Introduce

Vulkan基本框架:

- Create a VkInstance
- Select a supported graphics card (VkPhysicalDevice)
- Create a VkDevice and VkQueue for drawing and presentation
- Create a window, window surface and swap chain
- Wrap the swap chain images into VkImageView
- Create a render pass that specifies the render targets and usage
- Create framebuffers for the render pass
- Set up the graphics pipeline
- Allocate and record a command buffer with the draw commands for every possible swap chain image
- Draw frames by acquiring images, submitting the right draw command buffer and returning the images back to the swap chain

整个流程的伪代码或者说调用到Vulkan的API如下:

```
#include <vulkan/vulkan.h>
// Pseudocode of what an application looks like. I've omitted most creation structures,
// almost all synchronisation and all error checking. This is not a copy-paste guide!
void DoVulkanRendering()
{
  const char *extensionNames[] = { "VK_KHR_surface", "VK_KHR_win32_surface" };
  // future structs will not be detailed, but this one is for illustration.
  // Application info is optional (you can specify application/engine name and version)
  // Note we activate the WSI instance extensions, provided by the ICD to
  // allow us to create a surface (win32 is an example, there's also xcb/xlib/etc)
  VkInstanceCreateInfo instanceCreateInfo = {
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO, // VkStructureType sType;
    NULL, // const void* pNext;
    0, // VkInstanceCreateFlags flags;
    NULL, // const VkApplicationInfo* pApplicationInfo;
    0, // uint32_t enabledLayerNameCount;
    NULL, // const char* const* ppEnabledLayerNames;
    2, // uint32_t enabledExtensionNameCount;
    extensionNames, // const char* const* ppEnabledExtensionNames;
  };
  VkInstance inst;
  vkCreateInstance(&instanceCreateInfo, NULL, &inst);
  // The enumeration pattern SHOULD be to call with last parameter NULL to
  // get the count, then call again to get the handles. For brevity, omitted
  VkPhysicalDevice phys[4]; uint32_t physCount = 4;
  vkEnumeratePhysicalDevices(inst, &physCount, phys);
  VkDeviceCreateInfo deviceCreateInfo = {
    // I said I was going to start omitting things!
  };
  VkDevice dev;
  vkCreateDevice(phys[0], &deviceCreateInfo, NULL, &dev);
  // fetch vkCreateWin32SurfaceKHR extension function pointer via vkGetInstanceProcAddr
  VkWin32SurfaceCreateInfoKHR surfaceCreateInfo = {
    // HINSTANCE, HWND, etc
  };
  VkSurfaceKHR surf;
  vkCreateWin32SurfaceKHR(inst, &surfaceCreateInfo, NULL, &surf);
  VkSwapchainCreateInfoKHR swapCreateInfo = {
    // surf goes in here
  };
  VkSwapchainKHR swap;
  vkCreateSwapchainKHR(dev, &swapCreateInfo, NULL, &swap);
  // Again this should be properly enumerated
  VkImage images[4]; uint32_t swapCount;
  vkGetSwapchainImagesKHR(dev, swap, &swapCount, images);
  // Synchronisation is needed here!
  uint32_t currentSwapImage;
  vkAcquireNextImageKHR(dev, swap, UINT64_MAX, presentCompleteSemaphore, NULL, &currentSwapImage);
  // pass appropriate creation info to create view of image
  VkImageView backbufferView;
  vkCreateImageView(dev, &backbufferViewCreateInfo, NULL, &backbufferView);
  VkQueue queue;
  vkGetDeviceQueue(dev, 0, 0, &queue);
  VkRenderPassCreateInfo renderpassCreateInfo = {
    // here you will specify the total list of attachments
    // (which in this case is just one, that's e.g. R8G8B8A8_UNORM)
    // as well as describe a single subpass, using that attachment
    // for color and with no depth-stencil attachment
  };
  VkRenderPass renderpass;
  vkCreateRenderPass(dev, &renderpassCreateInfo, NULL, &renderpass);
  VkFramebufferCreateInfo framebufferCreateInfo = {
    // include backbufferView here to render to, and renderpass to be
    // compatible with.
  };
  VkFramebuffer framebuffer;
  vkCreateFramebuffer(dev, &framebufferCreateInfo, NULL, &framebuffer);
  VkDescriptorSetLayoutCreateInfo descSetLayoutCreateInfo = {
    // whatever we want to match our shader. e.g. Binding 0 = UBO for a simple
    // case with just a vertex shader UBO with transform data.
  };
  VkDescriptorSetLayout descSetLayout;
  vkCreateDescriptorSetLayout(dev, &descSetLayoutCreateInfo, NULL, &descSetLayout);
  VkPipelineCreateInfo pipeLayoutCreateInfo = {
    // one descriptor set, with layout descSetLayout
```

```
};
VkPipelineLayout pipeLayout;
vkCreatePipelineLayout(dev, &pipeLayoutCreateInfo, NULL, &pipeLayout);
// upload the SPIR-V shaders
VkShaderModule vertModule, fragModule;
vkCreateShaderModule(dev, &vertModuleInfoWithSPIRV, NULL, &vertModule);
vkCreateShaderModule(dev, &fragModuleInfoWithSPIRV, NULL, &fragModule);
VkGraphicsPipelineCreateInfo pipeCreateInfo = {
  // there are a LOT of sub-structures under here to fully specify
  // the PSO state. It will reference vertModule, fragModule and pipeLayout
  // as well as renderpass for compatibility
};
VkPipeline pipeline;
vkCreateGraphicsPipelines(dev, NULL, 1, &pipeCreateInfo, NULL, &pipeline);
VkDescriptorPoolCreateInfo descPoolCreateInfo = {
  // the creation info states how many descriptor sets are in this pool
};
VkDescriptorPool descPool;
vkCreateDescriptorPool(dev, &descPoolCreateInfo, NULL, &descPool);
VkDescriptorSetAllocateInfo descAllocInfo = {
  // from pool descPool, with layout descSetLayout
};
VkDescriptorSet descSet;
vkAllocateDescriptorSets(dev, &descAllocInfo, &descSet);
VkBufferCreateInfo bufferCreateInfo = {
  // buffer for uniform usage, of appropriate size
};
VkMemoryAllocateInfo memAllocInfo = {
  // skipping querying for memory requirements. Let's assume the buffer
  // can be placed in host visible memory.
};
VkBuffer buffer;
VkDeviceMemory memory;
vkCreateBuffer(dev, &bufferCreateInfo, NULL, &buffer);
vkAllocateMemory(dev, &memAllocInfo, NULL, &memory);
vkBindBufferMemory(dev, buffer, memory, 0);
void *data = NULL;
vkMapMemory(dev, memory, 0, VK_WHOLE_SIZE, 0, &data);
// fill data pointer with lovely transform goodness
vkUnmapMemory(dev, memory);
VkWriteDescriptorSet descriptorWrite = {
  // write the details of our UBO buffer into binding 0
};
vkUpdateDescriptorSets(dev, 1, &descriptorWrite, 0, NULL);
// finally we can render something!
// ...
// Almost.
VkCommandPoolCreateInfo commandPoolCreateInfo = {
  // nothing interesting
};
VkCommandPool commandPool;
vkCreateCommandPool(dev, &commandPoolCreateInfo, NULL, &commandPool);
VkCommandBufferAllocateInfo commandAllocInfo = {
  // allocate from commandPool
};
VkCommandBuffer cmd;
vkAllocateCommandBuffers(dev, &commandAllocInfo, &cmd);
```

```
    // Now we can render!
    vkBeginCommandBuffer(cmd, &cmdBeginInfo);
    vkCmdBeginRenderPass(cmd, &renderpassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
    // bind the pipeline
    vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
    // bind the descriptor set
    vkCmdBindDescriptorSets(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS,
    descSetLayout, 1, &descSet, 0, NULL);
    // set the viewport
    vkCmdSetViewport(cmd, 1, &viewport);
    // draw the triangle
    vkCmdDraw(cmd, 3, 1, 0, 0);
    vkCmdEndRenderPass(cmd);
    vkEndCommandBuffer(cmd);
    VkSubmitInfo submitInfo = {
      // this contains a reference to the above cmd to submit
    };
    vkQueueSubmit(queue, 1, &submitInfo, NULL);
    // now we can present
    VkPresentInfoKHR presentInfo = {
      // swap and currentSwapImage are used here
    };
    vkQueuePresentKHR(queue, &presentInfo);
    // Wait for everything to be done, and destroy objects
  }
```

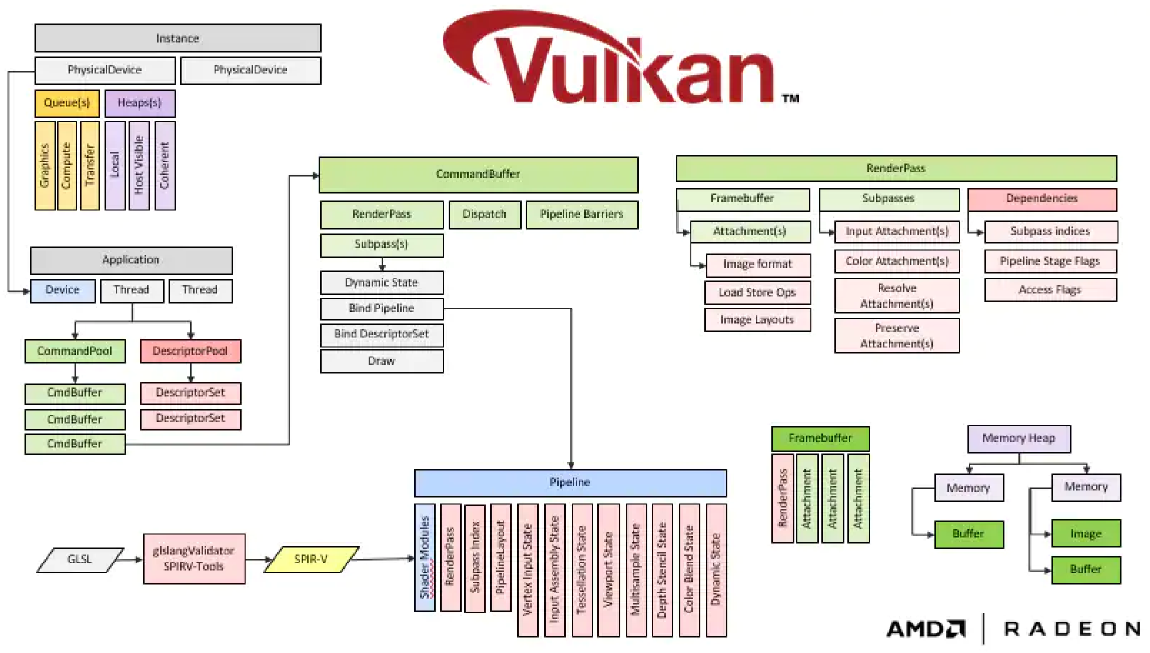
Vulkan in 30 minutes

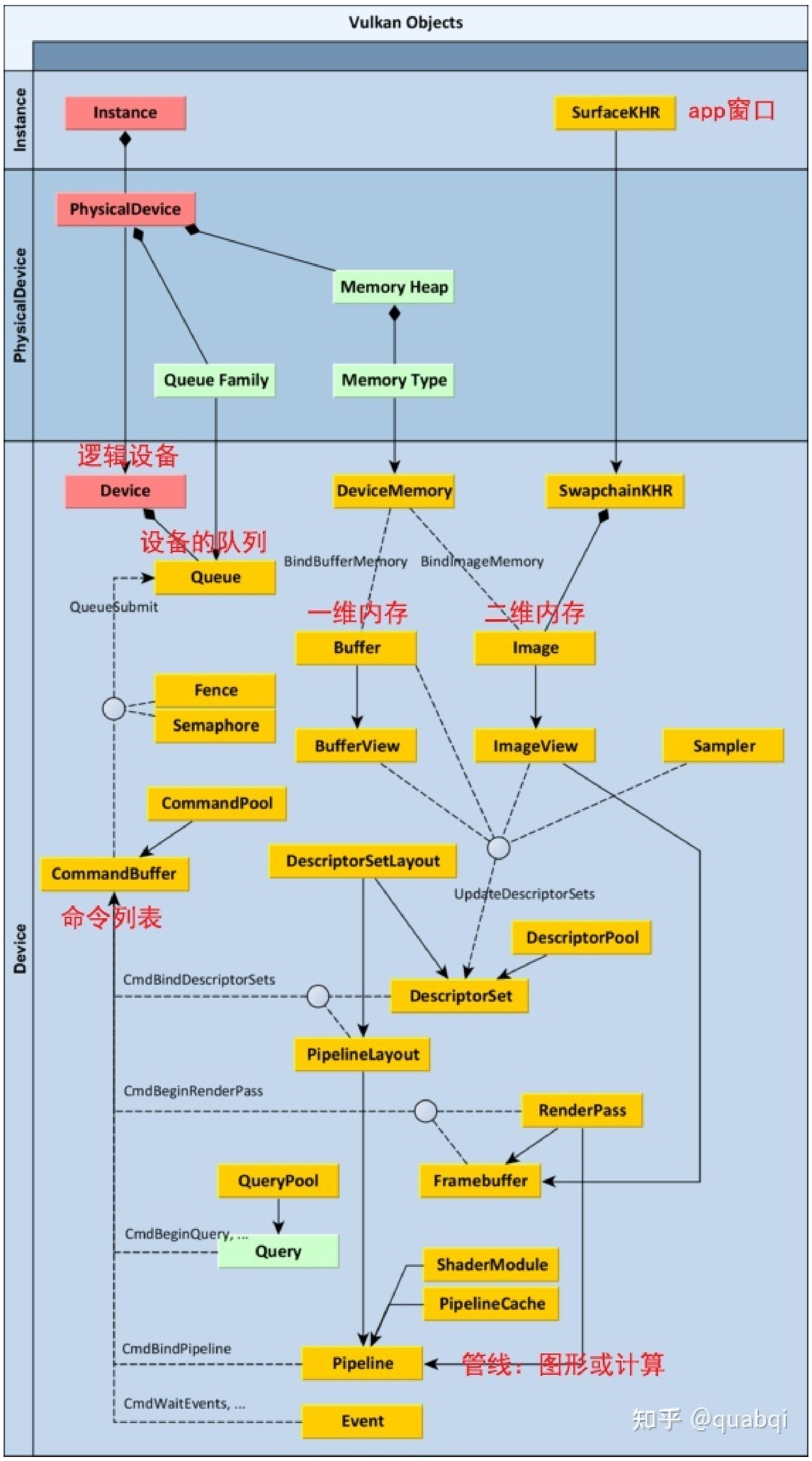
30 minutes not actually guaranteed. I've written this post with a specific target audience in mind, namely those who have a good grounding in existing APIs (e.g. D3D11 and GL) and understand the concepts of multithreading, staging resources, synchronisation and so on but want to know specifically how they are implemented in Vulkan.

https://renderdoc.org/vulkan-in-30-minutes.html

或者理解vulkan.h中的每个Handle

```
VK_DEFINE_HANDLE(VkInstance)
VK_DEFINE_HANDLE(VkPhysicalDevice)
VK_DEFINE_HANDLE(VkDevice)
VK_DEFINE_HANDLE(VkQueue)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
VK_DEFINE_HANDLE(VkCommandBuffer)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFence)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDeviceMemory)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBuffer)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImage)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkQueryPool)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBufferView)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImageView)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineCache)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineLayout)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkRenderPass)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipeline)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSampler)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorPool)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSet)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFramebuffer)
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkCommandPool)
```




Vulkan Objects
Instance
PhysicalDevice
Device
Instance
SurfaceKHR
app窗口
PhysicalDevice
Memory Heap
Queue Family
Memory Type
逻辑设备
Device
DeviceMemory
SwapchainKHR
设备的队列
Queue
BindBufferMemory
BindImageMemory
QueueSubmit
一维内存
二维内存
Buffer
Image
Fence
Semaphore
BufferView
ImageView
Sampler
CommandPool
CommandBuffer
DescriptorSetLayout
UpdateDescriptorSets
命令列表
DescriptorPool
CmdBindDescriptorSets
DescriptorSet
PipelineLayout
CmdBeginRenderPass
RenderPass
QueryPool
Framebuffer
CmdBeginQuery, ...
Query
ShaderModule
PipelineCache
CmdBindPipeline
Pipeline
管线：图形或计算
CmdWaitEvents, ...
Event
知乎 @quabqi