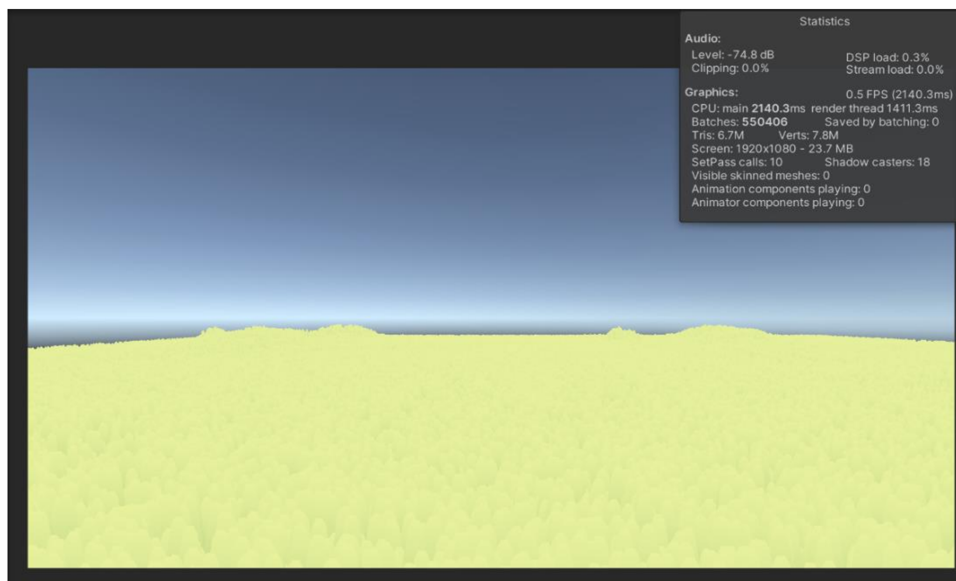

Рис. 3.5.Трава з дуже великої кількості об’єктів

Цей варіант виглядає дуже погано, тому необхідна оптимізація процесу.

3.2. Оптимізація рендеру

Щоб оптимізувати і покращити код, можна відмовитися від використання Game Object у процесі створення плейну та замість цього використовувати метод Graphics.DrawMeshInstancedIndirect. Цей підхід дозволяє збільшити продуктивність та зменшити обчислювальні витрати, шляхом масового малювання мешів.

Graphics.DrawMeshInstancedIndirect – це функція яка дозволяє відтворювати меші (3D моделі) багато разів, використовуючи GPU. Основна ідея полягає в тому, щоб уникнути надмірного навантаження на центральний процесор (CPU) і використовувати потужності графічного процесора (GPU) для ефективного масштабування об'єктів у сцені.

Зазвичай вона використовується з буферами, які містять індекси (наприклад, у вигляді буфера для рендерингу, що містить інформацію про розміщення, обертання і масштаб екземплярів).

Іноді вам не потрібно вручну задавати положення кожного екземпляра. Замість цього ви можете використовувати індиректні буфери, які автоматично генеруються, щоб уникнути надмірної роботи на стороні CPU.

Перепишемо код шейдеру так, щоб він працював з DrawMeshInstancedIndirect. Тепер шейдер буде приймати в вершинну функцію SV_InstanceID, що є унікальним ідентифікатором при його відображенні.

Передамо усередину шейдеру буфер positionBuffer. Звертаючись за допомогою унікального ідентифікатора, отримаємо розраховану позицію та розмір з цього буфера. Потім додамо цю позицію до локальної позиції вершини, щоб змістити його.

Тепер DrawMeshInstancedIndirect створить всі потрібні меші, а шейдер розставить їх по місцям.

Також трохи зменшимо розмір, щоб покращити візуал.

Перевіримо результати цієї оптимізації.

При рендері 4056 об’єктів з попереднім розміром маємо середні 380 fps.

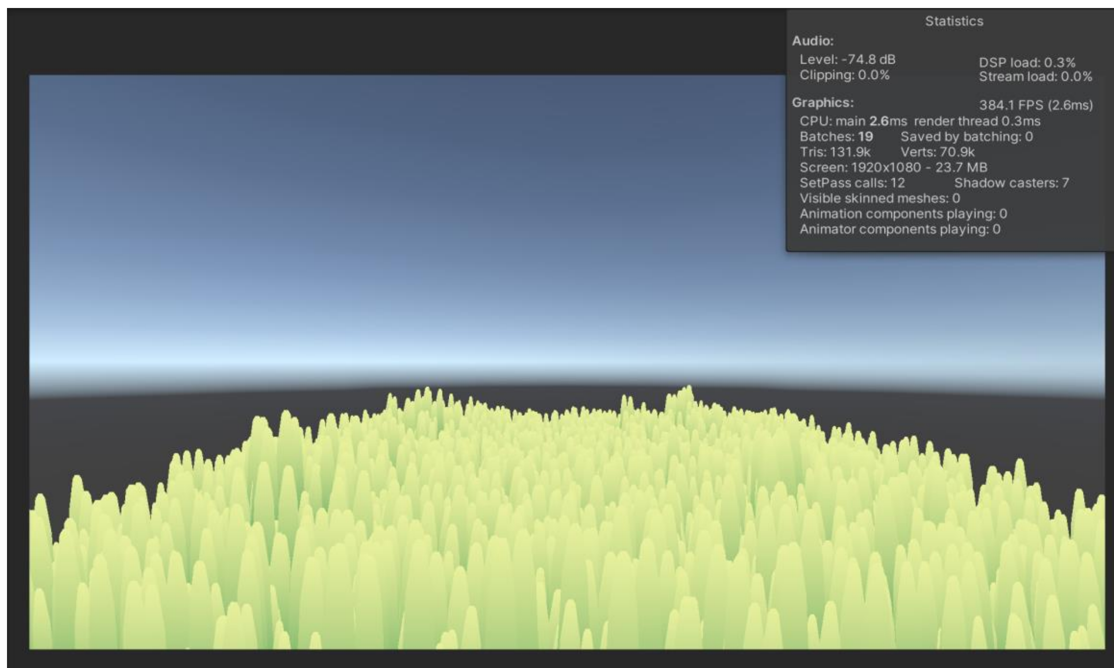

Рис. 3.6. Рендер трави через DrawMeshInstancedIndirect

При рендері 4056 об’єктів з новим розміром маємо середні 380 fps.

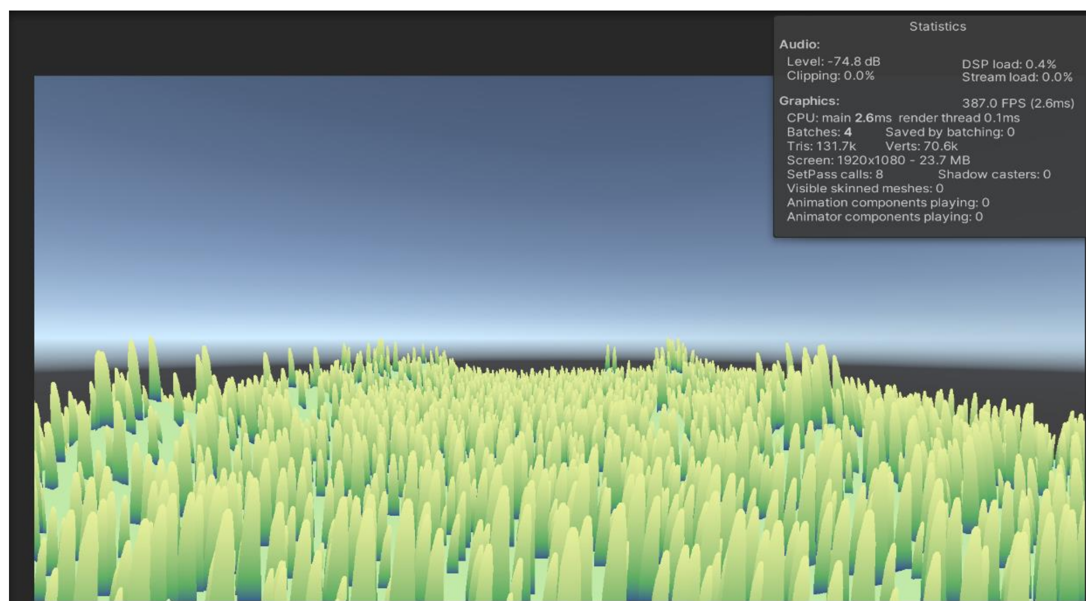

Рис. 3.7. Збільшена площа

При рендері 1048576 об'єктів з новим розміром маємо середні 100 fps.

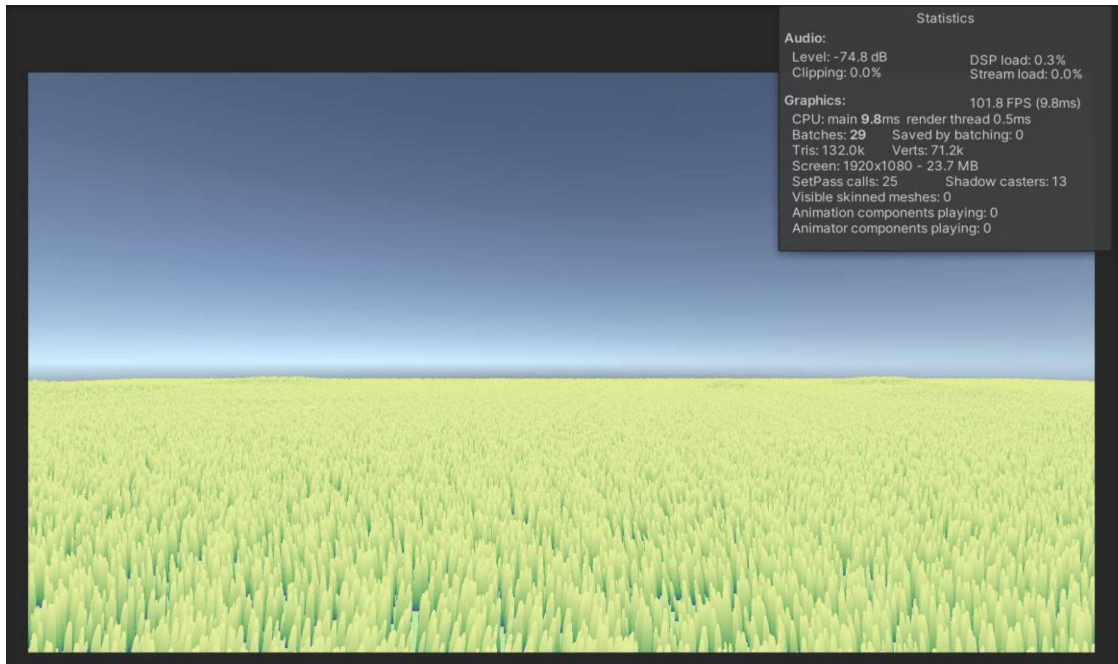

Рис. 3.8. Рендер великої килькоті об'єктів через DrawMeshInstancedIndirect

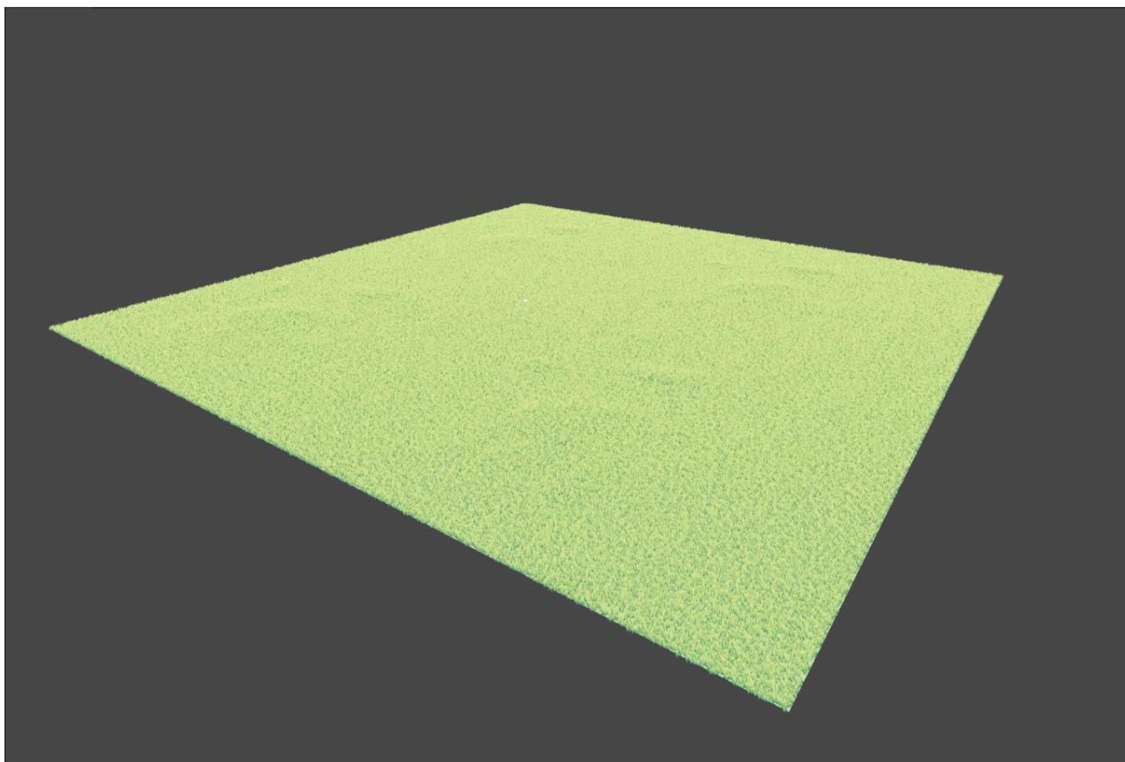

Рис. 3.9. Вид зверху

Отримали значний приріст у продуктивності процесу рендерінгу, який досить істотний для обох випадків використання. У першому випадку, коли кількість об'єктів ще не настільки велика, відзначається приблизно п'ятинний приріст ефективності. У другому випадку цей приріст сягає майже 200 разів.

Цей результат вражаючий і вказує на високу ефективність нашого розробленого рендерингу. Зазначимо, що ми досягли не лише вражаючих показників ефективності, але й розробили унікальний візуальний стиль.

Але ще не кінець, ми можемо оптимізувати ще сильніше. Для цього реалізуємо іншу техніку оптимізації – Frustrum Culling.

Щоб реалізувати Frustrum Culling розробимо новий Compute Shader який буде приймати створений нами раніше буфер позицій. Для кожної позиції проведемо разрахунок, щоб зрозуміти потрібно його відображати чи ні. Для цього переведмо позиції в ClipSpace.

Clip Space (простір обрізання) – це одна з фаз в графічному конвеєрі, яка визначає, як координати вершин графічних об'єктів перетворюються та обрізаються під час їхнього проходження через 3D-простір і перетворення на екранні координати. Після пройдення простору обрізання об'єкти, які можливо вже не знаходяться в області видимості, відсікаються для економії обчислювальних ресурсів та поліпшення продуктивності графічного двигуна.

ClipSpace використовується для нормалізації координат вершин, розташованих у тривимірному просторі, і перетворення їх у прямокутний область, яка відповідає екрану. Цей процес включає в себе використання матриць трансформації та процедур обрізання, які визначають, які частини об'єктів повинні бути відображені на екрані. ClipSpace є важливою частиною графічного конвеєра, яка дозволяє оптимально відображати тривимірні об'єкти на двовимірному екрані.

Перетворивши позицію з WorldSpace до ClipSpace, ми може обробити чи знаходиться цей об’єкт в межах екрану. Якщо отриманий результат від 0 до 1 то ми рендеремо цей об’єкт та навпаки.

Якщо треба рендерети – додаємо ідентифікатор до нового буфера. Цей буфер ми будемо оновлювати кожен кадр. А в шейдері рендерити, брати тільки ті позиції що знаходятся у новому буфері.

Перевіримо отриманий результат.

При рендері 1048576 об’єктів з Frustum Culling маємо середні 140 fps.

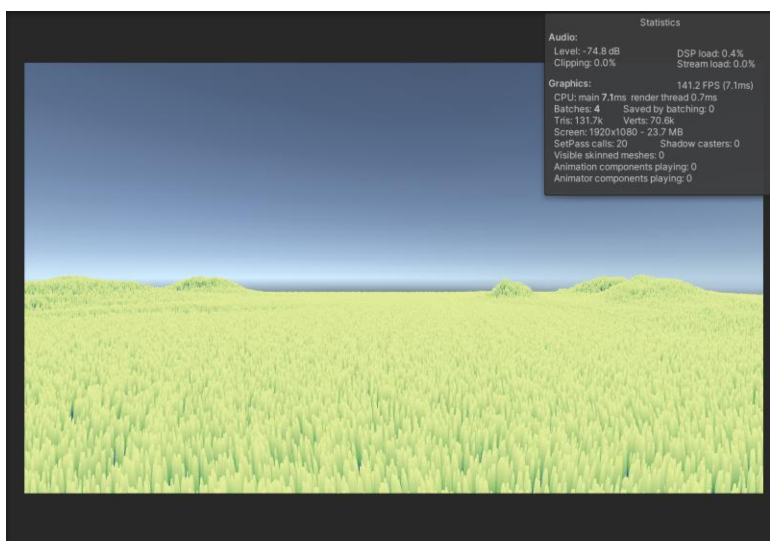

Рис. 3.10. Оптимізація через Frustrum Culliing

Це досягається завдяки тому, що графічний процессор не займається відображенням непотрібних нам об’єктів, а тратить свій ресурс тільки на потрібні.

Якщо переміщати гравця (камеру), цей результат може змінюватися. Наприклад, якщо рендерити менший участок, ефективність цього рендеру збільшится, бо в кадрі буде набагато менше трави.

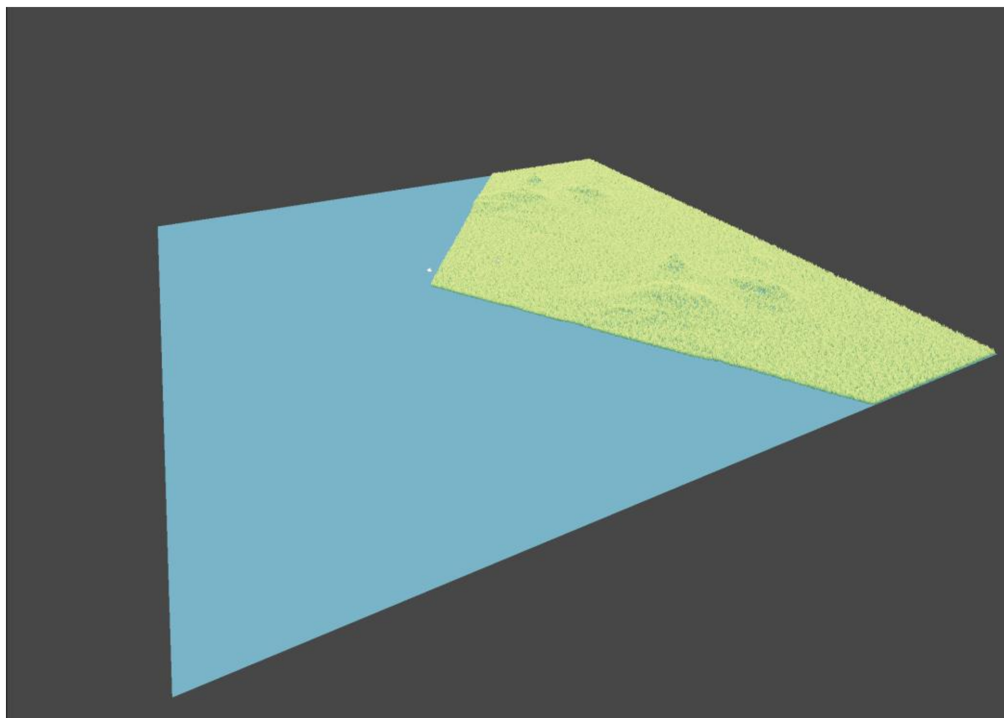

Рис. 3.11. Конус рендера

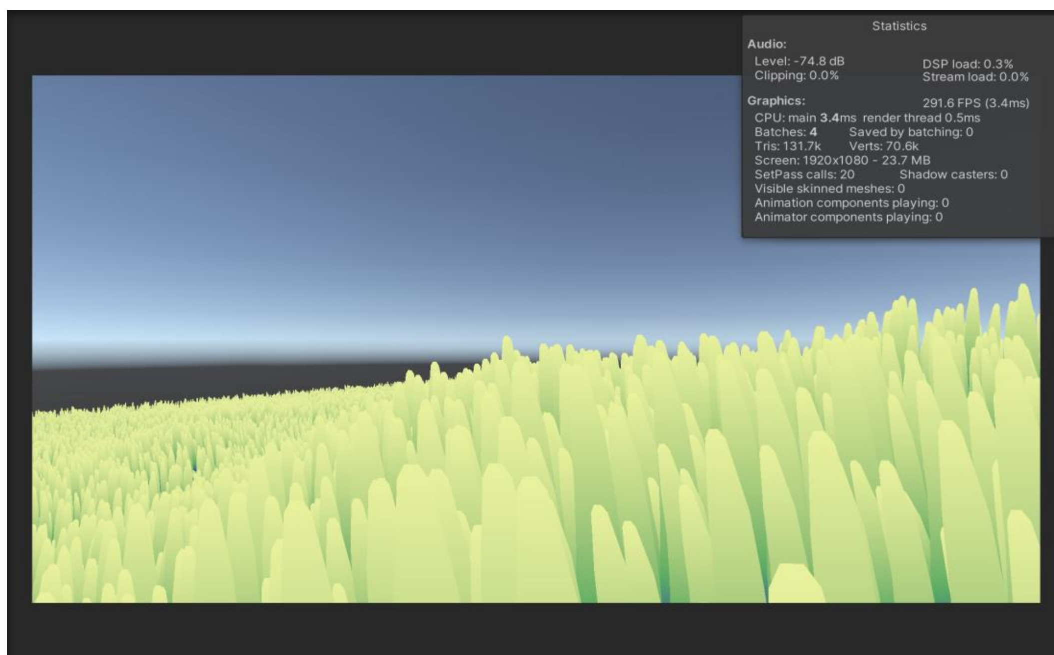

Рис. 3.12.Інший кадр трави

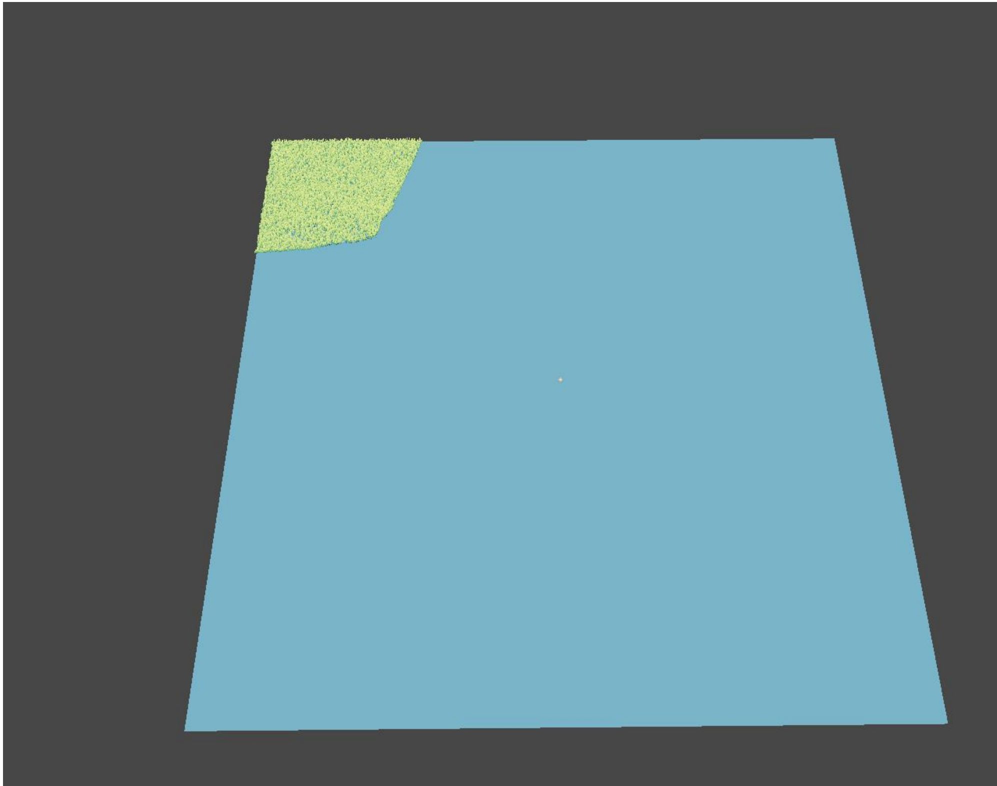

Рис. 3.13. Не рендерить не потрібні частини

Щоб обмежити великі площи трави, скористуємося іншою технікою оптимізації.

Distance Culling – дуже схожий на FrustrumCulling, але огранічує лише дистанцію до якої об’єкти потрібно рендерити, це дозволяє зберегти ресурс на великих участках трави. Додамо перевірку до тогож самого обчислювального шейдеру що займався Frustum Culling. Будемо передавати позицію камери, та обчислювати дістанцію до трави, якщо дистанція більша ніж потрібно то цей об’єкт не буде додан до буферу ідентифікаторів.

Маємо в середньому 180 fps

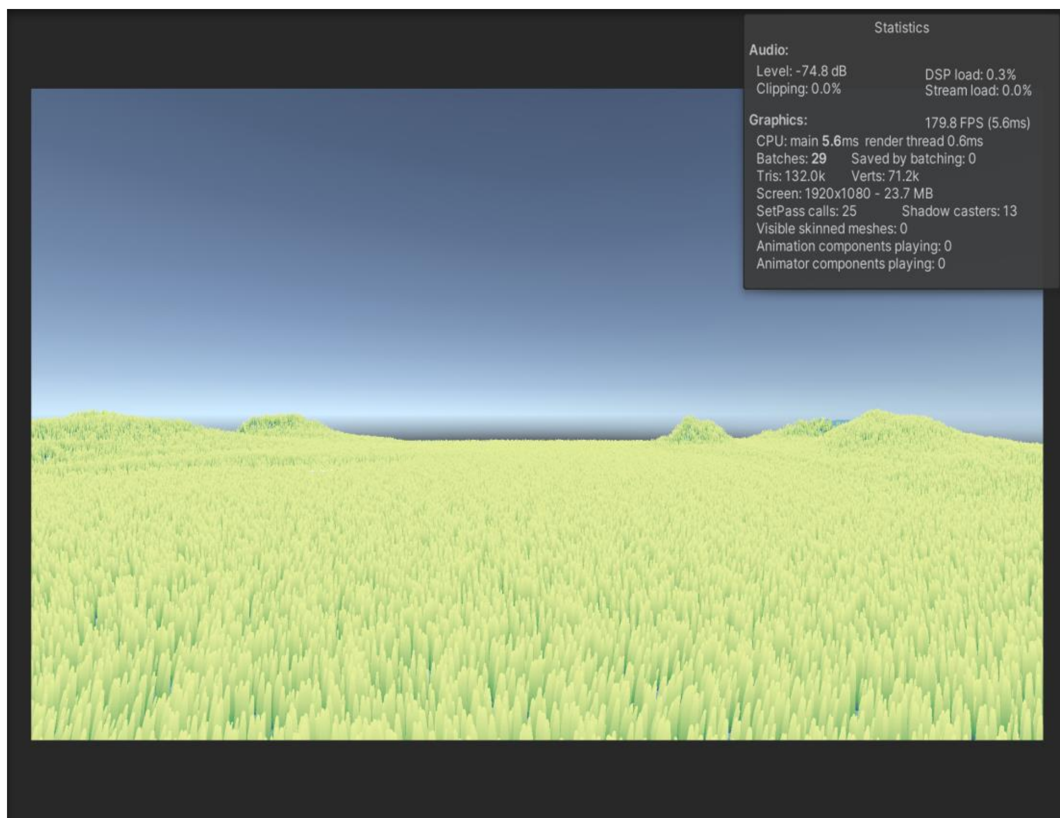

Рис. 3.14. Рендер трави з Distance Culling

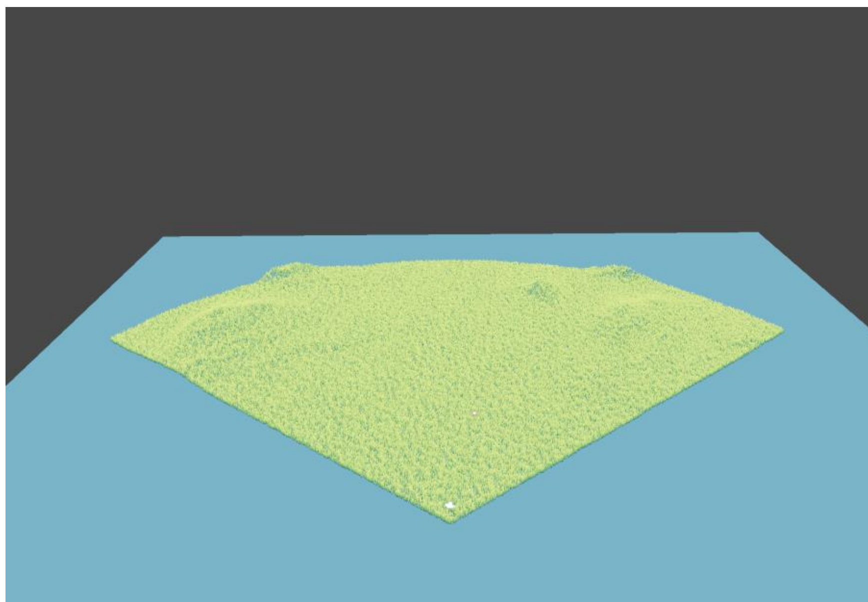

Рис. 3.15. Наглядна візуалізація DistanceCulling

У безперервному погоні за вдосконаленням вже було отримано вражаюче зростання ефективності графічного процесора. Однак, амбіції не

зупиняються тут – насправді, настав час використати потенціал по-максимуму.

Реалізовуємо останні 2 техніки, Chunking та LevelOfDetails.

Поділимо поле трави на сегменти. Кожен сегмент матиме свій матеріал та власні об'єкти ComputeShader для оптимізації та створення трави. Кожен сегмент також відповідатиме за власний рендеринг трави.

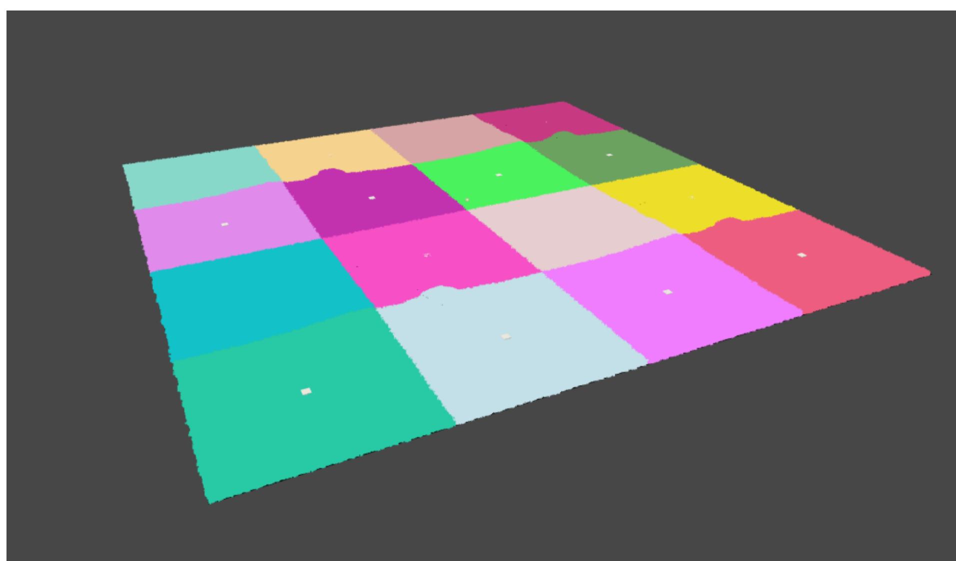

Рис. 3.16. Візуалізація чанків

Відобразимо кожен сегмент із власним унікальним випадковим кольором, щоб переконатися, що кожен сегмент працює правильно. Після розділення сцени на сегменти, ми переходимо до реалізації рівнів деталізації (LOD). В цьому випадку ми моніторимо відстань від камери до кожного сегмента. Залежно від цієї відстані, ми змінюємо меш сегмента, переходячи від високополігональної моделі до низькополігональної.

Цей процес дозволяє оптимізувати відображення об'єктів у сцені залежно від їх віддаленості від камери, забезпечуючи ефективне використання ресурсів та поліпшення продуктивності графічного двигуна. Розробимо потрібний меш в середовищі Blender.

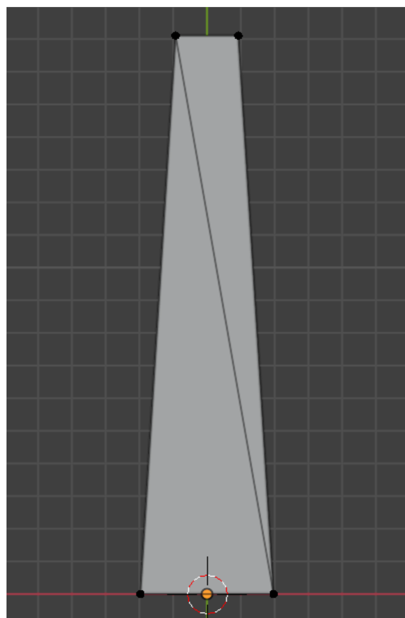

Рис. 3.17. Оптимізований меш

Цей меш складається лише з 4 точок та 2 трикутників. В залежності від відстані, цей меш автоматично замінює основну модель, спрямовуючи на себе менше навантаження на графічний процесор та зменшуючи кількість проходів у вертексному шейдері.

Перевіримо реалізацію LOD. При рендері 1048576 об’єктів маємо середні 220 fps.

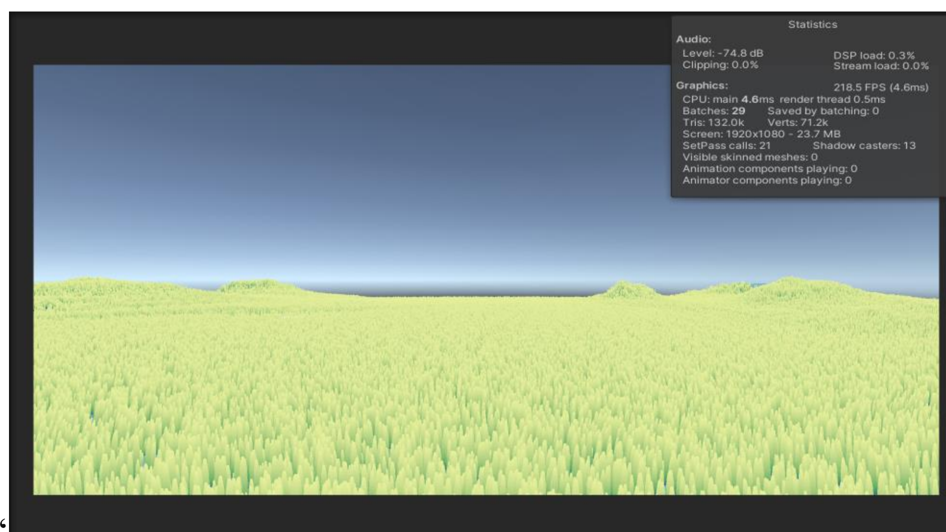

Рис. 3.18. Рендер об’єктів з технікою LOD

Отриманий результат виглядає вражаюче і важко виявити відмінності між звичайним та оптимізованим мешем на значній відстані. Це вельми значуще, оскільки максимально мінімізується відчуття різниці, забезпечуючи органічний та гармонійний вигляд.

Різниця помітна, тільки якщо поглянути з близької відстані.

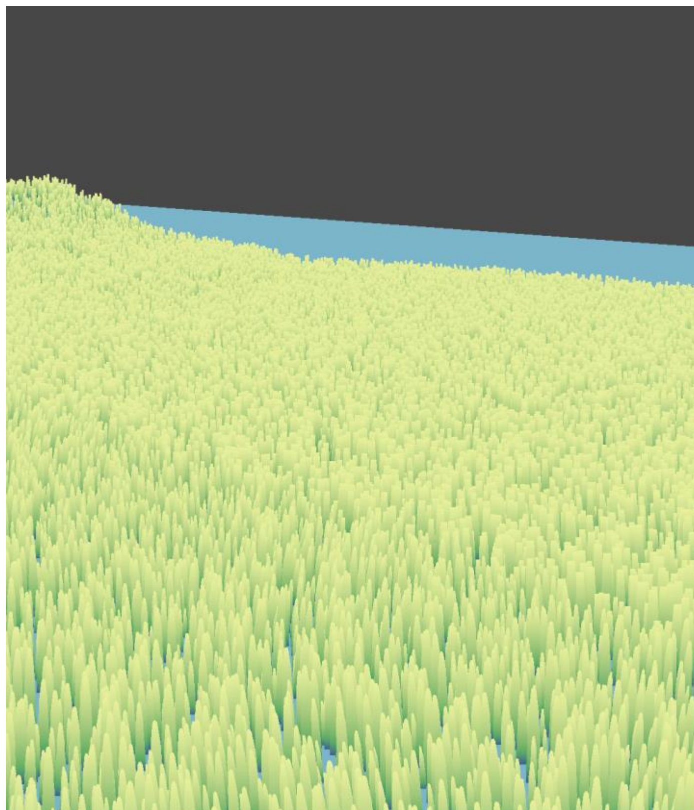

Рис. 3.19. Погляд на звичайний та оптимізований меш

Наші стартови значення при обробці1048576 об’єктів через GameObject – 0.5 fps.

Структурізуємо результати послідовного оптимізування:

Таблиця 3.1.

Таблиця покращення оптимізації з додаванням нових методів

Назва метода оптимізації	Середня кількість кадрів
Indirect mesh instancing	100
Frustum culling	140
Distance culling	180
LOD + Chunking	220

Тести були виконанні на операційній системі Mac OS,на процессорі M1.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було вивчено та досліджено широкий спектр методів створення шейдерів та оптимізації рендрінгу в ігровому середовищі. Це дозволило не лише ознайомитися з основними технічними аспектами графічного програмування, але й розкрити нові можливості для підвищення візуальної ефективності та стилю у гейм-девелопменті.

В рамках дослідження використовувався ігровий двигун Unity. Застосування графічного програмування та обчислювальних шейдерів дозволило досягти вражаючих результатів у відтворенні візуальних ефектів та створенні унікальних графічних рішень. Результат кваліфікаційної роботи може бути використано в розробці ігор. Він принесе до проекту незабуємий візуальний стиль, щоб дати гравцю естетичний досвід, за оптимальну нагрузку на графічний процессор. Набір досліджених варіантів оптимізації може бути використован, в інших проектах, які дають виклик у масштабності, так як підходить до оптимізації основних графічних проблем з якими може стикнутися розробник.

Загалом, це дослідження ілюструють способи та надобність в оптимізації, та показує наскільки може бути великим приріст в совокупности завдяки використанню цих методів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Методичні рекомендації до виконання кваліфікаційних робіт здобувачами другого (магістерського) рівня вищої освіти спеціальностей 121 «Інженерія програмного забезпечення» та 122 «Комп’ютерні науки» / Б.І. Мороз, О.В. Іванченко, О.В. Реута, О.С. Шевцова; М-во освіти і науки України, Нац. техн. ун-т “Дніпровська політехніка”. – Дніпро: НТУ «ДП», 2023. – 55 с.

2. Boulanger, K., Pattanaik, S. N., & Bouatouch, K. (2009). Рендеринг трави в реальному часі з динамічним освітленням. Computer Graphics and Applications, IEEE, 29(1), 32–41.

3. Farin, G. E., & Hansford, D. (2000). Основи CAGD. A K Peters.

4. Fernando, R. (Ред.). (2004). GPU Gems: Техніки програмування, поради та трюки для графіки в реальному часі. Перше видання.

5. Dob Byn, S., Hamill, J., O’Connor, K., & O'Sullivan, C. (2005). Geopostors: Система відображення та рендерингу толп в реальному часі. У SI3D ’05: Звіт про 2005 симпозіум з інтерактивної 3D-графіки та ігор, ACM Press, Нью-Йорк.

6. Advanced Graphics Summit. Процедурна трава в 'Ghost of Tsushima'. Wolff, D. (2018).

7. OpenGL 4 Shading Language Cookbook: Побудуйте високоякісну 3D-графіку в реальному часі з OpenGL 4.6, GLSL 4.6 та C++17, 3-е видання. (3-е видання).

8. Halladay, K. (2019). Практичний розвиток шейдерів: Вершинні та фрагментні шейдери для розробників ігор. (1-е видання).

9. Mike Bailey, Steve Cunningham . (2011). Графічні шейдери: Теорія та практика. (2-е видання).

10. Fosner, R. (2003). Програмування шейдерів в реальному часі (Серія Комп'ютерна графіка від Morgan Kaufmann).

11. JHAT Z. A., MIR A. H. AND RUBAB S. 2011. Fingerprint Texture Feature for Discrimination and Personal Verification. International Journal of Security and its Applications, Vol. 5, No. 3
12. V. Dixit, Deepti Singh, Parul Raj, M. Swathi, P. Gupta, kd-tree based fingerprint identification system 01/2008; DOI:10.1109/IWASID.2008.4688340
13. C. Watson, C. Wilson, M. Indovina, R. Snelick, K. Marshall, Studies of One-toOne Matching with Vendor SDK Matchers. Technical Report NISTIR 7119, July 2004.
14. Орещенко А.В. Способи програмної реалізації тривимірних
15. реалістичних картографічних моделей, 2009.
16. Вурста С.Ю., Літнарович Р.М. Побудова фрактальних поверхонь в комп'ютерній графіці – м. Рівне, 2010.
17. Sebastian Raschka. Python Machine Learning. – UK: Packt Publishing, 2015. – 454p.
18. Henrik Brink. Real-World Machine Learning. – USA: Manning Publications, 2016. – 264p.
19. Ensemble of Clustering Algorithms for Large Datasets / I. A. Pestunov, V. B. Berikov, E. A. Kulikova, S. A. Rylov // Optoelectronics, Instrumentation and Data Processing. – 2011. – Vol. 47, No. 3. – P. 245–252.
20. 12. Fern X. Z. Solving cluster ensemble problems by bipartite graph partitioning / X. Z. Fern, C. E. Brodley // Proceedings of the 21 st International Conference on Machine Learning, Canada, 2004. – P. 47.
21. Ackerman M. Measures of Clustering Quality: A Working Set of Axioms for Clustering / B.-D. Shai, M. Ackerman // Proceedings of NIPS Conference, 2008. – P. 121–128.

22. Unreal Engine 3 brings very expensive dev tools at a very low price . URL: https://arstechnica.com/information-technology/2010/09/unity-3-brings-very- expensive-dev-tools-at-a-very-low-price/ (дата звернення 29.11.2023).
23. Robert C. Martin ’s Principle C ollection. URL: http://principles-wiki.net/collections:robert_c._martin_s_principle_collection (дата звернення 29.11.2023).
24. Tickoo S. Autodesk 3Ds Max: A Comprehensive Guide. CADCIM Technologies, 2021. 674 p.
25. ZBrush Features URL: Pixologic.com (дата звернення 1.12.2023).
26. 16. What is Substance Painter. URL: https://conceptartempire.com/what-is- substance-painter (дата звернення 1.11.2023)
27. 17. Headus UVLayout. URL: https:// www.uvlayout.com/ (дата звернення 1.11.2023)
28. 18. Ultimate Guide to Marvellous Designer with Camille Kleinman. URL: https://discover.therookies.co/2019/09/13/ultimate-guide-to-marvellous-designer-with-camille-kleinman/ (дата звернення 4.11.2023).
29. Jira . URL: Atlassian.com (дата звернення 4.12.2023).
30. 3D Modelling Pipeline. URL:https://medium.com/@homicidalnacho/3d-modelling-pipeline-bd9be7dba136 (дата звернення: 25.11.2023)

31. What is Mixamo and How Can it be Used in Games. URL: https://cgobsession.com/what-is-mixamo-and-how-can-it-be-used-in-games/ (дата звернення 25.11.2023).
32. 8. Albahari J. C++ 10 in a Nutshell: The Definitive Reference. O`Reilly Media, 2022. 1058 p.
33. C#/.NET History Lesson. URL: http://jameskovacs.com/2007/09/07/cnet-history-lesson/ (дата звернення 26.11.2023).
34. 10. Unity architecture. URL: https://docs.unity3d.com/Manual/unity-architecture.html (дата звернення 28.11.2023).
35. Lavieri E. Getting Started with UE5: A Beginner's Guide to 2D and 3D game development, 3rd Edition . Packt Publishing, 2018. 336 p.

Додаток А.

Лістинг програми

```
public class GrassChunkProcessor : MonoBehaviour {
  [Header("Chunk Settings")]
  [SerializeField]
  private int _numChunks = 9;

  [SerializeField]
  public int _planeWidthSegments = 10;

  [SerializeField]
  private int _density = 5;

  [SerializeField]
  public float _planeWidth = 10f;

  [SerializeField]
  private Texture2D _heightMap;

  [SerializeField]
  private float _height = 1;

  [Header("Mesh Generate")]
  [SerializeField]
  private Material _material;

  [SerializeField]
  public int _planeVertCountSegment = 10;

  [SerializeField]
  private ComputeShader _groundCompute;

  [Header("Grass render")]
  [SerializeField]
  private ComputeShader _grassPositionDefinerCompute;

  [SerializeField]
  private Mesh _grassMesh;

  [SerializeField]
  private Mesh _grassMeshLOD;

  [SerializeField]
  private float _lodDistance = 5;

  [SerializeField]
  private Material _grassMaterial;


  [Header("Grass Culling")]
  [SerializeField]
  private ComputeShader _grassCullingCompute;

  private readonly List<Chunk> _chunks = new();

  private MeshFilter _meshFilter;
  private MeshRenderer _meshRenderer;


  public void Start() {
    Init();
  }

  public void LateUpdate() {
    UpdateChunks();
  }

  private void Init() {
    CreateGround();
```

```
    CreateChunks();
  }

  private void CreateGround() {
    _meshFilter = GetComponent<MeshFilter>();
    if (_meshFilter == null) {
      _meshFilter = gameObject.AddComponent<MeshFilter>();
    }

    _meshRenderer = GetComponent<MeshRenderer>();
    if (_meshRenderer == null) {
      _meshRenderer = gameObject.AddComponent<MeshRenderer>();
      _meshRenderer.material = _material;
    }

    GeneratePlane();
    CalculateHeight();
    Recalculate();
  }

  private void GeneratePlane() {
    _meshFilter.mesh = new PlaneMeshGenerator(_planeVertCountSegment - 1,
_planeWidth).GeneratePlane();
  }

  private void CalculateHeight() {
    _meshFilter.mesh.vertices = new MeshHeightMapScaler(_meshFilter.mesh, _groundCompute,
_heightMap, _height).CalculatePositions();
  }

  private void Recalculate() {
    _meshFilter.mesh.RecalculateNormals();
    _meshFilter.mesh.RecalculateBounds();
  }

  private void CreateChunks() {
    var countOfGrass = 0;
    for (var x = 0; x < _numChunks; x++) {
      for (var y = 0; y < _numChunks; y++) {
        CreateChunk(x, y);
        var segments = _planeWidthSegments * _density;
        countOfGrass += segments * segments;
      }
    }

    Debug.Log(countOfGrass);
  }

  private void CreateChunk(int x, int y) {
    var segments = _planeWidthSegments * _density;
    var material = new Material(_grassMaterial);

    var result = new ComputeBuffer(segments * segments, UnsafeUtility.SizeOf<float4>(),
ComputeBufferType.Structured);


    var ChunkWidth = _planeWidth / _numChunks;
    float chunk1 = _numChunks / 2;
    var offsetX = ChunkWidth * x + ChunkWidth * 0.5f;
    var offsetY = ChunkWidth * y + ChunkWidth * 0.5f;

    var dx = ChunkWidth / segments;
    var dz = ChunkWidth / segments;

    var pos = new Vector3();
    pos.x = -chunk1 * ChunkWidth + offsetX;
    pos.z = -chunk1 * ChunkWidth + offsetY;
```

```
    var o = GameObject.CreatePrimitive(PrimitiveType.Cube);
    o.transform.position = pos;


    var grassPositionProcessor = new ChunkGrassPositionProcessor(material, _heightMap, result,
      _numChunks, x, y, segments, _planeWidth, Instantiate(_grassPositionDefinerCompute),
_height);

    var args = new uint[5] { 0, 0, 0, 0, 0 };
    args[0] = _grassMesh.GetIndexCount(0);
    args[1] = 0;
    args[2] = _grassMesh.GetIndexStart(0);
    args[3] = _grassMesh.GetBaseVertex(0);

    var argsLod = new uint[5] { 0, 0, 0, 0, 0 };
    argsLod[0] = _grassMeshLOD.GetIndexCount(0);
    argsLod[1] = 0;
    argsLod[2] = _grassMeshLOD.GetIndexStart(0);
    argsLod[3] = _grassMeshLOD.GetBaseVertex(0);
    var _args = new ComputeBuffer(1, args.Length * sizeof(uint),
ComputeBufferType.IndirectArguments);

    _args.SetData(args);

    material.SetColor("_ChunkColor", new Color(Random.value, Random.value, Random.value, 1));

    var culling = new ChunkCulling(Instantiate(_grassCullingCompute), result, argsLod, args,
_args, material, segments * segments, segments);

    var chunkGrassRenderer = new ChunkGrassRenderer(_grassMesh, _grassMeshLOD, material);


    var chunk = new Chunk(grassPositionProcessor, culling, chunkGrassRenderer, _args, pos,
Camera.main.transform);

    _chunks.Add(chunk);
  }


  private void UpdateChunks() {
    foreach (var chunk in _chunks) {
      chunk.Update();
    }
  }
}

public class ChunkCulling {
    private Material _material;

    private ComputeShader _computeShader;
    private ComputeBuffer _grassPositions;
    private ComputeBuffer _result;
    private ComputeBuffer _args;
    private ComputeBuffer _sizeBuffer;

    private int _size;
    private int _density;
    int kernal;
    private uint[] _argsData = new uint[5] { 0, 0, 0, 0, 0 };
    private uint[] _argsDataNoLod = new uint[5] { 0, 0, 0, 0, 0 };

    public bool Lod = false;

    private Camera _camera;
```

```
    public ChunkCulling(ComputeShader computeShader, ComputeBuffer grassPositions, uint[]
argsDataLod,uint[] argsDataNoLod ,ComputeBuffer args, Material material, int size, int
density) {
        _args = args;
        _argsData = argsDataLod;
        _argsDataNoLod = argsDataNoLod;
        _computeShader = computeShader;
        _grassPositions = grassPositions;
        _material = material;

        _camera = Camera.main;

        _size = size;
        _density = density;
        kernal = _computeShader.FindKernel("Cull");

        _result = new ComputeBuffer( size,UnsafeUtility.SizeOf<uint>(),
ComputeBufferType.Append);
        ResetArgs();

        _computeShader.SetBuffer(kernal, "_GrassPosition", _grassPositions);
        _computeShader.SetBuffer(kernal, "_Result", _result);
        _computeShader.SetBuffer(kernal, "_ArgsBuffer", _args);
    }

    private void ResetArgs() {
        if (Lod) {
           _args.SetData(_argsData);
        } else {
            _args.SetData(_argsDataNoLod);
        }
    }

    public ComputeBuffer GetArgsBuffer() {
        return _args;
    }

    public void Cull() {
        ResetArgs();
        _result.SetCounterValue(0);

        Matrix4x4 P = _camera.projectionMatrix;
        Matrix4x4 V = _camera.transform.worldToLocalMatrix;
        Matrix4x4 VP = P * V;

        _computeShader.SetMatrix("_MATRIX_VP", VP);
        _computeShader.SetFloat("_Distance", 50);
        _computeShader.SetVector("_PlayerPosition",_camera.transform.position);
        _computeShader.SetFloat("_Density", _density);

        int count = _density / 8 ;
        _computeShader.Dispatch(kernal, count, count, 1);

        _material.SetBuffer("idBuffer", _result);
    }
}


public class ChunkGrassRenderer {
    private Material _material;

    private Mesh _grassMesh;
    private Mesh _grassMeshLod;

    public bool IsLod = false;

    public ChunkGrassRenderer(Mesh grassMesh,Mesh grassMeshLod ,Material material ) {
```

```
        _grassMesh = grassMesh;
        _grassMeshLod = grassMeshLod;
        _material = material;
    }


    public void Render(ComputeBuffer args) {



        if (IsLod) {
            Graphics.DrawMeshInstancedIndirect(_grassMeshLod, 0, _material, new
Bounds(Vector3.zero, new Vector3(100.0f, 100.0f, 100.0f)), args);
        } else {
            Graphics.DrawMeshInstancedIndirect(_grassMesh, 0, _material, new
Bounds(Vector3.zero, new Vector3(100.0f, 100.0f, 100.0f)), args);
        }
    }
}

public class ChunkGrassPositionProcessor {
    private Material _material;
    private Texture2D _heightmap;

    private int _planeWidthSegments;
    private float _planeWidth;
    private float _height;

    private ComputeShader _grassPositionDefinerCompute;
    private ComputeBuffer Result;

    private int _chunkSize;
    private int _chunkNumX;
    private int _chunkNumY;

    private Camera _camera;

    public  ChunkGrassPositionProcessor(Material material, Texture2D heightmap,ComputeBuffer
positionBuffer, int chunkSize, int chunkNumX, int chunkNumY,
        int planeWidthSegments, float planeWidth, ComputeShader grassPositionDefinerCompute,
float height) {
        Result = positionBuffer;

        _camera = Camera.main;

        _chunkSize = chunkSize;
        _chunkNumX = chunkNumX;
        _chunkNumY = chunkNumY;

        _material = material;
        _heightmap = heightmap;
        _planeWidthSegments = planeWidthSegments;
        _planeWidth = planeWidth;
        _grassPositionDefinerCompute = Object.Instantiate(grassPositionDefinerCompute);
        _height = height;

        PositionDefinerBuffer();
    }

    public void PositionDefinerBuffer() {
        int kernal = _grassPositionDefinerCompute.FindKernel("DefinePosition");

        Matrix4x4 P = _camera.projectionMatrix;
        Matrix4x4 V = _camera.transform.worldToLocalMatrix;
        Matrix4x4 VP = P * V;

        _grassPositionDefinerCompute.SetMatrix("_MATRIX_VP", VP);
```

```
        _grassPositionDefinerCompute.SetInt("ChunkX", _chunkNumX);
        _grassPositionDefinerCompute.SetInt("ChunkY", _chunkNumY);
        _grassPositionDefinerCompute.SetInt("ChunkNums", _chunkSize);
        _grassPositionDefinerCompute.SetFloat("_Density", _planeWidthSegments);
        _grassPositionDefinerCompute.SetFloat("_Width", _planeWidth);
        _grassPositionDefinerCompute.SetFloat("_Height", _height);
        _grassPositionDefinerCompute.SetBuffer(kernal, "_Result", Result);
        _grassPositionDefinerCompute.SetTexture(kernal, "_HeightMap", _heightmap);

        int count = _planeWidthSegments / 8;
        _grassPositionDefinerCompute.Dispatch(kernal, count,count, 1);

        _material.SetBuffer("positionBuffer", Result);
    }

    public ComputeBuffer GetPositionBuffer() {
        return Result;
    }
}


public class Chunk {
    public ChunkGrassPositionProcessor _grassPositionProcessor;
    public ChunkGrassRenderer _chunkGrassRenderer;
    public ChunkCulling _culling;
    public ComputeBuffer _args;

    public Vector3 _position;
    public Transform _player;

    public Chunk(ChunkGrassPositionProcessor grassPositionProcessor,ChunkCulling culling,
ChunkGrassRenderer chunkGrassRenderer, ComputeBuffer args, Vector3 position, Transform player)
{
        _grassPositionProcessor = grassPositionProcessor;
        _chunkGrassRenderer = chunkGrassRenderer;
        _culling = culling;
        _position = position;
        _args = args;
        _player = player;
        _grassPositionProcessor.PositionDefinerBuffer();

    }

    public void FixedUpdate() {
    }

    public void Update() {
        Vector3 playerScreenPos = Camera.main.WorldToScreenPoint(_player.position);
        Vector3 objectScreenPos = Camera.main.WorldToScreenPoint(_position);

        if (objectScreenPos.z - playerScreenPos.z < -20) {
            return;
        }

        if (Vector3.Distance(_position, _player.position) > 80) {
            return;
        }

        if (Vector3.Distance(_position, _player.position) > 30) {
            _culling.Lod = true;
            _chunkGrassRenderer.IsLod = true;
        } else {
            _culling.Lod = false;
            _chunkGrassRenderer.IsLod = false;
        }
```

```
        _culling.Cull();
        _chunkGrassRenderer.Render(_args);
    }
}

   Shader "Instanced/InstancedShader" {
    Properties {
        _BottomColor ("Bottom Color", Color) = (1, 1, 1)
        _MainColorBottom ("Main Color Bottom", Color) = (1, 1, 1)
        _MainColorTop ("Main Color Top", Color) = (1, 1, 1)

        _Scale("Scale", Float) = 1.0
    }
    SubShader {

        Pass {

            Tags {"LightMode"="ForwardBase"}

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_fwdbase nolightmap nodirlightmap nodynlightmap novertexlight
            #pragma target 4.5

            #include "UnityCG.cginc"
            #include "AutoLight.cginc"
            #include "Simplex.cginc"

            float4 _BottomColor ;
            float4 _MainColorBottom ;
            float4 _MainColorTop ;
            float4 _AmbientColor;

            float _Scale;

            StructuredBuffer<float4> positionBuffer;
           StructuredBuffer<uint> idBuffer;

            struct v2f
            {
                float4 posCord : SV_POSITION;
                float2 uvCord : TEXCOORD0;
            };

            v2f vert (appdata_full v, uint instanceID : SV_InstanceID)
            {
                float4 data = positionBuffer[idBuffer[instanceID]];
                //float4 data = positionBuffer[instanceID];

                const float3 rightCameraView = UNITY_MATRIX_V[0].xyz;
                const float3 upCameraView = UNITY_MATRIX_V[1].xyz;

                float3 localPosition = v.vertex.x  * rightCameraView;
                localPosition += v.vertex.y * upCameraView;

                float3 worldPosition = data.xyz + localPosition * data.w * _Scale;
                worldPosition += rightCameraView * noise(worldPosition.xz * 0.3 + _Time.xy  *
2 ) * 0.05 *  v.vertex.y *  v.vertex.y;

                v2f o;
                o.posCord = mul(UNITY_MATRIX_VP, float4(worldPosition, 1.0f));
                o.uvCord = v.texcoord;
                return o;
            }
```

```
            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 gradient ;

                if(i.uvCord.y < 0.2f)
                {
                    gradient = lerp(_BottomColor, _MainColorBottom, i.uvCord.y * 5);
                }
                if(i.uvCord.y  > 0.2f )
                {
                    gradient = lerp(_MainColorBottom, _MainColorTop, i.uvCord.y-0.2f);
                }

                fixed4 output = gradient;

                return output;
            }

            ENDCG
        }
    }
}

#pragma kernel Cull

#include "UnityShaderUtilities.cginc"
float4x4 _MATRIX_VP;
float _Density;

StructuredBuffer<float4> _GrassPosition;
RWStructuredBuffer<uint> _ArgsBuffer;
AppendStructuredBuffer<uint> _Result;

float _Distance;
float4 _PlayerPosition;

bool Vote( float4 position) {
    float4 viewspace = mul(_MATRIX_VP, position);

    float3 clipspace = viewspace.xyz;
    clipspace /= -viewspace.w;

    clipspace.x = clipspace.x / 2.0f + 0.5f;
    clipspace.y = clipspace.y / 2.0f + 0.5f;
    clipspace.z = -viewspace.w;

    bool inView = (clipspace.x < -0.2f || clipspace.x > 1.2f || clipspace.z <= 0.0f ? 0 : 1)
&& distance(_PlayerPosition.xyz, position.xyz) < _Distance;

    return inView;
}

[numthreads(8,8,1)]
void Cull (uint3 id : SV_DispatchThreadID)
{
    float4 position = _GrassPosition[id.x * ( _Density) + id.y];


    if(Vote(float4(position.xyz, 1)))
    {
        InterlockedAdd(_ArgsBuffer[1], 1);
        _Result.Append(id.x * ( _Density) + id.y);
    }
}

#pragma kernel DefinePosition
```

```
#include "Simplex.compute"
#include "UnityShaderUtilities.cginc"
float4x4 _MATRIX_VP;
float4x4 _BaseTransform;
float _Density;
float _Width;
float _Height;

Texture2D _HeightMap;
SamplerState sampler_HeightMap;

RWStructuredBuffer<float4> _Result;

bool Vote( float4 position) {
    float4 viewspace = mul(_MATRIX_VP, position);

    float3 clipspace = viewspace.xyz;
    clipspace /= -viewspace.w;

    clipspace.x = clipspace.x / 2.0f + 0.5f;
    clipspace.y = clipspace.y / 2.0f + 0.5f;
    clipspace.z = -viewspace.w;

    bool inView = clipspace.x < -0.2f || clipspace.x > 1.2f || clipspace.z <= -0.1f ? 0 : 1;

    return inView;
}

float Random(float2 uv)
{
    return frac(sin(dot(uv, float2(12.9898, 78.233))) * 43758.5453);
}

[numthreads(8,8,1)]
void DefinePosition (uint3 id : SV_DispatchThreadID)
{
    float dx = _Width / _Density;
    float dz = _Width / _Density;

    float4 pos = 0.0f;

    pos.x =  id.x * dx - _Width / 2;
    pos.z =  id.y * dz - _Width / 2;

    float noise1 = snoise(float2(pos.xz));
    float noise2 = snoise(pos.xz);

    pos.x += noise1 * Random(id.xy) * 0.2f;
    pos.z += noise2 * Random(id.xy + 1) * 0.4f;

    float randomHeightPercent = 0.6f;
    float randomHeight = Random(id.xy) * randomHeightPercent;
    float height = randomHeight + (1 - randomHeightPercent);
    pos.w = height;

    float2 uv = float2(pos.x/(_Width/2), pos.z/(_Width/2));

    float4 change = _HeightMap.SampleLevel(sampler_HeightMap, uv, 0);
    pos.y = change.r * _Height - 0.015f;

    _Result[id.x * ( _Density) + id.y] = (pos);
}


#pragma kernel CalculateGround
```

```
StructuredBuffer<float3> _Vertices;
StructuredBuffer<float2> _UV;
RWStructuredBuffer<float3> _Result;

Texture2D _HeightMap;
SamplerState sampler_HeightMap;

float _Height;

[numthreads(64,1,1)]
void CalculateGround (uint3 id : SV_DispatchThreadID)
{
    float3 vert = _Vertices[id.x];
    float2 uv = _UV[id.x];
    float4 change = _HeightMap.SampleLevel(sampler_HeightMap, uv, 0);

    _Result[id.x] = float3(vert.x, change.r * _Height, vert.z);
}
```

ДОДАТОК Б.

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Корнієнко.doc	Пояснювальна записка роботи. Документ Word
Диплом_Корнієнко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Корнієнко.pptx	Презентація роботи