C++的RAII原理来创建一个包裹类

auto 自动类型推断和返回值占位

template <typename T> //T代表一个类型

在模板定义语法中关键字class 与 typename的作用完全一样

**VkDestroyInstance--**Destroy an instance of Vulkan

每一个Vulkan对象不再使用后，都需要用一个清理函数来销毁，就像malloc 分配的内存要用free 销毁一样。手动来销毁每一个对象是一个非常繁杂且容易出错的任务。我们利用C++的RAII原理来创建一个包裹类，当对象离开有效作用域后自动为我们销毁对象。比如，关闭了应用，将自动使用vkDestroyInstance将Instance销毁，销毁Instance对象的方法是vkDestroyInstance

通过调用

void vkDestroyInstance(

VkInstance instance,

const VkAllocationCallbacks\* pAllocator);

来销毁 instance

：：instance 是销毁的instance的句柄

:：pAllocator 是可选的，允许对特定的内存分配器指定回调

**create an instance**

在创建instance时可以给定VkApplicationInfo和 VkInstanceCreateInfo 两个结构体信息

VkApplicationInfo 是一个可选项，提供一些有用的信息可以使驱动对我们的应用进行优化 url:https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkApplicationInfo.html

The VkApplicationInfo structure is defined as:

typedef struct VkApplicationInfo {

VkStructureType sType;

const void\* pNext;

const char\* pApplicationName;

uint32\_t applicationVersion;

const char\* pEngineName;

uint32\_t engineVersion;

uint32\_t apiVersion;

} VkApplicationInfo;

VkApplicationInfo appInfo = {};

appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;

appInfo.pApplicationName = "Hello Triangle"; // 自定义

appInfo.applicationVersion = 1 //自定义

appInfo.pEngineName = "No Engine"; //自定义

appInfo.engineVersion =1; //自定义

appInfo.apiVersion = VK\_API\_VERSION\_1\_0;

sType is the type of this structure.

pNext is NULL or a pointer to an extension-specific structure.

pApplicationName is a pointer to a null-terminated UTF-8 string containing the name of the application.

applicationVersion is an unsigned integer variable containing the developer-supplied version number of the application.

pEngineName is a pointer to a null-terminated UTF-8 string containing the name of the engine (if any) used to create the application.

engineVersion is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application.

apiVersion is the version of the Vulkan API against which the application expects to run

基本上所有的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。 apiVersion 是Vulkan的版本号，在vulkan.h中有如下定义

// Vulkan 1.0 version number

#define VK\_API\_VERSION\_1\_0 VK\_MAKE\_VERSION(1, 0, 0)

VkInstanceCreateInfo

在Vulkan中，几乎所有需要创建的 XXX对象，都要填充类似VkXXXCreateInfo结构体。VkInstanceCreateInfo 告诉Vulkan驱动，我们将使用的全局扩展(global extensions )和验证层(global validation layers )

VkInstanceCreateInfo createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;

createInfo.pApplicationInfo = &appInfo;//appInfo为VkApplicationInfo

因为Vulkan是跨平台的，所以它需要类似于“插件”一样的扩展来和窗体系统(window system)进行联系。

glfw 内置拥有获取扩展的方法:

unsigned int glfwExtensionCount = 0;

const char\*\* glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;

createInfo.ppEnabledExtensionNames = glfwExtensions;

Checking for extension support

虽然glfw提供了获取extension的内置方法，我们同样可以使用vulkan核心API中的方法来检测当前平台所支持的extension:

typedef enum VkResult {

VK\_SUCCESS = 0,

VK\_NOT\_READY = 1,

VK\_TIMEOUT = 2,

VK\_EVENT\_SET = 3,

VK\_EVENT\_RESET = 4,

VK\_INCOMPLETE = 5,

VK\_ERROR\_OUT\_OF\_HOST\_MEMORY = -1,

VK\_ERROR\_FEATURE\_NOT\_PRESENT = -8,

VK\_SUBOPTIMAL\_KHR = 1000001003,

VK\_ERROR\_OUT\_OF\_DATE\_KHR = -1000001004,

…

…

} VkResult;

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….

VkResult vkEnumerateInstanceExtensionProperties(

const char\* pLayerName,

uint32\_t\* pPropertyCount,

VkExtensionProperties\* pProperties);

它接收三个参数：

* pLayerName： 是一个Validation layer,用来过滤extension ,这里先置为nullptr。
* pPropertyCount： 返回extension 的数量。
* pProperties : extension的具体细节，如果这个值为nullptr,则，函数只返回extension的数量，即pPropertyCount，否则返回pPropertyCount个VkExtensionProperties。

uint32\_t extensionCount = 0;

vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);

如上可获得扩展的数量，也用这个数量创建容器:

std::vector<VkExtensionProperties> extensions(extensionCount)

最后获取平台上支持的extension:

vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());

打印出来看看每个extension具体是什么 ：

std::cout << "available extensions:" << std::endl;

for (const auto& extension : extensions) {

std::cout << "\t" << extension.extensionName << std::endl;

# Validation layers

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。

基本功能：

 检查参数是否被勿用。

 追踪对象的创建与销毁，检测是否与有资源泄露。

 追踪线程(thread)调用的源头，检测线程是否安全。

 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

//我们需要的layer

const std::vector<const char\*> validationLayers = {

"VK\_LAYER\_LUNARG\_standard\_validation"

};

//检查是否支持

VkResult vkEnumerateInstanceLayerProperties(uint32\_t\* pPropertyCount, VkLayerProperties\* pProperties);

Notice :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。

验证错误 需使用回调函数（callback） ，将debug信息带回到我们的程序中，然后显示出来。***回调函数需要明确地创建和销毁***

·具有PFN\_vkDebugReportCallbackEXT原型(prototype)的静态(static)成员函数debugCallback:

static VKAPI\_ATTR VkBool32 VKAPI\_CALL debugCallback(

VkDebugReportFlagsEXT flags,

VkDebugReportObjectTypeEXT objType,//消息主题的类型

uint64\_t obj, // [VkPhysicalDevice](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkPhysicalDevice.html)

size\_t location,

int32\_t code,

const char\* layerPrefix,

const char\* msg,

void\* userData) //传递给回调函数的数据

{

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

return VK\_FALSE;

}

第一个参数可选类型,是由如下bit flags 组合而成

* 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

objType 一般为 VK\_DEBUG\_REPORT\_OBJECT\_TYPE\_DEVICE\_EXT

该debugCallback是被VkDebugReportCallbackCreateInfoEXT 的pfnCallback属性调用的。

# Physical Device and Queue

选择一个支持我们所需特性的显卡（graphics card）

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：

VkResult vkEnumeratePhysicalDevices(

VkInstance instance,

uint32\_t\* pPhysicalDeviceCount,

VkPhysicalDevice\* pPhysicalDevices

);

使用 vkEnumeratePhysicalDevices来获取相应的physical devices

1. 从Physical Devices 中挑选我们想要的那个Physical Device
2. VkPhysicalDeviceProperties (显卡的属性)

typedef struct VkPhysicalDeviceProperties {

uint32\_t apiVersion;

uint32\_t driverVersion;

uint32\_t vendorID;

uint32\_t deviceID;

VkPhysicalDeviceType deviceType;

char deviceName[VK\_MAX\_PHYSICAL\_DEVICE\_NAME\_SIZE];

uint8\_t pipelineCacheUUID[VK\_UUID\_SIZE];

VkPhysicalDeviceLimits limits;

VkPhysicalDeviceSparseProperties sparseProperties;

} VkPhysicalDeviceProperties;

目前只需要用到 VkPhysicalDeviceType 字段

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

用 vkGetPhysicalDeviceProperties获取显卡的类型

1. VkPhysicalDeviceFeatures ( 特性支持 ):

typedef struct VkPhysicalDeviceFeatures {

VkBool32 robustBufferAccess;

VkBool32 fullDrawIndexUint32;

VkBool32 imageCubeArray;

VkBool32 independentBlend;

VkBool32 geometryShader;

VkBool32 tessellationShader;

VkBool32 sampleRateShading;

VkBool32 dualSrcBlend;

VkBool32 logicOp;

VkBool32 multiDrawIndirect;

VkBool32 drawIndirectFirstInstance;

VkBool32 depthClamp;

...

...

} VkPhysicalDeviceFeatures;

获取vkGetPhysicalDeviceFeatures的方法: vkGetPhysicalDeviceFeatures

1. 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)。

typedef struct VkQueueFamilyProperties {

VkQueueFlags queueFlags; // or VkQueueFlagBits

uint32\_t queueCount;

uint32\_t timestampValidBits;

VkExtent3D minImageTransferGranularity;

} VkQueueFamilyProperties;

通过考虑queueFlags和queueCount这两个字段来选择所需要的队列，get方法与之前的类似。

模拟需求:

1. 显卡类型可根据电脑显卡配置来定（双显卡随意，若只有集成显卡只能选择VK\_PHYSICAL\_DEVICE\_TYPE\_INTEGRATED\_GPU）。   
   2. 特性支持geometry shaders。   
   3. 队列支持图形处理命令。

# Logical Device

只有Physical Device还不行，需创建Logical Device来与之相联，其创建过程与VkInstace的创建类似，需要明确我们所需的特性(features)、extensions、Validation layers 、queue等。

必须保证其声明时位于VkInstance后面，因为需保证在销毁VkInstance之前就得把Logical Device对象销毁，

声明 device

VDeleter<VkDevice> device{vkDestroyDevice};

将 VkPhysicalDeviceFeatures deviceFeatures = {}; features特性采用默认值VK\_FALSE。

typedef struct VkDeviceQueueCreateInfo {

VkStructureType sType;

const void\* pNext;

VkDeviceQueueCreateFlags flags;

uint32\_t queueFamilyIndex;

uint32\_t queueCount;

const float\* pQueuePriorities;

} VkDeviceQueueCreateInfo;

后面3个参数表示，创建queueCount个queueFamilyIndex类型的队列，每个队列的优先级用pQueuePriorities数组表示。优先级的值为0.0~1.0 , 值越大优先级越高。

说明： 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，例如一个很重要的extension : VK\_KHR\_swapchain 支持。为简单化，正如在创建VkInstance时定义的那样，我们直接将那时定义的layers 和 extensions应用到这里。

在VkDeviceCreateInfo 里定义的队列(queue 类型为VkQueue)将会随着logical device 一同被创建。获得该队列的句柄

VkQueue graphicsQueue;//声明

vkGetDeviceQueue(device, indices.graphicsFamily, 0, &graphicsQueue);

·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。

# 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，我们在创建Instance时已经通过glfwGetRequiredInstanceExtensions允许了这个扩展，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。

(1) 创建VkSurfaceKHR

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

void createSurface() {

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

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

}

}

Glfw提供glfwCreateWindowSurface方法解决了平台的差异。

(2) 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。

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)。为检测队列是否支持将渲染结果提交presenting)到surface上，使用如下函数来检测

kResult vkGetPhysicalDeviceSurfaceSupportKHR(

VkPhysicalDevice physicalDevice,

uint32\_t queueFamilyIndex,

VkSurfaceKHR surface,

VkBool32\* pSupported);

# SwapChain

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。

(1) 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.

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 下有如下定义:

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

physicalDevice 必须支持此扩展，才能创建Swap Chain.

(2) 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:

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

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。 现在我们的Physical Device 越发强大。

(3) 为Swap Chain 选择最佳设置

如果swapChainAdequate的值为true.那么我们有理由肯定Swap Chain已经被支持了。而且Swap Chain 的所有特性支持也都被写入swapChainSupport变量中了。但是针对不同的优化有着不同的设置选项，为了寻找最佳的Swap Chain设置，我们可以从以下三个方面入手:

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

·***Surface 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。

通常*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.

***.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。

***·Creating the swap chain***