Міністерство освіти і науки України Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського» Факультет інформатики та обчислювальної техніки Кафедра обчислювальної техніки

Лабораторна робота №4

з дисципліни

«Комп’ютерна графіка та мультимедіа»

Виконав:

студент групи ІП-05

Гапій Денис Едуардович

номер залікової : 0504

Перевірив:

Родіонов П. Ю.

Київ 2022

Тема: «Створення та редагування шейдерів у середовищі API Vulkan»

Мета: навчитися створювати та редагувати шейдери у середовищі API Vulkan.

Контрольні запитання:

1. Поняття та відмінності вертексного шейдера та вертексного буфера.

   Dершинний буфер дозволяє графічному процесору зчитувати дані з буфера(алокованої частинки пам’яті) та надсилати їх до нашого вершинного шейдера. Щоб прочитати буфер вершин із шейдера, вам потрібно встановити стан введення вершин у графічному конвеєрі. Це дозволить Vulkan знати, як інтерпретувати заданий буфер у дані вершини. Коли це буде налаштовано, ми зможемо автоматично отримувати інформацію про вершини у вершинному шейдері, як-от кольори або позиції вершин.

2. Етапи опису вершин.

   - Кожна вершина, отримана з масивів вершин (буфера вершин), обробляється Vertex Shader . Кожна вершина в потоці по черзі обробляється у вихідну вершину.
   - Необов'язкові примітивні етапи мозаїки .
   - Додаткова примітивна обробка Geometry Shader . На виході виходить послідовність примітивів.

   Vertex Shader — це один з етапів програмованого шейдера, який можна спостерігати у графічному конвеєрі.

   Детальніше у Прикладі №1.

3. Створення та редагування буфера вершин.

   Створення буфера вимагає від нас заповнення структури *VkBufferCreateInfo*. Ми визначаємо розмір буфера в байтах, для яких цілей будуть використовуватися дані в буфері і тд. За бажання буфери також можуть

належати певній сім’ї черги або використовуватися одночасно декількома. Буфер використовуватиметься лише з черги графіки, тому ми можемо дотримуватися ексклюзивного доступу. Буфер має бути доступним для використання в командах візуалізації до кінця програми (через дескриптор класу), і він не залежить від ланцюжка обміну, тому ми очистимо його при виклику *cleanup* функції. Далі ми виконуємо пошук підходящої для нашого проекту пам’яті та виділяємо (розподіляємо) пам’ять для подальшої можливості взаємодіяти з нею (запис / читання даних). Не забуваємо, що пам’ять, яка прив’язана до об’єкта буфера, повинна бути звільнена, коли буфер більше не використовується. Наступним кроком варто скопіювати дані вершини в буфер. Це робиться шляхом відображення буферної пам’яті в доступну пам’ять ЦП. І наостанок: прив’язуємо буфер вершин під час операцій візуалізації.

Детальніше у Прикладі №2.

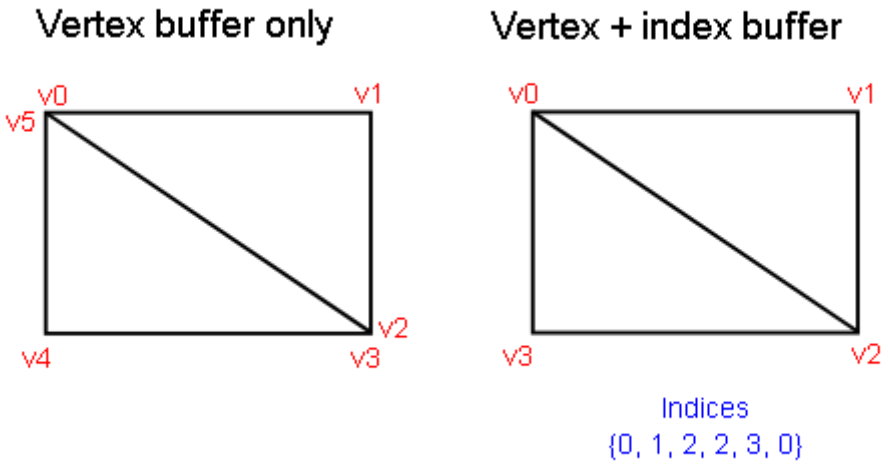
4. Налаштування вершинного буфера.

   Буфер вершин, який ми зараз маємо, працює правильно, але тип пам’яті, який дозволяє отримати доступ до нього з ЦП, може бути не найоптимальнішим типом пам’яті для самої відеокарти для читання. Тому ми збираємося створити два буфери вершин. Один проміжний буфер у доступній пам’яті ЦП для завантаження даних із масиву вершин, а останній буфер вершин у локальній пам’яті пристрою. Потім ми використаємо команду копіювання буфера, щоб перемістити дані з проміжного буфера до фактичного буфера вершин.

   Детальніше у Прикладі №3

5. Поняття та призначення індексного буферу.
   Індексний буфер — це, по суті, масив покажчиків на вершинний буфер. Це дозволяє змінювати порядок даних вершин і повторно використовувати існуючі дані для кількох вершин. Ілюстрація нижчк демонструє, як виглядав би буфер індексу для прямокутника, якщо у нас є буфер вершин, що містить кожну з чотирьох унікальних вершин. Перші три індекси визначають верхній

правий трикутник, а останні три індекси визначають вершини нижнього лівого трикутника.



Якщо в двох словах, то можна підсумувати, що індексний буфер - вдало економить пам’ять.

Детальніше у Прикладі №4.

Приклади коду:

Приклад №1:

```
#include <glm/glm.hpp>

#include <array>

struct Vertex { // структура з векторами позиції та кольору

    glm::vec2 pos;

    glm::vec3 color;

     //Зв'язування вершин

    static VkVertexInputBindingDescription getBindingDescription() {

        VkVertexInputBindingDescription bindingDescription{};

        bindingDescription.binding = 0;//індекс прив'язки в масиві
```

```
        bindingDescription.stride = sizeof(Vertex); //кількість байтів від одного запису до наступного,
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX; // Перехід до наступного запису даних після кожної вершини
        return bindingDescription;
    }
     // Опис атрибутів
    static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions{};

        attributeDescriptions[0].binding = 0; //повідомляє з якого зв’язку надходять дані для кожної вершини
        attributeDescriptions[0].location = 0; // осилається на директиву введення у вершинному шейдері.
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT; // тип даних для атрибута
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        attributeDescriptions[1].binding = 0;
        attributeDescriptions[1].location = 1;
        attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[1].offset = offsetof(Vertex, color);

        return attributeDescriptions;
    }
};
```

Приклад №2:

```
void initVulkan() {
    …
    createCommandPool();
    createVertexBuffer();
    …
}
VkBuffer vertexBuffer;

void createVertexBuffer() {
        VkBufferCreateInfo bufferInfo{}; // описано в к.п.3
        bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        bufferInfo.size = sizeof(vertices[0]) * vertices.size();
        bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
        bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

        if (vkCreateBuffer(device, &bufferInfo, nullptr,
&vertexBuffer) != VK_SUCCESS) {
            throw std::runtime_error("failed to create vertex
buffer!");
        }
        VkMemoryRequirements memRequirements;
        vkGetBufferMemoryRequirements(device, vertexBuffer,
&memRequirements);
        VkMemoryAllocateInfo allocInfo{}; // Розподіл пам’яті
        allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        allocInfo.allocationSize = memRequirements.size;
```

```
        allocInfo.memoryTypeIndex =
findMemoryType(memRequirements.memoryTypeBits,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

        if (vkAllocateMemory(device, &allocInfo, nullptr,
&vertexBufferMemory) != VK_SUCCESS) {

            throw std::runtime_error("failed to allocate vertex buffer
memory!");

        }

        vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory,
0);

        void* data;

        vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0,
&data); //  доступ до області зазначеного ресурсу пам’яті

            memcpy(data, vertices.data(), (size_t)
bufferInfo.size);//копіювання даних

        vkUnmapMemory(device, vertexBufferMemory);

    }

    uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags
properties) {

        VkPhysicalDeviceMemoryProperties memProperties;

        vkGetPhysicalDeviceMemoryProperties(physicalDevice,
&memProperties);

        for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
// перевірка підтримки властивості маніпуляції над пам’яттю:

            if ((typeFilter & (1 << i)) &&
(memProperties.memoryTypes[i].propertyFlags & properties) ==
properties) {

                return i;
```

```
            }
        }

        throw std::runtime_error("failed to find suitable memory type!");
    }

 void recordCommandBuffer(VkCommandBuffer commandBuffer, uint32_t imageIndex) {
// . . .
//
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, offsets);
// прив’язка вершинних буферів до прив’язок
vkCmdDraw(commandBuffer, static_cast<uint32_t>(vertices.size()), 1, 0, 0); // передати кількість вершин у буфері
}
```

Приклад №3:

```
void createVertexBuffer() { // Використання проміжного буфера
        VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
        VkBuffer stagingBuffer; // відображення та копіювання даних вершин
```

```
        VkDeviceMemory stagingBufferMemory;

        createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
stagingBufferMemory); // Буфер можна використовувати як джерело в
операції передачі пам’яті. та буфер можна використовувати як місце
призначення в операції передачі пам’яті.

        void* data;

        vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0,
&data);

            memcpy(data, vertices.data(), (size_t) bufferSize);

        vkUnmapMemory(device, stagingBufferMemory);

        createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT |
VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer,
vertexBufferMemory);

        copyBuffer(stagingBuffer, vertexBuffer, bufferSize);

        vkDestroyBuffer(device, stagingBuffer, nullptr);

        vkFreeMemory(device, stagingBufferMemory, nullptr);

    }


     void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage,
VkMemoryPropertyFlags properties, VkBuffer& buffer, VkDeviceMemory&
bufferMemory) {

        VkBufferCreateInfo bufferInfo{};

        bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;

        bufferInfo.size = size;

        bufferInfo.usage = usage;

        bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

```
        if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) !=
VK_SUCCESS) {
            throw std::runtime_error("failed to create buffer!");
        }
        VkMemoryRequirements memRequirements;
        vkGetBufferMemoryRequirements(device, buffer,
&memRequirements);
        VkMemoryAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        allocInfo.allocationSize = memRequirements.size;
        allocInfo.memoryTypeIndex =
findMemoryType(memRequirements.memoryTypeBits, properties);

        if (vkAllocateMemory(device, &allocInfo, nullptr,
&bufferMemory) != VK_SUCCESS) {
            throw std::runtime_error("failed to allocate buffer
memory!");
        }
        vkBindBufferMemory(device, buffer, bufferMemory, 0);
    }

    void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer,
VkDeviceSize size) {
        VkCommandBufferAllocateInfo allocInfo{}; //  тимчасовий буфер команд
        allocInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        allocInfo.commandPool = commandPool;
```

```
        allocInfo.commandBufferCount = 1;
        VkCommandBuffer commandBuffer;
        vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
        VkCommandBufferBeginInfo beginInfo{}; //  записування буфер
команд
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
        beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
        vkBeginCommandBuffer(commandBuffer, &beginInfo);
            VkBufferCopy copyRegion{};
            copyRegion.size = size;
            vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1,
&copyRegion); // Вміст буферів передається
        vkEndCommandBuffer(commandBuffer);
        VkSubmitInfo submitInfo{};
        submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
        submitInfo.commandBufferCount = 1;
        submitInfo.pCommandBuffers = &commandBuffer;
        vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
        vkQueueWaitIdle(graphicsQueue); //очікування поки черга
передачі стане неактивною
        vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
    }
```

Приклад №4:

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
VkBuffer indexBuffer;
```

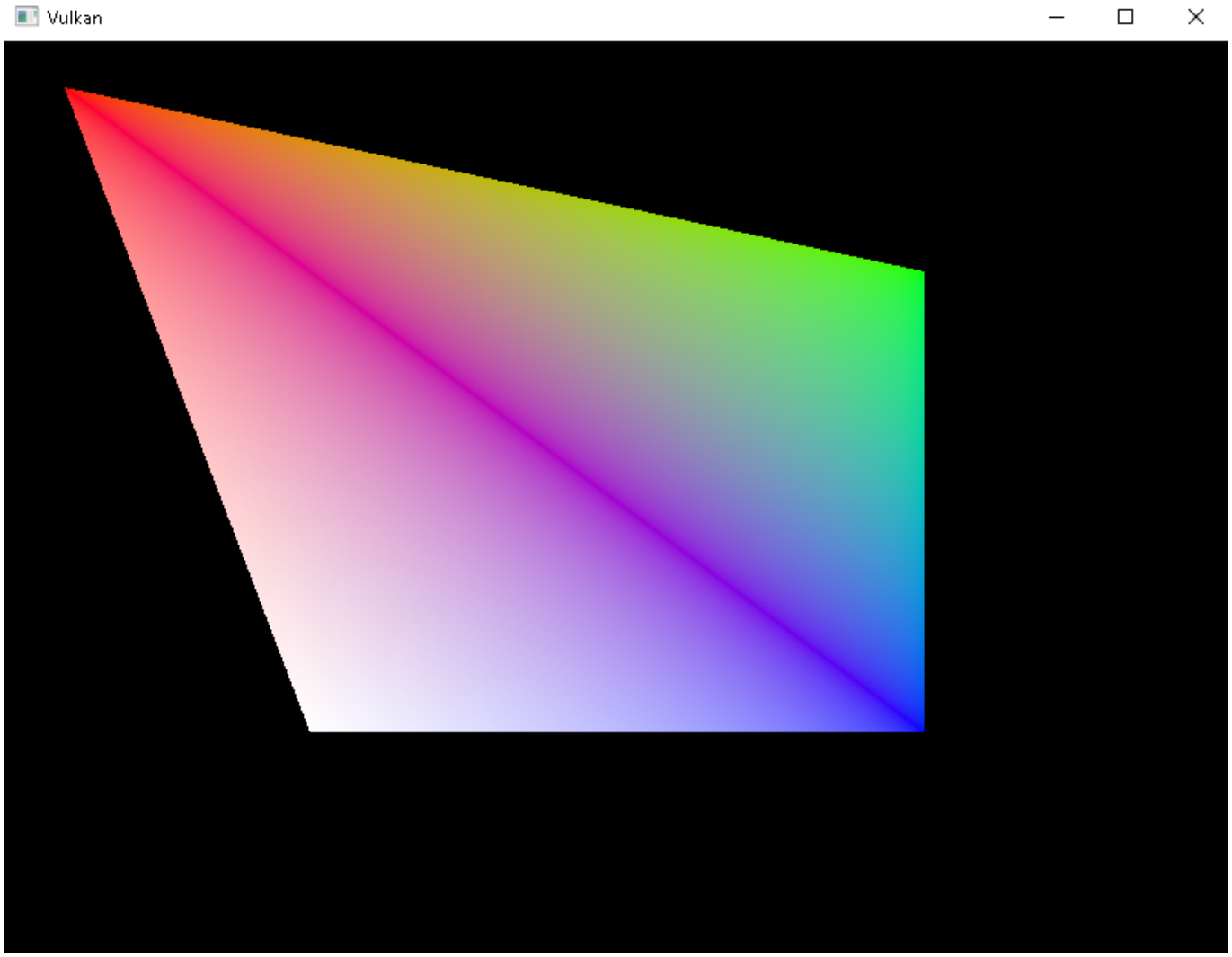
```
VkDeviceMemory indexBufferMemory;

void createIndexBuffer() { // копія create staging buffer, але з відмінною у bufferSize + indexBuffer. А загалом ми так само: створюємо проміжний буфер, щоб скопіювати вміст у indices, а потім скопіювати його в кінцевий локальний індексний буфер пристрою.

    VkDeviceSize bufferSize = sizeof(indices[0]) * indices.size();

    VkBuffer stagingBuffer;

    VkDeviceMemory stagingBufferMemory;

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

    void* data;

    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);

    memcpy(data, indices.data(), (size_t) bufferSize);

    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, indexBuffer, indexBufferMemory);

    copyBuffer(stagingBuffer, indexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);

    vkFreeMemory(device, stagingBufferMemory, nullptr);

}
```

Приклад нового вершинного шейдеру:

```
#version 450
```

```
layout(location = 0) in vec2 inPosition;

layout(location = 1) in vec3 inColor;



layout(location = 0) out vec3 fragColor;



void main() {

    gl_Position = vec4(inPosition, 0.0, 1.0);

    fragColor = inColor;

}
```

Результат запуску програми:



Вікно створене за допомогою GLFW відображає фігуру (стрілу) складену з двох інших примітивних фігур, а саме трикутників. Розфарбоване градієнтом відносно кольорів, що прикріплені до вершин у шейдері.

Висновок:

Я успішно навчився створювати та редагувати самі примітивні шейдери у середовищі API Vulkan, де головна мета написання самого шейдеру залежить від вхідних атрибутів, та віддає оброблені вершини далі для графічного конвеєру, що ми написали у лабораторній роботі №3, та доповнили у цій, використовуючи буфер вершин.

Додатково згадавши правила аллокацій (виділення) пам’яті, було реалізовано структуру даних, по типу масиву, що дозволяє нам зберігати дані про вершини.

Використавши індексний буфер оптимізували роботу над шейдерами, давши змогу змінювати порядок даних вершин і повторно використовувати існуючі дані для кількох вершин, не ініціалізуючи під них зайві частинки пам’яті. Також використання проміжного буферу, дало нам змогу зберігати їх тимчасово у ньому, а потім передати до локального буферу пристрою, звільнивши проміжний після.