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 №3

з дисципліни

«Комп’ютерна графіка та мультимедіа»

Виконав:

студент групи ІП-05

Гапій Денис Едуардович

номер залікової : 0504

Перевірив:

Родіонов П. Ю.

Київ 2022

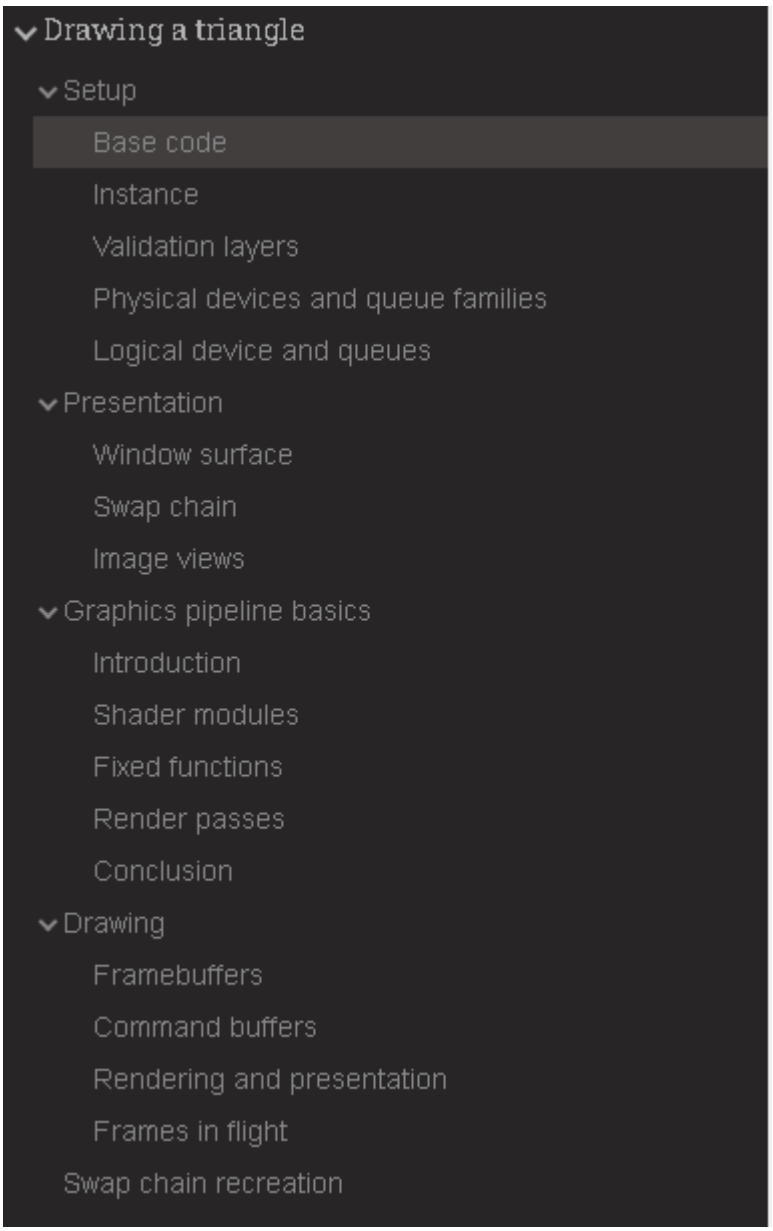
Тема: «Створення графічних примітивів у середовищі API Vulkan»

Мета: навчитися створювати та модифікувати графічні примітиви у середовищі API Vulkan

Контрольні запитання:

1. Структура програми, що виводить на екран графічний примітив.

   Загальна структура програми, що відмальовує примітив, буде виглядати наступним чином:



Але з попередньої лабораторної ми зрозуміли, що дана програма загорнута в клас, де зберігаються об’єкти Vulkan як приватні члени класу та додаються функції для ініціювання кожного з них, які викликатимуться з функції *initVulkan*. Коли все підготовлено, відбувається вхід в основний цикл *mainLoop*, який повторюється до тих пір, поки вікно не буде закрито; для початку рендерингу кадрів. Після того, як вікно закриється і mainLoop повернеться, необхідно звільнити ресурси, які були використані у функції очищення *cleanup*. Детальніше у Прикладі №1.

2. Поняття та створення екземпляра.

   Екземпляр — це зв’язок між програмою та бібліотекою Vulkan, і для її створення необхідно вказати драйверу деякі деталі про програму. Ці дані є технічно необов’язковими, але можуть надати деяку корисну інформацію драйверу для оптимізації конкретної програми. Детальніше у Прикладі №2.

3. Суть та призначення верифікаційних шарів (Verification Layers).

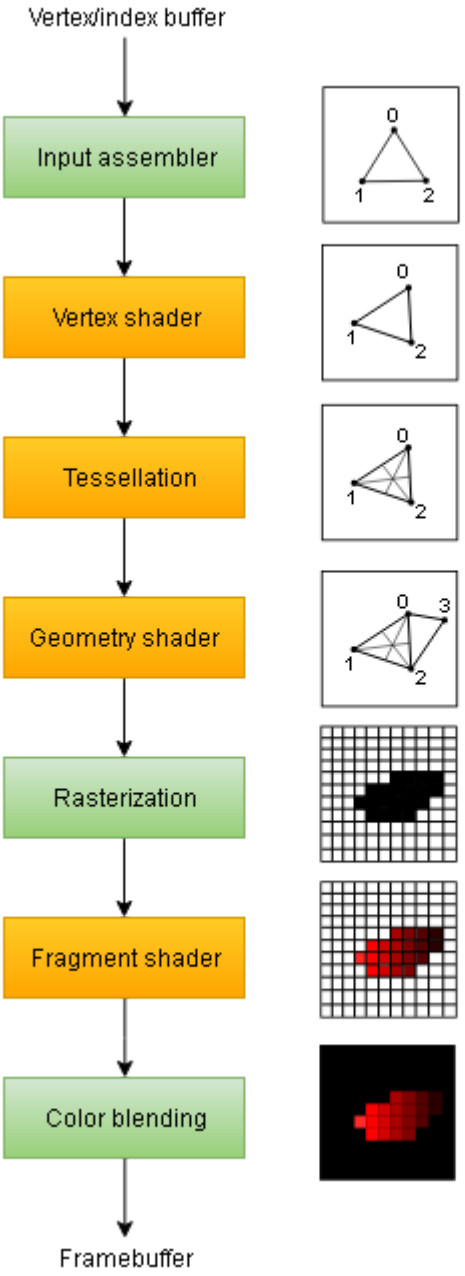
   Рівні перевірки — це додаткові компоненти, які підключаються до викликів функцій Vulkan для застосування додаткових операцій. Загальні операції в рівнях перевірки:

   - Перевірка значень параметрів на відповідність специфікації для виявлення неправильного використання
   - Відстеження створення та знищення об'єктів для пошуку витоків ресурсів
   - Перевірка безпеки потоків шляхом відстеження потоків, з яких походять виклики
   - Реєстрація кожного виклику та його параметрів у стандартний вихід
   - Відстеження Vulkan вимагає профілювання та повторного відтворення

   Детальніше у Прикладі №2.

4. Графічний конвеєр та його етапи.

Графічний конвеєр — це послідовність операцій, які переводять вершини та текстури сіток у пікселі (*у цілях візуалізації(. Нижче наведено спрощений вигляд:



Асемблер збирає необроблені дані вершин із вказаних буферів.

Вершинний шейдер запускається для кожної вершини і, як правило, застосовує перетворення позицій вершин із простору моделі на простір екрана.

Тесселяційні шейдери дозволяють розділяти геометрію на основі певних правил для підвищення якості сітки.

Геометричний шейдер запускається на кожному примітиві (трикутнику, лінії, точці) і може відкинути його або вивести більше примітивів, ніж надійшло.

Етап растеризації дискретизує примітиви на фрагменти . Це піксельні елементи, які вони заповнюють у кадровому буфері. Будь-які фрагменти, які виходять за межі екрана, відкидаються, а атрибути, виведені вершинним шейдером, інтерполюються між фрагментами.

Фрагментний шейдер викликається для кожного фрагмента, який залишається, і визначає, до якого буфера кадрів записуються фрагменти та з якими значеннями кольору та глибини.

На етапі змішування кольорів застосовуються операції для змішування різних фрагментів, які відображаються на той самий піксель у кадровому буфері. Фрагменти можуть просто перезаписувати один одного, складатися або змішуватися на основі прозорості.

Етапи із зеленим кольором відомі як каскади з фіксованою функцією . Ці етапи дозволяють налаштовувати їхні операції за допомогою параметрів, але спосіб їх роботи є заздалегідь визначеним.

Етапи з помаранчевим кольором, з іншого боку, це programmable, що означає, що ви можете завантажити власний код на графічну карту, щоб застосувати саме ті операції, які вам потрібні.

Детальніше у Прикладі №3

5. Поняття та призначення Swap Chain.
   Ланцюжок обміну — це, по суті, черга зображень, які очікують на показ на екрані. Наша програма отримає таке зображення, щоб намалювати його, а потім поверне в чергу. Загальна мета ланцюга обміну полягає в синхронізації представлення зображень із частотою оновлення екрана.
   Детальніше у Прикладі №4.

Приклади коду:

Приклад використаних заголовків та макросів:

```
#define GLFW_INCLUDE_VULKAN

#include <GLFW/glfw3.h>


#include <iostream>

#include <fstream>

#include <stdexcept>

#include <algorithm>

#include <vector>

#include <cstring>

#include <cstdlib>

#include <cstdint>

#include <limits>

#include <optional>

#include <set>

#define WIDTH = 800;

#define HEIGHT = 600;

#ifdef NDEBUG

const bool enableValidationLayers = false;

#else

const bool enableValidationLayers = true;

#endif
```

Приклад №1:

```
class HelloTriangleApplication {

public:

    void run() {

        initWindow();

        initVulkan();

        mainLoop();

        cleanup();

    }

private:

    void initWindow() {

        glfwInit(); // ініціалізує бібліотеку GLFW.

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API); // не створювати контекст OpenGL з наступним викликом:

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr); // створення вікна з відповідними розмірами та назвою

        glfwSetWindowUserPointer(window, this);// дозволяє зберігати довільний покажчик у ньому:

        glfwSetFramebufferSizeCallback(window, framebufferResizeCallback); // значення (розмір кадрового буфера) можна отримати з зворотного виклику, щоб правильно встановити позначку:

    }

    static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {

        auto app = reinterpret_cast<HelloTriangleApplication*>(glfwGetWindowUserPointer(window));
```

```
        app->framebufferResized = true;
    }
    void initVulkan() {
        createInstance(); // створення екземпляру
        setupDebugMessenger(); // налагодження повідомляча
        createSurface(); // створення ‘поверхні’ вікна
        pickPhysicalDevice(); // виділити потрібний пристрій (граф.)
        createLogicalDevice(); // створення логічного пристрою
        createSwapChain(); // створення ланцюжку обміну
        createImageViews(); // створює базове подання зображення для кожного зображення в ланцюжку обміну,
        createRenderPass(); // створюємо об’єкт, який містить к-сть буферів кольору та глибини, зразків використовувати для кожного з них і як слід обробляти їхній вміст
        createGraphicsPipeline(); // створення графічного конвеєру
        createFramebuffers(); // створення буфера кадрів
        createCommandPool(); // створення переліку команд з двох параметрів
        createCommandBuffers(); // створення буфера команд
        createSyncObjects(); // створення Семафору для стану зображення у ланцюжку обміну
    }
    void mainLoop() {
        while (!glfwWindowShouldClose(window)) {
            glfwPollEvents(); // обробка даних про події
            drawFrame(); // відмальовування наянвого кадру
        }
```

```
        vkDeviceWaitIdle(device); // синхронізація процесу перед
очищенням
    }
    void cleanup() { // повне очищення раніше створенних екземпляра,
конвеєра, ланцюга обміну, рендер пасу, семафору та інших об’єктів
        cleanupSwapChain();
        vkDestroyPipeline(device, graphicsPipeline, nullptr);
        vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
        vkDestroyRenderPass(device, renderPass, nullptr);
        for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
            vkDestroySemaphore(device, renderFinishedSemaphores[i],
nullptr);
            vkDestroySemaphore(device, imageAvailableSemaphores[i],
nullptr);
            vkDestroyFence(device, inFlightFences[i], nullptr);
        }
        vkDestroyCommandPool(device, commandPool, nullptr);
        vkDestroyDevice(device, nullptr);
        if (enableValidationLayers) {
            DestroyDebugUtilsMessengerEXT(instance, debugMessenger,
nullptr);
        }
        vkDestroySurfaceKHR(instance, surface, nullptr);
        vkDestroyInstance(instance, nullptr);
        glfwDestroyWindow(window);
        glfwTerminate();
    }
```

```
}
```

Приклад №2:

```
const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

VkResult CreateDebugUtilsMessengerEXT(VkInstance instance, const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo, const VkAllocationCallbacks* pAllocator, VkDebugUtilsMessengerEXT* pDebugMessenger) {
    auto func = (PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pDebugMessenger);
    }
    else {
        return VK_ERROR_EXTENSION_NOT_PRESENT;
    }
}

void DestroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT debugMessenger, const VkAllocationCallbacks* pAllocator) {
    auto func = (PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");
    if (func != nullptr) {
        func(instance, debugMessenger, pAllocator);
```

```
    }
}
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    bool isComplete() {
        return graphicsFamily.has_value() &&
presentFamily.has_value();
    }
};

    void createInstance() {
        if (enableValidationLayers && !checkValidationLayerSupport())
{
            throw std::runtime_error("validation layers requested, but
not available!");
        }

        VkApplicationInfo appInfo{};
        appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
        appInfo.pApplicationName = "Hello Triangle";
        appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
        appInfo.pEngineName = "No Engine";
        appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
        appInfo.apiVersion = VK_API_VERSION_1_0;
```

```
        VkInstanceCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
        createInfo.pApplicationInfo = &appInfo;

        auto extensions = getRequiredExtensions();
        createInfo.enabledExtensionCount =
static_cast<uint32_t>(extensions.size());
        createInfo.ppEnabledExtensionNames = extensions.data();

        VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo{};
        if (enableValidationLayers) {
            createInfo.enabledLayerCount =
static_cast<uint32_t>(validationLayers.size());
            createInfo.ppEnabledLayerNames = validationLayers.data();

            populateDebugMessengerCreateInfo(debugCreateInfo);
            createInfo.pNext =
(VkDebugUtilsMessengerCreateInfoEXT*)&debugCreateInfo;
        }
        else {
            createInfo.enabledLayerCount = 0;

            createInfo.pNext = nullptr;
        }

        if (vkCreateInstance(&createInfo, nullptr, &instance) !=
VK_SUCCESS) {
```

```
            throw std::runtime_error("failed to create instance!");
        }
    }

    void
populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT&
createInfo) {
        createInfo = {};
        createInfo.sType =
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
        createInfo.messageSeverity =
VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
        createInfo.messageType =
VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
        createInfo.pfnUserCallback = debugCallback;
    }

    void setupDebugMessenger() {
        if (!enableValidationLayers) return;

        VkDebugUtilsMessengerCreateInfoEXT createInfo;
        populateDebugMessengerCreateInfo(createInfo);
```

```
        if (CreateDebugUtilsMessengerEXT(instance, &createInfo,
nullptr, &debugMessenger) != VK_SUCCESS) {
            throw std::runtime_error("failed to set up debug
messenger!");
        }
    }
```

Приклад №3:

```
    void createGraphicsPipeline() {
        auto vertShaderCode = readFile("shaders/vert.spv");
        auto fragShaderCode = readFile("shaders/frag.spv");
        VkShaderModule vertShaderModule =
createShaderModule(vertShaderCode);
        VkShaderModule fragShaderModule =
createShaderModule(fragShaderCode);
        VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
        vertShaderStageInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
        vertShaderStageInfo.module = vertShaderModule;
        vertShaderStageInfo.pName = "main";

        VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
        fragShaderStageInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
        fragShaderStageInfo.module = fragShaderModule;
        fragShaderStageInfo.pName = "main";
```

```
        VkPipelineShaderStageCreateInfo shaderStages[] = {
vertShaderStageInfo, fragShaderStageInfo };

        VkPipelineVertexInputStateCreateInfo vertexInputInfo{};

        vertexInputInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

        vertexInputInfo.vertexBindingDescriptionCount = 0;

        vertexInputInfo.vertexAttributeDescriptionCount = 0;


        VkPipelineInputAssemblyStateCreateInfo inputAssembly{};

        inputAssembly.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

        inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

        inputAssembly.primitiveRestartEnable = VK_FALSE;

        VkPipelineViewportStateCreateInfo viewportState{};

        viewportState.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;

        viewportState.viewportCount = 1;

        viewportState.scissorCount = 1;

        VkPipelineRasterizationStateCreateInfo rasterizer{};

        rasterizer.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;

        rasterizer.depthClampEnable = VK_FALSE;

        rasterizer.rasterizerDiscardEnable = VK_FALSE;

        rasterizer.polygonMode = VK_POLYGON_MODE_FILL;

        rasterizer.lineWidth = 1.0f;

        rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;

        rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
```

```
        rasterizer.depthBiasEnable = VK_FALSE;

        VkPipelineMultisampleStateCreateInfo multisampling{};

        multisampling.sType =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

        multisampling.sampleShadingEnable = VK_FALSE;

        multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

        VkPipelineColorBlendAttachmentState colorBlendAttachment{};

        colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT
| VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT |
VK_COLOR_COMPONENT_A_BIT;

        colorBlendAttachment.blendEnable = VK_FALSE;

        VkPipelineColorBlendStateCreateInfo colorBlending{};

        colorBlending.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;

        colorBlending.logicOpEnable = VK_FALSE;

        colorBlending.logicOp = VK_LOGIC_OP_COPY;

        colorBlending.attachmentCount = 1;

        colorBlending.pAttachments = &colorBlendAttachment;

        colorBlending.blendConstants[0] = 0.0f;

        colorBlending.blendConstants[1] = 0.0f;

        colorBlending.blendConstants[2] = 0.0f;

        colorBlending.blendConstants[3] = 0.0f;

        std::vector<VkDynamicState> dynamicStates = {

            VK_DYNAMIC_STATE_VIEWPORT,

            VK_DYNAMIC_STATE_SCISSOR

        };

        VkPipelineDynamicStateCreateInfo dynamicState{};
```

```
        dynamicState.sType =
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;

        dynamicState.dynamicStateCount =
static_cast<uint32_t>(dynamicStates.size());

        dynamicState.pDynamicStates = dynamicStates.data();


        VkPipelineLayoutCreateInfo pipelineLayoutInfo{};

        pipelineLayoutInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

        pipelineLayoutInfo.setLayoutCount = 0;

        pipelineLayoutInfo.pushConstantRangeCount = 0;

        if (vkCreatePipelineLayout(device, &pipelineLayoutInfo,
nullptr, &pipelineLayout) != VK_SUCCESS) {

            throw std::runtime_error("failed to create pipeline
layout!");

        }

        VkGraphicsPipelineCreateInfo pipelineInfo{};

        pipelineInfo.sType =
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

        pipelineInfo.stageCount = 2;

        pipelineInfo.pStages = shaderStages;

        pipelineInfo.pVertexInputState = &vertexInputInfo;

        pipelineInfo.pInputAssemblyState = &inputAssembly;

        pipelineInfo.pViewportState = &viewportState;

        pipelineInfo.pRasterizationState = &rasterizer;

        pipelineInfo.pMultisampleState = &multisampling;

        pipelineInfo.pColorBlendState = &colorBlending;
```

```
        pipelineInfo.pDynamicState = &dynamicState;

        pipelineInfo.layout = pipelineLayout;

        pipelineInfo.renderPass = renderPass;

        pipelineInfo.subpass = 0;

        pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;

        if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1,
&pipelineInfo, nullptr, &graphicsPipeline) != VK_SUCCESS) {

            throw std::runtime_error("failed to create graphics
pipeline!");

        }

        vkDestroyShaderModule(device, fragShaderModule, nullptr);

        vkDestroyShaderModule(device, vertShaderModule, nullptr);

    }
```

Приклад №4:

```
const std::vector<const char*> deviceExtensions = {

    VK_KHR_SWAPCHAIN_EXTENSION_NAME

};

     void createSwapChain() {

        SwapChainSupportDetails swapChainSupport =
querySwapChainSupport(physicalDevice);

        VkSurfaceFormatKHR surfaceFormat =
chooseSwapSurfaceFormat(swapChainSupport.formats);

        VkPresentModeKHR presentMode =
chooseSwapPresentMode(swapChainSupport.presentModes);

        VkExtent2D extent =
chooseSwapExtent(swapChainSupport.capabilities);
```

```
        uint32_t imageCount =
swapChainSupport.capabilities.minImageCount + 1;

        if (swapChainSupport.capabilities.maxImageCount > 0 &&
imageCount > swapChainSupport.capabilities.maxImageCount) {

            imageCount = swapChainSupport.capabilities.maxImageCount;

        }

        VkSwapchainCreateInfoKHR createInfo{};

        createInfo.sType =
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

        createInfo.surface = surface;

        createInfo.minImageCount = imageCount;

        createInfo.imageFormat = surfaceFormat.format;

        createInfo.imageColorSpace = surfaceFormat.colorSpace;

        createInfo.imageExtent = extent;

        createInfo.imageArrayLayers = 1;

        createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

        QueueFamilyIndices indices =
findQueueFamilies(physicalDevice);

        uint32_t queueFamilyIndices[] = {
indices.graphicsFamily.value(), indices.presentFamily.value() };

        if (indices.graphicsFamily != indices.presentFamily) {

            createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;

            createInfo.queueFamilyIndexCount = 2;

            createInfo.pQueueFamilyIndices = queueFamilyIndices;

        }

        else {

            createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

```
        }

        createInfo.preTransform =
swapChainSupport.capabilities.currentTransform;

        createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;

        createInfo.presentMode = presentMode;

        createInfo.clipped = VK_TRUE;


        if (vkCreateSwapchainKHR(device, &createInfo, nullptr,
&swapChain) != VK_SUCCESS) {

            throw std::runtime_error("failed to create swap chain!");

        }


        vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
nullptr);

        swapChainImages.resize(imageCount);

        vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
swapChainImages.data());


        swapChainImageFormat = surfaceFormat.format;

        swapChainExtent = extent;

    }

    void recreateSwapChain() {

        int width = 0, height = 0;

        glfwGetFramebufferSize(window, &width, &height);

        while (width == 0 || height == 0) {

            glfwGetFramebufferSize(window, &width, &height);

            glfwWaitEvents();
```

```
        }

        vkDeviceWaitIdle(device);

        cleanupSwapChain();

        createSwapChain();
        createImageViews();
        createFramebuffers();
    }
    void cleanupSwapChain() { // очистка ланцюга обміну
        for (auto framebuffer : swapChainFramebuffers) {
            vkDestroyFramebuffer(device, framebuffer, nullptr);
        }
        for (auto imageView : swapChainImageViews) {
            vkDestroyImageView(device, imageView, nullptr);
        }
        vkDestroySwapchainKHR(device, swapChain, nullptr);
    }
```

Приклад Вершинного шейдеру:

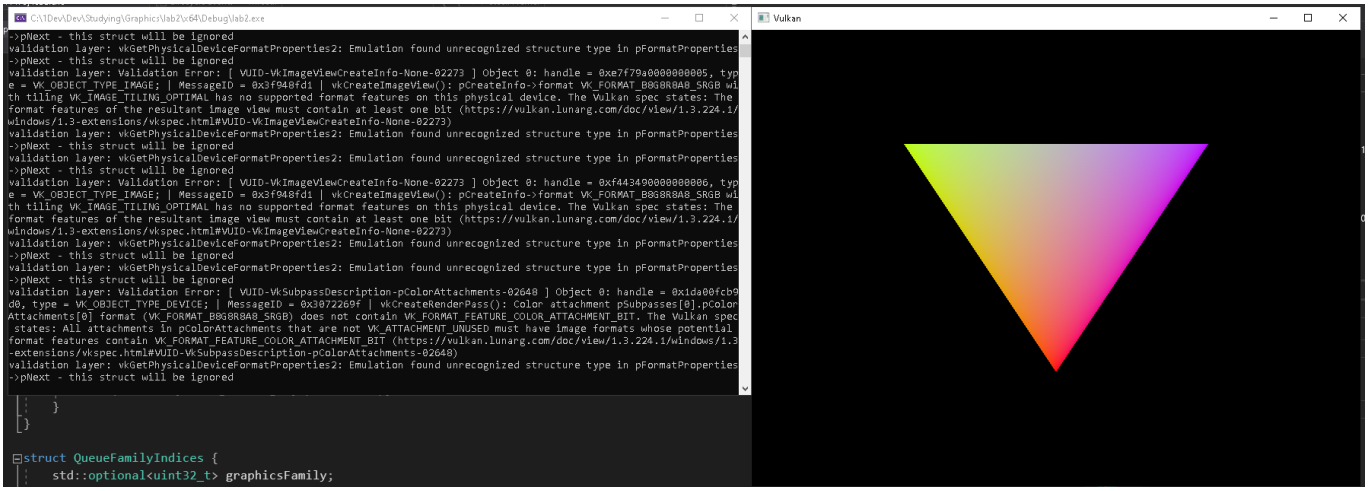
```
#version 450
layout(location = 0) out vec3 fragColor;
vec2 positions[3] = vec2[](
     vec2(-0.5, -0.5),
    vec2(0.5, -0.5),
```

```
     vec2(0.0, 0.5)
);
vec3 colors[3] = vec3[](
    vec3(0.5, 1.0, 0.0),
    vec3(0.5, 0.0, 1.0),
     vec3(1.0, 0.0, 0.0)
);
void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

Приклад Фрагментного шейдеру:

```
layout(location = 0) in vec3 fragColor;
layout(location = 0) out vec4 outColor;
void main() {
outColor = vec4(fragColor, 1.0);
}
```

Результат запуску програми:



Вивід у консоль відлагодження усіх процесів через діагностичні повідомлення, та вікно створене за допомогою GLFW, що відображає перевернутий трикутник із градієнтним зафарбуванням.

Висновок:

Я успішно навчився створювати та модифікувати графічні примітиви (у моєму випадку це трикутник) у середовищі API Vulkan, мовою c++.

Створив свої перші вертексний (вершинний) та фрагментний шейдери, через засіб **glslc,** що надається разом з Vulkan SDK.

Опанував загальний план розробки застосунку для відображення шейдерів за допомогою Vulkan API: від створення екземплярів, шарів перевірки, та пошуку фізичних пристроїв для відображення; й до реалізації графічного конвеєра, ланцюгу обміну, буферів команд та кадрів, семафорів і тд.

Труднощі виникли тільки в опрацюванні такої великої кількості теоретичного матеріалу, для розуміння головних тонкощей рендерингу та роботи шейдерів у Vulkan. Але завдяки якісно перекладеній методичці та офіційного підручника по опануванню Vulkan (*де до того ж ще й зручно / красиво оформлено розділи та частинки з кодом).