该文档主要是基于vulkan tutorial 在vs 2015上学习的笔记，只有对vulkan应用开发的基础搭建，对于渲染管道阶段及vertex buffers/framebuffer等部分就没做记录。

## 一、 vulkan 开发环境的搭建

### 1、兼容驱动的更新

a、windows上的兼容驱动更新可在官网直接下载，运行

b、Linux Ubuntu 16.04 Intel (更新Intel 兼容驱动 )

Vulkan tools and drivers —— https://launchpad.net/~canonical-x/+archive/ubuntu/vulkan

（1）Adding this PPA to your system

You can update your system with unsupported packages from this untrusted PPA by adding .

:: sudo add-apt-repository ppa:canonical-x/vulkan

:: sudo apt-get update

(2) vulkan loader has dropped it's demos (can't ship prebuilt shaders) and vkcube is broken, but the driver is now fixed and Willems demos work (some might crash though)

::sudoapt-add-repository ppa:canonical-x/vulkan

:: sudo apt update

:: sudo apt install vulkan-utils mesa-vulkan-drivers

then run "vulkaninfo"

### 2、sdk的配置

sdk 下载(Windows /Linux/Android) <https://vulkan.lunarg.com/>

sdk 安装教程

[https://vulkan-](https://vulkan-tutorial.com/Development_environment#page_Vulkan_SDK)tutorial.com/Development\_environment#page\_Vulkan\_SDK

a、windows下安装可参考教程

b、在ubuntu 16.04下安装

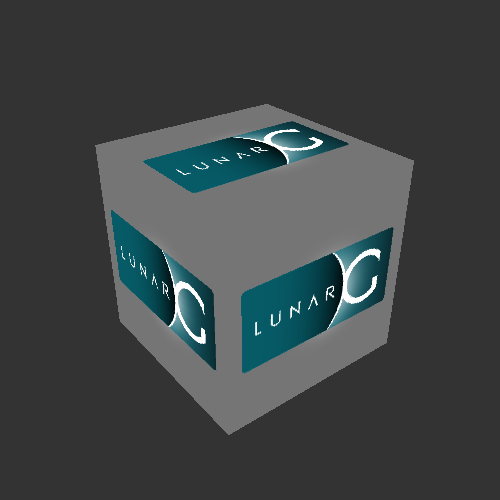
打开终端，执行以下命令 ,随后会自动生成一个VulkanSDK文件，

chmod +x vulkansdk-linux-x86\_64-xxx.run

./vulkansdk-linux-x86\_64-xxx.run

随后执行 sudo apt install libxcb1-dev xorg-dev 安装XCB依赖库

run ./build\_examples.sh 会生成./exmaples/build/cube可执行文件。运行该cube可看到



**notice**: 如果只下载了sdk ,但是显卡不支持vulkan的话，则会在运行cube示例时fail

### 3、依赖库的附加GLFW GLM（glfw用于窗口显示、glm为数学库）

glfw 下载链接 http://www.glfw.org/ glm 下载链接 https://github.com/g-truc/glm/releases

a、windows 在vs 2015上开发可直接参考教程

b、linux的安装

必备工具：gcc的版本需支持c++(4.8 or later) cmake make

（1）glfw的install

进入 glfw目录下 ，运行

cmake .

make

然后执行 sudo make install 安装GLFW

（2）glm 的 install 可直接执行sudo apt install libglm-dev 安装

## 二、vulkan用于应用开发的基础支持

### 1、Instance

(1) 声明变量 : VDeleter<VkInstance> instance {vkDestroyInstance};

(2) 填充两个结构体：VkApplicationInfo和 VkInstanceCreateInfo

**VkApplicationInfo**为可选项，一般提供一些有用的信息可以使驱动对我们的应用进行优化，其一些字段提供了Application和API的版本号，其为固定写法，不可更改。几乎所有VkXXXInfo结构都有sType字段，同样它们的值有一个固定模式：VK\_STRUCTURE\_TYPE XXX\_INFO

**VkInstanceCreateInfo** 提供vulkan的创建信息，其pApplicationInfo字段的值为VkApplicationInfo对象。该结构体主要是为了告诉vulkan driver ，我们将使用global extensions 和validation layers。全局扩展表示其不是只适用于一个特定的设备，因为vulkan是与平台无关的，所以需要一些扩展来与窗口系统（window system）联系，因为我是基于glfw来显示窗口系统的，所以使用glfw内置glfwGetRequiredInstanceExtensions来获得扩展。

为判断vulkan的一些基本信息是否创建成功，可使用Vulkan的函数返回值VkResult来判断，VkResult可取如下值：

­­成功值:

VK\_SUCCESS: 成功执行。

VK\_NOT\_READY: 尚未准备好。

VK\_TIMEOUT: 超时。

VK\_INCOMPLETE: 对于真正需要的结果A来说，返回的结果B小于A的数量。

VK\_SUBOPTIMAL\_KHR: Swap Chain 不能和Surface 完全匹配，但任然能用。

其他…

错误值：

VK\_ERROR\_FORMAT\_NOT\_SUPPORTED 设备不支持的格式。.

VK\_ERROR\_FRAGMENTED\_POOL 由于碎片的原因 Pool分配失败。

VK\_ERROR\_SURFACE\_LOST\_KHR Surface 不再可用。

其他….

(3) 创建instance

if (vkCreateInstance(&createInfo, nullptr, &instance) != VK\_SUCCESS) {

throw std**::**runtime\_error("failed to create instance!");

}

(4) Checking for extension support

Glfw提供来获得extension的内置方法，但是也可以通过vulkan来来检测当前平台所支持的extension:

通过填充vkEnumerateInstanceExtensionProperties结构体可先获得支持扩展的数量，然后通过该数量值获得每个扩展的详细信息。

### 2、Validation Layer

Vulkan的设计理念是：使驱动(driver)的负担最小化。一个明显的表现就是它有限的错误检测，像设置错误的枚举值或者将必须的函数参数传递为空指针(NULL)这类简单的操作都没有明确的处理，Vulkan只是简单的Crash或者产生一些未定义的行为(undefined behavior)。因为Vulkan要求你必须知道你正在做什么。但这并不意味着我们就无从下手了，相反，Vulakn引入了一个非常优雅的方法：Validation layers．Validation layers是一个可选的组件(optional components) ,它连接Vulkan的方法调用。

Validation layers 的基本功能有以下几个:

* Checking the values of parameters against the specification to detect misuse（检查参数是否被勿用）
* Tracking creation and destruction of objects to find resource leaks（检测是否有资源泄露）
* Checking thread safety by tracking the threads that calls originate from（检查线程是否安全）
* Logging every call and its parameters to the standard output（将方法调用的参数打印到标准输出）
* Tracing Vulkan calls for profiling and replaying

vulkan.h 中有定义:

#define VK\_EXT\_DEBUG\_REPORT\_EXTENSION\_NAME "VK\_EXT\_debug\_report" //这是验证所需支持的扩展(extension)

LunarG validation layers 只有在安装了LunarG SDK的PC上才能使用,这里我们不需要所有的layers，LunarG SDK 只需要VK\_LAYER\_LUNARG\_standard\_validation 即可。我们首先获取所有的layers , 然后检查是否支持VK\_LAYER\_LUNARG\_standard\_validation(该值需在程序中明确地给定)。

const std::vector<const char\*> validationLayers = { "VK\_LAYER\_LUNARG\_standard\_validation"};//我们需要的layer

VkResult vkEnumerateInstanceLayerProperties(uint32\_t\* pPropertyCount, VkLayerProperties\* pProperties); //检查是否支持

vkEnumerateInstanceLayerProperties()可以获取所有支持的VkLayerProperties，当pProperties为nullptr时，pPropertyCount将被赋予所被支持的VkLayerProperties数量；pProperties不为nullptr时，pPropertyCount将被赋予所被支持的VkLayerProperties数量的同时，pProperties指向pPropertyCount个VkLayerProperties的数组。，将这些获得的将获得的大小和值分别赋给VkInstanceCreateInfo对象。

在做完以上定义赋值操作后，还需要定义一个回调函数(callback)，将debug信息带回到我们的程序中，然后显示出来。定义一个具有PFN\_vkDebugReportCallbackEXT原型(prototype)的静态(static)成员函数debugCallbak:

static VKAPI\_ATTR VkBool32 VKAPI\_CALL debugCallback(VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objType, uint64\_t obj, size\_t location,

int32\_t code, const char\* layerPrefix, const char\* msg, void\* userData) {

std::cerr << "validation layer: " << msg << std::endl;

return VK\_FALSE;

}

* VkDebugReportFlagsEXT 可取如下值：

VK\_DEBUG\_REPORT\_INFORMATION\_BIT\_EXT

VK\_DEBUG\_REPORT\_WARNING\_BIT\_EXT

VK\_DEBUG\_REPORT\_PERFORMANCE\_WARNING\_BIT\_EXT

VK\_DEBUG\_REPORT\_ERROR\_BIT\_EXT

VK\_DEBUG\_REPORT\_DEBUG\_BIT\_EXT

* VKAPI\_ATTR VkBool32 确保有能被vulkan调用的有效签名
* objType 指定obj的对象的主题消息 eg： if obj is a VkPhysicalDevice then objType would be VK\_DEBUG\_REPORT\_OBJECT\_TYPE\_DEVICE\_EXT
* Msg 参数就是我们需要的debug信息，userData是我们要传给debugCallback的数据

VkDebugReportCallbackEXT callback; //声明

和创建VkInstance一样，需要填充一个VkDebugReportCallbackCreateInfoEXT结构体:

VkDebugReportCallbackCreateInfoEXT createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_DEBUG\_REPORT\_CALLBACK\_CREATE\_INFO\_EXT;

createInfo.flags = VK\_DEBUG\_REPORT\_ERROR\_BIT\_EXT | VK\_DEBUG\_REPORT\_WARNING\_BIT\_EXT; // VkDebugReportFlagsEXT

createInfo.pfnCallback = debugCallback;//pUserData 也可以指定数据，类似userData

然后就是创建debug回调函数，VkDebugReportCallbackCreateInfoEXT为扩展的函数，所以 不能自动加载，只能使用vkGetInstanceProcAddr来查找其地址

VkResult CreateDebugReportCallbackEXT(VkInstance instance, const VkDebugReportCallbackCreateInfoEXT\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkDebugReportCallbackEXT\* pCallback) {

auto func = (PFN\_vkCreateDebugReportCallbackEXT) vkGetInstanceProcAddr(instance, "vkCreateDebugReportCallbackEXT");

if (func != nullptr) {

return func(instance, pCreateInfo, pAllocator, pCallback);

} else {

return VK\_ERROR\_EXTENSION\_NOT\_PRESENT;

}

}

VkDebugReportCallbackEXT 对象需要被vkDestroyDebugReportCallbackEXT清除，现在用VDeleter包裹callbacks,并添加vkDestroyDebugReportCallbackEXT函数:

VDeleter<VkDebugReportCallbackEXT> callback{instance, DestroyDebugReportCallbackEXT};//声明

Destroycallback方法的创建

void DestroyDebugReportCallbackEXT(VkInstance instance, VkDebugReportCallbackEXT callback, const VkAllocationCallbacks\* pAllocator) {

auto func = (PFN\_vkDestroyDebugReportCallbackEXT) vkGetInstanceProcAddr(instance, "vkDestroyDebugReportCallbackEXT");

if (func != nullptr) {

func(instance, callback, pAllocator);

}

}

Note: 在*Vulkan SDK* 的目录下有个*Config*文件夹，这个文件下有个*vk\_layer\_settings.txt*文件，是你对**Validation layers**进行配置的文件，你可以对这个文件进行配置。

### 3、[Physical Device and Queue Family](http://blog.csdn.net/lbknxy/article/details/52430773)

使用vkInstance初始化vulkan的API后，需选择一个具有我们所需特性的显卡。

VkPhysicalDevice physicalDevice=Vk\_NULL\_HANDLE; //声明 VkPhysicalDevice 将同Instance一同销毁，这里不必使用VDeleter。

首先我们要考虑两个问题：

* 如何获取Physical Devices。
* 如何从Physical Devices 中挑选我们想要的那个Physical Device.

1. 获取Physical Device

Vulkan 提供了枚举(enumerate)出当前平台(platform)可用的所有显卡(graphics card or Physical Device)的简便方法：vkEnumeratePhysicalDevices 和寻找validation layer类似。

1. 从Physical Devices 中挑选我们想要的那个Physical Device

<1> 可通过查询相应的细节来评估是否合适，使用VkPhysicalDeviceProperties来查询name, type 和supported Vulkan version

VkPhysicalDeviceProperties deviceProperties;

vkGetPhysicalDeviceProperties(physicalDevice, &deviceProperties);

其VkPhysicalDeviceType字段表示显卡的类型。

typedef enum VkPhysicalDeviceType {

VK\_PHYSICAL\_DEVICE\_TYPE\_OTHER = 0, //other

VK\_PHYSICAL\_DEVICE\_TYPE\_INTEGRATED\_GPU = 1, //集成

VK\_PHYSICAL\_DEVICE\_TYPE\_DISCRETE\_GPU = 2, //独立

VK\_PHYSICAL\_DEVICE\_TYPE\_VIRTUAL\_GPU = 3, //虚拟

VK\_PHYSICAL\_DEVICE\_TYPE\_CPU = 4, //running on cpu

} VkPhysicalDeviceType

<2>VkPhysicalDeviceFeatures(特性支持)

VkPhysicalDeviceFeatures deviceFeatures;

vkGetPhysicalDeviceFeatures(physicalDevice, &deviceFeatures);

<3>VkQueueFamilyProperties(队列家族属性)

在Vulkan中，队列有很多种类(感觉family 译成种类好理解)，每种队列只支持Vulkan命令的一个子集，比如：一种队列只具有处理计算的命令(processing of compute commands) 或者只具有内存传递的命令(memory transfer related commands)。我们将从Physical Device里枚举出它所拥有的所有队列(VkQueue)的种类，并从中抽取出我们感兴的那种队列，或者说我们要通过判断Physical Device 是否支持我们感兴趣的队列来对Physical Device 进行筛选。

通过调用vkGetPhysicalDeviceQueueFamilyProperties方法来获得VkQueueFamilyProperties，并筛选出我们需要的队列。

挑选需求总结：deviceType显卡类型为VK\_PHYSICAL\_DEVICE\_TYPE\_DISCRETE\_GPU 类型的显卡，并且支持geometry shaders，VkPhysicalDeviceFeatures. geometryShader为Vk\_TRUE。想要队列支持图形处理命令，即VkQueueFamilyProperties . queueFlags 为VK\_QUEUE\_GRAPHICS\_BIT。

### 4、Logical Device

只有Physical Device 还不行，我们还需要创建Logical Device 来与它相联。Logical Device的创建和VkInstance的创建过程差不多，需要明确我们所需的特性(features)、extensions、Validation layers 、queue等。

VDeleter<VkDevice> device{vkDestroyDevice};//声明 device使用vkDestroyDevice来销毁

(1)其特性为默认值，VkPhysicalDeviceFeatures deviceFeatures = {};

(2)VkDeviceQueueCreateInfo结构体 //设备队列创建描述信息的填充

typedef struct VkDeviceQueueCreateInfo {

VkStructureType sType;

const void\* pNext;

VkDeviceQueueCreateFlags flags;

uint32\_t queueFamilyIndex;//索引值

uint32\_t queueCount;//队列数量

const float\* pQueuePriorities; //优先级 0-1

} VkDeviceQueueCreateInfo;

(3)创建Logical Device

VkDeviceCreateInfo设备创建的描述信息的填充，该结构除了对队列(queue)和特性(features)支持的限定外，还有对Validation layers 和 Extensions的限定

typedef struct VkDeviceCreateInfo {

VkStructureType sType;

const void\* pNext;

VkDeviceCreateFlags flags;//(future use)

uint32\_t queueCreateInfoCount;

const VkDeviceQueueCreateInfo\* pQueueCreateInfos;//队列创建描述信息

uint32\_t enabledLayerCount;

const char\* const\* ppEnabledLayerNames;

uint32\_t enabledExtensionCount;

const char\* const\* ppEnabledExtensionNames;

const VkPhysicalDeviceFeatures\* pEnabledFeatures;//设备属性

} VkDeviceCreateInfo;

enableValidationLayers和validationLayers直接取自创建VkInstances时已有的定义，在VkDeviceCreateInfo 里定义的队列(queue 类型为VkQueue)将会随着logical device 一同被创建。

VkQueue graphicsQueue;//图像处理队列

vkGetDeviceQueue(device, indices.graphicsFamily, 0, &graphicsQueue);//获得设备队列

* device : logical device.
* indices.graphicsFamily : 队列种类。
* queueIndex : 这里是 0 ，因为我们只创建了一个队列，所以这里索引为0.
* VkQueue \* : &graphicsQueue。 //handle

### 5、Window Surface

因为Vulkan是平台(platform)无关的，它不能直接与平台窗体系统(window system)进行通信，为了连接Vulkan和窗体系统,使得被渲染后的结果显示到屏幕上，我们需要使用WSI扩展(Window System Integration extensions),在这个章节我们将使用VK\_KHR\_surface，它提供的VkSurfaceKHR 是对surface的一个抽象，使得我们能够将渲染后的结果放到VkSurfaceKHR上。

VK\_KHR\_surface是一个Instance 级别的扩展，在创建Instance时已经通过glfwGetRequiredInstanceExtensions允许了这个扩展。

事实上，window surface的创建应该在Instance创建之后就应该完成，因为它会影响Physical Device的选取，如果只是需要off-screen rendering，那么window surface对于Vulkan来说只是一个可选的扩展。

**a、创建window surface**

VDeleter<VkSurfaceKHR> surface{instance, vkDestroySurfaceKHR};//声明 VkSurfaceKHR需要vkDestroySurfaceKHR来销毁

GLFW提供了glfwCreateWindowSurface方法，它自动为我们解决平台的差异性，所以其创建就变得很简单。

void createSurface() {

if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK\_SUCCESS) {

throw std::runtime\_error("failed to create window surface!");

}

}

**b、确定显卡支持WSI(请求显示支持)**

尽管Vulkan的实现可能支持WSI ，但并不代表你平台上的所有显卡也支持，它是指Physical Device 中存在一种将images提交到Surface上的命令队列。所以需要从显卡里寻找一种具有将渲染结果提交(presenting)到surface上的命令的队列(queue family)。

VkBool32 presentSupport = false;

vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport); // 检测队列是否支持将渲染结果提交(presenting)到surface上

// VK\_QUEUE\_GRAPHICS\_BIT 表示需queuefamily支持图像绘制 queueFamily.queueFlags表示需支持显示

*Notice*: 在创建Logical Device时已经创建了一个队列用于支持图形处理的graphicsQueue，现在又多了一个用于将渲染结果提交(presenting)到surface上的队列。

### 6、Swap Chain

Swap Chain为我们提供要渲染的图片，然后渲染结的果可以显到屏幕上，必须被Vulkan显示的创建。从本质上讲，Swap Chain就是一个图片的队列(a queue of images),这里的图片等着被显示到屏幕上。我们的应用将会获得一个图片，然后绘画它，之后将它提交到队列中去。Swap Chain 通常的作用是通过屏幕刷新率(refresh rate of the screen)来同步控制图片的显示。

**a、检查显卡是否支持 Swap chain**

显卡有各种理由阻止你将images直接提交给屏幕，例如系统只是个服务器(Server)，不支持显示。其次，图片显示和窗口系统(window system)以及和窗口相连的window surface密切相关。所以Swap Chain并不属于核心Vulkan 功能。这就要求必须支持VK\_KHR\_swapchain扩展

检查显卡是否支持Swap chain，首先获取显卡的所有扩展:

VkResult vkEnumerateDeviceExtensionProperties(

VkPhysicalDevice physicalDevice,

const char\* pLayerName,

uint32\_t\* pPropertyCount,

VkExtensionProperties\* pProperties);

pLayerName是一个过滤选项，它是一个Validation layer的名字，表示只枚举这个Validation layer所对应的所有扩展，如果pLayerName为nullptr (NULL)，表示要获取对应physicalDevice下的所有扩展。

Vulkan.h有如下定义：

#define VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME "VK\_KHR\_swapchain"

physicalDevice 必须支持此扩展，才能创建Swap Chain

定义该对象 const std::vector<const char\*> deviceExtensions = {

VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME

};

然后通过查找看在显卡确定的情况下是否有想要的swapchain的扩展

**b、查询Swap Chain的支持细节**

只是检查显卡是否支持Swap Chain还不够，因为Swap Chain还可能和我们的window surface不兼容。创建一个Swap Chain还需要一系列配置工作，比创建Instance和Logical Device复杂多了。

所以我们还要检查以下三种属性:

* Surface 的性能(Capabilities)(比如 : min/max number of images in swap chain, min/max width and height of images)。
* Surface 的格式(formats)(比如 : pixel format, color space)。
* 可用的显示模式(present mode)。

定义一个结构体来包含这三个属性对象

struct SwapChainSupportDetails {

VkSurfaceCapabilitiesKHR capabilities;

std::vector<VkSurfaceFormatKHR> formats;// VkSurfaceFormatKHR包括VkFormat、VkColorSpaceKHR

std::vector<VkPresentModeKHR> presentModes;

};

这三个字段的内容分别由如下方法获得:

vkGetPhysicalDeviceSurfaceCapabilitiesKHR

vkGetPhysicalDeviceSurfaceFormatsKHR

vkGetPhysicalDeviceSurfacePresentModesKHR

只要有一个format和presentMode我们就认为此显卡支持Swap Chain 并且兼容surface。

**c、为Swap Chain 选择最佳设置**

为了寻找最佳的Swap Chain设置，我们决定从以下三个方面入手:

* Surface (格式)format (如:color depth)
* 显示模式(Presentation mode)(如:渲染后的图片“交换”到显示器的时机).
* 交换的大小(Swap extent)(如:图片在swap chain里的分辨率)

*Format:* 所有被支持的Format都已存在 swapChainSupport.fromats中，每一个Format是一个VkSurfaceFormatKHR结构，包含format 和 colorSpace。c

*我们的需求:*   
format 为:VK\_FORMAT\_B8G8R8A8\_UNORM ，因为这种颜色比较通用;   
colorSpace 为:VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR，即支持SRGB颜色。

*Presentation mode：*

VkpresentModeKHR 有:

VK\_PRESENT\_MODE\_IMMEDIATE\_KHR = 0, //单 ，立即显示，可能出现撕裂

VK\_PRESENT\_MODE\_MAILBOX\_KHR = 1, // 三缓冲

VK\_PRESENT\_MODE\_FIFO\_KHR = 2, //双

VK\_PRESENT\_MODE\_FIFO\_RELAXED\_KHR = 3, //类似 FIFO

…

因为VK\_PRESENT\_MODE\_FIFO\_KHR 一定是可以得到的，又可以避免撕裂，所以presentMode 我们选择VK\_PRESENT\_MODE\_FIFO\_KHR（在最差的选择下，一般选择三缓冲）.

*Swap extent：*

extent 是Swap Chain中image的分辨率(resolution) ,通常它与window的尺寸一样，vulkan让我们通过设置currentExtent设置width和height来匹配window的分辨率。但是有些window Manager会将currentExtent设置为uint32\_t的最大值，来表示允许我们设置不同的值，这个时候我们可以从minImageExtent和maxImageExtent中选择最匹配window的尺寸值。

具体实现：

VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {

if (capabilities.currentExtent.width != std::numeric\_limits<uint32\_t>::max()) {

return capabilities.currentExtent;

} else {

VkExtent2D actualExtent = {WIDTH, HEIGHT}; // window 的尺寸

actualExtent.width = std::max(capabilities.minImageExtent.width, std::min(capabilities.maxImageExtent.width, actualExtent.width));

actualExtent.height = std::max(capabilities.minImageExtent.height, std::min(capabilities.maxImageExtent.height, actualExtent.height));

return actualExtent;

}}

解释：

1.当currentExtent.width/height 任何一个值不是uint32\_t的最大值时，直接选择Capabilities.currentExtent, currentExtent最符合window尺寸。

2.否则，从Window 尺寸和支持的最大尺寸(maxImageExtent)取最小的，因为尺寸不能超过window，也不能超过capabilities支持的最大尺寸。最后为了得到最好的效果，取结果值和minImageExtent的最大值。

**d、创建Swap Chain**

作为创建Vulkan对象的惯例，在创建Swap Chain 之前我们需要填充一个大的结构体

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; // image的层次，除非创建3D应用，否则这个值将为1

createInfo.imageUsage = VK\_IMAGE\_USAGE\_COLOR\_ATTACHMENT\_BIT;//将image作为颜色附件使用

// imageUsage：

如果你想先渲染一个单独的图片然后再进行处理，那就应该使用VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT并使用内存转换操作将渲染好的image 转换到SwapChain里。

VK\_IMAGE\_USAGE\_COLOR\_ATTACHMENT\_BIT 表示image可以用作创建VkImageView，在VkFrameBuffer中适合使用color 或者 reslove attachment.

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.queueFamilyIndexCount = 0; // Optional createInfo.pQueueFamilyIndices = nullptr; // Optional

}

如果grapics queue 和 present queue不相同，就会出现这多种队列访问image的情况:

我们在grapics queue 中绘画image,然后将它提交到presention queue 去等待显示

imageSharingMode取值：

VK\_SHARING\_MODE\_EXCLUSIVE : image 一段时间内只能属于一种队列，所有权的转换必须明确声明，这个选项可以提供较好的性能。

VK\_SHARING\_MODE\_CONCURRENT : image 可以跨多种队列使用，所有权的转换不必明确声明。

现在我们已经获得了所有创建Swap Chain的必要信息，接下来就可以直接创建交换链了。

VkSwapchainKHR swapChain; // 声明

if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK\_SUCCESS) {

throw std::runtime\_error("failed to create swap chain!");

}

std::vector<VkImage> swapChainImages swapChain里的image随swapChain一起创建、一起销毁。

获取swapChain 里所有的images:

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

swapChainImages.resize(imageCount);

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

Image 的清理工作将随着SwapChain的清理而被清理。

7、Image View

为了使用VkImage,不管是在Swap Chain 还是在Pipeline 中，我们都必须创建VkImageView,就如同它的字面意思一样,imageView是image的一个 view.他描述了我们如何访问image、访问image的哪一部分等.

创建imageView将其作为颜色附件

std::vector<VDeleter<VkImageView>> swapChainImageViews;//声明

和以往一样，需填充VkImageViewCreateInfo结构体

typedef struct VkImageViewCreateInfo {

VkStructureType sType

const void\* pNext;

VkImageViewCreateFlags flags;

VkImage image; //每一个swapchain 对象

VkImageViewType viewType; //一般为VK\_IMAGE\_VIEW\_TYPE\_2D

VkFormat format;

VkComponentMapping components;

VkImageSubresourceRange subresourceRange; //描述image的使用目的和要被访问的部分

} VkImageViewCreateInfo;

SubsourceRange下的aspectMask属性为VK\_IMAGE\_ASPECT\_COLOR\_BIT，表示image 将被用作color targets ,不使用mipmapping 和 multiplayers。如果要创建3D应用，则使用multiplayers

创建imageView的语句

if (vkCreateImageView(device, &createInfo, nullptr,&swapChainImageViews[i]) != VK\_SUCCESS) {

throw std::runtime\_error("failed to create image views!");

}

### 7、Swap chain recreation

当window surface（one reason:窗体大小的改变） 改变后，swap chain 就不再适用。所以我们需要捕获这些事件，并recreate the swap chain.

**a、Recreating the swap chain**

新建一个recreateSwapChain 方法，调用createSwapChain 及其他一些依赖 swap chain及window size 的函数。

在recreateSwapChain方法里首先需要调用的是vkDeviceWaitIdle，因为我们可能获取不到一些正在使用的字眼，所以等待他们free后再进行recreate操作。

首先需要做的是重新创建swapchain本身，然后因为image views是直接基于swapchain image,所以需要recreate;render pass因为直接取决于swap chain 的format，所以需要 recreate;因为Viewport and scissor rectangle size在pipeline创建时被指定（eg:viewport是swapChainExtent的宽高度），所以pipeline需要重新被创建；framebuffer和commandbuffer也直接取决于swapchain ,所以需要recreate。

**b、Window resizing**

当窗口被resize时，就需要recreate swap chain recreation 。

**c、Suboptimal or out-of-date swap chain**

当swap chain 在显示时不再兼容，就需要recreation。

[vkAcquireNextImageKHR](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkAcquireNextImageKHR.html)和[vkQueuePresentKHR](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkQueuePresentKHR.html)方法可通过返回一些特定的值来检查swap chain 是否需要更新。

* VK\_ERROR\_OUT\_OF\_DATE\_KHR: swap chain不能兼容当前的surface，并且也不能再用于rendering。
* VK\_SUBOPTIMAL\_KHR: swap chain仍然可以用于显示surface，但是surface的属性不能很好地匹配精确。eg:platform可能简单的调整了image来适应window。