НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«На правах рукопису»
УДК 004.2

До захисту допущено:
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою

«Системне програмування і спеціалізовані комп‘ютерні системи»

зі спеціальності 123 «Комп‘ютерна інженерія»

на тему: «Спосіб візуалізації об'єктів з використанням інтерфейсу графічного ядра для ігрового рушія»

Виконав:
студент ІІ курсу, групи КВ-22мп
Сторчило Іван Григорович _____

Науковий керівник:
старший викладач кафедри СПіСКС
кандидат технічних наук,
Коляда Костянтин Вячеславович _____

Рецензент:
доцент кафедри ОТ, кандидат технічних наук,
доцент, Марковський Олександр Петрович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2024 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп‘ютерна інженерія»

Освітньо-професійна програма «Системне програмування і спеціалізовані комп‘ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____2024 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Сторчило Івану Григоровичу

1. Тема дисертації «Спосіб візуалізації об'єктів з використанням інтерфейсу графічного ядра для ігрового рушія», науковий керівник дисертації старший викладач СПСКС Коляда Костянтин Вячеславович затверджені наказом по Університету від «9» листопада. 2023 р. №5217-С

2. Термін подання студентом дисертації 10.01.2024

3. Об’єкт дослідження: системи візуалізації об’єктів в ігровому рушії.

4. Предмет дослідження: способи візуалізації текстур в ігровому рушії.

5. Перелік завдань, які потрібно розробити: опис предметної області досліджень та проблематика існуючих методів візуалізації текстур, розробка власного способу відображення текстур.

6. Перелік ілюстративного матеріалу: презентація (кількість аркушів: 20).

7. Перелік публікацій: Сторчило І.Г., Коляда К.В. Вплив кількості рівнів деталізації на продуктивність в Unreal Engine 4. Прикладна математика та комп’ютинг. ПМК, 2023 : шістнадцята наук. конф. магістрантів та аспірантів, 28–30 листопада 2023 р.; VI Міжнародна наукова конференція «Здобутки та досягнення прикладних та фундаментальних наук XXI століття» (08.12.2023; м. Черкаси, Україна).

9. Дата видачі завдання 22.10.2023

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення літератури за тематикою проєкту	30.09.2023	
2	Розроблення та узгодження завдання магістерської дисертації	05.10.2023	
3	Аналіз існуючих засобів профайлінгу програм	13.10.2023	
4	Підготовка матеріалів першого розділу магістерської дисертації	16.10.2023	
5	Підготовка матеріалів другого розділу магістерської дисертації	20.10.2023	
6	Підготовка матеріалів третього розділу магістерської дисертації	17.11.2023	
7	Підготовка матеріалів четвертого розділу магістерської дисертації	23.11.2023	
8	Підготовка матеріалів доповіді пов'язаної з тематикою магістерської дисертації	25.11.2023	
9	Попередній розгляд магістерської дисертації на кафедрі	20.12.2023	

Студент Іван СТОРЧИЛО

Науковий керівник Костянтин КОЛЯДА

РЕФЕРАТ

Актуальність теми. Сучасні ігри, анімації та симуляції, що створюються, в тому числі, за допомогою ігрових рушіїв, за візуальною складовою все більше стають наближеними до реальності. Для цього необхідно використовувати сучасні методи візуалізації об’єктів, оскільки з використанням застарілих алгоритмів, навіть найпродуктивніші обчислювальні машини не зможуть бути ефективними. Нові методи є більш оптимізованими та пристосованими для роботи з новими технологіями обчислення в сучасному апаратному забезпеченні, тобто їх ККД стає значно вищим. Це особливо помітно у порівнянні їх продуктивності. Наприклад, аналіз використання ресурсів процесора при рендерингу зображення для створення симуляції де неефективне використання ресурсів буде напряму впливати на час виконання задачі. Або ж кількість кадрів в секунду (англ. FPS), якщо мова іде про рендеринг в реальному часі, наприклад, іграх, додатках віртуальної або додаткової реальності. Таким чином, використання сучасних оптимізованих методів візуалізації зекономить як і час обчислень, що зменшує витрати на виконання задач, так і надасть користувачу набагато більш реалістичний продукт, який зможе працювати плавно і на ненайновішому обчислювальному обладнані.

Об’єктом дослідження є системи візуалізації об’єктів в ігровому рушії.

Предметом дослідження є способи візуалізації текстур в ігровому рушії.

Мета роботи: дослідження способів візуалізації об’єктів з використанням ігрового рушія та розробка власного способу відображення рельєфних текстур зі збереженням якості зображення.

Наукова новизна. Запропонований спосіб візуалізації об'єктів з використанням інтерфейсу графічного ядра для ігрового рушія відрізняється тим, що за допомогою векторів нормалі імітує рельєфність об’єкту, і тим самим значно зменшує кількість полігонів та, відповідно, підвищує продуктивність системи візуалізації в цілому.

Практична цінність полягає в можливості значного прискорення роботи системи візуалізації, що дозволить зменшити витрати на рендеринг анімації,

наприклад, при використанні для створення зображень для фільмів або мультфільмів. Для систем, які генерують зображення в реальному часі, в іграх та додатках доповненої реальності, запропонований спосіб дозволить працювати на відносно застарілому обладнанні, що збільшить кількість користувачів.

Апробація результатів дисертації. Положення даної роботи та проміжні результати доповідались і обговорювались на наступних конференціях:

- Існуючі та використовувані способи візуалізації об’єктів та результати їх оптимізації були розглянуті на наукових конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2023 (Київ, 28-30 листопада 2023 р.)
- VI Міжнародна наукова конференція «Здобутки та досягнення прикладних та фундаментальних наук XXI століття» (08.12.2023; м. Черкаси, Україна).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків. Повний обсяг дисертації – 95 сторінок, у тому числі 75 сторінки основного тексту, 22 рисунки, 21 слайд презентації.

У вступі розглянуто області використання ігрових рушіїв та сучасні способи рендерингу реалістичного зображення.

У першому розділі розглянуто існуючі способи візуалізації різноманітних об’єктів у ігрових рушіях. Описані їх переваги та недоліки, приклади використання та базові концепції роботи.

У другому розділі описані схожі способи спрощення об’єктів.

В третьому розділі викладена теорія роботи запропонованого способу та вплив освітлення на відображення текстури.

В четвертому розділі описані інструменти роботи та порівняні результати роботи розробленого способу та існуючого, зауважені переваги та недоліки кожного зі способів.

Ключові слова: ігровий рушій, візуалізація об’єктів, рендеринг, комп’ютерна графіка.

ABSTRACT

Actuality of theme. Modern games, animations, and simulations created, among other things, with the help of game engines are becoming more and more close to reality in terms of their visual component. To achieve this, it is necessary to use modern methods of object visualization, since even the most productive computers will not be able to be effective using outdated algorithms. New methods are more optimized and adapted to work with new computing technologies in modern hardware, meaning that their efficiency is much higher. This is especially noticeable when comparing their performance. For example, analyzing the use of CPU resources when rendering an image to create a simulation where inefficient use of resources will directly affect the task execution time. Or the number of frames per second (FPS) when it comes to real-time rendering, such as games, virtual or augmented reality applications. Thus, the use of modern optimized visualization methods will save both computing time, which reduces the cost of performing tasks, and provide the user with a much more realistic product that can run smoothly and on the latest hardware.

The object of the research is object visualization systems in a game engine.

The subject of the research is ways to visualize textures in a game engine.

Purpose: to study the ways of visualizing objects in the game engine and to develop our own way of displaying relief textures while maintaining image quality.

The scientific novelty of the work The proposed method of visualizing objects using the graphics kernel interface for a game engine differs in that it simulates the relief of an object using normal vectors, and thus significantly reduces the number of polygons and, accordingly, increases the performance of the visualization system as a whole.

The practical value lies in the possibility of significant acceleration of the visualization system, which will reduce the cost of rendering animation, for example, when used to create images for movies or cartoons. For systems that generate images in real time, in games and augmented reality applications, the proposed method will

allow working on relatively outdated hardware, which will increase the number of users.

Work approbation. The provisions of this work and intermediate results were reported and discussed at the following conferences:

- Existing and used methods of visualization of objects and the results of their optimization were considered at the scientific conference of undergraduate and graduate students "Applied Mathematics and Computing" PMC-2023 (Kyiv, November 28-30, 2023)
- VI International Scientific Conference "Achievements and achievements of applied and fundamental sciences of the XXI century" (08.12.2023; Cherkasy, Ukraine).

Structure and scope of the work The master's thesis consists of an introduction, four chapters, conclusions and appendices. The total volume of the thesis is 95 pages, including 75 pages of the main text, 22 figures, 21 presentation slides..

The introduction discusses the areas of application of game engines and modern methods of rendering realistic images.

The first chapter discusses the existing methods of visualizing various objects in game engines. Their advantages and disadvantages, examples of use, and basic concepts are described.

The second section describes similar methods for simplifying objects.

The third section describes the theory of the proposed method and the effect of lighting on texture display.

The fourth section the tools and compares the results of the developed algorithm with the existing one, and notes the advantages and disadvantages of each method.

Keywords: game engine, object visualization, rendering, computer graphics.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

FPS (Frames Per Second) – кількість відображених кадрів за 1 секунду.

PBR (Physically Based Rendering) – підхід до рендерингу, що базується на фізичних принципах поведінки світла та матеріалів.

GPU (Graphics Processing Unit) – спеціалізований чіп, що відповідає за обробку та відображення графіки на екрані.

RTX (Ray Tracing Texel eXtreme) - платформа, створена Nvidia, забезпечує трасування променів у реальному часі.

Skeletal Mesh – модифікація Skeletal Animation, що використовується в Unreal Engine. Набір полігонів, з яких складається поверхня скелетної сітки, та ієрархічний набір взаємопов'язаних кісток, які можна використовувати для анімації вершин полігонів.

Undefined Behavior – це результат виконання програми, поведінка якої заздалегідь визначена як непередбачувана.

3Д-модель – це віртуальне або комп'ютерне представлення об'єкта чи сцени у тривимірному просторі, яке включає в себе геометричні та текстурні дані, визначаючи форму, розташування та вигляд об'єкта.

Ігровий рушій (Game Engine) – це програмне забезпечення, спеціально розроблене для створення та управління відтворенням ігрових середовищ, включаючи графіку, фізику, звук і інші елементи, необхідні для реалізації відображення та інтеракції у відеоіграх.

Рендеринг (з англ. Rendering) – процес генерації зображення або відображення графіки на екрані, що включає в себе обчислення та відображення об'єктів, освітлення, тіней та інших аспектів для створення візуальної представленості сцени чи об'єкта.

Фреймбуфер – це частина оперативної пам'яті, що містить растрове зображення, яке керує відеодисплеєм.

ВСТУП

У сучасному світі ігрові рушії почали використовуватися не лише для створення ігор, а й для генерації максимально реалістичних анімацій які неможливо, або вкрай складно створити використовуючи наявні, навіть найсучасніші, вузькоспеціалізовані програми. Це пов’язано з тим, що за останні 10 років простежується надзвичайний скачок в розвитку апаратного забезпечення, що за собою дає безліч нових можливостей для нового, високопродуктивного так якісного програмного забезпечення яке потребує постійного оновлення та підтримки. Нові технології дозволяють створювати ігри, максимально наближені до реальності (симулятори). Вже зараз вони використовуються для навчання пілотів літаків, дронів, хірургів, космонавтів та інших надважливих сучасних професій, де використання реального обладнання є надзвичайно дорогим процесом та навість може загрожувати життю.

Для створення такого високоякісного продукту використовується велика кількість новітніх технологій візуалізації, такі як: RTX [1] – дозволяє відтворювати реалістичні ефекти світла, тіней, відбиття та переломлення в режимі реального часу, PBR – дозволяє отримувати високоякісну графіку з фізично вірними властивостями матеріалів, Global Illumination – техніка, спрямована на моделювання глобального розподілу світла в сцені, Anti-Aliasing – використовується для поліпшення якості зображення. Таким чином ігрові рушії наблизились, а в деяких аспектах, і перегнали конкурентне вузькоспеціалізоване програмне забезпечення яке використовувалось для створення високоякісних анімацій або ж симуляцій.

Дана магістерська робота присвячена дослідженню роботи алгоритмів візуалізації та їх покращенням, що дозволяють отримувати максимально реалістичне зображення в реальному часі.

РОЗДІЛ 1
ОСНОВИ ВІЗУАЛІЗАЦІЇ ОБ’ЄКТІВ В ІГРОВОМУ РУШІЇ

1.1 Загальний алгоритм обчислення сцени

Розробники відеоігор використовують різні техніки для рендерингу сцен, залежно від ігрового рушія та апаратного забезпечення, що використовується. Однак найпоширеніша техніка, яка використовується сьогодні, називається растеризацією або рендерингом у реальному часі.

Під час растеризації 3D-моделі та об'єкти сцени розбиваються на окремі пікселі, а потім за допомогою низки алгоритмів затінюються та текстуруються. Потім ці пікселі збираються в цілісне зображення на екрані. Цей процес відбувається надзвичайно швидко, що дозволяє виконувати рендеринг у реальному часі з високою частотою кадрів, що дуже важливо для відеоігор.

Сучасні ігрові рушії також використовують різноманітні інші техніки, такі як динамічне освітлення, тіні та ефекти постобробки, щоб покращити візуальну достовірність гри. Крім того, деякі ігри можуть використовувати технологію трасування променів для створення більш реалістичного освітлення та віддзеркалень, хоча ця технологія може бути дорогою в обчислювальному плані і не може бути реалізована на всіх апаратних конфігураціях.

Загалом, розробники відеоігор використовують цілу низку методів для створення захоплюючих візуальних ефектів, зберігаючи при цьому високу продуктивність та інтерактивність, необхідну для високої продуктивності.

1.2 Відсічення невидимих об’єктів

Frustum Culling – це важлива техніка оптимізації графічного рендерингу, яка дозволяє збільшити продуктивність графічних програм, обчислюючи та відображаючи лише ті об'єкти, які можуть бути видимими для користувача в даний момент. Ідея полягає в тому, щоб визначити, чи перебувають об'єкти в межах

видимого фрустуму — області у тривимірному просторі, яку бачить камера [2]. Графічне зображення ідеї можна побачити на Рисунку 1.

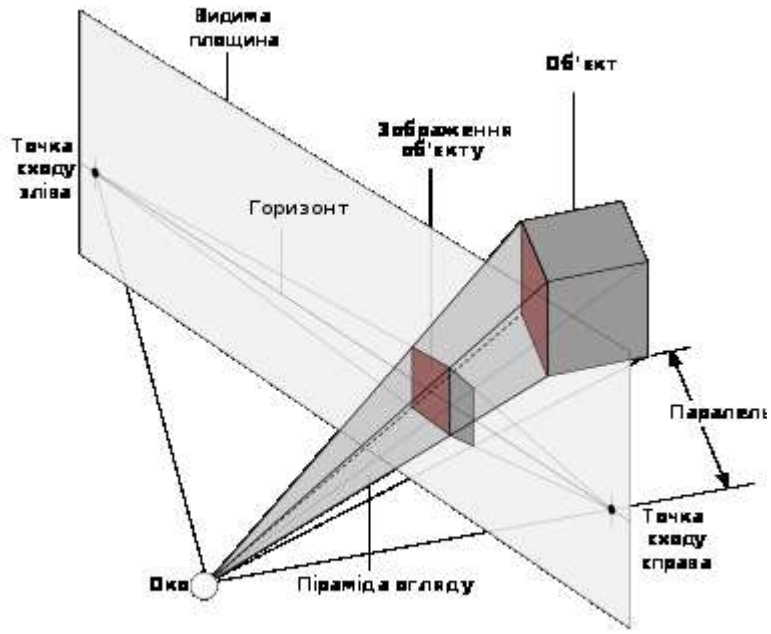

Рисунок 1 – Концепція появи об'єктів в піраміді зору

Процес Frustum Culling розпочинається з отримання матриці проекції та огляду камери, які визначають, як об'єкти перетворюються та відображаються на екрані. Використовуючи ці матриці, обчислюються координати вершин об'єктів у світових координатах. Приклад імплементації цих алгоритмів використовуючи мову програмування C++ та OpenGL на Рисунку 2. Для спрощення реалізації програми, використовується математична бібліотека GLM.

```
// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
const Frustum camFrustum = createFrustumFromCamera(camera, (float)SCR_WIDTH / (float)SCR_HEIGHT, glm::radians(camera.Zoom), 0.1f, 100.0f);
```

Рисунок 2 – Обчислення матриці проекції

Наступний крок — визначення, чи перебувають ці об'єкти в межах видимого фрустуму. Це виконується шляхом перевірки координат кожної вершини об'єкта. Якщо всі вершини знаходиться за межами фрустуму, об'єкт вважається невидимим, і його можна виключити з подальших обчислень та відображення.

Важливо відзначити, що техніка Frustum Culling використовується для всіх об'єктів у сцені, і вона дозволяє відфільтровувати лише ті об'єкти, які знаходяться поза полем зору камери. Це особливо корисно в ігрових додатках та графічних двигунах, де велика кількість об'єктів може призвести до зайвих обчислень та погіршення продуктивності.

Frustum Culling є важливим елементом оптимізації, і його ефективність полягає в тому, як швидко та точно можна визначити, чи об'єкт перебуває в межах видимого фрустуму. Відповідно, в розробці графічних програм використовуються різні методи та алгоритми для максимізації продуктивності Frustum Culling та забезпечення оптимального відображення сцени.

Приклад роботи цього алгоритму можна спостерігати на Рисунку 3(зліва) – камера знаходиться максимально віддалено від куль, щоб захопити максимально можливу площу з наявними об’єктами, на більшій відстані об’єкти не будуть відображатись, оскільки буде перевищена гранична дальність. Кількість об’єктів, що відображується – 118.

На Рисунку 3 (справа) камера знаходиться якнайближче до об’єкту, таким чином, щоб інші об’єкти не попадали в поле фруструму. Можемо бачити – що кількість об’єктів, які були надіслані до GPU для рендерингу – 1.

Цей алгоритм значно знижує навантаження на GPU, але, в той же час, підвищує навантаження на CPU, оскільки для кожного об'єкта потрібно обраховувати чи є він у фруструмі.

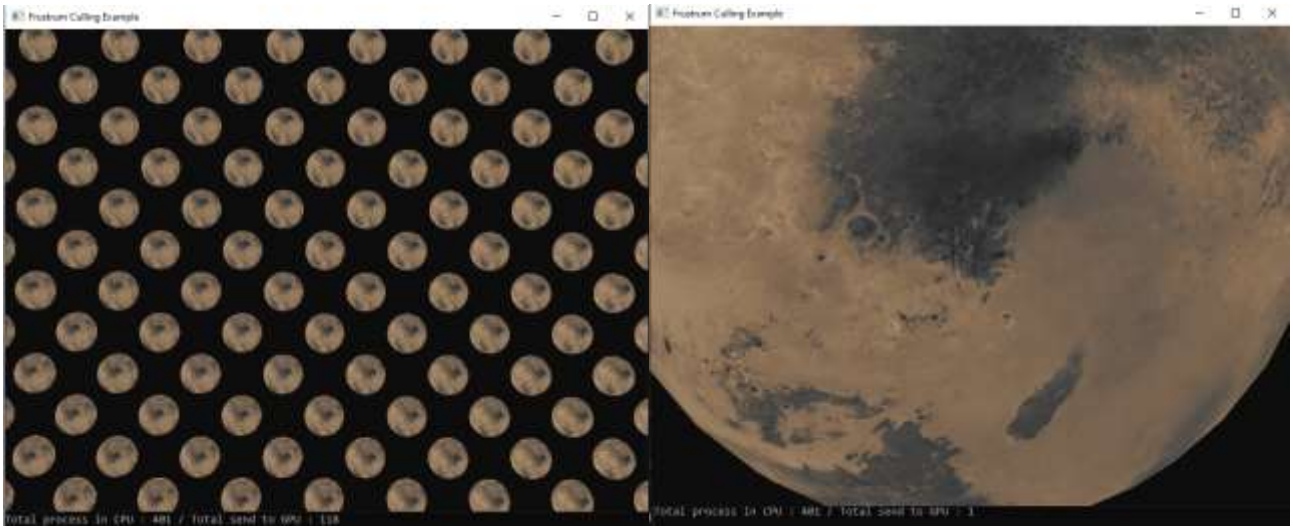

Рисунок 3 – Позиції з яких рендериться максимальна кількість об’єктів та з якої – лише 1 об’єкт

Часто, для цього, використовуються Compute Shaders [4], оскільки вони значно пришвидшують обчислення шаблонних алгоритмів. В той же час, варто звертати увагу, що відправлення даних з пам’яті процесора до пам’яті для обчислень та отримання результатів теж займає немало часу. Тому є сенс використовувати такий спосіб, лише при великій кількості об’єктів на сцені.

1.3 Трасування променів (RTX)

RTX (Real-Time Ray Tracing) представляє собою інноваційну технологію, яка дозволяє в реальному часі відтворювати складні ефекти світла та тіней, використовуючи променеве відстеження. Розроблена компанією NVIDIA, RTX стала ключовим компонентом графічних карт серії RTX, які поєднують в собі графічний процесор (GPU) та тензорне ядро для виконання завдань штучного інтелекту.

Nvidia RTX дозволяє трасування променів у реальному часі. Історично трасування променів використовувалося виключно для завдань, що не вимагали реального часу, таких як виробництво фільмів і відео, архітектурне та дизайнерське проектування, а також для наукової візуалізації, у той час як відеоігри мали

обмежитися прямим освітленням і попередньо розрахованими опосередкованими внесками для їхнього рендерингу. RTX відкриває новий етап у розвитку комп'ютерної графіки, де можливе створення інтерактивних зображень, що реагують на освітлення, тіні та відображення.

Однією з ключових характеристик RTX є здатність реалізовувати просунуті ефекти світла та тіней, які раніше були можливі лише за допомогою обчислювально-інтенсивних методів, таких як променеве відстеження. Дана технологія дозволяє сценам в іграх виглядати більш реалістично та натурально, сприяючи покращенню глибини та деталізації візуального представлення.

Приклади використання RTX стають особливо очевидними у великих та відкритих світах відомих відеоігор. Наприклад, у грі "Minecraft" RTX використовується для втілення реалістичного освітлення та відблисків, що змінюються в залежності від погодних умов та часу доби, Рисунок 4 . В інших іграх, таких як "Control" чи "Cyberpunk 2077", RTX впроваджує складні ефекти взаємодії світла з поверхнею об'єктів, надаючи їм фотореалістичний вигляд.

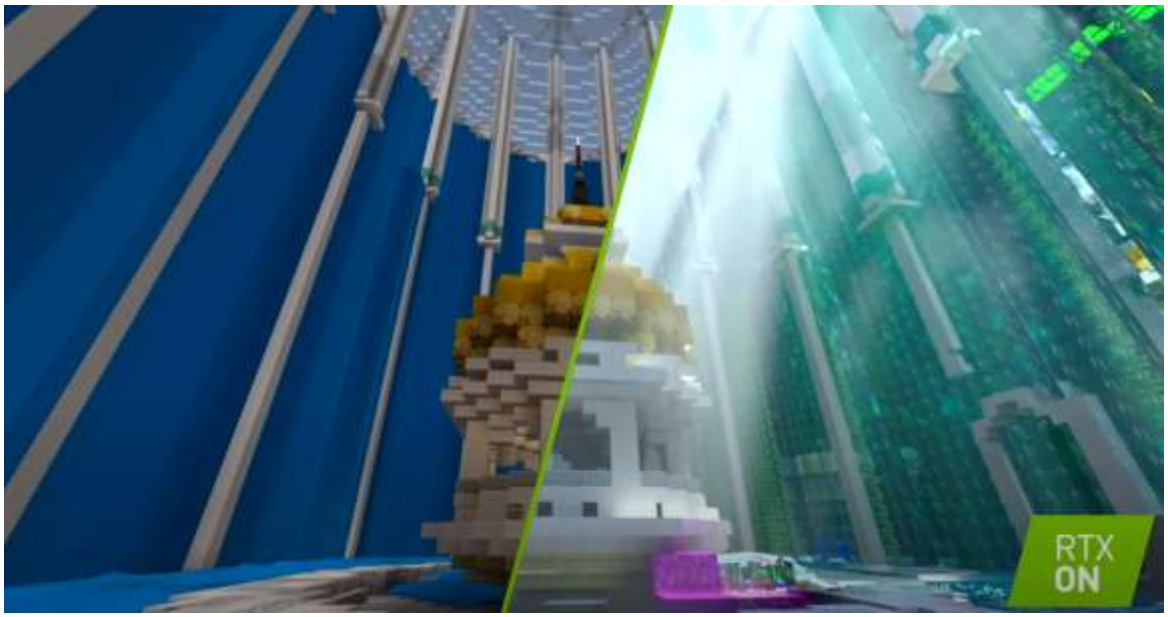

Рисунок4 – Порівняння зображення З та БЕЗ технології RTX у грі Minecraft

Порівняно з традиційними методами обчислення світла, які часто базуються на використанні карт тіней та приховуванні деталей, RTX дозволяє

використовувати фізично точне променеве відстеження. Це призводить до покращення реалізму освітлення, відображення відблисків та глобального освітлення, що робить візуальний досвід у іграх більш іммерсивним та природним.

В результаті впровадження технології RTX у галузі графічних обчислень ігрової індустрії відбулося значне покращення якості освітлення в іграх. Гравці можуть насолоджуватися більш реалістичними та деталізованими світами, де світло взаємодіє з навколишньою обстановкою набагато більш натуральним способом, ніж раніше.

Основна ідея полягає в тому, щоб відтворити, як світло взаємодіє з об'єктами та поверхнями, враховуючи різні властивості, такі як відбиття, преломлення та тіні. Рейтрейсінг є одним із найпотужніших методів візуалізації, здатним надавати зображення високого рівня фотореалізму.

Процес рейтрейсінгу розпочинається зі стартових променів, які випускаються з точки спостереження (камери) і прокладають свій шлях через кожен піксель екрану. Кожен промінь взаємодіє з об'єктами в сцені, ініціюючи ряд подій, які визначають його колір та яскравість.

Один з основних етапів рейтрейсінгу – це визначення того, як кожен промінь взаємодіє з об'єктами сцени. Для цього використовуються алгоритми визначення перетину, які встановлюють точку на поверхні об'єкта, з якою стикається промінь. Це включає визначення властивостей поверхні, таких як колір, текстури, матеріали та нормалі.

Промені, після визначення перетину, можуть взаємодіяти зі світлом, і їх колір визначається врахуванням різноманітних фізичних явищ. Врахування відбиття та преломлення дозволяє сценам виглядати більш реалістично, оскільки промені можуть відбиватися від зеркальних поверхонь чи змінювати напрямок при проходженні через прозорі матеріали.

Додатковий аспект рейтрейсінгу – це врахування тіней. Промені, які досягають світлових джерел, мають враховувати ті тіні, які проекціюють об'єкти у шляху світла. Це важливий елемент для створення реалістичних та емоційно насичених зображень.

Однією з ключових переваг рейтрейсінгу є його здатність до симуляції важливих ефектів, таких як м'яке освітлення, глобальна ілюмінація та реалістична моделювання матеріалів. Однак цей метод також є вибірково витратним з точки зору обчислювальних ресурсів через необхідність великої кількості променів для досягнення високої якості зображення.

Завдяки технологічному прогресу та використанню апаратних прискорювачів графіки, рейтрейсінг стає більш доступним для реального часу у відомих відеоіграх та застосунках візуалізації, вдосконалюючи реалістичність та іммерсивність віртуальних світів [5].

1.4 Рендеринг на основі фізичних властивостей (PBR)

Це методологія візуалізації у галузі комп'ютерної графіки, яка моделює світлові властивості матеріалів у найреалістичніший спосіб, враховуючи фізичні принципи та взаємодії світла з поверхнями. Цей підхід призначений для створення віртуальних об'єктів, що максимально відтворюють реальні матеріали, з урахуванням їхніх оптичних характеристик та взаємодій із світлом.

Основна мета PBR полягає в тому, щоб забезпечити візуально відчутний рівень реалізму в графіці та візуалізації, надаючи художникам та розробникам інструменти для створення віртуальних об'єктів, які реагують на світло так само, як і їхні реальні аналоги. Заснований на принципах фізики, PBR спрощує процес моделювання матеріалів, забезпечуючи єдиною системою управління параметрами, такими як металічність, грубість та інші, для досягнення бажаних властивостей.

Підхід PBR використовується в широкому спектрі сучасних візуальних додатків, зокрема в ігровій індустрії, віртуальній реальності та комп'ютерних програмах для моделювання та дизайну. У відеоіграх PBR вирізняється здатністю передавати реалістичність матеріалів, використовуючи фізично засновані текстури та параметри. Наприклад, металеві поверхні відтворюють відображення світла та

тіні з вираженою металічністю, тоді як матеріали з дерева чи тканини показують більше розсіяного світла та менше блиску.

Крім ігор, PBR широко використовується у віртуальному моделюванні та рендерингу для архітектурного проектування, фільмів та анімації. У таких сферах важливо досягти максимальної реалістичності для візуальної інтерпретації проектів або створення вражаючих сцен у фільмах.

Важливим аспектом PBR є його розширена сумісність та стандартизація, яка сприяє взаємодії різних програм та двигунів, а також полегшує роботу художників та розробників, дозволяючи їм консистентно втілювати свої ідеї та творіння в цифровому середовищі. За допомогою методології PBR графічні застосунки отримують можливість створювати неймовірно реалістичні та живописні зображення, які вражають своєю природністю та вірогідністю віртуального світу.

В ідеї цього алгоритма полягає ідея, що для формування остаточного матеріалу, використовуються текстури, які описують різні характеристики та властвості.

Рисунок5 – Набір текстур, що описують матеріал

На Рисунку 5 описані(зліва направо) такі текстурні карти: основний колір, нормалі, металевість, шорсткість, висоти, зовнішня оклюзія. Кожна із них описує певну особливість матеріалу:

- Основний колір (Base Color) - ця карта містить інформацію про колір матеріалу, вона також відома як карта альбедо(Albedo) або дифузна карта(Diffuse map). Термін "дифузна" використовувався до впровадження PBR і зазвичай означав, що тіні були впечені в саму карту. Сьогодні це термін для позначення колірної карти і специфікації плюс робочий процес, які виконують одну і ту ж функцію
- Нормалі (Normal Map) - виконують майже ту саму функцію, що й карта нормалей дотичного простору. містить інформацію про те, як відбиватиметься світло, виходячи з напрямку нормалей поверхні. Кожен піксель normal map кодує напрямок поверхні у формі кольорового вектора. При використанні цієї normal map в графічних програмах, система враховує ці вектори для правильного розрахунку освітлення, тіней та відбиття на поверхні об'єкта. Зазвичай normal map формується на основі карт висот (height maps) або інших джерел глибини.
- Металевість (Metallic Map) - Існує причина, чому він називається металевим, і це тому, що в реальному світі кожен матеріал можна звести до простого питання: є він металом чи ні. Якщо так, то значення на карті буде білим, якщо ні - чорним. Зазвичай ця мапа завжди буде чорно білою, за дуже рідкими виключеннями.
- Шорсткість (Roughness Map) - Карта шорсткості визначає, наскільки шорстким є матеріал. Світліше значення робить його більш шорстким, а темніше - більш блискучим. Саме тут проявляється індивідуальність матеріалу, наприклад, відбитки пальців на дзеркалі або пляма від кави на столі
- Висоти (Height / Displacement Map) - Карта висоти або зміщення - це карта у відтінках сірого, яка обробляє висоту або глибину матеріалу,

кожен піксель представляє висоту точки на поверхні об'єкта. Чим світліше піксель, тим більше висота. Як було сказано раніше, вона зазвичай використовується разом із звичайною картою і додає глибину геометрії.

Зовнішня оклюзія (Ambient Occlusion Map) - використовується для імітації розсіяного освітлення, зазвичай це просто біла карта з темнішим значенням, що представляє навколишні тіні. Це ефект, що виникає в результаті меншого проникнення амбієнтного світла в областях, де поверхні близько одна до одної. Ці області схильні до того, щоб затемнюватися, формуючи тіні та додаючи глибину об'єктам.

1.5 Скелетна анімація (Skeletal Animation)

Це техніка анімації в комп'ютерній графіці де рух та деформація 3D-моделі керуються ієрархічною структурою кісток, що утворює скелет. Кожна кістка має свої координати та впливає на частину об'єкта, дозволяючи створювати природні та гнучкі анімаційні рухи для персонажів, тварин чи об'єктів у віртуальних середовищах, таких як відеоігри та анімаційні фільми. Ця техніка дозволяє ефективно моделювати рухи, спрощуючи процес анімації та забезпечуючи більшу гнучкість у представленні реалістичних персонажів чи об'єктів [17].

Завдяки скелетній анімації, кожна кістка скелета може мати свої власні ключові точки анімації, що дозволяє точно контролювати рух та деформацію об'єкта. У процесі відтворення анімації система інтерполює позиції кісток між ключовими кадрами, забезпечуючи плавні та природні переходи між рухами. Це робить скелетну анімацію надзвичайно корисною для реалістичного відтворення рухів персонажів у віртуальних середовищах, а також для оптимізації процесу роботи над анімацією, оскільки зміни вносяться лише до кісток скелета, а не до кожного окремого вершка моделі. Таким чином, скелетна анімація стала ключовим елементом у сучасній комп'ютерній графіці та анімації, дозволяючи творцям створювати вражаючі та реалістичні віртуальні світи.

Важливим аспектом скелетної анімації є можливість використання ваг кісток, які визначають вплив кожної кістки на окремі вершки моделі. Це дозволяє створювати ефективну деформацію, таку як згинання чи обертання, забезпечуючи натуральність анімації. В сучасних іграх та анімаційних проектах скелетна анімація часто поєднується з різними техніками, такими як інверсійна кінематика (ІК), щоб реалістично відтворювати рухи та взаємодію персонажів з навколишнім середовищем. У результаті скелетна анімація визначає основу для створення живих та віртуальних світів, де персонажі набувають реалістичності та вираження через натуральні рухи та деформації. Вона є ключовим інструментом для розширення можливостей комп'ютерної графіки та анімації, роблячи віртуальний світ більш живим та захоплюючим для глядачів та гравців.

Переваги використання скелетної анімації:

- Реалістичність рухів – Скелетна анімація дозволяє створювати природні, реалістичні рухи персонажів та об'єктів, що робить візуальний досвід більш іммерсивним.
- Ефективність обчислень – Використання скелетів дозволяє оптимізувати обчислення анімації, оскільки рухаються лише кістки, а не кожен окремий вершок моделі.
- Гнучкість та Інтерактивність – Скелетна анімація легко адаптується до різних сценаріїв та взаємодіє з іншими анімаційними техніками, такими як інверсійна кінематика (ІК).
- Масштабованість – Технологія легко масштабується для роботи з різноманітними типами об'єктів, від персонажів до тварин чи механізмів.

Недоліки використання скелетної анімації:

- Неідеальна Деформація – У деяких випадках можуть виникати артефакти анімації, особливо при значних деформаціях, що може призвести до нереалістичного вигляду моделі.

- Складність Риггінгу – Процес створення рухомого скелета (ріггінг) може бути складним і вимагати досвіду та спеціалізованих інструментів.
- Велика Кількість Даних – Велика кількість кісток в скелеті та ключових точок анімації може призвести до значного обсягу даних, що впливає на продуктивність та зберігання.
- Обмежена Гнучкість – У деяких випадках скелетна анімація може обмежувати гнучкість в представленні екстремальних рухів або незвичайних форм об'єктів.
- Використання великої кількість ресурсів процесора – При невеликій кількості кісток або ж самих мешів на сцені, цей недолік буде непомітним. У випадку ж ААА ігр, де бувають сцени з великою кількість ботів, скелетні меші котрих складаються с сотень або ж тисяч компонентів, навість у найпродуктивніших обчислювальних систем може значно знизитись продуктивність. Особливо це помітно при грі на мобільних платформах, таких як мобільний телефон, Nintendo Switch, PSP.

Оскільки скелетні анімації можуть бути дуже вибагливими до апаратного забезпечення, то основним методом оптимізації є зменшення кількості рухомих елементів, наприклад волосся або елементи одягу. Спрощення таких елементів не буде впливати ігровий процес користувача, але зможе забезпечити комфортну та плавну гру. Також використовуються спрощення фізик окремих елементів. Наприклад, ігнорування вітру, рухів дрібними елементами одягу, або ж їх групування, внаслідок чого не потрібно буде окремо прораховувати вплив фізики на кожен елемент.

На базовому рівні анімації працюють за концепцією інтерполяції. Просте рівняння інтерполяції, що використовується для трансляції та масштабування, має такий вигляд **a = a * (1 - t) + b * t**. Воно відоме як рівняння лінійної інтерполяції або Lerp. Для обертання ми не можемо використовувати вектор. Причина в тому, що якщо ми спробуємо використати рівняння лінійної інтерполяції на векторі X(Pitch), Y(Yaw) і Z(Roll), інтерполяція не буде лінійною. Щоб уникнути цієї

проблеми, використовується Quaternion для обертання. Quaternion надає так звану Сферичну інтерполяцію або рівняння Slerp, яке дає той самий результат, що і Lerp, але для двох обертань A та B.

Компонентами анімованої моделі є шкіра, кістки та ключові кадри. Весь процес анімації починається з додавання першого компонента - "Скіну" (Skin) у таких програмах, як Blender або Maya. Шкіра - це не що інше, як сітка, яка додає візуальний аспект моделі, щоб показати глядачеві, як вона виглядає. Але якщо потрібно перемістити будь-яку сітку, то так само, як і в реальному світі, необхідно додати Кістки (Bones). Як це виглядає в програмному забезпеченні на кшталт Blender'a показано на Рисунку 6.

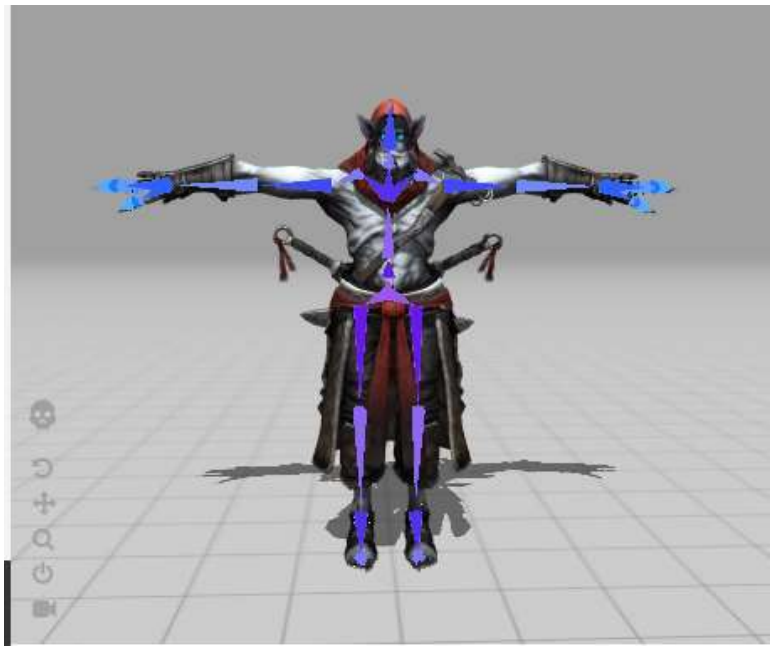

Рисунок 6 – Вигляд кісток та скіну у редакторі

Ці кістки зазвичай додаються в ієрархічному порядку для таких персонажів, як люди і тварини, і причина досить очевидна. Необхідно, щоб між кінцівками був зв'язок, як між батьками та дітьми. Наприклад, якщо рухається праве плече, то і правий біцепс, передпліччя, кість і пальці також повинні рухатися. Вигляд цієї ієрархії вказаний на Рисунку 7.

На Рисунку 7, якщо схопити стегнову кістку і рухати нею, но рух вплине і на всі її кінцівки.

Наступним етапом є створення ключових кадрів для анімації. Ключові кадри - це пози в різні моменти часу в анімації. Їх потрібно інтерполювати між цими ключовими кадрами, щоб плавно переходити від однієї пози до іншої. Для подальшого спрощення імплементації з метою дослідження роботи скелету анімації буде використовуватися бібліотека для завантаження різних форматів 3d-файлів у спільний формат, що зберігається в пам'яті - Open Asset Import Library (assimp) [6]. На Рисунку 8 вказана блок схема, яка описує механізм зберігання підгружених анімацій за допомогою бібліотеки assimp.

Вказівник aiScene, який містить вказівник на кореневий вузол, і масив Animations. Цей масив aiAnimation містить загальну інформацію, таку як тривалість анімації, представлену тут як mDuration, а також змінну mTicksPerSecond, яка контролює, як швидко ми повинні інтерполювати між кадрами. aiAnimation містить масив aiNodeAnim, який називається Channels. Цей масив містить усі кістки та їхні ключові кадри, які будуть задіяні в анімації. AiNodeAnim містить назву кістки, а також 3 типи клавіш для інтерполяції між ними: Переклад, Обертання та Масштаб.

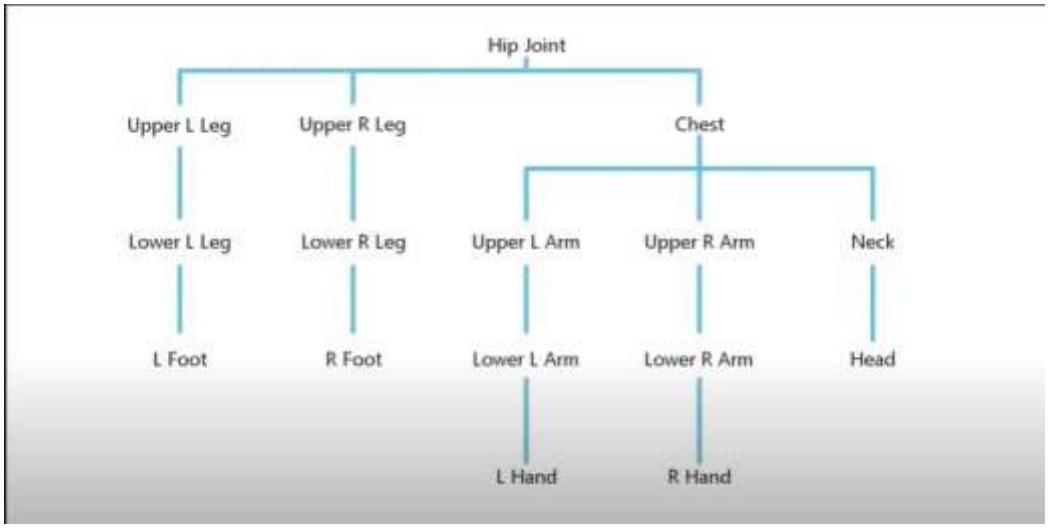

Рисунок 7 – Ієрархія елементів скелету у вигляді дерева

Вплив декількох кісток на вершини. Коли згинається передпліччя, видно, як піднімається біцепс. Можна також сказати, що трансформація кісток передпліччя впливає на вершини біцепса. Аналогічно, на одну вершину в сітці може впливати декілька кісток. Для таких персонажів, як суцільнометалеві роботи, на всі вершини передпліччя впливатиме лише кістка передпліччя, але для таких персонажів, як люди, тварини тощо, на вершину може впливати до 4 кісток. На Рисунку 9 зображено схему, як assimp зберігає цю інформацію.

Вказівник aiScene, який містить масив усіх aiMeshes. Кожен об'єкт aiMesh має масив aiBone, який містить інформацію про те, який вплив цей об'єкт матиме на набір вершин у сітці. aiBone містить назву кістки, масив aiVertexWeight, який, по суті, говорить нам про те, який вплив цей об'єкт матиме на які вершини у сітці. Додаткова змінна aiBone – offsetMatrix - це матриця 4x4, яка використовується для перетворення вершин з простору моделі в простір кістки. Коли вершини знаходяться в просторі кістки, вони будуть трансформовані відносно своєї кістки, як і повинно бути.

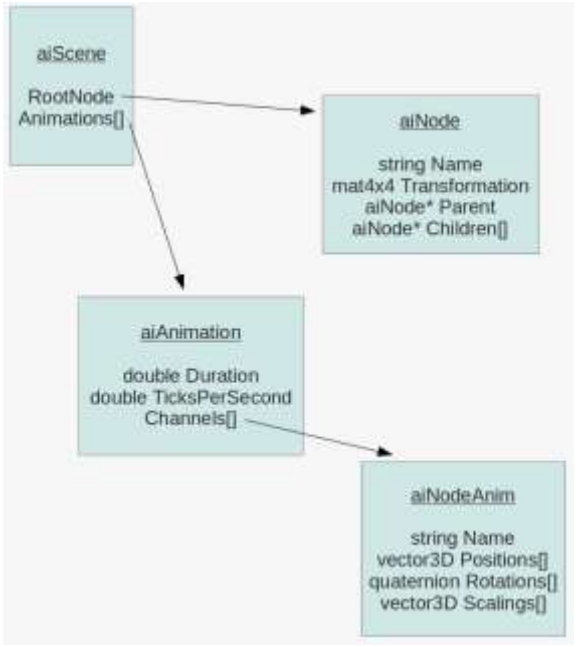

Рисунок 8 – Механізм зберігання підгружених анімацій

Для кожного кадру рендерингу необхідно плавно інтерполювати всі кістки в ієрархії та отримати їхні фінальні матриці перетворень, які будуть передані до шейдерної уніформи finalBonesMatrices. Ось що робить кожен клас:

- Bone – Окрема кістка, яка зчитує всі дані ключових кадрів з aiNodeAnim. Вона також інтерполює між своїми ключами, тобто Переклад, Масштаб та Обертання, на основі поточного часу анімації.
- AssimpNodeData – Ця структура допоможе нам ізолювати нашу анімацію від Assimp.
- Animation – Актив, який зчитує дані з aiAnimation і створює спадкоємний запис Bones
- Animator – Він зчитує ієрархію AssimpNodeData, рекурсивно інтерполює всі кістки, а потім готує для нас остаточні матриці трансформації кісток, які нам потрібні.

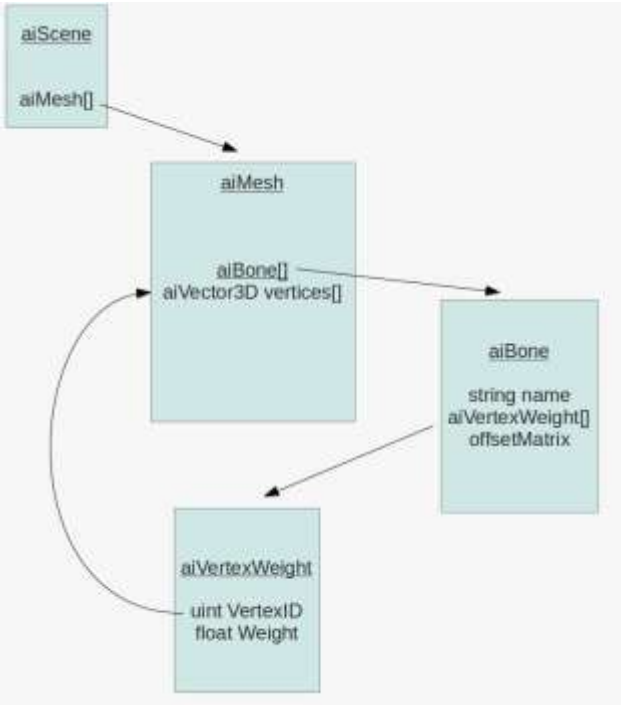

Рисунок 9 – Схема зберігання інформації бібліотекою assimp

Створення об'єкта анімації починається з конструктора. Він приймає два аргументи. Перший - шлях до файлу анімації, а другий параметр - модель цієї

анімації. Потім створюється імпортер Assimp::Importer для читання файлу анімації, після чого виконується перевірка тверджень, яка видасть помилку, якщо анімацію не вдасться знайти. Потім зчитуються загальні дані анімації, такі як тривалість анімації (mDuration) та швидкість анімації (mTicksPerSecond). Потім викликається ReadHeirarchyData, яка копіює спадкоємність aiNode з Assimp і створює спадкоємність AssimpNodeData.

Потім викликається функція ReadMissingBones. Ця функція зчитує інформацію про відсутні кістки і зберігає її в m_BoneInfoMap моделі, а також зберігає посилання на m_BoneInfoMap локально в m_BoneInfoMap.

Конструктор Animator отримує анімацію для відтворення, після чого скидає час анімації m_CurrentTime до 0. Він також ініціалізує m_FinalBoneMatrices, який є std::vector<glm::mat4>. Основна увага тут приділяється функції UpdateAnimation(float deltaTime). Вона просуває m_CurrentTime зі швидкістю m_TicksPerSecond, а потім викликає функцію CalculateBoneTransform. На початку ми передаємо два аргументи, перший - це m_RootNode з m_CurrentAnimation, а другий - матриця ідентичності, передана як parentTransform Ця функція перевіряє, чи задіяна кістка m_RootNode у цій анімації, знаходячи її у масиві m_Bones з Animation. Якщо кістка знайдена, вона викликає функцію Bone.Update(), яка інтерполює всі кістки і повертає локальну матрицю перетворення кістки до nodeTransform. Але це локальна просторова матриця, і якщо її передати у шейдерах, то вона змістить кістку навколо початку координат. Тому ми помножимо nodeTransform на parentTransform і збережемо результат у globalTransformation. Цього було б достатньо, але вершини все ще знаходяться у просторі моделі за замовчуванням. Для цього вираховується матриця зміщення у m_BoneInfoMap, а потім вона множиться на globalTransfromMatrix. Буде отримано ідентифікаційний індекс, який буде використано для запису остаточного перетворення цієї кістки в m_FinalBoneMatrices.

Потім викликається CalculateBoneTransform для кожного дочірнього вузла цього вузла і передається globalTransform як parentTransform. Цей рекурсивний цикл розривається, коли не залишиться дочірніх вузлів для подальшої обробки.

Завантаження починається з Моделі, яка встановить дані про вагу кісток для шейдера, а потім створюємо Анімацію, вказуючи їй шлях. Потім ми створюємо об'єкт Аніматора, передаючи йому створену анімацію. У циклі рендеру оновлюється аніматор, отримуємо остаточні перетворення кісток і передаємо їх шейдерам.

Висновки до розділу 1

У цьому розділі були розглянуті найбільш використовувані алгоритми різних способів візуалізації в ігровому рушії. В даних випадках використовувався графічний інтерфейс OpenGL. Для швидкодійних програм він не є найоптимальнішим вибором, оскільки через відносно високий рівень абстракції не дає розробнику повну свободу дій, але в той час і значно спрощує процес розробки. Оскільки ціллю було на практиці дослідити принципи розробки та покращення різноманітних способів рендерингу.

Спираючись на отриманні дані, можна зробити висновок що вибір тих чи інших технологій не є універсальним для всіх випадків і для кожної задачі необхідно обирати конкретний метод і підхід. Наприклад, якщо поставлена задача – це створення максимально реалістичної мультиплікації, то є сенс використовувати найбільш деталізовану скелетну модель та текстури з найбільшим розширенням для створення матеріалу з алгоритмом PBR.

Якщо ціллю є високопродуктивний продукт який використовується в режимі реального часу, то можливо дещо зменшити якість трасерованих лучів, щоб перенести вирахування симуляції фізики з центрального процесора на графічний. Також неможливо виділити якийсь один з способів, оскільки вони всі є взаємопов’язані, що і дає в результаті високоякісну картинку. Найбільш деталізовані скелети не зможуть дати гарну картинку, якщо, наприклад, алгоритм PBR буде працювати некорректно, то шкіра моделі буде неправильно передавати колір, відблискування, тощо.

Без якісно обрахованого глобального освітлення, технології рейтресингу не зможуть дати очікуваного результату оскільки вхідні дані будуть невалідні. Таким же чином неправильно працюючий алгоритм кулінгу, наприклад, не буде належним чином видаляти з обчислень невидимі об'єкти, які будуть займати і так дуже обмежені обчислювальні ресурси для інших алгоритмів. Або ж, навпаки, зайві видалені об'єкти будуть створювати ефект ‘битої карти’.

Такі комплексні об'єкти, як Skeletal Mesh потребують комбінації багатьох різних алгоритмів: відображення текстур, в тому числі і алгоритмом PBR, правильного обраховування освітлення та тіней, симуляція фізичних явищ тощо. Саме тому для успішного створення високоякісного зображення необхідно знати та використовувати різноманітні способи візуалізації об’єктів.

РОЗДІЛ 2
ІСНУЮЧІ СПОСОБИ СПРОЩЕННЯ СКЛАДНИХ МЕШІВ

2.1 Актуальність алгоритмів спрощення

Актуальність алгоритмів спрощення складних мешів у галузі комп'ютерної графіки та ігрової розробки визначається кількома ключовими факторами, які враховують потреби сучасних програм та проєктів:

Оптимізація Продуктивності - Зменшення кількості полігонів в 3D-моделях дозволяє прискорити обчислення та покращити продуктивність графічних застосунків, особливо на пристроях з обмеженими ресурсами, таких як мобільні пристрої та ігрові консолі.

- Оптимізація Пам'яті - Зменшення об'єму геометричних даних сприяє ефективнішому використанню оперативної пам'яті, що особливо важливо на пристроях з обмеженими ресурсами.
- Підтримка Великих Обсягів Даних - Сучасні ігрові світи та архітектури вимагають обробки великих обсягів геометричних даних. Алгоритми спрощення можуть покращити ефективність обробки та відображення таких об'ємів інформації.
- Оптимізація Мережевого Трафіку - У випадку онлайн-ігор та ігор з онлайн-режимом, зменшення обсягу геометричних даних допомагає зменшити обсяг передачі даних та поліпшує якість мережевого взаємодії.
- Підтримка Різних Платформ - Відмінності в характеристиках різних платформ (PC, консолі, мобільні пристрої) вимагають оптимізації графічного вмісту для максимального використання ресурсів кожної платформи.
- Розробка Віртуальної та Доповненої Реальності - У віртуальній та доповненій реальності, де обчислювальні та графічні ресурси обмежені, важливо використовувати ефективні алгоритми для спрощення геометрії та забезпечення зручної візуалізації.

- Збереження Анімацій - Під час анімації можливі значні зміни в полігонізації моделі. Алгоритми спрощення можуть допомогти зберегти анімації при зменшенні кількості полігонів.

Актуальність алгоритмів спрощення мешів визначається широким спектром вищезгаданих факторів, а також постійним розвитком технологій та зростанням вимог до графічних додатків. Забезпечення ефективності та візуальної якості водночас є ключовим викликом для індустрії геймдеву та графічного дизайну.

2.2 Загальний опис використовуваних методів спрощення

Спрощення складних мешів — це важливий етап у графічному дизайні та розробці ігор, який дозволяє зменшити кількість полігонів у 3D-моделях, зменшуючи при цьому вплив на їхню візуальну якість. Існує кілька способів спрощення мешів:

- Усереднення Вершин (Vertex Averaging) - Цей метод полягає у заміні групи сусідніх вершин однією вершиною, яка є середньою точкою для цієї групи. Це може бути ефективним для зниження кількості вершин, але враховуйте, що це також може змінити форму об'єкта.
- Спрощення Полігонів (Polygon Simplification) - Використовується алгоритм, який об'єднує суміжні трикутники у більші полігони. Найбільш відомий метод - алгоритм Quadric Error Metrics (QEM), який вибирає оптимальні трикутники для злиття, зберігаючи при цьому якість поверхні.
- Зниження деталізації (Detail Reduction) - Зменшення деталізації включає в себе видалення деяких деталей, які можуть бути менш важливими. Наприклад, менш деталізовані текстури чи відсутність невеликих виступаючих елементів.
- Видалення Непотрібних Геометричних Деталей - Видалення непотрібних геометричних деталей, таких як малих виступаючих

частинок, може значно зменшити кількість полігонів без помітного впливу на візуальний вигляд.

- Рівні деталізації (Level of Detail) - Застосування різних рівнів деталізації для об'єктів в залежності від їхнього положення та віддаленості від спостерігача. Це дозволяє використовувати менше деталей для об'єктів, які знаходяться далеко.
- Згладжування Кривих (Curve Simplification): - Використовується для спрощення кривих та поверхонь, зменшуючи кількість точок, що визначають їх форму.
- Приведення Полігонів до Примітивних Форм - Заміна складних геометричних форм (наприклад, круги чи еліпси) більш простими примітивними формами.

Кожен із цих методів має свої переваги та обмеження, і їх вибір залежить від конкретних потреб проекту. Оптимальний підхід може включати комбінацію різних методів для досягнення балансу між зменшенням обчислювальних витрат та збереженням візуальної якості моделі. Варто зазначити, що більшість із зазначених способів автоматизовані і застосовуються без втручання розробника або ж дизайнера. Наприклад, згладжування кривих або ж усереднення вершин можуть застосуватися під час запікання(кукінгу) - це процес попередньої генерації та зберігання різноманітних данних або текстур, які будуть використані і реальному часі під час рендерингу сцени. Важливо відзначити, що процес кукінгу може в себе включати різну кількість етапів, в залежності від потреб проєкту. Він може включати в себе, наприклад, освітлення, тіні, матеріали. Процес запікання дозволяє великою мірою зменшити обчислювальне навантаження під час рендерингу в реальному часі, оскільки попередньо згенеровані дані можуть бути використані замість складних обчислень під час відтворення сцени. Це особливо корисно в ігровій індустрії, де важлива висока продуктивність та якість візуалізації.

Процес кукінгу включає в себе 5 основних етапів:

- Вибір Типу Даних для Запікання - Визначення, які саме дані або параметри необхідно попередньо згенерувати. Це може бути освітлення, тіні, картки нормалей, картки висот або інші текстури. Зазвичай, це налаштовується попередньо розробник відповідно до вимог програми.
- Розміщення Умовних Маркерів (UV Unwrapping) - Розгортання поверхні об'єкта та розміщення умовних маркерів (UV-координат) для подальшого прив'язування попередньо згенерованих даних.
- Попереднє Генерування Даних (Cooking) - Власне запікання, тобто генерація текстур чи інших даних за умови розміщених маркерів. Зазвичай це включає в себе використання різних алгоритмів для розрахунку освітлення, тіней, кольору, висоти, амбієнтної оклюзії тощо.
- Експорт та Збереження Запіканих Даних - Збереження результатів у спеціальний формат для подальшого використання у графічному двигуні чи іншому програмному забезпеченні. Формати можуть включати зображення (такі як текстури) чи спеціалізовані файли (як для карт нормалей).
- Використання Запіканих Даних у Реальному Часі - В графічному двигуні чи ігровому середовищі запікані дані можуть бути використані для досягнення реалістичності та оптимізації рендерингу під час відтворення сцени в реальному часі.

2.3 Зменшення кількості полігонів використовуючи LOD

Алгоритми рівнів деталізації (LOD) та рівнів міп-карт (MIP level) є ключовими концепціями у графічному програмуванні, спрямованими на оптимізацію відображення зображень та текстур.

LOD (рівні деталізації) використовується для регулювання рівня деталізації геометрії чи текстур на основі відстані до об'єкта. Це дозволяє знижувати

обчислювальне навантаження та підвищувати продуктивність за рахунок використання менш деталізованих моделей або текстур для об'єктів, що знаходяться далеко від камери.

Рівні MIP (MIP level) використовуються для оптимізації відображення текстур на різних розмірах та відстанях. Замість використання одного розміру текстури для всіх умов, генеруються попередньо обчислені версії текстури з різними розмірами. Під час відображення вибирається та використовується та версія текстури, яка найкраще відповідає розміру об'єкта на екрані та відстані до камери. Наприклад, в ігровому рушії Unreal Engine є спеціальні файли конфігурації в яких можливо створити додаткові текстурні групи і там вказати мінімальний та максимальний розмір текстури для відповідних лодів.

Обидва ці підходи є ефективними методами оптимізації графічного відображення, дозволяючи знижувати навантаження на графічний процесор та покращувати продуктивність, при цьому забезпечуючи адекватну якість візуалізації.

Система рівнів деталізації давно успішно використовується як метод оптимізації графіки. Суть методу полягає в тому, що зі збільшенням відстані графічний рушій, замість оригінальної високодеталізованої моделі, буде завантажувати спрощену модель, зменшуючи навантаження на обчислювальну систему, проте користувач не повинен бачити зовнішньої різниці між ними. Однак у цього підходу є суттєвий недолік. Оскільки тепер до однієї оригінальної моделі додається одна або кілька спрощених моделей, збільшується вимоги до обсягу та швидкості накопичувача, обсягу оперативної пам'яті та пам'яті відеокарти. Сьогодні в умовах вибухового росту характеристик споживчої якості персональних комп'ютерів стала очевидною необхідність проведення досліджень, пов'язаних із вибором оптимальної кількості рівнів деталізації об'єктів на сцені для досягнення потрібної продуктивності та плавності переходів між рівнями деталізації. Ідею спрощення та відмінність рівнів деталізації можна побачити на Рисунку 10 та Рисунку 11.

Один з підходів до генерації LOD полягає в автоматичній редукції деталей моделі в залежності від відстані до спостерігача. Такий метод дозволяє автоматично оптимізувати модель, зменшуючи кількість полігонів та деталей, коли об'єкт знаходиться далеко від спостерігача. Однак цей підхід може призводити до втрати деталей на великих відстанях, що впливає на реалізм відображення.

Інший підхід включає в себе ручне або напівавтоматичне створення різних рівнів деталізації моделі. Це дозволяє дизайнерам більш точно контролювати якість та вигляд об'єкта на різних відстанях. Проте це вимагає більше трудовитрат і ручного втручання.

Третій підхід використовує комбінацію автоматичної та ручної генерації LOD. Це дозволяє поєднати автоматизацію з можливістю детального налаштування за потреби.

Основні критерії оцінки ефективності таких методів включають плавність переходів між рівнями деталізації, відсутність артефактів та оптимальне використання ресурсів обчислювальної системи.

Image					
Vertices	~5500	~2880	~1580	~670	140

Рисунок 10 – Приклад ймовірних рівнів деталізації

Багато застосувань у комп'ютерній графіці вимагають складних, високодеталізованих моделей. Однак рівень деталізації, який насправді необхідний, може суттєво відрізнятися. Щоб контролювати час обробки, часто бажано використовувати апроксимації замість надмірно деталізованих моделей.
Розроблений алгоритм може швидко створювати високоякісні апроксимації полігональних моделей. Алгоритм використовує ітеративні скорочення пар вершин

для спрощення моделей і підтримує наближення з поверхневою похибкою за допомогою квадратичних матриць. матрицями.

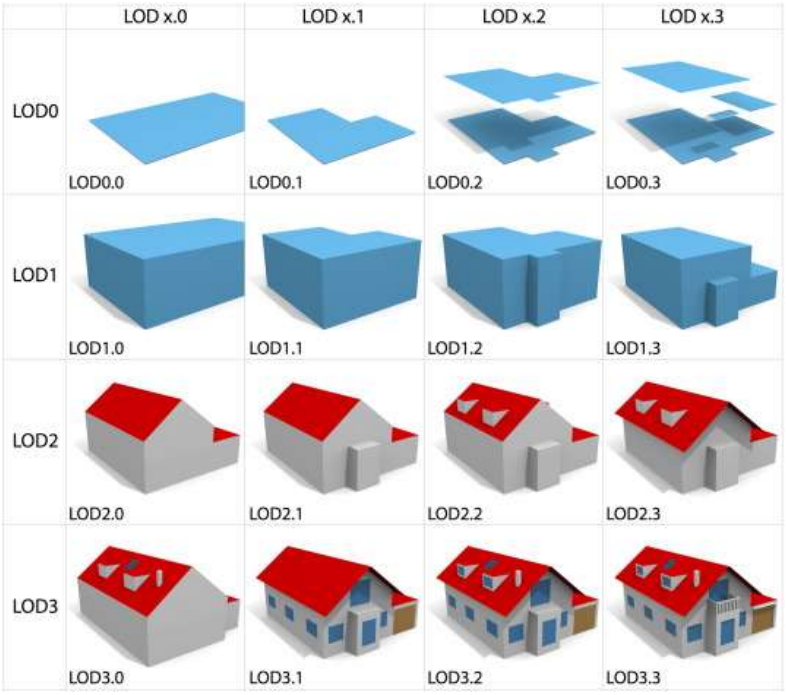

Рисунок 11 – Принцип роботи рівні деталізації

Висновки до розділу 2

В другому розділі були детально розглянуті алгоритми, що дозволяють зменшити кількість полігонів на об’єкті. Були проаналізовані і різні способи, що вбудовані в ігровий рушій або програмне забезпечення, що використовується для створення зображень.

Були досліджені різні аспекти оптимізації геометричних об'єктів у графічному програмуванні. Актуальність алгоритмів спрощення визначається постійним зростанням складності сучасних 3D-сцен у відеоіграх та комп'ютерній графіці. З огляду на велику кількість полігонів у складних мешах, актуальність оптимізації стає важливою для забезпечення плавності відтворення та підвищення продуктивності.

У загальному описі існуючих способів спрощення були розглянуті ключові аспекти та підходи до оптимізації геометричних моделей. Методи зменшення кількості полігонів за допомогою технології LOD дозволяють динамічно регулювати рівень деталізації в залежності від відстані до об'єкта, підвищуючи продуктивність у випадках, коли висока деталізація не є необхідною.

Особливий акцент було зроблено на спрощенні поверхні за допомогою квадратичної метрики похибок. Цей підхід дозволяє враховувати точність апроксимації та ефективно зменшує кількість полігонів, забезпечуючи збереження форми та деталей об'єкта.

Слід зазначити, що існуючі способи спрощення мешів виявляються важливими для розробників графічних застосунків, оскільки вони дозволяють досягати балансу між високою якістю візуалізації та оптимальним використанням ресурсів обчислювальної та графічної підсистем. Розглянуті аспекти та способи спрощення відкривають широкі можливості для оптимізації та поліпшення ефективності відображення 3D-сцен.

РОЗДІЛ 3
ВИКОРИСТАННЯ НОРМАЛЕЙ ДЛЯ ПОКРАЩЕННЯ ЯКОСТІ ЗОБРАЖЕННЯ

3.1 Математичний сенс нормалі

Перед тим як описувати поведінку та використання вектору нормалі у комп’ютерній графіці, необхідно детальніше розібрати його сутність та геометричні властивості. Це дозволить більш ґрунтовно розуміти проблематику та можливості у цільовій галузі цієї дисертації– комп’ютерній графіці, способах візуалізації об’єктів.

Норма́ль - це пряма, яка стоїть перпендикулярно до дотичного простору (дотичної прямої до кривої, дотичної площини до поверхні і так далі). Наприклад, у двовимірному просторі лінія нормалі до конкретної точки кривої є лінією, яка перпендикулярна до дотичної лінії в цій точці. У тривимірному просторі нормаль до точки поверхні P представляє собою вектор, який стоїть перпендикулярно до дотичної площини для цієї точки P поверхні.

У векторній алгебрі, вектор нормалі – це важлива концепція, яка визначається як вектор, перпендикулярний до поверхні в точці. Цей вектор вказує на напрямок або орієнтацію поверхні та є важливим елементом у багатьох математичних та фізичних контекстах.

У геометрії, вектор нормалі використовується для визначення перпендикулярного напрямку до поверхні об'єкта. Він грає ключову роль у розрахунках освітлення, тіней та інших візуальних аспектах у графіці.

В інших математичних галузях вектори нормалей також відіграють суттєву роль. Наприклад, у диференціальній геометрії вони використовуються для вивчення кривизни поверхонь. У теорії поля вони можуть представляти напрямок силових ліній. В геофізиці та аеродинаміці вектори нормалей використовуються для аналізу геологічних формацій та визначення аеродинамічних характеристик об'єктів.

Декілька прикладів, де нормалі використовуються в інших наукових галузях:

- Теорія поля - представляють напрямки силових ліній.
- Динаміка рідин - у фізиці рідин вектори нормалей використовуються для моделювання руху рідини та визначення її властивостей на поверхні.
- Механіка твердого тіла - використовуються для розрахунків контактних сил із поверхнею
- Геофізика - для дослідження геологічних формацій та структур.
- Аеродинаміка - для аналізу обтічної поверхні об'єктів та розрахунків аеродинамічних характеристик.

У всіх цих галузях вектори нормалей є потужним інструментом для аналізу та моделювання геометричних та фізичних властивостей об'єктів у просторі.

Для обчислення вектору нормалі у найпростішому випадку для одної вершини трикутника використовується наступний метод. Важливо зазначити, що саме трикутники є основною складовою будь-якого меша у комп’ютерній графіці, тому важливо знати як відбуваються обчислення.

Нехай меш представлений параметрично, де кожна вершина описується функцією ***P(u, v) = (x(u, v), y(u, v), z(u, v))***, де ***u*** та ***v*** – параметри

Використовуючи часткові похідні за ***u*** та ***v***, можна обчислити тангенти до параметрів [12]:

$$\overrightarrow{T_u} = \frac{\partial P}{\partial u} = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right) \text{ та } \overrightarrow{T_v} = \frac{\partial P}{\partial v} = \left(\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right)$$

Обчислюються нормалі як векторний добуток тангентів:

$$\vec{N} = \overrightarrow{T_u} \times \overrightarrow{T_v}$$

Щоб отримати одиничний вектор, вектор нормалі необхідно нормалізувати:

$$\overrightarrow{N_n} = \frac{\vec{N}}{\left|\left|\vec{N}\right|\right|}$$

3.2 Приклади використання нормалей в ігровому рушії

Вектори нормалі в ігровому рушії є важливим елементом для досягнення реалістичного відображення об'єктів та реалізації різних ефектів, таких як освітлення, тіні, колізії та текстури. Вони грають критичну роль у відтворенні фізичних та візуальних властивостей об'єктів у 3D-середовищі.

Окрім прорахунку освітлення, тіней, використання векторів нормалі займає центральне місце у фізичних симуляціях для досягнення реалістичних результатів. Одним із ключових аспектів використання цих векторів є їхня роль у прорахунках колізій та інших фізичних взаємодій об'єктів у 3D-середовищі.

Обчислення колізій в ігровому рушії — це складний процес, який вимагає ретельного розрахунку взаємодії об'єктів та їхніх поверхонь. Основний принцип полягає в тому, щоб визначити, чи та як об'єкти перетинаються або стикаються один з одним. Важливим елементом цього процесу є використання векторів нормалі та інших параметрів для правильного розрахунку реакції на колізію. Цей процес відбувається у декілька етапів:

- Визначення Границь Об'єктів - Кожен об'єкт в ігровому світі має свою геометрію, яка представляється у вигляді вершин, граней та ребер. Спочатку визначаються границі кожного об'єкта.
- Перевірка Перетинання Границь - За допомогою математичних алгоритмів, таких як AABB (Axis-Aligned Bounding Box) чи OBB (Oriented Bounding Box), визначається, чи перетинаються границі двох об'єктів. Це ефективні методи для швидкого визначення можливості колізії перед більш детальними перевірками.
- Обчислення Точок Колізії - Якщо границі перетинаються, визначаються точки колізії — місця на поверхні об'єктів, де вони стикаються або перетинаються.
- Розрахунок Векторів Нормалі - Для кожної точки колізії обчислюються вектори нормалі для поверхні об'єктів у цих точках.

Вектор нормалі вказує напрямок перпендикулярної поверхні в точці колізії.

- Визначення Глибини Колізії - Розраховується глибина колізії, що вказує, наскільки глибоко об'єкти перетинаються. Це важливий параметр для правильного розрахунку відштовхування.
- Визначення Реакції на Колізію - На основі векторів нормалі та глибини колізії визначається, як об'єкти повинні взаємодіяти. Це може включати відштовхування, зміну швидкості, анімаційні ефекти чи інші реакції, щоб симулювати реалістичну поведінку.
- Виконання Корекції - У деяких випадках виконується корекція позицій об'єктів, щоб уникнути перетинання та гарантувати правильну взаємодію.

Фізичні симуляції у галузі комп'ютерної графіки та ігрової розробки охоплюють широкий спектр ефектів та взаємодій, таких як сили тяжіння, деформації, руйнування, колізії, та інші. Вектори нормалі грають ключову роль у фізичних симуляціях, забезпечуючи правильне моделювання поведінки об'єктів під впливом різноманітних сил.

Основні аспекти використання векторів нормалі в фізичних симуляціях:

- Сили - Вектори нормалі допомагають визначити, які сили та торки діють на об'єкти в різних точках їхніх поверхонь. Наприклад, у симуляції твердих тіл вектор нормалі вказує напрямок сили відштовхування при колізії.
- Деформації та Матеріали - У симуляціях м'яких тіл чи рідин вектори нормалі допомагають визначити, як об'єкт деформується або розпливається в залежності від сил, що діють на нього. Це важливо при моделюванні матеріалів з різною текстурою та поведінкою.
- Зіткнення на колізії - Вектори нормалі використовуються для правильного визначення напрямку відштовхування об'єктів при колізіях.

Вони вказують, як повинні змінюватися швидкості та напрямки руху об'єктів для відтворення реалістичної реакції на зіткнення

- Симуляція Руйнування - При симуляції руйнування, вектори нормалі вказують, як частинки об'єкта мають розлітатися при вибуху чи подібному події. Вони допомагають визначити нові напрямки руху для кожної частинки.
- Гідродинаміка та Рідини - У симуляціях рідин вектори нормалі використовуються для визначення напрямку та інтенсивності сил, які виникають внаслідок гідродинамічних явищ, таких як тиск та опір.

Використання векторів нормалі дозволяє створювати більш точні та реалістичні фізичні симуляції у віртуальних середовищах. Однак успішна реалізація вимагає ретельної обробки великої кількості даних та врахування контексту конкретної сцени чи ефекту.

3.3 Використання освітлення

Освітлення в реальному світі надзвичайно складне і залежить від дуже багатьох факторів, які ми не можемо собі дозволити прорахувати на обмежених обчислювальних потужностях. Тому освітлення в комп’ютерній графіці базується на апроксимації реальності за допомогою спрощених моделей, які набагато легше обробляти і які виглядають відносно схожими. Ці моделі освітлення базуються на фізиці світла, як ми її розуміємо. Одна з таких моделей називається модель освітлення Фонга. Основні будівельні блоки моделі освітлення Фонга складаються з 3 компонентів: навколишнього, дифузного та дзеркального освітлення. Нижче ви можете побачити, як виглядають ці компоненти освітлення окремо та в поєднанні:

- Навколишнє освітлення – навіть коли темно, зазвичай десь у світі є світло (місяць, далеке джерело світла), тому об'єкти майже ніколи не бувають повністю темними. Щоб імітувати це, ми використовуємо

константу навколишнього освітлення, яка завжди надає об'єкту певного кольору.

- Розсіяне освітлення - імітує спрямований вплив світлового об'єкта на об'єкт. Це найбільш візуально значущий компонент моделі освітлення. Чим більше частина об'єкта звернена до джерела світла, тим яскравішим він стає.
- Дзеркальне освітлення - імітує яскраву пляму світла, яка з'являється на блискучих об'єктах. Спектральні відблиски більш схильні до кольору світла, ніж до кольору об'єкта.

Для створення візуально цікавих сцен необхідно принаймні імітувати ці 3 компоненти освітлення. Почнемо з найпростішого: зовнішнього освітлення. Розглянемо більш детально кожен з доступних методів.

Навколишнє освітлення

Зазвичай світло надходить не від одного джерела, а від багатьох джерел світла, розсіяних навколо нас, навіть якщо їх не видно одразу. Однією з властивостей світла є те, що воно може розсіюватися і відбиватися в різних напрямках, досягаючи місць, які безпосередньо не видно; таким чином, світло може відбиватися від інших поверхонь і мати непрямий вплив на освітлення об'єкта. Алгоритми, які враховують це, називаються алгоритмами глобального освітлення, але вони складні і дорогі в обчисленні.

Навколишнє освітлення є дуже спрощеною моделю глобального освітлення. Для цього прикладу використовується невеликий постійний (світлий) колір, який ми додаємо до кінцевого результату кольору фрагментів об'єкта, таким чином створюючи враження, що завжди є розсіяне світло, навіть коли немає прямого джерела світла.

Додавання зовнішнього освітлення до сцени дуже просте. Береться колір світла, множимо його на невеликий постійний коефіцієнт навколишнього освітлення, множимо на колір об'єкта і використовуємо його як колір фрагмента в

шейдері кубічного об'єкта. На Рисунку 12 можна побачити результат використання навколишнього освітлення з коефіцієнтом освітлення 0.5.

Рисунок 12 – Результат з коефіцієнтом освітлення 0.5

Цей алгоритм є найбільш легким у своїй реалізації, оскільки для отримання результату використовується лише 3 параметри – коефіцієнт освітлення, колір освітлення та колір об’єкту. Запропонований спосіб обрахування такого освітлення:

```
void main()
{
    // ambient
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

Розсіяне освітлення

Навколишнє освітлення саме по собі не дає найцікавіших результатів, але розсіяне освітлення починає давати значний візуальний вплив на об'єкт. Розсіяне освітлення надає об'єкту більшої яскравості, чим ближче його фрагменти розташовані до світлових променів від джерела світла. Для кращого розуміння, ілюстрація концепції представлена на Рисунку 13:

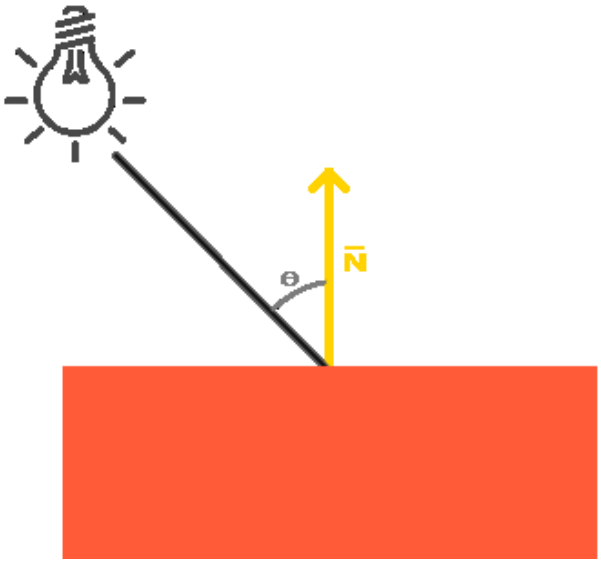

Рисунок 13 – Розсіяне освітлення

Зліва зображене джерело світла, промінь якого спрямований на окремий фрагмент нашого об'єкта. Потрібно виміряти, під яким кутом світловий промінь торкається фрагмента. Якщо промінь світла перпендикулярний до поверхні об'єкта, світло має найбільший вплив. Для вимірювання кута між променем світла і фрагментом ми використовуємо вектор, тобто вектор нормалі (тут зображений жовтою стрілкою). Оскільки 3D куб не є складною фігурою, ми можемо просто вручну додати їх до даних про вершини. Кут між цими двома векторами можна легко обчислити за допомогою точкового добутку.

Тепер коли є нормальний вектор для кожної вершини, але все ще потрібен вектор положення світла і вектор положення фрагмента. Оскільки позиція світла є єдиною статичною змінною, можна оголосити її як уніформу у шейдері фрагмента, а потім оновити форму у циклі рендерингу (або поза ним, оскільки вона не змінюється в кожному кадрі). Вектор lightPos використовується, як місце розташування джерела розсіяного світла:

Останнє, що потрібно - це власне положення фрагмента. Всі розрахунки освітлення будуть робитися у світовому просторі, тому потрібна позиція вершини, яка знаходиться у світовому просторі. Ми можемо досягти цього, помноживши атрибут положення вершини лише на матрицю моделі (не на матрицю вигляду і

проекції), щоб перетворити його на координати світового простору. Це можна легко зробити у вершинному шейдері, тому оголосимо вихідну змінну і обчислимо її координати у світовому просторі. Ця змінна in буде інтерпольована з 3 векторів світової позиції трикутника, щоб сформувати вектор FragPos, який є світовою позицією для кожного фрагмента. Тепер, коли всі необхідні змінні встановлено, ми можемо розпочати розрахунки освітлення.

Перше, що потрібно обчислити, це вектор напрямку між джерелом світла і положенням фрагмента. Вектор напрямку світла - це вектор різниці між вектором положення джерела світла і вектором положення фрагмента. Його можна легко обчислити цю різницю, віднявши обидва вектори один від одного. Також необхідно переконатися, що всі відповідні вектори є одиничними, тому нормалізуємо як нормаль, так і результуючий вектор напрямку.

При розрахунку освітлення зазвичай не цікавить величина вектора або його положення; нас цікавить лише його напрямок. Оскільки нас цікавить лише напрямок, майже всі розрахунки виконуються з одиничними векторами, оскільки це спрощує більшість обчислень (наприклад, точковий добуток).

Далі потрібно розрахувати дифузний вплив світла на поточний фрагмент, взявши точковий добуток між векторами norm і lightDir. Отримане значення потім множиться на колір світла, щоб отримати дифузну складову, в результаті чого дифузна складова буде тим темнішою, чим більший кут між обома векторами.

Якщо кут між обома векторами більший за 90 градусів, то результат точкового добутку стане від'ємним, і отримаємо від'ємну дифузну складову. З цієї причини використовується функція max, яка повертає найбільше значення з обох параметрів, щоб переконатися, що дифузна складова (а отже, і кольори) ніколи не стануть від'ємними. Освітлення для від'ємних кольорів насправді не визначено, тому краще триматися подалі від цього, якщо ви не один з тих ексцентричних художників.

Тепер, коли є і навколишній, і дифузний компонент, додається обидва кольори один до одного, а потім множиться результат на колір об'єкта, щоб отримати вихідний колір вихідного фрагмента.

На Рисунку 14 можна бачити, що при розсіяному освітленні куб знову починає виглядати як справжній куб. Якщо візуалізувати нормальні вектори в і переміщати камеру навколо куба, можна побачити, що чим більший кут між нормальним вектором і вектором напрямку світла, тим темнішим стає фрагмент.

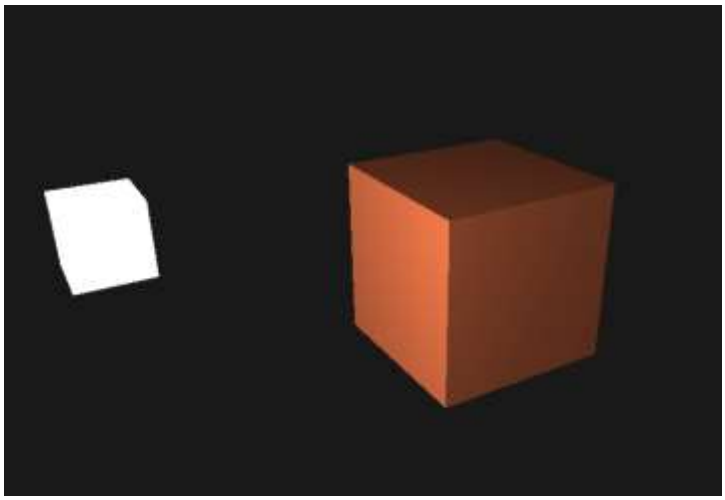

Рисунок 14 – Результат розсіяного освітлення

Дзеркальне освітлення

Подібно до розсіяного освітлення, дзеркальне освітлення базується на векторі напрямку світла та векторах нормалей об'єкта, але цього разу воно також базується на напрямку погляду, наприклад, з якого боку гравець дивиться на фрагмент. Специфічне освітлення базується на відбиваючих властивостях поверхонь. Якщо уявити поверхню об'єкта як дзеркало, то дзеркальне освітлення буде найсильнішим там, де світло відбите від поверхні. Цей ефект на наступному зображений на Рисунку 15.

Вектор відбиття обчислюється, відображаючи напрямок світла навколо вектора нормалі. Потім вираховується кутова відстань між цим вектором відбиття і напрямком погляду. Чим менший кут між ними, тим більший вплив дзеркального світла. В результаті видно відблиск, коли камера дивиться на напрямок світла, відбитого від поверхні.

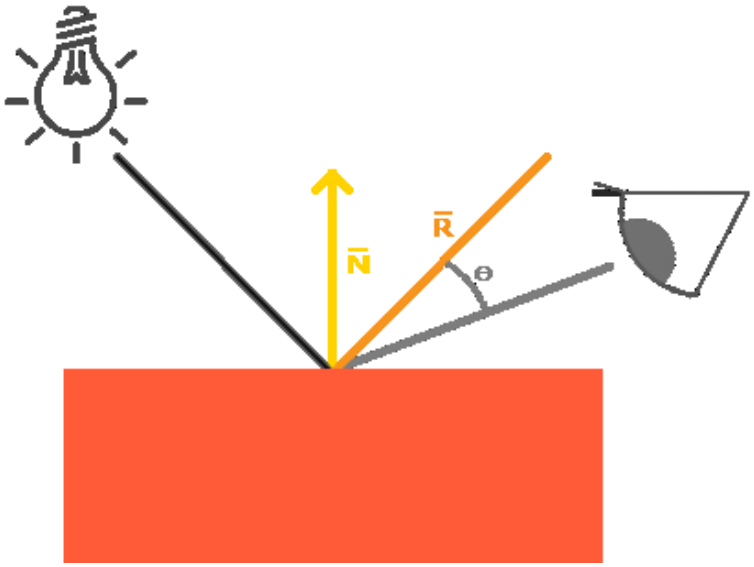

Рисунок 15 – Дзеркальне освітлення

Вектор погляду - це ще одна додаткова змінна, необхідна для дзеркального освітлення, яку можливо обчислити, використовуючи положення глядача в просторі та положення фрагмента. Потім обчислюється інтенсивність дзеркального освітлення, множиться на колір світла і додається до навколишнього та дифузного компонентів.

Щоб отримати координати світового простору глядача, береться вектор положення об'єкта камери (яким, звісно, є глядач). Тож додається ще одна уніформа до фрагментного шейдера і передається їй вектор положення камери.

Тепер, коли є всі необхідні змінні, можена обчислити інтенсивність світла. Спершу визначається значення спектральної інтенсивності, щоб надати спектральному висвітленню середньо-яскравий колір, щоб воно не мало надто сильного впливу:

Якщо встановити значення 1.0f, то отримаємо дуже яскраву дзеркальну складову, що трохи забагато для коралового куба. Вартно зазначити, що береться від'ємний вектор lightDir. Функція reflect очікує, що перший вектор буде спрямований від джерела світла до позиції фрагмента, але вектор lightDir наразі спрямований навпаки: від фрагмента до джерела світла (це залежить від порядку віднімання раніше, коли обчислювався вектор lightDir). Щоб переконатися, що

отриманий правильний вектор відбиття, змінимо його напрямок, спочатку від'єднавши вектор lightDir. Другий аргумент очікує вектор нормалі, тому надається нормалізований вектор нормалі.

Після цього залишається обчислити власне дзеркальну складову. Спочатку обчислюється точковий добуток між напрямком погляду і напрямком відбиття (переконайтеся, що він не від'ємний), а потім підноситься до степеня N. Це значення N є значенням блиску підсвічування. Чим вище значення блиску об'єкта, тим більше він відбиває світло, а не розсіює його навколо, і тим меншим стає відблиск. Рисунок 16 показує візуальний вплив різних значень блиску та як це впливає не результуючу картинку.

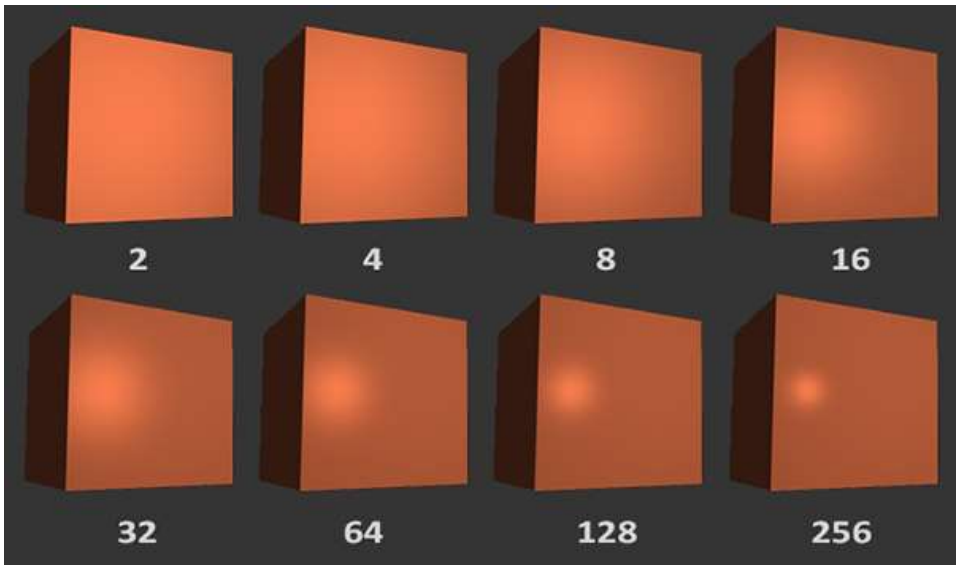

Рисунок 16 – Порівняння результату в залежності від коефіцієнту блиску

У цьому випадку значення коефіцієнту блиску буде 32. Єдине, що залишилося зробити, це додати його до компонентів навколишнього та дифузного освітлення і помножити об'єднаний результат на колір об'єкта. На Рисунку 17 можна бачити результат роботи алгоритму, а також пересідчитись в тому, що відносно різних кутів між вектором наглядача та вектору відбиття, змінюється яскравість відбитого освітлення.

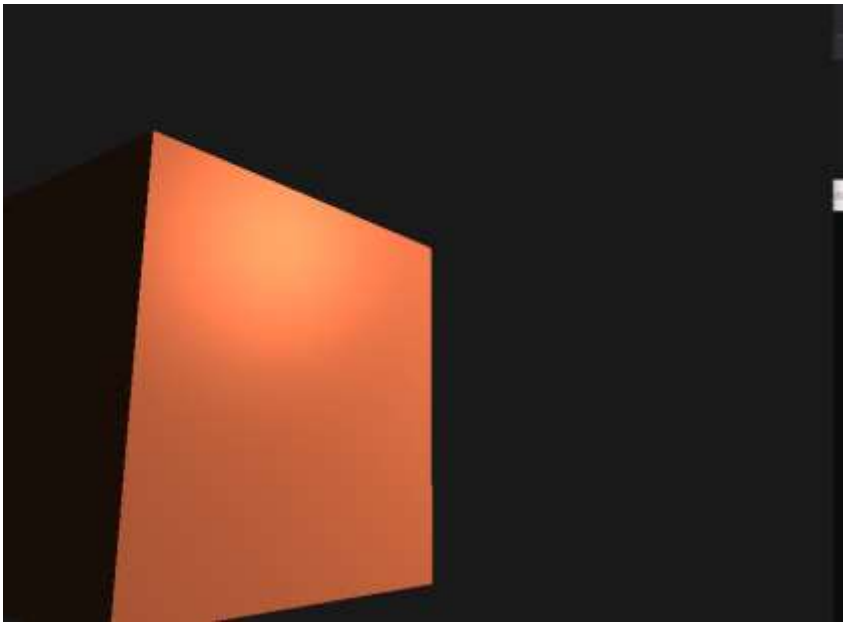

Рисунок 17 – Результат дзеркального освітлення 1

Варто зазначити, що тут розібрані лише найпростіші та базові алгоритми освітлення. В сучасних іграх та складних додатках застосовуються новіші та набагато більш складні прийоми. Але для розуміння того, як працює запропонований метод використання нормалей та для його реалізації, цього достатньо. Оскільки досліджуваний ефект досягається за допомогою підсвічування або затемнення певних ділянок.

3.4 Спрощення поверхні за допомогою квадратичної метрики похибок

Скорочуючи довільні пари вершин (а не лише ребра), алгоритм здатен об'єднувати незв'язані області моделей. Це може сприяти набагато кращим наближенням, як візуально, так і з точки зору геометричної похибки. Для того, щоб дозволити топологічне з'єднання, наша система також підтримує моделі поверхонь, що не є многовидом.

Сучасні програми комп'ютерної графіки вимагають складних, високодеталізованих моделей для підтримки переконливого рівня реалістичності. Відповідно, моделі часто створюються або купуються з дуже високою роздільною здатністю, щоб задовольнити цю потребу в деталях. Однак не завжди потрібна

повна складність таких моделей, а оскільки обчислювальні витрати на використання моделі безпосередньо пов'язані з її складністю, корисно мати простіші версії складних моделей.

Як і більшість інших робіт у цій галузі, алгоритм зосереджений на спрощенні полігональних моделей. Будемо вважати, що модель складається лише з трикутників. Це не означає втрати загальності, оскільки кожен полігон у вихідній моделі може бути тріангульований як частина фази попередньої обробки. Для досягнення більш надійних результатів, коли кути двох граней перетинаються в одній точці, грані слід визначати як такі, що мають спільну одну вершину, а не використовувати дві окремі вершини, які випадково збігаються у просторі. збігаються у просторі.

Розроблений алгоритм, створює спрощені версії полігональних моделей. Він базується на ітеративному скороченні пар вершин (узагальнення скорочення ребер). Під час роботи алгоритму в кожній вершині зберігається апроксимація геометричної похибки.

У кожній вершині поточної моделі. Ця апроксимація похибки представляється за допомогою квадратичних матриць. Основними перевагами нашого алгоритму є

- Ефективність: Алгоритм здатен досить швидко спрощувати складні моделі. Наприклад, наша реалізація може створити 100-кратну апроксимацію моделі з 70 000 облич за 15 секунд. Апроксимація похибки також є дуже компактною, вимагаючи лише 10 чисел з плаваючою комою на вершину.
- Якість: Наближення, отримані за допомогою алгоритму, зберігають високу точність до оригінальної моделі. Основні особливості моделі зберігаються навіть після значного спрощення.
- Універсальність: На відміну від більшості інших алгоритмів спрощення поверхні, цей алгоритм здатен об'єднувати незв'язані області моделі разом - процес, який називається агрегацією. За умови, що

збереження топології об'єкта не є важливою проблемою, це може сприяти кращому наближенню моделей з багатьма роз'єднаними компонентами.

Метою спрощення полігональної поверхні є отримання на вході полігональної моделі, а на виході - спрощеної моделі (тобто апроксимації оригіналу). Вважається, що вхідна модель (Mn) була тріангульована. Цільова апроксимація (Mg) задовольнятиме деякий заданий цільовий критерій, який зазвичай є або бажаною кількістю облич, або максимально допустимою похибкою. Нас цікавлять алгоритми спрощення поверхні, які можуть бути використані в системах рендерингу для моделювання з декількома розв'язками - генерації моделей з відповідним рівнем деталізації для поточного контексту.

Не припускається, що топологія моделі повинна бути збережена. У певних сферах застосування, наприклад, у медичній візуалізації, збереження топології об'єкта може бути необхідним. Однак у таких сферах застосування, як рендеринг, топологія менш важлива, ніж загальний вигляд. Наш алгоритм здатен як закривати топологічні діри, так і з'єднувати незв'язані області. Багато попередніх алгоритмів спрощення явно чи неявно припускали, що їхні вхідні поверхні були і мають залишатися поверхнями многовидів. Процес агрегації буде регулярно створювати нерозкладні області.

Спрощення поверхонь - останніми роками проблемі спрощення поверхонь та більш загальній проблемі багаторозв'язного моделювання приділяється все більше уваги. Для спрощення поверхонь було сформульовано декілька різних алгоритмів. Алгоритми, які мають найбільше відношення до нашої роботи, можна умовно розділити на 3 класи:

- Видалення вершин - Шредер та ін. [8] та інші описують алгоритм який називається децимацією вершин. Їх метод ітеративно вибирає вершину для видалення, видаляє всі суміжні грані та повторно аранжує отриману дірку. Soucy та Laurendeau [9] описали більш складний, але по суті схожий алгоритм. Хоча вони забезпечують достатню ефективність і якість, ці методи не зовсім підходять для нашої мети. Обидва методи використовують схеми класифікації вершин і ретріангуляції, які за своєю

суттю обмежені множиною поверхонь, і вони ретельно підтримують топологію моделі. Хоча це і є важливими особливостями в деяких областях, вони є обмеженнями для систем рендерингу з декількома роздільними здатностями.

- Кластеризація вершин - Алгоритм, описаний Россіньяком та Боррелом [10], є одним з небагатьох, здатних обробляти довільні полігональні вхідні дані. Навколо вихідної моделі розміщується обмежувальна рамка, яка розбивається на сітку. У кожній комірці вершини комірки об'єднуються в одну вершину, і відповідно оновлюються грані моделі. Цей процес може бути дуже швидким і вносити радикальні топологічні зміни в модель. Однак, хоча розмір комірок сітки обмежує геометричну похибку, якість результату часто є досить низькою. Крім того, важко побудувати апроксимацію з певною кількістю граней, оскільки кількість граней лише опосередковано визначається заданими розмірами сітки. Точність отриманої апроксимації також залежить від точного положення та орієнтації вихідної моделі відносно навколишньої сітки. Цей уніфікований метод можна легко узагальнити для використання адаптивної структури сітки, наприклад, вісімки [11]. Це може покращити результати спрощення, але все одно не забезпечує бажаної якості та контролю.
- Ітеративне стискання ребер - Було опубліковано кілька алгоритмів, які спрощують моделі шляхом ітеративного скорочення. Суттєва різниця між цими алгоритмами полягає в тому, як вони обирають ребро для скорочення. Деякі відомі приклади таких алгоритмів - це алгоритми Хоппе, Ронфарда і Россіньяка та Гезеца. Всі ці алгоритми, схоже, були розроблені для використання на різноманітних поверхнях, хоча стискання країв можна використовувати і на поверхнях, що не є різноманітними. Виконуючи послідовне стискання країв, вони можуть закрити дірки в об'єкті, але не можуть з'єднати незв'язні області. Якщо дуже важливо, щоб наближена модель знаходилась на деякій відстані від

оригінальної моделі і щоб її топологія залишалась незмінною, можна використовувати метод спрощувальних огинаючих Коена та ін. у поєднанні з одним з наведених вище алгоритмів спрощення. До тих пір, поки будь-яка модифікація, зроблена в моделі, не виходить за межі огинаючих, глобальна гарантія похибки може бути збережена. Однак, хоча це і забезпечує сильні обмеження на похибку, метод за своєю суттю обмежений орієнтованими поверхнями многовиду і ретельно зберігає топологію моделі. Знову ж таки, це часто є обмеженням з метою спрощення візуалізації. Жоден з цих раніше розроблених алгоритмів не забезпечує поєднання ефективності, якості та загальності, якого прагне цей спосіб. Алгоритми децимації вершин не підходять для наших потреб; вони обережно підтримують топологію моделі і зазвичай припускають множинну геометрію. Алгоритми кластеризації вершин є дуже загальними і можуть бути дуже швидкими. Однак вони погано контролюють свої результати, і ці результати можуть бути досить низької якості. Алгоритми стиснення ребер не можуть підтримувати агрегацію. Цей алгоритм, підтримує як агрегацію, так і високоякісні наближення. Він має багато спільного з кластеризацією вершин, а також з якістю та контролем алгоритмів ітеративного стиснення. Він також дозволяє швидше спрощення, ніж деякі більш якісні методи.

Висновок - алгоритм може створювати наближення високої точності за досить короткий проміжок часу. Для проходження тестів була вибрана модель корови. Вся ця послідовність корів була побудована приблизно за секунду. Зверніть увагу, що такі особливості, як роги та копита, залишаються впізнаваними завдяки багатьом спрощенням. Лише на дуже низьких рівнях деталізації вони починають зникати. Як описано раніше, наш алгоритм намагається оптимізувати розміщення вершин після скорочень. На дуже низьких рівнях деталізації ефект скромний. Однак на більш прийнятних рівнях деталізації ефект є суттєвим; оптимальне розміщення вершин може зменшити загальну похибку на цілих 50%. Під час експериментів було виявлено, що використання оптимального розміщення вершин

призводить до створення більш правильних за формою сіток. Хоча більшість деталей моделі зникла, основна структура об'єкта залишилася неушкодженою; основні риси, такі як голова, ноги, хвіст і вуха, є очевидними, хоча й дуже спрощеними. Хоча дрібномасштабна текстура рельєфу значною мірою зникла, всі основні риси рельєфу залишилися. Також зверніть увагу, що відкриті кордони були належним чином збережені. Без описаних раніше обмежень на кордоні, він був би значно розмитий.

Описаний алгоритм спрощення поверхні, який здатен швидко створювати високоточні апроксимації полігональних моделей. Наш алгоритм використовує ітеративні парні скорочення для спрощення моделей та метрики квадратичної похибки для відстеження наближеної похибки моделі в процесі її спрощення. Квадри, що зберігаються в кінцевих вершинах, також можуть бути використані для характеристики загальної форми поверхні. цей алгоритм має можливість об'єднувати незв'язані ділянки моделей, зберігаючи при цьому досить високу якість результатів. Хоча більшість попередніх алгоритмів також за своєю суттю обмежені різноманітними поверхнями, ця система цілком здатна обробляти і спрощувати об'єкти, що не є різноманітними. Цей алгоритм забезпечує корисну золоту середину між дуже швидкими, але низькоякісними методами, такими як кластеризація вершин, і дуже повільними, але високоякісними методами, такими як оптимізація сітки. На Рисунку 18 зображені моделі текстури після кожної ітерації алгоритму.

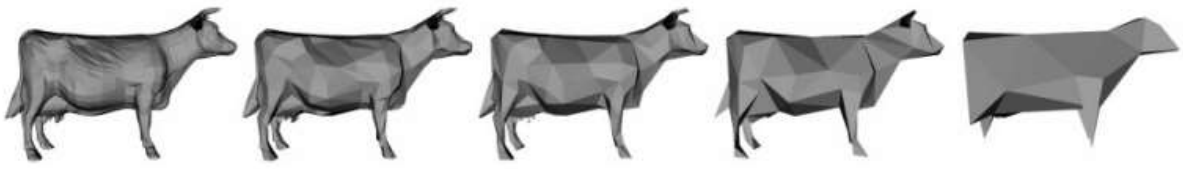

Рисунок 18 – Послідовність наближень, згенерованих за допомогою алгоритму

3.5 Порівняння способу використання мапи нормалей і запропонованого способу

Використання мапи нормалей для покращення візуальної складової гри було запропоновано досить давно, на початку 21 століття. Ця технологія дозволила значно підвисити якість зображення, при чому незначною мірою збільшити використання ресурів. З того часу, можливості персональних комп’ютерів значно виросли, але актуальність цього методу все ще залишилася.

В сучасних іграх та інших програмах, отримане зображення дуже наближене до реального світу. Деякі додатки здатні генерувати настільки реалістичне зображення, що доволі важко відрізнити від реальності. Мапи нормалей відіграють у цьому значну роль. Створити у спеціальних графічних додатках моделі та текстури надвисокої якості не є проблемою, але вони будуть складатися і десятків мільйонів мініатюрних полігонів. На ігровій сцені, зазвичай знаходяться десятки різних об’єктів, це можуть бути моделі людей, транспорту, дерев, будівель, тварин, тощо. Жодне найсучасніше апаратне забезпечення не зможе рендерити таку кількість об’єктів одночасно. Саме тому необхідні мапи нормалі, оскільки вони дають можливість щоб текстури з невеликою роздільною здатністю, виглядати аналогічно. Це чудовий спосіб економії ресурсів, але і в нього є свої недоліки:

- Залежність від Текстур - Використання мап нормалей передбачає наявність текстур відносно високої якості. Якщо текстури низької роздільності або погано створені, це може призвести до втрати деталей та артефактів у рендерингу.
- Обмежена Деталізація - Мапи нормалей дозволяють лише приховати або виокремити деталі, які вже присутні у текстурі. Вони не можуть додати нові деталі, що може бути обмежливим при створенні найреалістичніших ефектів. Тут важливо зауважити, що використання мапи нормалей, не додає нові елементи до текстури, а лише створює ілюзію наявності рельєфності.

- Артефакти на Границях Об'єктів - Під час використання мап нормалей можуть виникати артефакти, особливо на границях об'єктів або при різкій зміні напрямку нормалей.
- Потреба у Додаткових Ресурсах - Використання мап нормалей може збільшити вимоги до ресурсів, оскільки вони вимагають додаткової пам'яті для зберігання та обробки текстур. Оскільки якщо використовується текстура в роздільністю в 4К (4096×3072), то потрібно мати і карту нормалей відповідну. Це напряму впливає як на розмір текстур, при зберіганні на жорсткому диску, так і розміри в оперативній пам’яті.
- Специфічність Підготовки Даних - Підготовка високоякісних мап нормалей може бути трудомісткою та вимагати досить специфічних інструментів та навичок. Зазвичай мапи нормалей генеруються автоматизовано, під час створення текстури у спеціальних графічних додатках. Але для складних об’єктів, можуть виникати помилки, які дуже важко знайти, і, відповідно, для їх виправлення необхідне втручання спеціаліста.
- Чутливість до Освітлення - Ефективність мап нормалей може бути суттєво залежати від умов освітлення. У деяких сценах або при різних виглядах світла можуть виникати непередбачувані артефакти.

Для подальшого розгляду альтернативного алгоритму, що дозволить використання мапи нормалей, необхідно розуміти все етапи. В залежності від поставленої задачі та наявних засобів, ці етапи можуть видозмінюватися, але загальноприйнята структура виглядає так:

- Створення мапи нормалей (Генерація) - Мапа нормалей може бути створена різними способами. Один із найпоширеніших — використання програм для моделювання, де можна створити або редагувати поверхні об'єктів та згенерувати відповідні нормалі. Іншим методом є

використання спеціальних програм для генерації текстур нормалей, таких як CrazyBump, Substance Designer тощо.

- Перетворення у текстуру (Кодування) - Отримана мапа нормалей може бути закодована у текстурні дані. Зазвичай використовуються формати зображень, які підтримують альфа-канал (наприклад, PNG або TIFF). У текстурі, окрім самих векторів нормалей, можуть також міститися додаткові дані, такі як висота, шорсткість тощо.
- Використання під час рендерингу (відтворення) - Під час рендерингу мапа нормалей використовується для модифікації основного вектора нормалі поверхні, що дозволяє створювати ефекти відбиття світла та деталізацію.
- Використання під час рендерингу (трансформація) - Після отримання значень з текстури, векторні дані можуть бути відновлені. Такий процес часто включає в себе розкодування значень з текстурного формату, а потім їхню нормалізацію для отримання оригінальних векторів нормалей.

Процес генерації мапи нормалей полягає у створенні текстури, де кожен піксель представляє собою вектор нормалі для відповідної точки на поверхні об'єкта, етапи можуть змінюватися відповідно до поставлених задача, або програмного забезпечення, що використовує чи генерує їх. В загальному можна виділити наступні: визначення текстурних координат - для кожної вершини об'єкта визначаються текстурні координати, які представляють точки на текстурі. Генерація текстури висот - спочатку генерується текстура висот, де кожен піксель відображає висоту відносно базового рівня. Це може бути результатом висотної карти, текстурного шуму або інших методів. Наступним є обчислення нормалей - для кожного пікселя на текстурі висот обчислюється вектор нормалі. Це може виконуватися шляхом використання диференціації текстурних координат або інших алгоритмів. Потім відбувається конвертація результатів в текстуру(кодування) – кожен вектор конвертується в компонент (R, G, B) пікселя

та нормалізація - вектори нормалей нормалізуються, щоб забезпечити, що всі вони мають одиничну довжину. Це важливо для коректного використання мапи нормалей під час рендерингу. Згенерована мапа нормалей може бути збережена у форматі зображення (зазвичай у форматах, таких як PNG або TIFF) для подальшого використання у програмах графічного рендерингу.

Процес відтворення на основі мап нормалей включає кілька етапів і відбувається під час рендерингу 3D-сцени. Основними кроками є: підготовка моделі, використання мами нормалей, зміна нормалей, визначення освітлення, формування зображення. Модель об'єкта повинна мати включені координати вершин, текстурні координати та, важливо, вектори нормалей. Ці вектори нормалей можуть бути оригінальними або модифікованими за допомогою мап нормалей. Вектори нормалей зчитуються з мапи нормалей, яка зазвичай представлена у вигляді текстури. Значення пікселів цієї текстури визначають напрямок зміни нормалі для кожної конкретної точки на поверхні об'єкта. Оригінальні вектори нормалей, отримані з моделі, модифікуються на основі значень з мапи нормалей. Зазвичай, це включає в себе додавання або віднімання значень з мапи нормалей до оригінальних нормалей. Змінені вектори нормалей використовуються для розрахунку освітлення на поверхні об'єкта. Це може включати в себе взаємодію з джерелами світла, тіні, відбиття та інші аспекти освітлення. Залежно від використовуваного матеріалу, модифіковані вектори нормалей можуть бути використані для розрахунку різних атрибутів, таких як блеск, шорсткість, кольори та інші характеристики матеріалу. Остаточні обчислені значення використовуються для формування зображення об'єкта на екрані. Цей процес може бути виконаний за допомогою графічного двигуна, який враховує модифіковані вектори нормалей при визначенні кінцевого відображення.

Основою відмінністю запропонованого метода є те, що не використовується завчасно згенерована мапа нормалей. Вона ‘створюється’ під час процесу рендерингу. Головною перевагою є те, що непотрібно завчасно підготовлювати додаткову текстуру, що зменшую ‘вагу’ об’єкту. Але, в той же час необхідно використовувати альтернативні шляхи для їх обрахунку. Обчислення нормалей під

час рендерингу без використання завчасно підготовленої моделі часто здійснюється на основі геометричної інформації, що мається на момент рендерингу. Цей процес забезпечує більш гнучку і динамічну генерацію нормалей на льоту, враховуючи конкретні умови та обрані параметри.

В цьому випадку буде використовуватись алгоритм, що був описано попередньо при дослідженні математичних властивостей нормалей, а саме - обчислення нормалей за допомогою векторного множення - нормалі для кожного трикутника можуть бути обчислені за допомогою векторного множення векторів, які визначають ребра трикутника. Після цього може використовуватися середнє арифметичне для отримання нормалі поверхні.

Всі реальні сцени заповнені сітками, кожна з яких складається з сотень, а може й тисяч трикутників. Ми підвищили реалістичність, наклавши 2D текстури на ці плоскі трикутники, приховуючи той факт, що полігони - це просто крихітні плоскі трикутники. Текстури допомагають, але якщо уважно придивитися до сітки, то все одно досить легко побачити плоскі поверхні, що лежать під нею. Однак більшість реальних поверхонь не є пласкими і мають багато (нерівних) деталей.

Наприклад, візьмемо цегляну поверхню. Цегляна поверхня є досить шорсткою і, очевидно, не зовсім рівною: вона містить заглиблені цементні смуги та багато детальних маленьких отворів і тріщин. Якщо ми подивимось на таку цегляну поверхню в освітленій сцені, то ефект занурення легко порушиться. На Рисунку 19 ми бачимо текстуру цегли, нанесену на плоску поверхню, освітлену точковим світлом.

Рисунок 19 – Приклад текстури, накладеної на плоску поверхню

Освітлення не бере до уваги дрібні тріщини та дірки і повністю ігнорує глибокі смуги між цеглинами; поверхня виглядає ідеально плоскою. Ми можемо частково виправити плаский вигляд за допомогою дзеркальної карти, щоб уявити, що деякі поверхні менш освітлені через глибину або інші деталі, але це більше схоже на хакерство, ніж на реальне рішення. Що нам потрібно, так це спосіб повідомити систему освітлення про всі маленькі деталі поверхні, що нагадують глибину.

Якщо подумати про це з точки зору світла: чому поверхня освітлюється як абсолютно пласка? Відповідь - вектор нормалі поверхні. З точки зору світлотехніки, єдиний спосіб, яким вона визначає форму об'єкта, - це перпендикулярний вектор нормалі. Поверхня цегли має лише один нормальний вектор, і, як наслідок, поверхня рівномірно освітлюється відповідно до напрямку цього нормального вектора. А що, якщо замість загальної нормалі, однакової для кожного фрагмента, використати фрагментарну нормаль, різну для кожного фрагмента? Таким чином, ми можемо трохи відхилити вектор нормалі на основі дрібних деталей поверхні; це створює ілюзію, що поверхня набагато складніша.

Використовуючи нормалі пофрагментно, ми можемо створити ілюзію для освітлення, змусивши його повірити, що поверхня складається з крихітних площин

(перпендикулярних до векторів нормалей), що значно підвищує деталізацію поверхні. Ця техніка використання нормалей на фрагмент у порівнянні з нормалями на поверхню називається відображенням нормалей або відображенням нерівностей.

Вектори нормалей на нормальній карті виражаються в дотичному просторі, де нормалі завжди спрямовані приблизно в додатному напрямку z. Дотичний простір - це простір, локальний до поверхні трикутника: нормалі відносяться до локальної системи відліку окремих трикутників. Думайте про нього як про локальний простір векторів нормальної карти; всі вони визначені як такі, що вказують на додатний напрямок z, незалежно від остаточного перетвореного напрямку. Використовуючи спеціальну матрицю, ми можемо перетворити нормальні вектори з цього локального простору дотичних у світові координати або координати виду, орієнтуючи їх вздовж кінцевого напрямку відображеної поверхні.

Така матриця називається матрицею TBN, де літери позначають вектор дотичної, бітангенса та нормалі. Саме ці вектори нам потрібні для побудови цієї матриці. Щоб побудувати таку матрицю зміни базису, яка перетворює вектор дотичного простору в інший координатний простір, нам потрібні три перпендикулярні вектори, вирівняні вздовж поверхні нормальної карти: вектор вгору, вектор вправо і вектор вперед; подібно до того, як ми це робили в розділі про камеру.

Вектор вгору, є вектор нормалі поверхні. Вектор вправо і вектор вперед - це вектор дотичної і бітангенса відповідно. Рисунок 20 показує всі три вектори на поверхні.

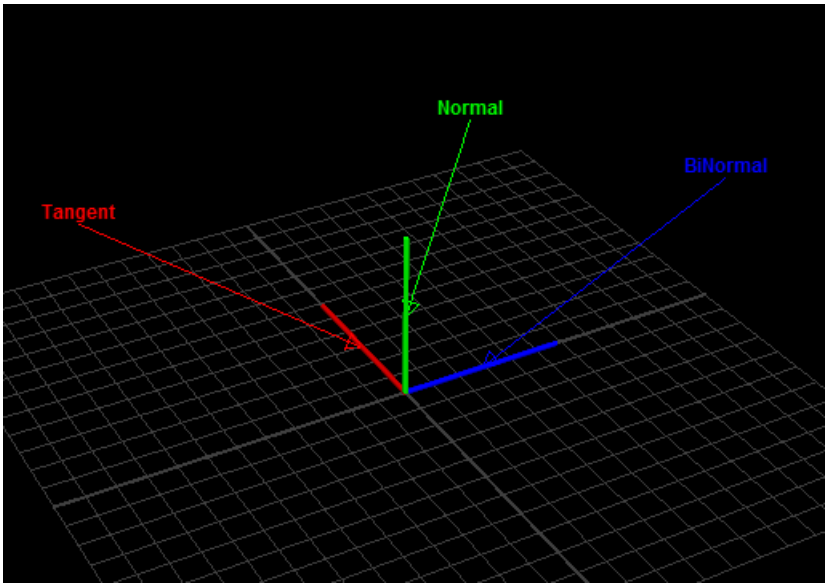

Рисунок 20 – Розташування нормалей на площині

Обчислення векторів дотичної та бітангенса не таке просте, як обчислення вектора нормалі. На зображенні ми бачимо, що напрямок вектора дотичної та бітангенса нормальної карти збігається з напрямком, в якому ми визначили координати текстури поверхні. Ми використаємо цей факт для обчислення векторів дотичної та бітангенса для кожної поверхні. Для їх отримання скористуємось Рисунком. 21.

Карта нормалей визначена у дотичному просторі, тому одним із способів вирішення проблеми є обчислення матриці для перетворення нормалей з дотичного простору в інший простір таким чином, щоб вони були вирівняні з напрямком нормалі поверхні: тоді всі вектори нормалей будуть спрямовані приблизно у додатному напрямку y. Чудовою особливістю дотичного простору є те, що ми можемо обчислити цю матрицю для будь-якого типу поверхні, щоб належним чином вирівняти напрямок z дотичного простору з напрямком нормалі поверхні.

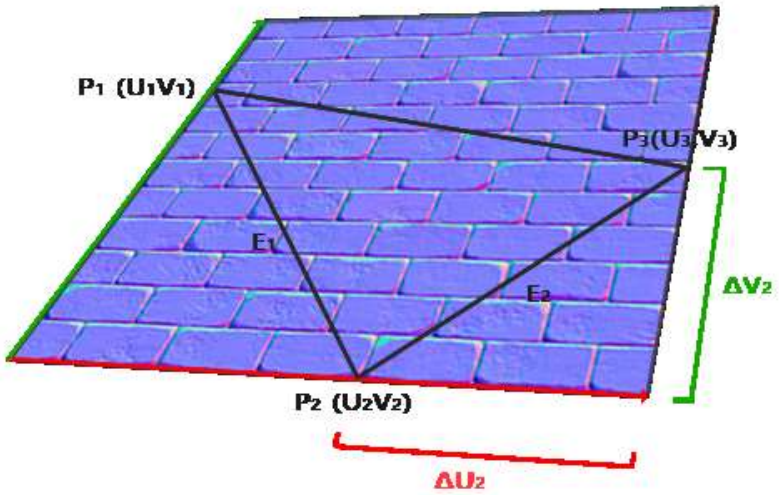

Рисунок 21 – Ілюстрація ребер поверхні

Із рисунку видно, що різниця текстурних координат ребра $\mathbf{E_2}$ трикутника (позначені як **ΔU2** та **ΔV2**) виражені у тому ж напрямку, що й тангент вектор **T** та бітангенс вектор **B** . Тому ми можемо записати обидва відображені ребра $\mathbf{E_1}$ та $\mathbf{E_2}$ трикутника як лінійну комбінацію дотичного вектора **T** і дотичного вектора **B**.

$$E_1 = \Delta U_1 T + \Delta V_1 B$$
$$E_2 = \Delta U_2 T + \Delta V_2 B$$

Також це рівняння може бути записане у іншому вигляді:

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (Tx, Ty, Tz) + \Delta V_1 (Bx, By, Bz)$$
$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (Tx, Ty, Tz) + \Delta V_2 (Bx, By, Bz)$$

Ми можемо обчислити **E** як вектор різниці між двома положеннями трикутника, а **ΔU** і **ΔV** - як різницю текстурних координат. У нас залишається дві невідомі (тангенс **T** і бітангенс **B**) і два рівняння. Останнє рівняння дозволяє записати його в іншій формі: у формі матричного множення [13]:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} Tx & Ty & Tz \\ Bx & By & Bz \end{bmatrix}$$

Це дозволяє нам розв'язати для **T** та **B [19]** . Для цього нам потрібно обчислити обернену матрицю координат дельта-текстури. Це приблизно дорівнює 1 над визначником матриці, помноженим на матрицю-суміжну до неї.

$$\begin{bmatrix} Tx & Ty & Tz \\ Bx & By & Bz \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Останнє рівняння дає нам формулу для обчислення дотичного вектора **T** і вектора бітангенса **B** від двох ребер трикутника та його текстурних координат. Тепер, маючи значення векторів **T** і **B**, можна розрахувати значення вектору нормалі $N = T \times B$.

Ми можемо запустити цей процес для кожного трикутника в сітці і згенерувати вектори дотичних і бітангенсів для кожного трикутника (однакові вектори для трьох вершин трикутника). Загальноприйнятою практикою є зберігання дотичної/бітангенса для кожної вершини шляхом усереднення всіх дотичних/бітангенсів трикутників, які мають спільну вершину (так само, як ми робимо для вершинних нормалей). Це робиться для того, щоб згладити ефект, коли ми рухаємося по грані трикутника, і уникнути різких розривів на ребрах сусідніх трикутників. Це тріо тангент-бітангент-нормаль тепер може слугувати основою системи координат і використовуватися для перетворення нормалі з карти нормалей у локальний об'єктний простір. Наступним кроком буде перетворення її у світовий простір і використання для розрахунків освітлення.

Існує одна особливість, яку потрібно врахувати у цих обчисленнях. На рівні пікселів дотична-бітангенс нормалі не є ортонормальним базисом (три вектори одиничної довжини, які перпендикулярні один до одного). Це пояснюється двома причинами - по-перше, ми усереднюємо дотичні та нормалі до вершини відповідно до трикутників, які її розділяють, а по-друге, дотичні та нормалі інтерполюються растеризатором, і на піксельному рівні ми бачимо інтерпольований результат. Це призводить до того, що тангент-бітангент і нормаль втрачають частину своїх "ортонормальних якостей". Але для переходу від дотичного простору до світового нам знадобиться ортонормований базис. Рішенням є використання процесу Грама-

Шмідта. Цей процес бере групу векторів і перетворює їх на ортонормований базис. У двох словах, процес виглядає наступним чином: виберіть вектор "A" з групи і нормалізуйте його. Потім виберіть вектор "B" і розбийте його на дві компоненти (дві компоненти - це вектори, сума яких дорівнює "B"), де перша компонента вказує в напрямку "A", а друга компонента перпендикулярна до нього. Тепер замініть "B" компонентою, яка перпендикулярна до "A", і нормалізуйте її. Цей процес повторюється для всіх векторів у групі.

Висновки до розділу 3

У цьому розділі були розглянуті основні способи підсвічування об'єктів на сцені, особливості кожного з варіантів та порівняли їх візуальний результат. Було досліджено як саме освітлення впливає, в тому числі, і на візуальний ефект рельєфності, створений з використанням технології мапінгу нормалей.

Було розібрано математичний та геометричний сенс вектора нормалі та методи його розрахунку, які надалі були використанні і розрахунках для власного алгоритму.

Було досліджено в яких ще аспектах ігрового рушія використовується принцип нормалі. В деталях були розглянуті випадки використання для фізичних симуляцій, та способи обрахунки деяких з них.

Як вже зрозуміло, вектори нормалі є ключовим елементом в багатьох алгоритмах. На розглянутих прикладах – це освітлення та створення ілюзії об'ємності у плоских об'єктах. Також цей вектор активно використовується у фізичних симуляціях. Таким чином, запропонований спосoб, може бути перенесений і на інші обчислення, що виконуються під час роботи ігрового рушія.

При порівнянні вже існуючого алгоритму мапінгу і в запропонованому новому способу, були розглянуті його недоліки та методи як їх можна вирішити. Деякі з них і вирішує запропонований спосіб. А оскільки він не потребує попередньої генерації текстури нормалей, то в результаті текстури будуть займати менше місця. Особливо це важливо в контексті оперативної пам’яті, тому що при

рендерингу масштабних сцен одночасно може відображатися близько тисячі об’єктів – сумарно це перетворюється у значну економію пам’яті. Також запропонований спосіб є більш гнучким якщо об’єкт, на якого накладена текстура буде деформований від впливу інших явищ. Оскільки текстура с нормалями не може модифікуватися під час виконання програми, то ефект мапінгу нормалей може бути накладений невірно.

В той же час, варто зазначити, що запропонований спосіб дещо збільшує навантаження на центральний процесор, що в теорії може викликати недостачу ресурсів для інших обчислень. Ця проблема вирішується методами розпаралелювання програм та перенесення обчислень на графічне ядро, але це значно впливає на рівень складності самої програми.

РОЗДІЛ 4
ІМПЛЕМЕНТАЦІЯ ТА ПОРІВНЯННЯ РЕЗУЛЬТАТІВ

4.1 Використання інтерфейсу графічного ядра

Графічні інтерфейси, такі як OpenGL, DirectX і Vulkan, представляють собою набори програмних інтерфейсів для розробки графічних додатків, зокрема в ігровій індустрії та комп'ютерній графіці. Кожен із цих інтерфейсів має свої унікальні характеристики, проте всі вони спрямовані на надання програмістам доступу до ресурсів графічного апарату комп'ютера для створення візуально захоплюючих зображень та взаємодії з графічним обладнанням.

- OpenGL

Є відкритим стандартом для реалізації 2D і 3D графіки. Він підтримується на багатьох платформах і використовується в широкому спектрі додатків, включаючи ігри, візуалізацію даних та наукові дослідження. Розробники використовують OpenGL для створення графічних додатків, які працюють на різних платформах. Він надає можливість реалізації широкого спектру графічних ефектів та взаємодії з графічним обладнанням.

Переваги: OpenGL підтримується на різних операційних системах, включаючи Windows, Linux та MacOS, що робить його привабливим для кросплатформенної розробки; велика спільнота розробників та багато ресурсів для вивчення та допомоги; багатий функціонал та можливості для реалізації різноманітних графічних ефектів; легкий для опанування

Недоліки: у порівнянні з Vulkan, OpenGL може бути менш ефективним на низькорівневому рівні; у порівнянні з сучасними стандартами, такими як Vulkan, OpenGL може виглядати застарілим.

- DirectX розроблений корпорацією Microsoft і є стандартом для реалізації графіки, аудіо та введення у Windows-платформах. Використовується для розробки ігор та інших графічних додатків для платформ Windows [20]. Включає в себе графічні, аудіо та введення API.

Переваги: DirectX є невід'ємною частиною платформи Windows, що полегшує розробку графічних додатків для цієї операційної системи; включає в себе не тільки графічний API, але й аудіо та введення, що забезпечує повний пакет для розробки ігор.

Недоліки: DirectX обмежений платформою Windows, що робить його менш привабливим для кросплатформенної розробки; ліцензійні обмеження, пов'язані з ліцензією Microsoft, можуть впливати на деякі аспекти використання.

- Vulkan є відкритим стандартом, розробленим групою Khronos, і призначений для низькорівневого програмування графіки та обчислень. Забезпечує потужний та ефективний доступ до графічного обладнання, особливо для важкодоступних завдань обчислень на GPU.

Переваги: Низькорівневий доступ: Vulkan надає більший контроль над апаратним обладнанням і дозволяє оптимізувати продуктивність для конкретної архітектури; відкритий стандарт Khronos Group дозволяє використовувати Vulkan на різних платформах; підтримка мультитредінгу робить Vulkan ефективним для сучасних багатоядерних процесорів.

Недоліки: Складність розробки - Vulkan є більш низькорівневим та вимагає більше коду для досягнення тих самих результатів, що може зробити розробку більш складною; велика кількість деталей - Завдяки низькорівневому характеру, розробка в Vulkan може вимагати докладного управління пам'яттю та іншими аспектами, що може призвести до складнощів.

Оскільки в даній роботі використовується графічний інтерфейс OpenGL, необхідно детальніше розглянути основні його історію створення, концепції та способи взаємодії [14].

У 1980-х роках розробка програмного забезпечення, яке могло б працювати з широким спектром графічного обладнання, була справжнім викликом.

Розробники програмного забезпечення писали власні інтерфейси та драйвери для кожного обладнання. Це коштувало дорого і призводило до множення зусиль.

На початку 1990-х компанія Silicon Graphics (SGI) була лідером у галузі 3D-графіки для робочих станцій. Їхній IRIS GL API[18][19] став галузевим стандартом, який використовувався ширше, ніж PHIGS, заснований на відкритих стандартах.[необхідне посилання] Це було пов'язано з тим, що IRIS GL вважався простішим у використанні[хто?] і тому, що він підтримував миттєвий режим рендерингу. На противагу цьому, PHIGS вважався складним у використанні та застарілим за функціональністю.

Конкуренти SGI (включаючи Sun Microsystems, Hewlett-Packard та IBM) також змогли вивести на ринок 3D обладнання, що підтримувало розширення стандарту PHIGS, що змусило SGI відкрити вихідний код версії IRIS GL як публічний стандарт під назвою OpenGL.

Однак у SGI було багато клієнтів, для яких перехід з IRIS GL на OpenGL вимагав би значних інвестицій. Крім того, IRIS GL мав функції API, які не мали відношення до 3D-графіки. Наприклад, він містив API для роботи з вікнами, клавіатурою та мишею, частково тому, що його було розроблено до появи X Window System та Sun's NeWS. А бібліотеки IRIS GL були непридатні для відкриття через проблеми з ліцензуванням та патентами[потрібне додаткове пояснення]. Ці фактори вимагали від SGI продовжувати підтримувати просунуті і запатентовані програмні API Iris Inventor і Iris Performer, в той час як ринкова підтримка OpenGL дозрівала.

Одним з обмежень IRIS GL було те, що він надавав доступ лише до функцій, які підтримувалися базовим обладнанням. Якщо графічне обладнання не підтримувало функцію нативно, то програма не могла її використовувати. OpenGL подолав цю проблему, надавши програмну реалізацію функцій, які не підтримуються апаратним забезпеченням, що дозволило програмам використовувати просунуту графіку на відносно малопотужних системах. OpenGL стандартизував доступ до апаратного забезпечення, переклав відповідальність за розробку програм апаратного інтерфейсу (драйверів пристроїв) на виробників

обладнання та делегував функції роботи з вікнами базовій операційній системі. Оскільки існує так багато різних видів графічного обладнання, примусити їх говорити однією мовою у такий спосіб мало неабиякий вплив, надавши розробникам програмного забезпечення платформу вищого рівня для розробки 3D-програмного забезпечення.

OpenGL (надалі "GL") займається лише рендерингом у фреймбуфер (і читанням значень, що зберігаються у цьому фреймбуфері). Не передбачено підтримки інших периферійних пристроїв інших периферійних пристроїв, які іноді асоціюються з графічним обладнанням, таких як миші та клавіатури. Програмістам доводиться покладатися на інші механізми для отримання користувацького вводу.

GL малює примітиви за допомогою декількох режимів і шейдерів, які можна вибрати. Кожен примітив є точкою, відрізком або багатокутником. Кожен режим можна змінювати незалежно, налаштування одного з них не впливає на налаштування інших (хоча багато режимів можуть взаємодіяти, визначаючи, що зрештою потрапить до фреймбуфера). Режими задаються, примітиви визначаються, а інші операції GL описуються шляхом надсилання команд у вигляді викликів функцій або процедур.

Примітиви визначаються групою з однієї або декількох вершин. Вершина визначає точку, кінцеву точку ребра або кут багатокутника, де перетинаються два ребра. З вершиною асоціюються такі дані, як координати положення, кольори, нормалі, координати текстури тощо. асоціюються з вершиною, і кожна вершина обробляється незалежно, у певному порядку і однаковим чином. Єдиний виняток з цього правила - якщо група вершин повинна бути обрізати так, щоб вказаний примітив вписувався у визначену область; у цьому випадку дані про вершини можуть бути змінені і створені нові вершини. Тип відсікання залежить від того, який примітив представляє група вершин.

Команди завжди обробляються у порядку їхнього надходження, хоча може бути невизначена затримка, перш ніж ефект команди буде реалізовано. Це означає, наприклад, що один примітив має бути намальований повністю, перш ніж будь-який наступний примітив зможе вплинути на буфер кадру. Це також означає, що

запити та операції зчитування пікселів повертають стан, що відповідає повному виконанню всіх попередньо викликаних команд GL, якщо явно не вказано інше. Загалом, вплив команди GL на режими GL або на буфер кадру повинен бути завершеним до того, як будь-яка наступна команда зможе мати такі ефекти.

У GL зв'язування даних відбувається за викликом. Це означає, що дані, передані команді, інтерпретуються при отриманні цієї команди. Навіть якщо команда вимагає вказівника на дані, ці дані інтерпретуються під час виклику, і будь-які подальші зміни даних не впливають на GL (якщо тільки той самий вказівник не використовується у наступній команді).

GL забезпечує прямий контроль над основними операціями 3D і 2D графіки. Це включає в себе визначення параметрів шейдерів, що визначаються додатком програми, що виконують операції трансформації, освітлення, текстурування та затінення, а також вбудовані функції, такі як згладжування та фільтрація текстур. Він не надає надає засобів для опису або моделювання складних геометричних об'єктів. Інший спосіб описати цю ситуацію - це сказати, що GL надає механізми для опису того, як складні геометричні об'єкти мають бути зображені, а не механізми для опису самих складних об'єктів. для опису самих складних об'єктів.

Модель інтерпретації команд GL - клієнт-сервер. Тобто, програма (клієнт) видає команди, і ці команди інтерпретуються та обробляються GL (сервером). Сервер може працювати на тому ж комп'ютері, що і клієнт, а може і не працювати. У цьому сенсі GL є "мережево-прозорим". Сервер може підтримувати декілька контекстів GL, кожен з яких є інкапсуляцією поточного стану GL. Клієнт може підключитися до будь-якого з цих контекстів. Виконання команд GL, коли програма не підключена до контексту, призводить до Undefined Behavior.

GL взаємодіє з двома класами фреймбуферів: віконними, наданими системою та створеними програмою. Існує щонайбільше один системний фреймбуфер вікна який називається фреймбуфером за замовчуванням. фреймбуфери, створені програмою, які називаються об'єктами фреймбуфера, можуть бути створені за бажанням. Ці два типи фреймбуферів відрізняються, насамперед, інтерфейсом для конфігурування та керування їх станом.

Вплив команд GL на стандартний фреймбуфер зрештою контролюється віконною системою, яка розподіляє ресурси фреймбуфера, визначає, до яких частин стандартного фреймбуфера GL може отримати доступ у будь-який момент часу, і повідомляє GL, як ці частини структуровані. Таким чином, немає команд GL для ініціалізації контексту GL або конфігурації фреймбуфера за замовчуванням. Аналогічно, відображення вмісту фреймбуфера на фізичному пристрої відображення (включно з перетворенням окремих значень фреймбуфера за допомогою таких методів, як гамма-корекція) не розглядається у GL.

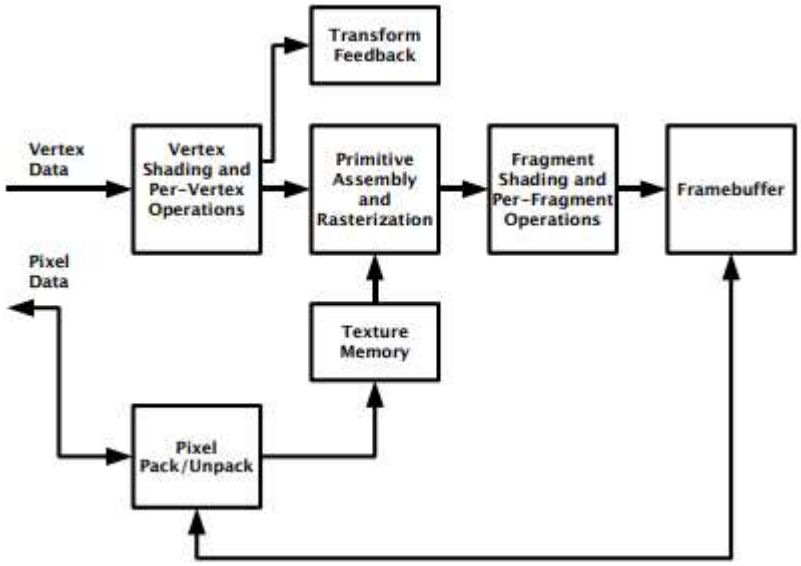

Рисунок 22 – Схема роботи GL

На Рисунку 22 показано принципову схему роботи GL. Команди вводяться у GL ліворуч. Деякі команди визначають геометричні об'єкти, які потрібно намалювати, а інші керують як об'єкти обробляються на різних етапах. Команди ефективно надсилаються через конвеєр обробки.

Перший етап оперує з геометричними примітивами, описаними вершинами: точками, відрізками ліній та багатокутниками. На цьому етапі вершини можуть трансформуватися і підсвічуватися, а потім збиратися в геометричні примітиви, які за бажанням можуть бути використані на наступному етапі, затінення геометрії, для створення нових примітивів. Отримані примітиви обрізаються до об'єму

перегляду для підготовки до наступного етапу - растеризації. Растеризатор створює серію адрес і значень фреймбуфера, використовуючи двовимірний опис точки, відрізка лінії або полігону. Кожен створений таким чином фрагмент передається на наступний етап, який виконує операції над окремими фрагментами, перш ніж вони остаточно змінять фреймбуфер. Ці операції включають умовне оновлення фреймбуфера на основі вхідних і попередньо збережених значень глибини (для ефекту буферизації глибини), змішування кольорів вхідних фрагментів зі збереженими кольорами, а також маскування та інші логічні операції над значеннями фрагментів.

Нарешті, значення можуть також зчитуватися з фреймбуфера або копіюватися з однієї частини фреймбуфера в іншу. Ці передачі можуть включати певний тип декодування або кодування.

Цей порядок розглядається лише як інструмент для опису ГЗ, а не як суворе правило реалізації ГЗ, і ми представляємо його лише як засіб для організації різних операцій ГЗ. Такі об'єкти, як криві поверхні, наприклад, можуть можуть бути перетворені перед перетворенням у полігони.

4.2 Імплементація запропонованого способу

Для реалізації алгоритму було обрано мову програмування C++, а в якості графічного інтерфейсу було обрано OpenGL. Дана мова програмування була обранана, оскільки більшість існуючих ігрових рушіїв було створено з її використанням. Це пов'язано з її високою швидкодією при правильному використанні, у порівнянні з іншими мовами. Графічний інтерфейс обирався за принципом швидкості розробки та поширеності у використанні. Перевагою і одночасно недоліком є її відносно високий рівень абстракціїї, у порівнянні з іншими популярними інтерфейсами – DirectX, Vulkan. Це значно облегшує процес розробки, але і зачасту, графічні додатки написані в використанням OpenGL будуть значно повільніші, ніж ті, що створені з використанням вищевказаних альтернатив.

Початок будь якого графічного додатку з використанням OpenGL – це ініціалізація інтерфейсу

Зазвичай, всі ці виклики є шаблонними та однакові для більшості програм. Наступним буде ініціалізація специфічних для цієї програми змінних. Спочатку такі виклики – компіляція програми шейдера, завантаження текстур, встановлення відповідностей між змінними в шейдері та C++ програмі, ініціалізація точки-джерела освітлення та включення вертикальної синхронізації.

Реалізуємо нормальне відображення за допомогою простору дотичних, щоб ми могли орієнтувати цю площину як завгодно, і нормальне відображення все одно працювало. Використовуючи раніше розглянуту математику, ми вручну обчислимо вектори дотичних і бітангентів цієї поверхні.

```
// positions
glm::vec3 pos1(-1.0f,  1.0f, 0.0f);
glm::vec3 pos2(-1.0f, -1.0f, 0.0f);
glm::vec3 pos3( 1.0f, -1.0f, 0.0f);
glm::vec3 pos4( 1.0f,  1.0f, 0.0f);
// texture coordinates
glm::vec2 uv1(0.0f, 1.0f);
glm::vec2 uv2(0.0f, 0.0f);
glm::vec2 uv3(1.0f, 0.0f);
glm::vec2 uv4(1.0f, 1.0f);
// normal vector
glm::vec3 nm(0.0f, 0.0f, 1.0f);
```

Припустимо, що площина побудована з наступних векторів (з 1, 2, 3 і 1, 3, 4 як двома трикутниками).

Спочатку обчислюємо ребра першого трикутника та дельта UV координати:

```
glm::vec3 edge1 = pos2 - pos1;
glm::vec3 edge2 = pos3 - pos1;
glm::vec2 deltaUV1 = uv2 - uv1;
glm::vec2 deltaUV2 = uv3 - uv1;
```

Маючи необхідні дані для обчислення тангенсів і бітангенсів, ми можемо перейти до виконання рівняння з попереднього розділу:

```
// calculate tangent/bitangent vectors of both triangles
glm::vec3 tangent1, bitangent1;
glm::vec3 tangent2, bitangent2;

float f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);

tangent1.x = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent1.y = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent1.z = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);

bitangent1.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent1.y = f * (-deltaUV2.x * edge1.y + deltaUV1.x * edge2.y);
bitangent1.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);
```

Тут ми спочатку попередньо обчислюємо дробову частину рівняння як f, а потім для кожного векторного компонента виконуємо відповідне матричне множення, помножене на f. Якщо порівняти цей код з остаточним рівнянням, то можна побачити, що це прямий переклад. Оскільки трикутник завжди є плоскою фігурою, нам потрібно обчислити лише одну пару дотична/бітангенс для кожного трикутника, оскільки вони будуть однаковими для кожної з вершин трикутника.

Результуючий вектор дотичної та бітангенса повинен мати значення (1,0,0) та (0,1,0) відповідно, що разом з нормаллю (0,0,1) утворює ортогональну матрицю TBN.

Тут ми спочатку переводимо всі вектори TBN у систему координат, в якій ми хочемо працювати, в даному випадку це світовий простір, оскільки ми перемножуємо їх з матрицею моделі. Потім ми створюємо власне матрицю TBN, безпосередньо надаючи конструктору mat3 відповідні вектори-стовпчики. Зауважте, що якщо ми хочемо бути дійсно точними, ми перемножимо вектори TBN на нормальну матрицю, оскільки нас цікавить лише орієнтація векторів.

Для того, щоб змусити normal mapping працювати, нам спочатку потрібно створити матрицю TBN у шейдері. Для цього ми передаємо раніше обчислені вектори дотичних та бітангентів до вершинного шейдера як атрибути вершин:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;
```

Потім у головній функції вершинного шейдера створюємо матрицю TBN:

```
mat3 normalMatrix = transpose(inverse(mat3(model)));
vec3 T = normalize(normalMatrix * aTangent);
vec3 N = normalize(normalMatrix * aNormal);
T = normalize(T - dot(T, N) * N);
vec3 B = cross(N, T);

mat3 TBN = transpose(mat3(T, B, N));
```

Технічно немає потреби у змінній бітангенса у вершинному шейдері. Всі три вектори TBN перпендикулярні один до одного, тому ми можемо обчислити бітангенс самостійно у вершинному шейдері, взявши перехресний добуток векторів **T** і **N**: ***vec3 B = cross(N, T)***;

Ми беремо матрицю TBN, яка трансформує будь-який вектор з дотичної у світовий простір, передаємо її шейдеру фрагментів і трансформуємо вибрану нормаль з простору дотичних у світовий простір за допомогою матриці TBN; тоді нормаль буде в тому ж просторі, що й інші змінні освітлення.

Вектор нормалі, який ми вибираємо з карти нормалей, виражений у дотичному просторі, тоді як інші вектори освітлення (світло і напрямок погляду) виражені у світовому просторі. Передаючи матрицю TBN фрагментному шейдеру, ми можемо помножити вибрану нормаль дотичного простору на цю матрицю TBN, щоб перетворити вектор нормалі в той самий опорний простір, що й інші вектори освітлення. Таким чином, всі розрахунки освітлення (зокрема, точковий добуток) мають сенс.

Надіслати матрицю TBN до фрагментного шейдера дуже просто:

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;
```

```
vs_out.TangentLightPos = TBN * lightPos;
vs_out.TangentViewPos  = TBN * viewPos;
vs_out.TangentFragPos  = TBN * vs_out.FragPos;
```

У фрагментному шейдері ми аналогічно беремо mat3 як вхідну змінну:

```
in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} fs_in;
```

Маючи цю матрицю TBN, ми можемо оновити код нормального мапування, включивши до нього перетворення дотичної до світового простору:

```
// obtain normal from normal map in range [0,1]
vec3 normal = texture(normalMap, fs_in.TexCoords).rgb;
// transform normal vector to range [-1,1]
normal = normalize(normal * 2.0 - 1.0);  // this normal is in tangent space
```

Оскільки результуюча нормаль тепер знаходиться у світовому просторі, немає необхідності змінювати код шейдерів інших фрагментів, оскільки код освітлення передбачає, що вектор нормалі знаходиться у світовому просторі. Використовуємо отриманий вектор нормалі для обрахування параметрів текстури:

```
// get diffuse color
vec3 color = texture(diffuseMap, fs_in.TexCoords).rgb;
// ambient
vec3 ambient = 0.1 * color;
// diffuse
vec3 lightDir = normalize(fs_in.TangentLightPos - fs_in.TangentFragPos);
float diff = max(dot(lightDir, normal), 0.0);
vec3 diffuse = diff * color;
// specular
vec3 viewDir = normalize(fs_in.TangentViewPos - fs_in.TangentFragPos);
vec3 reflectDir = reflect(-lightDir, normal);
vec3 halfwayDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0);

vec3 specular = vec3(0.2) * spec;
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

4.3 Результати роботи програми

Проєкт збирався використовуючи IDE Microsoft Visual Studio 2019, версія OpenGL – 3.3.0.

Для порівняння швидкодії роботи програм використовувався власноруч зроблений FPS лічильник. Виміри збиралися протягом 1 хвилини для обох програм. Оскільки ввімкнена вертикальна синхронізація, то максимально можливий FPS – 144. При виконанні програми з запропонованим алгоритмом, середнє значення – 143, для загальноприйнятого методу – 141.

Висновки до розділу 4

Опираючись на отримані результати роботи програми можна зробити висновок, що алгоритм у запропанованому способі виправдав очікування та зміг майже ідентично, відобразити тестовану текстуру.

Якщо порівнювати швидкодію двух програм, то вони однакові. Присутня різниця в 1 FPS, але це, скоріш за все, похибки в вимірюваннях або в певний час виконувались зі сторони операційної системи більш або менш затратні процеси.

Найбільш відчутна різниця – це економія пам’яті, оскільки алгоритм в розробленому способі не використовує додаткову структуру, розмір якої 559КБ.

ВИСНОВКИ

Аналізуючи всі розділи магістерської дисертації, можна зробити висновки, що в нинішній час існує безліч різноманітних методів візуалізації об’єктів. Всі вони застосовуються в різних цілях та при різних обставинах. Деякі були розроблені та представлені в останні роки, а певним – вже десятиріччя, але через те, що вони постійно оновлюються згідно сучасних тенденцій, вимог та задач, все же залишаються актуальними. В той же час, вони всі пов’язані між собою, як і в реалізації так і в використанні.

Перший розділ присвячений дослідженню різних способів та методів візуалізації абсолютно різних типів об’єктів. Розглянуто використовувані алгоритми для зображення освітлення, реалістичних предметів, скелетальної анімації та найпоширенішому методу оптимізації – ігнорування невидимих об’єктів. Цей розділ відображає важливість розуміння взаємодії між різними об’єктами задля створення якнайякіснішого продукту.

У другому розділі описані алгоритми для одного з найбільш ефективних методів оптимізації – зменшення кількості полігонів. Звернуто увагу на проблематику такої оптимізації та ймовірні результати. Висновком є те, що варто комбінувати наявні методи оптимізація задля найкращого результату.

У третьому розділі детально пояснено про сутність, необхідність та області використання вектору нормалі. Під час гри або спостереженню за анімацією і не спаде на думку, що одиничний вектор відіграє ключову роль у більшості видимих ефектів. Також приділяється увага математичній складовій, оскільки саме знання основ дає піґрунтя для подальших досліджень. Розглянуто теоретичне подання запропонованого способу візуалізації, який є похідним від Normal Mapping.

Четвертий розділ містить відображення роботи та порівняння двох способів з підкресленням переваг на недоліків обох сторін.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Prelude to GeForce RTX [Електронний ресурс] https://www.anandtech.com/show/13282/nvidia-turing-architecture-deep-dive/5
2. Essentials of Interactive Computer Graphics: Concepts and Implementation, 14.2.2 – Perspective Projection : A Viewing Frustrum https://books.google.com/books?id=PqT3RRVo4isC&pg=PA390
3. Graphics Processing Unit [Електронний ресурс] https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html
4. Compute Shader [Електронний ресурс] https://www.khronos.org/opengl/wiki/Compute_Shader
5. RTX Technology [Електронний ресурс] https://developer.nvidia.com/rtx/ray-tracing
6. Open Asset Import Library (assimp) [Електронний ресурс] https://github.com/assimp/assimp
7. Texture Groups [Електронний ресурс] https://docs.unrealengine.com/udk/Three/TextureStreaming.html
8. William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. - Decimation of triangle meshes. Computer Graphics (SIGGRAPH ’92 Proc.), 26(2):65–70, July 1992.
9. Marc Soucy and Denis Laurendeau. Multiresolution surface modeling based on hierarchical triangulation. Computer Vision and Image Understanding, 63(1):1–14, 1996.
10. Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, Modeling in Computer Graphics: Methods and Applications, pages 455–465, 1993
11. David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In SIGGRAPH 97 Proc., August 1997.
12. Дотична площина до поверхні [Електронний ресурс] https://ukrayinska.libretexts.org/

13. Nykamp, Duane. "Multiplying matrices and vectors". Math Insight. Вересень 2020.
14. OpenGL - The Industry's Foundation for High Performance Graphics - The Khronos Group. July 19, 2011. Retrieved March 18, 2021. https://www.khronos.org/opengl/
15. SGI – OpenGL Overview [Електронний ресурс] http://www.sgi.com/products/software/opengl/overview.html
16. What is GPU? https://aws.amazon.com/what-is/gpu
17. Skeletal Meshes https://docs.unrealengine.com/4.26/en-US/WorkingWithContent/Types/SkeletalMeshes/
18. Jason Gregory - Game Engine Architecture (2nd edition) – 2014
19. Kuldeep Singh - Linear Algebra Step by Step – Oxford – 2014
20. Johnson M. Hart - Windows System Programming 4th Edition - 2010 Pearson Education, Inc.