Instance & Devices & Queue &SwapChain

https://github.com/GameTechDev/IntroductionToVulkan/tree/master/Project/Tutorials/02

以上面的代码为例讲解:

Instance

CreateInstance():

```
vkCreateInstance

typedef struct VkInstanceCreateInfo {
    VkStructureType             sType;  // 一般为方法对应的类型
    const void*                 pNext; // 一般为 null 就好了
    VkInstanceCreateFlags       flags;  // 留着以后用的，设为 0 就好了
    const VkApplicationInfo*    pApplicationInfo; // 对应新的一个结构体 VkApplicationInfo
    uint32_t                    enabledLayerCount; // layer 和 extension 用于调试和拓展
    const char* const*          ppEnabledLayerNames;
    uint32_t                    enabledExtensionCount;
    const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

Vulkan中没有OpenGL中的global state，每个应用程序的所偶遇状态都存储在VkInstance中

Devices & Queue

`vkCreateInstance()` → `vkEnumeratePhysicalDevices()` → `vkCreateDevice()`

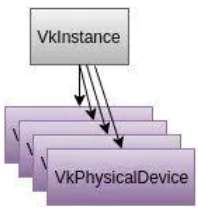
`Device` 具体指的是逻辑上的设备，可以说是对物理设备的一个逻辑上的封装，而物理设备就是 `VkPhysicalDevice` 对象。

`VkPhysicalDevice` 定义为一个 Vulkan 所支持的物理设备，比如显卡。一个受支持物理设备提供了它的基本信息（生产商，设备标识符等），它所支持的特性 extension 和它的限制 limits。同时，它也提供了用于提交命令的队列类型 queue families 以及一系列属于一个或多个了类型的队列 queues。

此之外，它也提供了设备内存 Device Memory，以内存堆 Memory Heaps 的方式呈现，表示某一块内存区域。每一块内存堆都有自己的内存类型，有些内存堆物理连接到 GPU 上，CPU 不可直接映射，称为专用 GPU 内存；有些内存堆为系统内存，GPU 可以通过 PCI-E 去访问并在 GPU 本地做读缓存，称为共享 GPU 内存。当然，就像计算机存储层级一样，底层的驱动程序允许将不同的内存类型（例如 RAM 和 Cache）抽象成一种对外可见的内存类型。在 Vulkan 中，存储类型可以是 `DEVICE_LOCAL`，`HOST_COHERENT`，`HOST_VISIBLE` 和 `HOST_CACHED`，其抽象方法依照硬件和驱动而定，上层应用不需要多加考虑。

将一个物理设备以及需要的特性和需要的队列类型打包在一起则形成了一个逻辑设备，逻辑设备成为了进一步操作的必需品

在某些情况下，可能会具有多个物理设备，如下图所示，因此要先枚举一下所有的物理设备：



CreateDevice()

```
uint32_t num_devices = 0;
    if( (vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, nullptr ) != VK_SUCCESS) ||
        (num_devices == 0) ) {
      std::cout << "Error occurred during physical devices enumeration!" << std::endl;
      return false;
    }

    std::vector<VkPhysicalDevice> physical_devices( num_devices );
    if(vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, physical_devices.data() ) != VK_SUCCESS ) {
      std::cout << "Error occurred during physical devices enumeration!" << std::endl;
      return false;
    }
```

Vulkan API 调用的一个 **固定套路** 了，调用两次来获得不同的数据,当再一次调用 `vkEnumeratePhysicalDevices` 函数时，第三个参数不为 null，而是相应的 `VkPhysicalDevice` 容器，那么 gpus 会填充 `gpu_size` 个的 `VkPhysicalDevice` 对象。

有了 `VkPhysicalDevice` 对象之后，可以查询 `VkPhysicalDevice` 上的一些属性，以下函数都可以查询相关信息：

- vkGetPhysicalDeviceQueueFamilyProperties
- vkGetPhysicalDeviceMemoryProperties
- vkGetPhysicalDeviceProperties
- vkGetPhysicalDeviceImageFormatProperties
- vkGetPhysicalDeviceFormatProperties

在接下来的CheckPhysicalDeviceProperties()函数中,调用vkGetPhysicalDeviceQueueFamilyProperties去查询`VkPhysicalDevice`中的Queue.

```
uint32_t queue_families_count = 0;
    vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count, nullptr );
    if( queue_families_count == 0 ) {
      std::cout << "Physical device " << physical_device << " doesn't have any queue families!" << std::endl;
      return false;
    }

    std::vector<VkQueueFamilyProperties>  queue_family_properties( queue_families_count );
    std::vector<VkBool32>                 queue_present_support( queue_families_count );

    vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count, queue_family_properties.data() );
```

queueFlags:

```
typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,         // 图像相关
    VK_QUEUE_COMPUTE_BIT = 0x00000002,          // 计算相关
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
    VK_QUEUE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkQueueFlagBits;
typedef VkFlags VkQueueFlags;
```

物理设备可能会有多个 `Queue`，不同的 `Queue` 对应不同的特性。`Command-buffer` 是提交到了 `Queue` ，`Queue` 再提交给 `Device` 去执行。`Queue` 可以看成是应用程序和物理设备沟通的桥梁，我们在 `Queue` 上提交命令，然后再交由 GPU 去执行。

NVIDIA exposes 16 Queues

SwapChain

创建 SwapChain 有的基本步骤如下：

1. 创建 VkSurfaceKHR 组件，解决平台之间的差异。在 Android 上就是根据 Native 中的 `ANativeWindow` 去创建。
2. 从某个物理设备 (也就是 GPU，因为可能存在多个物理设备) 所有的 `Queue` 中找到那个即支持图形（graphics）又支持显示（present）的 `Queue` 的索引（index）。
3. 如果没有 `Queue` 同时支持两者，那么就找到两个各自支持的，分别是：
    a. present queue（用于展示的 Queue）
    b. graphics queue（用于图形的 Queue）
    c. 有了这两个索引之后，要得到索引所对应的 Queue 。
4. 如果连各自支持的都没有，那 SwapChain 也建立不了了，就退出吧。
5. 从某个物理设备 (也就是 GPU，因为可能存在多个物理设备) 找到所有支持 VKSurfaceKHR 的色彩空间格式（VKFormat），并选取第一个。
6. 根据 VKSurfaceKHR 的能力和呈现模式，以及相关参数设定去创建 SwapChain 。
    a. Surface 能力对应 SurfaceCapabilitiesKHR
    b. Surface 呈现模式对应于 SurfacePresenttModesKHR
    c. Surface 旋转的设定，对应于 SurfaceTransformFlagBitsKHR
    d. Surface 透明度合成的设定，对应于 CompositeAlphaFlagBitsKHT
    e. Surface 相关的参数设定有很多，但是对于有些不常用的设定基本可以选择固定值了
    f. 相关参数的设定都明确之后，就创建 SwapChain
7. 创建 SwapChain 之后，获取 SwapChain 支持的 Image 对象列表以及个数，并创建相应数量的 ImageView 数量。

1.CreatePresentationSurface()

```
if( vkCreateWin32SurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr, &Vulkan.PresentationSurface ) == VK_SUCCESS ) {
      return true;
    }
```

这里创建的是Win32 的KHR

2.3.4 CheckPhysicalDeviceProperties()

接着上面使用vkGetPhysicalDeviceQueueFamilyProperties得到了结果之后进行查询判断

```
uint32_t graphics_queue_family_index = UINT32_MAX;
    uint32_t present_queue_family_index = UINT32_MAX;

    for( uint32_t i = 0; i < queue_families_count; ++i ) {
      vkGetPhysicalDeviceSurfaceSupportKHR( physical_device, i, Vulkan.PresentationSurface, &queue_present_support[i] );

      if( (queue_family_properties[i].queueCount > 0) &&
          (queue_family_properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) ) {
        // Select first queue that supports graphics
        if( graphics_queue_family_index == UINT32_MAX ) {
          graphics_queue_family_index = i;
        }

        // If there is queue that supports both graphics and present - prefer it
        if( queue_present_support[i] ) {
          selected_graphics_queue_family_index = i;
          selected_present_queue_family_index = i;
          return true;
```

```
      }
    }
  }

  // We don't have queue that supports both graphics and present so we have to use separate queues
  for( uint32_t i = 0; i < queue_families_count; ++i ) {
    if( queue_present_support[i] ) {
      present_queue_family_index = i;
      break;
    }
  }

  // If this device doesn't support queues with graphics and present capabilities don't use it
  if( (graphics_queue_family_index == UINT32_MAX) ||
      (present_queue_family_index == UINT32_MAX) ) {
    std::cout << "Could not find queue family with required properties on physical device " << physical_device << "!" << std::endl;
    return false;
  }

  selected_graphics_queue_family_index = graphics_queue_family_index;
  selected_present_queue_family_index = present_queue_family_index;
  return true;
```

得到两个index之后会先创建Device:GetDeviceQueue()

```
vkGetDeviceQueue( Vulkan.Device, Vulkan.GraphicsQueueFamilyIndex, 0, &Vulkan.GraphicsQueue );
vkGetDeviceQueue( Vulkan.Device, Vulkan.PresentQueueFamilyIndex, 0, &Vulkan.PresentQueue );
```

最后调用CreateSemaphores() 创建SwapChain:

```
if( vkCreateSwapchainKHR( Vulkan.Device, &swap_chain_create_info, nullptr, &Vulkan.SwapChain ) != VK_SUCCESS ) {
      std::cout << "Could not create swap chain!" << std::endl;
      return false;
    }
```