VulkanMemoryAllocator

Vulkan example 中的内存分配

看起来是自己分配管理的。

```
VkResult VulkanDevice::createBuffer(VkBufferUsageFlags usageFlags, VkMemoryPropertyFlags memoryPropertyFlags, vks::Buffer *buffer, VkDevice
  {
    buffer->device = logicalDevice;

    // Create the buffer handle
    VkBufferCreateInfo bufferCreateInfo = vks::initializers::bufferCreateInfo(usageFlags, size);
    VK_CHECK_RESULT(vkCreateBuffer(logicalDevice, &bufferCreateInfo, nullptr, &buffer->buffer));

    // Create the memory backing up the buffer handle
    VkMemoryRequirements memReqs;
    VkMemoryAllocateInfo memAlloc = vks::initializers::memoryAllocateInfo();
    vkGetBufferMemoryRequirements(logicalDevice, buffer->buffer, &memReqs);
    memAlloc.allocationSize = memReqs.size;
    // Find a memory type index that fits the properties of the buffer
    memAlloc.memoryTypeIndex = getMemoryType(memReqs.memoryTypeBits, memoryPropertyFlags);
    // If the buffer has VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT set we also need to enable the appropriate flag during allocation
    VkMemoryAllocateFlagsInfoKHR allocFlagsInfo{};
    if (usageFlags & VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT) {
      allocFlagsInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHR;
      allocFlagsInfo.flags = VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR;
      memAlloc.pNext = &allocFlagsInfo;
    }
    VK_CHECK_RESULT(vkAllocateMemory(logicalDevice, &memAlloc, nullptr, &buffer->memory));

    buffer->alignment = memReqs.alignment;
    buffer->size = size;
    buffer->usageFlags = usageFlags;
    buffer->memoryPropertyFlags = memoryPropertyFlags;

    // If a pointer to the buffer data has been passed, map the buffer and copy over the data
    if (data != nullptr)
    {
      VK_CHECK_RESULT(buffer->map());
      memcpy(buffer->mapped, data, size);
      if ((memoryPropertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT) == 0)
        buffer->flush();

      buffer->unmap();
    }

    // Initialize a default descriptor that covers the whole buffer size
    buffer->setupDescriptor();

    // Attach the memory to the buffer object
    return buffer->bind();
  }
```

由上代码可以看出整个buffer的 内存申请流程，先通过vkCreateBuffer申请handle，然后根据handle通过vkGetBufferMemoryRequirements生成VkMemoryRequirements ，最后根据生成的VkMemoryAllocateInfo 调用 vkAllocateMemory 才算是真正的申请到了buffer 的内存

Unreal 关于Vulkan的内存封装

```
//VulkanMemory.h
```

```
constexpr static uint32 BufferSizes[(int32)EPoolSizes::SizesCount + 1] =
    {
//      64 * 1024,
//      64 * 1024,
      128 * 1024,
      128 * 1024,
      256 * 1024,
      256 * 1024,
      512 * 1024,
      512 * 1024,
      1024 * 1024,
      1 * 1024 * 1024,
    };

//
AllocateBufferPooled
```

可以看到其实它的内存是自己分配的，和Unreal的内存管理是一样的，通过内存池来分配

可以通过搜索关键词vkAllocateMemory ，在vulkan 中查看其调用，也可以发现确实是这样的，可以看下代码：

```
// New Buffer
    const uint32 BufferSize = FMath::Max(Size, BufferSizes[PoolSize]);

    VkBuffer Buffer;
    VkBufferCreateInfo BufferCreateInfo;
    ZeroVulkanStruct(BufferCreateInfo, VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO);
    BufferCreateInfo.size = BufferSize;
    BufferCreateInfo.usage = BufferUsageFlags;
    VERIFYVULKANRESULT(VulkanRHI::vkCreateBuffer(Device->GetInstanceHandle(), &BufferCreateInfo, VULKAN_CPU_ALLOCATOR, &Buffer));

    VkMemoryRequirements MemReqs;
    VulkanRHI::vkGetBufferMemoryRequirements(Device->GetInstanceHandle(), Buffer, &MemReqs);
    Alignment = FMath::Max((uint32)MemReqs.alignment, Alignment);
    ensure(MemReqs.size >= BufferSize);

    uint32 MemoryTypeIndex;
    VERIFYVULKANRESULT(Device->GetDeviceMemoryManager().GetMemoryTypeFromProperties(MemReqs.memoryTypeBits, MemoryPropertyFlags, &MemoryTyp

    bool bHasUnifiedMemory = DeviceMemoryManager->HasUnifiedMemory();
    FDeviceMemoryAllocation* DeviceMemoryAllocation = DeviceMemoryManager->Alloc(true, MemReqs.size, MemoryTypeIndex, nullptr, Priority, fa
```

可以看到和example中的调用其实是一样的，只不过大小是由BufferSize 来决定，BufferSize 由池子里分配的内存来决定。

VMA（Vulkan Memory Allocator）

VMA（Vulkan Memory Allocator），是AMD提供的Vulkan内存分配管理器。

那么Vulkan的内存分配为何要使用VMA这种内存分配器呢？原因就在于其显存的分配次数是有限的（比如4096次），那么我们就需要分配一整块显存，然后自己使用offset以及size来进行分割使用，这个过程冗长繁杂，而且容易出错，那么VMA就成为了我们管理Vulkan内存的首要选择。

以下是使用指南和官方链接：

https://gpuopen.com/vulkan-memory-allocator/

https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/

https://gitee.com/masa-laboratory/vulkan-memory-allocator#vulkan-memory-allocator

1. Initialize Vulkan to have `VkPhysicalDevice`, `VkDevice` and `VkInstance` object.
2. Fill **VmaAllocatorCreateInfo** structure and create **VmaAllocator** object by calling **vmaCreateAllocator()**.

在创建的时候需要绑定一下：

```
VmaAllocatorCreateInfo allocator_info = {};
  allocator_info.physicalDevice = physical_device;
  allocator_info.device = logical_device;
  vmaCreateAllocator(&allocator_info, &allocator_);
```

之后如果需要申请内存的话可以

```
const VkResult res = vmaCreateBuffer(allocator_, conv_buff_info, alloc_info, &alloc_buffer, &buff_alloc, nullptr);

vmaDestroyBuffer(allocator, buffer, allocation);
vmaDestroyAllocator(allocator);
```

在vmaCreateBuffer里面会封装好了相应的申请封装。

至于vma底层是怎么分配的，可以看底下的部分代码，不详细探究了，但大概率猜测也是通过某种pool来进行分配的

```
if(createInfo.pool != VK_NULL_HANDLE)
    {
        const VkDeviceSize alignmentForPool = VMA_MAX(
            vkMemReq.alignment,
            GetMemoryTypeMinAlignment(createInfo.pool->m_BlockVector.GetMemoryTypeIndex()));

        VmaAllocationCreateInfo createInfoForPool = createInfo;
        // If memory type is not HOST_VISIBLE, disable MAPPED.
        if((createInfoForPool.flags & VMA_ALLOCATION_CREATE_MAPPED_BIT) != 0 &&
            (m_MemProps.memoryTypes[createInfo.pool->m_BlockVector.GetMemoryTypeIndex()].propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BI
        {
            createInfoForPool.flags &= ~VMA_ALLOCATION_CREATE_MAPPED_BIT;
        }

        return createInfo.pool->m_BlockVector.Allocate(
            m_CurrentFrameIndex.load(),
            vkMemReq.size,
            alignmentForPool,
            createInfoForPool,
            suballocType,
            allocationCount,
            pAllocations);
    }
```