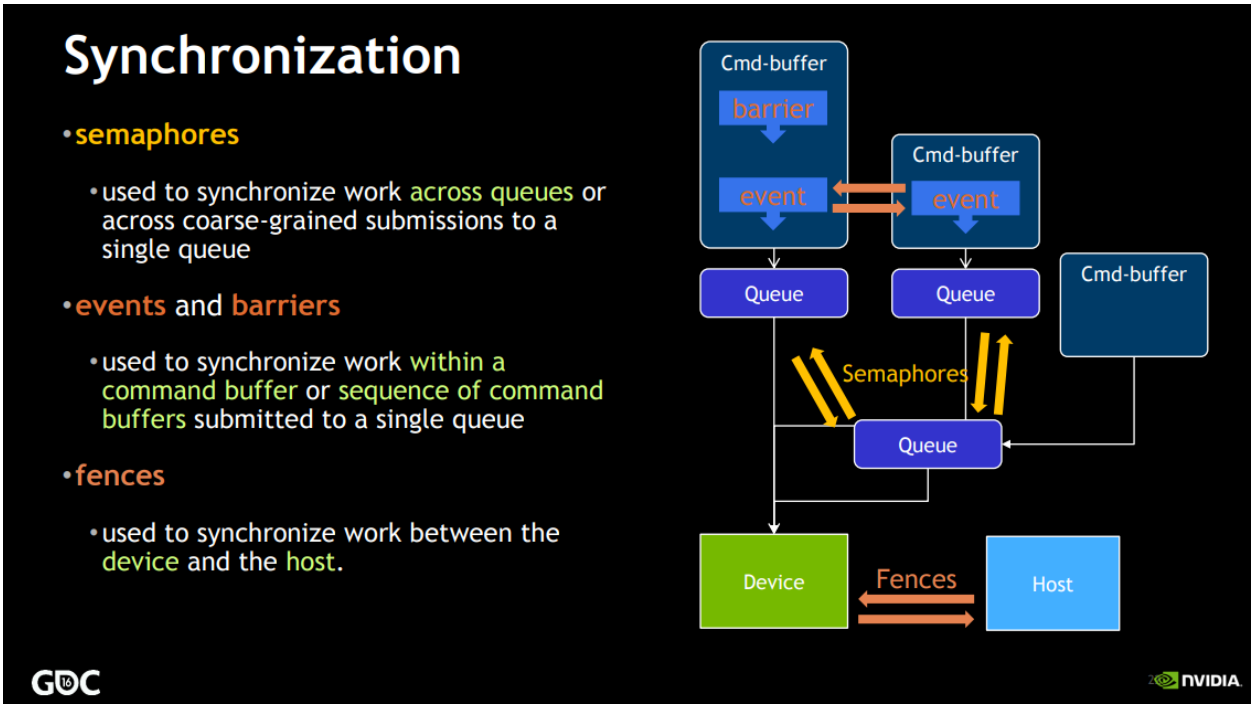
Synchronization



看图可以大概了解了这4个概念分别代表什么意思:

Fence:

Fence提供了一种粗粒度的，从Device向Host单向传递信息的机制。Host可以使用Fence来查询通过vkQueueSubmit/vkQueueBindSparse所提交的操作是否完成

```
VkResult vkQueueSubmit(
    VkQueue                                     queue,
    uint32_t                                    submitCount,
    const VkSubmitInfo*                         pSubmits,
    VkFence                                     fence);
```

在队列提交时候可以加上VkFence,后续可以通过其查询当前状态

Fence本身只有两种状态，unsignaled或者signaled，大致可以认为fence是触发态还是未触发态。当使用vkCreateFence创建fence对象的时候，如果在标志位上填充了

VkFenceCreateFlagBits(3)的VK_FENCE_CREATE_SIGNALED_BIT，那么创建出来的fence就是signaled状态，否则都是unsignaled状态的。销毁一个fence对象需要使用vkDestroyFence(3)。

伴随着vkQueueSubmit/vkQueueBindSparse一起提交的fence对象，可以使用vkGetFenceStatus(3)来查询fence的状态。注意vkGetFenceStatus是非阻塞的，如果fence处于signaled状态，这个API返回VK_SUCCESS，否则，立即返回VK_NOT_READY。

当然，fence被触发到signaled状态，必须存在一种方法，将之转回到unsignaled状态，这个功能由vkResetFences(3)完成，这个API一次可以将多个fence对象转到unsignaled状态。这个API结合VK_FENCE_CREATE_SIGNALED_BIT位，可以达到一种类似于C中do {} while;的效果，即loop的代码有着一致的表现：loop开始之前，所有的fence都创建位signaled状态，每次loop开始的时候，所用到的fence都由这个API转到unsignaled状态，伴随着submit提交过去。

等待一个fence，除了使用vkGetFenceStatus轮询之外，还有一个API vkWaitForFences(3)提供了阻塞式地查询方法。这个API可以等待一组fence对象，直到其中至少一个，或者所有的fence都处于signaled状态，或者超时（时间限制由参数给出），才会返回。如果超时的时间设置为0，则这个API简单地看一下是否满足前两个条件，然后根据情况选择返回VK_SUCCESS，或者（虽然没有任何等待）VK_TIMEOUT。

简而言之，对于一个fence对象，Device会将其从unsignaled转到signaled状态，告诉Host一些工作已经完成。所以fence使用在Host/Device之间的，且是一种比较粗粒度的同步机制。

Semaphore

VkSemaphore用以同步不同的queue之间，或者同一个queue不同的submission之间的执行顺序。需要注意的是semaphore只对device有效

```
typedef struct VkSubmitInfo {
    VkStructureType                sType;
    const void*                    pNext;
    uint32_t                       waitSemaphoreCount;
    const VkSemaphore*             pWaitSemaphores;
    const VkPipelineStageFlags*    pWaitDstStageMask;
```

```
    uint32_t                        commandBufferCount;
    const VkCommandBuffer*          pCommandBuffers;
    uint32_t                        signalSemaphoreCount;
    const VkSemaphore*              pSignalSemaphores;
} VkSubmitInfo;
```

所提交的command buffer将在执行到每个semaphore等待阶段时候，检查并等待每个对应的wait semaphore数组中的semaphore是否被signal, 且等到command buffer执行完毕以后，将所有signal semaphore数组中的semaphore都signal起来。

VkSubmitInfo(3)通过这种方式，实际上提供了一种非常灵活的同步queue之间或者queue内部不同command buffer之间的方法，通过组合使用semaphore，AP可以显式地指明不同command buffer之间的资源依赖关系，从而可以让driver在遵守这个依赖关系的前提下，最大程度地并行化，以提高GPU的利用效率。

另外Vulkan 1.1以后Semaphore可以用于Host和Device间的同步

Barrier

Barrier是**同一个queue**中的command，或者**同一个subpass**中的command所明确指定的依赖关系。

```
void vkCmdPipelineBarrier(
    VkCommandBuffer                             commandBuffer,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    VkDependencyFlags                           dependencyFlags,
    uint32_t                                    memoryBarrierCount,
    const VkMemoryBarrier*                      pMemoryBarriers,
    uint32_t                                    bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*                pBufferMemoryBarriers,
    uint32_t                                    imageMemoryBarrierCount,
    const VkImageMemoryBarrier*                 pImageMemoryBarriers);
```

vulkan API一贯味道的三个数组了 ，分别是memoryBarrier bufferMemoryBarrier以及最后的ImageMemoryBarrier。当着三个数组都为空的时候，将会在当前执行环境创建一个Execution Barrier，否则，则创建一个Memory Barrier。

Memory Barrier

memory barrier是一种更严格意义上的barrier。一个memory barrier同时兼备了execution barrier语义。memory barrier的引入主要是为了解决execution barrier中，无法有效控制缓存的缺点。

在Pipeline Barrier API中，可以指定三个数组。这三个数组，分别定义了不同类型的memory barrier:

- 全局memory barrier
- buffer上的memory barrier
- image上的memory barrier

全局memory barrier只有src的访问mask和dst的访问mask，因此作用于当前所有的resource。需要具体操纵某个resource的时候，根据resource的类型，分别使用buffer或者image的memory barrier.

Event

一个event，基本上和semaphore或者fence一样，由host创建，API为vkCreateEvent(3)：

```
VkResult vkCreateEvent(
    VkDevice                                    device,
    const VkEventCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkEvent*                                    pEvent);
```

创建event基本不需要额外的信息，并且在host端使用event也非常简单明了，比较复杂的是如何在device端使用event。

Event支持的操作

device上可以使用vkCmdSetEvent(3)触发(set)一个event，可以使用vkCmdResetEvent(3)重置一个event，还可以使用vkCmdWaitEvents(3)等待一个event被触发。其中，WaitEvents有着和barrier极为类似的设计，可以支持缓存控制。

host上可以使用vkSetEvent(3)触发event，也可以使用vkResetEvent(3)重置一个Event。如果host上需要等待event，需要使用vkGetEventStatus(3)来查询状态

其实由图可以简单的总结这四个同步如下:

Semaphores，用于多个 queue 之间的同步或者是一个 queue 的任务提交同步。

Events，用于一个 command buffer 内部的同步或在同一个 queue 内部多个 command buffer 的同步。

Fences，用于提供 devive 和 host 之间的同步。

barriers，用于精确控制 pipeline 中各个 pipeline 阶段的资源流动。