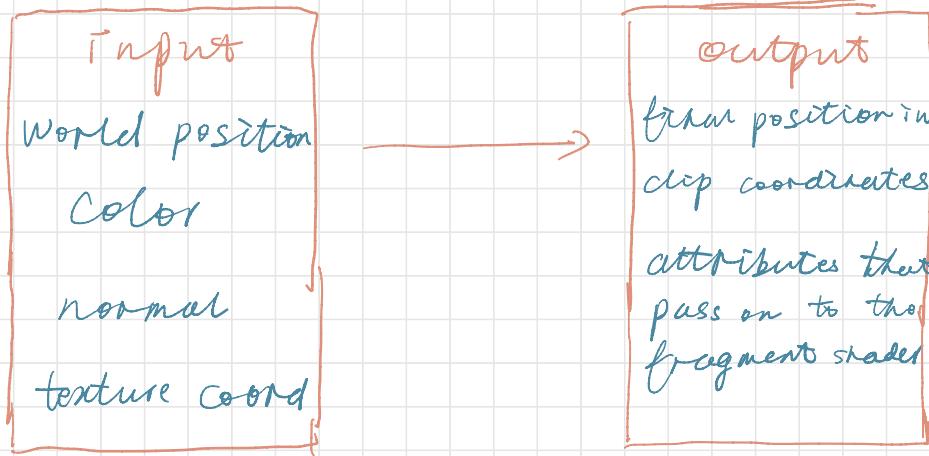


Vulkan 学习笔记

Vertex Shader

The vertex shader processes each incoming vertex,



顶点着色器将返回顶点
缓冲区的输入使用 `in` 关键字。

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

`layout(location = 1) in vec3 color;`

(build-in) `gl_VertexIndex` variable contains the index of the current vertex

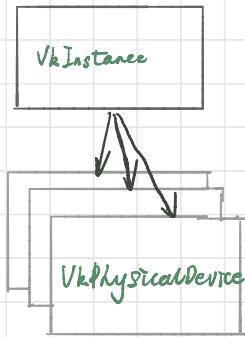
Fragment Shader

The fragment shader is invoked on fragments to produce a color and depth for framebuffer (or framebuffers)

The main function is call for every fragments just like the vertex shader main function is call for every vertex.

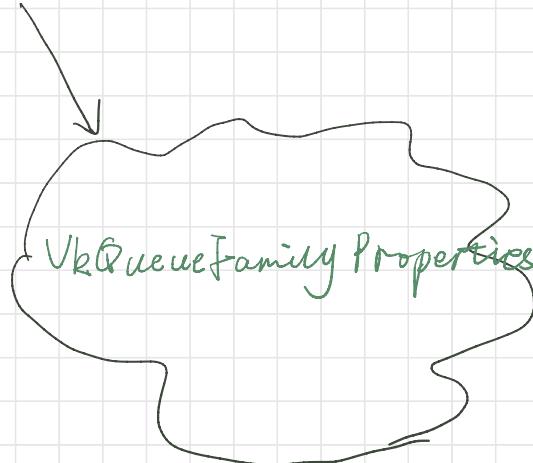
Unlike `gl-position` in vertex shader, there is no build-in variable to output a color for each framebuffer where the `layout (location = 0)` modifier specifies the index of the framebuffer.

物理设备与队列



通过 `VkInstance` 初始化后，需要在系统中查找并选择一个支持所需功能的显示（即物理设备，`physical device`）

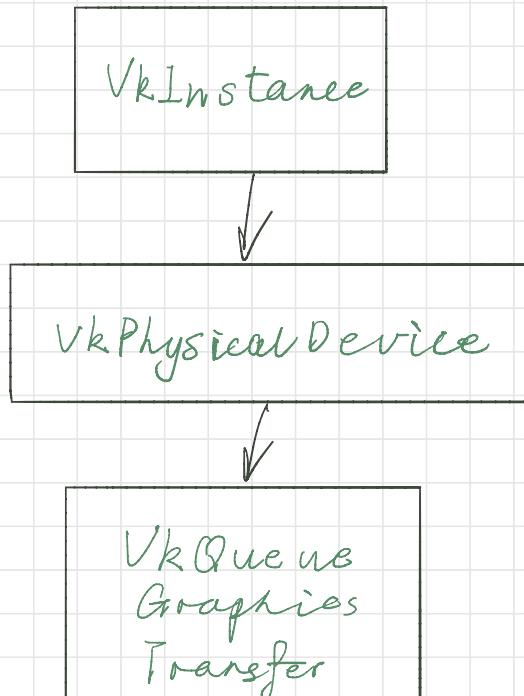
几乎所有的 vulkan 操作，从绘图到上传纹理，都需要将命令推送到队列中。



有关队列簇，结构体 `VkQueue Family Properties` 包含了硬件信息，包括支持的操作类型和基于当前队列簇可以创建的有效队列数。

逻辑设备名与队列

在选择要使用的物理设备之后，需要设置一个逻辑设备（VkDevice）用于交互。



Queue Family Properties =

圖形管絆

Vertex / Index Buffer

Input Assembly

↓
Vertex shader



Tessellation



Geometry Shader



Rasterization



Fragment Shader

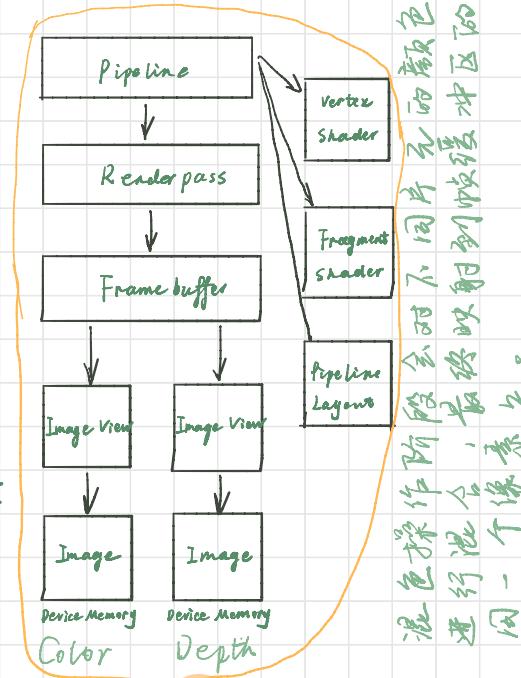


Color Blending

所調圖形管絆就是指
這些根據你所圖的形
狀來產生頂點座標

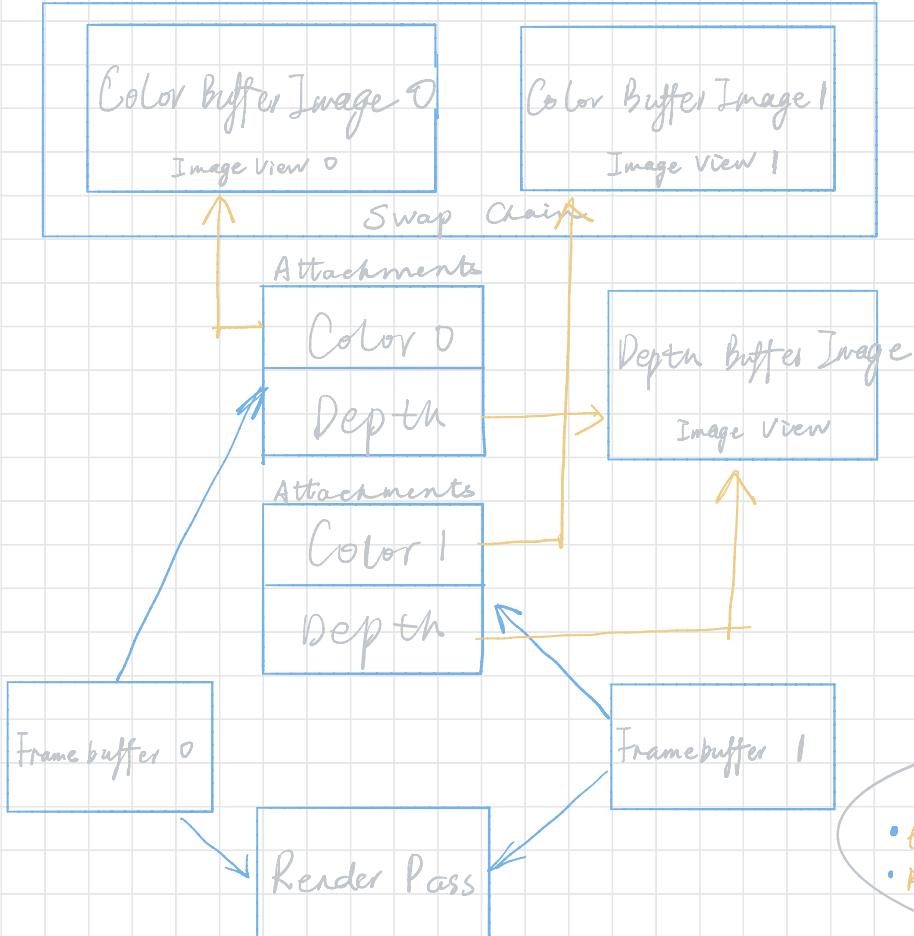
綠色的阶段被称为圖形流水线。
这个阶段允许使用自定义的参数。

绿色阶段被称为可编程阶段。
程序员可以向 GPU 提交
自己编写的代码执行且修改



Frame buffers

帧缓冲区



```
1 for (size_t i = 0; i < swapChainImageViews.size(); i++) {  
2     VkImageView attachments[] = {  
3         swapChainImageViews[i]  
4     };  
5  
6     VkFramebufferCreateInfo framebufferInfo{};  
7     framebufferInfo.sType =  
8         VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;  
9     framebufferInfo.renderPass = renderPass;  
10    framebufferInfo.attachmentCount = 1;  
11    framebufferInfo.pAttachments = attachments;  
12    framebufferInfo.width = swapChainExtent.width;  
13    framebufferInfo.height = swapChainExtent.height;  
14    framebufferInfo.layers = 1;  
15  
16    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr,  
17        &swapChainFramebuffers[i]) != VK_SUCCESS) {  
18        throw std::runtime_error("failed to create framebuffer!");  
19    }  
20 }
```

A framebuffer object references all of the Vk ImageView objects that represent the attachments.

附件中包含的 ImageView 内容

Summary

- **Shader stages:** 颜色器核纹理又了图形管道可编程的 pipeline
- **fixed-function state:** 结构体与固定管道功能，比如 Input Assembly ...
- **pipeline layout:** 管道布局又 uniform 及 push values
- **render pass:** 渲染通道通过管线阶段引用附件，并定义它的引用方式

渲染通道

在完成管道的创建工作之前，需要告诉 Vulkan 渲染时候使用的 framebuffer 附件相关信息。需要指定多少个颜色和深度缓冲区将会被使用，指定多少个采样器被用到，及在整个渲染操作中相关信息如何处理。所有的这些信息都被封装在一个叫做 render pass 的对象中。

LoadOp 和 storeOp 应用在颜色和深度数据，同时
stencilLoadOp / stencilStoreOp 应用在模板数据

附件描述 (VkAttachmentDescription)

• format：与交换链中图像的格式相匹配

• samples：采样器的大小

• LoadOp：

VK_ATTACHMENT_LOAD_OP_LOAD

保存已经存在于当前附件的内容

VK_ATTACHMENT_LOAD_OP_CLEAR

初始阶段以一个完全清理附件内容

VK_ATTACHMENT_LOAD_OP_DONT_CARE

存在的内容不关心，忽略它们

• storeOp：

VK_ATTACHMENT_STORE_OP_STORE

渲染的内容会存在内存，并在之后进行读取操作

VK_ATTACHMENT_STORE_OP_DONT_CARE

帧缓冲区的内容在渲染操作完毕后沿用为 undefined

• stencilLoadOp / stencilStoreOp

• initialLayout 图像在开始进入渲染通道 render pass 前将要使用如布局结构

• finalLayout 当渲染通道结束的调度时使用的布局

一些简单的附件布局

- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

Images used as color attachment

- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

Images to be presented in swapchain

- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL

Images to be used as destination for memory copy operation

CPU 对内存中的读写往往需要线性顺序的 linear memory layout，

但是在很多时候对于像素级别的操作是跳跃性连续的。这种情况更多地发生在 GPU 操作中，所以 GPU 硬件更多的是基于平铺 (Tiled) 或者基于最佳的内存布局结构，来降低从 GPU 处理的数据的开销。

所以从 CPU linear layout 内存数据到 GPU optimal layout

内存数据的读写存在数据存储格式的优化以及聚合。

顶点输入

顶点着色器接受顶点缓冲区的输入使用顶点属性

创建一个CPU写入的缓冲区



使用memcpy将顶点数据直接复制到缓冲区



使用暂存缓冲区将顶点数据赋值到高性能内存

属性描述

VkVertexInputBindingDescription

- binding: 指定了数组中对应的哪项属性
- stride: 指定一个条目到下一个条目的字节数
- inputRate:

VK_VERTEX_INPUT_RATE_VERTEX: 移动到每个顶点后的下一个数据条目

VK_VERTEX_INPUT_RATE_INSTANCE: 在每个instance之后移动到下一个数据条目

```
1 #version 450
2 #extension GL_ARB_separate_shader_objects : enable
3
4 layout(location = 0) in vec2 inPosition;
5 layout(location = 1) in vec3 inColor;
6
7 layout(location = 0) out vec3 fragColor;
8
9 void main() {
10     gl_Position = vec4(inPosition, 0.0, 1.0);
11     fragColor = inColor;
12 }
```

```
struct Vertex{
    glm::Vec2 pos;
    glm::Vec3 color;
}
```

VkVertexInputAttribute

- offset: 指定了每个顶点数据读取的字节偏移量

VkVertexInputAttribute

- binding: 每个顶点的数据来源
- location: 引用了 vertex shader 为输入的 location 声名。
顶点着色器中, location 为 0 表示 position
- format:

float: VK_FORMAT_R32_SFLOAT

Vec2: VK_FORMAT_R32G32_SFLOAT

Vec3: VK_FORMAT_R32G32B32_SFLOAT

Vec4: VK_FORMAT_R32G32B32A32_SFLOAT

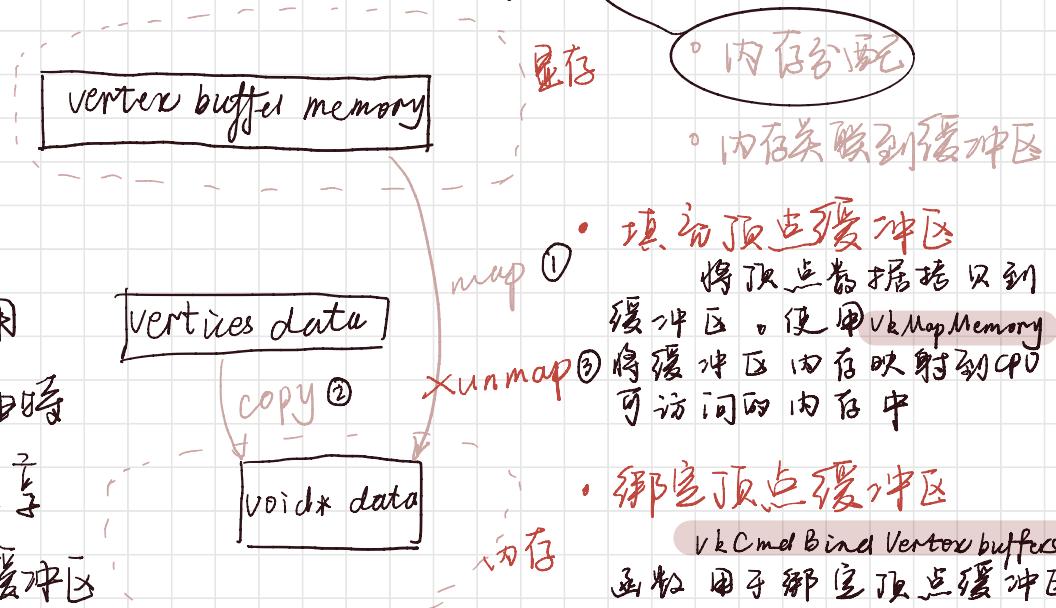
图形管线接受封装好的 vertices 顶点数据并将其格式的顶点数据传递到 Vertex shader

创建顶点缓冲区

缓冲区是内存的一块区域，该区域用于向显卡提供需要读取的数据。它们可以用来存储顶点数据，也可以用于其他目的。

VkBufferCreateInfo

- size：指定缓冲区大小
- usage：缓冲区数据将如何使用
- sharingMode：缓冲区也可以由多个队列族占有或者多个同时共享
- flags：用于配置共享的内存缓冲区



Stage Buffer 临时缓冲区

stage buffer 的顶点数据内存映射和持久，将使用不一样的 usage flags:

当顶点数据量很大的时候，再开一个

GPU 的 local buffer，将 CPU buffer 复制到

GPU 的 local buffer 中，这样可以提升性能

将使用缓冲区拷贝的命令将数据从暂存缓冲区

拷贝到实际的图形显卡中。

从一个缓冲区拷贝到另一个缓冲区

```
1 void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize  
size) {  
2     VkCommandBufferAllocateInfo allocInfo{};  
3     allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;  
4     allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;  
5     allocInfo.commandPool = commandPool;  
6     allocInfo.commandBufferCount = 1;  
7  
8     VkCommandBuffer commandBuffer;  
9     vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);  
10 }
```

And immediately start recording the command buffer:

```
1 VkCommandBufferBeginInfo beginInfo{};  
2 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;  
3 beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;  
4  
5 vkBeginCommandBuffer(commandBuffer, &beginInfo);
```

• VK_BUFFER_USAGE_TRANSFER_SRC_BIT

缓冲区可以用于源内存传输操作

usage

• VK_BUFFER_USAGE_TRANSFER_DST_BIT

缓冲区可以用于目标内存传输操作

usage

Stage buffer

vertex buffer

内存分配类型为 host

内存分配类型为 device

• VK_BUFFER_USAGE_VERTEX_BUFFER_BIT

index buffer (同 vertex buffer) 的创建步骤类似
Usage flag: VK_BUFFER_USAGE_INDEX_BUFFER_BIT

描述符和缓冲区

可以将任意属性传递给每个顶点着色器使用。但是全局变量呢？当涉及到引用图形相关内容时，需要一个模型视图投影矩阵。可以将其封装为顶点数据，但是这非常浪费带宽、内存，并且需要在变换的时候更新顶点缓冲区的数据。

在Vulkan中正确处理此问题的途径是使用资源描述符(resource descriptors)。描述符是着色器自由访问缓冲区和图像资源的一种方式。描述符使用由三部分组成：

- 在管程创建时指定描述符布局结构
- 从描述符对象池中分配描述符集合
- 在渲染阶段绑定描述符集合

1.6 uniform buffer objects (UBO) 方便

```
struct UniformBufferObject{  
    glm::mat4 model;  
    glm::mat4 view;  
    glm::mat4 proj;  
};
```

在每一帧更新
模型视图和投
影矩阵

VRDescriptorSetLayoutBinding

- binding：Used in shader
- descriptorType: UNIFORM_BUFFER
- descriptorCount: 可以表示 UBO 的数量
- stageFlags：在着色器哪些阶段被引用
- pImmutableSamplers: 与图像采样有关

在创建管程的时候指定描述符集合的布局

VRPipelineLayoutCreateInfo

· pSetLayouts

Uniform缓冲区和Vertex缓冲区创

建方式相同

在每一帧更新uniform数据

{
 map
 ↓
 copy
 ↓
 unmap

图像

添加一个纹理贴图需要以下几步：

① 创建支持图像的设备内存

② 从图像文件读取像素

③ 创建图像采样器

④ 添加图像采样器描述符，并从纹理中采样颜色信息

△ 创建临时缓冲区

缓冲区必须对于主机可见

△ 纹理图像

VkImageCreateInfo

· imageType = 图像类型

· extent = 指定图像尺寸

· format = the same as format for the texels as the pixels in the buffer

· tiling

◦ VK_IMAGE_TILING_LINEAR = 基于行主序的布局

◦ VK_IMAGE_TILING_OPTIMAL = 基于块的实现来定义布局

· initialLayout

◦ VK_IMAGE_LAYOUT_UNDEFINED = GPU不能使用，第一个渲染将重新设置

◦ VK_IMAGE_LAYOUT_PREINITIALIZED = GPU不能使用，第一个渲染将重新设置

· usage = 同缓冲区创建过程中的 usage 属性语义相同

· sharingMode = 图像会在同一队列簇中使用

· samples = 多重采样相关

为图像分配内存与为缓冲区分配内存类似，使用

vkGetImageMemoryRequirements 替代

vkGetBufferMemoryRequirements 并使用

vkBindImageMemory 替代 vkBindBufferMemory

△ 布局转换

通常主流处理图像变换的方法是用 image memory barrier
管线屏障通常用于访问资源时进行同步

VkImageBarrier



图像视图和平滑器

纹理图像视图

图像不能直接访问而是通过图像视图
他会帮助图像视图来访问纹理图像

采样器

着色器直接从图像中读取像素是可能的，但是它们访问纹理图像的时候并不常见。纹理图像通常使用采样器来访问，应用过滤器和变换来计算最终颜色。

• Compareable、CompareOp 如果开启

比较功能，那么像素首先和值进行比较，并且比较后的值用于且源操作

• mipmapMode、mipLevelBias、

minLod、maxLod

采样器通过 VkSampleCreateInfo 结构配置

- magFilter 和 minFilter 指定像素放大和缩小的插值方式
- addressMode 指定每个轴使单的寻址模式

• RFBAT：原样往返环顶点

• MIRRORBD：反向镜像放界

• CLAMP_TO_EDGE：边缘最近的颜色顶点

• MIRROR_CLAMP_TO_EDGE：相反的边缘

• CLAMP_TO_BORDER：纯色填充

• anisotropyEnable、maxAnisotropy 是否使用各向异性

过滤器

• borderColor 指定采样范围超过图像时返回的颜色。

只能是黑色、白色或透明色，不能指定任意颜色

• unnormalizedCoordinates 指定使用的坐标系统。

可用于访问图像的纹素 [0, texWidth] [0, texHeight]
或 [0..1]

深度缓冲区

深度排序阶段问题通常使用深度缓冲区(depth buffer)来解决。

深度缓冲区是一个额外的附件，用于存储每个顶点的深度信息就像颜色附件存储每个位置的颜色信息一样。每次光栅化生成片段时，深度测试将检查新片段是否比上一个片段更远。如果有，新的片段被丢弃。一个片段将深度测试的值写入深度缓冲区。

深度图像和视图

深度附件是属于图像的，不是着色器附件。所不同的差异在于不会自动创建深度图像。深度图像再次申请需要三种资源：图像、内存和资源视图

format • VK_FORMAT_D32_SFLOAT
• VK_FORMAT_D32_SFLOAT_S8_UINT
• VK_FORMAT_D32_UNORM_S8_UINT

VkImage depthImage

VkDeviceMemory depthImageView

VkImageView depthImageView

findSupportedFormat 根据选择的表单根据限制空间的原则
序原则，检测第一个得到支持的格式

findDepthFormat 选择具有深度组件的格式

hasStencilComponent 选择的深度格式是否包含模板组件

transitionImageLayout 定义布局可以作为初始布局，因为深度图像内容非常紧密

最后添加正确的访问掩码和管道阶段

```
1 if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==  
2     VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {  
3     barrier.srcAccessMask = 0;  
4     barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;  
5  
6     sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;  
7     destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;  
8 } else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL &&  
9           newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {  
10    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;  
11    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;  
12  
13 } else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==  
14             VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {  
15    barrier.srcAccessMask = 0;  
16    barrier.dstAccessMask =  
17        VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |  
18        VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;  
19  
20    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;  
21    destinationStage = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;  
22 } else {  
23     throw std::invalid_argument("unsupported layout transition!");  
24 }
```

渲染过程

首先指定 VkAttachmentDescription, format
应该与深度图像一致

添加第一个(唯一的一个)子通道的附件引用

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

与颜色附件不同的是子通道仅仅使用一个深度(或颜色)附件。对多个缓冲区进行深度测试并没有任何意义。[最后更新](#) `VkRenderPassCreateInfo` 结构体将引用两个附件。

```
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;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

中景缓冲区

修改中景缓冲区的创建，将深度图像绑定到深度附件

```
std::array<VkImageView, 2> attachments =
{swapchainImageViews[1], depthImageView};
```

每个子通道图像的颜色附件不同，但是所有这些都是使用相同的深度图像，同一时间只有一个渲染通道执行

- `depthWriteEnable` 指是否深度测试时新的深度值是否应该被实际写入深度缓冲区。在绘制透明对象的时候将启用

`depthCompareOp` 指是否执行深度或者丢弃阶段的比较操作

`depthBoundsTestEnable`, `minDepthBounds`, `maxDepthBounds` 用于可选择的纹理深度范围测试，基准值选择浮点保留高位在指定深度范围内执行。

`stencilTestEnable`, `front`, `back` 用于配置模板缓冲区的操作

清除标记

因为现在有多个带 `VK_ATTACHMENT_LOAD_OP_CLEAR` 的附件，还需要指明各个清除值。

```
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();
```

深度模板状态

深度附件现在已经准备好了，但是深度测试仍然需要在图形管线开启。

它通过 `VkPipelineDepthStencilStateCreateInfo` 结构体来配置

- `depthTestEnable` 指是否应该将新的深度缓冲区与深度缓冲区比较

Vulkan 和 OpenGL 区别

Vulkan 和 OpenGL 相比，可以更详细的向显卡描述应用程序打算做什么，从而可以获得更好的性能和更小的驱动开销。Vulkan 的设计理念和 DirectX 12 和 Metal 堪称类似，但 Vulkan 作为 OpenGL 的替代者，它的设计之初就是为了跨平台实现，可以在 Windows、Linux 和 Android 开发。甚至在 Mac OS 系统上，Kronos 也提供了 Vulkan 的 SDK，虽然这个 SDK 底层还是 Molten VR 实现的。

Vulkan 的优势

· 异步数据提交

使用 OpenGL 时，如果把数据提交放到另一个独立线程中完成，将会引起冲突。这原因除了资源和进行绘制时都需要改变上下文。由于绘制时要改变上下文，OpenGL 的并行绘制是不可能了。Vulkan 可以并行创建 Command Buffer。Command Buffer 修改后立即由 GPU 驱动去执行。

- 复用 Command Buffer
- 便子模块化

