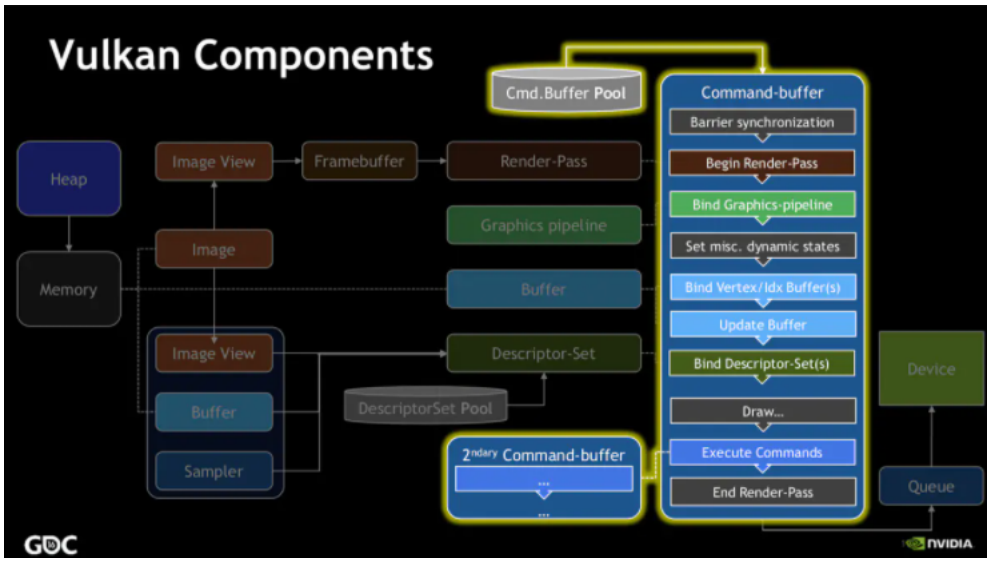
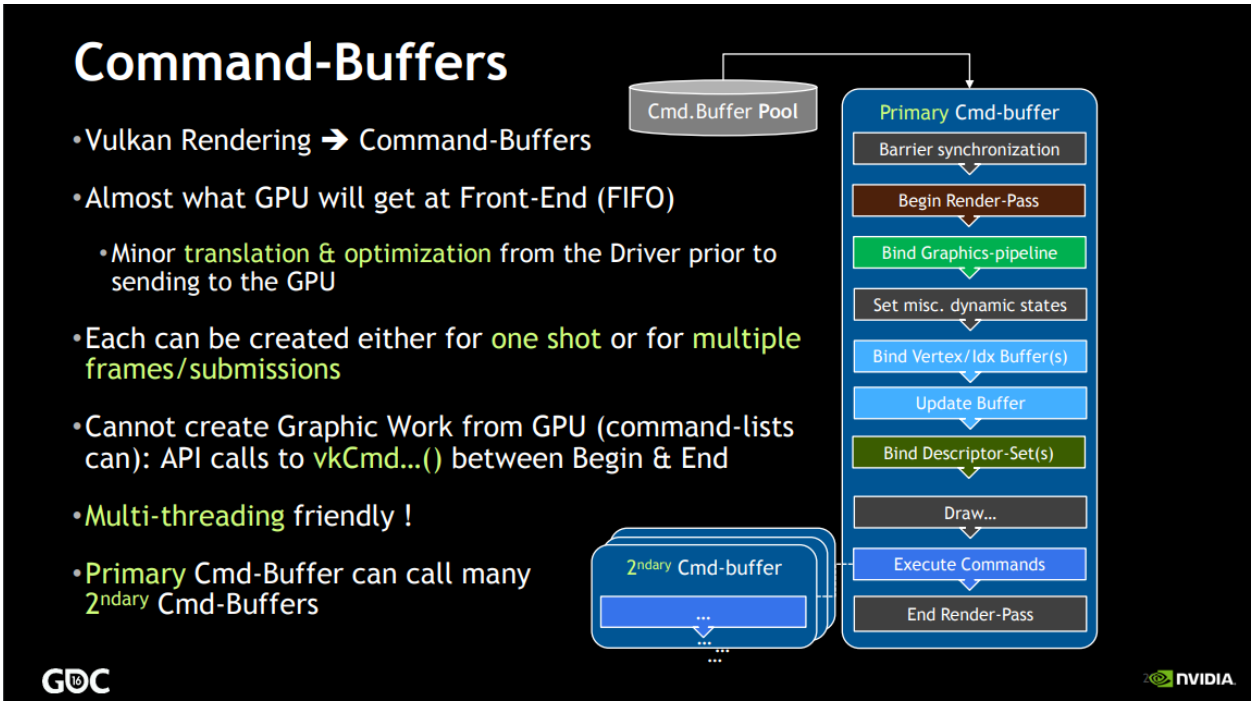
CommandBuffer & Barrier& PipeLine

前面已经创建了Instance &Devices & Queue,而CommandBuffer则为向Queue提交命令的缓存组件.

以https://github.com/GameTechDev/IntroductionToVulkan/tree/master/Project/Tutorials/03中的代码 为例进行分析

CommandBuffer





Vulkan provides two levels of command buffers:

- Primary command buffers, which can execute secondary command buffers, and which are submitted to queues;
- Secondary command buffers, which can be executed by primary command buffers, and which are not directly submitted to queues.

在 Vulkan 中，使用辅助命令缓冲区可能会显着提高性能。应用程序可以在为下一帧准备命令的单独线程上构建多个辅助命令缓冲区，而主线程正在执行主命令缓冲区。一旦通过某种线程同步发出信号，辅助命令缓冲区准备就绪，就可以将工作排入主命令缓冲区，并且可以重复该过程。

由于Command Buffer对象创建销毁成本较高，因此vulkan增加了一个CommandPool的对象，用于进行Command Buffer创销加速节省开销。

在创建 `Command-Buffer` 之前，需要创建 `Command-Pool` 组件，从 `Command-Pool` 中去分配 `Command-Buffer`

```
VkCommandPoolCreateInfo cmd_pool_create_info = {
      VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,     // VkStructureType               sType
      nullptr,                                        // const void                   *pNext
      0,                                              // VkCommandPoolCreateFlags      flags
      queue_family_index                              // uint32_t                      queueFamilyIndex
    };

    if( vkCreateCommandPool( GetDevice(), &cmd_pool_create_info, nullptr, pool ) != VK_SUCCESS ) {
      return false;
    }
    return true;
```

其中的queue_family_index为之前创建Queue时保存的index,也就意味者它们三者其实使相互关联的,即从 `Command-Pool` 中分配的的 `Command-Buffer` 必须提交到同一个 `Queue` 中.

另外上面的VkCommandPoolCreateFlags定义如下:

```
typedef enum VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001, //表示该 Command-Buffer 的寿命很短，可能在短时间内被重置或释放
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002, //表示从 Command-Pool 中分配的 Command-Buffer 可以通过 vkResetCommandBuffer 或
    VK_COMMAND_POOL_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCommandPoolCreateFlagBits;
```

接下来使用AllocateCommandBuffers创建真正的CommandBuffer

```
if( !CreateCommandPool( GetGraphicsQueue().FamilyIndex, &Vulkan.GraphicsCommandPool ) ) {
      std::cout << "Could not create command pool!" << std::endl;
      return false;
    }

    uint32_t image_count = static_cast<uint32_t>(GetSwapChain().Images.size());
    Vulkan.GraphicsCommandBuffers.resize( image_count, VK_NULL_HANDLE );

    if( !AllocateCommandBuffers( Vulkan.GraphicsCommandPool, image_count, Vulkan.GraphicsCommandBuffers.data() ) ) {
      std::cout << "Could not allocate command buffers!" << std::endl;
      return false;
    }
    return true;
```

vulkan提供两层Command Buffer结构：

1. Primary Command Buffer，初级CB，可以实现对二级CB的调用执行，这个CB会被提交到Queue中
2. Secondary Command Buffer，二级CB，不会直接提交到Queue，可以被初级CB调用执行。如果有了SCB的话，那么Primary 就不会做cb的记录，只会接受SCB的提交

CB中的命令包含绑定pipeline的命令、设置CB的descriptor的命令、修改dynamic states的命令、渲染命令、dispatch命令、执行二级CB的命令、拷贝buffer/image的命令等。每个CB对state的管理是完全独立的，且初级CB跟二级CB之间的State是没有继承关系的，且每个CB在开始记录命令之前，其状态是undefined，且初级CB在一个execute的二级CB开始之后，状态也会变成undefined的。

CommandBuffer是先收集一大堆命令，然后用vkQueueSubmit提交给设备的Queue。这个CommandBuffer是可以创建多个，BeginRenderPass调用时候传的CommandBuffer一般就是主CommandBuffer，而没传到RenderPass的都是子CommandBuffer，这样在多个线程上可以分别处理自己的命令到自己的子CommandBuffer上，最后都做完后通过ExecuteCommands把子CommandBuffer都合并到主CommandBuffer上，然后提交主CommandBuffer

Command-Buffer的生命周期

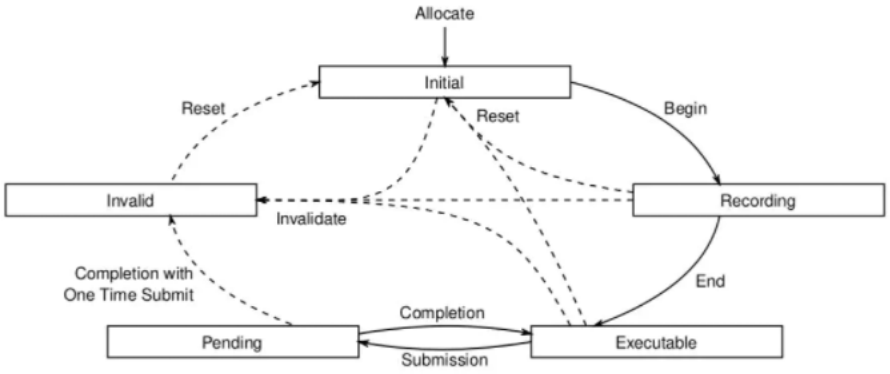


Figure 1. Lifecycle of a command buffer

- Initial 状态

在 `Command-Buffer` 刚刚创建时，它就是处于初始化的状态。从此状态，可以达到 `Recording` 状态，另外，如果重置之后，也会回到该状态。

- Recording 状态

调用 `vkBeginCommandBuffer` 方法从 `Initial` 状态进入到该状态。一旦进入该状态后，就可以调用 `vkCmd*` 等系列方法记录命令。

- Executable 状态

调用 `vkEndCommandBuffer` 方法从 `Recording` 状态进入到该状态，此状态下，`Command-Buffer` 可以提交或者重置。

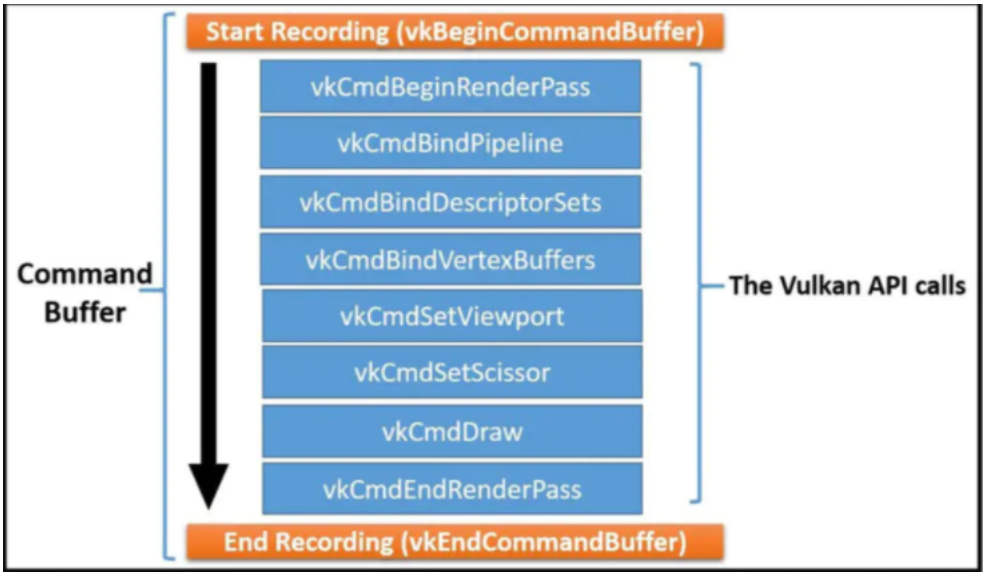
- Pending 状态

把 `Command-Buffer` 提交到 `Queue` 之后，就会进入到该状态。此状态下，物理设备可能正在处理记录的命令，因此不要在此时更改 `Command-Buffer`，当处理结束后，`Command-Buffer` 可能会回到 `Executable` 状态或者 `Invalid` 状态。

- Invalid 状态

一些操作会使得 `Command-Buffer` 进入到此状态，该状态下，`Command-Buffer` 只能重置、或者释放。

Command-Buffer的使用周期:



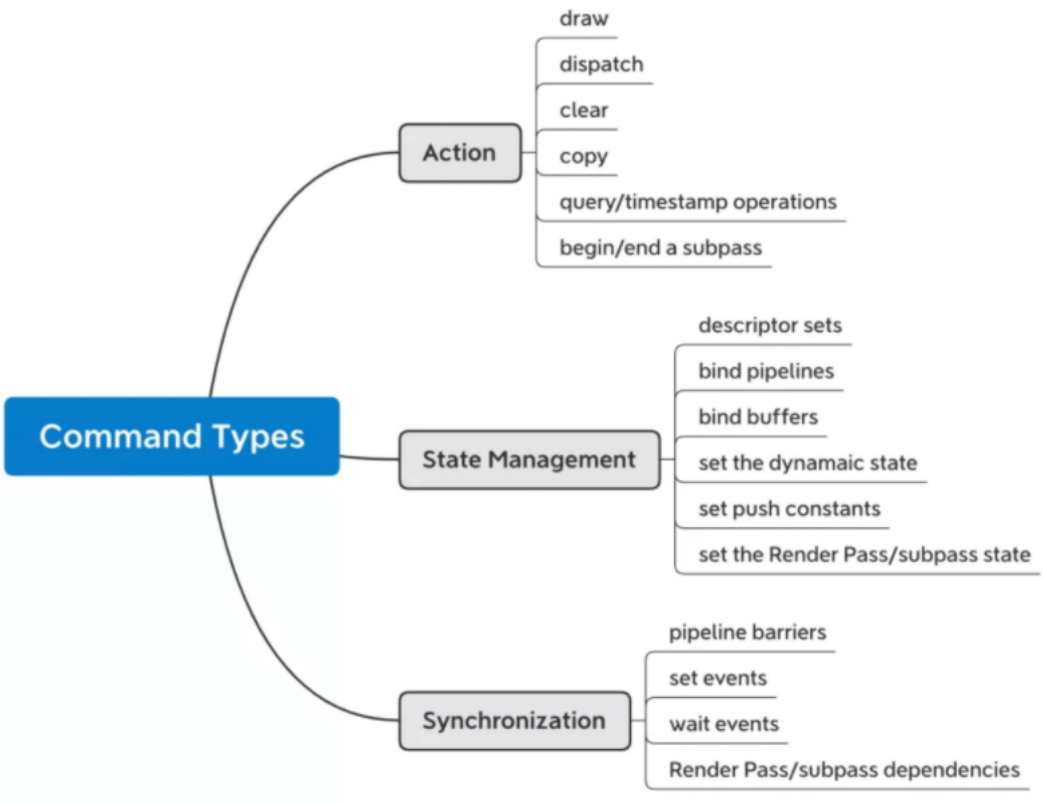
可以查看RecordCommandBuffers()函数,如下为其核心的提交部分:

```
vkCmdBeginRenderPass( Vulkan.GraphicsCommandBuffers[i], &render_pass_begin_info, VK_SUBPASS_CONTENTS_INLINE );

vkCmdBindPipeline( Vulkan.GraphicsCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, Vulkan.GraphicsPipeline );

vkCmdDraw( Vulkan.GraphicsCommandBuffers[i], 3, 1, 0, 0 );

vkCmdEndRenderPass( Vulkan.GraphicsCommandBuffers[i] );
```

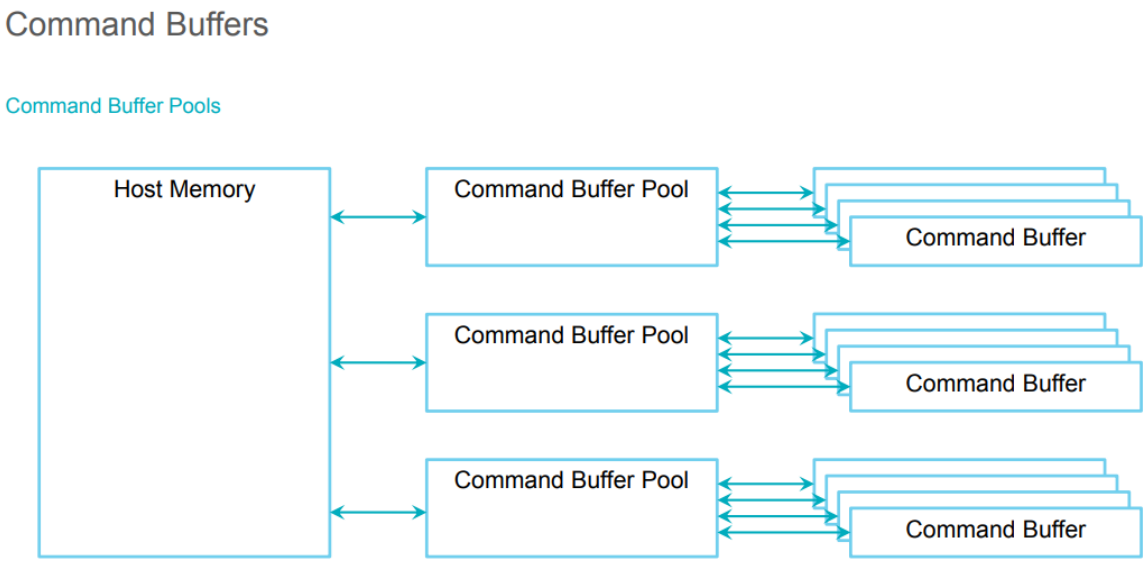
最后为其提交部分.

```
VkSubmitInfo submit_info = {
      VK_STRUCTURE_TYPE_SUBMIT_INFO,               // VkStructureType              sType
      nullptr,                                     // const void                  *pNext
      1,                                           // uint32_t                     waitSemaphoreCount
      &Vulkan.ImageAvailableSemaphore,             // const VkSemaphore           *pWaitSemaphores
      &wait_dst_stage_mask,                        // const VkPipelineStageFlags  *pWaitDstStageMask;
      1,                                           // uint32_t                     commandBufferCount
      &Vulkan.GraphicsCommandBuffers[image_index], // const VkCommandBuffer       *pCommandBuffers
      1,                                           // uint32_t                     signalSemaphoreCount
      &Vulkan.RenderingFinishedSemaphore           // const VkSemaphore           *pSignalSemaphores
    };

    if( vkQueueSubmit( GetGraphicsQueue().Handle, 1, &submit_info, VK_NULL_HANDLE ) != VK_SUCCESS ) {
      return false;
    }
```

CommandBuffer记录的命令可以有很多种类型,大体可以分为以下3种:



多线程的Command Buffer 构型：



Barrier

在这个示例中只使用到了vkCmdPipelineBarrier.

注意前缀vkCmd,说明其实这个也是一个Command 命令,在示例中可以看到vkCmdPipelineBarrier会跟进条件是否满足插入到每个Command buffer的生命周期vkBeginCommandBuffer 之后进行同步执行顺序

Execution Barrier就是简单的执行屏障。我们这里举一个来自于khronos官方的例子.

```
vkCmdDispatch(...);

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    ...);

vkCmdDispatch(...);
```

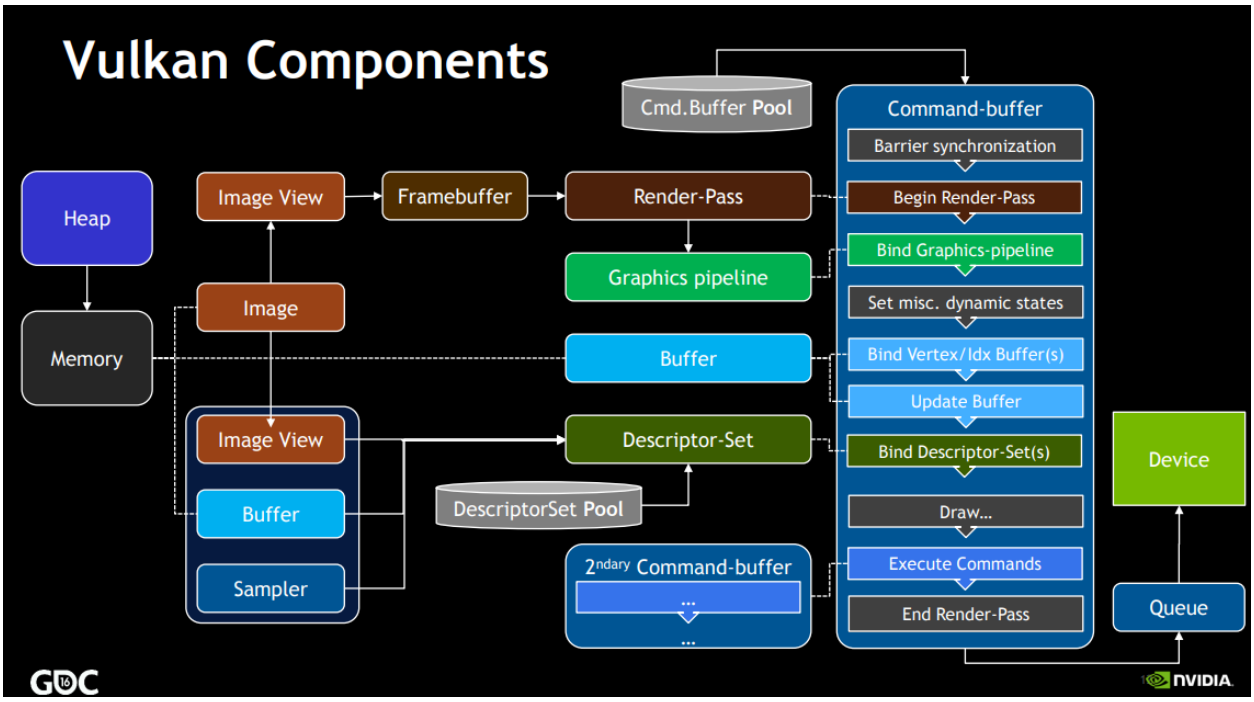
两个dispatch中，如果涉及到执行的先后顺序，就需要一个execution barrier。如果没有这个barrier，那么这两个dispatch先后顺序是无法预测的。很可能的情况是，第一个dispatch开始后，第二个dispatch也马上开始。至于谁先结束，无法预测。

对于理想模型而言，第一个dispatch执行完毕，更新resource的内容。等到第二个dispatch开始的时候，第二个dispatch可以立即获取resource中更新后的结果。然而，事实上，由于现代GPU同样采取了复杂的缓存控制机制，这个理想的模型是不存在的。一种可能的结果是，第一个dispatch执行完毕后, resource最新的内容被缓存到了某一级cache中。不幸的是，第二个dispatch开始执行的时候，这个cache对第二个dispatch不可见（例如，两个dispatch被分派到了不同的执行单元中）。尽管从顺序上，这的确保证了第一个dispatch先执行，然后才是第二个dispatch，但是我们仍然无法保证第二个dispatch能够看到第一个dispatch更新后的结果。

更多的Vlukan 同步问题,请参考:

Synchronization

PipeLine



更多请参考:

Graphics Pipeline