

## Staging buffer;

**Transform queue** ，**finish**

## Staging buffer;

**mangae memory buffer**

You can either implement such an allocator yourself, or use the [VulkanMemoryAllocator](https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator) library provided by the GPUOpen initiative. However, for this tutorial it's okay to use a separate allocation for every resource, because we won't come close to hitting any of these limits for now.

**1使用 VMA**

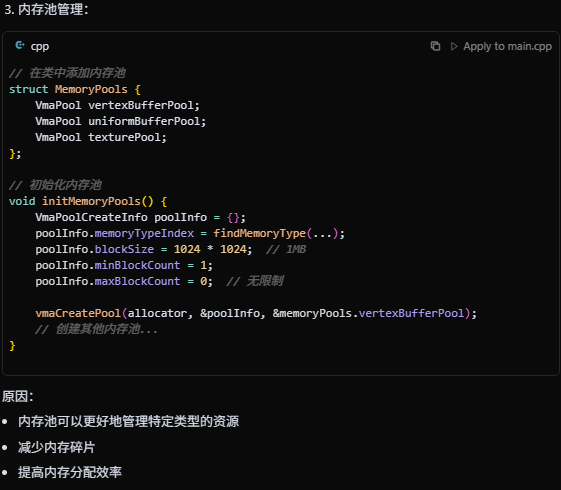
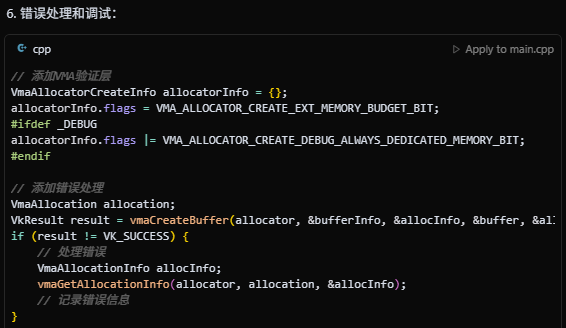
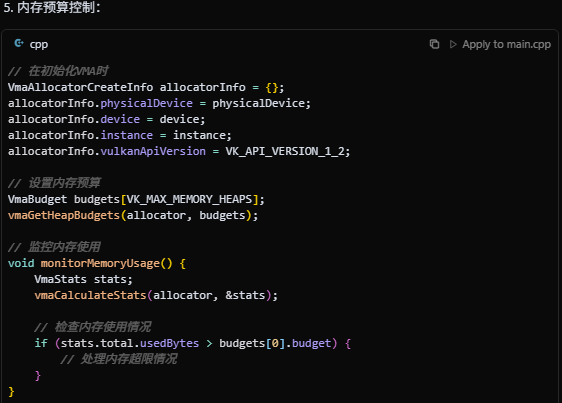
顶点缓冲区（Vertex Buffer）

索引缓冲区（Index Buffer）

统一缓冲区（Uniform Buffer）

纹理图像内存（textureimage，depthresourceimage）

命令缓冲区内存**？？？**

**** ****

**2自定义分配器**

Vulkan内存分配器接口：VkAllocationCallbacks

内存分配策略：内存池管理，内存碎片处理，内存对齐要求，内存类型选择

性能优化：内存重用，内存池预分配，内存对齐优化，内存访问模式优化

内存追踪：内存泄漏检测，内存使用统计，内存分配/释放日志

## Index buffer；go further in managing memory buffer

**Buffer Packing 目前将indexbuffer和vertexbuffer合并使用**

**Memory Aliasing**

The previous chapter already mentioned that you should allocate multiple resources like buffers from a single memory allocation, but in fact you should go a step further. [Driver developers recommend](https://developer.nvidia.com/vulkan-memory-management) that you also store multiple buffers, like the vertex and index buffer, into a single [VkBuffer](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkBuffer.html) and use offsets in commands like [vkCmdBindVertexBuffers](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkCmdBindVertexBuffers.html). The advantage is that your data is more cache friendly in that case, because it's closer together. It is even possible to reuse the same chunk of memory for multiple resources if they are not used during the same render operations, provided that their data is refreshed, of course. This is known as *aliasing* and some Vulkan functions have explicit flags to specify that you want to do this.

在前一个章节的基础上修改为四个顶点（注意背面剔除）

利用inputAssembly.topology = VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_STRIP;的效果



## [Descriptor pool and sets](https://docs.vulkan.org/tutorial/latest/05_Uniform_buffers/01_Descriptor_pool_and_sets.html)

**Alignment requirements**

## [Images](https://docs.vulkan.org/tutorial/latest/06_Texture_mapping/00_Images.html)

这里在 createTextureImage 函数内，将 graphicsQueue 和 transferQueue 分开使用，因为前面提前单独检测是否有transfer函数

**See**[**the documentation**](https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/chap7.html#VkPipelineStageFlagBits)**for more information and other examples of pseudo-stages.**

**setupCommandBuffer**

All of the helper functions that submit commands so far have been set up to execute synchronously by waiting for the queue to become idle. For practical applications it is recommended to combine these operations in a single command buffer and execute them asynchronously for higher throughput, especially the transitions and copy in the createTextureImage function. Try to experiment with this by creating a setupCommandBuffer that the helper functions record commands into, and add a flushSetupCommands to execute the commands that have been recorded so far. It’s best to do this after the texture mapping works to check if the texture resources are still set up correctly.

### textureImage 与swapChain

**textureImage 与 textureImageMemory 的关系：**

* **textureImage (VkImage)**:
* 这只是一个**逻辑上的图像对象句柄**。它定义了图像的格式、尺寸、用途（例如，用作纹理），但它**本身不包含任何像素数据**。
* 你可以把它想象成一个“纹理的规格说明书”。
* **textureImageMemory (VkDeviceMemory)**:
* 这是真正**分配在 GPU 上的物理内存块**。这块内存就是用来存储 textureImage 的**实际像素数据**的地方。
* 你可以把它想象成一个“存储像素数据的箱子”。
* **关系：** 你通过 vkCreateImage 创建 textureImage 这个“规格说明书”，然后通过 vkAllocateMemory 分配 textureImageMemory 这个“箱子”，最后通过 vkBindImageMemory 把这个“规格说明书”和“箱子”**连接起来**。应用程序完全掌控 VkImage 和 VkDeviceMemory 的生命周期和数据传输。

**swapChain 链式组件的关系：**

* **swapChain (VkSwapchainKHR)**:
* 这是 Vulkan 提供的一个**高级抽象对象**，它负责与底层的**窗口系统**（如 Windows、Linux X11/Wayland）进行交互，管理用于**屏幕呈现**的一组图像。
* 它不是直接存储像素数据，而是**管理着一系列可以显示在屏幕上的图像**。
* **swapChainImages (std::vector<VkImage>)**:
* 当 swapChain 创建时，Vulkan 驱动和窗口系统会**自动在内部创建和管理**若干个 VkImage 对象（通常是双缓冲或三缓冲）。
* swapChainImages 向量里存储的，就是这些由 swapChain **拥有并管理的 VkImage 的句柄**。你只是获取这些句柄来引用它们，但你**不负责这些 VkImage 底层内存的分配和绑定**。这些都是由 swapChain 内部处理的，因为它们是用于呈现的特殊图像。
* 这些 VkImage 才是真正用于承载渲染结果的像素数据。
* **swapChainImageViews (std::vector<VkImageView>)**:
* VkImage 是原始的内存块，GPU 管线不能直接使用它。你需要一个 **VkImageView 来定义如何“看待”和访问**这个 VkImage。
* 每个 swapChainImageViews[i] 都是为 swapChainImages[i] 创建的**一个“视图”**。这个视图定义了图像的类型（2D 纹理）、格式、哪些层和 Mipmap 级别是可见的等等。
* GPU 在渲染时，是通过 ImageView 来访问 Image 中的像素数据的。
* **swapChainFramebuffers (std::vector<VkFramebuffer>)**:
* VkFramebuffer 是一个**渲染目标的集合**，它将一个或多个 VkImageView（例如，一个颜色附件 ImageView，一个深度附件 ImageView）组合在一起，作为 Render Pass 的实际渲染目标。
* 每个 swapChainFramebuffers[i] 都是将对应的 swapChainImageViews[i] 绑定到你的 Render Pass 上。当你在命令缓冲区中开始一个 Render Pass 时，你指定要渲染到哪个 Framebuffer，从而间接指定了渲染结果应该写入哪个 swapChainImage。

**为什么不一样？ (核心原因)**

关键在于**所有权和职责分工**：

1. **普通纹理 (e.g., textureImage)：**

* 所有权在你应用程序手里。你需要完全控制内存的分配、绑定、数据上传和生命周期管理。
* 所以你需要显式地 vkCreateImage (逻辑对象) 和 vkAllocateMemory/vkBindImageMemory (物理存储)。

1. **交换链图像 (e.g., swapChainImages)：**

* 所有权在 Vulkan 驱动和底层窗口系统手里。这些图像是专门用于“呈现”的，它们的内存管理和同步机制非常特殊，需要与操作系统显示服务器紧密配合。
* 作为应用程序开发者，你**不需要关心这些图像的底层内存是如何分配和绑定的**，VkSwapchainKHR 会为你处理。你只获取到这些图像的句柄 (VkImage)，然后为它们创建**视图** (VkImageView)，再将视图组织成**帧缓冲** (VkFramebuffer)，供渲染管线使用。

## Combined image sampler

It is possible to **use texture sampling in the vertex shader**, for example to dynamically deform a grid of vertices by a [heightmap](https://en.wikipedia.org/wiki/Heightmap).

### textureImage 和 colorAttachment 通过渲染管线和描述符集进行连接

**textureImage 通过描述符集与着色器连接：**

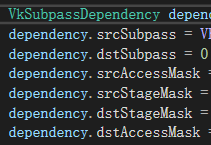
* 你在 createDescriptorSetLayout() 中定义了一个 VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER 类型的描述符绑定，这个绑定会接收一个图像视图（textureImageView）和一个采样器（textureSampler）。
* 在 createDescriptorSets() 中，你将 textureImageView 和 textureSampler 绑定到这个描述符上。
* 在 recordCommandBuffer() 中，你通过 vkCmdBindDescriptorSets 命令将描述符集绑定到管线，这样你的片元着色器就可以访问这个纹理了。
* 在着色器中，你会声明一个 sampler2D 或 sampler3D 变量，并通过它来采样纹理图像。

**colorAttachment 通过渲染通道和帧缓冲区连接：**

* 在 createRenderPass() 中，你定义了一个 VkAttachmentDescription 作为颜色附件，并指定了它的格式 (swapChainImageFormat) 和初始/最终布局。
* 在 createFramebuffers() 中，你将**交换链图像的视图** (swapChainImageViews[i]) 和深度图像的视图 (depthImageView) 作为附件连接到帧缓冲区。
* 在 recordCommandBuffer() 中，当调用 vkCmdBeginRenderPass 时，你指定了要使用的 renderPass 和 framebuffer。这个帧缓冲区包含了颜色附件（即交换链图像的视图），渲染结果将写入到这些图像中。

## Depth buffering

对于不透明的物体，深度测试是最佳选择。只有在处理透明物体时，才需要考虑使用从后到前的排序方法

这个的作用是什么

## Generating Mipmaps

Beware if you are using a **dedicated transfer queue** (as suggested in [Vertex buffers](https://vulkan-tutorial.com/Vertex_buffers/Staging_buffer)): [vkCmdBlitImage](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkCmdBlitImage.html) must be submitted to a **queue with graphics capability.完成**

View controller **完成**

There are two alternatives in this case. You could implement a function that searches common texture image formats for one that *does* support linear blitting, or you could implement the mipmap generation in software with a library like [stb\_image\_resize](https://github.com/nothings/stb/blob/master/stb_image_resize.h). Each mip level can then be loaded into the image in the same way that you loaded the original image.

It should be noted that it is uncommon in practice to generate the mipmap levels at runtime anyway. Usually they are pregenerated and stored in the texture file alongside the base level to improve loading speed. **Implementing resizing in software and loading multiple levels from a file is left as an exercise to the reader.**

### Multisampling

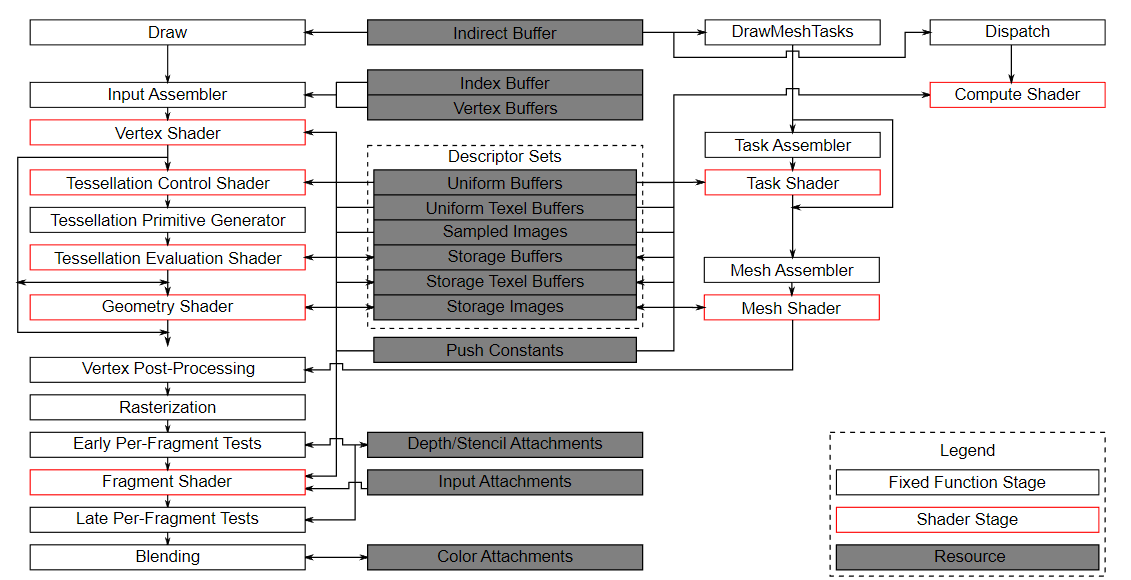
mipmap 优化纹理采样（输入），MSAA 优化几何边缘的渲染（输出）



MSAA 是在**光栅化阶段的后期**以及**片段着色器输出写入渲染目标时**起作用的。

* 在光栅化阶段，MSAA 会决定几何图元覆盖了像素内的哪些子样本。
* 片段着色器计算出的颜色会写入到这些被覆盖的子样本中。
* 当渲染通道结束或执行解析操作时，这些多重采样的子样本才会被混合（解析）成一个最终的单采样像素颜色，并存储到 swapChainImage 中。
* **MSAA 与 Mipmap 是独立的**：MSAA 发生在像素内部，处理的是几何边缘的锯齿，而 Mipmap 发生在纹理采样时，处理的是纹理细节的混叠。

## Compute Shader



**async compute**

**Shared memory**

**Asynchronous compute**

**Atomic operations**

**Subgroups**