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

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування і спеціалізованих комп'ютерних систем

До захисту допущено

Завідувач кафедри
_____ В. О. РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
"___"_____2023 р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою**
**«Системне програмування та спеціалізовані комп'ютерні системи»**
**спеціальності 123 «Комп'ютерна інженерія»**

по спеціальності **123 «Комп'ютерна інженерія»**
на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

Виконав (-ла):

студент (-ка) IV курсу, групи КВ-91
(шифр групи)
Дзямулич Даніель Васильович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник асистент кафедри СПСКС, к.т.н, Морозов К.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю, доц.каф.СПСКС, к.т.н. Клятченко Я.М.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) _ (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 року

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

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)
**Освітньо-професійною програма**
«**Системне програмування** та **спеціалізовані**
**комп'ютерні системи»**
**Спеціальність 123 «Комп'ютерна інженерія»**

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.О. РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
«___»_____2023____р.

**ЗАВДАННЯ**
**на дипломний проєкт студента**
Дзямулича Даніеля Васильовича
(прізвище, ім'я, по батькові)

1. Тема проєкту: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі» ,
керівник проєкту Морозов К.В., к.т.н., доц.каф.СПСКС ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «__»_____ 2023_ р. №_____

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту Технічне завдання
_____
_____
_____

4. Зміст пояснювальної записки _____

1) Аналіз існуючих рішень та обґрунтування теми дипломного проєкту
2) Графічні бібліотеки для керування апаратним прискорювачем

3) Програмне забезпечення для взаємодії із апаратним прискорювачем

4) Аналіз отриманих результатів та порівняння із використання високорівневих графічних бібліотек

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень, плакатів, презентацій тощо)

1) Обчислювальний шейдер. Схема алгоритму

2) Пермутація та компіляція шейдерного коду. Схема алгоритму

3) Керування виконанням списків команд. Схема алгоримту

4) Виставлення необхідних ресурсів в регістри апаратного прискорювачу. Схема алгоритму

6. Консультанти розділів проєкту*

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Нормоконтроль | Клятченко Я.М., доцент каф. СПСКС | | |
| | | | |

7. Дата видачі завдання _____

Календарний план

| № з/п | Назва етапів виконання дипломного проєкту | Термін виконання етапів проєкту | Примітка |
|---|---|---|---|
| 1. | Вивчення літератури за тематикою проекту | 15.04.2023 | |
| 2. | Розроблення та узгодження технічного завдання | 30.04.2023 | |
| 3. | Аналіз існуючих рішень | 05.05.2023 | |
| 4. | Підготовка матеріалів першого розділу дипломного проекту | 10.05.2023 | |
| 5. | Підготовка матеріалів другого розділу дипломного проекту | 18.05.2023 | |
| 6. | Підготовка графічної частини дипломного проекту | 20.05.2023 | |
| 7. | Оформлення документації дипломного проекту | 25.05.2023 | |
| 8. | Попередній огляд матеріалів диплому на кафедрі | 29.05.2023 | |

Студент _____ Даніель ДЗЯМУЛИЧ
(підпис)

Керівник проєкту _____ Костянтин МОРОЗОВ
(підпис)

* Консультантом не може бути зазначено керівника дипломного проєкту.

# АНОТАЦІЯ

Бакалаврський дипломний проєкт включає пояснювальну записку (58 с., 5 рис., список використаної літератури з 16 найменувань, 3 додатків).

Об’єкт розробки – алгоритм та програма управління апаратним прискорювачем тривимірної графіки, для обробки великої кількості даних з використанням низькорівневого прикладного програмного інтерфейсу Direct3D 12 у реальному часі.

Ця програма дозволяє у реальному часі: компілювати та пермутувати шейдера (програма, що виконується на апаратному прискорювачі); компілювати об’єкти стану конвеєра та змінювати їх динамічні налаштування; змінювати виставлені динамічні ресурси в шейдерах; керувати виконанням списків команд апаратним прискорювачем; читати та писати дані в локальну пам’ять апаратного прискорювача для подальшої їх обробки. Продукт використовує інтерфейс прикладного програмування Direct3D12.

У ході розробки:

- Проаналізовано існуючі програми управління апаратним прискорювачем тривимірної графіки у реальному часі;
- розроблено механізм компіляції та пермутації шейдерів у реальному часі;
- розроблено систему для обробки виконаних списків команд та їх перевикористання;
- створено механізм динамічної компіляції об’єктів стану конвеєра;

Використання цього програмного продукту може допускати не тільки застосування апаратних прискорювачів, що підтримують Direct3D 12, але й мати реалізацію для спеціалізованої апаратури. Також при відсутності такої, алгоритм допускає використання тільки центрального процесора.

Ключові слова:

ПРОГРАМА УПРАВЛІННЯ АПАРАТНИМ ПРИСКОРЮВАЧЕМ ТРИВИМІРНОЇ ГРАФІКИ, УПРАВЛІННЯ ВІДЕОКАРТОЮ, ПРОГРАМА УПРАВЛІННЯ ГРАФІЧНИМ ПРОЦЕСОРОМ.

# ABSTRACT

A bachelor's thesis project includes an explanatory note (58 pages, 5 figures, a list of references with 16 titles, 3 appendices).

The object of development – algorithm and control software of hardware accelerator of three-dimensional graphics, to manage big amount of data with use of low-level API Direct3D 12 in real time.

This software allows in real time: compile and permutate shaders (software, which runs on hardware accelerator); compile pipeline state objects and change their dynamic parameters; change binding of dynamic resources in shaders; manage execution of command lists by hardware accelerator; read and write data to local memory of hardware accelerator for its further processing. Software uses Direct3D12 API.

During the development:

- analyzed the existing control software of hardware accelerator of three-dimensional graphics in real time;
- has been developed a mechanism for compiling and permuting shaders in real time;
- has been developed a system which manages execution of command lists and its reusement;
- has been created a mechanism of dynamic compilation pipeline state objects;

The use of this software product can allow not only the use of hardware accelerators that support Direct3D 12, but also have an implementation for specialized hardware. Also, in the absence of such, the algorithm allows the use of only the central processor.

Ключові слова:

CONTROL SOFTWARE OF HARDWARE ACCELERATOR OF THREE-DIMENSIONAL GRAPHICS, DIRECT3D 12, DIRECTX 12, PROGRAM GPU, GRAPHICS CARD CONTROL.

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045200.002 ТЗ | Алгоритм та програма | 3 | | |
| | | | управління апаратним | | | |
| | | | прискорювачем тривимірної | | | |
| | | | графіки у реальному часі. | | | |
| | | | Технічне завдання | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.003 ТП | Алгоритм та програма | 1 | | |
| | | | управління апаратним | | | |
| | | | прискорювачем тривимірної | | | |
| | | | графіки у реальному часі. | | | |
| | | | Відомість технічного | | | |
| | | | проєкту | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.004 ПЗ | Алгоритм та програма | 58 | | |
| | | | управління апаратним | | | |
| | | | прискорювачем тривимірної | | | |
| | | | графіки у реальному часі. | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

| Змін. | Арк. | № докум. | Підпис | Дата | ***ІАЛЦ.045200.001 ОА*** | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Дзямулич Д.В. | | | Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі. ***Опис альбому*** | Літ. | Аркуш 1 / Аркушів 2 |
| Перевірив | | Морозов К.В. | | | | | |
| Н. контроль | | Клятченко Я.М. | | | | **НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-91** | |
| Зав. каф. | | Романкевич В.О. | | | | | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045200.005 Д1 | Обчислювальний шейдер. | 1 | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.006 Д2 | Пермутація та компіляція | 1 | | |
| | | | шейдерного коду. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.007 Д3 | Керування виконанням | 1 | | |
| | | | списків команд. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.008 Д4 | Виставлення необхідних | 1 | | |
| | | | ресурсів в регістри | | | |
| | | | апаратного прискорювача. | | | |
| | | | Схема алгоритму. | | | |
| | | | | | | |
| | | Диск CD-RW | Текст пояснювальної | | | |
| | | | записки. Код програми. | | | |
| | | | Презентація дипломного | | | |
| | | | проєкту. | | | |
| | | | Графічний матеріал | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

# ТЕХНІЧНЕ ЗАВДАННЯ

**До дипломного проєкту**

на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

Київ – 2023

**ЗМІСТ**

## 1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі».

Галузь застосування: розробка комп'ютерних ігор, програм для обробки комп'ютерної графіки.

## 2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

## 3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проекту є створення алгоритму та програми управління апаратним прискорювачем тривимірної графіки у реальному часі.

## 4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації у періодичних виданнях та електронні статті у мережі Інтернет.

## 5. ТЕХНІЧНІ ВИМОГИ

### 5.1. Вимоги до програмного продукту, що розробляється

- отримання правильних результатів вираховувань за допомогою апартного прискорювача;

5.2. Вимоги до апаратного забезпечення

- відеокарта з підтримкою DirectX 12.

5.3. Вимоги до програмного та апаратного забезпечення користувача

- операційна система: Windows 10, Windows 11.

## 6. ЕТАПИ РОЗРОБКИ

| № з/п | Назва етапів виконання дипломного проекту | Термін виконання етапів |
|---|---|---|
| 1. | Вивчення літератури за тематикою проекту | 15.04.2023 |
| 2. | Розроблення та узгодження технічного завдання | 30.04.2023 |
| 3. | Аналіз існуючих рішень | 05.05.2023 |
| 4. | Підготовка матеріалів першого розділу дипломного проекту | 10.05.2023 |
| 5. | Підготовка матеріалів другого розділу дипломного проекту | 18.05.2023 |
| 6. | Підготовка графічної частини дипломного проекту | 20.05.2023 |
| 7. | Оформлення документації дипломного проекту | 25.05.2023 |
| 8. | Попередній огляд матеріалів диплому на кафедрі | 29.05.2023 |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | А4 | ІАЛЦ.045200.004 ПЗ | Алгоритм та програма | 58 | | |
| | | | апаратним прискорювачем | | | |
| | | | тривимірної графіки у | | | |
| | | | часі. Пояснювальна записка | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.005 Д1 | Обчислювальний шейдер. | 1 | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.006 Д2 | Пермутація та компіляція | 1 | | |
| | | | шейдерного коду. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.007 Д3 | Керування виконанням | 1 | | |
| | | | списків команд. | | | |
| | | | Схема алгоритму | | | |
| | | | | | | |
| | А4 | ІАЛЦ.045200.008 Д4 | Виставлення необхідних | 1 | | |
| | | | ресурсів в регістри | | | |
| | | | апаратного прискорювача. | | | |
| | | | Схема алгоритму. | | | |

| Змін. | Арк. | № докум. | Підпис | Дата | **ІАЛЦ.045200.003 ТП** | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Дзямулич Д.В. | | | Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі. | Літ. | Аркуш 1 / Аркушів 2 |
| Перевірила | | Морозов К.В. | | | | | |
| Н. контроль | | Клятченко Я.М. | | | ***Відомість технічного проєкту*** | **НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-91** | |
| Зав. каф. | | Романкевич В.О. | | | | | |

| Поз. | Формат | ПОЗНАЧЕННЯ | НАЙМЕНУВАННЯ | Кількість аркушів | № прим. | Примітки |
|---|---|---|---|---|---|---|
| | | Диск CD-RW | Текст пояснювальної | | | |
| | | | записки. Код програми. | | | |
| | | | Презентація дипломного | | | |
| | | | проєкту. | | | |
| | | | Графічний матеріал | | | |

| Змін. | Арк. | № докум. | Підпис | Дата | ***ІАЛЦ.045200.003 ТП*** | Арк. 2 |
|---|---|---|---|---|---|---|

# ПОЯСНЮВАЛЬНА ЗАПИСКА

**До дипломного проєкту**

на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

Київ – 2023

## ЗМІСТ



**ДОДАТКИ**

**Додаток А. Копії графічних матеріалів**

- ІАЛЦ.045200.005 Д1. Обчислювальний шейдер. Схема алгоритму;
- ІАЛЦ.045200.006 Д2. Пермутація та компіляція шейдерного коду. Схема алгоритму;

- ІАЛЦ.045200.007 Д3. Керування виконанням списків команд. Схема алгоримту;
- ІАЛЦ.045200.008 Д4. Виставлення необхідних ресурсів в регістри апаратного прискорювачу. Схема алгоритму.

**Додаток Б. Фрагменти програмного коду**

**Додаток В. Презентація бакалаврського проєкту**

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Відеокарта, GPU (Graphics Processing Unit) – апаратний прискорювач тривимірної графіки або графічна процесорна одиниця.

Шейдер – програма, що виконується на шейдерних блоках апартного прискорювача тривимірної графіки.

ПЗ – програмне забезпечення.

API (Application Programming Interface) – графічно-прикладний програмний інтерфейс.

C++ – мова програмування, яка була використана для написання програмного забезпечення.

CPU (Central Processing Unit) – головний або центральний процесор, процесорна одиниця.

DX12, DirectX12, D3D12, Direct3D12 – низькорівневий графічно-прикладний програмний інтерфейс для програмування апаратних прискорювачів тривимірної графіки.

HLSL – пропріетарна мова програмування для написання шейдерів, яка розроблена компанією Microsoft.

Windows – графічна операційна система, розроблена компанією Microsoft.

# ВСТУП

У сучасному світі апаратні прискорювачі тривимірної графіки виконують важливу роль у створенні тривимірної графіки, відео-ефектів, прискоренні кодування та декодування відео. Також завдяки швидкому розвитку відеокарт і збільшення їх обчислювального потенціалу, суттєво прискорився розвиток та застосування штучного інтелекту. Адже, штучний інтелект використовується для розв'язання завдань, які вимагають великої обчислювальної потужності, таких як обробка великих обсягів даних, комплексний аналіз інформації, комп'ютерне зорове сприйняття, машинне навчання та глибоке навчання. Відеокарти можуть значно прискорити виконання цих завдань, завдяки великій кількості паралельних обчислювальних одиниць на апаратних прискорювачах. Наприклад, глибокі нейронні мережі, які є ключовим інструментом у галузі машинного навчання, вимагають великої кількості матричних операцій, які можуть бути розпаралелені та ефективно виконані на відеокартах. Проте, програмування графічних процесорів потребує від розробника, вирішення багатьох задач, що включають управління пам'яттю, синхронізацію центрального процесора та процесора апаратного прискорювача, оптимізації команд, які відправляються на виконання до графічного процесора та багато іншого.

Одна з найважливіших задач при програмуванні відеокарт – ефективний розподіл завдань між обчислювальними ядрами графічного процесору. Враховуючи архітектуру апаратного прискорювача, необхідно рівномірно розподілити задачі між всіма його ядрами.

Інша складність виникає при управлінні його локальною пам'яттю. Це включає передачу даних між пам'яттю центрального процесору та власною пам'яттю апаратного прискорювача. Ефективне використання пам'яті є критичним аспектом, оскільки неправильне використання може призвести до значного зниження продуктивності та зайвого копіювання даних.

Крім того, програмування апаратних прискорювачів вимагає вирішення проблеми синхронізації різних потоків обчислень, кешів ядер та загальної пам’яті. Правильна синхронізація є необхідною для уникнення конфліктів та некоректної роботи при паралельних обчисленнях.

Також варто не забувати про синхронізацію між центральним процесором та процесором апаратного прискорювача, адже неправильне використання синхронізацій може призвести до непередбачуваної поведінки або некоректного результату.

Оптимізація є ще однією важливою задачею при програмуванні відеокарт. Використання спеціалізованих функцій та операцій відеокарти, оптимізація алгоритмів та уникнення зайвих копіювань даних можуть суттєво підвищити продуктивність програми на відеокарті.

# 1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ

На сьогодні існує велика кількість рішень для програмування апаратного прискорювача такі як Unreal Engine, Unity, Godot та інші. Але головним недоліком їх являється перевантаженість різноманітними функціями, потреба у більших ресурсів центрального процесору та пам’яті у порівнянні з використанням програмних інтерфейсів низького рівня для програмування апаратних прискорювачів.

Але для початку необхідно зрозуміти для чого нам потрібно використовувати окремий апаратний пристрій для математичних вираховувань, адже в нас вже є центральний процесор, який вміє виконувати такі операції. Звичайно що причиною є час виконання і кількість витраченої на це енергії. Апаратний прискорювач тривимірної графіки може виконувати математичні операції у 100 і більше разів швидше ніж центральний процесор при тих же споживаннях енергії. Така різниця криється в причині створення центрального процесора та апаратного прискорювача тривимірної графіки.

Центральний процесор був створений для виконання широкого спектру задач в комп’ютерних системах. Таких, як запуск і керування операційною системою, виконання різних типів програм, керування пам’яттю і тд. Центральний процесор був спроектований для виконання задач послідовно, натомість апаратний прискорювач тривимірної графіки був створений для вузького спектру задач, які виконуються паралельно, у вигляді SIMD (Одна інструкція, декілька даних), тобто виконання одної інструкції з різними вхідними даними за один такт. Сучасні центральні процесори також вміють виконувати задачі у вигляді SIMD, але кількість ядер у центрального процесору суттєво відрізняється від апаратного прискорювача.

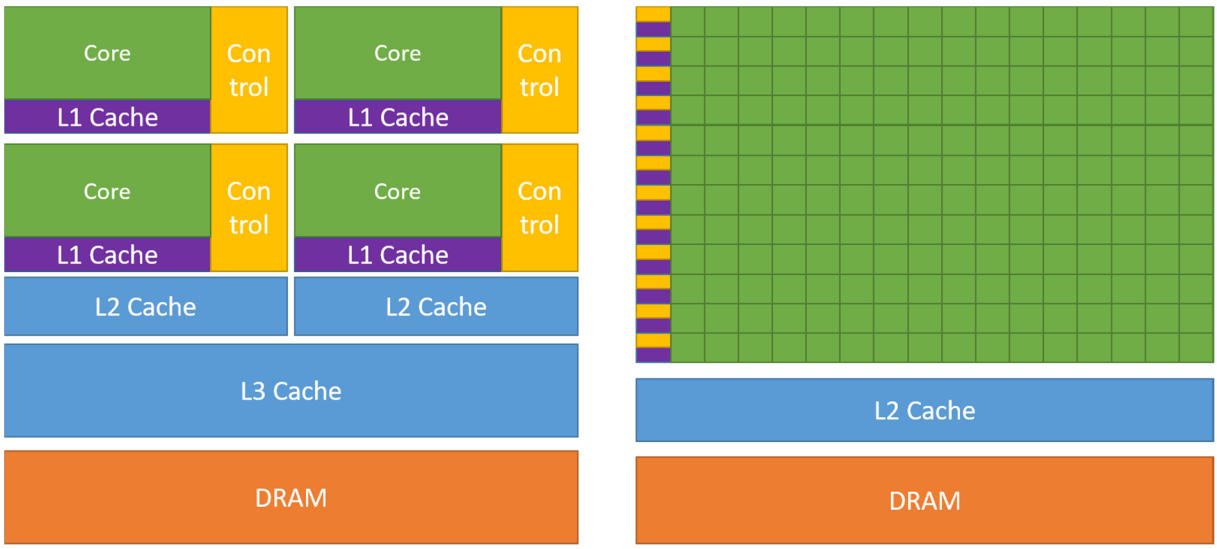



Рисунок 1 – Спрощений апаратний дизайн центрального процесора та апаратного прискорювача тривимірної графіки

На рисунку 1, можна побачити спрощені апаратні дизайни центрального та графічного процесора. Перше що кидається в очі це кількість ядер в графічного процесору. У ньому ядра поєднуються в кластери, або сегменти ядер, які мають єдиний спільний кеш, та єдину контролюючу одиницю. Також можна помітити різницю у розмірі кешу, кількості та розмірі ALU (арифметична логічна одиниця). Виходячи із розміру ALU можна порівняти кількість транзисторів виділене під кожне ядро.

На рисунку 2 можна більш детально зрозуміти як влаштована архітектура апаратного прискорювача тривимірної графіки.

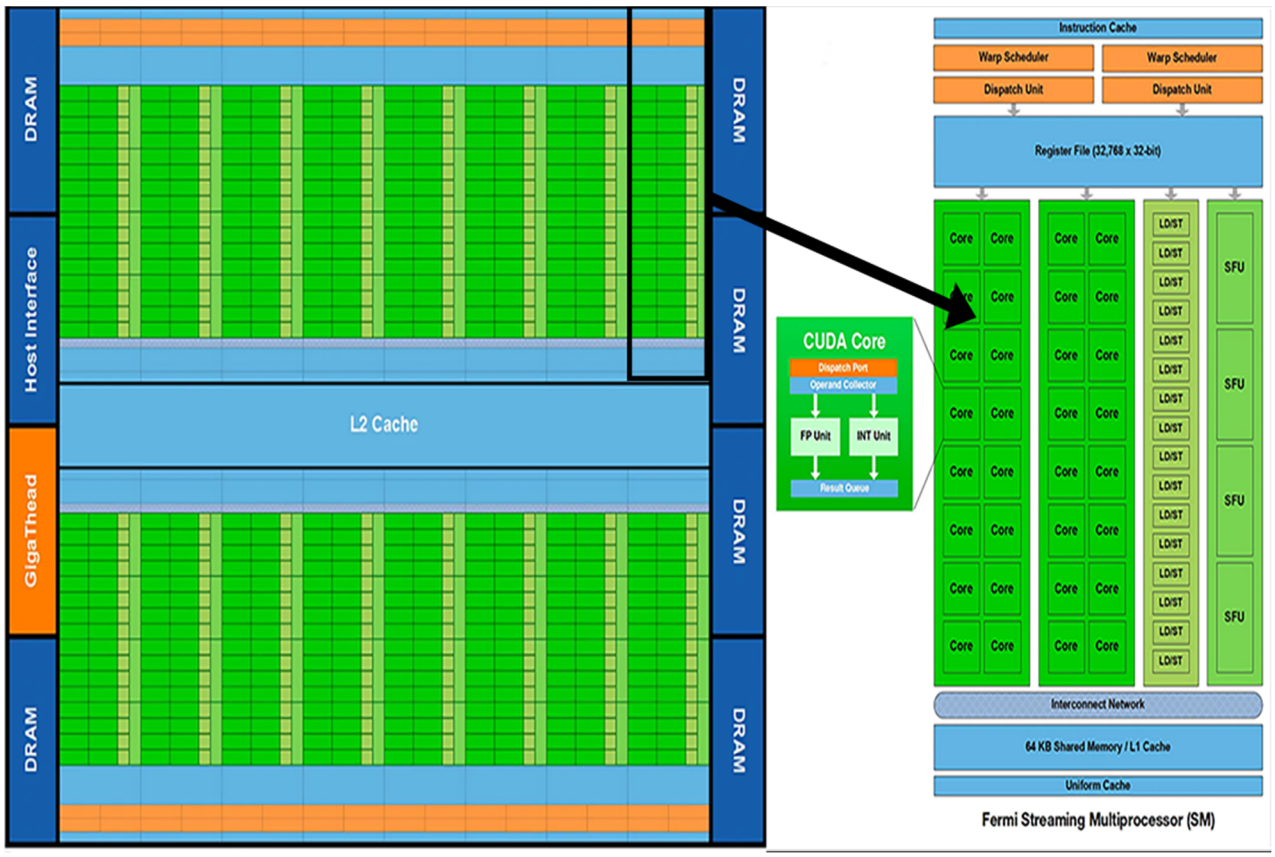



Рисунок 2 – Детальне зображення архітектури апаратного прискорювача тривимірної графіки. Nvidia GTX 480

На відміну від центрального процесора, у графічного процесора кеш розділений між декількома ядрами. Для прикладу кеш L1 доступний для сегменту ядер, а L2 для всіх сегментів. Це сильно відрізняється від центрального процесора, де кожне ядро має один або два рівня кешів і спільний кеш між всіма ядрами. І розмір кеша для одного ядра виходить в рази більше.

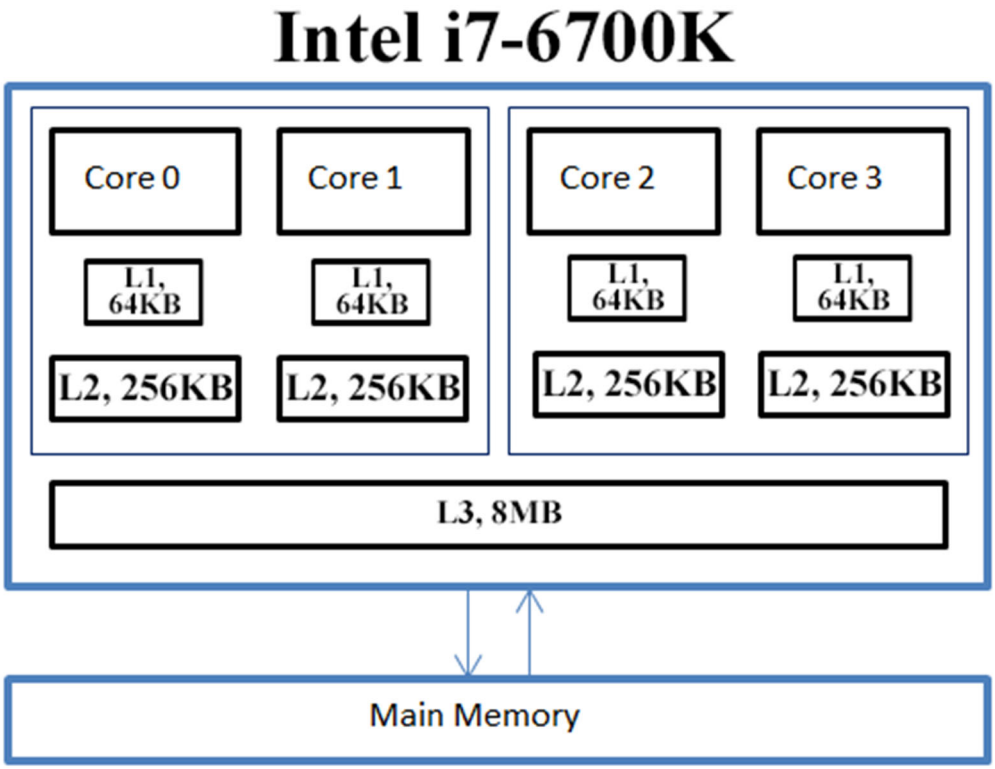



Рисунок 3 – Схематичне зображення ядер сучасного центрального процесора та його кешів

На рисунку 3 можна побачити що розмір кеша L1 – 64 кБайт у процесора intel core i7-6700K, і виділене воно під одне ядро, коли у апаратного прискорювача на попередній картинці можна побачити що розмір кеша L1 такий же, але розділене воно між всім сегментом, який складається з 32 ядер. Розмір L2 кешу на центральному процесорі дорівнює 256 кБайт і також воно виділене окремо під кожне ядро, а в апаратного прискорювача, виходячи із документації розмір L2 кешу дорівнює 768 кБайт і розділене воно між всіма ядрами. На рисунку 2 модель апаратного прискорювача яка була презентована у 2010 році, тому якщо подивитися на розмір кешів у останній моделі, Nvidia RTX 4090, то кількість сегментів виросла з 16, у моделі на рисунку 2, до 128 у сучасній моделі. А розмір кешів L1 та L2 дорівнює 128 кБайт та 72 МБайти відповідно. Повертаючись до рисунку 3, можна також помітити що у центрального процесора присутній ще один рівень кешу L3, який вже є спільним для всіх ядер.

Тобто якщо підсумувати, центральному процесору потрібен більший розмір кешу, щоб зберігати в кешах більше інформації, це в свою чергу зменшить кількість звертань у повільну оперативну пам’ять, що дозволяє швидко переключатись між задачами. Коли на графічному процесорі, розмір кеша малий та розділений між декількома десятками ядер, що при виконанні команди розгалуження може призвести до простою деяких ядер сегменту, що зменшить ефективність відеокарти. Тому програмісту при написанні коду, для виконання на апаратному прискорювачі, необхідно уникати умовних операторів.

Також ядра центрального процесору працюють на суттєво вищих тактових частотах аніж графічний процесор, що зменшує затримку у відклику.

Тепер коли ми розуміємо призначення апаратного прискорювача тривимірної графіки можна перейти до готових рішень які дозволяють використовувати його. Для початку розглянемо Unreal Engine.

- **Unreal Engine** (UE) – це потужне програмне середовище для розробки ігор та 3D візуалізацій з вражаючою та реалістичною графікою, яка побудована на фізиці відбивання світла від різних об’єктів. Це можливо завдяки системі освітлення Lumen, яка забезпечує реалістичне освітлення в реальному часі. Також в цьому програмному середовищі є система геометрії Nanite, яка дозволяє відмальовувати деталізовані об'єкти без необхідності вручну створювати геометрії із різними ступенями деталізації. У Unreal Engine можна побачити широкий набір інструментів, що полегшують процес розробки. Він включає в себе візуальний редактор Blueprints, який дозволяє створювати логіку гри без необхідності програмування, а також вбудовану систему фізики, анімації, частинок та багато іншого. Крім того він підтримує розробку ігор для різних платформ, включаючи ПК, консолі, мобільні пристрої та

пристрої віртуальної реальності. Це дозволяє розробникам легко переносити свої проекти на різні платформи без значних зусиль.

Перевагами Unreal Engine є:

- велика кількість інструментів для створення реалістичної графіки. Unreal Engine відомий своїми технологіями, які дозволяють малювати фото-реалістичну графіку використовуючи сучасні технології апаратних прискорювачів тривимірної графіки;
- кросплатформеність. Unreal Engine дозволяє створювати програми для запуску на різних платформах, таких як, персональні комп'ютери, ігрові приставки, мобільні пристрої, ВР гарнітури та інше;
- велика база користувачів. Це дозволяє знайти відповідь на своє питання або проблему в інтернеті досить просто. Адже можливе це питання поставало раніше, або задати його і швидко отримати відповідь на нього від інших користувачів Unreal Engine;
- внутрішній ринок ресурсів. У Unreal Engine є свій ринок ресурсів, таких як, текстури, шейдери, моделі і тд, де користувачі можуть ділитися один з одним, ними, без проблем з інтегруванням їх в програмне середовище;
- велика кількість розширень та додатків, які дозволяють розширити функціонал Unreal Engine;
- візуальний редактор. Цей програмний продукт має потужний візуальний редактор, який дозволяє розробникам швидко створювати та редагувати ігрові об'єкти, сцени та інші елементи гри. Це полегшує розробку і прототипування без необхідності написання великої кількості коду;
- підтримка сучасних низькорівневих графічних API.

Недоліками є:

- складність програмного середовища;
- продуктивність. Незважаючи на те, що Unreal Engine відомий своїми захоплюючими графічними ефектами, він може потребувати значних ресурсів. Розробка високоякісних графічних продуктів з використанням Unreal Engine може вимагати потужного обладнання та мати погану продуктивність на менш потужних пристроях або старому обладнанні;
- вартість. Unreal Engine можна використовувати безкоштовно у некомерційних цілях, але при використанні його у комерційних цілях, є необхідність платити процент від прибутку.
- складність розробки мультиплатформового продукту. Цей програмний продукт не може гарантувати хорошу продуктивність на всіх платформах при використанні одного і того ж функціоналу. Тому розробникам доведеться змінювати та оптимізувати код під кожну з платформ.

❖ **Unity** – це потужний і часто використовуваний програмний продукт для розробки відеоігор. Він пропонує велику кількість інструментів для створення інтерактивної 2D та 3D графіки. Він відомий гнучкістю у ході розробки, зручним та зрозумілим інтерфейсом, що дозволяє швидко почати розробку новим користувачам. Також Unity дуже схожий на Unreal Engine.

Перевагами Unity є:

- кросплатформеність. Unity підтримує розробку ігор для різних платформ, таких як ПК, консолі, мобільні пристрої та веб-браузери;
- велика спільнота розробників;

- внутрішній ринок ресурсів;
- візуальний редактор;
- підтримка сучасних низькорівневих графічних API.

Недоліками є:

- вартість. Unity як і Unreal Engine працює по схожій системі ліцензіювання. Тобто якщо є необхідність створити комерційний продукт, то потрібно буде виплачувати процент від прибутку;
- продуктивність. Зазвичай Unity використовують як швидкий прототип, але для розробки кінцевого продукту використовують Unreal Engine або власний програмний продукт із-за низької продуктивності кінцевого продукту;
- великі розміри кінцевих згенерованих файлів;
- підтримка і використання тільки високорівневої мови програмування C#.

❖ **Godot** – безплатне програмне середовище з відкритим вихідним кодом для створення інтерактивних додатків та відеоігор.

Перевагами Godot є:

- відкритий код. Godot є повністю відкритим ігровим рушійом, що означає, що його вихідний код доступний для всіх. Це дозволяє розробникам налагоджувати, модифікувати та вдосконалювати рушій під свої потреби. Крім того, це сприяє активному співробітництву в спільноті розробників Godot;
- мультиплатформова підтримка. Godot підтримує багато платформ, включаючи ПК (Windows, macOS, Linux), мобільні пристрої (Android, iOS), веб-браузери та багато інших;
- візуальний редактор. Godot має потужний візуальний редактор, який дозволяє розробникам створювати сцени, розташовувати

об'єкти, встановлювати властивості та взаємодіяти зі своїми об'єктами з використанням простого та інтуїтивно зрозумілого інтерфейсу. Це дозволяє швидко прототипувати ігрові ідеї та швидко розробляти ігри без значного написання коду;

- скриптовий рушій. Godot використовує свій власний скриптовий рушій, який базується на мові GDScript. GDScript є простою інтерпретованою мовою, яка подібна до Python, Крім того, Godot також підтримує інші мови програмування, такі як C#, C++, VisualScript тощо;
- активна спільнота. Godot має активну спільноту розробників, яка надає підтримку, документацію, навчальні матеріали та взаємодопомогу. Це робить Godot дружнім середовищем для новачків у галузі розробки ігор та надає доступ до великої кількості ресурсів;
- розширення та плагіни. Godot дозволяє розробникам створювати власні розширення та плагіни для розширення можливостей рушія. Це відкриває широкі можливості для створення спеціалізованих інструментів, бібліотек та компонентів, які можна використовувати у власних проектах.

Недоліками є:

- обмежена підтримка 3D-графіки. Godot позначається більше як ігровий рушій для 2D-графіки, ніж для 3D;
- використання OpenGL ES в якості графічної API. Це обмежує використання останніх технологій створених розробниками апаратних прискорювачів тривимірної графіки;
- немає підтримки ігрових приставок. Ігрові приставки використовують свої власні графічні API і поширюють їх тільки під договором про конфіденційність. А так як Godot це програмне

середовище з відкритим вихідним кодом, то поширювати графічні API ігрових приставок неможливо.

**Висновки**.

Сьогодні існують багато рішень для генерації тривимірної графіки у реальному часі, які дозволяють використовувати апаратний прискорювач, але в кожного з них є свої недоліки та переваги. Задля максимально ефективного використання потужностей процесорів, центрального та графічного, та пам’яті, є необхідність у використанні сучасних низькорівневих графічних API, які дозволяють краще керувати ресурсами комп’ютера. Також, у порівнянні з більш старими високорівневими графічними бібліотеками, вони підтримують новітні технології розроблені виробниками апаратних прискорювачів тривимірної графіки.

Більшість із доступних на ринку програмних середовищ потребують сплати за користування продуктом та перевантажені інструментами, які можуть не використовуватись в кінцевому продукті, що може погано позначитись на продуктивності продукту. Тому зі створенням власного програмного продукту, зазвичай створюють власне програмне середовище, яке буде мати тільки ті функції які необхідні продукту.

Використання готових алгоритмів для управління апаратним прискорювачем тривимірної графіки, може значно спростити розробку програмного середовища.

# 2. ГРАФІЧНІ БІБЛІОТЕКИ ДЛЯ КЕРУВАННЯ АПАРАТНИМ ПРИСКОРЮВАЧЕМ

Для ефективного керування функціями відеокарти і використання її потенціалу необхідне використання спеціальних бібліотек або API. Ці бібліотеки дозволяють розробникам програмного забезпечення взаємодіяти з апаратними можливостями відеокарт.

Існує велика кількість API, такі як DirectX9, OpenGL, WebGL, DirectX 11, OpenCL, Vulkan, Metal, WebGPU, DirectX 12 і кожна з них має свої переваги та недоліки. Для прикладу, DirectX 9 та OpenGL не потребує багато знань про архітектуру апаратного прискорювача. Недоліками є відсутність підтримки нових технологій апаратних прискорювачів та повільна робота на сучасних версіях графічних процесорів. Тепер детальніше про кожну із них.

- **OpenGL** (Open Graphics Library) – крос-платформовий прикладний програмний інтерфейс для 2D та 3D візуалізації векторної графіки з відкритим програмним кодом. Цей інтерфейс часто використовується для взаємодії із графічною процесорною одиницею для досягнення апаратно-прискореної візуалізації. Була розроблена Silicon Graphics, Inc. (SGI) у 1992 році для використання у сферах автоматизованого комп’ютерного проектування, віртуальної реальності, наукової візуалізації, симуляції польотів та багато іншого. У 2006 році управління та розвиток було передане некомерційній організації Khronos Group. Пізніше з’явились нові версії OpenGL ES для мобільних пристроїв та WebGL для веб.

  Перевагами OpenGL є:

  - кросплатформеність – можливість запускати програму, написану з використанням OpenGL на різних операційних системах та платформах таких як Windows, macOS, Linux та інші;

- відкритий програмний код. Це полегшує розуміння документації та дозволяє написання власних функцій та розширень.

Недоліками є:

- відсутність єдиного API. Різні платформи, апаратні прискорювачі, драйвери можуть підтримувати різні версії OpenGL, різні розширення. Це призводить до великих складнощах розробки програми під різні пристрої;
- відсутність підтримки останніх технологій, які пропонують сучасні апаратні прискорювачі. Таких як трасування променів у реальному часі, mesh шейдери і тд.;
- обмежене низькорівневе керування відеокартою. Велика кількість абстракцій призводить до гіршої продуктивності ніж при використанні низькорівнений графічних API;
- функціональний стиль програмування. Відсутність об'єктно орієнтованої архітектури ускладнює написання гнучкого коду який простіше читати та розуміти.

❖ **DirectX** – це колекція APIs для вирішення задач пов'язаних здебільшого з програмуванням ігор, 2D та 3D графіки, відео-редакторів тощо на платформах від Microsoft. Також використовується для автоматизованого комп'ютерного проектування. Дев'ята версія була представлена в 2002 році для Windows 98, Me та XP. Була дуже популярна протягом 2000-2010 роки серед творців ігор. Існують багато програмних продуктів, які дотепер використовують цей програмний інтерфейс.

Перевагами DirectX 9 є:

- широка сумісність. DirectX 9 до сих пір підтримується новими версіями Windows. А з моменту випуску пройшло уже 21 рік;
- об'єктно орієнтований стиль API.

Недоліками є:

- відсутність підтримки останніх технологій, які пропонують сучасні апаратні прискорювачі;
- обмежене керування відеокартою. Велика кількість абстракцій призводить до гіршої продуктивності ніж при використанні низькорівнених графічних API;
- обмеження на підтримуванні платформи Windows.

❖ **DirectX 11** – це одинадцята версія графічної та мультимедійної бібліотеки DirectX, ключовими змінами якої стали додавання нових програмованих шейдерних етапів, окремий шейдер для вираховувань та багатопоточне заповнення команд для виконання на апаратному прискорювачі.

Перевагами DirectX 11 є:

- більша кількість програмованих шейдерних етапів. Таких як domain шейдер, шейдер геометрії, шейдер вираховувань;
- краще використання багатопоточності сучасних центральних процесорів, за рахунок можливості заповнення команд на виконання для відеокарти на різних потоках асинхронно;
- об’єктно орієнтований стиль API.
- широка сумісність. Є підтримка у нових версіях Windows.

Недоліками є:

- відсутність підтримки останніх технологій, які пропонують сучасні апаратні прискорювачі;
- обмежене керування відеокартою. Велика кількість абстракцій призводить до гіршої продуктивності ніж при використанні низькорівнених графічних API;
- обмеження на підтримуванні платформи Windows.

❖ **Vulkan** – це низькорівневий, крос-платформовий API (інтерфейс програмування додатків) для програмування графіки та математичних обчислень використовуючи апаратні прискорювачі тривимірної графіки, розроблений Khronos Group, консорціумом технологічних компаній. Вперше він був випущений у 2016 році як наступник OpenGL і має на меті забезпечити високу продуктивність та ефективний доступ до можливостей графіки та обчислень на різноманітних пристроях, включаючи ПК, мобільні пристрої та консолі.

Перевагами Vulkan є:

- висока продуктивність. За рахунок низької кількості абстракцій, на виконання однієї і тієї ж задачі теоретично буде витрачено менше пам’яті та процесорного часу. Але все залежить від ефективності написаного програмістом коду;
- кросплатформеність;
- паралелізм та мультипоточність. Є можливість формувати і заповнювати списки команд, для виконання на графічному процесорі на різних процесорних потоках. Це дозволяє ефективно розподіляти завдання між потоками, що призводить до покращення продуктивності та кращого використання ресурсів;
- підтримка новітніх технологій. Є підтримка майже всіх новітніх технологій представлених розробниками апаратних прискорювачів.

Недоліками є:

- складність розробки. Вулкан це низькорівнене графічне API, яке потребую від програміста контролю над ресурсами, пам’яттю, синхронізаціями та іншим;
- відсутність єдиного API. Різні платформи, апаратні прискорювачі, драйвери можуть підтримувати різні версії Vulkan, різні

розширення, яке призводить до великих складнощах розробки програми під різні пристрої;

- складність вивчення. Через свій низькорівневий характер та збільшену складність, вивчення вулкану може бути складнішим для розробників, які тільки починають займатися програмуванням графіки або які працювали тільки з викорівневими API. Для успішного використання Vulkan потрібно мати хороше розуміння архітектури апаратного прискорювача тривимірної графіки, методами синхронізації, управління пам'яттю та інших низькорівневих концепцій;
- офіційна підтримка новітніх технологій зазвичай з’являється пізніше ніж у конкурентів;
- функціональний стиль програмування.

❖ **DirectX 12** – це дванадцята версія графічної та мультимедійної бібліотеки DirectX. Головною ознакою цієї версії є показ покращеного низькорівневного графічного API Direct3D 12, яке має знизити роботу драйвера. Розробники можуть створювати свої власні списки команд та черги, для виконання на апаратному прискорювачеві тривимірної графіки.

Перевагами DirectX 12 є:

- висока продуктивність. За рахунок низької кількості абстракцій, на виконання однієї і тієї ж задачі теоретично буде витрачено менше пам’яті та процесорного часу. Але все залежить від ефективності написаного програмістом коду;
- паралелізм та мультипоточність. Є можливість формувати і заповнювати списки команд, для виконання на графічному процесорі на різних процесорних потоках. Це дозволяє ефективно

розподіляти завдання між потоками, що призводить до покращення продуктивності та кращого використання ресурсів;

- підтримка новітніх технологій. Є підтримка майже всіх новітніх технологій представлених розробниками апаратних прискорювачів.
- об’єктно орієнтований стиль API;
- вбудована підтримка в Windows. Немає необхідності встановлювати додаткове програмне забезпечення для запуску програм з використанням DirectX12 в операційних системах Windows.

Недоліками є:

- складність розробки. DirectX 12 це низькорівнене графічне API, яке потребую від програміста контролю над життям ресурсів, локальною пам’яттю графічного процесора, синхронізаціями процесорів та графічних ядер, та іншим;
- складність вивчення;
- менша кількість підтримуваних платформ на відміну від Vulkan;
- немає офіційної підтримки Windows 8, Windows 7.

Всі з вище перерахованих графічних API написанні мовами програмування C або C++, тому написання програми керування апаратним прискорювачем здійснювалось з використанням мови програмування C++.

**Висновки.**

Використання старих високорівневих графічних API полегшує програмування апаратного прискорювача, але ціною великої кількості абстракцій, поганого застосування багатопочності сучасних центральних процесорів та найважливіше відсутності підтримки сучасних технологій графічних процесорів. Натомість у сучасних графічних API відсутні ці

недоліки, але дуже сильно ускладнилась розробка програмного продукту, адже програмний інтерфейс сильно ускладнився. Тому для розробки алгоритму для взаємодії з апаратним прискорювачем було використано низькорівневу графічну API DirectX 12, адже вона дозволяє створити ефективне програмне забезпечення без великої кількості абстракцій з підтримкою сучасних технологій.

# 3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ВЗАЄМОДІЇ ІЗ АПАРАТНИМ ПРИСКОРЮВАЧЕМ

## 3.1. Алгоритми для взаємодії апаратним прискорювачем

**Встановлення динамічних ресурсів для шейдеру.**

У DirectX 12 існує кореневий опис, який нагадує описання або оголошення API функції, де ми визначаємо тип ресурсу на який очікує шейдер, але не визначаємо ресурс або його дані. Кореневий опис може складатися з 3 типів параметрів, таких як, коренева константа, кореневий дескриптор (об’єкт який зберігає адрес пам’яті та деяку інформацію про цю пам’ять, як її читати або писати) та таблиця дескрипторів, а точніше вказівник на діапазон дескрипторів. Кореневий опис може містити до 64 DWORDs, що дорівнює 256 байтів. Також розробники апаратних прискорювачів рекомендують використовувати саме перші 10 параметрів, для максимальної швидкості доступу. Тепер перейдем до розмірів кожного з типів параметрів та їх переваги й недоліки.

Коренева константа – це 32 бітне значення яке зберігається напряму в кореневому описі. Швидкість доступу апаратного прискорювача до цієї пам’яті максимальна, адже відбувається без непрямих переходів по пам’яті. Але в одному параметрі можливе зберігання лише 32 бітного значення.

Кореневий дескриптор або сирий кореневий дескриптор зберігає в собі 64 бітний вказівник на пам’ять. Назва сирий означає що він не зберігає в собі інформацію про пам’ять на яку вказує, тому його використання обмежене для константних буферів та сирих UAV або SRV буферів, де UAV (Unordered Access View) – вид на випадковий доступ до пам’яті ресурсу, та SRV (Shader Resource View) – вид на ресурс шейдера, який дозволяє тільки читати. Перевага цього типу параметру кореневого опису в тому, що тут буде лише один непрямий перехід при доступу до ресурса та відсутність необхідності копіювати дескриптор в загальний буфер дескрипторів, а

недоліками будуть обмеження в зберіганні лише сирих дескрипторів та розмір в 2 DWORDs.

Таблиця дескрипторів це 32 бітне значення яке зберігає зміщення в буфері дескрипторів. Це зміщення вказує на масив дескрипторів, а кожний із них вказує на ресурс в буфері ресурсів. Перевагами є зберігання великої кількості повноцінних дескрипторів використовуючи лише один параметр кореневого опису, який займає 1 DWORD. Недоліками є 2 рівня непрямого переходу при доступу до пам’яті. Також недоліком є потреба в копіюванні дескрипторів в буфер дескрипторів апаратного прискорювача зберігаючи порядок який оголошений в шейдері. Тобто якщо шейдер очікує що на першому місці буде дескриптор на ресурс 2, на другому – ресурс 1, то нам необхідно відсортувати дескриптори в буфері дескрипторів в порядку, дескриптор на ресурс 2, дескриптор на ресурс 1.

Кореневий опис необхідно передавати при створенні об’єкту стана конвеєра. А заповнювати опис перед викликом команди малювання (Draw) або команди розподілу (Dispatch). Також у кореневий дескриптор можна додати обмеження про використання його аргументів в різних шейдерних етапах, по прикладу використання їх тільки в вершинному шейдері або піксельному. Це повинно покращити продуктивність, адже драйвер може оптимізувати швидкодію маючи інформацію про місце використання параметрів кореневого опису.

Для спрощення встановлення динамічних ресурсів для шейдера, було розроблено уніфіковану схему параметрів кореневого опису. Це значно спрощує встановлення ресурсів користувачеві.

Схема кореневого опису для шейдеру малювання складається з:

- **N** кореневих дескрипторів для константних буферів у вершинному шейдері, де **N** – кількість буферів які очікує шейдер;
- **N** кореневих дескрипторів для константних буферів у піксельному шейдері, де **N** – кількість буферів які очікує шейдер;
- Якщо піксельний шейдер очікує дескриптори типу SRV, то додається таблиця дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів, цього типу;
- Якщо піксельний шейдер очікує Sampler (правила для читання текстури), то додається таблиця дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів типу Sampler;

Схема кореневого опису для шейдеру вираховувань:

- **N** кореневих дескрипторів для константних буферів, де **N** – кількість буферів які очікує шейдер;
- Якщо шейдер вираховувань очікує дескриптори типу SRV, то додається таблиця дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів, цього типу;
- Якщо шейдер вираховувань очікує дескриптори типу UAV, то додається таблиця дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів, цього типу;
- Якщо шейдер вираховувань очікує Sampler (правила для читання текстури), то додається таблиця дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів типу Sampler;

Та програмний код для створення кореневого опису буде виглядати так.

```
bool Direct3D12Renderer::createGraphicsRootSignature(ID3D12RootSignature
**rootSig, const uint numVSCBuffers, const uint numPSCBuffers, const uint
numPSSRVs, const uint numPSSamplers) const {
	const uint paramsCount = numVSCBuffers + numPSCBuffers + bool(numPSSRVs) +
bool(numPSSamplers);
	D3D12_ROOT_PARAMETER1 *params = new D3D12_ROOT_PARAMETER1[paramsCount];
	uint slot = 0;
	for (size_t i = 0; i < numVSCBuffers; i++) {
		params[slot].Descriptor = CD3DX12_ROOT_DESCRIPTOR1(i, 0,
D3D12_ROOT_DESCRIPTOR_FLAG_DATA_VOLATILE);
		params[slot].ParameterType = D3D12_ROOT_PARAMETER_TYPE_CBV;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_VERTEX;
	}
	for (size_t i = 0; i < numPSCBuffers; i++) {
		params[slot].Descriptor = CD3DX12_ROOT_DESCRIPTOR1(i, 1,
D3D12_ROOT_DESCRIPTOR_FLAG_DATA_VOLATILE);
		params[slot].ParameterType = D3D12_ROOT_PARAMETER_TYPE_CBV;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL;
	}
	auto rangePSSRVs =
CD3DX12_DESCRIPTOR_RANGE1(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, numPSSRVs, 0);
	if (numPSSRVs) {
		params[slot].DescriptorTable = CD3DX12_ROOT_DESCRIPTOR_TABLE1(
			1, &rangePSSRVs);
		params[slot].ParameterType =
D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL;
	}

	auto rangePSSamplers =
CD3DX12_DESCRIPTOR_RANGE1(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, numPSSamplers, 0);
	if (numPSSamplers) {
		params[slot].DescriptorTable = CD3DX12_ROOT_DESCRIPTOR_TABLE1(
			1, &rangePSSamplers);
		params[slot].ParameterType =
D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
		params[slot].ShaderVisibility = D3D12_SHADER_VISIBILITY_PIXEL;
	}

	const auto rootSigDesc = CD3DX12_VERSIONED_ROOT_SIGNATURE_DESC(paramsCount,
params, 0, 0,

D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

	ComPtr<ID3DBlob> rootSigBlob, errorBlob;
	if (FAILED(D3D12SerializeVersionedRootSignature(&rootSigDesc, &rootSigBlob,
&errorBlob))) {
		OutputDebugString((LPSTR)errorBlob->GetBufferPointer());
		return false;
	}
	delete[] params;
	if (FAILED(device->CreateRootSignature(0, rootSigBlob->GetBufferPointer(),
rootSigBlob->GetBufferSize(), IID_PPV_ARGS(rootSig)))){
		ErrorMsg("Can't create Graphics Root Signature");
		return false;
	}
	return true;
}
```

*numVSCBuffers* – кількість оголошених константних буферів у вершинному шейдері.

*numPSCBuffers* – кількість оголошених константних буферів у піксельному шейдері.

*numPSSRVs* – кількість оголошених дескрипторів типу SRV у піксельному шейдері.

*numPSSamplers* – кількість оголошених дескрипторів типу Sampler у піксельному шейдері.

Як і вказувалось в схемі, на початку ми заповнюємо слоти кореневого опису сирими дескрипторами константних буферів для вершинного та піксельного шейдерів. Далі якщо *numPSSRVs > 0,* то додаємо таблицю дескрипторів із відповідним розміром. І так само для Sampler. Далі ми компілюємо кореневий опис і повертаємо його.

Програмний код для шейдеру вираховувань виглядає подібним чином. Відрізняється він відсутністю константних буферів для другого шейдерного етапу та наявністю таблиці дескрипторів типу UAV.

```
bool Direct3D12Renderer::createComputeRootSignature(ID3D12RootSignature**
rootSig, const uint nCBuffers,
	const uint numSRVs, const uint numUAVs, const uint numSamplers) const {

	const uint paramsCount = nCBuffers + bool(numSRVs) + bool(numUAVs) +
bool(numSamplers);
	D3D12_ROOT_PARAMETER1 *params = new D3D12_ROOT_PARAMETER1[paramsCount];
	uint slot = 0;
	for (size_t i = 0; i < nCBuffers; i++) {
		params[slot].Descriptor = CD3DX12_ROOT_DESCRIPTOR1(i, 0,
D3D12_ROOT_DESCRIPTOR_FLAG_DATA_VOLATILE);
		params[slot].ParameterType = D3D12_ROOT_PARAMETER_TYPE_CBV;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
	}

	auto rangeSRVs = CD3DX12_DESCRIPTOR_RANGE1(D3D12_DESCRIPTOR_RANGE_TYPE_SRV,
numSRVs, 0);
	if (numSRVs) {
		params[slot].DescriptorTable = CD3DX12_ROOT_DESCRIPTOR_TABLE1(
			1, &rangeSRVs);
		params[slot].ParameterType =
D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
	}
	auto rangeUAVs = CD3DX12_DESCRIPTOR_RANGE1(D3D12_DESCRIPTOR_RANGE_TYPE_UAV,
numUAVs, 0);
	if (numUAVs) {
		params[slot].DescriptorTable = CD3DX12_ROOT_DESCRIPTOR_TABLE1(
			1, &rangeUAVs);
		params[slot].ParameterType =
D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
	}

	auto rangeSamplers =
CD3DX12_DESCRIPTOR_RANGE1(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, numSamplers, 0);
	if (numSamplers) {
		params[slot].DescriptorTable = CD3DX12_ROOT_DESCRIPTOR_TABLE1(
			1, &rangeSamplers);
		params[slot].ParameterType =
D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
		params[slot++].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
	}

	const auto rootSigDesc = CD3DX12_VERSIONED_ROOT_SIGNATURE_DESC(paramsCount,
params, 0, 0,

D3D12_ROOT_SIGNATURE_FLAG_NONE);
	ComPtr<ID3DBlob> rootSigBlob, errorBlob;
	if (FAILED(D3D12SerializeVersionedRootSignature(&rootSigDesc, &rootSigBlob,
&errorBlob))) {
		ErrorMsg((LPSTR)errorBlob->GetBufferPointer());
		return false;
	}
	delete[] params;
	if (FAILED(device->CreateRootSignature(0, rootSigBlob->GetBufferPointer(),
rootSigBlob->GetBufferSize(), IID_PPV_ARGS(rootSig)))){
		ErrorMsg("Can't create Compute Root Signature");
		return false;
	}
	return true;
	}
```

Далі маючи уніфіковану схему кореневого опису, можна вказуючи тільки задану назву ресурса в шейдері, встановлювати його дескриптор або Sampler в правильному порядку без потреби отримання додаткових даних від користувача.

Код виставлення ресурсу в заданий шейдер буде виглядати так: Для початку користувачеві необхідно вказати унікальний ідентифікатор шейдеру.

```
void Direct3D12Renderer::setShader(const ShaderID shader);
```

Далі викликати функцію для виставлення ресурсу, знаючи її назву в шейдері та унікальний ідентифікатор, який отриманий при створенні цього ресурсу.

```
void Direct3D12Renderer::setTexture(const char *textureName, const TextureID texture);
void Direct3D12Renderer::setSamplerState(const char *samplerName, const SamplerStateID samplerState);
```

Та викликати команду малювання

```
void Direct3D12Renderer::drawArrays(const Primitives primitives, const int firstVertex, const int nVertices);
```

Або розподілу

```
void Direct3D12Renderer::dispatch(uint ThreadGroupCountX, uint ThreadGroupCountY, uint ThreadGroupCountZ);
```

в залежності від потреби користувача.

Тепер перейдемо до реалізації кожної з функцій. Функція `setShader(const ShaderID shader)` – зберігає заданий shader в полі `ShaderID selectedShader`. Це потрібно для того, щоб алгоритм розумів для якого шейдера виконувати наступні команди користувача, такі як `setTexture()`, `setSamplerState()`, `drawArrays()`, `dispatch()`, `setShaderConstantX()` і так далі.

Реалізація функція `setTexture(const char *textureName, const TextureID texture)` виглядає так:

```
void Direct3D12Renderer::setTexture(const char *textureName, const TextureID
texture)
{
	ASSERT(selectedShader != SHADER_NONE);

	const Sampler *s = getSampler(shaders[selectedShader]->textures,
shaders[selectedShader]->nTextures, textureName);
	if (s)
	{
		changeTextureState(texture, shaders[selectedShader]->isGraphicsPS()
? D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE :
D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE);
		if (s->psIndex >= 0)
		{
			selectedTexturesPS[s->psIndex] = texture;
			selectedTextureSlicesPS[s->psIndex] = NO_SLICE;
		}
		if (s->csIndex >= 0)
		{
			selectedTexturesCS[s->csIndex] = texture;
			selectedTextureSlicesCS[s->csIndex] = NO_SLICE;
		}
	}
	else
	{
#ifdef DEBUG
		char str[256];
		sprintf(str, "Invalid texture \"%s\"", textureName);
		outputDebugString(str);
#endif
	}
}
```

На початку у нас стоїть перевірка чи користувач вибрав шейдер, якщо ні то програма аварійно завершується. Далі ми проходимось по кожній текстурі вибраного шейдеру і шукаємо відповідність імені. Якщо текстури з таким іменем не знайдено, то ми повідомляємо користувача про це і виходимо з функції. Якщо знайдено, то створюємо бар'єр для цього ресурсу, вказуючи яке буде використання цього ресурсу і додаємо його в буфер бар'єрів. Далі знаходимо індекс регістра в якому шейдер очікує побачити цей ресурс, і записуємо ідентифікатор текстури в масив `selectedTexturesX`.

Функція `setSamplerState(const char *samplerName, const SamplerStateID samplerState)` працює таким же чином.

```
void Direct3D12Renderer::setSamplerState(const char *samplerName, const
SamplerStateID samplerState)
{
	ASSERT(selectedShader != SHADER_NONE);

	const Sampler *s = getSampler(shaders[selectedShader]->samplers,
shaders[selectedShader]->nSamplers, samplerName);
	if (s)
	{
		if (s->vsIndex >= 0) selectedSamplerStatesVS[s->vsIndex] =
samplerState;
		if (s->gsIndex >= 0) selectedSamplerStatesGS[s->gsIndex] =
samplerState;
		if (s->psIndex >= 0) selectedSamplerStatesPS[s->psIndex] =
samplerState;
		if (s->csIndex >= 0) selectedSamplerStatesCS[s->csIndex] =
samplerState;
	}
	else
	{
#ifdef DEBUG
		char str[256];
		sprintf(str, "Invalid samplerstate \"%s\"", samplerName);
		outputDebugString(str);
#endif
	}
}
```

Відмінність тільки в відсутності бар'єру, адже Sampler View, це не ресурс, а тільки опис або інструкція для шейдера, як необхідно читати текстуру, та відсутності вибору зріза текстури, яке теж присутнє тільки для текстур.

Далі при виклику draw*() або dispatch(), викликаються функції applyTextures() та applySamplersState(), які для зручності обгорнуті в функцію apply().

```
void Direct3D12Renderer::apply()
{
	applyTextures();
	applySamplerStates();

	if (selectedShader != DONTCARE){
		changeShader(selectedShader);
		applyConstants();
	}
	applyVertexStates();
}
```

```
void Direct3D12Renderer::applyTextures()
{

	ASSERT(selectedShader != SHADER_NONE);

	SRVID SRVsToCopy[MAX_TEXTUREUNIT];

	if (shaders[selectedShader]->isGraphicsPS()) {
		const auto gShader =
static_cast<Graphics_Pipeline_State*>(shaders[selectedShader]);
		const uint count = fillSRV(SRVsToCopy, selectedTexturesPS, gShader-
>currentTexturesPS, selectedTextureSlicesPS, gShader->currentTextureSlicesPS,
textures.getArray());
		if (count) { //TODO VS SRVs
			SRVID newOffset;
			SRVsShaderVisibleHeap.copyDescriptors(newOffset, SRVsUAVsHeap,
SRVsToCopy, count);
			gShader->SRVsDescHeapOffset = newOffset;
		}
	}
	else {
		const auto cShader =
static_cast<Compute_Pipeline_State*>(shaders[selectedShader]);
		const uint count = fillSRV(SRVsToCopy, selectedTexturesCS, cShader-
>currentTextures, selectedTextureSlicesCS, cShader->currentTextureSlices,
textures.getArray());
		if (count) {
			SRVID newOffset;
			SRVsShaderVisibleHeap.copyDescriptors(newOffset, SRVsUAVsHeap,
SRVsToCopy, count);
			cShader->SRVsDescHeapOffset = newOffset;
		}
	}

}
```

На початку у нас стоїть перевірка чи користувач вибрав шейдер, якщо ні то програма закривається. Далі ми перевіряємо чи вибраний шейдер є для малювання, чи для вираховувань і викликаємо функцію fillSRV() із `selectedTexturesPS` або `selectedTexturesCS`. Ця функція заповнює локальний масив ідентифікаторами SRV дескрипторів в порядку, якому очікує шейдер та повертає кількість дескрипторів які необхідно скопіювати. Далі ми копіюємо ці дескриптори в буфер дескрипторів апаратного прискорювача і зберігаємо зміщення від початку буфера щоб перезаписати значення в кореневому параметрі.

Реалізація функції fillSRV() виглядає так:

```
uint fillSRV(SRVID SRVsToCopy[], const TextureID selectedTextures[], TextureID
currentTextures[],
                                    const TextureID selectedTextureSlices[],
TextureID currentTextureSlices[],
                                    const Texture* textures)
{
        // Do we need to change
        bool hasChanged = false;
        uint max = 0;
        for (uint i = 0; i < MAX_TEXTUREUNIT; ++i) {
                if (selectedTextures[i] == TEXTURE_NONE) {
                        max = i;
                        break;
                }
                if (!hasChanged && (selectedTextures[i] != currentTextures[i] ||
selectedTextureSlices[i] != currentTextureSlices[i])) {
                        hasChanged = true;
                }
        }
        if (!hasChanged)
                return 0;

        uint count = 0;

        for (; count < max; ++count) {
                if (selectedTextureSlices[count] == NO_SLICE)
                        SRVsToCopy[count] = textures[selectedTextures[count]].srv;
                else
                        SRVsToCopy[count] =
textures[selectedTextures[count]].srvArray[selectedTextureSlices[count]];

                currentTextures[count] = selectedTextures[count];
                currentTextureSlices[count] = selectedTextureSlices[count];
        }

        return count;
}
```

Реалізація функції applySamplerStates() подібна до функції для текстур, відмінність тільки в тому що немає зрізів текстури.

```
void Direct3D12Renderer::applySamplerStates()
{
	SamplerStateID SamplersToCopy[MAX_SAMPLERSTATE];

	if (shaders[selectedShader]->isGraphicsPS()) {
		const auto gShader =
dynamic_cast<Graphics_Pipeline_State*>(shaders[selectedShader]);
		const uint count = fillSS(SamplersToCopy, selectedSamplerStatesPS,
gShader->currentSamplersPS, samplerStates.getArray());
		if (count) { //TODO VS Samplers
			SamplerStateID newOffset;
			samplersShaderVisibleHeap.copyDescriptors(newOffset,
samplersHeap, SamplersToCopy, count);
			gShader->SamplerDescHeapOffset = newOffset;
		}
	}
	else {
		const auto cShader =
dynamic_cast<Compute_Pipeline_State*>(shaders[selectedShader]);
		const uint count = fillSS(SamplersToCopy, selectedSamplerStatesCS,
cShader->currentSamplerStates, samplerStates.getArray());
		if (count) {
			SamplerStateID newOffset;
			samplersShaderVisibleHeap.copyDescriptors(newOffset,
samplersHeap, SamplersToCopy, count);
			cShader->SamplerDescHeapOffset = newOffset;
		}
	}
}
```

Та відповідно функція fillSS()

```
uint fillSS(SamplerStateID SamplersToCopy[], const SamplerStateID
selectedSamplerStates[], SamplerStateID currentSamplerStates[], const
SamplerState *samplerStates)
{

	// Do we need to change
	bool hasChanged = false;
	uint max = 0;
	for (uint i = 0; i < MAX_SAMPLERSTATE; ++i) {
		if (selectedSamplerStates[i] == SS_NONE) {
			max = i;
			break;
		}
		if (!hasChanged & (selectedSamplerStates[i] !=
currentSamplerStates[i])) {
			hasChanged = true;
		}
	}
	if (!hasChanged)
		return 0;

	uint count = 0;

	for (; count < max; ++count) {
		SamplersToCopy[count] = currentSamplerStates[count] =
selectedSamplerStates[count];
	}
	return count;
}
```

Реалізація методу copyDescriptors():

```
D3D12_CPU_DESCRIPTOR_HANDLE Descriptor_heap::copyDescriptors(int& destPosition,
const Descriptor_heap& source,
	const int* sourcePositions, const int count) {

	ASSERT(!source.isShaderVisible());

	Microsoft::WRL::ComPtr<ID3D12Device> device = nullptr;
	descriptorHeap->GetDevice(IID_PPV_ARGS(&device));
	ASSERT(device);

	const auto destDescriptorHandle = addDescriptors(destPosition, count);
	for (int i = 0; i < count; ++i) {
		device->CopyDescriptorsSimple(1, getCPUDescriptorHandle(destPosition
+ i),

source.getCPUDescriptorHandle(sourcePositions[i]), descHeapType[heapType]);
	}
	return destDescriptorHandle;
}
```

Імплементація цього метода досить проста. Для початку ми викликаємо метод addDescriptors() в який передаємо необхідну кількість слотів – **count**, під копіювання дескрипторів. Далі ця функція повертає індекс для першого дескриптора в якому точно буде місце під всі дескриптори. Наступним кроком ми копіюємо їх по одному, адже порядок звідки копіюються може бути не послідовний. Копіювання виконується з буфера дескрипторів який знаходиться в локальній пам’яті центрального процесора в буфер дескрипторів пам’яті апаратного прискорювача. І в кінці ми повертаємо адресу першого дескриптора, яка не використовується в applyTextures() та applySamplerStates() та зміщення першого дескриптора.

Реалізація методу addDescriptors() виглядає так:

```
D3D12_CPU_DESCRIPTOR_HANDLE Descriptor_heap::addDescriptors(int& position, const
size_t count) {
	position = allocator->allocate(count);
	return CD3DX12_CPU_DESCRIPTOR_HANDLE(CPUStart, position, descriptorSize);
}
```

Для наочності реалізація методу ініціалізації буфера дескрипторів така:

```
void Descriptor_heap::Init(const Microsoft::WRL::ComPtr<ID3D12Device>& device,
const size_t size,
	const DescriptorType type, const bool _isShaderVisible) {
	heapType = type;
	D3D12_DESCRIPTOR_HEAP_DESC heap_desc{};
	heap_desc.NumDescriptors = numDescriptors = size;
	heap_desc.Flags = _isShaderVisible ?
D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE : D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
	heap_desc.Type = descHeapType[heapType];

	if (FAILED(device->CreateDescriptorHeap(&heap_desc,
IID_PPV_ARGS(&descriptorHeap))))
		ErrorMsg("Can't create Desc Heaps");

	if (_isShaderVisible) {
		allocator = new LinearAllocator(size);
		GPUStart = descriptorHeap->GetGPUDescriptorHandleForHeapStart();
	}
	else {
		allocator = new ForwardListAllocator(size);
		GPUStart = D3D12_GPU_DESCRIPTOR_HANDLE{};
	}
	descriptorSize = device-
>GetDescriptorHandleIncrementSize(descHeapType[heapType]);
	CPUStart = descriptorHeap->GetCPUDescriptorHandleForHeapStart();
}
```

У DirectX12 існує два типи буфера дескрипторів, який доступний для читання з шейдеру і який ні. Як раніше згадувалось, коли ми використовуємо таблицю дескрипторів в якості кореневого параметру, то нам необхідно розмістити дескриптори в буфері який доступний саме для читання з шейдеру. Також я створив різні типи аллокаторів (об'єкти які керують пам'яттю), адже використання цих буферів відрізняється. Буфер, який не є доступний для читання з шейдеру, тобто який зберігається в локальній пам'яті центрального процесору, використовується як статичний, який рідко змінюється. У моєму алгоритмі він використовується для початкового зберігання всіх створених дескрипторів. Він підтримує видалення існуючих дескрипторів, а не тільки додавання нових. Для прикладу, коли користувач видалив текстуру, то нам необхідно і видалити дескриптор.

А буфер, який доступний для читання з шейдеру, використовує простий лінійний аллокатор, з кожним викликом allocate() він збільшує поточну позицію до тих пір поки не достигне ліміту або не буде викликано функцію reset().

**Пермутація шейдерного коду та компіляція об’єктів стану конвеєра.**

У попередніх версіях DirectX та високорівневих графічних API, для виконання шейдера на апаратному прискорювачові тривимірної графіки достатньо було скомпілювати шейдерний код в байткод, створити об’єкт шейдеру використовуючи цей байткод і викликати SetShader(). Натомість у останній версії DirectX 12 або інших низькорівневих графічних API, немає можливості виставити окремо кожний шейдер та окремо настройки конвеєра, все це запаковано в один об’єкт під назвою об’єкт стану конвеєра. Він містить в собі байткод кожного з шейдерних етапів, кореневий опис, настройки растеризатора, глибини та трафарету, змішування, схему вхідних даних, формати вихідних буферів. І при зміні хоча б одного із цих параметрів, необхідно перекомпілювати весь об’єкт стану конвеєра. Тому для ефективної зміни шейдерів або будь яких з цих налаштуваннь в реальному часі, перекомпіляцію об’єкту стану конвереєра повинна виконуватись тільки один раз. Для виконання цієї задачі необхідно розробити гнучку систему, яка буде зберігати всі необхідні настройки, байткод кожного з шейдерів, скомпільований кореневий опис і тд, і при запиту на отримання об’єкту стану конвеєра з вибраними параметрами, буде перевіряти чи не був він створений раніше, якщо ні, то створити і зберегти.

Для пермутації шейдерного коду, на початку необхідно прочитати та розібрати шейдерний код, написаний напередодні програмістом, та виділити в ньому межі шейдерних етапів, таких як, вершинний шейдер та піксельний шейдер, або шейдер вираховування. Далі потрібно розділити шейдерний код по етапам та вставити в кожний з них визначення, які можна налаштовувати в реальному часі, за допомогою яких відбувається пермутація шейдерного коду. Наступним кроком буде компіляція цього коду в байткод та отримання інформації по ресурсам, які використовуються в ньому. Такої як:

- тип очікуваних динамічних ресурсів (текстура, буфер або sampler), їхнє оголошене ім’я, використаний ресурсний регістр, простір та тип використання;
- кількість та розмір константних буферів, використаний ресурсний регістр, простір, ім’я, тип та зміщення кожної змінної цього буфера.

Програмний код для пермутації та компіляції шейдерного коду виглядає так.

```
	if (vsText != NULL)
	{
		String shaderString;
		if (extra != NULL) shaderString += extra;
		if (header != NULL) shaderString += header;
		shaderString += vsText;

		const char *target = (feature_level == D3D_FEATURE_LEVEL_11_0)?
"vs_5_1" : (feature_level == D3D_FEATURE_LEVEL_10_1)? "vs_4_1" : "vs_4_0";

		if (SUCCEEDED(D3DCompile(shaderString, shaderString.getLength(),
fileName, NULL, NULL, "main", target, compileFlags, 0, &vsShader, &errorsBuf)))
		{
				D3DReflect(vsShader->GetBufferPointer(), vsShader-
>GetBufferSize(), IID_PPV_ARGS(&vsRefl));
		}
		else
		{
			ErrorMsg((const char *) errorsBuf->GetBufferPointer());
		}
		SAFE_RELEASE(errorsBuf);
	}
```

*vsText* – вказівник на початок вершинного шейдерного коду із термінальним нулем в кінці. Створюємо об’єкт типу рядок та вставляємо в нього визначення які вказані програмістом для цього шейдеру. Далі копіюємо вершинний шейдерний код в цей об’єкт і передаємо його в компілятор шейдерного коду D3DCompile(). Якщо при компіляції виникли помилки, виводимо їх на екран. Якщо помилок не виникло, то передаємо шейдерний байткод в функцію D3DReflect(), яка повертає нам інформацію про шейдер.

Наступним кроком буде читання цієї інформації і зберігання її в зручному для нас форматі для подальшого виставлення констант або динамічних ресурсів. Цю інформацію ми зберігаємо для кожного шейдеру.

```
	D3D12_SHADER_DESC vsDesc;
	if (vsRefl)
	{
		vsRefl->GetDesc(&vsDesc);

		if (vsDesc.ConstantBuffers)
		{
			gShader->nVSCBuffers = vsDesc.ConstantBuffers;

			gShader->vsCBuffers = new GfxBuffer[gShader->nVSCBuffers];

			gShader->vsConstMem = new ubyte *[gShader->nVSCBuffers];
			gShader->vsDirty = new bool[gShader->nVSCBuffers];
			ZeroMemory(gShader->vsDirty, gShader->nVSCBuffers *
sizeof(bool));

		}
	}
```

Перевіряємо чи є інформація про вершинний шейдер та отримуємо кількість константних буферів, так як в шейдері можливо мати декілька буферів.

```
		for (uint i = 0; i < gShader->nVSCBuffers; i++) {
			vsRefl->GetConstantBufferByIndex(i)->GetDesc(&sbDesc);

			auto cBufSize = Multiple256(sbDesc.Size);

			gShader->vsCBuffers[i].Init(device, cmdList, cBufSize,
VERTEX_CONSTANT_BUFFER, nullptr, true);

			gShader->vsConstMem[i] = new ubyte[cBufSize];
			for (uint k = 0; k < sbDesc.Variables; k++) {
				D3D12_SHADER_VARIABLE_DESC vDesc;
				vsRefl->GetConstantBufferByIndex(i)-
>GetVariableByIndex(k)->GetDesc(&vDesc);

				Constant constant;
				size_t length = strlen(vDesc.Name);
				constant.name = new char[length + 1];
				strcpy(constant.name, vDesc.Name);
				constant.vsData = gShader->vsConstMem[i] +
vDesc.StartOffset;
				constant.gsData = NULL;
				constant.psData = NULL;
				constant.csData = NULL;
				constant.vsBuffer = i;
				constant.gsBuffer = -1;
				constant.psBuffer = -1;
				constant.csBuffer = -1;
				constants.add(constant);
			}
		}
```

Проходимось по кожному з буферів. Перевіряємо розмір, та проводим ініціалізацію об’єктів типу GfxBuffer. Отримуємо ім’я та зміщення кожної зі змінних цього буферу від початку буфера та зберігаємо це в загальному масиві констант. Адже можливе перевикористання константного буферу різними шейдерами без зміни інформації.

Клас GfxBuffer зберігає D3D12 буфери в пам’яті апаратного прискорювача та оперативній пам’яті центрального процесора. Оновлення буферу відбувається шляхом копіювання графічним процесором даних з пам’яті центрального процесора до своєї локальної пам’яті. Так як процесори працюють асинхронно, то необхідно зберігати дані в оперативній пам’яті до тих пір, поки графічний процесор не виконає копіювання, інакше буде скопійовано неправильні дані. Тому центральному процесору потрібно слідкувати за виконанням команд копіювання графічного процесору і перезаписувати цю пам’ять тільки після виконання цих команд.

GfxBuffer виділяє пам’ять в оперативній пам’яті більше ніж потрібно, щоб у випадку, коли графічний процесор ще не виконав команду копіювання, можна було б записати інформацію в інше місце та не очікувати графічний процесор.

Далі необхідно обробити інформацію по динамічним ресурсам, які очікує шейдер.

```
		uint nMaxVSRes = vsRefl? vsDesc.BoundResources : 0;
		D3D12_SHADER_INPUT_BIND_DESC siDesc;
		for (uint i = 0; i < nMaxVSRes; i++)
		{
			vsRefl->GetResourceBindingDesc(i, &siDesc);

			if (siDesc.Type == D3D_SIT_TEXTURE)
			{
				size_t length = strlen(siDesc.Name);
				shader->textures[shader->nTextures].name = new
char[length + 1];
				strcpy(shader->textures[shader->nTextures].name,
siDesc.Name);
				shader->textures[shader->nTextures].vsIndex =
siDesc.BindPoint;
				shader->textures[shader->nTextures].gsIndex = -1;
				shader->textures[shader->nTextures].psIndex = -1;
				shader->textures[shader->nTextures].csIndex = -1;
				shader->nTextures++;
			}
			else if (siDesc.Type == D3D_SIT_SAMPLER)
			{
				size_t length = strlen(siDesc.Name);
				shader->samplers[shader->nSamplers].name = new
char[length + 1];
				strcpy(shader->samplers[shader->nSamplers].name,
siDesc.Name);
				shader->samplers[shader->nSamplers].vsIndex =
siDesc.BindPoint;
				shader->samplers[shader->nSamplers].gsIndex = -1;
				shader->samplers[shader->nSamplers].psIndex = -1;
				shader->samplers[shader->nSamplers].csIndex = -1;
				shader->nSamplers++;
			}
		}
```

Запис імені та використаного ресурсного регістру для кожного з динамічних ресурсів в шейдері.

Далі ми сортуємо всі константи та динамічні ресурси по імені, щоб при їх виставленні можна було використати бінарний пошук для пошуку їхньої позиції та регістру.

```
int constantComp(const void *s0, const void *s1)
{
	return strcmp(((Constant *) s0)->name, ((Constant *) s1)->name);
}
	memcpy(shader->constants, constants.getArray(), shader->nConstants *
sizeof(Constant));
		qsort(shader->constants, shader->nConstants, sizeof(Constant),
constantComp);
```

Наступним кроком буде створення кореневого опису, використовуючи отриману інформацію, створення вхідної схеми вершинного буферу, у випадку створення шейдеру малювання та створення

й компіляція об’єкту стану конвеєра. Якщо користувач не виставив настройки растеризатора, глибини та трафарету, змішування, то будуть використані стандартні налаштування. Хешуємо ці налаштування та зберігаємо скомпільований об’єкт стану конвеєра в хеш-таблиці для подальшого швидкого доступу до нього. Також зберігаємо скомпільований кореневий опис, параметри вхідної схеми, байткоди шейдерного коду та поточні виставленні ресурси. Тому при зміні будь яких з настройок об’єкту стану конвеєра не буде потреби в читанні, пермутації та компіляції шейдерного коду, компіляції кореневого опису, виділенні пам’яті та створення константних буферів. Це все буде відбуватися тільки один раз для кожного з шейдерів. Це дозволяє значно пришвидшити зміну шейдерів для апаратного прискорювача.

```
void Direct3D12Renderer::createPSO(ShaderID shader, RasterizerStateID rsID,
DepthStateID dsID, BlendStateID bsID) {
	ASSERT(shader != SHADER_NONE);
	if (shaders[shader]->isGraphicsPS()) {
		const auto gShader =
static_cast<Graphics_Pipeline_State*>(shaders[shader]);

		D3D12_GRAPHICS_PIPELINE_STATE_DESC pipeline_state_desc{};
		ASSERT((gShader->msaaSamples == 1 || gShader->msaaSamples % 2 == 0)
&& gShader->msaaSamples > 0 && gShader->msaaSamples <=8);

		static const D3D12_RASTERIZER_DESC defaultRasterizerState =
CD3DX12_RASTERIZER_DESC(D3D12_FILL_MODE_SOLID, D3D12_CULL_MODE_NONE, false, 0, 0,
0, false, true, false, 0, D3D12_CONSERVATIVE_RASTERIZATION_MODE_OFF);

		gShader->loadPSODesc(pipeline_state_desc);
		pipeline_state_desc.DS = D3D12_SHADER_BYTECODE({nullptr,0});
		pipeline_state_desc.HS = D3D12_SHADER_BYTECODE({nullptr,0});
		pipeline_state_desc.GS = D3D12_SHADER_BYTECODE({nullptr,0});
		pipeline_state_desc.DSVFormat = getDXGIFormat(depthStencilFormat);
		pipeline_state_desc.RTVFormats[0] = getDXGIFormat(backBufferFormat);
		pipeline_state_desc.NumRenderTargets = 1;
		pipeline_state_desc.RasterizerState = rsID == RS_NONE ?
defaultRasterizerState : rasterizerStates[rsID].rsDesc;
		pipeline_state_desc.DepthStencilState =  dsID == DS_NONE ?
CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT) : depthStates[dsID].depthStencilDesc;
		pipeline_state_desc.BlendState = bsID == BS_NONE ?
CD3DX12_BLEND_DESC(D3D12_DEFAULT) : blendStates[bsID].blendDesc;
		pipeline_state_desc.PrimitiveTopologyType =
D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
		pipeline_state_desc.SampleMask = UINT_MAX;
		pipeline_state_desc.SampleDesc = {gShader->msaaSamples,0};

		ID3D12PipelineState *pso;

		if (FAILED(device->CreateGraphicsPipelineState(&pipeline_state_desc,
IID_PPV_ARGS(&pso))))
```

```
                        ErrorMsg("Can't create Graphics PSO");

                gShader->PSOs[{rsID, dsID, bsID}] = pso;
        }
        else {
                const auto cShader =
static_cast<Compute_Pipeline_State*>(shaders[shader]);

                D3D12_COMPUTE_PIPELINE_STATE_DESC pipeline_state_desc{};

                cShader->loadPSODesc(pipeline_state_desc);

                if (FAILED(device->CreateComputePipelineState(&pipeline_state_desc,
IID_PPV_ARGS(&cShader->PSO))))
                        ErrorMsg("Can't create Compute PSO");
        }
}
```

На початку ми перевіряємо, який шейдер передано, для малювання чи для вираховування. Далі ми викликаємо функцію *shader->loadPSODesc()* для заповнення раніше скомпільованих полів. Таких як шейдерний байткод кожного з етапів, кореневий опис та параметри вхідної схеми, при використанні шейдеру для малювання. Далі ми компілюємо новий об’єкт стану конвеєра, хешуємо настройки з якими ми його зкомпілювали й зберігаємо в хеш-таблицю. Функція хешування виглядає так.

```
namespace std {
        template <>
        struct hash<PSOSettings>
        {
                size_t operator()(const PSOSettings& obj) const noexcept {
                        std::size_t h1 = hash<int>{}(obj.rasterizerID);
                        std::size_t h2 = hash<int>{}(obj.depthID);
                        std::size_t h3 = hash<int>{}(obj.blendID);
                        return h1 ^ ((h2 ^ (h3 << 1)) << 1);
                }
        };
}
```

Функції для зміни шейдеру та кореневої сигнатури.

```
void Direct3D12Renderer::changeShader(const ShaderID shaderID) {
        shaders[shaderID]->isGraphicsPS() ? changeGraphicsShader(shaderID) :
changeComputeShader(shaderID);
}
```

```
void Direct3D12Renderer::changeGraphicsShader(const ShaderID shaderID)
{
	const auto pShader =
static_cast<Graphics_Pipeline_State*>(shaders[shaderID]);
	if (selectedPSOSettings != currentPSOSettings || shaderID != currentShader)
{
		if (pShader->PSOs.find(selectedPSOSettings) == pShader->PSOs.end())
			createPSO(shaderID, selectedPSOSettings);

		cmdList->SetPipelineState(pShader->PSOs[selectedPSOSettings]);
		currentPSOSettings = selectedPSOSettings;
	}
	if (shaderID != currentShader)
	{
		cmdList->SetGraphicsRootSignature(pShader->rootSig);
		currentShader = shaderID;
	}
	{
		uint slot = 0;
		for (uint i = 0; i < pShader->nVSCBuffers; ++i)
			cmdList->SetGraphicsRootConstantBufferView(slot++, pShader-
>vsCBuffers[i].getGPUVirtualAddress());

		for (uint i = 0; i < pShader->nPSCBuffers; ++i)
			cmdList->SetGraphicsRootConstantBufferView(slot++, pShader-
>psCBuffers[i].getGPUVirtualAddress());

		if (pShader->nTextures > 0)
			cmdList->SetGraphicsRootDescriptorTable(slot++,
SRVsShaderVisibleHeap[currentBufIdx].getGPUDescriptorHandle(pShader-
>SRVsDescHeapOffset));
		if (pShader->nSamplers > 0)
			cmdList->SetGraphicsRootDescriptorTable(slot++,
samplersShaderVisibleHeap[currentBufIdx].getGPUDescriptorHandle(pShader-
>SamplerDescHeapOffset));

	}
	cmdList->OMSetStencilRef(selectedStencilRef);
}
```

На початку ми перевіряємо чи змінились в нас параметри об’єкту стану конвеєра, такі як параметри растеризатора, глибини та трафарету або параметри змішування та чи змінився в нас шейдер по відношенню з попередньо виставленого. Якщо один із параметрів змінився, то шукаємо об’єкт стану конвеєра з такими параметрами в хеш-таблиці, якщо він відсутній то компілюємо новий за допомогою функції *Direct3D12Renderer::createPSO()*. Також якщо в нас змінився шейдер, то необхідно встановити кореневий опис, який створений під цей шейдер. Далі ми заповнюємо кореневий опис параметрами, знаючи її схему.

Функція для встановлення шейдеру вираховувань відрізняється викликом API функцій, призначених для об’єктів стану конвеєра вираховувань, та відсутністю пошуку в хеш-таблиці, адже відсутня сама хеш-таблиця й динамічні параметри малювання.

```
void Direct3D12Renderer::changeComputeShader(const ShaderID shaderID)
{
	const auto pShader =
static_cast<Compute_Pipeline_State*>(shaders[shaderID]);
	if (shaderID != currentShader)
	{
		cmdList->SetPipelineState(pShader->PSO);
		cmdList->SetComputeRootSignature(pShader->rootSig);

		currentShader = shaderID;
	}
	{
		uint slot = 0;
		for (uint i = 0; i < pShader->nCBuffers; ++i)
			cmdList->SetComputeRootConstantBufferView(slot++,  pShader-
>cBuffers[i].getGPUVirtualAddress());
		if (pShader->nTextures > 0)
			cmdList->SetComputeRootDescriptorTable(slot++,
SRVsShaderVisibleHeap[currentBufIdx].getGPUDescriptorHandle(pShader-
>SRVsDescHeapOffset));
		if (pShader->nUAVs > 0)
			cmdList->SetComputeRootDescriptorTable(slot++,
SRVsShaderVisibleHeap[currentBufIdx].getGPUDescriptorHandle(pShader-
>UAVsDescHeapOffset));
		if (pShader->nSamplers > 0)
			cmdList->SetComputeRootDescriptorTable(slot++,
samplersShaderVisibleHeap[currentBufIdx].getGPUDescriptorHandle(pShader-
>SamplerDescHeapOffset));
	}
}
```

Структура Graphics_Pipeline_State виглядає так.

```
struct Graphics_Pipeline_State : public Shader
{
	Graphics_Pipeline_State();

	Graphics_Pipeline_State(const Graphics_Pipeline_State&) = delete;
	Graphics_Pipeline_State(const Graphics_Pipeline_State&&) = delete;
	Graphics_Pipeline_State& operator=(const Graphics_Pipeline_State&) =
delete;
	Graphics_Pipeline_State& operator=(const Graphics_Pipeline_State&&) =
delete;

	~Graphics_Pipeline_State();

	void loadPSODesc(D3D12_GRAPHICS_PIPELINE_STATE_DESC& desc) const {
		desc.VS = VS ? CD3DX12_SHADER_BYTECODE(VS) :
D3D12_SHADER_BYTECODE({nullptr,0});
		desc.PS = PS ? CD3DX12_SHADER_BYTECODE(PS) :
D3D12_SHADER_BYTECODE({nullptr,0});

		desc.pRootSignature = rootSig;
		desc.InputLayout = InputLayout;
```

```
        }

        std::unordered_map<PSOSettings, ID3D12PipelineState*> PSOs;

        ID3DBlob* VS;
        ID3DBlob* PS;

        D3D12_INPUT_LAYOUT_DESC InputLayout;

        GfxBuffer* vsCBuffers;
        GfxBuffer* psCBuffers;

        uint nVSCBuffers;
        uint nPSCBuffers;

        ubyte** vsConstMem;
        ubyte** psConstMem;

        bool* vsDirty;
        bool* psDirty;

        TextureID currentTexturesPS[MAX_TEXTUREUNIT];
        TextureID currentTextureSlicesPS[MAX_TEXTUREUNIT];

        SamplerStateID currentSamplersPS[MAX_SAMPLERSTATE];

};
```

У ній ми можемо бачити такі поля: *PSOs*, хеш-таблиця для зберігання скомпільованих об’єктів стану конвеєра з вибраними налаштуваннями; *VS* та *PS*, для зберігання байткодів скомпільованих шейдерних етапів; *InputLayout*, для збереження вхідної схеми вершинного шейдеру; *vsCBuffers* та *psCBuffers* типу *GfxBuffer**, під масив константних буферів; *vsConstMem* та *psConstMem*, для збереження виставлених констант у оперативній пам’яті; *vsDirty* та *psDirty*, для розуміння що константи змінились і необхідно перезаписати константні буфери; *currentTexturesPS*, *currentTextureSlicesPS* та *currentSamplersPS*, для визначення які дескриптори уже виставленні в кореневому описі.

Спільні поля між шейдером малювання та шейдером вираховувань були винесені в окрему абстрактну структуру *Shader*.

```
struct Shader
{
        Shader();
        virtual ~Shader();

        bool isGraphicsPS() const { return m_IsGraphicsPS; }
        bool m_IsGraphicsPS;

        ID3D12RootSignature *rootSig;
```

```
        Constant *constants;
        Sampler *textures;
        Sampler *samplers;

        uint nConstants;
        uint nTextures; //SRVs
        uint nSamplers;
        uint nUAVs;


        SRVID SRVsDescHeapOffset;
        SamplerStateID SamplerDescHeapOffset;
        SRVID UAVsDescHeapOffset;
};
```

У ній присутні такі поля: *IsGraphicsPS,* інформації про тип шейдеру; *rootSig*, скомпільований кореневий опис; *constants*, інформація про всі змінні константних буфер; *textures*, інформація про всі очікуванні дескриптори типу SRV; *samplers*, інформація про всі дескриптори типу Sampler; *nConstants, nTextures, nSamplers, nUAVs*, інформація про кількість констант та дескрипторів; *SRVsDescHeapOffset, SamplerDescHeapOffset, UAVsDescHeapOffset*, зміщення в буфері дескрипторів апаратного прискорювача для кожного з типів дескрипорів.

Структура Compute_Pipeline_State подібна до графічної, але відрізняється відсутністю хеш-таблиці, адже відсутні настройки растеризатора, глибини та трафарету й змішування, й присутнім лише одним масивом буферів констант, адже можливий тільки один шейдерний етап.

```
struct Compute_Pipeline_State : public Shader
{
        Compute_Pipeline_State();

        Compute_Pipeline_State(const Compute_Pipeline_State&) = delete;
        Compute_Pipeline_State(const Compute_Pipeline_State&&) = delete;
        Compute_Pipeline_State& operator=(const Compute_Pipeline_State&) = delete;
        Compute_Pipeline_State& operator=(const Compute_Pipeline_State&&) = delete;

        ~Compute_Pipeline_State();

        void loadPSODesc(D3D12_COMPUTE_PIPELINE_STATE_DESC& desc) const {
                desc.CS = CS ? CD3DX12_SHADER_BYTECODE(CS) :
D3D12_SHADER_BYTECODE({nullptr,0});
                desc.pRootSignature = rootSig;
        }

        ID3D12PipelineState* PSO;
        ID3DBlob* CS;
```

```
        GfxBuffer* cBuffers;
        uint nCBuffers;
        ubyte** constMem;
        bool* cDirty;
        TextureID currentTextures[MAX_TEXTUREUNIT];
        TextureID currentTextureSlices[MAX_TEXTUREUNIT];
        SamplerStateID currentSamplerStates[MAX_SAMPLERSTATE];
};
```

### 3.2. Напрямки подальшого розвитку алгоритмів

- Не копіювати дескриптори в буфер дескрипторів, якщо порядок в кореневому описі не змінився. У поточній версії алгоритму, при виставленні динамічних ресурсів, відбуваються копіювання дескрипторів в буфер дескрипторів апаратного прискорювача, в незалежності чи ресурси змінились чи ні. Це можна покращити, шляхом використання більш розумного алгоритму, який буде шукати вільне місце в буфері при зміні дескрипторів.
- Перевикористання кореневого опису між різними шейдерами. DirectX12 дозволяє нам використовувати один кореневий опис для декількох об’єктів стану конвеєра. У моєму алгоритмі цей функціонал використовується. Для прикладу, при зміні лише параметрів об’єкту стану конвеєрів, але без зміни самого шейдеру, використовується один і той же кореневий опис. Але цей функціонал можна розширити, шляхом перевикористання кореневого опису між різними шейдерами. Це потребує можливо зміни шейдерного коду та модифікації алгоритму, але зменшить кількість викликів для виставлення кореневого опису.
- Додати підтримку для інших програмованих шейдерних етапів. У DirectX12 існують такі програмовані шейдерні етапи як, вершинний (Vertex), корпусний (Hull), доменний (Domain), геометричний (Geometry), піксельний (Pixel) та шейдерний етап вираховування (Compute). Наразі, алгоритм підтримує основні етапи, вершинний, піксельний та шейдерний етап вираховування. Але алгоритм допускає

використання й інших програмованих шейдерний етапів, потрібно тільки написати імплементацію під них.

- Використання асинхронної черги вираховувань на апаратному прискорювачеві. DirectX12 дозволяє нам виконувати декілька команд вираховувань паралельно. Це дозволяє максимально заповнити ядра графічного процесору роботою і мінімізує кількість тактів на виконання однієї і тієї ж кількості задач. Поточна версія алгоритму підтримує заповнення тільки однієї черги задач апаратного прискорювача.
- Додати можливість заповнювання списків команд асинхронно. У останній версії DirectX, додали списки команд на виконання апаратним прискорювачем, які дозволяють відправляти команди на різних потоках центрального процесора. Це дозволяє краще використовувати багатопоточність сучасних центральних процесорів, адже дозволяє виконувати роботу, необхідну для формування команд для графічного процесора, асинхронно.
- Додати можливість змінювати вихідний формат малювання, глибини й трафарету в об’єкті стану конвеєра. Вихідні ресурси в шейдері малювання можуть мати різний формат, але алгоритм використовує формат головного буферу малювання. Це допустимо, але сильно обмежує функціонал і гнучкість системи.
- Додати можливість виставляти власні значення для швидкого очищення ресурсу, при його створенні.

**Висновки.**

Використання низькорівневого графічного API хоч і надає широкий контроль над апаратним прискорювачем, але змушує нас правильно контролювати пам’яттю кожного з процесорів, життям ресурсів, синхронізаціями процесорів і так далі. Це все потребує від розробників

написання алгоритмів, які будуть керувати цими функціями, мати зручний інтерфейс для контролю та могти працювати із іншими алгоритмами.

Тому був створений алгоритм для встановлення динамічних ресурсів в шейдері, який має простий інтерфейс, де необхідно вказати ідентифікатор ресурсу для встановлення, та його оголошене ім’я в шейдері. Для створення цього алгоритму було створено уніфіковану схему кореневого опису, створено алгоритм для обробки інформації про ресурси і зберіганні її в зручному для алгоритму форматі. Він також вміє сортувати дескриптори у порядку, який потребує шейдер.

Також був створений алгоритм для пермутації шейдерного коду і компіляції об’єктів стану конвеєра із різними налаштуваннями растеризатора, глибини та трафарету й змішування, без необхідності у перекомпіляції шейдерного коду й кореневого опису при зміні налаштувань. При створенні цього алгоритму, була розроблена структура для зберігання незмінної частини об’єкту стану конвеєра, такої як кореневий опис, вхідної схеми у вершинний шейдер, байткод шейдерних етапів та константні буфери шейдерів. Також було розроблено функцію хешування інформації по динамічним налаштуванням.

Також було проаналізовано подальші напрямки розвитку алгоритму, які додадуть більше функціоналу і гнучкості системі та оптимізують її швидкодію.

# 4. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ ТА ПОРІВНЯННЯ ІЗ ВИКОРИСТАННЯ ВИСОКОРІВНЕВИХ ГРАФІЧНИХ БІБЛІОТЕК

Для перевірки правильності роботи алгоритму можна використати діагностичний шар Direct3D12, режим налагодження апаратного прискорювача та порівняти отримані результати з результатами, які отримані використовуючи високорівневу графічну бібліотеку. У якості вхідних даних було використано демонстраційну програму ігрового рушія “Humus GameEngine II”. Ця програма використовує графічну API OpenGL, для керуванням апаратним прискорювачем тривимірної графіки. ЇЇ було переписано із використанням створеного алгоритму та базуючись на графічній бібліотеці DirectX12, задля перевірки працездатності алгоритму.

Діагностичний шар Direct3D12 проводить перевірку правильності викликів API команд. Правильність переданих параметрів, правильність порядку викликів і тд. Також це хороший інструмент для базового розуміння архітектури апаратного прискорювача та його програмування, адже цей шар прямо вказує на те що програміст зробив не так.

Режим налагодження апаратного прискорювача проводить велику кількість перевірок на стороні графічного процесору. Він перевіряє, чи не використовується неініціалізований або неправильний дескриптор в шейдері, чи дескриптор не вказує на видалений ресурс, чи шейдер не використовує дескриптор який виходить за межі буферу дескрипторів і тд.

При ввімкненні режиму налагодження апаратного прискорювача та діагностичного шару Direct3D12 у демонстраційній програмі, яка переписана під використання створеного алгоритму, можна помітити лише декілька однакових попереджень:

*D3D12 WARNING: ID3D12CommandList::ClearRenderTargetView: The clear values do not match those passed to resource creation. The clear operation is typically slower as a result; but will still clear to the desired value. [ EXECUTION WARNING #820: CLEARRENDERTARGETVIEW_MISMATCHINGCLEARVALUE]*

Що означає, що було викликано команду очищення текстури для малювання із значеннями, які відрізняються від тих що були передані при створенні ресурсу. Операція очищення все-одно буде виконана, але зазвичай повільніше ніж із значеннями, які були задані при створенні ресурсу.

У випадку з цією демонстраційною програмою, це допустимо, адже ці команди очищення з власними значеннями, що відрізняються від тих, що вказані при створенні ресурсу, виконуються декілька разів і тільки при запуску програми, що не впливає на подальшу продуктивність. Для вирішення цього попередження необхідно додати можливість встановлювати власні значення для швидкого очищення текстури при її створенні. Але так як це ускладнить інтерфейс створення ресурсів, а швидкодію не покращить, було вирішено відкласти імплементацію цієї можливості.

Також можна порівняти вихідні дані після генерації однієї і тієї ж тривимірної сцени використовуючи високорівневу графічну API, а саме OpenGL.

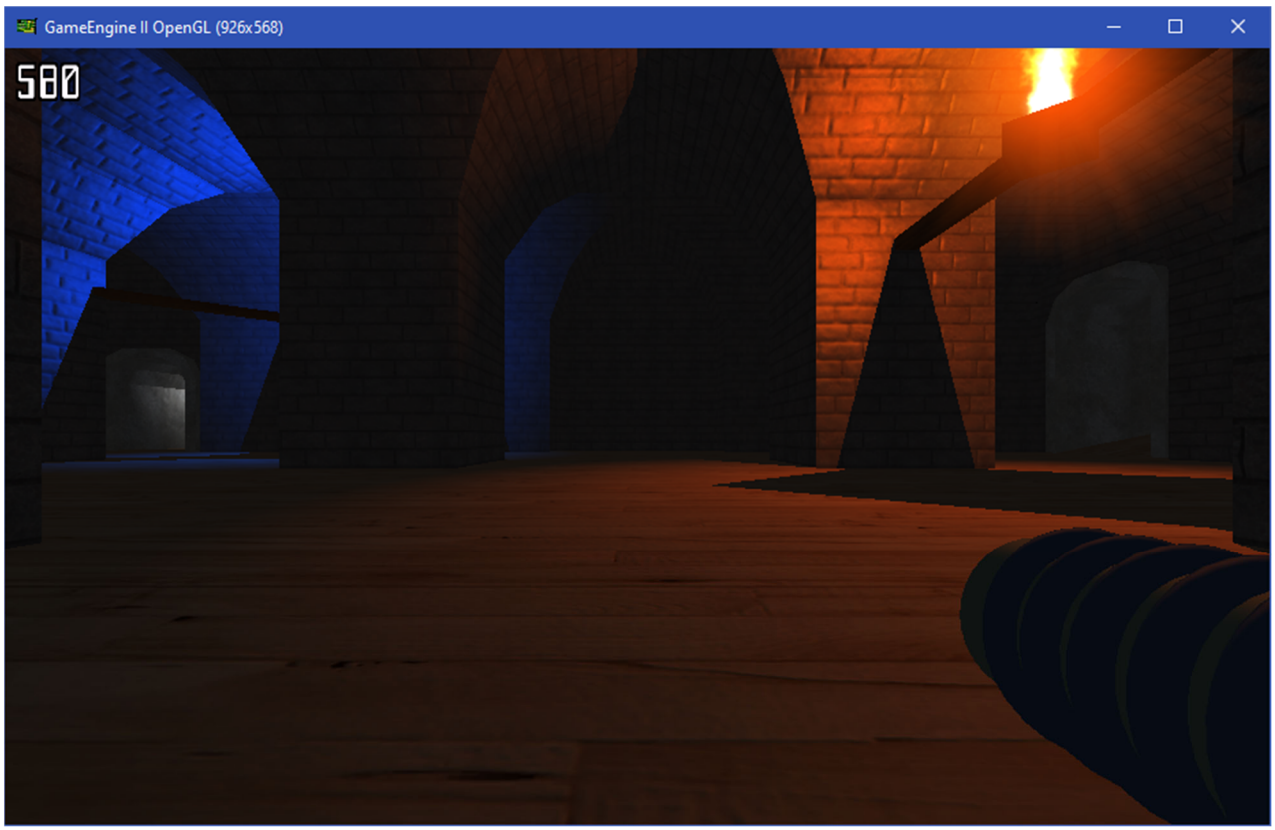



Рисунок 4 – Результат генерації тривимірної сцени демонстраційної програми “Humus GameEngine II” використовуючи високорівневу графічну бібліотеку OpenGL

Та результат генерації тривимірної сцени з використанням низькорівневої графічної API, DirectX12 та з використанням створених алгоритмів.

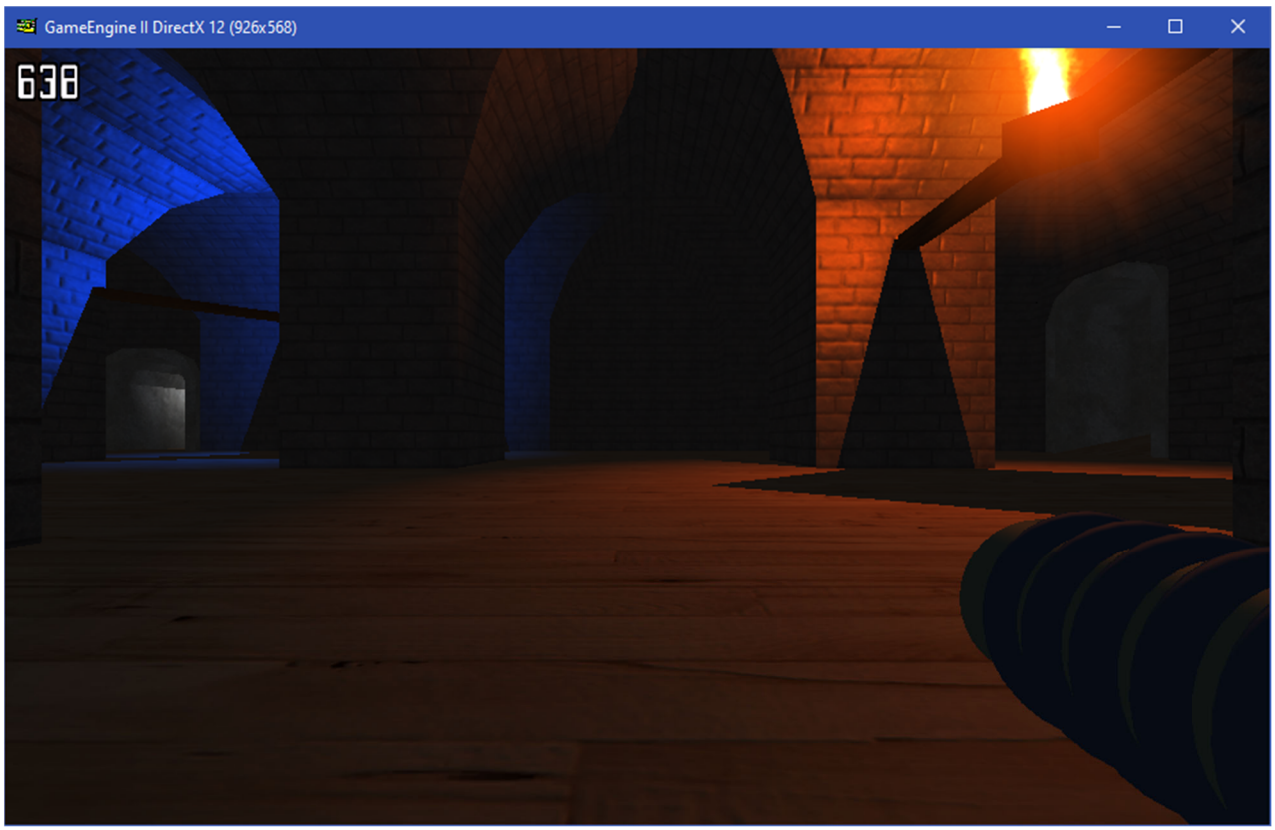



Рисунок 5 – Результат генерації тривимірної сцени демонстраційної програми “Humus GameEngine II” використовуючи низькорівневу графічну бібліотеку DirectX12

Різниці в вихідних даних не було виявлено, тому можна сказати що алгоритм працює правильно.

# ВИСНОВКИ

Тривимірна графіка, відео-ефекти, кодування та декодування відео сьогодні не обходяться без апаратних прискорювачів тривимірної графіки. Крім того, штучний інтелект, який застосовується для розв’язання складних обчислювальних завдань, таких як аналіз великої кількості даних, комп’ютерне зорове сприйняття та машинне навчання отримали значний поштовх завдяки розвитку графічних процесорів. Вони мають велику кількість обчислювальних одиниць, які дозволяють паралельно швидко виконувати матричні операції, необхідні для роботи глибоких нейронних мереж при малих споживаннях електроенергії.

У цьому дипломного проєкту було проаналізовано існуючі рішення для програмування апаратного прискорювача тривимірної графіки, такі як Unreal Engine, Unity та Godot. Кожен програмний продукт має свої переваги та недоліки. Більшість із доступних на ринку програмних середовищ потребують сплати за користування продуктом та перевантажені інструментами, що може погано позначитись на загальній продуктивності продукту. Тому зі створенням власного програмного продукту, зазвичай створюють своє власне програмне середовище, яке буде мати тільки ті функції які необхідні кінцевому продукту.

Також було проаналізовано інструменти, а також графічні бібліотеки, які надають керування апаратним прискорювачем, такі як OpenGL, DirectX9, DirectX11, DirectX12 та Vulkan. Їх можна розділити на високорівневі та низькорівневі графічні бібліотеки. Використання високорівневих графічних API полегшує програмування апаратного прискорювача, але ціною великої кількості абстракцій, поганого застосування багатопоточності сучасних центральних процесорів та найважливіше відсутності підтримки сучасних технологій графічних процесорів. Натомість у сучасних графічних API, таких як DirectX12 або Vulkan, відсутні ці недоліки, але їх використання потребує знання у

програмуванні багатопроцесорних систем, керуванні пам’яттю і базового розуміння архітектури апаратного прискорювача.

Тому для спрощення розробки програмного продукту, але без втрати підтримки нових технологій, був розроблений алгоритм, використання якого полегшує програмування апаратного прискорювача. У ньому наявна мінімальна кількість абстракцій та простий програмний інтерфейс.

Також цей алгоритм не прив’язується до наявних апаратних прискорювачів і може бути використаний для спеціалізованої апаратури, за умови написання для неї імплементації функцій алгоритму.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Програмне середовище. Unreal Engine: URL: https://www.unrealengine.com/en-US (дата звернення: 25.05.2023).
2. Програмне середовище. Unity: URL: https://unity.com/ (дата звернення: 25.05.2023).
3. Програмне середовище. Godot: URL: https://godotengine.org/ (дата звернення: 25.05.2023).
4. Прикладний програмний інтерфейс. OpenGL: URL: https://www.opengl.org/ (дата звернення: 26.05.2023).
5. Графічна бібліотека. Direct3D 9 Graphics: URL: https://learn.microsoft.com/en-us/windows/win32/direct3d9/dx9-graphics (дата звернення: 26.05.2023).
6. Графічна бібліотека. Direct3D 11 Graphics: URL: https://learn.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11 (дата звернення: 26.05.2023).
7. Графічна бібліотека. Direct3D 12 graphics: URL: https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (дата звернення: 26.05.2023).
8. Прикладний програмний інтерфейс. Vulkan: URL: https://www.vulkan.org/ (дата звернення: 26.05.2023).
9. Прикладний програмний інтерфейс. OpenCL: URL: https://www.khronos.org/opencl/ (дата звернення: 27.05.2023).
10. Прикладний програмний інтерфейс. WebGL: URL: https://www.khronos.org/webgl/ (дата звернення: 27.05.2023).
11. Прикладний програмний інтерфейс. WebGPU: URL: https://www.w3.org/TR/webgpu/ (дата звернення: 27.05.2023).
12. Прикладний програмний інтерфейс. Metal Overview: URL: https://developer.apple.com/metal/ (дата звернення: 27.05.2023).

13. CUDA C++ Programming Guide: URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (дата звернення: 20.05.2023).
14. A Review on GPU Programming Strategies and Recent Trends in GPU Computing: URL: https://www.semanticscholar.org/paper/A-Review-on-GPU-Programming-Strategies-and-Recent-Chandran-Gangodkar/4ac719d2ded5972959aef5b62e265683b9bcb433 (дата звернення: 20.05.2023).
15. Intel Core i7-6700K Processor: URL: https://www.intel.com/content/www/us/en/products/sku/88195/intel-core-i76700k-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата звернення: 20.05.2023).
16. Демонстраційна програма ігрового рушія “Humus GameEngine II”. Humus: URL: https://www.humus.name/index.php?page=3D&ID=63 (дата звернення: 20.05.2023).

# ДОДАТОК А

**До дипломного проєкту**

на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

Копії графічних матеріалів

**Обчислювальний шейдер. Схема алгоритму.**
ІАЛЦ.045200.005 Д1

**Пермутація та компіляція шейдерного коду. Схема алгоритму.**
ІАЛЦ.045200.006 Д2

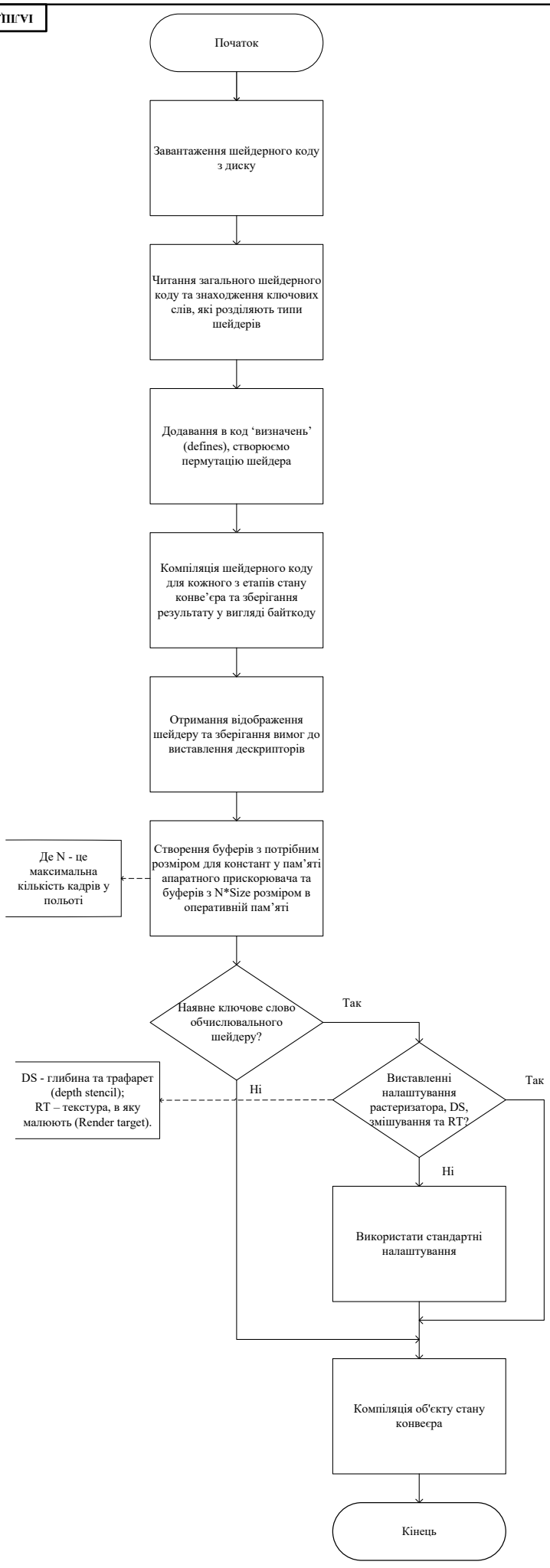
**Керування виконанням списків команд. Схема алгоримту.**
ІАЛЦ.045200.007 Д3

**Виставлення необхідних ресурсів в регістри апаратного прискорювачу. Схема алгоритму.**
ІАЛЦ.0045200.008 Д4

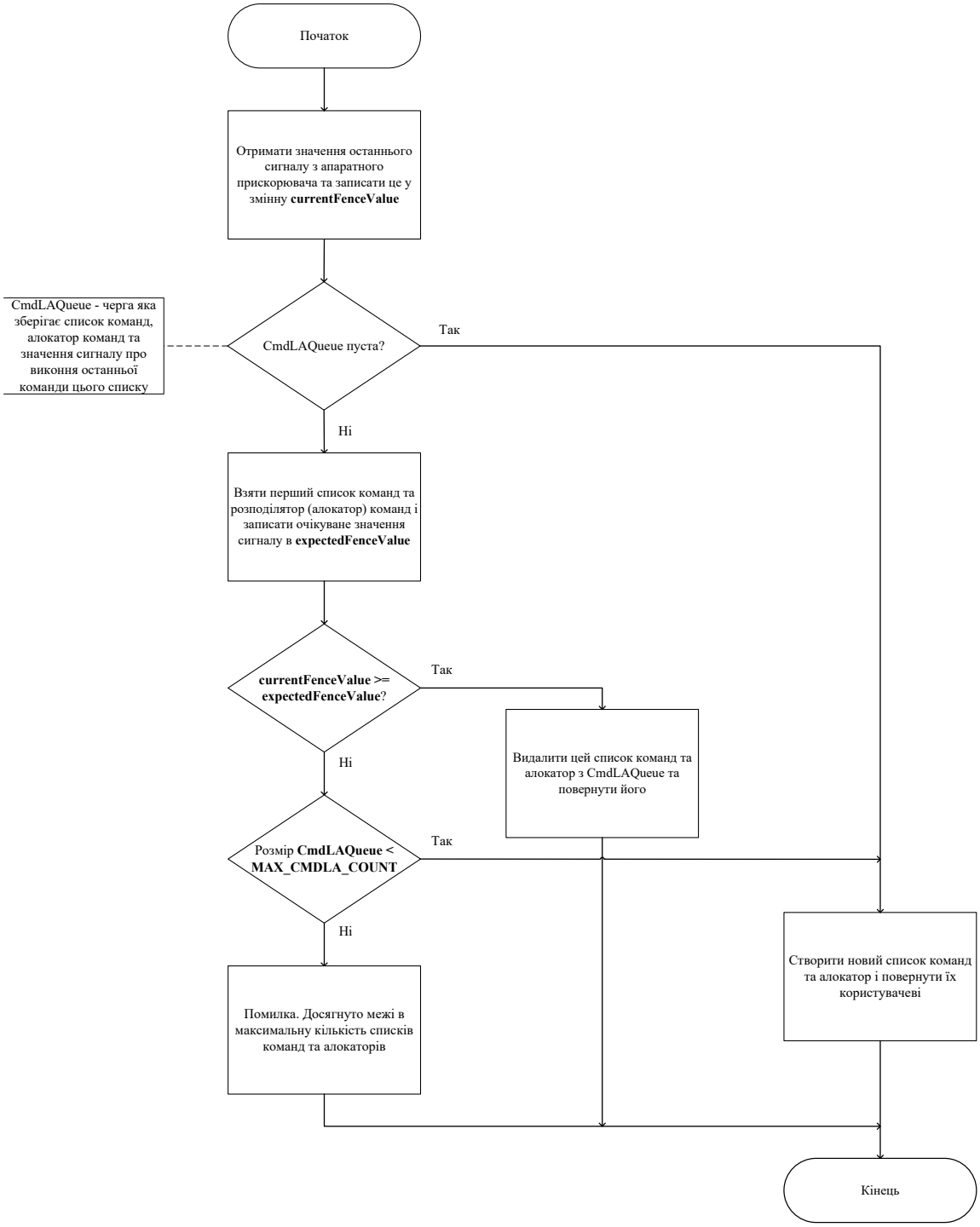
Аркушів 4

Київ – 2023

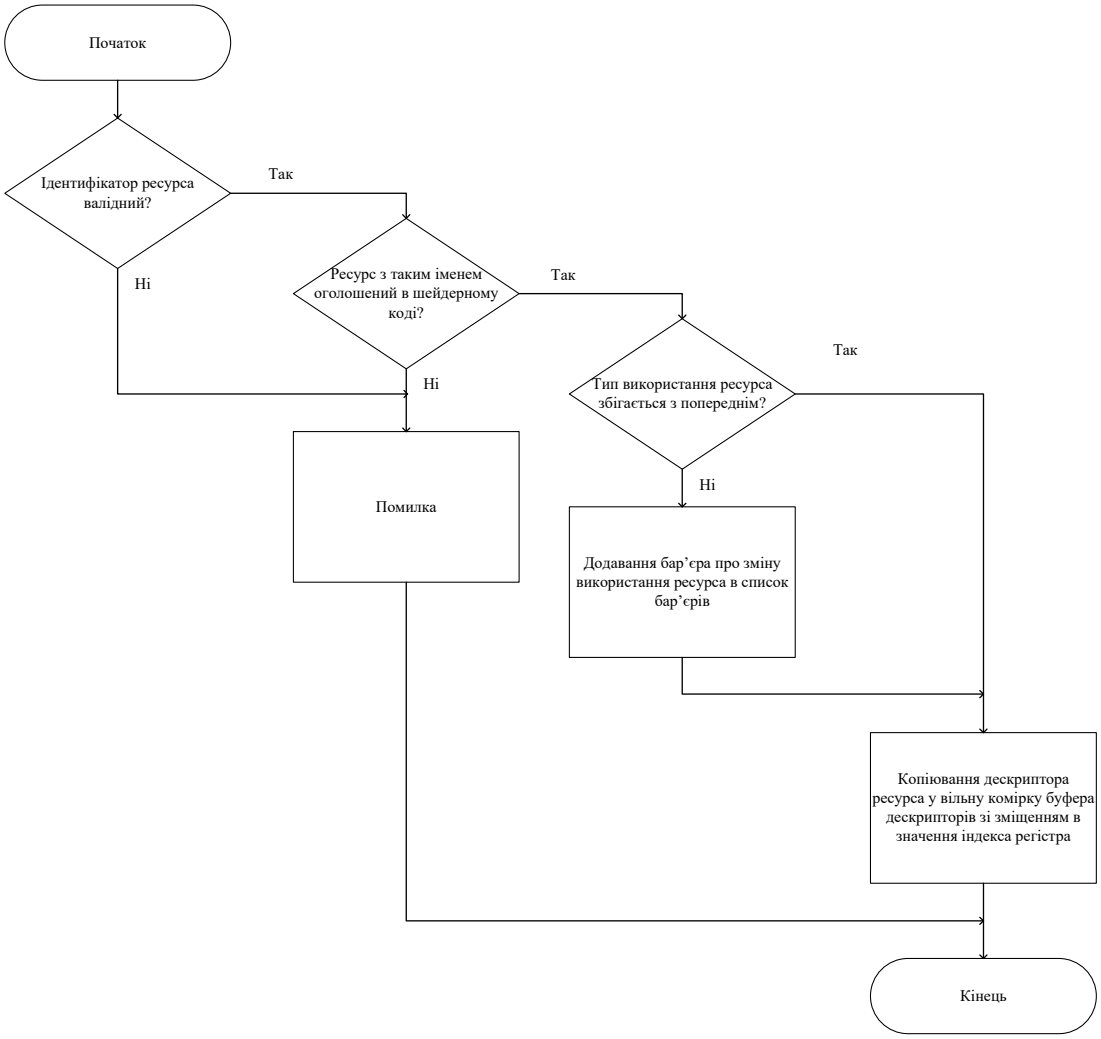
| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | ІАЛЦ.045200.005 Д1 | | |
| | № документу | Підп. | Дата | Обчислювальний шейдер. Схема алгоритму | Літера | Маса | Масштаб |
| Розроб. | Дзямулин Д.В. | | | | | | |
| Перевір. | Романкевич В.О. | | | | Аркуш 1 | Аркушів 1 |
| Н. контр. | Клятченко Я.М. | | | НТУУ «КПІ ім. Ігоря Сікорського», ФІМ, КВ-91 | | |
| Затв. | Романкевич В.О. | | | | | |




| | | | | | |
|---|---|---|---|---|---|
| | | | | **ІАЛЦ.045200.006 Д2** | |
| | № документу | Підп. | Дата | Пермутація та компіляція шейдерного коду. Схема алгоритму | Літера / Маса / Масштаб |
| Розроб. | Дзямулин Д.В. | | | | |
| Перевір. | Романкевич В.О. | | | | Аркуш 1 / Аркушів 1 |
| Н. контр. | Клятченко Я.М. | | | НТУУ «КПІ ім. Ігоря Сікорського», ФІМ, КВ-91 | |
| Затв. | Романкевич В.О. | | | | |




| | | | | | |
|---|---|---|---|---|---|
| | | | | **ІАЛЦ.045200.007 Д3** | |
| | № документу | Підп. | Дата | Керування виконанням списків команд. Схема алгоритму | Літера / Маса / Масштаб |
| Розроб. | Дзямулин Д.В. | | | | Аркуш 1 / Аркушів 1 |
| Перевір. | Романкевич В.О. | | | | |
| Н. контр. | Клятченко Я.М. | | | | НТУУ «КПІ ім. Ігоря Сікорського», ФІМ, КВ-91 |
| Затв. | Романкевич В.О. | | | | |




| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | ІАЛЦ.045200.008 Д4 | | |
| | № документу | Підп. | Дата | Виставлення необхідних ресурсів в регістри апаратного прискорювачу. Схема алгоритму | Літера | Маса | Масштаб |
| Розроб. | Дзямулин Д.В. | | | | | | |
| Перевір. | Романкевич В.О. | | | | Аркуш 1 | Аркушів 1 |
| Н. контр. | Клятченко Я.М. | | | | НТУУ «КПІ ім. Ігоря Сікорського», ФІМ, КВ-91 | |
| Затв. | Романкевич В.О. | | | | | |

# ДОДАТОК Б

**До дипломного проєкту**

на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

## Фрагменти програмного коду

Аркушів 44

Київ – 2023

# ДОДАТОК В

**До дипломного проєкту**

на тему: «Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі»

**Презентація бакалаврського проєкту**

Аркушів 15

Київ – 2023

# Алгоритм та програма управління апаратним прискорювачем тривимірної графіки у реальному часі

Виконав: студент IV курсу, групи КВ-91,
Дзямулич Даніель Васильович
Керівник: к.т.н, асист.каф.СПСКС Морозов К. В.

# Постанова задачі

- Метою даного проєкту є створення алгоритму та простого програмного інтерфейсу для управління апаратним прискорювачем тривимірної графіки у реальному часі використовуючи низькорівневий прикладний програмний інтерфейс Direct3D 12

# Що таке апаратний прискорювач тривимірної графіки?

- Апаратний прискорювач тривимірної графіки, графічний процесор або відеокарта – це спеціалізований пристрій, призначений для обчислення великої кількості математичних операцій паралельно, адже має велику кількість ядер. Зазвичай використовується для 3D візуалізації, машинного навчання, штучного інтелекту, створення фото-реалістичних відеоігор, відео-ефектів, прискоренні кодування та декодування відео та багато іншого.

# Аналіз існуючих рішень. Unreal Engine


- Перевагами є:
  - велика кількість інструментів для створення фото-реалістичної графіки;
  - кросплатформеність;
  - велика база користувачів;
  - внутрішній ринок ресурсів;
  - велика кількість розширень та додатків які дозволяють розширити функціонал Unreal Engine;
  - візуальний редактор;
  - підтримка сучасних низькорівневих графічних API.
- Недоліками є:
  - складність програмного середовища;
  - продуктивність;
  - вартість;
  - складність розробки мультиплатформенного продукту.

# Аналіз існуючих рішень. Godot

- Перевагами Godot є:
  - відкритий код;
  - мультиплатформена підтримка;
  - візуальний редактор;
  - скриптовий рушій;
  - активна спільнота;
  - розширення та плагіни.
- Недоліками є:
  - обмежена підтримка 3D-графіки. Godot позначається більше як програмний комплекс для розробки 2D-графіки, ніж для 3D;
  - використання OpenGL ES в якості графічної API. Це обмежує використання останніх технологій створених розробниками апаратних прискорювачів тривимірної графіки;
  - немає підтримки ігрових приставках.

# Аналіз існуючих рішень. Висновок

Більшість із доступних на ринку програмних середовищ потребують сплати за користування продуктом та перевантажені інструментами, які можуть не використовуватись в кінцевому продукті, що може погано позначитись на продуктивності продукту. Тому зі створенням власного програмного продукту, зазвичай створюють власне програмне середовище, яке буде мати тільки ті функції які необхідні продукту.

Використання готових алгоритмів для управління апаратним прискорювачем тривимірної графіки, може значно спростити розробку програмного середовища.



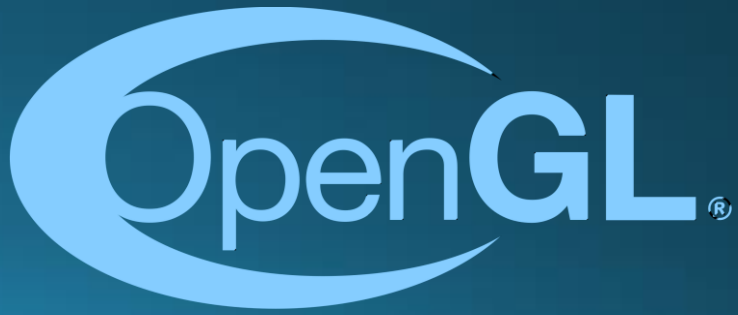
# Вибір графічної API. OpenGL

Перевагами є:

- крос-платформеність – можливість запускати програму, написану з використанням OpenGL на різних операційних системах та платформах таких як Windows, macOS, Linux та інші;
- відкритий програмний код. Полегшує розуміння документації та дозволяє написання власних функцій та розширень.

Недоліками є:

- відсутність єдиного API;
- відсутність підтримки останніх технологій, які пропонують сучасні апаратні прискорювачі;
- обмежене низькорівневе керування відеокартою;
- функціональний стиль програмування.



# Вибів графічної API. DirectX 12

Перевагами є:

- потенційно висока продуктивність на багатоядерних процесорах;
- підтримка новітніх технологій;
- об’єктно орієнтований стиль API;
- нативна підтримка в Windows.

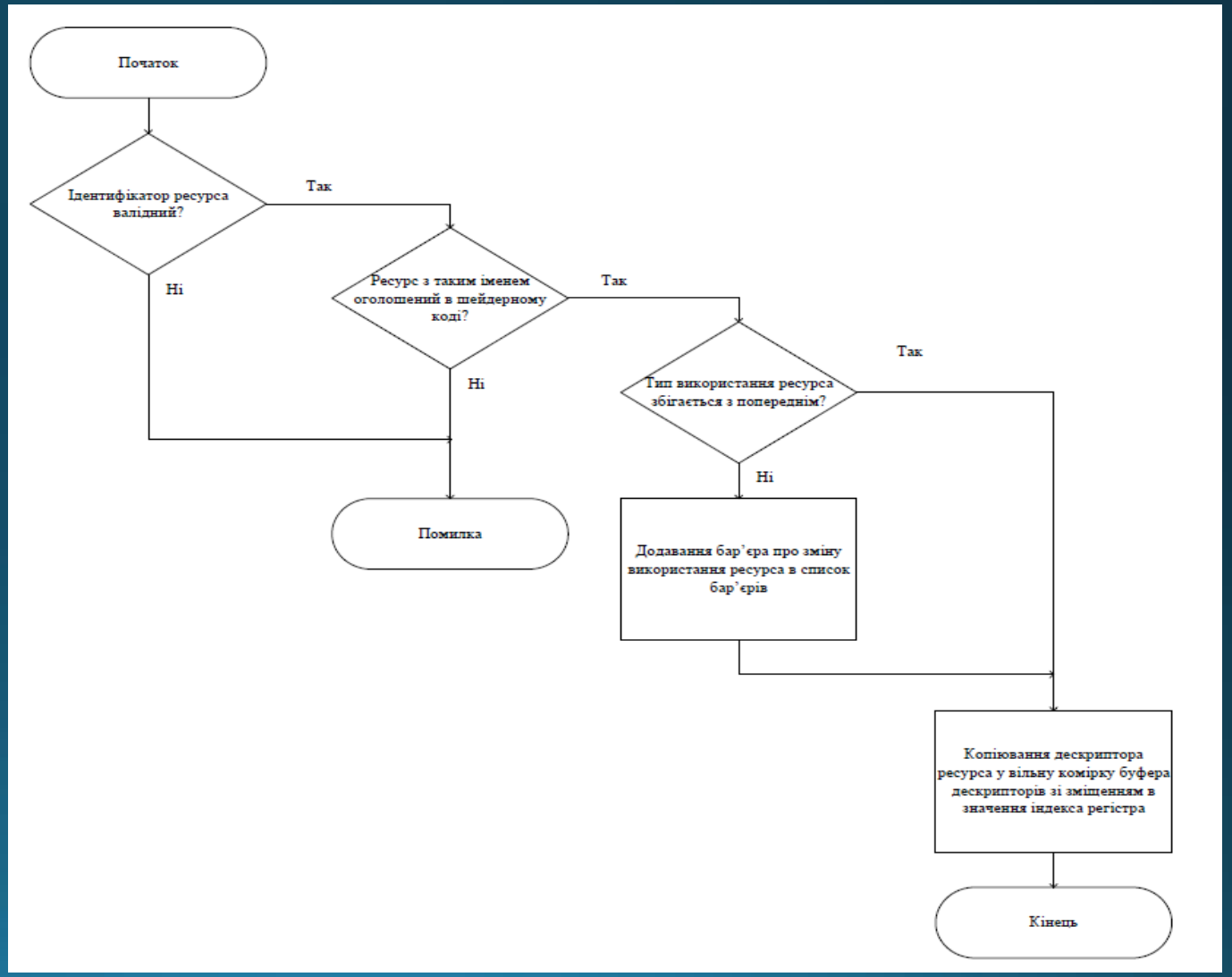
Недоліками є:

- складність розробки;
- складність вивчення;
- менша кількість підтримуваних платформ на відміну від OpenGL;
- немає офіційної підтримки версій Windows 8 та старіше.

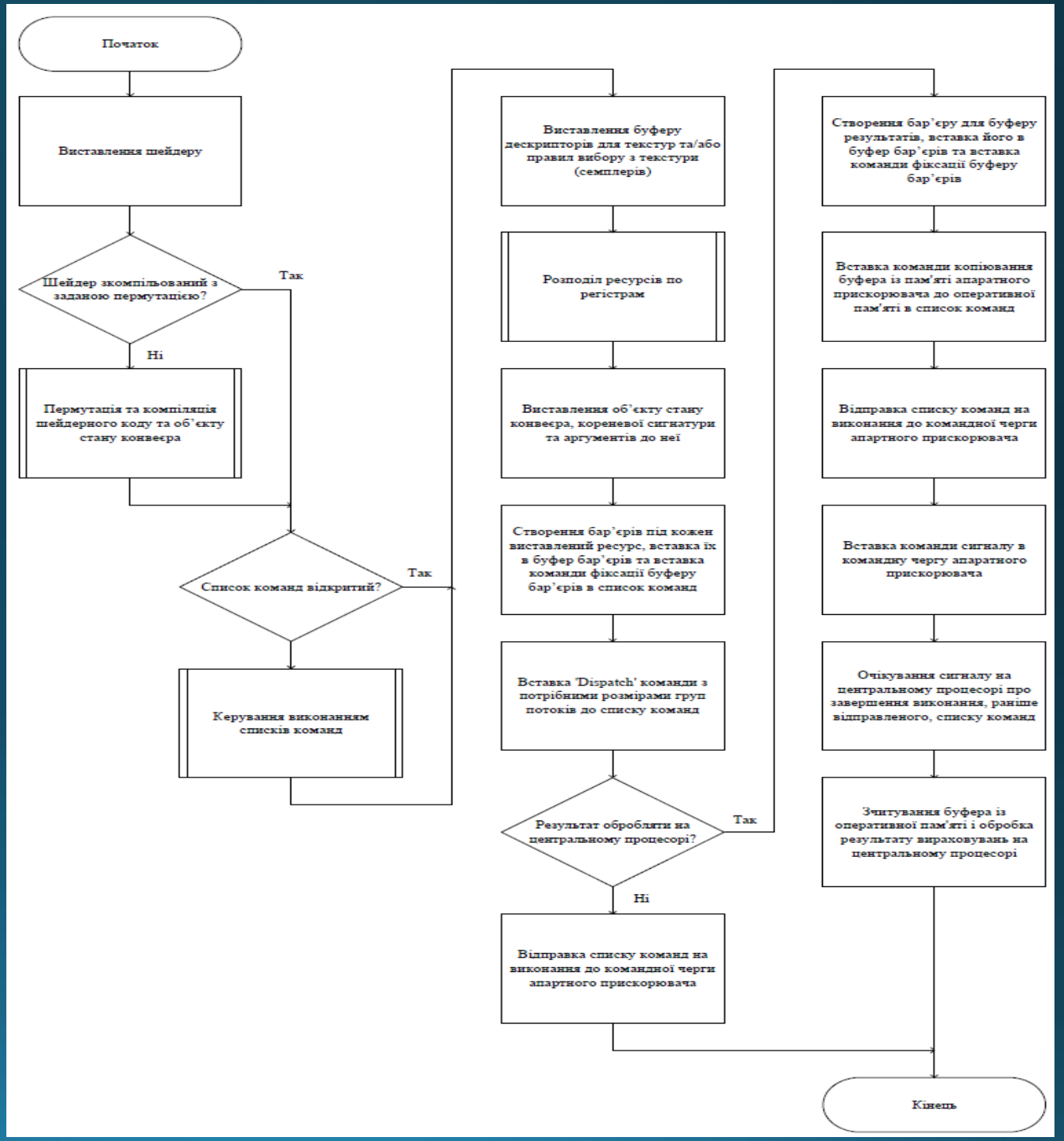
# Спрощений алгоритм для встановлення динамічних ресурсів для шейдеру

Схема кореневого опису для шейдеру малювання складається з:

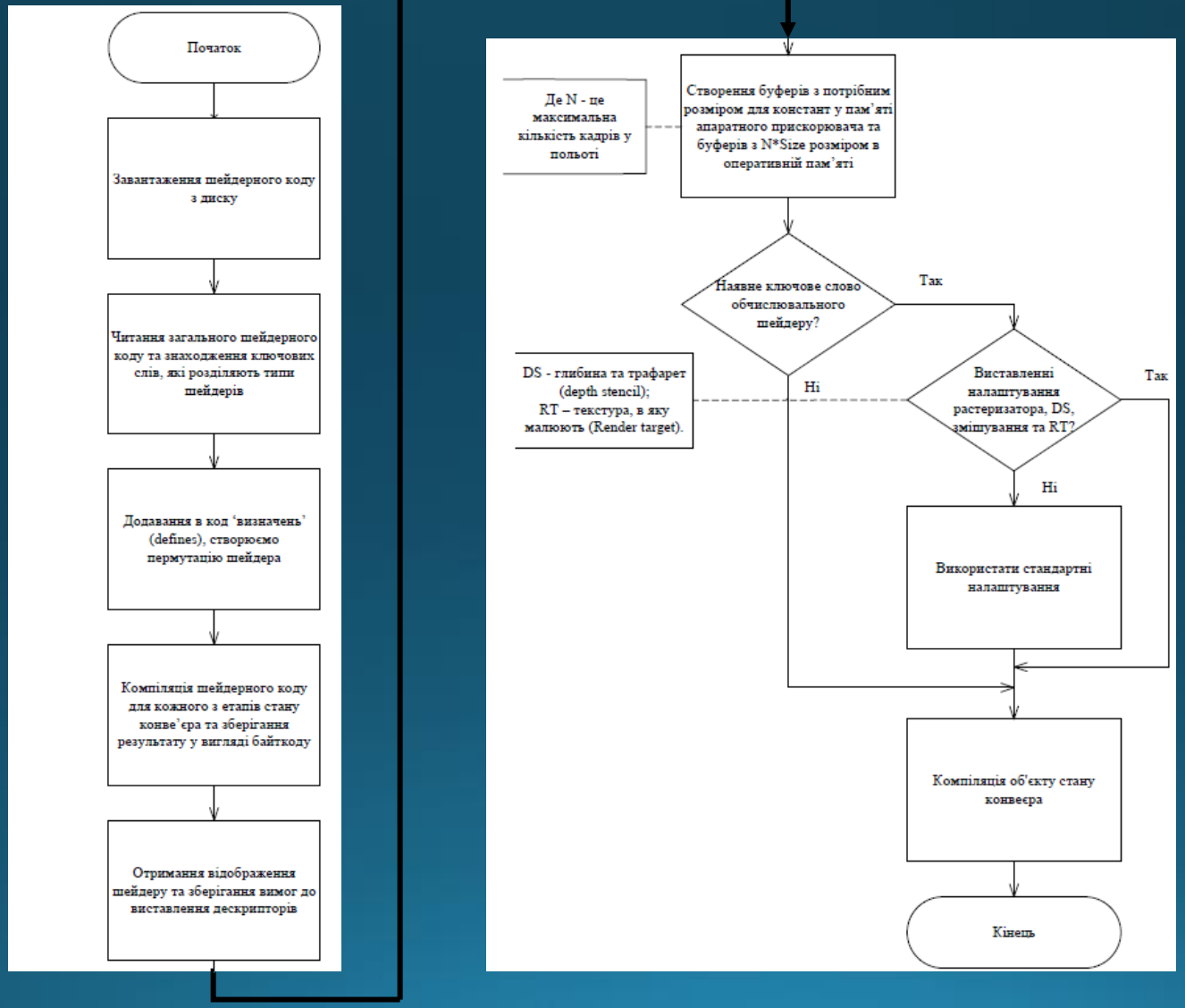
- **N** кореневих дескрипторів для константних буферів у **вершинному** шейдері, де **N** – кількість буферів які очікує шейдер;
- **N** кореневих дескрипторів для константних буферів у **піксельному** шейдері, де **N** – кількість буферів які очікує шейдер;
- Якщо піксельний шейдер очікує дескриптори типу SRV, то додавання таблиці дескрипторів із діапазоном під кількість очікуваних шейдером дескрипторів, цього типу;
- Якщо піксельний шейдер очікує Sampler View (правила для читання текстури), то додавання таблиці дескрипторів із діапазоном під кількість очікуваних шейдером Sampler Views;



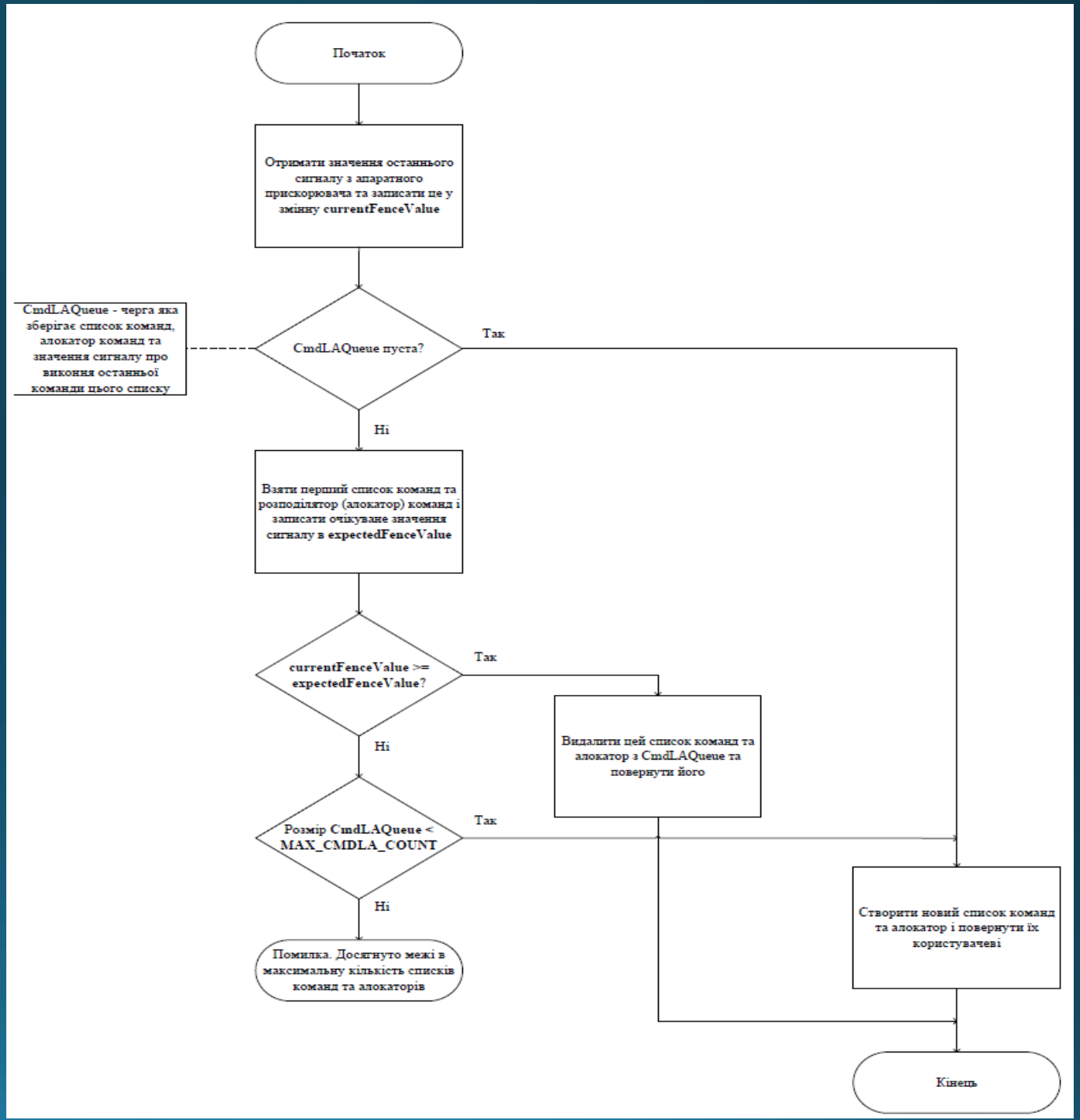
# Схема алгоритму для виставлення обчислювального шейдеру.



# Спрощена схема алгоритму для пермутація та компіляція шейдерного коду.



# Спрощена схема алгоритму керування виконанням списків команд апаратним прискорювачем.



# Напрямки подальшого розвитку алгоритмів

- Не копіювати дескриптори в буфер дескрипторів, якщо порядок в кореневому описі не змінився;
- перевикористання кореневого опису між різними шейдерами;
- додати підтримку для інших програмованих шейдерних етапів;
- використання асинхронної черги вираховувань на апаратному прискорювачеві;
- додати можливість заповнювання списків команд асинхронно;
- додати можливість змінювати вихідний формат малювання, глибини й трафарету в об’єкті стану конвеєра;
- додати можливість виставляти власні значення для швидкого очищення ресурсу, при його створенні.

# Аналіз отриманих результатів та порівняння із використання високорівневої графічної бібліотеки


Результат генерації тривимірної сцени демонстраційної програми “Humus GameEngine II” використовуючи високорівневу графічну бібліотеку OpenGL


Результат генерації тривимірної сцени демонстраційної програми “Humus GameEngine II” використовуючи низькорівневу графічну бібліотеку DirectX12

Доповідь завершено.
Дякую за увагу!