КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп’ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота на здобуття ступеня бакалавра

за спеціальністю 121 Інженерія програмного забезпечення
на тему:

РОЗРОБКА НИЗЬКОРІВНЕВОГО ГРАФІЧНОГО РЕНДЕРЕРУ ДЛЯ ВІЗУАЛІЗАЦІЇ 3D-СЦЕН З ВИКОРИСТАННЯМ DIRECTX 12

Виконав студент 4-го курсу
Гліб НАВКА

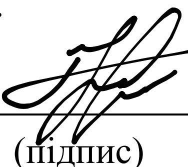
(підпис)

Науковий керівник:
асистент, кандидат фізико-математичних наук
Костянтин ЖЕРЕБ

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент



(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем

«29» травня 2023 р.,
протокол № 11

Завідувач кафедри
Олександр ПРОВОТАР

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 65 сторінок, 17 ілюстрацій, 26 використаних джерел.

3D-ГРАФІКА, 3D-МОДЕЛЬ, C++, DIRECTX 12, GPU, HLSL, ГРАФІЧНИЙ КОНВЕЄР, ГРАФІЧНИЙ РЕНДЕРЕР, НИЗЬКОРІВНЕВЕ ГРАФІЧНЕ API, ШЕЙДЕР.

Об'єктом розробки є графічний рендерер для візуалізації 3D-сцен з використанням DirectX 12. Предметом є розробка графічного рендереру.

Метою роботи є проведення детального дослідження роботи GPU, аналіз архітектури сучасних низькорівневих графічних API на прикладі DirectX 12 та розробка власного рендереру з використанням передових технології та методик рендерінгу.

Методи розроблення: розробка зверху-вниз (top-down development), об’єктно-орієнтоване програмування. Інструменти розроблення: мова програмування C++, мова програмування шейдерів HLSL, низькорівневе графічне API DirectX 12, Windows API, інтегроване середовище розробки Visual Studio 2022 Community, графічний debugger RenderDoc, конфігурація збірки Premake.

Результати роботи: розроблено власний рендерер, а також досліджено структуру, компоненти та функціональні можливості графічного API DirectX 12. В процесі розробки було створено високорівневу абстракцію для функціоналу DirectX 12, завдяки чому створено рендерер, який здатний відображати 3D-моделі у форматі glTF. Крім того, реалізовано підхід відкладеного рендерінгу разом із фізично заснованим рендерінгом. За допомогою цього оптимізовано час рендерінгу та досягнуто більш реалістичного візуального ефекту. Також додано можливість відображати довкілля, що дозволяє створювати графічні сцени з ефектом присутності.

Сфера застосування: робота має широкий спектр застосувань у різних галузях комп'ютерної індустрії: розробка комп'ютерних ігор, візуалізація 3D-моделей, віртуальна реальність тощо.

ЗМІСТ

СКОРОЧЕНННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 5
ВСТУП..... 6
РОЗДІЛ 1 ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ 8
1.1 Типи пам'яті та складові GPU 8
1.2 DirectX 9
1.3 DXGI. Графічна інфраструктура DirectX 9
1.3 DirectX 3D. DirectX 12 10
1.3.1 Синхронізація CPU та GPU 10
1.3.2 GPU-ресурси, перегляди ресурсів і дескриптори 12
1.3.3 GPU-конвеєри 13
РОЗДІЛ 2 ПОНЯТТЯ 3D-ГРАФІКИ ТА МЕТОДИ РЕНДЕРІНГУ 16
2.1 3D-модель..... 16
2.2 Простори 19
2.3 Підходи до розробки рендереру 21
РОЗДІЛ 3 РОЗРОБКА ВЛАСНОГО РЕНДЕРЕРУ 23
3.1 Структура проєкту 23
3.1.1 Бібліотека залежностей..... 23
3.1.2 Рендер-бібліотека 24
3.1.3 Пісочниця..... 25
3.2 Низькорівневий рендерер 26
3.2.1 Графічний Backend..... 26
3.2.2 Графічний девайс 28
3.2.3 Черги команд для кожного типу GPU рушія..... 29
3.2.4 Ланцюг обміну. Синхронізація CPU та GPU 30
3.2.5 Робота з ресурсами..... 32
3.2.6 Менеджер дескрипторів 33
3.2.7 Єдиний кореневий підпис. Динамічні ресурси 34

3.2.8 Менеджер шейдерів 37
3.3 Модуль роботи з сутностями 39
3.3.1 Структури 3D-моделі 39
3.3.2 Завантаження 3D-моделі з файлу glTF 42
3.3.3 Робота з просторами 43
3.4 Високорівневий рендерер 47
3.4.1 Прохід геометрії 47
3.4.2 Прохід освітлення 50
3.4.3 Прохід довкілля 54
3.5 Результати рендереру 58
ВИСНОВКИ 61
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНННЯ 63

СКОРОЧЕНННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API – Application Programming Interface, інтерфейс програмних застосунків;

CPU – Central Processing Unit, центральнний процесор;

DirectX – це набір API, для розробки високопродуктивних ігрових мультимедійних додатків;

DXC – DirectX Shader Compiler, компілятор шейдерів DirectX;

ECS – Entity Component System;

FPS – Frame per Second, кількість кадрів в секунду;

GPU – Graphics Processing Unit, ядро відеокарти;

HLSL – High Level Shader Language, мова програмування шейдерів;

IDE – Integrated development environment, інтегроване середовище розробки;

PBR – Physically based rendering, фізично заснований рендерінг;

SDK – Software Development Kit, набір для розробки програмного забезпечення;

Меш – Mesh, набір вершин та багатокутників, що визначають форму тривимірного об'єкта;

Рендерер – Renderer, програма, яка перетворює 3D-сцени в 2D-зображення;

Рендерінг – Rendering, процес перетворення 3D-сцени в 2D-зображення;

Семплер – Sampler, компонент для отримання даних з текстур;

Тексель – Texel, одиниця виміру для текстур;

Шейдер – Shader, графічні програми для обробки та візуалізації графіки та ефектів.

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. У галузі графічного рендерінгу та візуалізації 3D-сцен було здійснено значну кількість досліджень та розробок графічних рендерерів, які працюють з різними графічними API. Вони можуть зосереджуватися на досягненні високої продуктивності, оптимізації алгоритмів рендерінгу та керуванні графічними ресурсами, вивчати нові техніки та інші аспекти, що покращують якість візуалізації 3D-сцен.

Актуальність роботи та підстави для її виконання. Існує декілька основних аспектів, чому дана тема актуальна на сьогодні:

1. Зростання популярності розробки ігрового контенту та віртуальної реальності, що створює потребу в розробці більш потужних та ефективних графічних рендерерів, здатних досягати високої якості зображення та реалістичних ефектів.
2. Постійний розвиток ринку комп'ютерної техніки та графічних пристроїв і як наслідок необхідність у розробці графічних рендерерів, що використовують останні низькорівневі графічні API та відповідають вимогам сучасної апаратної бази.
3. Необхідність оптимізації та ефективного використання ресурсів, забезпечуючи високу продуктивність, зниження накладних витрат та покращення фінального зображення.
4. Важливість інновацій та конкурентної переваги на ринку.

Мета й завдання роботи. Метою роботи є розробка графічного рендереру з використанням DirectX 12, а також дослідження сучасної архітектури та методів розробки рендерерів за допомогою новітніх графічних API.

Для досягнення цієї мети поставлено такі завдання:

- дослідити принцип роботи GPU та її основні компоненти;
- проаналізувати етапи графічного конвеєру та архітектуру DirectX 12;

– створити архітектуру низькорівневого графічного рендерера, який використовує можливості DirectX 12 та оптимально використовує апаратні ресурси;
– розробити алгоритми та методи для реалістичної візуалізації 3D-сцен, зокрема освітлення, текстурування та інші ефекти.

Об’єкт, методи й засоби розробки. Об'єктом розробки є графічний рендерер для візуалізації 3D-сцен.

Рендерер розроблено на мові програмування C++, що дає змогу досягти високої продуктивності. Також останні стандарти мови надають розробникам розширену стандартну бібліотеку та інші можливості.

В якості сторонніх бібліотек використовуються:

– *DirectX 12 DDSTextureLoader* для завантаження текстур формату .dds;
– *entt* для використання ECS підходу;
– *fmt* для швидкого та зручного форматування тексту;
– *ImGui* для роботи з графічним інтерфейсом;
– *magic enum* для роботи з переліками (enum);
– *tinygltf* для завантаження 3D-моделі формату glTF.

Шейдери написані на мові HLSL, а для їх компіляції використовується DXC, щоб мати доступ до останніх функцій шейдерних моделей. В якості IDE обрано Visual Studio 2022 Community. Для простого та зрозумілого опису проєктів використовується Premake. Також було використано RenderDoc для пошуку проблем під час рендерінгу кадру.

Можливі сфери застосування. Здобуті навички при реалізації даної роботи мають широкий спектр застосувань у різних галузях комп'ютерної індустрії. Особливо корисними вони можуть бути в таких областях: розробка комп'ютерних ігор, візуалізація 3D-моделей, віртуальна реальність тощо.

РОЗДІЛ 1 ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

1.1 Типи пам'яті та складові GPU

Типи пам'яті, які використовуються при розробці графічних програм:

1. *Оперативна пам'ять (RAM)* використовується для збереження активних даних, з якими працює CPU.
2. *Спільна пам'ять (Shared memory)* використовується для спільного використання між CPU та GPU.
3. *Відеопам'ять (Video memory, VRAM)* використовується в GPU та призначена для елементів, необхідних для обробки графіки.

Спільна пам'ять зазвичай має меншу швидкодію доступу порівняно з відеопам'яттю для GPU.

При розробці графічних програм використовуються такі складові відеокарти:

1. *Графічний процесор (Graphics Processing Unit, GPU)* відповідає за обробку графіки та відображення зображень на моніторі. Він має велику кількість ядер, що дозволяє швидко виконувати паралельні обчислення.
2. *Відеопам'ять (Video Memory, VRAM)*.
3. *Растеризатор (Rasterizer)* перетворює геометричні примітиви на пікселі, що можуть бути відображені на екрані, а також відповідає за відсікання та інтерполяцію.
4. *Шейдерний процесор (Shader Processor)* відповідає за виконання шейдерів і дозволяє виконувати складні обчислення для створення графічних ефектів.

Типи GPU-рушіїв:

1. *Рушій копіювання (Copy Engine)* відповідає за швидкий обмін даними між спільною пам’яттю та відеопам'яттю.
2. *Рушій обчислення (Compute Engine)* призначений для виконання обчислювальних завдань.

3. *3D-прискорювач (3D-Accelerator)* відповідає за обробку та відтворення тривимірної графіки.

1.2 DirectX

DirectX є набором API, розроблених компанією Microsoft, які використовуються для розробки та запуску високопродуктивних ігор та мультимедійних додатків на платформі Windows та Xbox.

Компоненти DirectX, які дозволяють створювати потужні графічні програми:

1. *DXGI (DirectX Graphics Infrastructure)* відповідає за керування відеокартами та відображення графіки на екрані.
2. *DirectX3D* включає набір функцій для створення складних 3D-сцен, відображення текстур, управління освітленням тощо.

Windows дозволяє створювати найпотужніші графічні програми порівняно з іншими платформами (Linux, MacOS). Крім DirectX, існує Vulkan, але він розроблений для підтримки багатьох платформ, тому DirectX забезпечує найкращу продуктивність та оптимальну роботу на платформі Windows.

1.3 DXGI. Графічна інфраструктура DirectX

Абстракції, за допомогою яких можна створювати графічні програми:

1. *Фабрика (factory)* використовується для створення інших об'єктів DXGI, запит інформації про доступні графічні адаптери та функції, а також керування поверхнями.
2. *Адаптер (adapter)* використовується для взаємодії з GPU в системі та при створенні графічного девайсу, оскільки саме від адаптера залежить підтримка певних графічних можливостей відеокарти.
3. *Ланцюг обміну (swap hain)* використовується для керування буферами зображення (двома або більше) та обміну ними між програмою та вікном.

– *задній буфер (back buffer)* містить кадр, який буде відображено користувачу;
– *передній буфер (front buffer)* відображає зображення із заднього буферу після завершення обробки кадру.

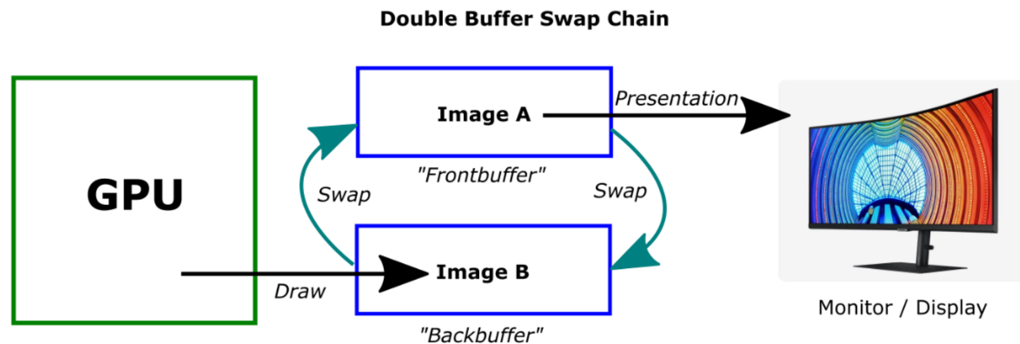


Рисунок 1.1 – Процедура відображення буферів ланцюга обміну

1.3 DirectX 3D. DirectX 12

DirectX 12 (DirectX3D 12) – це остання версія DirectX 3D API. Основною метою DirectX 12 було поліпшення продуктивності та ефективності графічних програм.

Особливості DirectX 12:

– низькорівневий доступ;
– багатопоточна обробка графіки;
– зменшення накладних витрат;
– покращена підтримка нових функцій апаратного забезпечення.

1.3.1 Синхронізація CPU та GPU

Кожен з GPU-рушіїв може виконувати команди паралельно, тому DirectX 12 забезпечує доступ до 3D-прискорювача, рушія обчислень і рушія копіювання за допомогою черги команд (command queue) та списків команд (command list).

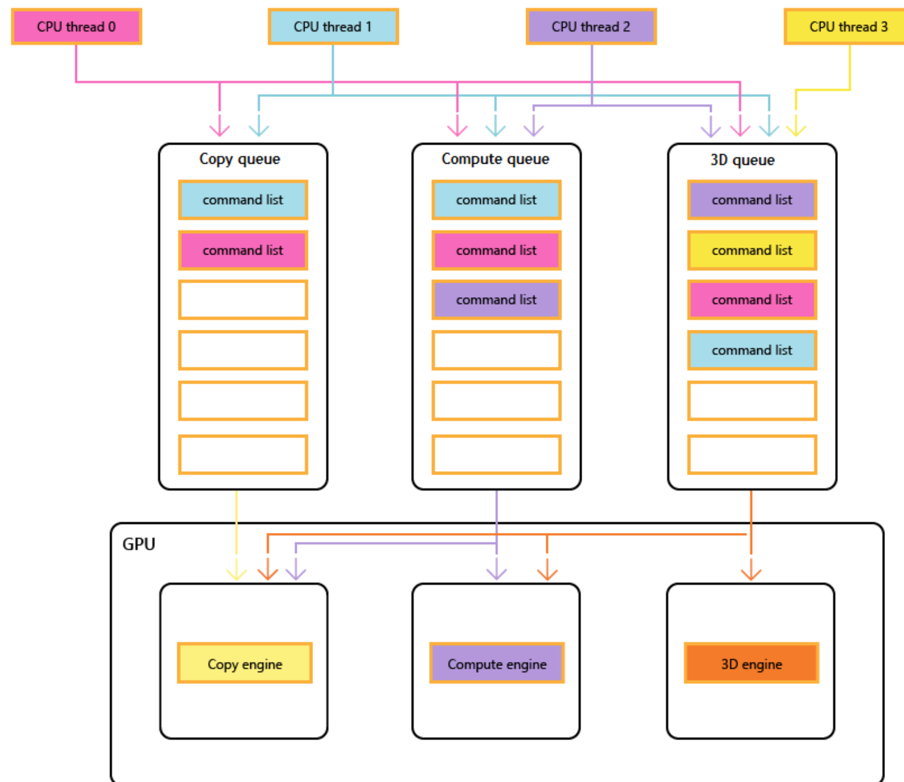


Рисунок 1.2 – Паралельний робочий процес з кількома GPU-рушіями

Список команд (command list) містить команди для виконання графічних, обчислювальних та копіювальних завдань. Список команд це як std::vector, не він займається алокаціями пам’яті для команд. Цю роботу виконує алокатор команд.

Алокатор команд (command allocator) використовується для виділення та керування пам’яті для команд. Алокатор команд може бути пов’язаним із конкретним типом списку команд. Створювання алокатора асоціюється з чергою команд.

Черга команд (command queue) використовується для надсилання командних списків в порядку їх надходження на виконання на GPU. Також використовується для очікування завершення виконання команд за допомогою бар’єрів синхронізації.

Бар’єр синхронізації (Fence) –використовується для синхронізації декількох GPU-рушіїв, за допомогою значення UINT64. Це значення може бути отримано зі сторони CPU, а оновлювати значення можна як зі сторони CPU, так і зі сторони GPU.

1.3.2 GPU-ресурси, перегляди ресурсів і дескриптори

Ресурс (GPU-ресурс) – це область у пам’яті, до якої може ефективно отримати доступ конвеєр DirectX 3D. Типи ресурсів:

1. *Буфери* – це набори повністю типізованих даних.
2. *Текстури* – це структуровані набори даних для зберігання текселів. На відміну від буферів, за текстурою можна робити вибірку, використовуючи семплери.

Перегляд ресурсу – це термін, використовується в значенні «дані в необхідному форматі» та потрібний для звернення до даних ресурсу з конвеєра. Перегляд ресурсу надає загальну модель доступу до ресурсу. Типи переглядів ресурсу:

– перегляд константного буфера (Constant Buffer View, CBV);
– перегляд ресурсу шейдера (Shader resource view, SRV);
– перегляд невпорядкованого доступу (Unordered Access view, UAV);
– перегляд рендер-цілі (Render Target View, RTV);
– перегляд глибини та трафатеру (Depth Stencil View, DSV);
– семплер (Sampler).

Дескриптор (Descriptor) – це невеликий блок даних, де зберігається перегляд ресурсу, що описує об’єкт для конвеєра. Основний спосіб використання дескрипторів – розміщення їх у купах дескрипторів (безперервний набір дескрипторів).

Ідентифікатор дескриптора (Descriptor Handle) – це унікальна адреса дескриптора. Це як покажчик, але непрозорий, оскільки його реалізація залежить від апаратного забезпечення. Типи ідентифікаторів дескриптора:

– *CPU ідентифікатор дескриптора* призначений для негайного використання;
– *GPU ідентифікатор декспиптора* використовуються під час роботи графічного конвеєра.

1.3.3 GPU-конвеєри

GPU-ресурси не прив'язані безпосередньо до конвеєра, натомість на них посилаються через дескриптори (як це було описано вище).

Кореневий підпис (root signature) – це одна з концепцій прив'язки дескрипторів до конвеєра, яка використовується шейдерами для визначення ресурсів, до яких їм потрібен доступ.

Кореневий підпис зберігає такі типи кореневих аргументів*:*

– кореневі дескрипторні таблиці (root descriptor tables);
– кореневі константи (root constants);
– кореневий дескриптор (root descriptor).

Також кореневий підпис може містити статичні (вбудовані) семплери.

Але такий підхід має недолік: для кожного об'єкту стану конвеєра повинен бути створений окремий кореневий підпис, який буде підходити під кожен шейдер в поточному конвеєрному стані, що іноді стає доволі незручно.

Шейдерна модель 6.6 представляє можливість звертатися до GPU-ресурсів шляхом прямого індексування дескрипторну купу, що позбавляє від створення кореневого підпису.

Шейдерна модель – це певна версія програмних моделей для реалізації шейдерів, яка визначає набір функцій, операторів та можливостей, які можуть використовуватись у шейдерах.

GPU-конвеєр (GPU pipeline) використовується в графічних процесорах для виконання багатьох обчислювальних завдань паралельно, розбиваючи обчислювальні задачі на багато менших кроків. Об'єкт стану конвеєра (pipeline state object, PSO) відповідає значній частині стану GPU. Цей стан включає всі шейдери та певні об'єкти фіксованого стану функції.

Існує два типи GPU конвеєрів:

1. *Графічний конвеєр (graphics pipeline)* відповідає за те, що дані переходять від входу до виходу через кожен із конфігурованих або програмованих етапів. Етапи графічного конвеєра, які можна налаштувати за допомогою DirectX 12:

– асемблер введення (input assembler);
– вершинний шейдер (vertex shader);
– тесселяція (tessellation): шейдер оболонки (hull shader), тесселятор (tessellator) та доменний шейдер (domain shader);
– геометричний шейдер (geometry shader);
– растеризатор (rasterizer);
– віксельний шейдер (pixel shader);
– об'єднання виведення (output merger, om): стан глибини та трафарету (depthstecil state) та стан змішуванння (blend state).

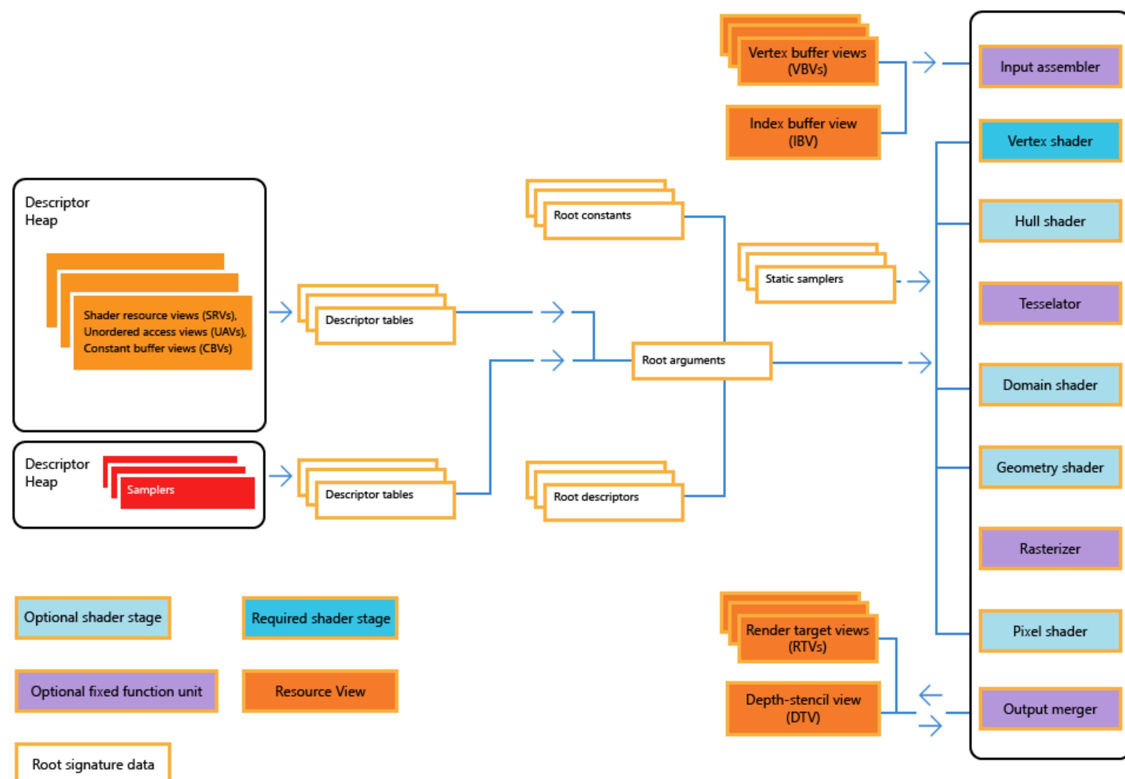


Рисунок 1.3 – Графічний конвеєр DirectX 12

2. *Обчислювальний конвеєр (compute pipeline)* призначений для обробки обчислень, які можна виконувати паралельно з графічним конвеєром. Є лише один етап обчислювального конвеєра – це етап програмованого обчислювального шейдера (compute shader), який забезпечує високошвидкісні обчислення загального призначення (general purpose) та використовує переваги великої кількості паралельних процесорів у GPU.

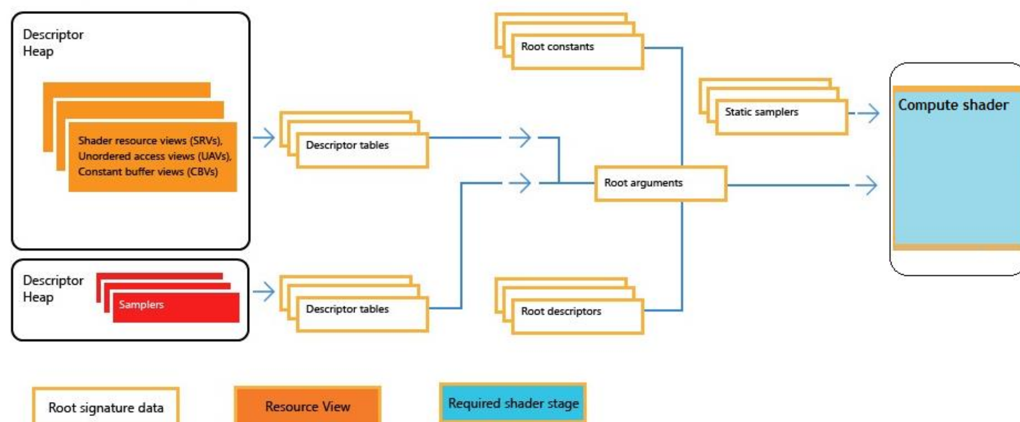


Рисунок 1.4 – Обчислювальний конвеєр DirectX 12

РОЗДІЛ 2 ПОНЯТТЯ 3D-ГРАФІКИ ТА МЕТОДИ РЕНДЕРІНГУ

2.1 3D-модель

Меш (в 3D-графіці) використовується для опису геометричної форми об'єкта. Меш складається зі з'єднаних трикутників, які визначають форму об'єкта та його поверхневу структуру. Меш можна уявити як скелет, на якій будується 3D-модель.

Кожна вершина має свої координати у тривимірному просторі (x, y, z), може мати нормаль (вектор, який перпендикулярний площині) та текстурні координати (це локація, за якої можна звернутись до текстури).

Зберігати меш в програмі можна як список вершин та індексів. У цьому підході список вершин містить усі унікальні вершини, що входять до складу меша. Кожна вершина має свій унікальний індекс. Натомість, список індексів визначає трикутники, використовуючи індекси вершин. За допомогою списків вершин та індексів можна ефективно зберігати меш, особливо якщо у геометрії є багато повторюваних вершин.

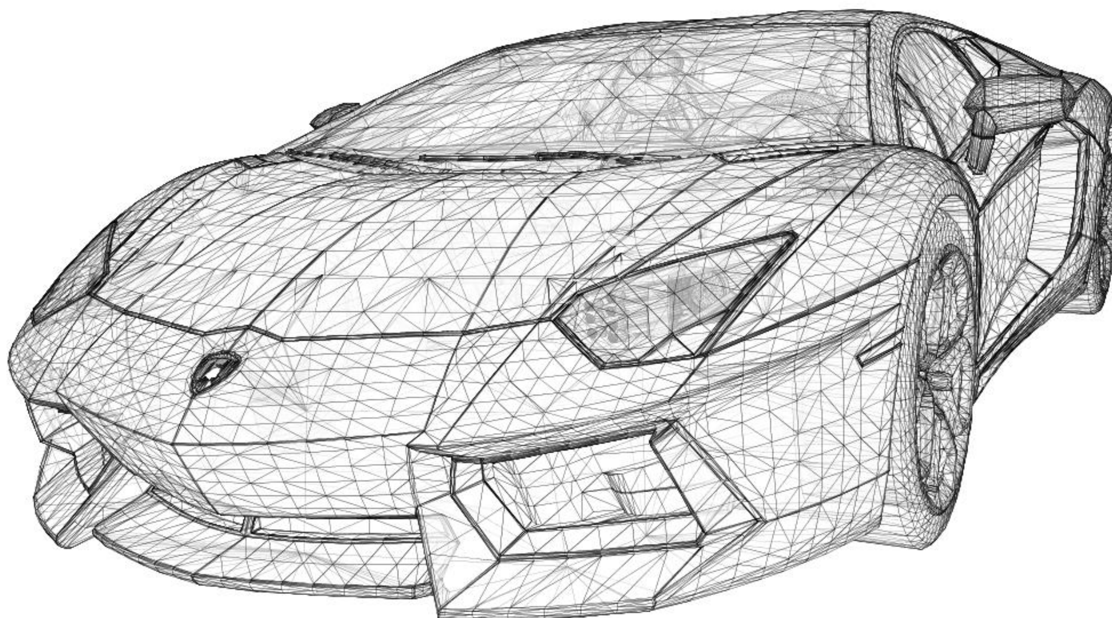


Рисунок 2.1 – Приклад меша: автомобіль

Матеріал (в 3D-графіці) – атрибут (властивість) поверхні об'єкта, який визначає його зовнішній вигляд і взаємодію з освітленням. Існує декілька популярних підходів, які визначають, як саме матеріал буде взаємодіяти зі світлом:

1. *Підхід Блінна-Фонга (Blinn-Phong)* є одним з простих підходів для симуляції освітлення поверхонь (тобто є апроксимацією). Підхід використовує для розрахунку фонове (ambient), дифузне (diffuse) та блискуче (specular) освітлення.

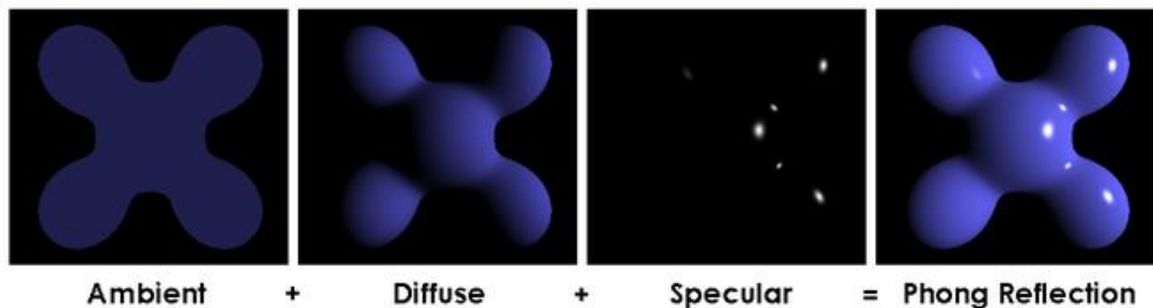


Рисунок 2.2 – Приклад підходу Блінна-Фонга

2. *Фізично заснований рендерінг (Physically Based Rendering, PBR)* спрямований на створення більш фізично вірних та реалістичних зображень. Основні принципи PBR базуються на законах фізики та взаємодії світла з матеріалами. Він спирається на модель змішаного матеріалу (microfacet model). Компоненти PBR:

– альбедо (albedo, base color);
– грубість (roughness);
– металічність (metallic);
– карта нормалей (normal map);
– BRDF (bidirectional reflectance distribution function).

PBR є кращим підходом порівняно з підходом Блінна-Фонга, оскільки він забезпечує більш реалістичне відображення матеріалів, дозволяє досягти більшої спрощеності та однорідності в роботі з матеріалами та освітленням між різними програмами та ігровими рушіями.

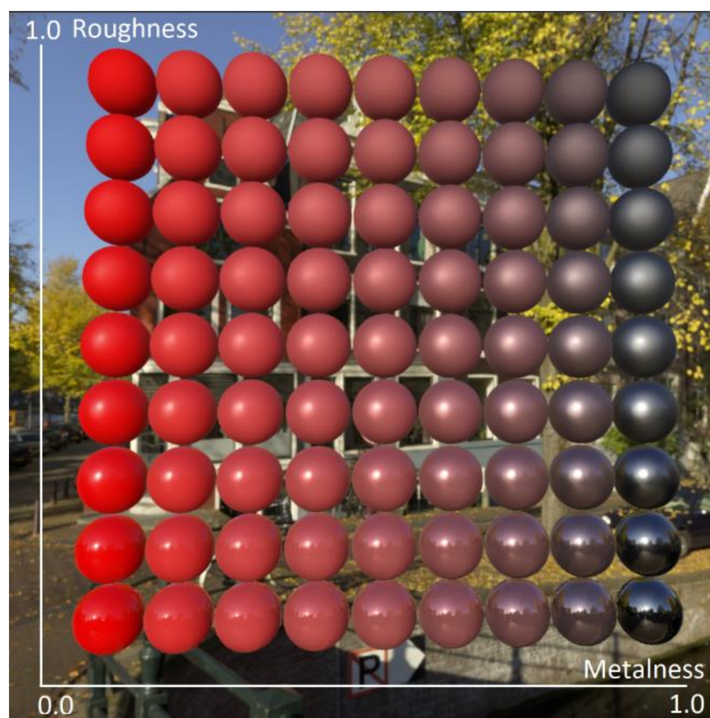


Рисунок 2.3 – Приклад фізично заснованого рендерінгу

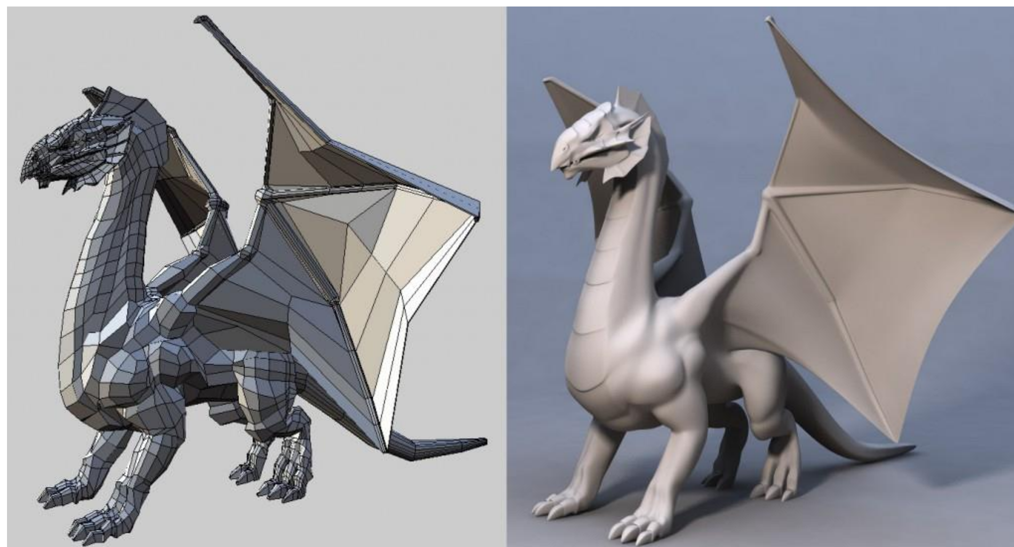


Рисунок 2.4 – Порівняння меша та 3D-моделі

glTF (Graphics Library Transmission Format) призначений для спрощення обміну 3D-моделей та сцен між різними програмами та пристроями.

glTF використовує компактний формат JSON або бінарний формат (з розширенням .glb) для зберігання геометрії моделей, текстур, матеріалів та інших

параметрів. Він підтримує стандартні функції, такі як текстури, нормалі, кольори, альфа-канали, скелетна анімація тощо.

glTF складається з наступних компонентів:

– файл .gltf – це головний файл, що містить опис сцени, об'єктів, матеріалів та анімації в форматі JSON;
– бінарні буфери .bin містять геометрію, текстури та інші дані, які можуть бути компактно збережені в бінарному форматі;
– текстури .jpg та .png – це зображення, які використовуються для текстурування об'єктів у сцені.

2.2 Простори

DirectX вимагає, щоб всі вершини, які повинні бути відображеними на екрані, мали *нормалізовані координати пристрою (Normalize Device Coordinates, NDC)* після кожного запуску вершинного шейдера. Це означає, що координати x, y і z кожної вершини повинні знаходитися в діапазоні від -1.0 до 1.0 та 0.0 до 1.0 відповідно. Якщо координати виходять за цей діапазон, то вони не будуть видимими.

Цей процес виконується поетапно, переходячи з одного простору в інший:

1. *Локальний простір (local space)* використовується для визначення положення, повороту та масштабу конкретного об'єкта відносно його власної системи координат.

2. *Простір світу (world space)* містить координати всіх вершин об'єкта відносно сцени. Для такого перетворення треба задіяти *світову матрицю (world matrix)* – матрицю перетворення, яка обертає, масштабує та переміщує об'єкт, щоб розмістити його у світі у відповідному місці та з відповідною орієнтацією.

3. *Простір огляду (view space)* – це простір камери, який представляє собою координатну систему, яка попадає в поле зору камери.

4. *Простір відсікання (clip space)* містить вершини, які будуть видимі на екрані та приймає дві різні форми.

Матриця проекції (projection matrix) переводить координати в нормалізований вигляд. Типи:

a. *Ортографічна проекція (orthogrphic projection)* визначає паралелепіпед. Координати кожної вершини відображаються без будь-якого поділу в перспективі. Через це, об’єкти, розташовані далі, не здаються меншими.

b. *Перспективна проекція (perspective projection)* визначає форму відсіченої піраміди (frustum). Для кожної вершини виконується *перспективний поділ* – це те, що перетворює 4D-координати простору відсікання в 3D-нормалізовані координати пристрою. Об’єкти, які розташовані далі, здаються меншими (як і в реальному житті).

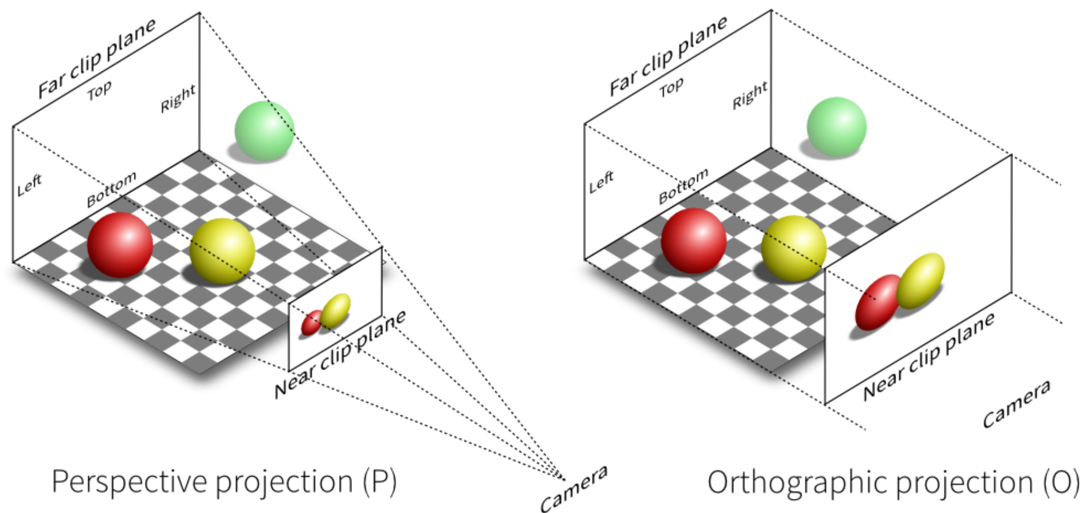


Рисунок 2.5 – Приклад різниці об'єма при перспективній та ортографічній проекції

5. *Простір екрану (screen space)* містить нормалізовані координати, які були перетворені за допомогою вікна перегляду (viewport).

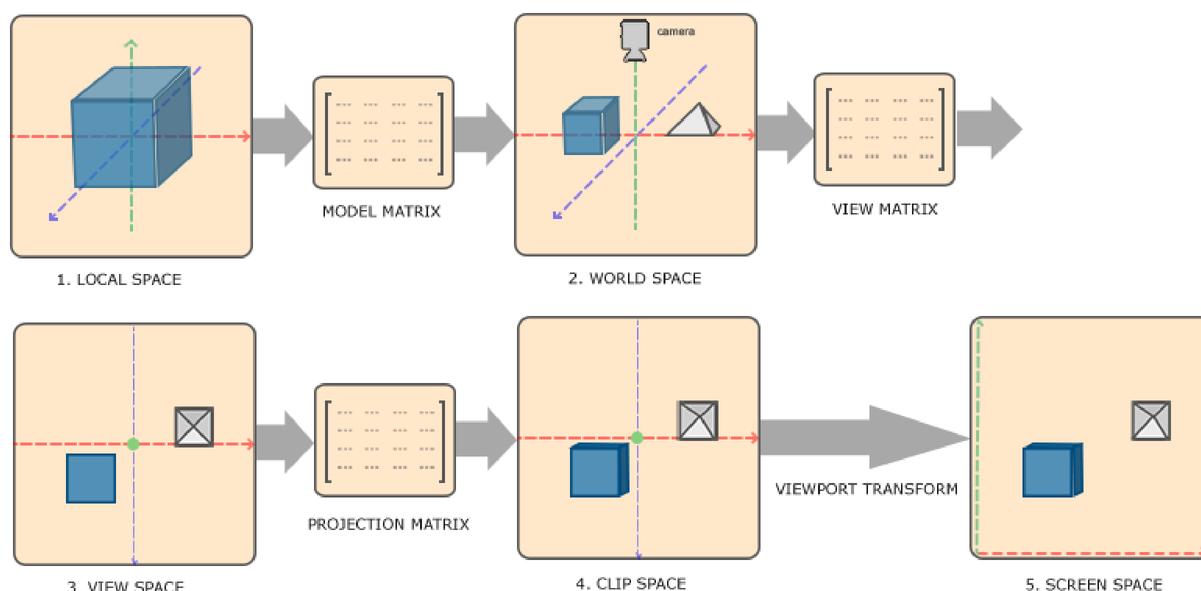


Рисунок 2.6 – Повний процес перетворення з локального простору у простір екрану

2.3 Підходи до розробки рендереру

Для того, щоб намалювати об’єкт потрібно: геометрія об’єкту, матеріали об’єкту та освітлення. Але маючи ці 3 компоненти, можна по різному підійти до процесу рендерінгу. Існує декілька основних підходів рендерінгу об'єктів:

1. *Прямий рендерінг (forward rendering)*: кожен об'єкт сцени обробляється окремо від інших.

Перевагами є простота та прямолінійность (добре підходить для сцен з невеликою кількістю об'єктів та джерел освітлення), а також можливість точно керувати процесом рендерінгу. Недоліком є його обробка світла: у випадку багатьох джерел освітлення кількість розрахунків зростає значно, що негативно впливає на продуктивність.

2. *Відкладений рендерінг (deferred rendering)* використовує буфери, щоб зберегти проміжну інформацію про сцену для подальшого оброблення.

Перевагою є можливість працювати з великою кількістю джерел освітлення при малому впливі на продуктивність. Недоліками є необхідність більшої кількості пам'яті

для зберігання буферу геометрії та він може бути менш ефективним у випадку сцен з прозорими об'єктами.

Сцена розбивається на два основних кроки:

a. *Геометричний прохід (geometry pass)*: для кожного пікселя зберігається інформація про необхідні властивості об'єктів у спеціальний буфер геометрії (GBuffer, Geometry buffer). Цей буфер містить всю необхідну інформацію про сцену для подальшого освітлення. Gbuffer, як правило, складається з декількох текстур.
b. *Прохід освітлення (lighting pass)*: кожен піксель обчислює освітлення на підставі інформації, збереженої в буфер геометрії. Результати обчислень записуються в кінцевий буфер зображення.

3. *Прямий+ рендерінг (forward+ rendering)* є модифікацією прямого рендерінгу, яка використовує кластеризацію для поліпшення продуктивності при обробці багатьох джерел освітлення.

Сцена розбивається на кластери – це група пікселів, які належать до одного регіону простору. Кожен кластер має свій власний список видимих джерел світла, які мають потенційний вплив на пікселі в цьому кластері.

Перевагами є забезпечення більшої паралелізації та ефективності завдяки кластеризації. Недоліком є те, що він підходить переважно для сцен з великою кількістю джерел світла, оскільки в інших випадках перевага його використання може бути менш виразною.

РОЗДІЛ 3 РОЗРОБКА ВЛАСНОГО РЕНДЕРЕРУ

3.1 Структура проєкту

Розроблений рендерер складається з трьох проєктів: *бібліотека залежностей*, *рендер-бібліотека* та *проєкт-пісочниця*. Для зручності збірки проєкту використовується Premake [18]. Це дозволяє збірку проєкту для Visual Studio здійснити шляхом створення одного Lua-файлу, що містить опис рішення. Такий підхід спрощує управління залежностями та налаштування проєкту, дозволяючи швидко збирати та налагоджувати рендерер.

3.1.1 Бібліотека залежностей

Бібліотека залежностей є колекцією сторонніх бібліотек, які були використані під час розробки рендер-бібліотеки та проєкту-пісочниці. У даному проєкті не використовується система збірок, що надається сторонніми бібліотеками, оскільки підтримка кількох систем збірок одночасно може бути складною. Тому всі сторонні бібліотеки компілюються з наявних файлів у єдиному великому проєкті. Опис проєкту має наступний вигляд:

```
project "third_party"
    kind "StaticLib"
    language "C++"
    cppdialect "C++20"
    location "../source/third_party"

    targetname "third_party_%{cfg.buildcfg}"

    targetdir "../lib"
    objdir "../build/third_party/%{cfg.buildcfg}"

    files {
        -- directx
        "../source/third_party/directx/**.h",
        "../source/third_party/directx/**.cpp",

        -- entt
        "../source/third_party/entt/**.h",
        "../source/third_party/entt/**.hpp",
        "../source/third_party/entt/**.cpp",
```

```
        --fmt
        "../source/third_party/fmt/**.h",
        "../source/third_party/fmt/**.cc",

        -- imgui
        "../source/third_party/imgui/**.h",
        "../source/third_party/imgui/**.cpp",

        -- magic_enum
        "../source/third_party/magic_enum/*.hpp",

        -- tinygltf
        "../source/third_party/tinygltf/*.h",
        "../source/third_party/tinygltf/*.hpp",
        "../source/third_party/tinygltf/*.cc",
    }
```

3.1.2 Рендер-бібліотека

Рендер-бібліотека включає компоненти, які дозволяють вже на даному етапі створити високорівневий рендерер. Ця бібліотека включає готову реалізацію низькорівневого рендерера з використанням DirectX 12, включає створення вікна на платформі Windows, файли конфігурації графіки, допоміжні компоненти для побудови архітектури та інші утиліти. В рамках цієї бібліотеки використовується попередньо скомпільований файл заголовка (precompiled header file), що допомагає зменшити час компіляції проєкту. Опис проєкту має наступний вигляд:

```
project "benzin"
    kind "StaticLib"
    language "C++"
    cppdialect "C++20"
    location "../source/benzin"

    targetname "benzin_%{cfg.buildcfg}"

    targetdir "../lib"
    objdir "../build/benzin/%{cfg.buildcfg}"

    pchheader "benzin/config/bootstrap.hpp"
    pchsource "../source/benzin/config/bootstrap.cpp"

    defines { "BENZIN_PROJECT" }

    files {
        "../source/benzin/**.hpp",
        "../source/benzin/**.inl",
        "../source/benzin/**.cpp",
```

```
    }

    includedirs {
        "../",
        "../source",
        "../source/benzin",
    }
```

3.1.3 Пісочниця

Пісочниця є високорівневим рендерером, що використовує бібліотеку залежностей та рендер-бібліотеку для створення додатку. Основною функцією пісочниці є реалізація створення та взаємодії з рендер-проходами. Вона забезпечує високорівневий інтерфейс для керування рендерером, дозволяючи зосередитись на створенні і налаштуванні рендер-проходів для досягнення потрібного візуального ефекту.

```
project "sandbox"
    kind "ConsoleApp"
    language "C++"
    cppdialect "C++20"
    location "../source/sandbox"

    targetname "sandbox_%{cfg.buildcfg}"

    targetdir "../bin"
    objdir "../build/sandbox/%{cfg.buildcfg}"

    links {
        "third_party",
        "benzin",
    }

    pchheader "bootstrap.hpp"
    pchsource "../source/sandbox/bootstrap.cpp"

    files {
        "../source/sandbox/**.hpp",
        "../source/sandbox/**.inl",
        "../source/sandbox/**.cpp",

        "../source/shaders/**.hlsl",
        "../source/shaders/**.hlsli",
    }

    includedirs {
        "../source/",
        "../source/sandbox",
    }
```

```
vpaths {
    ["shaders/*"] = { "../source/shaders/**.hlsl", "../source/shaders/**.hlsli" }
}
```

Це були проєкти, з яких складається рішення для Visual Studio. Але логічно рендерер складається з 3 модулів, такі як:

- низькорівневий рендерер;
- модуль роботи з сутностями;
- високорівневий рендерер.

3.2 Низькорівневий рендерер

Низькорівневий рендерер – це модуль, який відповідає за зручну роботу з DirectX 12 API. Його основні задачі:

- створення GPU об’єктів;
- управління відео пам’яттю;
- копіювання даних з CPU у буфер чи текстуру;
- запис та відправка команд на GPU;
- синхронізація CPU та GPU;
- компіляція щейдерів.

Кожен клас, який описує DirectX 12 інтерфейс, в конструкторі створює відповідний інтерфейс, та у деструкторі його звільняє. Це дозволяє уникати витіку пам'яті, оскільки при заверешенні програми перевіряється кількість посилань на об’еект. Для зручного користування в ренедерері було використано розумні покажчики C++ (std::unique_ptr чи std::shared_ptr) для роботи з GPU об’єктами.

3.2.1 Графічний Backend

Для зручного створення/користуванням фабрики та вибору кращого адаптера був створений клас *Backend*.

В залежності від того, який тип збірки використовується (це Debug або Release), фабрика буде створена з Debug контектом чи ні. Це дозволяє отримавати додаткову інформацію від DXGI при помилках.

```
void Backend::CreateDXGIFactory()
{
    UINT dxgiFactoryFlags = 0;
#if defined(BENZIN_DEBUG_BUILD)
    dxgiFactoryFlags |= DXGI_CREATE_FACTORY_DEBUG;
#endif

    ComPtr<IDXGIFactory2> dxgiFactory2;
    BENZIN_HR_ASSERT(CreateDXGIFactory2(dxgiFactoryFlags, IID_PPV_ARGS(&dxgiFactory2)));
    BENZIN_HR_ASSERT(dxgiFactory2->QueryInterface(IID_PPV_ARGS(&m_DXGIFactory)));
}
```

Також важливо знати, який адаптер буде використовуватися при рендерінгу, коли в системі їх декілька. Згодом до DXGI API було додано функціонал, за допомогою якого можна отримати список адаптерів за певною перевагою (за мінімальною кількістю потужності, яка відеокарта споживає, чи за високою ефективністю). Оскільки при розробці даного рендереру будуть малюватись великі сцени, тому вибирається найефективніший адапер.

```
void Backend::CreateDXGIMainAdapter()
{
    ComPtr<IDXGIAdapter> dxgiAdapter;
    BENZIN_HR_ASSERT(m_DXGIFactory->EnumAdapterByGpuPreference(
        config::GetMainAdapterIndex(),
        DXGI_GPU_PREFERENCE_HIGH_PERFORMANCE,
        IID_PPV_ARGS(&dxgiAdapter)
    ));
    BENZIN_HR_ASSERT(dxgiAdapter->QueryInterface(IID_PPV_ARGS(&m_MainDXGIAdapter)));

    DXGI_ADAPTER_DESC3 dxgiAdapterDesc;
    m_MainDXGIAdapter->GetDesc3(&dxgiAdapterDesc);

    /* ... */
}
```

При запуску програми у консоль буде виведена інформація про те, який адаптер був обраний та які в нього є характеристики.

```
[Trace][0:0:00.125][backend.cpp:89]:
==============================================
Main Adapter
AMD Radeon RX 6800
VendorID: 4098
DeviceID: 29631
DedicatedVideoMemory: 16339MB, 16GB
DedicatedSystemMemory: 0MB, 0GB
SharedSystemMemory: 32606MB, 32GB
==============================================
```

Рисунок 3.1 – Приклад виводу інформації про головний адаптер

Також DirectX 12 API надає певний функціонал, за допомогою якого можна увімкнути Debug шар як ці сторони CPU, так і зі сторони GPU. За це також відповідає класс *Backend*. При ініціалізації, він вмикає Debug шар, коли використовується Debug збірка.

```
void EnableD3D12DebugLayer()
{
    ComPtr<ID3D12Debug5> d3d12Debug;
    BENZIN_HR_ASSERT(D3D12GetDebugInterface(IID_PPV_ARGS(&d3d12Debug)));
    d3d12Debug->EnableDebugLayer();
    d3d12Debug->SetEnableGPUBasedValidation(true);
    d3d12Debug->SetEnableSynchronizedCommandQueueValidation(true);
    d3d12Debug->SetEnableAutoName(true);
}
```

3.2.2 Графічний девайс

Графічний девайс використовується при створенні будь-якого графічного об'єкта. В свою чергу для створення девайсу потрібен адаптер, щоб використовувати властивості та можливості адаптера. Девайс в конструкторі створює відповідний DirectX 12 інтерфейс, а також єдину на весь рендерер кореневий підпис, та три черги команд на кожен з GPU рушіїв.

```
Device::Device(const Backend& backend)
{
    BENZIN_HR_ASSERT(D3D12CreateDevice(
        backend.GetDXGIMainAdapter(),
        D3D_FEATURE_LEVEL_11_0,
        IID_PPV_ARGS(&m_D3D12Device)
    ));

#if defined(BENZIN_DEBUG_BUILD)
    BreakOnD3D12Error(true);
```

```
#endif

    CreateBindlessRootSignature();

    {
        m_DescriptorManager = new DescriptorManager{ *this };
        m_TextureLoader = new TextureLoader{ *this };
    }

    {
        m_CopyCommandQueue = new CopyCommandQueue{ *this };
        m_ComputeCommandQueue = new ComputeCommandQueue{ *this };
        m_GraphicsCommandQueue = new GraphicsCommandQueue{ *this };
    }
}
```

Також при створенні девайсу вмикається зупинка програми, при попередження (warning), помилці (error) чи пошкодження (corruption), якщо використовується Debug збірка. Це робиться з тою метою, щоб одразу побачити, де була помилка.

```
void Device::BreakOnD3D12Error(bool isBreak)
{
    ComPtr<ID3D12InfoQueue> d3d12InfoQueue;
    BENZIN_HR_ASSERT(m_D3D12Device->QueryInterface(IID_PPV_ARGS(&d3d12InfoQueue)));

    d3d12InfoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_WARNING, isBreak);
    d3d12InfoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_ERROR, isBreak);
    d3d12InfoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_CORRUPTION, isBreak);
}
```

3.2.3 Черги команд для кожного типу GPU рушія

Розроблений рендер надає можливість звертатися до окремої черги команд, яка буде запущена на тому чи іншому GPU рушії.

Наприклад, черга команд для копіювання використовується при завантажені 3D-моделі з файлу, щоб завантажити на GPU дані про текстури, буфер вершин, буфер індексів тощо.

Черга команд для обчислення використовується при генерації кубічної текстури з рівнопрямокутної текстури довкілля.

Черга команд для графіки використовується для малювання 3D-моделей, малювання довкілля.

Для простоти роботи наразі черга команд для копіювання та черга команд для обчислення використовує тільки один командний алокатор. Натомість черга команд для графіки використовує стільки командних алокаторів, скільки містить ланцюг обміну буферів. В даному випадку кількість буферів дорівнює 3, але це значення може змінюватись за потребою. При створенні ланцюга обміну, передається черга команд для графіки, щоб зв'язати ланцюг та чергу.

3.2.4 Ланцюг обміну. Синхронізація CPU та GPU

При створенні ланцюгу обміну до нього прив'язується черга команд для графіки. Ланцюг обміну надає зручний доступ до поточного буферу, який може бути використаний для малювання. Також надає метод, для змінення розміру буферів. Цей метод повинен бути викликаний, при зміні розміру екрану, оскільки буфер ланцюга обміну повинен дорівнювати розміру екрану.

```
void SwapChain::ResizeBackBuffers(uint32_t width, uint32_t height)
{
    m_Device.GetGraphicsCommandQueue().Flush();

    for (auto& backBuffer : m_BackBuffers)
    {
        backBuffer.reset();
    }

    BENZIN_HR_ASSERT(m_DXGISwapChain->ResizeBuffers(
        static_cast<UINT>(m_BackBuffers.size()),
        width,
        height,
        static_cast<DXGI_FORMAT>(config::GetBackBufferFormat()),
        m_DXGISwapChainFlags
    ));

    RegisterBackBuffers();
    UpdateViewportDimensions(static_cast<float>(width), static_cast<float>(height));
}
```

Але перед цим потрібно обов'язково очистити чергу команд для графіки та звільнити буфери ланцюгу обміну, що і робиться з початку. Це робиться, оскільки при зміні розміру буфери, вони не повинні використовуватись GPU.

Також ланцюг обміну контролює як саме буде відбуватись перемикання буферів, з вертикальною синхронізацію, чи ні. Цей параметр може бути змінний під час виконання рендереру. Але спочатку треба перевірити, чи є апаратна підтримка цього.

```
uint32_t isAllowTearing = 0;
BENZIN_HR_ASSERT(backend.GetDXGIFactory() >CheckFeatureSupport(
    DXGI_FEATURE_PRESENT_ALLOW_TEARING,
    &isAllowTearing,
    sizeof(isAllowTearing)
));

if (isAllowTearing)
{
    m_DXGISwapChainFlags |= DXGI_SWAP_CHAIN_FLAG_ALLOW_TEARING;
    m_DXGIPresentFlags |= DXGI_PRESENT_ALLOW_TEARING;
}
```

Та якщо є апаратна підтримка, то ланцюг обміну потрібно створити з відповідним прапором і використовувати потрібний флаг, при перемиканні буферу.

Кадр в польоті – це той кадр, який вже був оброблений CPU, але не був оброблений до кінця GPU. Тобто CPU може продовжувати працювати над наступним кадром чи за потреби чекати, якщо немає вільного кадру. Це робиться для того, щоб CPU та GPU були завжди в роботі і час, який CPU чекає на GPU, був мінімальним

Різниця в кадрах, який відображається на екрані, та кадру, який насправді обчислюється на стороні CPU, може дорівнювати кількості буферів в ланцюгу обміну. Синхронізація відбувається коли, різниця кадрів дорівнює кількості буферів в ланцюгу обміну при черговому виклику перемикання буферів.

```
void SwapChain::Flip()
{
    m_Device.GetGraphicsCommandQueue().UpdateFenceValue(m_FrameFence, ++m_CPUFrameIndex);

    BENZIN_HR_ASSERT(m_DXGISwapChain->Present(m_IsVSyncEnabled, m_DXGIPresentFlags));

    m_GPUFrameIndex = m_FrameFence.GetCompletedValue();

    // Wait for GPU to not exceed maximum queued frames
    const uint64_t frameDistance = m_CPUFrameIndex - m_GPUFrameIndex;
    if (frameDistance >= config::GetBackBufferCount())
    {
```

```
            const uint64_t gpuFrameIndexToWait = m_CPUFrameIndex -
config::GetBackBufferCount() + 1;
            m_FrameFence.WaitForGPU(gpuFrameIndexToWait);

            m_GPUFrameIndex = m_FrameFence.GetCompletedValue();
        }
    }
```

3.2.5 Робота з ресурсами

Для роботи з ресурсами DirectX 12 надає єдиний інтерфейс, але це не зручно використовувати єдиний інтерфейс для роботи з буферами та ресурсами.

Тому було створено 2 класи для роботи – це *BufferResource* та *TextureResource*. При створенні ресурсу будуть передані тільки ті параметрі, які притаманні одному з типу ресурсу.

Як було описано вище, щоб використовувати ресурси у конвеєрі, треба створити перегляд ресурсу. Тому у кожному ресурсу є хеш таблиця, яка зберігая всі перегляди, в залежності від типу.

```
std::unordered_map<Descriptor::Type, std::vector<Descriptor>> m_Views;
```

Щоб додати певний перегляд, треба викликати один з методів у ресурсу, який додасть його до хеш таблиці. Є можливість передати структуру, яка буде описувати перегляд, але якщо це не потрібно, кожна відповідна структура містить значення полів за замовчуванням. В основному використовується перегляди за замочуванням, але при окремих випадках є можливість використовувати саме той перегляд, який потрібен. Щоб мати змогу використовувати дескриптори повторно, в деструкторі ресурс не лише звільняє відповідних DirectX 12 інтерфейс, а також всі алоковані дескриптори.

```
Resource::~Resource()
{
    for (const auto& [descriptorType, descriptors] : m_Views)
    {
        for (const auto& descriptor : descriptors)
        {
            m_Device.GetDescriptorManager().DeallocateDescriptor(
```

```
                descriptorType,
                descriptor
            );
        }
    }

    dx::SafeRelease(m_D3D12Resource);
}
```

Також кожен ресурс містить в собі дані про поточний стан ресурсу. Це було додано в DirectX 12 API, переклавши відповідальність за управління станом ресурсу з графічного драйвера на програму, щоб зменшити загальне використання CPU і забезпечити багатопоточність і попередню обробку драйвера. При необхідній зміні стану ресурсу, треба звернутись до командного списку і передати ресурс на новий стан.

```
const D3D12_RESOURCE_BARRIER d3d12ResourceBarrier
{
    .Type{ D3D12_RESOURCE_BARRIER_TYPE_TRANSITION },
    .Flags{ D3D12_RESOURCE_BARRIER_FLAG_NONE },
    .Transition
    {
        .pResource{ resource.GetD3D12Resource() },
        .Subresource{ D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES },
        .StateBefore
        {
            static_cast<D3D12_RESOURCE_STATES>(resource.GetCurrentState())
        },
        .StateAfter{ static_cast<D3D12_RESOURCE_STATES>(resourceStateAfter) }
    }
};

m_D3D12GraphicsCommandList->ResourceBarrier(1, &d3d12ResourceBarrier);

resource.SetCurrentState(resourceStateAfter);
```

За допомогою даного функціоналу, не треба десь запам’ятовувати, який стан ресурсу був до цього.

3.2.6 Менеджер дескрипторів

Для використання перегляду ресурсу треба мати відповідний дескриптор з потрібної купи дескрипторів. Окремо мати купи не зручно, та не зручно слідкувати за тим, чи можна використовувати даний дескриптор чи ні. Тому було розроблено клас

DescriptorManager, який займається створенням та звільненням дескрипторів. Є відповідні методи, які виділяють та звільняють дескриптори за типом для їх повторного використання.

```
enum Descriptor::Type : uint8_t
{
    RenderTargetView,
    DepthStencilView,
    ConstantBufferView,
    ShaderResourceView,
    UnorderedAccessView,
    Sampler
};

class DescriptorManager
{        /* ... */
    Descriptor AllocateDescriptor(Descriptor::Type descriptorType);
    void DeallocateDescriptor(
        Descriptor::Type descriptorType,
        const Descriptor& descriptor
    );
    /* ... */
}
```

Менеджер дескрипторів містить в собі 4 необхідних купи дескрипторів – купа дескрипторів для ресурсів, купа дескрипторів для семплерів, купа дескрипторів для рендер цілей та купа дескрипторів для буферів глибини-трафарету.

3.2.7 Єдиний кореневий підпис. Динамічні ресурси

Використовувати функціонал, який надає стандартний DirectX 12 для роботи з ресурсами доволі трудомісткий.

По-перше обов’язкове використання першого етапу графічного конвеєра, тобто йде мова про асемблер введення. Потрібно створювати опис під кожну нову комбінацію вхідних параметрів для вершинного шейдера.

По-друге створення нових кореневих сигнатур для опису ресурсів, які буде використовувати шейдери. Створення кореневої сигнатури є доволі трудомістким, оскільки потрібно описувати кожен параметр, а це робиться не завжди тривіально.

По-третє зростання кількості станів конвеєра через попередні дві умови. Якщо позбавитися їх, кількість станів конвеєра стане менше та створення їх стане простіше.

Тому було прийнято рішення про використання нової шейдерної моделі, яка надає функцію *динамічні ресурси*. За допомогою цієї функції у шейдерах можна звертатись до ресурсів напряму за індексом у відповідних дескрипторних купах. Використовувати даний функціонал можна тільки при певній версії Windows SDK. Тому у файлі конфігурації DirectX 12 вказується потрібна версія та шлях до SDK.

```
#if defined(BENZIN_PROJECT)
extern "C"
{
    __declspec(dllexport) extern const UINT D3D12SDKVersion = 608;
    __declspec(dllexport) extern const char* D3D12SDKPath = "./";
}
#endif
```

Але все таки одну кореневу сигнатуру потрібно зробити, щоб мати змогу передавати індекси ресурсів у шейдери. Коренева сигнатура буде складатися з одного кореневого елементу (кореневі 32-бітні константи) та списку статичних семплерів. Статичні семплери – це такі самі семплери, але які знаходяться безпосередньо всередині кореневої сигнатури.

За створення єдиного кореневого підпису відповідає графічний девайс, і спочатку йде перевірка на підтримку шейдерної моделі 6.6.

```
D3D12_FEATURE_DATA_SHADER_MODEL d3d12FeatureDataShaderModel
{
    .HighestShaderModel{ D3D_SHADER_MODEL_6_6 }
};

BENZIN_HR_ASSERT(m_D3D12Device->CheckFeatureSupport(
    D3D12_FEATURE_SHADER_MODEL,
    &d3d12FeatureDataShaderModel,
    sizeof(d3d12FeatureDataShaderModel))
);
BENZIN_ASSERT(
    d3d12FeatureDataShaderModel.HighestShaderModel >= D3D_SHADER_MODEL_6_6
);
```

Після успішної перевірки апаратнох підтримки, створюється потрібний список статичних семплерів та опис для кореневої сигнатури.

```
const D3D12_VERSIONED_ROOT_SIGNATURE_DESC d3d12VersionedRootSignatureDesc
{
    .Version{ D3D_ROOT_SIGNATURE_VERSION_1_1 },
    .Desc_1_1
    {
        .NumParameters{ 1 },
        .pParameters{ &d3d12RootParameter },
        .NumStaticSamplers{ static_cast<UINT>(d3d12StaticSamplers.size()) },
        .pStaticSamplers{ d3d12StaticSamplers.data() },
        .Flags
        {
            D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT |
            D3D12_ROOT_SIGNATURE_FLAG_CBV_SRV_UAV_HEAP_DIRECTLY_INDEXED |
            D3D12_ROOT_SIGNATURE_FLAG_SAMPLER_HEAP_DIRECTLY_INDEXED
        }
    }
};

ComPtr<ID3DBlob> d3d12SerializedRootSignatureBlob;
ComPtr<ID3DBlob> d3d12ErrorBlob;
BENZIN_HR_ASSERT(D3D12SerializeVersionedRootSignature(
    &d3d12VersionedRootSignatureDesc,
    &d3d12SerializedRootSignatureBlob,
    &d3d12ErrorBlob
));

BENZIN_HR_ASSERT(m_D3D12Device->CreateRootSignature(
    0,
    d3d12SerializedRootSignatureBlob->GetBufferPointer(),
    d3d12SerializedRootSignatureBlob->GetBufferSize(),
    IID_PPV_ARGS(&m_D3D12BindlessRootSignature)
));
```

Так виглядає створення єдиної кореневої сигнатури на весь рендерер. Відповідний HLSL код, буде виглядати наступним чином:

```
struct RootConstants
{
    uint4 Constants[8];

    uint Get(uint index)
    {
        return Constants[index >> 2][index & 3];
    }
};

ConstantBuffer<RootConstants> g_RootConstants : register(b0, space0);

SamplerState g_PointWrapSampler : register(s0);
SamplerState g_PointClampSampler : register(s1);
```

```
SamplerState g_LinearWrapSampler : register(s2);
SamplerState g_LinearClampSampler : register(s3);
SamplerState g_Anisotropic16WrapSampler : register(s4);
SamplerState g_Anisotropic16ClampSampler : register(s5);
SamplerComparisonState g_ShadowSampler : register(s6);
```

В структурі *RootConstants* зберігається масив з 32 елементів. При написанні масивів у HLSL потрібно бути дуже уважним, оскільки тут правила вирівнювання відрізняються від правил мови C++. Uint Constants[32] != uint4 Constants[8], оскільки мінімальна одиниця вирівнювання в HLSL – це 16 байтів. Тобто Uint Constants[32] насправді буде перетворено у Uint4 Constants[32], а це не те, що очікується.

Також для швидкого доступу до елементів масиву було використано побітові операції, які зможуть незначно покращити продуктивність.

Щоб прив’язувати ресурси до конвеєра, існує відповідні методи в списку команд. Тобто ці методи будуть змінювати значення 32-бітних констант за переданим індексом, які описує єдиний кореневий підпис.

```
class CommandList
{
    /* ... */
    void SetRootConstant(uint32_t rootIndex, uint32_t value);
    void SetRootConstantBuffer(uint32_t rootIndex, const Descriptor& cbv);
    void SetRootShaderResource(uint32_t rootIndex, const Descriptor& srv);
    void SetRootUnorderedAccess(uint32_t rootIndex, const Descriptor& uav);
    /* ... */
};
```

3.2.8 Менеджер шейдерів

Цей компонент доволі простий, але одночасно дуже зручний у використанні. Його мета надавати шейдерний код, який вже був скомпільований. Тобто при передачі однакових параметрів шейдер повинен компілюватись тільки при першому такому випадку. При всіх інших – менеджер повинен надавати скомпільований код. Оскільки шейдер характеризують такі параметри, як назва файлу, назва точки входу та масив визначень (defines), з цього можна утворити унікальну строку, яку можна захешувати. Тим самим у кожного шейдера буде свій хеш код.

```
size_t GetShaderKey(
    std::string_view fileName,
    std::string_view entryPoint,
    const std::vector<std::string>& defines
)
{
    const std::string key = std::accumulate(
        defines.begin(),
        defines.end(),
        fmt::format("{}{}", fileName, entryPoint)
    );
    return std::hash<std::string>{}(key);
}
```

Скомпільований шейдерних код зберігається у хеш таблиці, ключом якої як раз і є хеш код шейдера. Щоб отримати скомпільований шейдерний код, треба викликати функцію, яка перевірить, чи є вже такий шейдер, інакше скомпілює його.

```
std::span<const std::byte> GetShaderBlob(
    ShaderType shaderType,
    std::string_view fileName,
    std::string_view entryPoint,
    const std::vector<std::string>& defines
)
{
    const size_t shaderKey = GetShaderKey(fileName, entryPoint, defines);

    auto shaderIterator = g_Shaders.find(shaderKey);

    if (shaderIterator == g_Shaders.end())
    {
        const ComPtr<IDxcBlobEncoding> dxcShaderSourceBlob =
g_ShaderCompiler.LoadShaderFromFile(fileName);
        std::vector<std::byte> shaderByteCode = g_ShaderCompiler.CompileShader(
            shaderType,
            entryPoint,
            defines,
            dxcShaderSourceBlob
        );

        shaderIterator = g_Shaders.insert(std::make_pair(
            shaderKey,
            std::move(shaderByteCode)
        )).first;
    }

    return shaderIterator->second;
}
```

За компіляцію шейдерів відповідає компілятор DXC. Він використовується, оскільки минулий компілятор підтримував шейдерні моделі до 5.1 включно. Так як в

цій роботі було використано функцію динамічні ресурси з шейдерної моделі 6.6, тільки DXC може підтримати цей функціонал.

3.3 Модуль роботи з сутностями

Цей модуль відповідає за роботу з сутностями. Це не є основним модулем роботи, тому він повинен бути простим та легким у використанні. Було прийнято рішення використовувати ECS підхід. Даний підхід позбавляє він написання окремих класів сутностей, тому що сутність – це набір компонентів.

Таким чином дуже легко звернутись до всіх сутностей, у яких є певний компонент чи набір компонентів.

3.3.1 Структури 3D-моделі

3D-модель складається з двох основних елементів – це меш та набір матеріалів. Але це вершина айсбергу.

До складу меша входить:

– буфер вершин;

– буфер індексів;

– буфер меш примітивів та меш примітиви.

В даному контексті меш примітив – це якась частина від всього меша. Тобто примітив повинен визначати, з якої вершини він починається, з якого індексу та скільки індексів відноситься для цього примітиву. Але меш примітив набуває різного вигляду на CPU та на GPU, оскільки для створення команди малювання потрібен один перелік даних (це сторона CPU), а для отримання доступу вершини зі сторони GPU – інший.

```
struct Primitive
{
    uint32_t IndexCount{ 0 };
    PrimitiveTopology PrimitiveTopology{ PrimitiveTopology::Unknown };
};
```

```
struct GPUPrimitive
{
    uint32_t StartVertex;
    uint32_t StartIndex;
    uint32_t IndexCount;
};
```

При написанні відповідної команди для малювання треба вказати кількість вершин, які будуть опрацьовані. В даному випадку перший етап графічного конвеєра можна пропустити, оскільки доступ до буферу вершин та буферу індексів є напряму з вершинного шейдера. Тому кількість вершин, які будуть відправлені на опрацювання графічним конвеєром буде дорівнювати кількості індексів.

У вершинному шейдері будуть використані 3 буфери, щоб дістати вершину, це буфер меш примітивів, буфер індексів та власне буфер вершин. При створенні точки входу в вершинний шейдер, на вхід можна отримати параметр, який буде містити індекс вершини (анотація SV_VertexID), яка повинна бути опрацьована цим шейдером.

```
    StructuredBuffer<MeshVertex> vertexBuffer =
ResourceDescriptorHeap[BENZIN_GET_ROOT_CONSTANT(g_VertexBufferIndex)];
    StructuredBuffer<uint32_t> indexBuffer =
ResourceDescriptorHeap[BENZIN_GET_ROOT_CONSTANT(g_IndexBufferIndex)];

    StructuredBuffer<MeshPrimitive> meshPrimitiveBuffer =
ResourceDescriptorHeap[BENZIN_GET_ROOT_CONSTANT(g_MeshPrimitiveBufferIndex)];
    const MeshPrimitive meshPrimitive = meshPrimitiveBuffer[meshPrimitiveIndex];
    const uint32_t index = indexBuffer[meshPrimitive.StartIndex + vertexID];
    const MeshVertex vertex = vertexBuffer[meshPrimitive.StartVertex + index];
```

До складу моделі входить:

– меш;

– буфер для примітивів малювання та примітиви малювання;

– буфер вузлів та вузли;

– текстури, які використовуються в матеріалах;

– буфер матеріалів.

Примітив – це просто частина меша. Тобто він на має ніяких характеристик, за допомогою яких можна зрозуміти, як він буде виглядати. Цим займається матеріал. Але один і той самий матеріал може мати декілька примітивів, тому не можна просто додати необхідну інформацію до примітива. Для цього існує примітив для малювання, який зберігає індекс на меш примітив та індекс на потрібний матеріал.

```
struct DrawPrimitive
{
    uint32_t MeshPrimitiveIndex{ 0 };
    uint32_t MaterialIndex{ 0 };
};
```

Вузол – це перелік примітивів для малювання. Він існує, тому що при проєктуванні 3D-моделей існують такі ситуації, коли використовується одна і та сама частина моделі, але в різних місцях. З точки зору економії пам'яті нема сенсу створювати цю частину декілька разів, а можна просто застосувати матрицю, яка буде змінювати положення цієї частини. Приклад: у автомобіля є 4 однакових колеса, тому можна один раз створити колесо, і далі примінивши 4 рівних матриці отримаємо 4 колеса, але в різних позиціях. В даному випадку вузол буде приймати різний вигляд для CPU та GPU.

```
struct Node
{
    IterableRange<uint32_t> DrawPrimitiveRange;
};
struct GPUNode
{
    DirectX::XMMATRIX TransformMatrix;
    DirectX::XMMATRIX InverseTransformMatrix;
};
```

DrawPrimitiveRange використовується при створенні команди для малювання. Зі сторони CPU малювання 3D-моделі буде виглядити так:

```
Model model = /* ... */;
Mesh& mesh = model.GetMesh();

for (const auto& node : model.GetNodes())
{
    for (uint32_t drawPrimitiveIndex : node.DrawPrimitiveRange)
    {
```

```
            const auto& dPrimitive = model.GetDrawPrimitives()[drawPrimitiveIndex];
            const auto& mPrimitive = mesh.GetPrimitives()[dPrimitive.MeshPrimitiveIndex];

            commandList.SetPrimitiveTopology(mPrimitive.PrimitiveTopology);
            commandList.DrawVertexed(mPrimitive.IndexCount);
        }
    }
```

TransformMatrix буде використано при трансформації вершини у вершинному шейдері, а InverseTransformMatrix дл трансформації нормалей.

3.3.2 Завантаження 3D-моделі з файлу glTF

Вище було описано складові формату glTF і тепер стоїть задача перенести структуру glTF на структуру, яка описана для 3D-моделі.

Для завантаження моделі є бібліотека tinygltf. Щоб завантажити модель, потрібно передати відповідний шлях до моделі і в результаті буде отримано *tinygltf::Model.* Оскільки до складу моделі входить декілька компонентів, розбір *tinygltf::Model* буде в декілька етапів:

1. Отримання даних для меша.
2. Отримання даних про вузли.
3. Отримання даних про текстури.
4. Отримання даних про матеріали.
5. Створення відповідних буферів.

Отримання більшості даних не є проблемою, крім даних про вузли. Структура вузлів у форматі glTF має деревоподібну структуру. Вузол містить певну матрицю трансформації і дочірній вузол повинен використовувати не тільки свою матрицю, а ще й матрицю батька. Тому матриця трансформації поточного вузла буде складатися з добутку матриць батьків у порядку їх проходження та трансформації поточного вузла.

Завдяки цьому можна перетворити деревоподібну структуру у масив, бо опрацювання дерева кожен кадр не є оптимальним підходом (йдеться про добуток матриць).

3.3.3 Робота з просторами

Для роботи з просторами треба мати 3 матриці – це матриця світу, матриця огляду та матриця перспективи.

Матриця світу буде відноситись до певної сутності (об’єкта). Тобто сутність повинна мати компонент, яка відповідає за його положення в світі. Це *TransformComponent*:

```
struct TransformComponent
{
    DirectX::XMFLOAT3 Scale{ 1.0f, 1.0f, 1.0f };
    DirectX::XMFLOAT3 Rotation{ 0.0f, 0.0f, 0.0f };
    DirectX::XMFLOAT3 Translation{ 0.0f, 0.0f, 0.0f };

    DirectX::XMMATRIX GetMatrix() const
    {
        const DirectX::XMMATRIX scaling = DirectX::XMMatrixScaling(
            Scale.x,
            Scale.y,
            Scale.z
        );

        const DirectX::XMMATRIX rotation =
            DirectX::XMMatrixRotationX(Rotation.x) *
            DirectX::XMMatrixRotationY(Rotation.y) *
            DirectX::XMMatrixRotationZ(Rotation.z);

        const DirectX::XMMATRIX translation = DirectX::XMMatrixTranslation(
            Translation.x,
            Translation.y,
            Translation.z
        );

        return scaling * rotation * translation;
    }

    DirectX::XMMATRIX GetInverseMatrix() const
    {
        const DirectX::XMMATRIX transform = GetMatrix();
        auto transformDeterminant = DirectX::XMMatrixDeterminant(transform);

        return DirectX::XMMatrixInverse(&transformDeterminant, transform);
    }
```

```
};
```

Цей компонент визначає де саме буде малюватись об'єкт і як саме (мається на увазі розмір та обертання).

Матриця огляду та матриця проекції буде відноситись до класу камери (Camera). Положення, орієнтація та передній напрямок камери будуть використані для створення матриці огляду.

```
void Camera::UpdateViewMatrix()
{
    m_ViewMatrix = DirectX::XMMatrixLookAtLH(
        m_Position,
        DirectX::XMVectorAdd(m_Position, m_FrontDirection),
        m_UpDirection
    );

    DirectX::XMVECTOR viewDeterminant = DirectX::XMMatrixDeterminant(m_ViewMatrix);
    m_InverseViewMatrix = DirectX::XMMatrixInverse(&viewDeterminant, m_ViewMatrix);
}
```

Для матриці проекції насамперед треба визначити тип (перспективна чи орографічна). В основному для рендерінгу використовується перспективна матриця, тому саме про неї буде йти мова.

Камера – це просто набір з 2 матриць. За певними параметрами вона може оновлювати внутрішні матриці, але за введення користувача відповідає окремий клас *FlyCameraController*. Як правило, для переміщення використовується клавіатура, а для орієнтації – мишка.

При натисканні на клавіатуру буде здійснене переміщення камери в залежності від клавіші клавіатури.

```
void FlyCameraController::OnUpdate(float dt)
{
    const float delta = m_CameraTranslationSpeed * dt;
    const auto& position = m_Camera->GetPosition();

    DirectX::XMVECTOR updatedPosition = DirectX::XMVectorZero();

    // Front / Back
    {
        const auto& frontDirection = m_Camera->GetFrontDirection();
```

```
        if (Input::IsKeyPressed(KeyCode::W))
        {
            updatedPosition = DirectX::XMVectorAdd(
                position,
                DirectX::XMVectorScale(frontDirection, delta)
            );
        }
        else if (Input::IsKeyPressed(KeyCode::S))
        {
            updatedPosition = DirectX::XMVectorSubtract(
                position,
                DirectX::XMVectorScale(frontDirection, delta)
            );
        }
    }

    // Left / Right
    {
        const auto& rightDirection = m_Camera->GetRightDirection();

        if (Input::IsKeyPressed(KeyCode::A))
        {
            updatedPosition = DirectX::XMVectorAdd(
                position,
                DirectX::XMVectorScale(rightDirection, delta)
            );
        }
        else if (Input::IsKeyPressed(KeyCode::D))
        {
            updatedPosition = DirectX::XMVectorSubtract(
                position,
                DirectX::XMVectorScale(rightDirection, delta)
            );
        }
    }

    // Up / Down
    {
        const auto& upDirection = m_Camera->GetUpDirection();

        if (Input::IsKeyPressed(KeyCode::Space))
        {
            updatedPosition = DirectX::XMVectorAdd(
                position,
                DirectX::XMVectorScale(upDirection, delta)
            );
        }
        else if (Input::IsKeyPressed(KeyCode::C))
        {
            updatedPosition = DirectX::XMVectorSubtract(
                position,
                DirectX::XMVectorScale(upDirection, delta)
            );
        }
    }
```

```
        if (!DirectX::XMVector4Equal(updatedPosition, DirectX::XMVectorZero()))
        {
            m_Camera->SetPosition(updatedPosition);
        }
    }
```

Для орієнтації (передній напрямок) камери використовується Ейлерові кути, зокрема 2, які відповідають за орієнтацію вздовж осі X та осі Y. Використання даних кутів є зручним, бо при русі мишки можна отримати дельту по цим двом кутам, а потім перетворити Ейлерові кути в декартовий напрямок.

```
    void FlyCameraController::OnMouseMoved(MouseMovedEvent& event)
    {
        if (Input::IsMouseButtonPressed(MouseButton::Left))
        {
            const float deltaX = event.GetX<float>() - m_LastMousePosition.x;
            const float deltaY = event.GetY<float>() - m_LastMousePosition.y;

            m_Pitch += m_MouseSensitivity * deltaY;
            m_Yaw += m_MouseSensitivity * deltaX;

            // Clamp the up down view
            m_Pitch = std::clamp(
                m_Pitch,
                -DirectX::XM_PIDIV2 + 0.01f,
                DirectX::XM_PIDIV2 - 0.01f
            );

            // 360 rotation
            if (m_Yaw > DirectX::XM_PI)
            {
                m_Yaw = -DirectX::XM_PI;
            }

            if (m_Yaw < -DirectX::XM_PI)
            {
                m_Yaw = DirectX::XM_PI;
            }

            m_Camera.SetFrontDirection(GetCameraFrontDirection());
        }

        m_LastMousePosition.x = event.GetX<float>();
        m_LastMousePosition.y = event.GetY<float>();
    }
```

А для трансформації з Ейлерових кутів декартовий добуток використовується дана формула.

```
DirectX::XMVECTOR FlyCameraController::GetCameraFrontDirection() const
{
    const DirectX::XMVECTOR frontDirection
    {
        DirectX::XMVectorSet(
            DirectX::XMScalarCos(m_Yaw) * DirectX::XMScalarCos(m_Pitch),
            DirectX::XMScalarSin(m_Pitch),
            DirectX::XMScalarSin(m_Yaw) * DirectX::XMScalarCos(m_Pitch),
            0.0f
        )
    };

    return DirectX::XMVector3Normalize(frontDirection);
}
```

3.4 Високорівневий рендерер

Високорівневий рендерер – це модуль, який відповідає за те, як саме буде малюватись кадр. В основному кадр малюється за декілька проходів, для забезпечення оптимальної продуктивності і відображення комплексної сцени з великою кількістю об'єктів і ефектів.

Як було описано вище, існують 3 основні підходу при рендерінгу об’єктів. В даній роботі було використано підхід відкладеного рендерінгу, бо він надає значну перевагу за продуктивністю для обробки декількох джерел освітлення та надає можливість легко додавати нові техніки, що не можна сказати про прямий підхід. Прямий+ підхід доволі складний при розробці та він становиться актуальним тільки при певних випадках.

Відкладений рендерінг складається з 2 проходів, це *прохід геометрії* (*geometry pass*) та *прохід освітлення* (*lighting pass*). Також було додано ще один прохід – це прохід довкілля.

3.4.1 Прохід геометрії

В даній реалізації геометричний буфер складається з 5 текстур:

– текстура, яка зберігає альбедо матеріалу;

– текстура, яка зберігає нормалі в світовому просторі;

– текстура, яка зберігає емісійний фактор матеріалу;

– текстура, яка зберігає шорсткість та металічність;

– текстура глибини-трафарету.

Повний алгоритм рендерінгу в геометричний буфер виглядає наступним чином:

1. Прив'язка всіх необхідних буферів, щоб намалювати модель:
   - буфер вершин;
   - буфер індексів;
   - буфер меш примітивів;
   - буфер примітивів для малювання;
   - буфер вузлів;
   - буфер матеріалів.
2. В вершинному шейдері дістаються всі необхідні буфери, щоб отримати вершину та застосувати до неї:
   - локальну матрицю вузла;
   - матрицю світу;
   - матрицю огляду;
   - матрицю проекції.

Ці матриці будуть застосовані до позиції вершини, але дані матриці в такому вигляді не можна застосувати до нормалей, бо нормаль вказує на напрямок, а не на позицію. До нормалі не повинні застосуватись переміщення та масштабування, тільки поворот. Це можна зробити за допомогою одного прийому та властивостей матриць. Матриця для нормалей буде приймати вигляд *Transpose(Matrix ^ -1)*, тобто транспонувати обернену матрицю. Цей прийом треба застосувати до локальної матриці вузла та матриці світу, оскільки для роботи з освітленння потрібно нормалі в світовому просторі.

3. Останній програмований етап – це піксельний шейдер. У піксельному шейдері потрібно дістати матеріал, а вже за допомогою матеріала дістати всі необхідні текстури:

a. Якщо є альбедо текстура, то перевіряємо альфа канал. Якщо значення альфа каналу менше, ніж альфа поріг матеріалу, то піксель відкидається. Це можна зробити за допомогою ключового слова мови HLSL *discard*.
b. Якщо є карта нормалі, то дістаємо нормаль, яка знаходить в дотичному просторі. Це можна зробити за допомогою побудови TBN базису, де T – дотична (tangent), B – бідотична (bitangent), N – нормаль в світовому просторі. Щоб побудувати цей базис, треба звернутись до вбудованих функцій мови HLSL *ddx* та *ddy* (похідна по X та Y відповідно).
c. Якщо є текстура емісійності матеріалу, то зберігаємо цей фактор.
d. Якщо є текстура шорсткості та металічності, то зберігаємо ці два фактори.

Щоб записати дані кольору в рендер цілі, які прив’язані до графічного конвеєра, треба звернутись до змінних, які анотовані ключовим словом *SV_Target*. Оскільки в даному випадку прив’язано одразу 4 рендер цілі, кожна ціль має свій індекс, від 0 до 3 включно. Тобто ціль буде мати анотацію SV_Target0, друга – SV_Target1 і т.д.

```
struct PS_Output
{
    float4 Color0 : SV_Target0;
    float4 Color1 : SV_Target1;
    float4 Color2 : SV_Target2;
    float4 Color3 : SV_Target3;
};
```

У результаті будемо мати заповнений геометричний буфер.

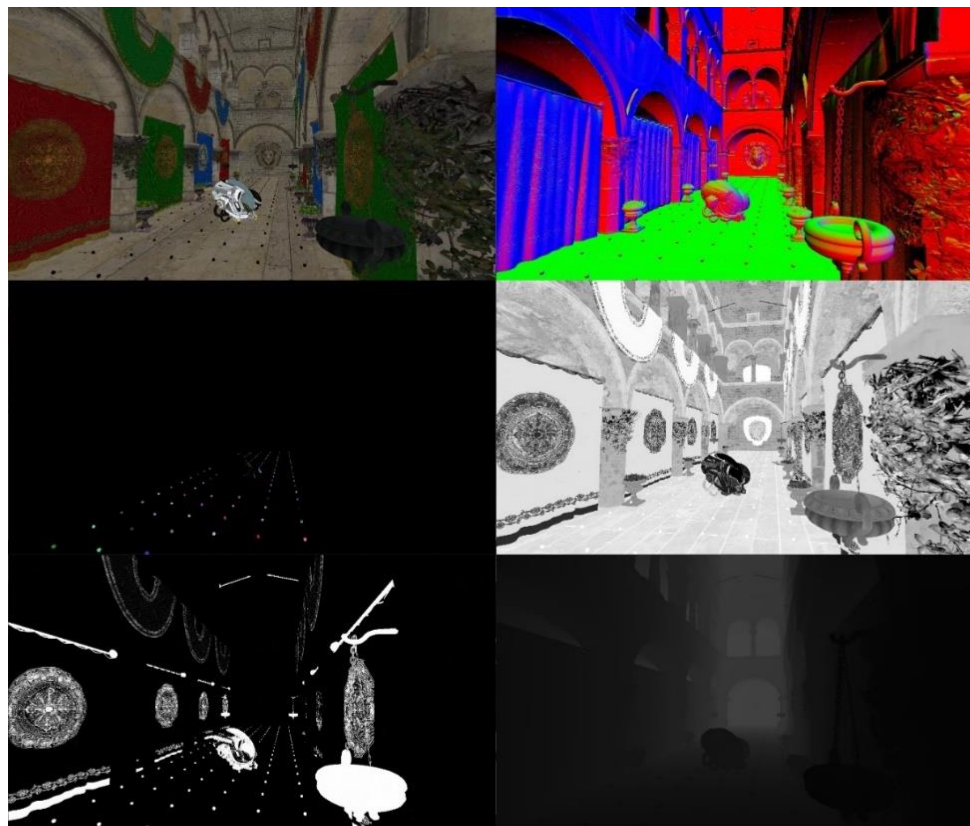


Рисунок 3.2 – Отриманий буфер геометрії

На рисунку зображений буфер геометрії після проходження графічного конвеєра. Для розуміння на рисунку зображено 6 текстур, де:

1. Альбедо.
2. Нормалі в світовому просторі.
3. Емісійний фактор.
4. Шорсткість.
5. Металічніть.
6. Глибина.

3.4.2 Прохід освітлення

Для реалізацій освітлення було обрано фізично заснований рендерінг, бо завдяки йому результат рендерінгу виглядає значно краще, ніж якщо використовувати підхід Блінна-Фонга.

Реалізація фізичного заснованого рендерінга стандартна. Але є декілька нюансів, які не є зовсім очевидні.

Розрахунок освітлення проводиться у світовому просторі. А для розрахунку освітлення окрім властивостей матеріалу треба ще деякі параметри:

- тип джерела освітлення;
- позиція у світовому просторі;
- вектор огляду у світовому просторі;
- нормаль у світовому просторі.

Розроблений рендерер підтримує 2 типи освітлення:

1. *Направлене світло (Directional light)*: використовується для джерел освітлення, які знаходиться доволі далеко (наприклад сонце), тому можна вважати, що світові промені направлені паралельно один до одного. Воно характеризується цвітом, інтенсивністю та направленням в світовому просторі.

2. *Точкове світло (Point light)*: моделює джерело світла, яке розташоване в певній точці у просторі і випромінює світло в усі напрямки. Він поширюється у всіх напрямках від своєї початкової точки, подібно до лампи або свічки. Воно характеризується цвітом, інтенсивністю, позицією у світовому просторі та параметрами затухання.

Вектор огляду – це вектор, який направлений від камери до позиції. За допомогою *нормалі* можна визначити, у якому саме напрямку буде відбитий промінь від джерела світла.

Але де взяти позицію у світовому просторі? Дана позиція рахується як проміжний етап при розрахунку позиції відсікання у вершинному шейдері при геометричному проході. При цьому в геометричному буфері нема текстури, яка б зберігала позицію у світовому просторі. Але є трюк, за допомогою якого позицію можна отримати зі значення глибини.

Для цього знадобиться значення глибини, текстурні координати повноекранного чотирикутника, обернена матриця огляду та обернена матриця проекції. Тобто алгоритм такий, що позиція в нормалізованому просторі пристрої буде трансформована в світовий простір за допомогою обернених матриць. За допомогою значення глиби та текстурних координат можна побудувати позицію в нормалізованому просторі координат, а потім примінивши обернену матрицю проекції можна отримати позицію в просторі огляду. Але ще обов'язково треба зробити перспективне ділення.

```
float3 ReconstructViewPositionFromDepth(
    float2 uv,
    float depth,
    float4x4 inverseProjectionMatrix
)
{
    // Get x/w and y/w from the viewport position
    const float x = uv.x * 2.0f - 1.0f;
    const float y = (1.0f - uv.y) * 2.0f - 1.0f;
    const float z = depth;

    const float4 ndcPosition = float4(x, y, z, 1.0f);
    const float4 viewPosition = mul(ndcPosition, inverseProjectionMatrix);

    return viewPosition.xyz / viewPosition.w;
}
```

Потім до отриманої позиції в просторі огляду треба примінити обернену матрицю огляду, щоб отримати позицію у світовому просторі.

```
float3 ReconstructWorldPositionFromDepth(
    float2 uv,
    float depth,
    float4x4 inverseProjectionMatrix,
    float4x4 inverseViewMatrix
)
{
    const float3 viewPosition = ReconstructViewPositionFromDepth(
        uv,
        depth,
        inverseProjectionMatrix
    );
    const float3 worldPosition = mul(
        float4(viewPosition, 1.0f),
        inverseViewMatrix
    ).xyz;
```

```
    return worldPosition;
}
```

Розроблений рендерер підтримує одне направлене світло та кілька точкових джерел освітлення. Кількість точкових джерел освітлення не обмежується, але треба дивить на вихідну продуктивність. Але завдяки відкладеному підходу, рендерер може використовувати біля 200 джерел освітлення без проблем з продуктивністю.

Вихідний колір буде складатись з декількох параметрів:

– вплив направленого світла;

– вплив точкового світла;

– вплив навколишнього освітлення;

– вплив емісійності матеріалу.

```
internal::pbr::Material material;
material.Albedo = gbuffer.Albedo;
material.Roughness = gbuffer.Roughness;
material.Metalness = gbuffer.Metalness;
material.F0 = internal::pbr::GetF0(gbuffer.Albedo.rgb, gbuffer.Metalness);

const float3 ambient = 0.1f * gbuffer.Albedo.rgb;
float3 color = 0.0f;

{
    internal::DirectionalLight sunLight;
    sunLight.Color = passData.SunColor;
    sunLight.Intensity = passData.SunIntensity;
    sunLight.WorldDirection = passData.SunDirection;

    color += internal::UseDirectionalLight(
        sunLight,
        material,
        worldViewDirection,
        gbuffer.WorldNormal
    );
}

{
    for (uint32_t i = 0; i < passData.PointLightCount; ++i)
    {
        color += internal::UsePointLight(
            pointLightBuffer[i],
            material,
            worldPosition,
            worldViewDirection,
            gbuffer.WorldNormal
        );
```

```
        }
    }

    color += ambient + gbuffer.Emissive;
```

3.4.3 Прохід довкілля

На етапі проходу довкілля (environment pass) в рендерері використовується кубічна текстура (cubemap texture) для відображення довкілля. Кубічна текстура є спеціальним типом текстури, який можна уявити як текстуру, яка обгортає куб. Кожна грань кубічної текстури представляє різні напрямки довкола об'єкта або сцени. Це дозволяє створити реалістичний ефект оточення та відображення оточуючого середовища на поверхнях об'єктів.

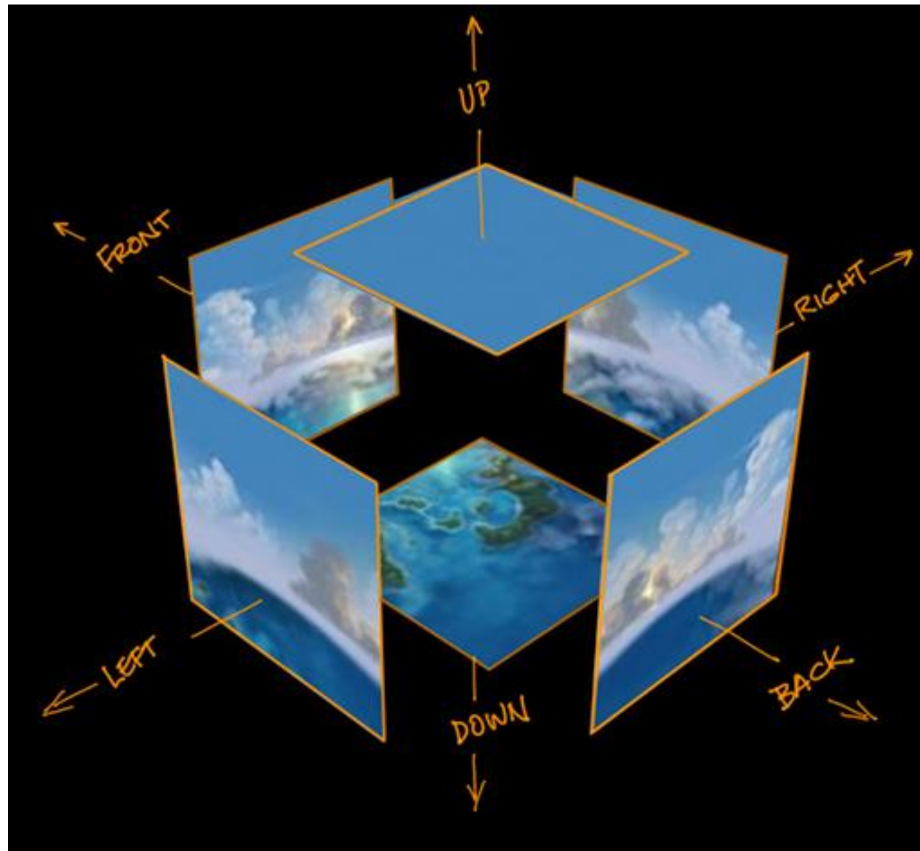


Рисунок 3.3 – Приклад кубічної текстури

Можна уявити, що сцена розташована всередині цього куба, а сам куб виступає як об'єкт, що оточує сцену. Кожен бік куба представляє певну напрямну від точки

початку, якою може бути центр куба (0, 0, 0). Таким чином, для отримання текселів з кубічної текстури можна використовувати вектор напрямку, що виходить з центру куба і спрямований до певної точки на його поверхні. Застосування цього вектору допоможе визначити, який кубічний бік текстури буде використовуватись для отримання кольору або інформації для даної напрямної.

Існує різноманітні методи малювання довкілля з використанням кубічних текстур, але багато з них потребують додаткової геометрії у вигляді вершин куба. Ці вершини растеризуються, а після цього інтерполюються, щоб стати векторами напрямку для вибірки текселя з кубічної текстури.

В розробленому рендерері малювання довкілля не потребує додаткової геометрії. Оскільки довкілля може займати всю площу вихідного зображення, можна вважати, що позиції вершин у нормалізованих координатах пристрою повністю покривають площу зображення. Мінімальною геометричною одиницею, яку можна використовувати для покриття площі, є трикутник. Прямокутну площу зображення можна уявити як два трикутники. Однак, можна йти ще далі і обійтися лише одним трикутником, який буде покривати всю площу зображення. Це буде виглядати наступним чином.

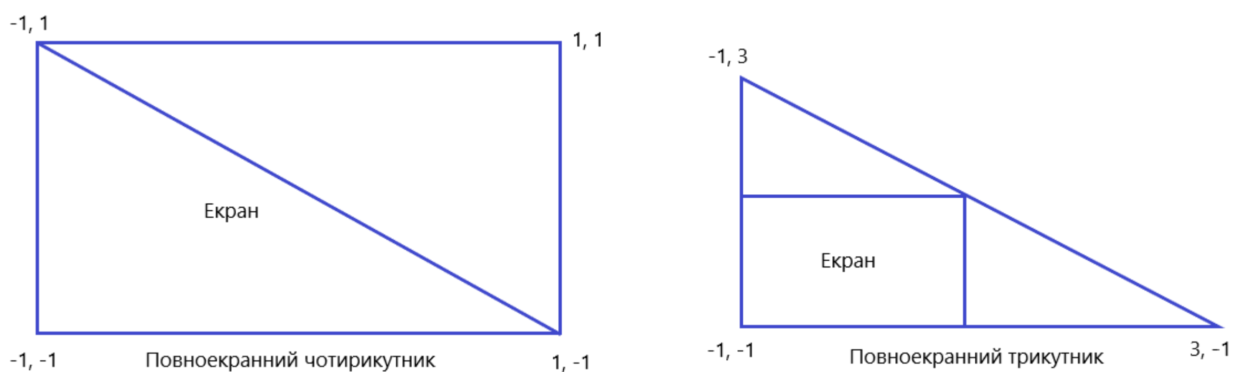


Рисунок 3.4 – Різниця між повноекранним чотирикутником та трикутником

Для визначення потрібної позиції вершини потрібен тільки індекс вершини. Наступний код, демонструє, як це можна зробити.

```
float4 GetTriangleHomogeneousPosition(uint32_t vertexID)
{
    const float x = (float)(vertexID >> 1) * -4.0f + 1.0f;
    const float y = (float)(vertexID & 1) * -4.0f + 1.0f;

    return float4(x, y, 0.0f, 1.0f);
}
```

Таким чином у вершинному шейдері було визначено позицію вершини ґрунтуючись тільки на її індексі. Після вершинного шейдера потрібна частина трикутника буде відрізана, там самим піксельний шейдер буде викликаний тільки на видимих пікселях. Але нема сенсу малювати довкілля, якщо вже на цьому місці намальований певний об’єкт. Тому при створенні стану конвеєра вказується, що тест глибини буде проходити тільки в тому разі, якщо глибина пікселя довкілля буде дорівнювати 1.0f, що є значенням при чистці буферу глибини.

```
const benzin::PipelineState::GraphicsConfig config
{
    .VertexShader{ "fullscreen_triangle.hlsl", "VS_MainDepth1" },
    .PixelShader{ "environment.hlsl", "PS_Main" },
    .PrimitiveTopologyType{ benzin::PrimitiveTopologyType::Triangle },
    .DepthState
    {
        .IsEnabled{ true },
        .IsWriteEnabled{ false },
        .ComparisonFunction{ benzin::ComparisonFunction::Equal },
    },
    .RenderTargetViewFormats{ benzin::GraphicsFormat::RGBA8Unorm },
    .DepthStencilViewFormat{ benzin::GraphicsFormat::D24Unorm_S8Uint },
};
```

В якості буферу глибини використовується той буфер, який був використаний при проході геометрії. Також важливо, щоб позиція, яка створювались у вершинному шейдері, мала глибину 1.0f.

```
namespace fullscreen_helper
{

    struct VS_Output
    {
        float4 HomogeneousPosition : SV_Position;
        float4 ClipPosition : ClipPosition;
        float2 UV : UV;
    };

} // namespace fullscreen_helper
```

```
    fullscreen_helper::VS_Output VS_MainDepth1(uint32_t vertexID : SV_VertexID)
    {
        float4 homogeneousPosition =
fullscreen_helper::GetTriangleHomogeneousPosition(vertexID);
        homogeneousPosition.z = 1.0f;

        fullscreen_helper::VS_Output output = (fullscreen_helper::VS_Output)0;
        output.HomogeneousPosition = homogeneousPosition;
        output.ClipPosition = homogeneousPosition;
        output.UV = fullscreen_helper::GetTriangleUV(vertexID);

        return output;
    }
```

Тут є нюанс, що вершинний шейдер має однорідну позицію (HomogeneousPosition) та відсічену позицію (ClipPosition). Різниця в тому, що поле, яке анатоване SV_Position в піксельному шейдері буде мати координати пікселя, а при вибірки кубічної текстури потрібно саме позиція в нормалізованому просторі пристрою з певними трансформаціями. Тому сгенерована позиція дублюється у відсічену позицію.

Далі на етапі піксельного шейдеру позиція в нормалізованому просторі пристрою за допомогою оберненої матриці добутку матриць огляду і проекції, можна отримати позицію світу, яку можна нормалізувати і отримати вектор направлення.

```
    enum : uint32_t
    {
        g_PassBufferIndex,
        g_CubeTextureIndex,
    };

    struct PassData
    {
        float4x4 InverseViewDirectionProjectionMatrix;
    };

    float4 PS_Main(fullscreen_helper::VS_Output input) : SV_Target
    {
        ConstantBuffer<PassData> passData =
ResourceDescriptorHeap[BENZIN_GET_ROOT_CONSTANT(g_PassBufferIndex)];
        TextureCube<float4> cubeMap =
ResourceDescriptorHeap[BENZIN_GET_ROOT_CONSTANT(g_CubeTextureIndex)];

        const float4 worldPosition = mul(
            input.ClipPosition,
            passData.InverseViewDirectionProjectionMatrix
        );
        const float3 direction = normalize(worldPosition.xyz);
```

```
        return cubeMap.Sample(common::g_LinearWrapSampler, direction);
    }
```

Таким чином використовуючи даний метод можна намалювати довкілля не використовуючи геометрію, а використати тільки індекс вершини.

3.5 Результати рендереру

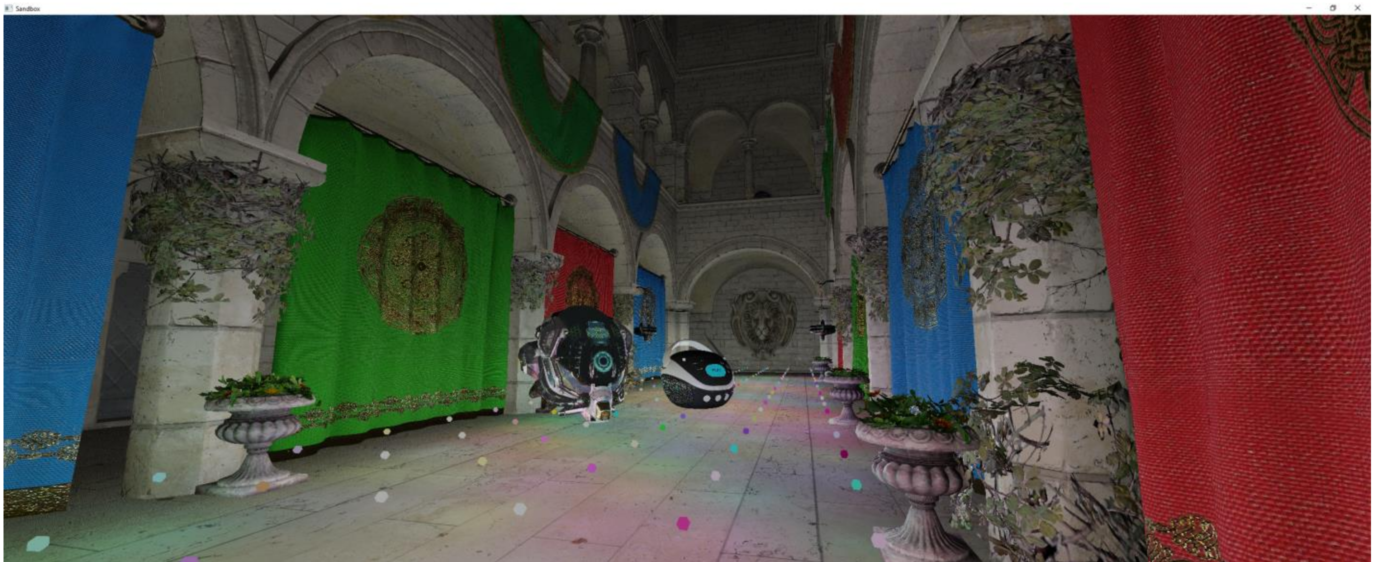


Рисунок 3.5 – Приклад фінального зображення рендереру

На рисунку 3.5 можна побачити 3 моделі на сцені: замок (262267 трикутника), шолом (15452 трикутника) та бумбокс (6036 трикутника). Також на сцені присутні 192 точкових джерела освітлення, які освітлюють сцену в реальному часі, тобто при зміні положення моделей чи зміні кольору джерела світла зміни фінальгого зображення будуть миттєвими. Позаду замку є довкілля, але воно не малюється, оскільки замок повністю його перекриває.

Виміри проводились на 2 відеокартах та на роздільній здатності 1920 на 1080:

– використовуючи AMD RX 6800 середній показник FPS буде складати не менше 190 (що відповідає часу рендеринга одного кадра 5.263 ms);

– використовуючи Nvidia GeForce RTX 3060 середній показник FPS буде складати не менше 112 (що відповідає часу рендеринга одного кадра 8.928 ms).

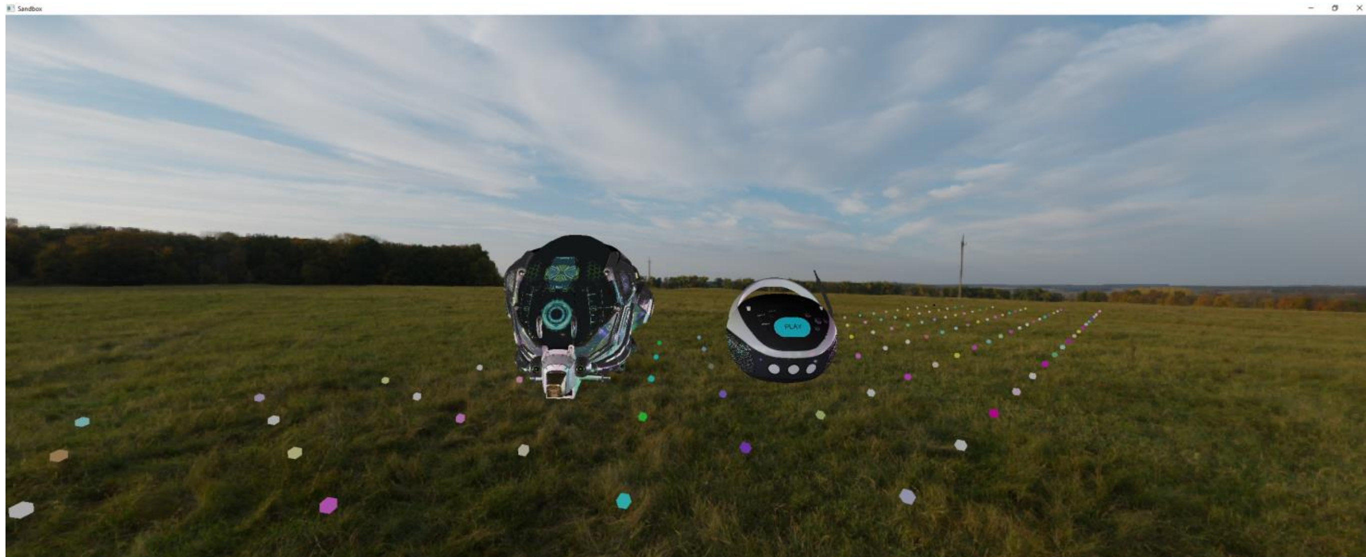


Рисунок 3.6 – Приклад фінального зображення рендереру разом з довкіллям

На рисунку 3.6 можна побачити тільки шолом, бумбокс та так само 192 точкових джерела освітлення. Наразі можна побачити довкілля.

Виміри проводились на 2 відеокартах та на роздільній здатності 1920 на 1080:

– використовуючи AMD RX 6800 середній показник FPS буде складати не менше 1300 (що відповідає часу рендеринга одного кадра 0.769 ms);
– використовуючи Nvidia GeForce RTX 3060 середній показник FPS буде складати не менше 430 (що відповідає часу рендеринга одного кадра 2.325 ms).

Якщо порівнювати перший приклад з другим видно, що обробка замку доволі трудомістка задача. Але дані приклади вдачно показуть параралельну роботу відеокарти, ґрунтуючись на кількості трикутників та на кількості оброблених пікселів.

Рисунок 3.7 – Приклад фінального зображення з малою кількістю джерел освітлення

Якщо залишити тільки 6 точкових джерел освітлення, то маємо наступні результати:

– використовуючи AMD RX 6800 середній показник FPS буде складати не менше 800 (що відповідає часу рендеринга одного кадра 1.25 ms);
– використовуючи Nvidia GeForce RTX 3060 середній показник FPS буде складати не менше 390 (що відповідає часу рендеринга одного кадра 2.564 ms).

Це показує, що освітлення в реальному часі є одним з найдорожчих ефектів, саме тому і існують декілька підходів рендерінгу.

ВИСНОВКИ

Сьогодні DirectX 12 є одним із найсучасніших і потужних графічних API. Він розроблений для максимального використання потенціалу сучасних GPU та надає розширені можливості в графічному програмуванні. DirectX 12 пропонує низькорівневий доступ до апаратних ресурсів, що дозволяє максимально ефективно управляти ресурсами та отримувати високу продуктивність. Завдяки своїм передовим можливостям та оптимізаціям, DirectX 12 є потужним інструментом для розробки сучасних ігрових та графічних додатків.

Досліджено сучасні компоненти GPU та їх взаємодію. Були розглянуто всі етапи сучасного графічного конвеєра. Також була детально проаналізовано архітектуру останніх низькорівневих графічних API на прикладі DirectX 12 та виявлені проблеми, які вони допомагають вирішити. Отримані результати були застосовані під час розробки графічного рендерера.

Розроблений рендерер доступний для перегляду на GitHub у репозиторії [1]. Архітектура рендерера включає три модулі:

1. Низькорівневий рендерер відповідає за взаємодію з DirectX 12 API.
2. Модуль роботи з сутностями використовує підхід ECS для зручної та ефективної роботи з об'єктами. Також реалізована підтримка формату glTF для завантаження 3D-моделей, що дозволяє легко і зручно використовувати ці моделі у проєкті.
3. Високорівневий рендерер відповідає за поетапне створення кадру. В рамках рендерера реалізовано три рендер-проходи: прохід геометрії, прохід освітлення та прохід довкілля. Кожен прохід виконує свою функцію в процесі рендерінгу, що дозволяє ефективно відображати сцену з геометричними об'єктами, застосовувати освітлення та наносити довкілля на сцену.

Розроблений рендерер простий у використанні та виступає як надійна основа для розробки різноманітних технік рендерінгу. Він надає зручний інтерфейс, що спрощує процес використання рендерера та дозволяє швидко розробляти нові методи рендерінгу. Завдяки його простоті та гнучкості можна експериментувати з різними технiками та вдосконалювати рендерер відповідно до своїх потреб і вимог проєкту.

Робота має широке застосування в різних галузях комп'ютерної індустрії, таких як розробка комп'ютерних ігор, візуалізація 3D-моделей та віртуальна реальність. Робота також має значущість, оскільки вона допомагає зрозуміти принципи роботи низькорівневих графічних API, зокрема DirectX 12. Це особливо важливо при реалізації рендерера в реальному часі, де швидкість та ефективність графічного оброблення мають вирішальне значення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНННЯ

1. Розроблений низькорівневий рендерер для візуалізації 3D-сцен з використанням DirectX 12 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/navkagleb/benzin.

2. Luna F. Introduction to 3D-Game Programming with DirectX 12 / Frank Luna., 2016. – 900 с. – (Illustrated edition).

3. Akenine-Mo¨ller T. Real-Time Rendering / T. Akenine-Mo¨ller, E. Haines, N. Hoffman., 2018. – 1178 с. – (Fourth Edition).

4. Gregory J. Game Engine Architecture / Jason Gregory., 2018. – 1240 с. – (Third Edition).

5. Overview of the DXGI technology [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/windows/win32/api/_direct3ddxgi/.

6. Direct3D-12 programming guide [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide.

7. DirectX Specs [Електронний ресурс] – Режим доступу до ресурсу: https://microsoft.github.io/DirectX-Specs/.

8. DirectXMath [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/windows/win32/dxmath/directxmath-portal.

9. DirectX Shader Compiler [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/microsoft/DirectXShaderCompiler/wiki.

10. Microsoft DirectX 12 Graphics Samples [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/microsoft/DirectX-Graphics-Samples.

11. Документація glTF [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/KhronosGroup/glTF.

12. tinygltf [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/syoyo/tinygltf.

13. Comparing Forward, Deferred, and Forward+ rendering algorithms using DirectX 11. [Електронний ресурс] – Режим доступу до ресурсу: https://www.3dgep.com/forward-plus/.

14. Yuksel C. Introduction to Computer Graphics Course. University of Utah [Електронний ресурс] / Cem Yuksel – Режим доступу до ресурсу: https://youtu.be/vLSphLtKQ0o.

15. Yuksel C. Interactive Computer Graphics Course. University of Utah [Електронний ресурс] / Cem Yuksel – Режим доступу до ресурсу: https://youtu.be/UVCuWQV_-Es.

16. Physically Based Rendering in Filament [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://google.github.io/filament/Filament.html.

17. Learning D3D12 from D3D11 – Part 2: GPU Resources, Heaps, and Descriptors [Електронний ресурс] – Режим доступу до ресурсу: https://alextardif.com/D3D11To12P2.html.

18. Premake [Електронний ресурс] – Режим доступу до ресурсу: https://premake.github.io/.

19. RenderDoc [Електронний ресурс] – Режим доступу до ресурсу: https://renderdoc.org/.

20. magic_enum [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Neargye/magic_enum.

21. Announcing HLSL Shader Model 6.6 [Електронний ресурс] – Режим доступу до ресурсу: https://devblogs.microsoft.com/directx/hlsl-shader-model-6-6/.

22. HLSL Shader Model 6.6 [Електронний ресурс] – Режим доступу до ресурсу: https://youtu.be/5rYBLjUmGkA.

23. Reader Question Answered 1 – Learning D3D12 [Електронний ресурс] – Режим доступу до ресурсу: https://www.jendrikillner.com/post/d3d12-learning-plan/.

24. How to load an HDRI Panorama as a Cubemap in OpenGl [Електронний ресурс] – Режим доступу до ресурсу: https://www.turais.de/how-to-load-hdri-as-a-cubemap-in-opengl/.

25. GSN Composer. Shaders Monthly [Електронний ресурс] – Режим доступу до ресурсу: https://youtu.be/mJOqVeiLOf0.

26. Secrets of Direct3D-12: Resource Alignment [Електронний ресурс] – Режим доступу до ресурсу: https://asawicki.info/news_1726_secrets_of_direct3d_12_resource_alignment.