Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Модель обробки зображень на основі API Vulkan
(тема)

Виконав:
студент 2 курсу, групи *СКСм-20-1*
Ворона І.О.
(прізвище, ініціали)

Спеціальність
123 Комп'ютерна інженерія
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Спеціалізовані комп'ютерні системи
(повна назва освітньої програми)
Керівник Чумаченко С.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри АПОТ _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти другий (магістерський)

Спеціальність 123 Комп'ютерна інженерія
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Спеціалізовані комп'ютерні системи
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Ворона Ілля Олегович
(прізвище, ім'я, по батькові)

1. Тема роботи Модель обробки зображень на основі API Vulkan

затверджена наказом університету від «_04_» __11__2021 р. № 1635Ст_

2. Термін подання студентом роботи до екзаменаційної комісії ____ _____ 20__ р.

3. Вихідні дані до роботи : вимоги до системи;теоретичні відомості, OpenGl ES, Vulkan,

4. Перелік питань, що потрібно опрацювати в роботі

Аналіз предметної області та постановка задачі.

Розробка програмної моделі

Інструменти релізації

Аналіз отриманих результатів

Створення пояснювальної записки

5. Перелік графічного матеріалу

13 слайдів

6. Дата видачі завдання 13.09.2021 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи (проекту)	Термін виконання етапів роботи (проекту)	Примітка
1	Видача теми проекту, узгодження і затвердження теми		
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів		
3	Проєктування моделі		
4	Оптимізація графічного конвеєру		
5	Розробка операційного інтерфейсу		
6	Оформлення пояснювальної записки		
7	Оформлення графічного матеріалу		
8	Захист проекту		

Студент _____
(підпис)

Керівник проекту _____ Чумаченко С.В.
(підпис) (прізвище, ім’я, по батькові, посада, звання)

« » _____ 2021р.

РЕФЕРАТ

Пояснювальна записка містить 52 сторінки, 8 рисунків, 3 таблиці, 15 джерел за переліком посилань.

ДЕМОНСТРАЦІЙНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, 3D ГРАФІКА, MESH, OPENGL, VULKAN

Задачею даного кваліфікаційного проекту є аналіз актуальних засобів роботи з графічними драйверами для відображення та маніпуляції з тривимірними сценами, а також розгляд існуючих аналоги та рішень, які необхідні для відображення та обробки сцен на процесорних GPU.

Метою кваліфікаційної роботи є створення моделі для роботи та відображення сцен та графічних 3D-об'єктів для вбудованих систем на базі технологій OpenGL та Vulkan, а також робота з математичними засобами, які необхідні для обробки сцен та графічних 3D-об'єктів, розробка програмного демонтстраційного додатку роботи даної моделі.

Результатом кваліфікаційної роботи є розроблена модель для відображення графічних примітивів із використанням платформо-залежного графічного API для роботи з дисплейним буфером.

ABSTRACT

The explanatory note contains 52 pages, 8 figures, 3 tables, 15 sources according to the list of links.

DEMONSTRATION FIRMWARE EQUIPMENT, 3D GRAPHICS, MESH, OPENGL, VULKAN

The aim of this qualification project is to analyze the current tools for working with graphics drivers for displaying and manipulating three-dimensional scenes, as well as consideration of existing analogues and solutions needed to display and process scenes on GPU processors.

The purpose of the qualification work is to create a model for working and displaying scenes and graphic 3D-objects for embedded systems based on OpenGL and Vulkan technologies, as well as working with mathematical tools needed to process scenes and graphic 3D-objects, software development demonstration application of this model.

The result of the qualification work is a developed model for displaying graphics primitives using a platform-dependent graphics API for working with the display buffer.

ЗМІСТ

ВСТУП

Сучасні обчислювальні потужності збільшуються кожен день, створюються нові вимоги та практики у галузі вбудованих рішень та графічних застосунків. Графічний та програмний інтерфейс змінюється та актуалізуються для суйчасних реалій. Нові глобальні API надають нові можливості та інтрументи для роботи із графічними девайсами та графікої. Такі API як Vulkan мають підтримку базового інтерфейсу для безпосередньої роботи з графічними процесорами.

Сучасний функціонал API та постійне обновлення стандартизація інтерфейсу графічних процесорів, під майже щорічні оновлення, надає можливість створювати спеціалізовані програми для роботи з графікою з максимальним рівнем оптимізація, а такі технології як GLFW дають змогу зробити кросс платформений інтерфейс для таких операційних систем як Windows, Unix, Mac OS.

3D-моделювання активно зростає та знаходить застосунки у великій кількості областей, таких як проектування, ігри, медицина, аналітика та багато інших. У сьогодняшній час надзвичально актуальною темою є створення спеціалізованих моделей обробки та аналізу зображень, які могли би надати легкий у використанні інтерфейс та високоефективну імплементацію для знаження ціни необхідного обладання.

Метою даної кваліфікаційної роботи є створення кросс платформеної моделі для високоефективної обробки 3D-сцен та 3D-моделей, яка має змогу використовуватись на вбудовани системах з мінімальними графічним процесором та аналіз результатів роботи моделі, порівняння ефективності реалізації імплементації із використанням різних API.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Загальні відомості про Vulkan API

Vulkan розроблено як кросплатформна абстракція над графічними процесорами. Проблема більшості цих API полягає в тому, що епоха, в яку вони були розроблені, мала графічне обладнання, яке здебільшого обмежувалося фіксованою функціональністю, що налаштовується. Програмісти повинні були надати дані про вершини в стандартному форматі і були на милість виробників графічних процесорів щодо параметрів освітлення та затінення. У міру розвитку архітектури відеокарт вони почали пропонувати все більше програмованих функцій. Всю цю нову функціональність потрібно було якимось чином інтегрувати з існуючими API. Це призвело до неідеальних абстракцій і великої кількості припущень на стороні графічного драйвера, щоб зіставити наміри програміста з сучасними графічними архітектурами. Ось чому існує так багато оновлень драйверів для підвищення продуктивності в іграх, іноді зі значними відривами. Через складність цих драйверів розробникам додатків також доводиться мати справу з невідповідністю між постачальниками, такими як синтаксис, прийнятний для шейдерів. Окрім цих нових функцій, за останнє десятиліття також спостерігається приплив мобільних пристроїв із потужним графічним обладнанням. Ці мобільні графічні процесори мають різну архітектуру залежно від потреби в енергії та просторі. Одним із таких прикладів є рендеринг із плитками, який виграє від покращення продуктивності, пропонуючи програмісту більше контролю над цією функціональністю. Іншим обмеженням, що походить від віку цих API, є обмежена підтримка багатопотокової роботи, що може призвести до вузького місця на стороні ЦП.

Vulkan вирішує ці проблеми, розробляючи з нуля для сучасних графічних архітектур. Це зменшує накладні витрати драйвера, дозволяючи програмістам чітко вказувати свої наміри за допомогою більш докладного API, а також дозволяє кільком потокам створювати та подавати команди паралельно. Це зменшує невідповідності під час компіляції шейдерів шляхом перемикання на стандартизований формат байтового коду за допомогою одного компілятора. Нарешті, він визнає загальні можливості обробки сучасних відеокарт, об'єднуючи графічні та обчислювальні функції в єдиний API [1].

Таблиця 1.1 – версії API Vulkan для графічних адаптерів

Графічний адаптер	Версія Vulkan API	Остання версія драйверу
GeForce GTX 1060	1.2.178	465.42.0.0
TITAN V	1.2.178	465.42.0.0
GeForce GTX 1070	1.2.164	461.92.0.0
GeForce GTX 1080	1.2.162	461.92.0.0
Intel(R) HD Graphics 530	1.2.196	101.11.21
GeForce RTX 2080	1.2.162	462.59.0.0
GeForce GTX 970	1.2.155	461.92.0.0
GeForce GTX 1080 Ti	1.2.155	461.40.0.0

Програма Vulkan починається з налаштування Vulkan API через VkInstance. Об'єкт створюється описом вашої програми та будь-яких розширень API, які ви будете використовувати. Після створення екземпляра ви можете запитати обладнання, яке підтримує Vulkan, і вибрати один або кілька VkPhysicalDevices для використання для операцій. Ви можете запитати такі властивості, як розмір VRAM і можливості пристрою, щоб вибрати потрібні пристрої, наприклад, віддати перевагу використанню виділених графічних карт.

1.2 Графічний конвеєр

Графічний конвеєр у Vulkan налаштовується шляхом створення об’єкта VkPipeline. Він описує конфігурований стан графічної карти, такий як розмір області перегляду та роботу буфера глибини, а також програмований стан за допомогою об’єктів VkShaderModule. Об'єкти VkShaderModule створюються з байт-коду шейдера. Драйвер також повинен знати, які цілі візуалізації будуть використовуватися в конвеєрі, який ми вказуємо, посилаючись на проходження візуалізації. Однією з найбільш відмінних особливостей Vulkan порівняно з існуючими API є те, що майже всю конфігурацію графічного конвеєра потрібно налаштувати заздалегідь. Це означає, що якщо користувач хоче переключитися на інший шейдер або трохи змінити розмітку вершин, то необхідно повністю відтворити графічний конвеєр. Це означає, що програмісту доведеться заздалегідь створити багато об’єктів VkPipeline [2] для всіх різних комбінацій, необхідних для операцій візуалізації. Динамічно можна змінювати лише деяку базову конфігурацію, розмір області перегляду та чіткий колір. Весь стан також потрібно чітко описати, наприклад, не існує стану змішування кольорів за замовчуванням. Виконуючи еквівалент завчасної компіляції порівняно з компіляцією «точно вчасно», є більше можливостей для оптимізації драйвера, а продуктивність під час виконання стає більш передбачуваною, оскільки великі зміни стану, такі як перехід на інший графічний конвеєр, робляться дуже чіткими.

1.3 Характеристики графічного АПІ OpenGL

У минулому розробка у безпосередньому режимі (конвеєр фіксованої функції) передбачалась при використані OpenGL. Через свою відносну простоту це був стандартний метод для відмальовування графіки. Однак це передбачало приховування великого відсотку доступного функціоналу від

розробника у глибині бібіліотеки, що унеможливлювало повноцінний контроль над процесом зі сторони програміста. З подальшим розвитком технологій користувачи потребували збільшення можливостей API, що було набуто з часом, а специфікації у стали більш гнучкими. Розробники отримали здатність контролювати обчислення відносно своєї графіки.

Конвеєрний, або безпосередній, режим (рис. 1.1) – це простий у використанні , але малоефективний. Саме тому специфікація почала скорочувати функціональні можливості безпосереднього режиму з версії 3.2, а згодом і зовсім його позбулася та почала підштовхувати програмістів працювати в режимі основного профілю OpenGL, який видалив весь застарілий функціонал та можливості у специфікації OpenGL.

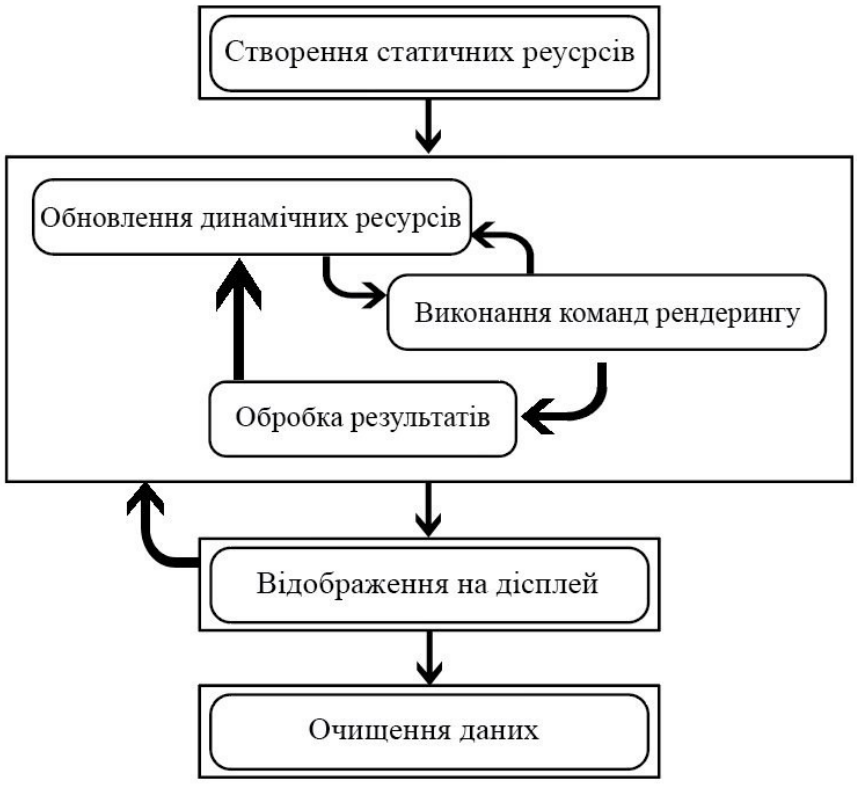

Рисунок 1.1 – Конвеєр OpenGL

OpenGL є найбільш поширеним API 2D і 3D-графіки в галузі, що забезпечує тисячі додатків на найрізноманітніших комп’ютерних платформах. Він незалежний від віконної системи та операційної системи, а також прозорий

для мережі. OpenGL дає змогу розробникам програмного забезпечення для ПК, робочих станцій і суперкомп'ютерного обладнання створювати високопродуктивні, візуально привабливі графічні програмні програми на таких ринках, як САПР, створення контенту, енергетика, розваги, розробка ігор, виробництво, медичне обслуговування та віртуальна реальність. OpenGL надає всі можливості новітнього графічного обладнання.

У OpenGL надається базова бібліотека функцій для визначення графічних примітивів [3], атрибутів, геометричних перетворень, перетворення перегляду та багатьох інших операцій. OpenGL розроблен, щоб бути апаратно незалежним, тому багато операцій, таких як процедури введення та виведення, не включені в основну бібліотеку. Проте процедури введення та виведення та багато додаткових функцій доступні в допоміжних бібліотеках, які були розроблені для програм OpenGL.

На додаток до базової (основної) бібліотеки OpenGL існує ряд пов'язаних бібліотек для обробки спеціальних операцій. Утиліта OpenGL (GLU) надає підпрограми для налаштування матриць перегляду та проекції, опису складних об'єктів з наближеннями ліній і багатокутників, відображення квадрик і B-сплайнів за допомогою лінійних наближень, обробки операцій відтворення поверхні та інших складних завдань. Кожна реалізація OpenGL включає бібліотеку GLU, і всі імена функцій GLU починаються з префікса glu. Існує також об'єктно-орієнтований набір інструментів на основі OpenGL, який називається Open Inventor, який надає підпрограми та попередньо визначені форми об'єктів для інтерактивних тривимірних додатків. Цей набір інструментів написаний на C++.

OpenGL має кілька бібліотек віконної системи, які підтримують функції для різних машин. Розширення OpenGL до системи X Window (GLX) надає набір підпрограм, які мають префікс glX. Системи Apple можуть використовувати інтерфейс Apple GL (AGL) для операцій керування вікнами. Назви функцій для цієї бібліотеки мають префікс agl [4]. Для систем Microsoft

Windows підпрограми WGL забезпечують інтерфейс Windows-OpenGL. Ці підпрограми мають префікс з літерами wgl. Presentation Manager to OpenGL (PGL) — це інтерфейс для IBM OS/2, який використовує префікс pgl для бібліотечних процедур. OpenGL Utility Toolkit (GLUT) надає бібліотеку функцій для взаємодії з будь-якою системою екранних вікон. Функції бібліотеки GLUT мають префікс переповнення, і ця бібліотека також містить методи для опису та візуалізації квадратичних кривих і поверхонь

Лістинг 1.1 – Фрагмент опису примітиву

```
glBegin (GL_LINES);
      glVertex2i (180, 15);
      glVertex2i (10, 145);
glEnd ( );
```

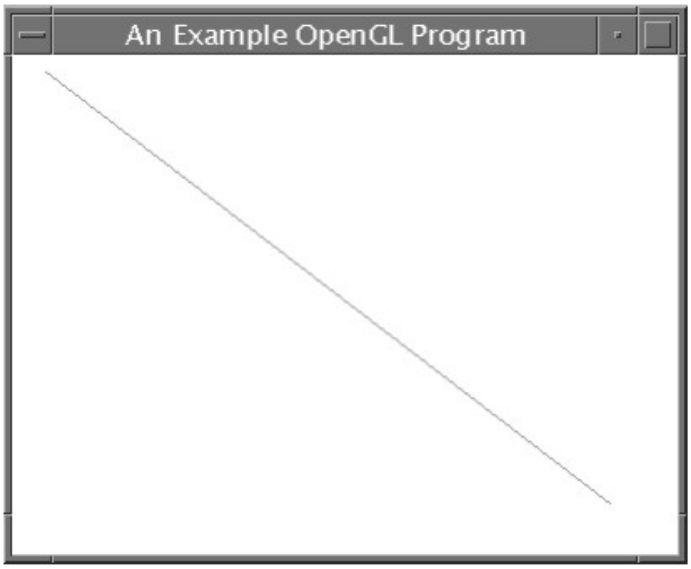

Рисунок 1.1 – Приклад базового вікна на OpenGL

1.4 Порівняння та аналіз графічних драйверів низького та високого рівня

В результаті роботи було виявлено деякі основні ключові відмінності між OpenGL і Vulkan:

І OpenGL, і Vulkan є міжплатформним API з відкритим вихідним кодом, що означає, що обидва безкоштовні для використання з великою кількістю хороших функцій, і не потрібно турбуватися про будь-яку підписку, плани та час щодо використання будь-кого з них. Обидва API розроблені одним і тим самим розробником, та Vulkan був представлений в 2015 році як неприбутковий API групою Khronos в GDC, і спочатку він іменувався як «ініціатива OpenGL наступного покоління» або OpenGL next, але пізніше його змінили на Vulkan.

Vulkan надає низку переваг, оскільки пропонує потужний прямий контроль над графічним процесором, зменшує використання ЦП, а також зменшує навантаження на нього. Його загальна концепція та функції схожі на Mantle, який також був прийнятий у Direct3D 12 Microsoft Windows і Metal Apple. OpenGL генерує буфери команд для кількох потоків і одночасну їх обробку в конвеєр команд, і через цю причину розробнику не потрібно працювати для підтримки фреймворка, або якщо він хоче це робити, то він може зробити це з використанням невеликих зусиль.

OpenGL створює свій власний компілятор для GLSL, який є його мовою високого рівня, і ця мова пише шейдери, які змушують драйвер OpenGL реалізувати свій власний компілятор для цієї мови та виконувати додатки у режимі рантайму для переклада шейдерів програми в машинний код GPU. Тоді як у Vulkan має шейдери, які вже переведені в проміжний двійковий формат, який називається SPIR-V (стандартне переносне проміжне представлення).

Vulkan має кращу інтеграцію інструментів у порівнянні з OpenGL, оскільки є можливість ввімкнути перевірку та діагностику шарів незалежно один від одного. Через відсутність жорсткої різниці у API між версіями обох програм для мобільних пристроїв і ПК перенесення ігор між цими платформами є значно легшим, у порівняння з OpenGL[5].

У майбутньому в OpenGL будуть додані нові розширення, які є такими ж, як і у Vulkan, і це такі розширення як NV_command_list, що відповідає

«парадигмі живлення графічного процесора» Vulkan, і це покращить ігрові можливості OpenGL. Оскільки Vulkan представився як наступне покоління API OpenGL, він буде включати більше функцій, такі як покращені методики для виконання різних типів завдань, пов'язаних із його сферою, для досягнення найкращих результатів у різних проектах. Тож є можливість легко досліджувати функції Vulkan через OpenGL[6].

На основі теоретичних відомостей та наведених вище ключових відмінностей було проаналізовано та сформовано наступну порівняльну характеристику OpenGL та Vulkan технологій, яка наведена у таблиці 1.2.

Таблиця 1.2 – Порівняльна характеристика OpenGL та Vulkan

	OpenGL	Vulkan
Визначення	Це відкритий і міжплатформний API, який працює для візуалізації 2D і 3D векторної графіки.	Це кросплатформний API, який працює як для програмування відеоігор, так і для 3D-графіки для досягнення ряду хороших результатів у відповідних завданнях.
Розробник	Silicon Graphics Inc. розпочала розробку цього API в 1991 році і випустила його 30 червня 1992 року, але його розробником була група Khronos, яка раніше була відома як ARB.	Розробником цього API була група AMD, DICE і Khronos, яка спочатку випустила його в лютому 2016 року.
Операційна система	Цей API може взаємодіяти з операційними системами такими як, Linux, Microsoft Windows, Mac OS.	Vulkan може працювати на різних операційних системах, таких як Linux, Android, Unix, Microsoft Windows, Nintendo, BSD, Mac OS, iOS та багатьох інших операційних системах, з якими він сумісний.
Найновіша версія	31 липня 2017 року була випущена його остання версія,	Остання версія була випущена 1 березня 2021 року з низкою

	названа як 4.6 з великою кількістю позитивних функцій і покращенням drag bag технології для перетягування попередніх версій.	оновлень, що робить його роботу більш гладкою, і це була 1.2.171.
Доступність	Можно почати працювати з OpenGL і отримати його, відвідавши його офіційний веб-сайт www.opengl.org	На ресурсі www.khronos.org/vulkan, множна завантажити дане API, а також отримати іншу інформацію, таку як основні вимоги, необхідні речі, тощо.
Язики, які підтримуються	C або C++ – це комп'ютерні мови, на яких написаний OpenGL і полегшує роботу з ним.	C є основною мовою цього програмного забезпечення, тобто воно написано цією комп'ютерною мовою.

1.5 Існуючі техніки оптимізацій графічного конвейеру

Протягом останніх кількох років конвеєр рендеринга з апаратним прискоренням швидко збільшувався у складності, приносячи з собою все більш складні та потенційно заплутані характеристики продуктивності. Основним методом збільшеня продуктивности було зниження кількості внутрішніх циклів процессору на етапі рендерингу. Цей цикл ідентифікації та оптимізації є фундаментальним для налаштування гетерогенної багатопроцесорної системи; Основна ідея полягає в тому, що конвеєр, за визначенням, настільки швидкий, наскільки швидка його найповільніша стадія. Таким чином, якщо передчасна і нецілеспрямована оптимізація в однопроцесорній системі може призвести лише до мінімального підвищення продуктивності, то в багатопроцесорній системі така оптимізація дуже часто призводить до нульового приросту.

Конвеєр на найвищому рівні можна розбити на дві частини: ЦП і ГП. На рисунку 1.2 показано, що всередині графічного процесора існує ряд

функціональних блоків, які працюють паралельно, які, по суті, діють як окремі процесори спеціального призначення, а також ряд місць, де може виникнути критична секція. До них належать вибірка вершин і індексів, затінення вершин (перетворення та освітлення, або T&L), затінення фрагментів і растрові операції (ROP).

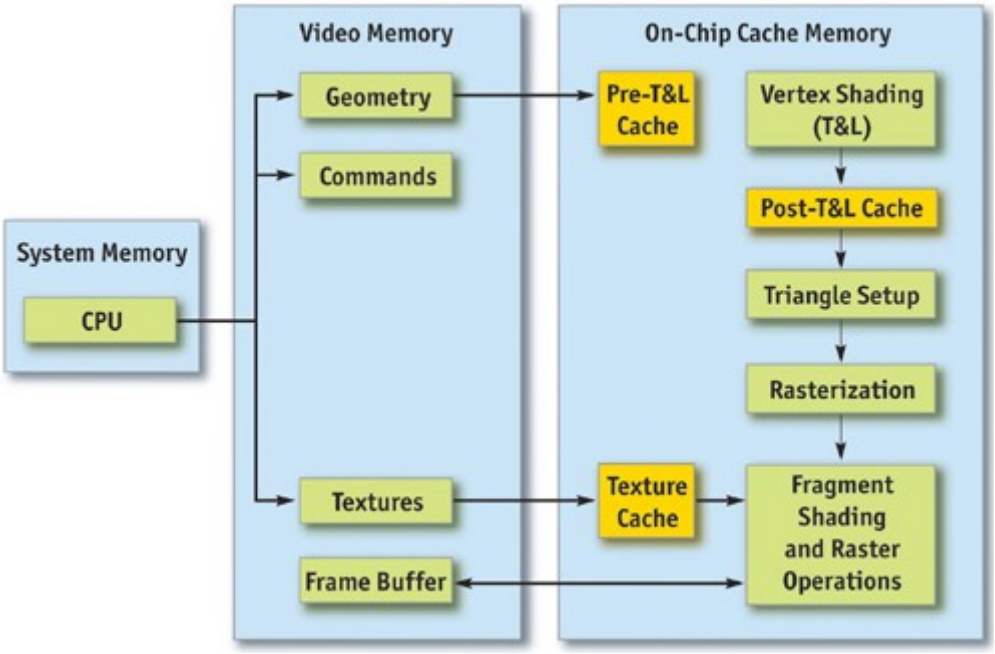

Рисунок 1.2 – Модель графічного конвеєра

1.6 Проектування моделі кросс платформеного оптимізованого графічного конвейеру

Для створення зручного інтерфейсу для взаємодії з графічними API різних платформ та операційних систем використовується GLFW. GLFW — це безкоштовна багатоплатформна бібліотека з відкритим кодом для OpenGL, OpenGL ES та розробки програм Vulkan. Він надає простий, незалежний від платформи API для створення вікон, контекстів і поверхонь, читання введених даних, обробки подій та інших операцій. Сам GLFW потребує лише заголовків і бібліотек для ОС і віконної системи. Модуль не потребує заголовки для

створення контексту (WGL, GLX, EGL, NSGL, OSMesa) або API візуалізації (OpenGL, OpenGL ES, Vulkan).

Лістинг 1.2 – Фрагмент обробника подій на GLFW

```
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
      if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
            glfwSetWindowShouldClose(window, GLFW_TRUE);
}
```

1.7 Існуючі алгоритми обробки зображень з використанням прискорення API Vulkan

Маючи геометричний каркас об’єкту модель може накласти мапінг на його вершини. Мапінг використовується для того щоб сопоставити вершини текстури з їх кольорами що робить модель текстури схожу на реальний об’єкт. Це також дозволить завантажувати та малювати основні 3D-моделі.

Щоб додати обробку текстур до загальної моделі потрібно имплементувати настпуні кроки:

Створити об’єкт текстури для зберігання у пам’яті девайсу

Накласти мапу пикселей із текстури на обьект зображення у пам’яті

Створити семплер зображення для подальшого його використання у модели

Створення зображення та наповнення його даними схоже на створення буфера вершин. Починаючи зі створення проміжного ресурсу та заповнення його даними пікселів, а потім копіюємо це в кінцевий об’єкт зображення, який буде використовуватися для візуалізації. Для цієї мети можна створити проміжний образ, Vulkan також дозволяє копіювати пікселі з VkBuffer до зображення, і API для цього швидше якщо графічний адаптер оптимізован для

цієї обробки. Модель створює цей буфер і заповнимо його значеннями пікселів, а потім створює зображення для копіювання пікселів. Створення схоже на створення буферів. Він включає запити про алокації до пам’яті, виділення пам’яті пристрою та її прив’язування.

Оптимізація без належної ідентифікації вузьких місць є причиною великих витрат на розробку, тому відбувається формалізація процесц у фундаментальний цикл ідентифікації та оптимізації:

Визначення вузького місце. Для кожного етапу конвеєра потрібно змунювати його робоче навантаження або його обчислювальну здатність (тобто тактову частоту). Якщо продуктивність змінюється, то це звузьке місце.

Оптимізування. З огляду на вузьке місце, можна зменшити його робоче навантаження, доки продуктивність не перестане покращуватися або поки не досягнете бажаного рівня продуктивності.

Повторення. Потрібно повторювати кроки 1 і 2, поки не буде досягнутий бажаний рівень продуктивності.

Знайти вузьке місце — це половина справи в оптимізації, оскільки воно дає змогу приймати розумні рішення щодо зосередження ваших реальних зусиль з оптимізації. На рисунку 1.3 показана блок-схема, що зображує серію кроків, необхідних для визначення точного вузького місця у програмних додатках.

Потрібно зауважити, що ми початок знаходиться на задньому кінці конвеєра, з операціями з буфером кадрів (також званими растровими операціями) і закінчується на ЦП. Також будь-який примітив (зазвичай трикутник), за визначенням, має одне вузьке місце, протягом кадру вузьке місце, швидше за все, змінюється. Таким чином, зміна робочого навантаження на більш ніж одній стадії конвеєра часто впливає на продуктивність. З цієї причини часто допомагає змінювати навантаження на основі об’єкта за об’єктом або від матеріалу до матеріалу.

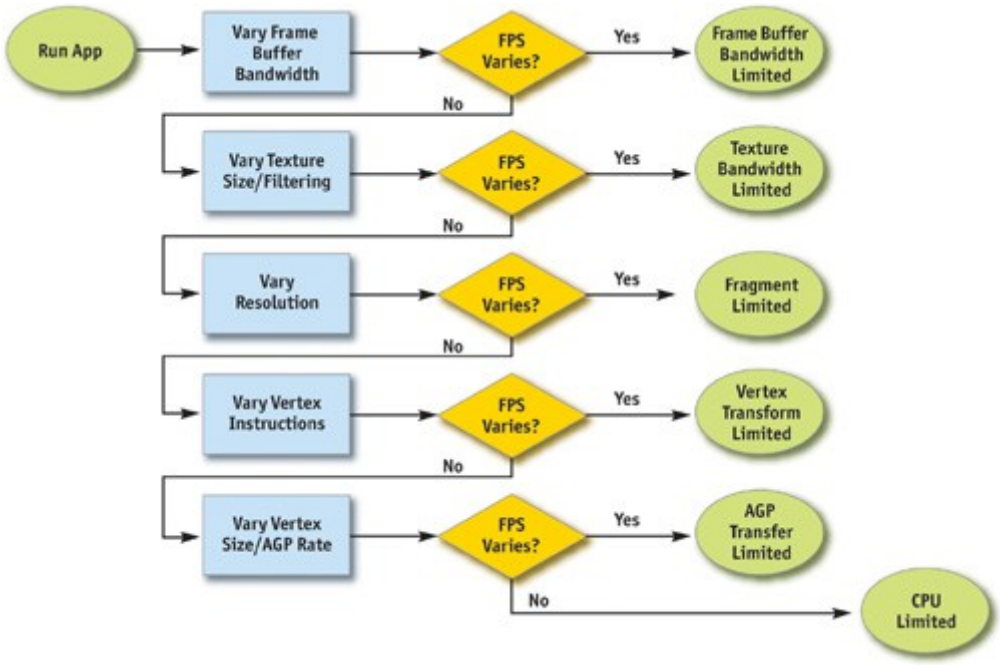

Рисунок 1.3 – Модель вузького місця

Для кожного етапу конвеєра також потрібно згадувати тактовий режим графічного процесора, до якого він прив'язаний (тобто ядро або пам'ять). Ця інформація корисна в поєднанні з такими інструментами, як PowerStrip (EnTech Taiwan 2003), що дозволяє зменшити відповідну тактову частоту та спостерігати за змінами продуктивності додатку

Графічний конвеєр у контексті растрової графіки.

Самий задній кінець конвеєра, растрові операції (часто званий ROP), відповідає за глибину читання та запису та трафарет, порівняння глибини та трафарету, читання та запис кольору, а також виконання альфа-змішування та тестування. Як бачите, велика частина робочого навантаження ROP обкладає доступну пропускну здатність кадрового буфера.

Найкращий спосіб перевірити, чи обмежена програма пропускною здатністю кадру-буфера, - це змінити бітову глибину кольору або буфери глибини, або обидва. Якщо зменшення бітової глибини з 32-розрядної до 16-розрядної значно покращує продуктивність, то є, безсумнівно, обмеження пропускною здатністю кадру до буфера.

Пропускна здатність кадрового буфера є функцією тактової частоти пам'яті графічного процесора, тому зміна тактових частот пам'яті є ще одним методом, який допомагає виявити це вузьке місце..

Пропускна здатність текстури

Пропускна здатність текстури споживається кожного разу, коли запит на вибірку текстури надходить у пам'ять. Незважаючи на те, що сучасні графічні процесори мають кеш текстур, призначений для мінімізації запитів сторонньої пам’яті, вони, очевидно, все ще виникають і споживають достатню кількість пропускної здатності пам’яті.

Зміна форматів текстур може бути складнішою, ніж зміна форматів буфера кадрів, як було виконано під час перевірки ROP; натомість можна змінити ефективний розмір текстури, використовуючи велику кількість позитивного зміщення рівня деталізації (LOD) mipmap. Завдяки цьому вибірки текстур мають доступ до дуже грубих рівнів піраміди mipmap, що ефективно зменшує розмір текстури. Якщо ця модифікація спричиняє значне покращення продуктивності, ви обмежені пропускною здатністю текстури.

Пропускна здатність текстури також є функцією частоти пам'яті графічного процесора.

Затінення фрагментів

Затінення фрагмента відноситься до фактичних витрат на створення фрагмента з відповідними значеннями кольору та глибини. Це вартість запуску «піксельного шейдера» або «фрагментного шейдера». Слід зауважити, що затінення фрагментів і пропускна здатність кадрового буфера часто об’єднуються під швидкістю заповнення заголовка, оскільки обидва є функцією роздільної здатності екрана. Однак вони є двома різними етапами в конвеєрі, і здатність розрізняти їх має вирішальне значення для ефективної оптимізації.

До появи високопрограмованих графічних процесорів з обробкою фрагментів рідко було пов’язане затіненням фрагментів. Часто саме пропускна здатність кадрового буфера спричиняла неминучу кореляцію між роздільною

здатністю екрана та продуктивністю. Цей маятник тепер починає розгойдуватися до фрагментного затінення, оскільки нова гнучкість дозволяє розробникам витрачати масу циклів на створення модних пікселів.

Першим кроком у визначенні того, чи є затінення фрагментів вузьким місцем, є просто змінити роздільну здатність. Оскільки пропускна здатність вже була виключена з кадрового буфера, спробувавши різну бітову глибину кадрового буфера, якщо налаштування роздільної здатності спричиняє зміну продуктивності, виною, швидше за все, є затінення фрагментів. Додатковим підходом було б змінити довжину ваших фрагментних програм і перевірити, чи впливає це на продуктивність [7]. Але потрібно пом’ятати, щоб не додавати інструкції, які можна легко оптимізувати за допомогою розумного драйвера пристрою.

Швидкість затінення фрагментів є функцією тактової частоти ядра графічного процесора.

Вершинна обробка

Етап перетворення вершин у конвеєрі візуалізації відповідає за прийняття вхідного набору атрибутів вершин (таких як положення в просторі моделі, нормалі вершин, координати текстури тощо) і створення набору атрибутів, придатних для відсікання та растеризації (наприклад, однорідне положення кліп-простору, результати освітлення вершин, координати текстури тощо). Природно, продуктивність на цьому етапі є функцією роботи, виконаної на вершину, а також кількості вершин, що обробляються.

З програмованими перетвореннями визначити, чи є обробка вершин вашим вузьким місцем, просто змінити довжину вашої вершинної програми.

Якщо продуктивність змінюється, ви прив’язані до обробки вершин. Якщо є необхідність додати інструкції, то потрібно бути пильними та додавати лише ті інструкції, які дійсно виконують значущу роботу; в іншому випадку інструкції можуть бути оптимізовані компілятором або драйвером. Наприклад, безопераційні дії, які посилаються на постійні регістри (наприклад, додавання

константного регістра, що має нульове значення), часто не можна оптимізувати, оскільки драйвер зазвичай не знає значення константи під час компіляції програми.

Якщо використотвувати перетворення з фіксованою функцією, становиться трохи складніше. Сліб спроббуваим змінити навантаження, змінивши роботу вершин, наприклад, дзеркальне освітлення або стан генерації текстурних координат [8].

Швидкість обробки вершин є функцією тактової частоти ядра графічного процесора.

1.8 Передача вершин і індексів

Вершини та індекси витягуються GPU як перший крок у частині конвеєра GPU. Ефективність вибору вершин і індексів може змінюватися залежно від того, де розміщені фактичні вершини та індекси. Зазвичай вони знаходяться або в системній пам’яті, що означає, що вони будуть передані в графічний процесор по шині, наприклад AGP або PCI Express, або в локальній пам’яті кадрового буфера. Часто, особливо на платформах ПК, це рішення залишається за драйвером пристрою, а не за програмою, хоча сучасні графічні API дозволяють програмам надавати підказки щодо використання, щоб допомогти драйверу вибрати правильний тип пам’яті.

Визначення того, чи є вибірка вершин або індексів вузьким місцем у вашій програмі, тягне за собою зміну розміру формату вершини.

Продуктивність вибору вершин і індексів є функцією швидкості AGP/PCI Express, якщо дані поміщені в системну пам'ять; якщо дані поміщаються в локальну пам'ять кадрового буфера - це функція годинника пам'яті.

Якщо жоден з цих тестів суттєво не впливає на продуктивність, то в першу чергу має місце прив’язаність до ЦП. Можна перевірити цей факт,

підвищивши розгін ЦП: якщо продуктивність змінюється пропорційно, то йде прив’язка до ЦП.

Після визначення вузьких місць, є необхідність оптимізувати цей конкретний етап, щоб покращити продуктивність програми. Наступні поради класифікуються за стадією порушення.

Багато додатків пов’язані з процесором — іноді з поважних причин, наприклад, із складною фізикою чи штучним інтелектом, а іноді через погане групування чи керування ресурсами. Якщо було виявлено, що ваша програма прив’язана до ЦП, можна спробувати винайдені пропозиції, які наведені нижче, щоб зменшити роботу ЦП у конвеєрі візуалізації.

Щоразу, коли виконується синхронна операцію, яка вимагає доступу до ресурсу графічного процесора, існує потенційна можливість масового гальмування конвеєра графічного процесора, що коштує циклів як ЦП, так і ГП. Цикли ЦП витрачаються даремно, оскільки ЦП повинен сидіти і обертатися в циклі, чекаючи, поки (дуже глибокий) конвеєр графічного процесора простоїть і поверне запитуваний ресурс. Цикли графічного процесора потім витрачаються даремно, оскільки конвеєр простоює і його потрібно заповнювати.

Це блокування може статися в будь-який час:

- блокування або читання з поверхні, на яку ви раніше відображали;
- запис на поверхню, з якої GPU читає, наприклад текстуру або буфер вершин.

Загалом, вам слід уникати доступу до ресурсу, який GPU використовує під час візуалізації.

По-друге, збільшити максимальний розмір пакетів.

Цю пораду можна також назвати «Зменште кількість пакетів до мінімуму». Пакет — це група примітивів, які відображаються за допомогою одного виклику відтворення API (наприклад, DrawIndexedPrimitive в DirectX 9). Розмір пакету – це кількість примітивів, які він містить. Кожен виклик функції API для малювання геометрії має пов’язану вартість ЦП, тому максимізація

кількості трикутників, поданих під час кожного виклику малювання, зведе до мінімуму роботу ЦП, виконану для заданої кількості відтворених трикутників.

Було знайдено декілька способів максимально збільшити розмір пакетів:

Якщо потрібно використовувати трикутні смужки, замість них можна використати вироджені трикутники, щоб зшити неперетинаючися смуги.

Використовувати текстурні сторінки. Пакети часто розбиваються, коли різні об’єкти використовують різні текстури. Розташувавши багато текстур в одній 2D-текстурі та встановивши відповідні координати текстури, можна надіслати геометрію, яка використовує декілька текстур за один виклик малювання. Необхідно зауважити, що ця техніка може мати проблеми з mipmapping та згладжуванням. Однією з технік, яка обходить багато з цих проблем, є упаковка окремих 2D текстур у кожну грань карти куба.

Використовувати розгалуження шейдерів GPU, щоб збільшити розмір пакету. Сучасні графічні процесори мають гнучкі конвеєри обробки вершин і фрагментів, які дозволяють розгалужуватися всередині шейдера. Наприклад, якщо два пакета є окремими, тому що для однієї потрібен шейдер з чотирьох кісток, а для іншої — вершинний шейдер з двома кістками, натомість можна написати вершинний шейдер, який зациклює кількість необхідних кісток, накопичуючи ваги змішування, а потім вививає ться з циклу, коли ваги сумуються до одиниці. Таким чином, два пакета можна об’єднати в одну. В архітектурах, які не підтримують розгалуження шейдерів, подібну функціональність можна реалізувати за рахунок циклів шейдерів, використовуючи шейдер із чотирьох кісток на всьому і просто обнуляючи вагу кісток для вершин, які мають менше чотирьох впливів на кістки.

Використовувати константну пам’ять вершинного шейдера як таблицю пошуку матриць. Часто пакети розбиваються, коли багато маленьких об’єктів мають усі властивості матеріалу, але відрізняються лише станом матриці. У цих випадках можна завантажити n різних матриць у постійну пам’ять вершинного шейдера та зберігати індекси в постійній пам’яті у форматі вершин для кожного

об’єкта [9]. Потім це дозволить використовувати цей індекс для пошуку постійної пам’яті у вершинному шейдері та використовувати правильну матрицю перетворення, таким чином відображаючи n об’єктів одночасно.

Відкладати рішення якомога далі у графічний конвеєр. Швидше використовувати альфа-канал текстури як фактор блиску, а не розбивати пакет, щоб встановити константу піксельного шейдера для блиску. Аналогічно, розміщення даних затінення у текстури та вершини може дозволити подавати більші пакети.

1.9 Оптимізація обробки вершин

Передача вершин рідко є вузьким місцем у програмі, але це, безумовно, не є неможливим. Якщо передача вершин або, менш ймовірно, індексів є вузьким місцем у програмі, були винайдені наступні засоби вирішення даної «несправності» :

Використовувати якомога менше байтів у форматі вершин. Не використовувати float для всього, якщо вистачить байтів (наприклад, для кольорів).

Генерувати потенційно отримані атрибути вершин у програмі вершин замість того, щоб зберігати їх у форматі вхідних вершин.

Використовувати 16-бітові індекси замість 32-бітових індексів. 16-бітові індекси дешевше отримувати, дешевше переміщати і займають менше пам’яті.

Отримувати доступ до даних вершин у відносно послідовному порядку. Сучасні графічні процесори звертаються до кеш-пам'яті під час отримання вершин. Як і в будь-якій ієрархії пам’яті, просторова локальність посилання допомагає максимізувати звернення до кешу, таким чином зменшуючи вимоги до пропускної здатності.

Вершинна обробка рідко є вузьким місцем для сучасних графічних процесорів, але вона все одно може виникати залежно від ваших моделей використання та цільового обладнання та на це необхідно звертати увагу.

Якщо що обробка вершин є вузьким місцем у додатку можна використати наступні виявлені способи усунення недоліків:

Оптимізація для кешу вершин після T&L. Сучасні графічні процесори мають невеликий кеш-пам’ять «першим прийшов, першим вийшов» (FIFO), в якому зберігаються результати останніх перетворених вершин; попадання в цей кеш зберігає всю роботу з трансформації та освітлення, а також всю роботу, виконану раніше в конвеєрі. Щоб скористатися перевагами цього кешу, потрібно використовувати індексовані примітиви, і впорядкувати вершини, щоб максимізувати місцевість посилання на сітку [10]. Доступні інструменти, включно з D3DX і NVTriStrip (NVIDIA 2003), які можуть допомогти вам із цим завданням.

Зменшити кількість оброблених вершин. Це рідко є фундаментальним питанням, але використання простої схеми рівня деталізації, такої як набір статичних LOD, безумовно, допомагає зменшити навантаження на обробку вершин.

Використання LOD для обробки вершин. Поряд із використанням LOD для кількості оброблених вершин, слід спробувати виконати LOD обчислення вершин.

Витягнути обчислення для кожного об’єкта в ЦП. Часто для зручності у вершинному шейдері виконується обчислення, яке змінюється один раз на об’єкт або на кадр. Наприклад, перетворення вектора спрямованого світла в простір очей іноді виконується у вершинному шейдері, хоча результат обчислення змінюється лише один раз за кадр.

Використовувати правильний координатний простір. Часто вибір простору координат впливає на кількість інструкцій, необхідних для обчислення значення у програмі вершин. Наприклад, під час виконання

вершинного освітлення, якщо нормалі ваших вершин зберігаються в просторі об’єктів, а вектор світла зберігається в просторі очей, тоді доведеться трансформувати один із двох векторів у вершинному шейдері. Якби натомість вектор світла був перетворений в простір об’єктів один раз на об’єкт на ЦП, перетворення по вершинах не було б необхідним, зберігаючи інструкції вершин GPU.

Використовувати розгалуження вершин для «раннього завершення» обчислень. Якщо перебирати кілька світильників у вершинному шейдері та робите звичайне освітлення з низьким динамічним діапазоном [0..1].

1.10 Прискорення затінення фрагментів

Якщо використовувати довгі та складні фрагментні шейдери, дуже ймовірно, що йде прив’язака до фрагментного затінення. Наразі був винайден алгоритм оптимізації даного вузького місця.

Спершу можна спробувати оптимізувати глибину візуалізації. Відтворення проходу лише глибини (без кольору) перед відтворенням основних проходів затінення може значно підвищити продуктивність, особливо в сценах із високою глибиною складності, за рахунок зменшення кількості фрагментного затінення та доступу до пам’яті кадрового буфера, який необхідно виконати. Щоб отримати всі переваги проходу лише глибини, недостатньо просто вимкнути запис кольорів у буфер кадру; вам також слід вимкнути всі затінення на фрагментах, навіть затінення, яке впливає на глибину, а також колір (наприклад, альфа-тест).

Далі йде early-z оптимізація шляхом відкинення обробки фрагментів. Сучасні графічні процесори мають силікон, призначений для уникнення затінення закритих фрагментів, але ці оптимізації покладаються на знання сцени до поточного моменту; їх можна значно покращити, відтворюючи приблизно в порядку спереду-назад. Крім того, встановлення глибини спочатку

в окремому проході може суттєво пришвидшити наступні проходи (де виконується все дороге затінення), ефективно зменшуючи їх складність затіненої глибини до 1.

Наступним кроком йде зберігання складних функції в текстурах. Текстури можуть бути надзвичайно корисними як таблиці пошуку, а їх результати фільтруються безкоштовно. Канонічним прикладом тут є карта куба нормалізації, яка дозволяє нормалізувати довільний вектор з високою точністю за ціною одного пошуку текстури.

Далі рекомендовано перенести роботу за фрагментами до вершинного шейдера. Подібно до того, як роботу над об’єктом у вершинному шейдері слід перемістити в центральний процесор, обчислення по вершинах (разом із обчисленнями, які можна правильно лінійно інтерполювати в просторі екрана) слід перемістити до вершинного шейдера. Поширені приклади включають обчислювальні вектори та вектори перетворення між системами координат.

Відповідно до вказаних вище заключень, рекомендовано використовувати найнижчу необхідну точність. API, такі як DirectX 9, дозволяють вказувати точні підказки в коді фрагментного шейдера для кількостей або обчислень, які можуть працювати з меншою точністю. Багато графічних процесорів можуть скористатися цими підказками, щоб зменшити внутрішню точність і підвищити продуктивність.

Також потрібно уникати надмірної нормалізації. Поширеною помилкою є отримання «нормалізації-щасливого»: нормування кожного окремого вектора на кожному кроці під час виконання обчислень [11]. Розпізнати, які перетворення зберігають довжину, а які обчислення не залежать від довжини вектора

Використання рівня деталізації фрагментного шейдера теж є продуктивним способом оптимізації даного вузького місця. Хоча він пропонує менший ефект, ніж вершинний LOD (просто тому, що об’єкти на відстані природно самі LOD щодо обробки пікселів, завдяки перспективі), зменшуючи

складність шейдерів на відстані та зменшуючи кількість проходів через поверхні, може зменшити навантаження на обробку фрагментів.

Ще одним способом, або додатковим пунктом до вище сказаних, є коректна робота з трилінійною фільтрацією, а саме вимкнення її там, де це не потрібно. Трилінійна фільтрація, навіть якщо не споживає додаткову пропускну здатність текстур, вимагає додаткових циклів для обчислення у фрагментному шейдері на більшості сучасних архітектур GPU.

Також, як показала практика, слід використовувати найпростіший тип шейдера. Як в Direct3D, так і в OpenGL існує ряд різних способів затінювання фрагментів. Наприклад, у Direct3D 9 можна вказати фрагментне затінення, використовуючи, у порядку збільшення складності та потужності, стани етапу текстури, піксельні шейдери версії 1.x (ps.1.1 – ps.1.4), піксельні шейдери версії 2.x. або піксельні шейдери версії 3.0. Загалом, слід використовувати найпростіший тип шейдера, який дозволяє створити задуманий ефект. Простіші типи шейдерів пропонують ряд неявних припущень, які часто дозволяють компілювати їх драйвером графічного процесора до швидшого рідного коду обробки пікселів. Приємним побічним ефектом є те, що ці шейдери працюватимуть на більш широкому спектрі обладнання.

Оскільки потужність і програмованість зростають у сучасних графічних процесорах, збільшується складність вилучення кожного елемента продуктивності з машини. Незалежно від того, чи є мета покращити продуктивність повільної програми чи шукати області, де можна покращити якість зображення «безкоштовно», потрібне глибоке розуміння внутрішньої роботи графічного конвеєра. Оскільки конвеєр GPU продовжує розвиватися, фундаментальні ідеї оптимізації все ще будуть застосовуватися: спочатку необхідно визначити вузьке місце, змінюючи навантаження або обчислювальну потужність кожного блоку; потім систематично атакувати ці вузькі місця, використовуючи своє розуміння того, як поводиться кожен блок трубопроводу.

2 ПРОЕКТУВАННЯ МОДЕЛІ З ВИКОРИСТАННЯМ API VULKAN

2.1 Компоненти моделі

Загальна модель описує усі кроки від створення процессу операційної системи та вибору графічного драйверу до обробки та відправлення комманд конвеєру. Модель обробки та редерінгу можно поділити на етапи опису фундаментальних механізмів роботи Vulkan.

2.1.1 Створення екземпляра моделі та фізичного пристрою

Модель Vulkan починається з налаштування Vulkan API через VkInstance. Екземпляр створюється описом застосунка та будь-яких розширень API, які будуть використовувати [12]. Після створення екземпляра необхідно створити запитат обладнання, яке підтримує Vulkan, і вибрати один або кілька VkPhysicalDevices для використання. Модель також має можливість створити запит на властивості девайсу драйвера, такі як розмір VRAM і можливості пристрою, щоб вибрати потрібні пристрої, наприклад, віддати перевагу використанню виділених графічних карт.

Лістинг 2.1 – Створення VkInstance

```
            VkInstanceCreateInfo instCrtInfo = {};
                                              instCrtInfo.sType     =
VK_STRUCTURE_TYPE_INSTANCE_INSTCRTINFO;
            instCrtInfo.enabledExtensionCount = extensions_count;
            instCrtInfo.ppEnabledExtensionNames = extensions;
    #ifdef IMGUI_DEBUG_REPORT
                                    const    char*    layers[]    =
{ "VK_LAYER_KHRONOS_validation" };
            instCrtInfo.enabledLayerCount = 1;
            instCrtInfo.ppEnabledLayerNames = layers;
                          const   char**   extensionsExt   =   (const
char**)malloc(sizeof(const char*) * (extensions_count + 1));
```

```
            memcpy(extensionsExt, extensions, extensions_count *
sizeof(const char*));
                                extensionsExt[extensions_count]   =
"VK_EXT_debug_report";
             instCrtInfo.enabledExtensionCount = extensions_count
+ 1;
           instCrtInfo.ppEnabledExtensionNames = extensionsExt;
```

2.1.2 Логічний девайс та сімейства черг

Після вибору потрібного апаратного пристрою для використання створюється VkDevice (логічний пристрій), де описується, які функції VkPhysicalDeviceFeature будуть використовуватися. У данній модель необхідні такі опції як візуалізація в кількох вікнах та 64-бітні float. Також потрібно вказати, які сімейства черги будуть використовуватись. Більшість операцій, що виконуються за допомогою Vulkan, такі як команди відображення та операції з пам’яттю, виконуються асинхронно шляхом подання їх до VkQueue. Черги виділяються з сімейст, де кожне сімейство підтримує певний набір операцій у своїх чергах. Наприклад, можуть бути окремі сімейства черги для графічних, обчислювальних операцій і операцій передачі пам’яті. Наявність сімейств черги також може використовуватися як відмінний фактор у виборі фізичного пристрою. Пристрій з підтримкою Vulkan може не пропонувати жодних графічних функцій, однак усі відеокарти з підтримкою Vulkan на сьогоднішній день, як правило, підтримують усі операції з чергою.

Лістинг 2.2 – Аналіз доступних графічних драйверів

```
    void API::VulkanDevice::queryPhysicalDeviceInfo() {
      vkGetPhysicalDeviceProperties(physicalDvc_, &properties);
                 vkGetPhysicalDeviceMemoryProperties(physicalDvc_,
&memoryProperties);
      vkGetPhysicalDeviceFeatures(physicalDvc_, &features);
      uint32_t queueFamilyCount;
            vkGetPhysicalDeviceQueueFamilyProperties(physicalDvc_,
&queueFamilyCount,                                          nullptr);
queueFamilyProperties.resize(queueFamilyCount);
            vkGetPhysicalDeviceQueueFamilyProperties(physicalDvc_,
&queueFamilyCount, queueFamilyProperties.data());
```

```
        uint32_t extCount = 0;
         vkEnumerateDeviceExtensionProperties(physicalDvc_, nullptr,
&extCount, nullptr);
        std::cout << "Vulkan picked following GPU for a test: "
                  << properties.deviceName << std::endl;
    }
```

2.1.3 Пространство вікна та ланцюг підкачки

Як вже було описано у розділі 1.6 для створення кросс платформеного інтерфейсу для взаємодії з віконною системою був використан GLFW.

Щоб фактично відобразити вікно у моделі потрібно також створити поверхню вікна (VkSurfaceKHR) і ланцюг підкачки (VkSwapchainKHR). Постфікс KHR, який означає, що ці об’єкти є частиною розширення Vulkan. PHYSICALDVC Vulkan сам по собі повністю не залежить від платформи, тому потрібно використовувати стандартизоване розширення WSI (Window System Interface) для взаємодії з менеджером вікон. Поверхня є кросплатформною абстракцією над вікнами для візуалізації та зазвичай створюється шляхом надання посилання на власний дескриптор вікна, наприклад HWND у Windows. Бібліотека GLFW має вбудовану функцію для вирішення конкретних деталей цієї платформи.

Ланцюг підкачки – це набір цілей візуалізації. Його основна мета полягає в тому, щоб зображення, яке ми зараз відтворюємо, відрізнялося від того, яке зараз знаходиться на екрані. Це важливо, щоб переконатися, що відображаються лише повні зображення. Кожен раз, коли модель надсилає команду відобразити кадр, ланцюжок підкачки використовується для отримання зображення для візуалізації. Коли процесс обробки кадру завершено, зображення повертається в ланцюг підкачки, щоб у певний момент воно було представлено на екрані.

Кількість цілей візуалізації та умов для відображення готових фреймів на екрані залежить від поточного режиму [13]. Найчастіше зустричаються такі

режими як подвійна буферизація (vsync) і потрійна буферизація. В даній моделі використовується подвійна буферизація.

Лістинг 2.3 – Аналіз доступних графічних драйверів

```
     API::VulkanSwapChain::VulkanSwapChain(APITest::VulkanRenderIm
pl       *parent,VkSurfaceKHR       surface):       _parent(parent),
_surface(surface) {
                        fpGetPhysicalDeviceSurfaceSupportKHR        =
reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceSupportKHR>(vkGetIn
stanceProcAddr(_parent->getInstance(),
"vkGetPhysicalDeviceSurfaceSupportKHR"));
                    fpGetPhysicalDeviceSurfaceCapabilitiesKHR       =
reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceCapabilitiesKHR>(vk
GetInstanceProcAddr(_parent->getInstance(),
"vkGetPhysicalDeviceSurfaceCapabilitiesKHR"));
                        fpGetPhysicalDeviceSurfaceFormatsKHR        =
reinterpret_cast<PFN_vkGetPhysicalDeviceSurfaceFormatsKHR>(vkGetIn
stanceProcAddr(_parent-
>getInstance(),"vkGetPhysicalDeviceSurfaceFormatsKHR"));
                    fpGetPhysicalDeviceSurfacePresentModesKHR       =
reinterpret_cast<PFN_vkGetPhysicalDeviceSurfacePresentModesKHR>(vk
GetInstanceProcAddr(_parent-
>getInstance(),"vkGetPhysicalDeviceSurfacePresentModesKHR"));
```

2.1.4 Об’єкти зображень та буфер кадрів

Для відображення фрейму отриманого з ланцюга підкачки, фрейм має бути загорнут у VkImageView та VkFramebuffer. VkImageView посилається на конкретну частину зображення, яке буде використано, а кадровий буфер посилається на зображення, які мають використовуватися для цілей кольору, глибини та трафарету. Оскільки в ланцюгу підкачки може бути багато різних зображень, обгортки для відображення для буферизації мають бути попередньо створені для кожного з них.

Лістинг 2.4 – Створення VkImageViewCreateInfo

```
     for (uint32_t i = 0; i < imageCount; i++) {
         VkImageViewCreateInfo colorAttachmentView = {};
                                 colorAttachmentView.sType       =
VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
         colorAttachmentView.pNext = NULL;
```

```
        colorAttachmentView.format = colorFormat;
        colorAttachmentView.components = {
                  VK_COMPONENT_SWIZZLE_R,  VK_COMPONENT_SWIZZLE_G,
VK_COMPONENT_SWIZZLE_B,
            VK_COMPONENT_SWIZZLE_A};
               colorAttachmentView.subresourceRange.aspectMask   =
VK_IMAGE_ASPECT_COLOR_BIT;
        colorAttachmentView.subresourceRange.baseMipLevel = 0;
        colorAttachmentView.subresourceRange.levelCount = 1;
        colorAttachmentView.subresourceRange.baseArrayLayer = 0;
        colorAttachmentView.subresourceRange.layerCount = 1;
        colorAttachmentView.viewType = VK_IMAGE_VIEW_TYPE_2D;
        colorAttachmentView.flags = 0;
```

2.1.5 Проходи візуалізації

Проходи візуалізації у Vulkan описують типи зображень, які опрацьовуються під час операцій візуалізації, як вони будуть використовуватися та як слід обробляти їх вміст. У базовій версії модели на базі Vulkan ми будемо використовувати одне зображення як цільовий колір і що ми хочемо, щоб воно було очищено до суцільного кольору безпосередньо перед операцією малювання. У той час як проходження візуалізації описує лише тип зображень, VkFramebuffer насправді прив’язує конкретні зображення до цих слотів.

Лістинг 2.4 – Створення VkImageViewCreateInfo

```
        {
            VkRenderPassBeginInfo info = {};
            info.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
            info.renderPass = wd->RenderPass;
            info.framebuffer = fd->Framebuffer;
            info.renderArea.extent.width = wd->Width;
            info.renderArea.extent.height = wd->Height;
            info.clearValueCount = 1;
            info.pClearValues = &wd->ClearValue;
                  vkCmdBeginRenderPass(fd->CommandBuffer,  &info,
VK_SUBPASS_CONTENTS_INLINE);  }
```

2.2 Проектування примітивів графічного конвейеру та растерізація

Конвеєр комп’ютерної графіки, конвеєр візуалізації або просто графічний конвеєр — це концептуальна модель, яка описує, які кроки графічна система повинна виконати, щоб відобразити 3D-сцену на 2D-екран. Після створення 3D-моделі, наприклад у відеогрі або будь-якій іншій комп’ютерній тривимірній анімації, графічний конвеєр – це процес перетворення цієї 3D-моделі в те, що відображає комп’ютер.

Часто більшість кроків конвеєра реалізовано апаратно, що дозволяє проводити спеціальні оптимізації. Термін «конвеєр» використовується в тому ж значенні, що й конвеєр у процесорах: окремі кроки конвеєра виконуються паралельно, поки будь-який крок має те, що йому потрібно.

Крок растеризації є останнім кроком перед конвеєром фрагментного шейдера, за допомогою якого всі примітиви растеризовані. На етапі растеризації дискретні фрагменти створюються з безперервних примітивів.

На цьому етапі графічного конвеєра точки сітки також називають фрагментами для більшої виразності. Кожен фрагмент відповідає одному пікселю в буфері кадру, і це відповідає одному пікселю екрана. Вони можуть бути кольоровими (і, можливо, освітленими). Крім того, необхідно визначити видимий, ближчий до спостерігача фрагмент, у разі перекриття багатокутників. Для цього так званого визначення прихованої поверхні зазвичай використовується Z-буфер [14]. Колір фрагмента залежить від освітлення, текстури та інших властивостей матеріалу видимого примітиву і часто інтерполюється за допомогою властивостей вершини трикутника. Якщо доступно, фрагментний шейдер (також званий піксельним шейдером) запускається на етапі растрування для кожного фрагмента об’єкта. Якщо фрагмент видно, тепер його можна змішати з уже наявними значеннями кольору на зображенні, якщо використовується прозорість або багатовибірка. На цьому кроці один або кілька фрагментів стають пікселем.

Найважливішими одиницями шейдерів є вершинні шейдери, шейдери геометрії та піксельні шейдери.

Лістинг 2.5 – Опис графічного конвеєру

```
        VkCommandBuffer transitCmd;
        VK_CHECK_RESULT(
                                        parent_->getQueueManager()-
>createPrimaryCommandBuffers(&transitCmd, 1))
        VkCommandBufferBeginInfo cmdBufInfo =
            initializers::commandBufferBeginInfo();
                 VK_CHECK_RESULT(vkBeginCommandBuffer(transitCmd,
&cmdBufInfo))

        VkImageMemoryBarrier barrier{};
        barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        barrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        barrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

        barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

        barrier.image = images[i];
                        barrier.subresourceRange.aspectMask     =
VK_IMAGE_ASPECT_COLOR_BIT;
        barrier.subresourceRange.baseMipLevel = 0;
        barrier.subresourceRange.levelCount = 1;
        barrier.subresourceRange.baseArrayLayer = 0;
        barrier.subresourceRange.layerCount = 1;

        VkPipelineStageFlags sourceStage;
        VkPipelineStageFlags destinationStage;

        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_MEMORY_READ_BIT;

        sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
        destinationStage = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
```

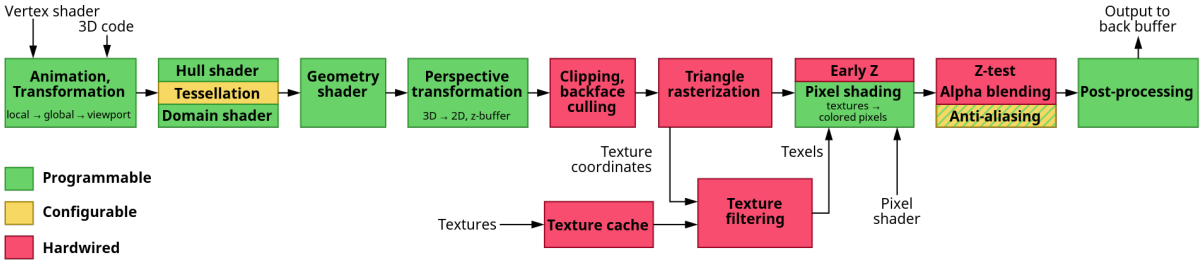

Рисунок 2.1 – Модель графічного конвеєра

2.3 Програмна реалізація моделі

Після описаного вище аналізу доступних графічних драйверів необхідно перевіри їх сімейства. Це допожить визначити доступні функції та загальне призначення конкретного девайсу, також цє допожить відсіяти потенційні вторичні графічні адаптері які не задовільняють нашим потребам.

Лістинг 2.6 – Перевірка сімейства графічного девайсу

```
     uint32_t
APITest::VulkanDevice::getQueueFamilyIndex(VkQueueFlagBits
queueFlags) const {
       // Dedicated queue for compute
        // Try to find a queue family index that supports compute
but not graphics
       if (queueFlags & VK_QUEUE_COMPUTE_BIT) {
         for (uint32_t i = 0;
                                                              i      <
static_cast<uint32_t>(queueFamilyProperties.size()); i++) {
            if ((queueFamilyProperties[i].queueFlags & queueFlags)
&&
                           ((queueFamilyProperties[i].queueFlags  &
VK_QUEUE_GRAPHICS_BIT) ==
                0)) {
             return i;
           }
         }
       }
```

Після цього необхідно виділити пам’ять для роботи с GPU та описати права доступу. Це необхідно для подальшого створення графічного конвеєру та роботи із даними.

Лістинг 2.6 – Створення буферу даних

```
     VulkanDepthBuffer::VulkanDepthBuffer(const VulkanMemoryManager
*alloc,
                                                    VulkanDepthBufferCI
createInfo)
         : VulkanImage(
               alloc, MemoryType::GPU_PRIVATE,
```

```
                static_cast<VkBufferUsageFlagBits>(
                    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT |
                     convertInterfaceImageUsageFlags(createInfo.addi
tionalUsage)),
                compileDepthStencilImageCreateInfo(createInfo)) {
                      VkImageViewCreateInfo        viewInfo        =
initializers::imageViewCreateInfo();
       viewInfo.format = createInfo.depthFormat;
       viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
       viewInfo.image = image();
              viewInfo.components     =     {VK_COMPONENT_SWIZZLE_R,
VK_COMPONENT_SWIZZLE_G,
                                             VK_COMPONENT_SWIZZLE_B,
VK_COMPONENT_SWIZZLE_A};
        viewInfo.subresourceRange = {VK_IMAGE_ASPECT_DEPTH_BIT, 0,
1, 0, 1};

       VK_CHECK_RESULT(
            vkCreateImageView(alloc->device(), &viewInfo, nullptr,
&depthView_))
     }
```

Маючи буфер даних та информацію об інтерфейсі та функціональности драйверу девайса було створено конвеєр даних.

Для перенесення команд до графічного адаптеру є необхідним реалізувати менеджер черги. У тривіальному варіанті менеджер команд має запросити дві черги від графічного адаптору – графічну та передаючу. Перед запитом черг має бути вкионано створення командного пулу. В залежності від пристрою, це створення може бути завершено нормально або з помилкою. У випадку помилки подальше виконання не є можливим, тому доцільним є викидання виключення з описом помилки, що була отримано. Всю процедуру ініціалізації є коректним виконати в конструкторі об'єкту VulkanQueueManager що є доцільним з боку парадигми об’єктно-орієнтованого програмування. Фрагмент ініціалізації командного буферу наведено у лістингу 2.7.

Лістинг 2.7 – Ініціалізація менеджеру черги команд на графічний пристрій

```
       VkCommandPoolCreateInfo cmdPoolInfo = {};
                                  cmdPoolInfo.sType                =
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
```

```
                cmdPoolInfo.queueFamilyIndex      =      device_-
>queueFamilyIndices.graphics;
                                    cmdPoolInfo.flags               =
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
          if   (auto   err   =   (vkCreateCommandPool(device_->get(),
&cmdPoolInfo, nullptr,
                                              &commandPool_) -
                       VK_SUCCESS))
             throw  std::runtime_error("[ERROR][VkAPI]:  Could  not
create a command pool "
                                  "on given device. Err code: " +
                                              std::to_string(err +
VK_SUCCESS));
                      vkGetDeviceQueue(device->get(),        device-
>queueFamilyIndices.graphics, 0,
                         &queue_);

                      vkGetDeviceQueue(device->get(),        device-
>queueFamilyIndices.transfer, 0,
                         &transferQueue_);
```

2.4 Система збирання проекту

Для збирання проекту використовується кросс-платформена та кросс-архітектурна система CMake що дозволяє виконати конфігурацію та коміляцію проекта у заданому оточенні без зав’язки на конкретну IDE та оточення. Для встановлення сторонніх бібліотек, а саме glfw та glew використано систему пакетого менеджменту залежностей conan яка дозволяє встановити необхідні компоненти у вигляді зібраних образів або виконати збирання на цільовій платформі. За рахунок особливостей написання рецептів для пакетів на мові програмування Python є можливим швидко описувати нові залежності або редагувати існуючі рецепти пакетів.

Для конфігурації conan разом з CMakeLists є необхвдним виконати встановлення скрипту conan-cmake що дозволяє вказати всі необхідні опції безпосередньо у файлу опису проекту.

Приклад опису залежностей проекту для використання з Vulkan SDK наведено у Лістингу 2.8.

Лістинг 2.8 – Фрагмент опису конфігурації залежностей для проекту

```
    if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
                  message(STATUS "Downloading conan.cmake from
https://github.com/conan-io/cmake-conan")
            file(DOWNLOAD "https://raw.githubusercontent.com/conan-
io/cmake-conan/v0.16.1/conan.cmake"
                          "${CMAKE_BINARY_DIR}/conan.cmake"
                                                  EXPECTED_HASH
SHA256=396e16d0f5eabdc6a14afddbcfff62a54a7ee75c6da23f32f7a31bc85db
23484
                          TLS_VERIFY ON)
        endif()

        list(APPEND CMAKE_MODULE_PATH ${CMAKE_BINARY_DIR})
        list(APPEND CMAKE_PREFIX_PATH ${CMAKE_BINARY_DIR})

        include(${CMAKE_BINARY_DIR}/conan.cmake)

        conan_cmake_configure(REQUIRES glfw/3.3.3 glew/2.2.0
                              GENERATORS cmake_find_package
                                         IMPORTS "bin, *.dll -> $
{CMAKE_CURRENT_BINARY_DIR}"
                              OPTIONS glew:shared=True
                              )

        conan_cmake_autodetect(settings)

        conan_cmake_install(PATH_OR_REFERENCE .
                            BUILD missing
                            REMOTE conancenter
                            SETTINGS ${settings})
```

Після встановлення залежностей проект є необхідним налаштувати шляхи до Vulkan SDK та налаштувати структуру проекту з розбиттям на бібліотеки, що будуть влінковані до результуючого файлу. У випадку реалізації структури проеку є доцільним розбити проект на директорію з прикладами, спільними компонентами та конкретними компонентами OpenGL та Vulkan. Для реалізації спільних компонентів можливо використати функціонал глобування файлів до одієї бібліотеки та створення конекретних бібліотек. Фрагмент реалізації бібліотеки спільних компонентів наведено у Лістингу 2.9.

Лістинг 2.9 – Фрагмент реалізації скриптів збирання спільних компонентів та залежних бібліотек

```
    file(GLOB RENDER_API_SOURCES *.cpp *.h  glfw_window_impl/*.h
glfw_window_impl/*.cpp)

    set(COMMON_RENDER_HEADERS        ../include/RenderInterface.h
WindowInterface.h)
    include_directories(${CMAKE_CURRENT_SOURCE_DIR})

    file(GLOB         VULKAN_IMPL_SOURCES         vulkan_impl/*.h
vulkan_impl/*.cpp vulkan_impl/util/*.cpp vulkan_impl/util/*.h)
    file(GLOB         OPENGL_IMPL_SOURCES         opengl_impl/*.h
opengl_impl/*.cpp)
    file(GLOB IMGUI_IMPL_SOURCES imgui_impl/*.h imgui_impl/*.cpp)

    file(GLOB                    IMGUI_SOURCE                    $
{PROJECT_SOURCE_DIR}/external/imgui/*.cpp                        $
{PROJECT_SOURCE_DIR}/external/imgui/*.h)

    option(LINK_RENDER_API_SHARED  "link  shared  library  for
APIDriver instead of static one" OFF)

    if(LINK_RENDER_API_SHARED AND UNIX)
      add_compile_options(-fPIC)
    endif()

    add_library(IMGUI_SOURCE_LIB OBJECT ${IMGUI_SOURCE})

    set(CMAKE_RUNTIME_OUTPUT_DIRECTORY                           $
{CMAKE_CURRENT_BINARY_DIR}/..)

    add_library(VULKAN_IMPL_LIB  OBJECT  ${VULKAN_IMPL_SOURCES}  $
{COMMON_HEADERS})
    add_library(OPENGL_IMPL_LIB  OBJECT  ${OPENGL_IMPL_SOURCES}  $
{COMMON_HEADERS})
    add_library(IMGUI_IMPL_LIB  OBJECT  ${IMGUI_IMPL_SOURCES}    $
{COMMON_HEADERS})

    if(LINK_RENDER_API_SHARED)
                    add_library(${RENDER_API_LIB}    SHARED    $
{RENDER_API_SOURCES})
    else()
        add_library(${RENDER_API_LIB} ${RENDER_API_SOURCES})
    endif()
```

Для компіляції файлів шейдерів використовується GLSL з викликанням його після знаходження файлів VulkanSDK та механізмом додавання custom_tagets. Знаходження програми glslc дозволяє додати компіляцію

шейдерів [15]. Сама функція збирання шейдерів реалізована з використанням вбудованого до CMake механізму створення макросів для об’єднання компонентів. Фрагмент лістингу для коректної компіляції шейдерів наведено у Лістингу 2.10.

Лістинг 2.10 – Фрагмент реалізації функції для компіляції шейдерів з використанням glslc

```
    function(compileShaders          SHADER_DIR          OUTPUT_DIR
DEPENDENT_TARGET INSTALL_DIR)
    find_program(GLSL     NAMES     glslc.exe     glslc     PATHS
$ENV{VK_SDK_PATH}/Bin $ENV{VULKAN_SDK}/bin)

        if(GLSL-NOTFOUND)
            message("GLSL compiler not found - shader compilation
skipped")
        ELSE()
            message("Found following GLSL compiler:")
            message(STATUS ${GLSL})

                    file(GLOB  SHADERS  RELATIVE  ${SHADER_DIR}    $
{SHADER_DIR}/*.vert ${SHADER_DIR}/*.frag)

            foreach(SHADER ${SHADERS})
                        compileShader(${SHADER}  ${SHADER_DIR}  $
{OUTPUT_DIR})
            endforeach(SHADER)

             list(TRANSFORM SHADERS APPEND ".spv" OUTPUT_VARIABLE
SHADERS_BINS)
            list(TRANSFORM SHADERS_BINS PREPEND ${OUTPUT_DIR}/)
                    install(FILES  ${SHADERS_BINS}  DESTINATION  $
{INSTALL_DIR})
            add_dependencies(${DEPENDENT_TARGET} ${SHADERS})
        ENDIF()
    endfunction(compileShaders)
```

Шейдерні компоненти можуть бути реалізовані як з використанням опису GLSL так і з описом SPIR-V шо є доцільними до використання з графічним API Vukan. Розробка виконана з підтримкою обох варіантів, різниця між якими є у мінімальних компонентах ситаксису.Для прикладу наведена реалізація фрагментного шейдеру на мові опису SPIR-V.

Лістинг 2.11 – Реалізація фрагментного шейдеру з використанням мови опису SPIR-V

```
#version 450

layout (location = 0) in vec3 inUVW;

layout (location = 0) out vec4 outFragColor;


void main(){

    int triIndex = int(inUVW.z);

     outFragColor = vec4( (vec2(1.0) + cos(inUVW.xy * 50.0) * 0.6 + sin(inUVW.yx * 100.0 + 30.0) * 0.4) * 0.5,
                                    0.5 , 1.0f);

    if(triIndex % 2 == 0)
          outFragColor = vec4(outFragColor.g, outFragColor.b, outFragColor.r, 1.0f);
}
```

Для порівняння, реалізація того самого шейдеру на мові GLSL. Різниця наведена у моментах, пов’язаних з визначенням вхідних та вихідних параметрів шейдеру та задання значень до змінних. Реалізація шейдеру на мові GLSL наведена у Лістингу 2.12.

Лістинг 2.12 – Реалізація фрагментного шейдеру на мові опису GLSL

```
#version 400

out vec4 frag_colour;

in vec3 uv;

void main(){

    int triIndex = int(uv.z);

     frag_colour = vec4( (vec2(1.0) + cos(uv.xy * 50.0) * 0.6 + sin(uv.yx * 100.0 + 30.0) * 0.4) * 0.5,
    0.5 , 1.0f);

    if(triIndex % 2 == 0)
            frag_colour = vec4(frag_colour.g, frag_colour.b, frag_colour.r, 1.0f);
}
```

3 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

Для аналізу характеристик розробленої моделі було виконано експериментальне дослідження з використанням рендеренгу сцени з повторюваними компонентами.

Цей приклад є достатньо типовим під час реазації систем візуалізації та менеджменту часток, що мають динамічний характер, але мають бути відмальовані найбільш оптимальним шляхом. У рамках цієї задачі є актуальним забепечення максимальноефективного завантаження графічного конвейеру без надлишкових блокувань конвейеру.

Розвитком прикладу менеджменту часток може бути розглядання розширеного варіанту з текстуруванням часток та динамічним ренденрінгом освітлення та/ або тайлинговим завантаженням текстурних компонент. Подібна задача може бути актуальною під час реалізації графічного бекенду для компонентів фреймворків, таких як Qt, коли необхідно забезпечити максимальну кількість кадрів на платформі з обмеженими ресурсами.

На деяких платформах реалізація OpenGL частково або повністю може не відповідати вимогам з точки зору оптимального використання системних ресурсів. Наприклад, на процесорах ARMз графічними адапторами Mali-400 не завжди є можливим та доцільним використовувати OpenGL через наявність надлишкових блокувань системних мьютексів та особливості організації моделі пам’яті ARM архітектури.

Навіть при розгляданні специфікацій OpenGLрізних виробників, все частіше можна визначити тренд в сторону підтримки специфікації Vulkan, яка надає необхідний контроль за ресурсами.

Компіляція та аналіз розроблених прикладів виконано з використанням компілятору MSVC. Всі сторонні бібліотеки окрім VulkanSDK було встановлено до системи з використанням пакетного менеджеру Conan. Conan

дозволив виконати максимально гнучко конфігурацію для бібліотек, що є платформо-залежними.

У рамках проведеного дослідження виконана реалізація моделі з атрибутами та використанням uniform buffer-у.

Різниця у отриманому FPS складає 10 порядків. Для варіанту з Uniform buffer результат складає у середньому 320 FPS, у той час як реалізація з attributes видає 27 кадрів за секунду. Тестування проведено на апаратній платформі з GPU AMD Radeon RX 5700XT та NVidia GeForce GXT650. Результати тестування наведено на рисунках 3.1 та 3.2.

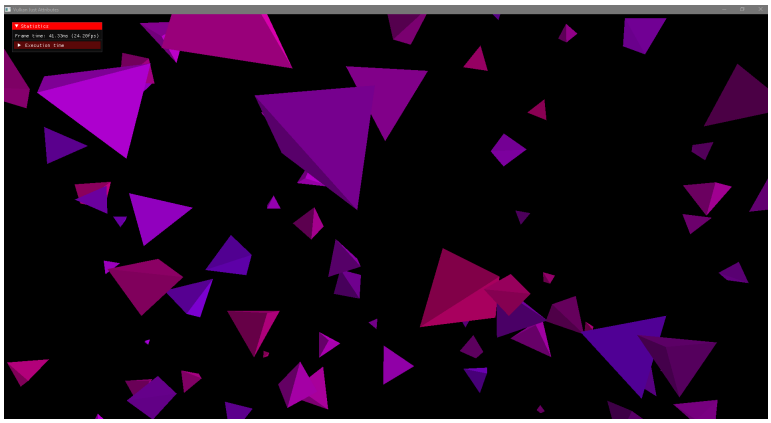

Рисунок 3.1– Реалізація з використанням attributes

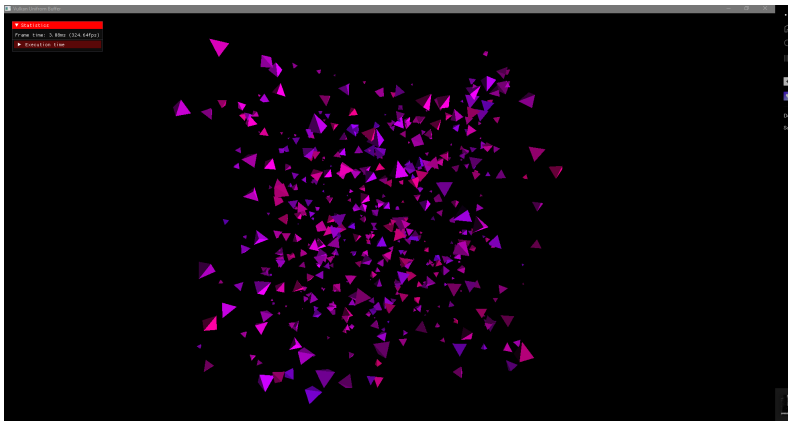

Рисунок 3.2 – Реалізація з використанням uniform buffery

Одним з реалізованих прикладів також є фрагмент з базовим малюванням трикутнику з використанням графічного API Vulkan. Для профілювання викликів до графічного драйверу виконано малювання статистики викликів з

використанням бібліотеки Imgui що має можливість бути вбудованою до графічної поверхні, де малювання виконується з використаням іншого драйверу. У прикладі виведено основні параметри, такі як час початку малювання фрейму, час на рисування графічного інтерфейсу ImGUI, час на виконання переключення черги малювання та запис команд до графічного адаптеру. Окремо виділено час на виконаня додаткового користувацького коду. На рисунку 3.3 зображено результат виконання розробленого базового прикладу.

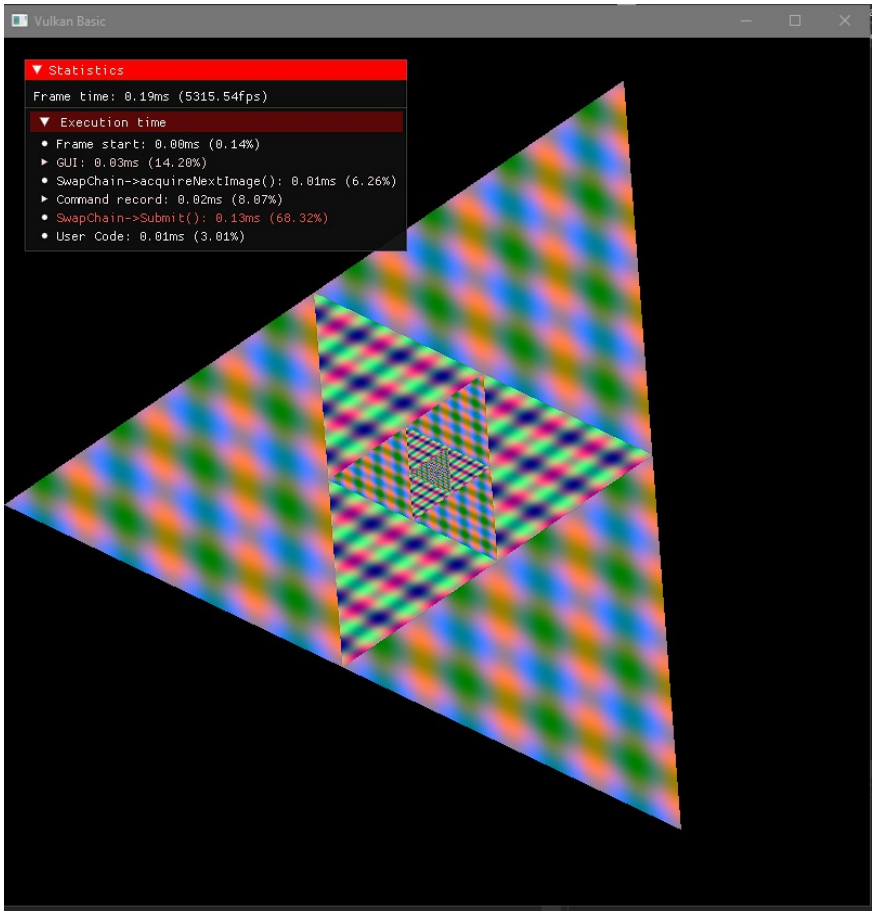

Рисунок 3.3 – Результат виконання базового прикладу з відображенням основних характеристик конвейеру рендерінгу зображення

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи була розроблена модель обробки зображень на базі графічного API Vulkan. В ході дослідження виконано аналіз низькорівневих та високорівневих графічних API. Реалізовано компоненти для роботи з рендерінгом графічних примітивів та сцен з використанням OpenGL та Vulkan та допоміжних бібліотек glew та glm для отримання платформо-залежних десктрипторів вікна та поверхні для рендерінгу зображення. Запропоновано модель мінімального графічного рендереру для 3D сцен.

Дослідження показало перспективність що до використання низькорівневих графічних драйверів для рішення спеціалізованих завдань в області ефективного рендерінгу на системах з обмеженими ресурсами. Порівняльний аналіз кодової бази з OpenGL та Vulkan демонструє переваги що до забезпечення необхідного рівня гнучкості для отримання найкращих результатів та ручного керування усіма доступними ресурсами на системі.

Подальші дослідження для проектування графічних рендерерів можуть включати в себе використання спеціалізованих рішень від виробників SoC. Такими варіантами можуть бути драйвери Metal, драйвери для графічних адаптерів на платформі ARM з компонентами Mali. Також одним з перспективним напрямом досліджень може бути розгляд C++23 компонентів для програмування з використанням libunifex кросс-пристройної реалізації обчислень з передачею керування через канали даних.

Одним з перспективних напрямів може бути розгляд написання шейдерів для архітектури RISK-V та графічних адаптерів на його базі. Також, є варіанти розроблення шейдерів для відктритих архітектур графічних процесорів, де може бути виконано оптимізаційні шейдерні інструкції з використанням

компілятору Clang та його оптимізаційних можливостей для різних мов опису програмних компонентів.

У ході дослідження виявлено перспективи що до створення програмних моделей обробки графіки з налаштуванням графічного API Vulkan для конкретої апаратної платформи з оптимізаціями, що мають як алгоритмічний так і апаратний потенціал.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Тестування програмного забезечення [Електронний ресурс] // arobenko. – 2018. – Режим доступу до ресурсу: http://www.protesting.ru/

2. QA, QC і тестування [Електронний ресурс]– Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/qa-qc-i-testuvannya/

3. Автоматизація тестування [Електронний ресурс]– Режим доступу до ресурсу: http://www.nicotech.ru/quality-assurance/test-automation/

4. Автоматическое тестирование ASP.NET [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/147902/

5. Пример создания Unit-теста [Електронний ресурс] - Режим доступу до ресурсу: https://www.bestprog.net/ru/2018/08/27/example-of-creating-a-unit-test-in-ms-visual-studio-2010-c_ru/

6. luncliff. Exploring MSVC Coroutine¶ [Електронний ресурс] / luncliff. – 2019. – Режим доступу до ресурсу: https://luncliff.github.io/coroutine/articles/exploring-msvc-coroutine/.

7. Perfect Software and Other Illusions About Testing / Gerald M. Weinberg, 2010 - 142c.

8. Craig Scott Professional CMake: A Practical Guide. 2018. 429 с.

9. Bug Life Cycle [Електронний ресурс] - Режим доступу до https://coderlessons.com/tutorials/bolshie-dannye-i-analitika/professiia-biznes-analitik/bug-life-cycle-2

10. Modern OpenGL [Електронний ресурс] - Режим доступу до https://habr.com/ru/post/457380/

11. Awesome Vulkan [Електронний ресурс] - Режим доступу до https://project-awesome.org/vinjn/awesome-vulkan

12. Modern Software Problems [Електронний ресурс] - Режим доступу до https://habr.com/ru/post/596517/

13. Vulkan. Environment settings [Електронний ресурс] - Режим доступу до https://habr.com/ru/post/526320/

14. Extra Software Libraries [Електронний ресурс] - Режим доступу до https://github.com/ValentiWorkLearning

15. Ruspberry Pi 4 Windows setup [Електронний ресурс] - Режим доступу до https://habr.com/ru/news/t/511570/