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 №7

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: «Робота з буфером глибини у середовищі API Vulkan»

Мета: навчитися працювати з буфером глибини у середовищі API Vulkan.

Контрольні запитання:

1. Поняття та призначення буферу глибини.

Вкладення глибини базується на зображенні, як і вкладення кольору. Різниця полягає в тому, що ланцюжок обміну не буде автоматично створювати зображення глибини для нас. Нам потрібне лише одне зображення глибини, оскільки одночасно виконується лише одна операція малювання. Зображення глибини знову вимагатиме трьох ресурсів: зображення, пам’ять і перегляд зображення. Буфер глибини — це додаткове вкладення, яке зберігає глибину для кожної позиції, так само як вкладення кольору зберігає колір кожної позиції. Щоразу, коли растеризатор створює фрагмент, перевірка глибини перевірятиме, чи новий фрагмент ближчий за попередній. Якщо ні, то новий фрагмент відкидається. Фрагмент, який пройшов перевірку глибини, записує власну глибину в буфер глибини. Загалом буфер глибини дозволяє представити декілька фігур з текстурами без артефактних накладань одна на одну.

2. Охарактеризуйте етапи, необхідні для задання глибини зображення.

- Створення іншої фігури для наявності 3Д.
- Нам потрібне лише одне зображення глибини, оскільки одночасно виконується лише одна операція малювання. Зображення глибини знову вимагатиме трьох ресурсів: зображення, пам’ять і перегляд зображення
- Буфер глибини зчитується з буфера глибини, щоб перевірити, чи видимий фрагмент, і буде записаний, коли малюється новий фрагмент.
- Тепер ми збираємося змінити createRenderPass, щоб включити глибинне кріплення.
- Переконатися, що немає конфлікту між переходом зображення глибини та його очищенням під час операції завантаження. До зображення

глибини вперше звертаються на стадії конвеєра тестування раннього фрагмента, і оскільки ми маємо операцію завантаження, яка очищає , ми повинні вказати маску доступу для записів.

- Наступним кроком є зміна створення кадрового буфера, щоб прив’язати зображення глибини до вкладення глибини.

3. Кадровий буфер та його роль у процесі створення 3D.
Це частина пам'яті комп'ютера , яка використовується комп'ютерною програмою для представлення вмісту, який буде показано на дисплеї комп'ютера . Екранний буфер також можна назвати відеобуфером
Кольорове вкладення відрізняється для кожного зображення ланцюжка обміну, але однакове зображення глибини може використовуватися всіма, оскільки завдяки нашим семафорам одночасно працює лише один підпрохід. У 3D це допоможе зображенню причепити глибину.

4. Використовувані формати для зображення глибини.
Попри велику поширенність та підтримку формату VK_FORMAT_D32_SFLOATформат ми реалізували функцію, що буде шукати для кожного кандидата зручний йому формат, що ним підтримується.
Серед наявних, є:

- linearTilingFeatures: сценарії використання, які підтримуються лінійним мозаїкою
- optimalTilingFeatures: сценарії використання, які підтримуються оптимальною мозаїкою
- bufferFeatures: випадки використання, які підтримуються для буферів

5. Функції Vulkan, які використовуються під час роботи з буфером глибини.

- vkGetPhysicalDeviceFormatProperties - перераховує можливості форматування фізичного пристрою
- vkCreateImageView / vkDestroyImageView

- vkCreateImage / vkDestroyImage
- звісно ж vkFreeMemory

Приклади коду:

Приклад нових функцій / методів:

Оновлення структури вершини:

```
struct Vertex {
    glm::vec3 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> getAttribut
        std::array<VkVertexInputAttributeDescription, 3> attributeDescr

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

Змінюємо координати умовних фігур:

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

    {{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
};

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4
};
```

Змінні для збереження глибини:

```
VkImage depthImage;
VkDeviceMemory depthImageMemory;
VkImageView depthImageView;
```

Редагуємо функції створення вигляду зображення та шляху рендеру:

```
        for (uint32_t i = 0; i < swapChainImages.size(); i++) {
            swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat, VK_IMAGE_ASPEC
T_COLOR_BIT);
        }
    }

    void createRenderPass() {
        VkAttachmentDescription colorAttachment{};
        colorAttachment.format = swapChainImageFormat;
        colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
        colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
        colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

        VkAttachmentDescription depthAttachment{};
        depthAttachment.format = findDepthFormat();
        depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
        depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

        VkAttachmentReference colorAttachmentRef{};
        colorAttachmentRef.attachment = 0;
        colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

        VkAttachmentReference depthAttachmentRef{};
        depthAttachmentRef.attachment = 1;
        depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

        VkSubpassDescription subpass{};
        subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
        subpass.colorAttachmentCount = 1;
        subpass.pColorAttachments = &colorAttachmentRef;
        subpass.pDepthStencilAttachment = &depthAttachmentRef;

        VkSubpassDependency dependency{};
        dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
        dependency.dstSubpass = 0;
        dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRA
GMENT_TESTS_BIT;
        dependency.srcAccessMask = 0;
        dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRA
GMENT_TESTS_BIT;
        dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_
WRITE_BIT;

        std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};
        VkRenderPassCreateInfo renderPassInfo{};
        renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
        renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
        renderPassInfo.pAttachments = attachments.data();
        renderPassInfo.subpassCount = 1;
```

Виставляємо параметри у графічному конвеєрі:

```
    VkPipelineDepthStencilStateCreateInfo depthStencil{};
    depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
    depthStencil.depthTestEnable = VK_TRUE;
    depthStencil.depthWriteEnable = VK_TRUE;
    depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
    depthStencil.depthBoundsTestEnable = VK_FALSE;
    depthStencil.stencilTestEnable = VK_FALSE;
```

Доповнення конвеєра та буфера кадру:

```
        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.pDepthStencilState = &depthStencil;
        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);
    }

    void createFramebuffers() {
        swapChainFramebuffers.resize(swapChainImageViews.size());

        for (size_t i = 0; i < swapChainImageViews.size(); i++) {
            std::array<VkImageView, 2> attachments = {
                swapChainImageViews[i],
                depthImageView
            };

            VkFramebufferCreateInfo framebufferInfo{};
            framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
            framebufferInfo.renderPass = renderPass;
            framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
            framebufferInfo.pAttachments = attachments.data();
            framebufferInfo.width = swapChainExtent.width;
            framebufferInfo.height = swapChainExtent.height;
            framebufferInfo.layers = 1;

            if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, &swapChainFramebuffers[i]) != VK_SUCCE
SS) {
                throw std::runtime_error("failed to create framebuffer!");
            }
        }
```

Нові функції створення зображення з глибиною, підтримки / пошуку формату:

```
    void createDepthResources() {
        VkFormat depthFormat = findDepthFormat();

        createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTIMAL, VK_I
MAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
        depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);
    }

    VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling, VkFormatFeatu
reFlags features) {
        for (VkFormat format : candidates) {
            VkFormatProperties props;
            vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

            if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
                return format;
            } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) == featu
res) {
                return format;
            }
        }

        throw std::runtime_error("failed to find supported format!");
    }

    VkFormat findDepthFormat() {
        return findSupportedFormat(
        {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
            VK_IMAGE_TILING_OPTIMAL,
            VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
        );
    }

    bool hasStencilComponent(VkFormat format) {
        return format == VK_FORMAT_D32_SFLOAT_S8_UINT || format == VK_FORMAT_D24_UNORM_S8_UINT;
    }
```

Так як ми маємо багато вкладень з VK_ATTACHMENT_LOAD_OP_CLEAR, ми мусимо додати й значення для очищення. В recordCommandBuffer ми створюємо масив структур VkClearValue:

```
    void recordCommandBuffer(VkCommandBuffer commandBuffer, uint32_t imageIndex) {
        VkCommandBufferBeginInfo beginInfo{};
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

        if (vkBeginCommandBuffer(commandBuffer, &beginInfo) != VK_SUCCESS) {
            throw std::runtime_error("failed to begin recording command buffer!");
        }

        VkRenderPassBeginInfo renderPassInfo{};
        renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
        renderPassInfo.renderPass = renderPass;
        renderPassInfo.framebuffer = swapChainFramebuffers[imageIndex];
        renderPassInfo.renderArea.offset = {0, 0};
        renderPassInfo.renderArea.extent = swapChainExtent;

        std::array<VkClearValue, 2> clearValues{};
        clearValues[0].color = {{0.0f, 0.0f, 0.0f, 1.0f}};
        clearValues[1].depthStencil = {1.0f, 0};

        renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
        renderPassInfo.pClearValues = clearValues.data();
```

Очищення ланцюга:

```
    void cleanupSwapChain() {
        vkDestroyImageView(device, depthImageView, nullptr);
        vkDestroyImage(device, depthImage, nullptr);
        vkFreeMemory(device, depthImageMemory, nullptr);
```

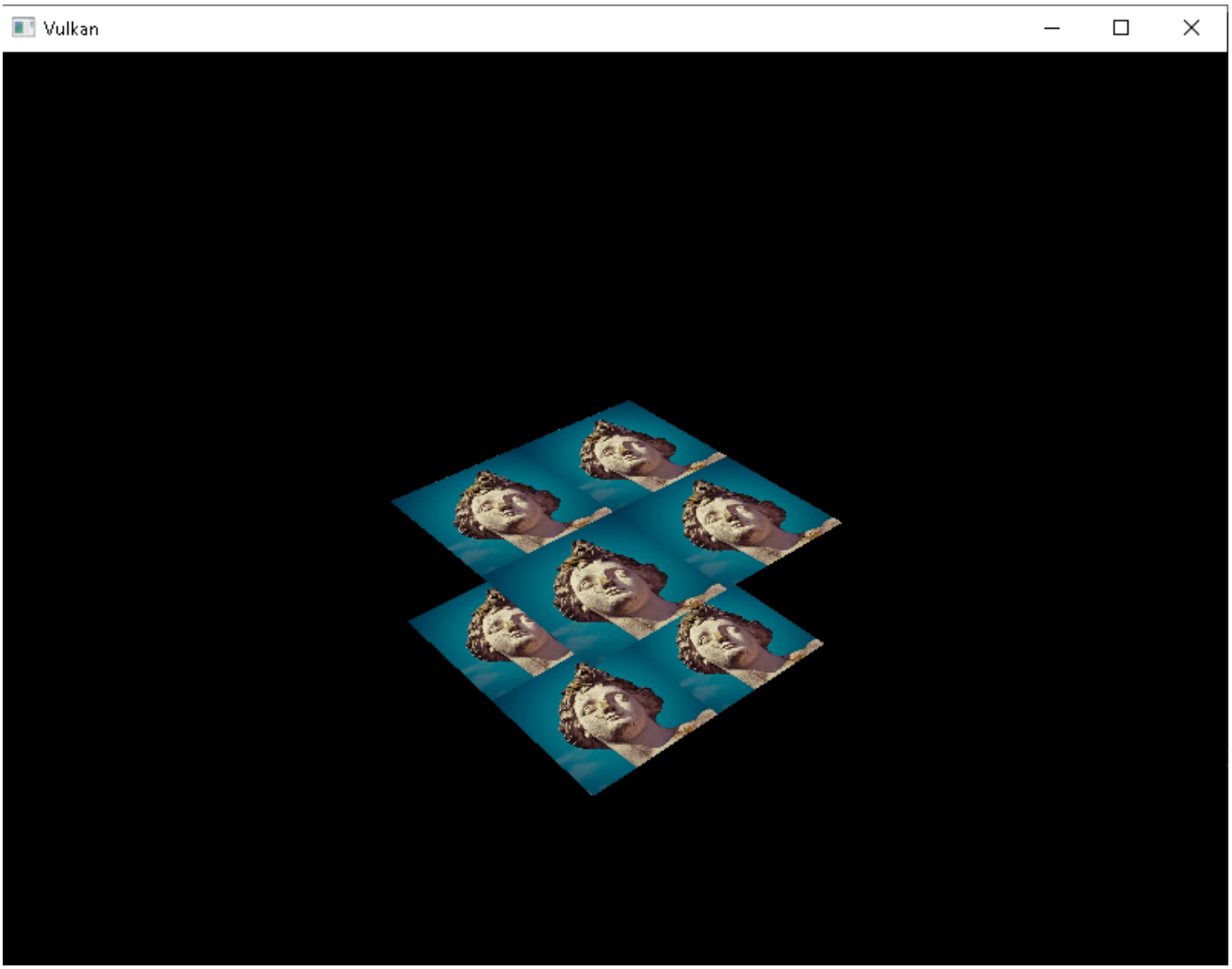
Приклад нового вершинного шейдеру:

```
#version 450


layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
layout(location = 0) in vec3 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,
1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
```

```
}
```

Результат запуску програми:



Вікно створене за допомогою GLFW відображає дві квадратні площину, яка знаходяться на різній висоті і відображають можливості трьох-вимірного представлення фігур з коректною глибиною зображення.

Висновок:

Я успішно навчився працювати з буфером глибини у середовищі API Vulkan.

Дізнався та вдосконалив знання про: буфер кадрів, глибину зображення, формат відтворення зображення, і так далі.

Додавши буфер глибини до нашого застосунку, тепер я можу реалізовувати цікаві трьохвимірні моделі за допомогою полігонів, фігур, текстур та й шейдерів загалом.