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 №6

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 працювати з текстурами у середовищі API Vulkan.

Контрольні запитання:

1. Етапи та особливості роботи з текстурами.

Основні 4 кроки для можливості відображення текстури на нашій моделі:

- Створення об’єкту зображення, що підтримується пам’яттю пристрою
- Заповнення його пікселями з файлу зображення
- Створення вибірки(шаблону) зображення
- Додавання комбінованого дескриптора семплера(шаблону) зображень для вибірки кольорів із текстури

*Деякі кроки можуть розбиватися на додаткові.

Створення зображення не дуже відрізняється від створення буферів, в тому плані що ми заповнюємо виділену пам’ять пікселями з певним кольором, проте є відмінність у тому, що зображення можуть мати різні макети , які впливають на те, як організовано пікселі в пам’яті.

2. Поняття та використання перехідних бар’єрних масок (barrier masks).

Конвеєрні бар’єри в основному використовуються для синхронізації доступу до ресурсів, наприклад, щоб переконатися, що зображення було записане перед його читанням, але їх також можна використовувати для переходу між макетами.

Бар’єри можна додатково використовувати для передачі права власності на сім’ю черги під час використання VK_SHARING_MODE_EXCLUSIVE

Одним із найпоширеніших способів виконання переходів макета є використання бар’єру пам’яті зображення . Подібний конвеєрний бар’єр зазвичай використовується для синхронізації доступу до ресурсів, наприклад для забезпечення завершення запису в буфер перед читанням з нього, але його

також можна використовувати для переміщення макетів зображень і передачі права власності на сім'ю черги, якщо VK_SHARING_MODE_EXCLUSIVEвін використовується. Для цього існує еквівалентний бар'єр буферної пам'яті для буферів.
Детальніше у Прикладі нових функцій / методів..
3. Перегляд зображень та робота з вибірками.
Нам потрібно було створити таке зображення, подібне до того з ланцюга обміну та буфер кадрів отримують доступ до зображень через перегляди зображень, а не безпосередньо для зображення текстури.
Ми зберігаємо зображення текстури у VkImageView.
Доступ до текстур зазвичай здійснюється за допомогою семплерів, які застосовують фільтрацію та перетворення для обчислення кінцевого кольору, який отримується.
Ці фільтри корисні для вирішення таких проблем, як надмірна вибірка.
Розглянемо текстуру, яка відображається на геометрію з більшою кількістю фрагментів, ніж текселів.
Якщо ви просто взяли найближчий тексель для координати текстури в кожному фрагменті, то ви отримаєте результат, як на першому зображенні.
Недостатня вибірка є протилежною проблемою, коли у вас більше текселів, ніж фрагментів. Це призведе до артефактів під час вибірки високочастотних шаблонів, таких як текстура шахової дошки під гострим кутом.

4. Поняття та використання анізотропної фільтрації.
Анізотропна фільтрація — це метод покращення якості зображення текстур на поверхнях комп'ютерної графіки, які знаходяться під косими кутами огляду відносно камери, де проекція текстури (а не багатокутник чи інший примітив , на якому він відображається), виглядає не ортогональним.
Подібно до білінійної та трилінійної фільтрації , анізотропна фільтрація усуває ефект накладення, але вдосконалює ці інші методи, зменшуючи розмиття та зберігаючи деталі під екстремальними кутами огляду. Справжня анізотропна

фільтрація анізотропно досліджує текстуру на льоту на піксельній основі для будь-якої орієнтації анізотропії.
Головне при створенні семплу (шаблону) зображення вказати `anisotropyEnable` як `true`.
Якщо ми хочемо отримати максимальну якість, ми можемо просто використати значення безпосередньо:
samplerInfo.maxAnisotropy = properties.limits.maxSamplerAnisotropy;

5. Робота з координатами текстур.
Координати визначають, як зображення насправді відображається на геометрії.
Знову не забуваємо додати можливість використовувати координати текстури через атрибути.
Й не менш важливо змінити матрицю(вектор) нашого квадрату / плитки та положень (координат) вершин використовуючи координати від 0, 0 верхнього лівого кута до 1, 1 нижнього правого кута.
* але немає жодного обмеження у змінних, адже можна навіть ввести від'ємне значення для дивакуватого відображення.
Детальніше у Прикладі зміненого коду (частина класу Vertex, etc).

Приклади коду:

Приклади зміненого коду:

```
Зміна Vertex структури, щоб включити vec2 координати текстури.Додали
VkVertexInputAttributeDescription, щоб ми могли використовувати
координати текстури доступу як вхідні дані у вершинному шейдері. Це
необхідно, щоб мати можливість передати їх у фрагментний шейдер для
інтерполяції по поверхні квадрата. Це фактичні координати для кожної
вершини. Координати визначають, як зображення насправді відображається
на геометрії.Заповнили матрицю текстурою, використовуючи координати
від 0, 0 до 1, 1
```

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription{};
        bindingDescription.binding = 0;
        bindingDescription.stride = sizeof(Vertex);
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions{};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
        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);

        attributeDescriptions[2].binding = 0;
        attributeDescriptions[2].location = 2;
        attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[2].offset = offsetof(Vertex, texCoord);

        return attributeDescriptions;
    }
};

struct UniformBufferObject {
    alignas(16) glm::mat4 model;
    alignas(16) glm::mat4 view;
    alignas(16) glm::mat4 proj;
};

const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
};
```

Класи для зберігання зображення та відповідних залежностей:

```
VkImage textureImage;
VkDeviceMemory textureImageMemory;
VkImageView textureImageView;
VkSampler textureSampler;
```

Додаємо функції створення текстури зображення, його відображення та шаблон до функції ініціалізації застосунку:

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createDescriptorSetLayout();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffers();
    createDescriptorPool();
    createDescriptorSets();
    createCommandBuffers();
    createSyncObjects();
}
```

У функції очищення не забуваємо звільняти пам’ять зайняту текстурами / зображенням.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyRenderPass(device, renderPass, nullptr);

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }

    vkDestroyDescriptorPool(device, descriptorPool, nullptr);

    vkDestroySampler(device, textureSampler, nullptr);
    vkDestroyImageView(device, textureImageView, nullptr);

    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);

    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);
```

```
    void createLogicalDevice() {
        QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

        std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
        std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily.value(), indices.presentFamily.val
ue()};

        float queuePriority = 1.0f;
        for (uint32_t queueFamily : uniqueQueueFamilies) {
            VkDeviceQueueCreateInfo queueCreateInfo{};
            queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
            queueCreateInfo.queueFamilyIndex = queueFamily;
            queueCreateInfo.queueCount = 1;
            queueCreateInfo.pQueuePriorities = &queuePriority;
            queueCreateInfos.push_back(queueCreateInfo);
        }

        VkPhysicalDeviceFeatures deviceFeatures{};
        deviceFeatures.samplerAnisotropy = VK_TRUE;
```

Оптимізуємо функцію створення представлення зображень через ланцюжок

```
    for (size_t i = 0; i < swapChainImages.size(); i++) {
        VkImageViewCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        createInfo.image = swapChainImages[i];
        createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
        createInfo.format = swapChainImageFormat;
        createInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        createInfo.subresourceRange.baseMipLevel = 0;
        createInfo.subresourceRange.levelCount = 1;
        createInfo.subresourceRange.baseArrayLayer = 0;
        createInfo.subresourceRange.layerCount = 1;

        if (vkCreateImageView(device, &createInfo, nullptr, &swapChainImageViews[i]) != VK_SUCCESS) {
            throw std::runtime_error("failed to create image views!");
        }
    }
}
```

```
504     for (size_t i = 0; i < swapChainImages.size(); i++) {
505         swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat);
506     }
507 }
```

Додаємо дескриптор вибірки комбінованого зображення. Ми просто помістимо його в прив’язку після уніфікованого буфера. Ми також повинні створити більший пул дескрипторів, щоб звільнити місце для виділення комбінованого зразка зображень, додавши ще один VkPoolSize. Дескриптори мають бути оновлені цією інформацією про зображення, як і буфер. Цього разу ми використовуємо масив pImageInfo замість pBufferInfo

```
void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding{};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorCount = 1;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.pImmutableSamplers = nullptr;
    uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

    VkDescriptorSetLayoutCreateInfo layoutInfo{};
    layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    layoutInfo.bindingCount = 1;
    layoutInfo.pBindings = &uboLayoutBinding;

    if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK_SUCCESS)
{
        throw std::runtime_error("failed to create descriptor set layout!");
    }
}
```

```
551 void createDescriptorSetLayout() {
552     VkDescriptorSetLayoutBinding uboLayoutBinding{};
553     uboLayoutBinding.binding = 0;
554     uboLayoutBinding.descriptorCount = 1;
555     uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
556     uboLayoutBinding.pImmutableSamplers = nullptr;
557     uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
558
559     VkDescriptorSetLayoutBinding samplerLayoutBinding{};
560     samplerLayoutBinding.binding = 1;
561     samplerLayoutBinding.descriptorCount = 1;
562     samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
563     samplerLayoutBinding.pImmutableSamplers = nullptr;
564     samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
565
566     std::array<VkDescriptorSetLayoutBinding, 2> bindings = {uboLayoutBinding, samplerLayoutBinding};
567     VkDescriptorSetLayoutCreateInfo layoutInfo{};
568     layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
569     layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
570     layoutInfo.pBindings = bindings.data();
571
572     if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK_SUCCESS)
{
573         throw std::runtime_error("failed to create descriptor set layout!");
574     }
575 }
576
```

```
    void createDescriptorPool() {
        VkDescriptorPoolSize poolSize{};
        poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        poolSize.descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);



        VkDescriptorPoolCreateInfo poolInfo{};
        poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
        poolInfo.poolSizeCount = 1;
        poolInfo.pPoolSizes = &poolSize;
        poolInfo.maxSets = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);

        if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
            throw std::runtime_error("failed to create descriptor pool!");
        }
    }
```

```
    void createDescriptorPool() {
        std::array<VkDescriptorPoolSize, 2> poolSizes{};
        poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        poolSizes[0].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
        poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        poolSizes[1].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);

        VkDescriptorPoolCreateInfo poolInfo{};
        poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
        poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
        poolInfo.pPoolSizes = poolSizes.data();
        poolInfo.maxSets = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);

        if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
            throw std::runtime_error("failed to create descriptor pool!");
        }
    }
```

```
void createDescriptorSets() {
    std::vector<VkDescriptorSetLayout> layouts(MAX_FRAMES_IN_FLIGHT, descriptorSetLayout);
    VkDescriptorSetAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    allocInfo.descriptorPool = descriptorPool;
    allocInfo.descriptorSetCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
    allocInfo.pSetLayouts = layouts.data();

    descriptorSets.resize(MAX_FRAMES_IN_FLIGHT);
    if (vkAllocateDescriptorSets(device, &allocInfo, descriptorSets.data()) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate descriptor sets!");
    }

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        VkDescriptorBufferInfo bufferInfo{};
        bufferInfo.buffer = uniformBuffers[i];
        bufferInfo.offset = 0;
        bufferInfo.range = sizeof(UniformBufferObject);

        VkWriteDescriptorSet descriptorWrite{};
        descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
        descriptorWrite.dstSet = descriptorSets[i];
        descriptorWrite.dstBinding = 0;
        descriptorWrite.dstArrayElement = 0;
        descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        descriptorWrite.descriptorCount = 1;
        descriptorWrite.pBufferInfo = &bufferInfo;

        vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
    }
}
```

```
    void createDescriptorSets() {
        std::vector<VkDescriptorSetLayout> layouts(MAX_FRAMES_IN_FLIGHT, descriptorSetLayout);
        VkDescriptorSetAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
        allocInfo.descriptorPool = descriptorPool;
        allocInfo.descriptorSetCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
        allocInfo.pSetLayouts = layouts.data();

        descriptorSets.resize(MAX_FRAMES_IN_FLIGHT);
        if (vkAllocateDescriptorSets(device, &allocInfo, descriptorSets.data()) != VK_SUCCESS) {
            throw std::runtime_error("failed to allocate descriptor sets!");
        }

        for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
            VkDescriptorBufferInfo bufferInfo{};
            bufferInfo.buffer = uniformBuffers[i];
            bufferInfo.offset = 0;
            bufferInfo.range = sizeof(UniformBufferObject);

            VkDescriptorImageInfo imageInfo{};
            imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
            imageInfo.imageView = textureImageView;
            imageInfo.sampler = textureSampler;

            std::array<VkWriteDescriptorSet, 2> descriptorWrites{};

            descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[0].dstSet = descriptorSets[i];
            descriptorWrites[0].dstBinding = 0;
            descriptorWrites[0].dstArrayElement = 0;
            descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
            descriptorWrites[0].descriptorCount = 1;
            descriptorWrites[0].pBufferInfo = &bufferInfo;

            descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            descriptorWrites[1].dstSet = descriptorSets[i];
            descriptorWrites[1].dstBinding = 1;
            descriptorWrites[1].dstArrayElement = 0;
            descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
            descriptorWrites[1].descriptorCount = 1;
            descriptorWrites[1].pImageInfo = &imageInfo;

            vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()), descriptorWrites.data(), 0, nullptr);
        }
    }
```

```
    VkCommandBuffer beginSingleTimeCommands() {
        VkCommandBufferAllocateInfo allocInfo{};
        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{};
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
        beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

        vkBeginCommandBuffer(commandBuffer, &beginInfo);

        return commandBuffer;
    }


    void endSingleTimeCommands(VkCommandBuffer commandBuffer) {
        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);

        vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
    }

    void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkBufferCopy copyRegion{};
        copyRegion.size = size;
        vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

        endSingleTimeCommands(commandBuffer);
    }


    bool isDeviceSuitable(VkPhysicalDevice device) {
        QueueFamilyIndices indices = findQueueFamilies(device);

        bool extensionsSupported = checkDeviceExtensionSupport(device);

        bool swapChainAdequate = false;
        if (extensionsSupported) {
            SwapChainSupportDetails swapChainSupport = querySwapChainSupport(device);
            swapChainAdequate = !swapChainSupport.formats.empty() && !swapChainSupport.presentModes.empty();
        }

        VkPhysicalDeviceFeatures supportedFeatures;
        vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

        return indices.isComplete() && extensionsSupported && swapChainAdequate && supportedFeatures.samplerA
nisotropy;
    }
```

Приклад нових функцій / методів:

```
    void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageLayout newLayout) { //  для обробки переходів макета

        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkImageMemoryBarrier barrier{}; // бар’єру пам’яті зображення

        barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

        barrier.oldLayout = oldLayout;

        barrier.newLayout = newLayout;

        barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

        barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

        barrier.image = image; // вказання зображення

        barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT; // вказання конкретну частину зображення, що зазнає впливу.

        barrier.subresourceRange.baseMipLevel = 0;

        barrier.subresourceRange.levelCount = 1;

        barrier.subresourceRange.baseArrayLayer = 0;

        barrier.subresourceRange.layerCount = 1;

        VkPipelineStageFlags sourceStage;

        VkPipelineStageFlags destinationStage;

        if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {

            barrier.srcAccessMask = 0;

            barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

            sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;

            destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
```

```
        } else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
&& newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
            barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
            barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
            sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
            destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
        } else {
            throw std::invalid_argument("unsupported layout
transition!");
        }
        vkCmdPipelineBarrier(
            commandBuffer, // привласнюємо буфер
            sourceStage, // на якому етапі конвеєра виконуються
операції, які мають відбуватися перед бар’єром
            destinationStage, // визначає стадію конвеєра, на якій
операції чекатимуть на бар’єрі.
            0, //  або 0або VK_DEPENDENCY_BY_REGION_BIT
            0, nullptr,
            0, nullptr,
            1, &barrier // перетворює бар'єр на умову для кожного
регіону.
        );
        endSingleTimeCommands(commandBuffer);
    }
    void copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t
width, uint32_t height) { // Копіювання буфера в зображення
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();
```

```
        VkBufferImageCopy region{};
        region.bufferOffset = 0; // Вказує зміщення байтів у буфері, з якого починаються значення пікселів
        region.bufferRowLength = 0;
        region.bufferImageHeight = 0 // (above too)спосіб розміщення пікселів у пам’яті
        region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        region.imageSubresource.mipLevel = 0;
        region.imageSubresource.baseArrayLayer = 0;
        region.imageSubresource.layerCount = 1;
        region.imageOffset = {0, 0, 0};
        region.imageExtent = {
            width,
            height,
            1
        }; // до якої частини зображення ми хочемо скопіювати пікселі.
        vkCmdCopyBufferToImage(commandBuffer, buffer, image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &region);
        endSingleTimeCommands(commandBuffer);
    }
```

Приклад нового вершинного шейдеру:

```
#version 450
layout(binding = 0) uniform UniformBufferObject {
```

```
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 2) in vec2 inTexCoord;
layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;
void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition,
0.0, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Приклад фрагментного шейдеру:

```
#version 450
layout(binding = 1) uniform sampler2D texSampler;

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;
```

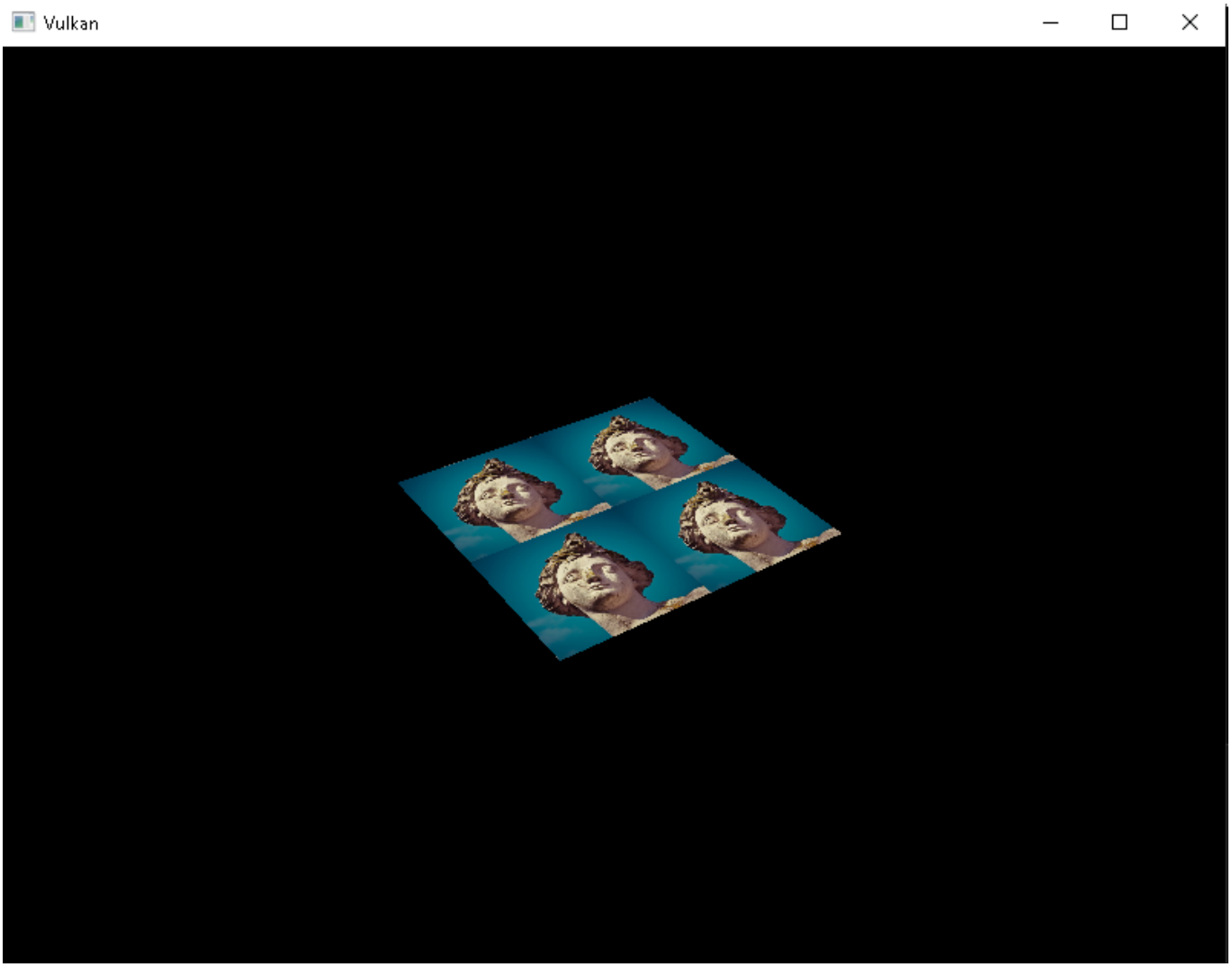
```
layout(location = 0) out vec4 outColor;

void main() {

    outColor = texture(texSampler, fragTexCoord * 2.0);

}
```

Результат запуску програми:



Вікно створене за допомогою GLFW відображає квадратну площину, яка замість звичайного розфарбовування накладена з текстурою.За допомоги зміни шейдеру фрагменту, ми розбили наш існуючий квадрат на уявних 4 (лише для відображення текстури, адже к-сть полігонів не змінилась).

Висновок:

Я успішно навчився створювати, редагувати, використовувати шейдери для взаємодії з текстурами у середовищі API Vulkan.

Дізнався про додаткову бібліотеку для завантаження зображень stb_image, яка дійсно зручна у поєднанні з GLFW. Але потім згадав свої спроби реалізувати считувач та растровий редактор на основі BMP файлів.

Опанував та додатково підкріпив базу знань про основні моделі / шаблони взаємодії чи створення текстур: Staging buffer, Текстура, Layout transitions, Копіювання зображення до буферу, Бар'єрні маски, Sample, Anisotropy filtering.