Vulkan 编程指南

Alexander Overvoorde 著
fangcun 译

序

下文来自仓颉输入法的发明人朱邦复先生，放在这里的原因是本人认为 Vulkan 之于汇编有着相似的地位。

一、结构基础

物质文明之有今天的成就，是因为人类掌握了物质的基本结构。物质的种类无穷，但是却都由基本元素交互组成，只要根据一定的法则，就能得到一定的结果。

计算机技术虽然日新月异，应用软件的变化也无止无尽，而其基本因子却非常有限。各种微处理器的汇编语言，正是计算机软件的基础结构，任何要通过软件以完成的动作，都是经由汇编语言的指令群，逐步执行的。

因为计算机结构复杂，各种任务分工极精，即使是一位资深的高级程序员，终其生也不过局限在若干固定的程序中钻研，很难以宏观的立场认知全貌。再加上市场压力，局外人莫名其奥妙，局中人又忙得不可开交，所以还没有任何人能作出全盘的评估。

汇编语言首先成为被误解的牺牲者，包括应用它的系统工程师在内，都一致认为它「难学难用」,（中文也是一种组合形式的应用，其所组合者是人的概念。无独有偶，人们在不求甚解之余，都视之为畏途。）事实上大谬不然，现在是科学挂帅，而科学的精义就在于系统的分类和应用。问题是我们能不能归纳出一些学习、应用的法则，将组合的过程化繁为简，以符合各种应用范畴。

二、个人体验

我个人对此感受极为深切，我原是个十足的外行，1978 年第一次接触计算机，曾以不到两周的时间，就学会计算机操作，并应用「培基语言」设计完成”仓颉输入”程序。当时我认为培基语言易学易用，是计算机上最好的工具。

后来，我开始用培基语言设计”仓颉向量组字”程序，每秒可生成两个字，当时与我合作的宏碁公司建议我采用汇编语言，他们说组字程序速度要快，培基语言不能胜任。如改用汇编语言，效率可提高十倍，由此开始了我与汇编语言的不解之缘。1979 年９月我们正式推出了由国人自行设计、具有完整的计算机功能、可运用数万中文字的”天龙中文计算机”。

宏碁公司动用了三位资深工程师，采用 Z80 MCZ 系统，以六个月的时间完成了向量组字及系统程序，记忆空间占 60KB，处理速度每秒约组成 30 字。

这是我首次发现到汇编语言的威力，深究之下，才理解到计算机的全部工作原理。简单说来，汇编语言就是组合计算机所有功能的控制指令，利用它，就可以直接控制计算机。

其它高级语言，只是让人省事，用一些格式化的手续，把人的想法化为过程的指令，这种情形就相当于为了迁就开车的人，建了密如蛛网的高速公路。本来走路只要几分钟就可到达的地方，以车代步的结果，反而需要耗费半个小时。

1980 年，我决定自己动手，又重新设计了一套字数较多，字形较美观的组字程序。只用了三个月的时间，结果不仅记忆空间缩小了三分之一，速度也快了十倍，达到每秒 300 字。这个产品，就是 1 苹果机上用的「汉卡」。

1983 年，再经分析，我发现以往写的程序很不精简，技术也不成熟。我坚信中文字形在计算机上的应用，将是中国文化存亡兴衰的根本因素，不仅值得投注自己的时间及精力，且也有此必要。所以我又抛掉了一切，重头设计，加入更多的变化参数，并根据人的辨识原理，设计成第三代至第五代等多种字形产生器。每一代之间，速度都明显地提高，功能也不断加强。在这样一再重复的摸索中，尝试了各种可行的途径，充份认识了汇编语言的特性及长处。

由于汇编语言灵活无比的特性，我发现它就如同画家的画笔一般，只为了牟利，可以用它画成各种廉价速成的商品；一旦投入自己的理想与心智，画笔就不再只是一枝笔，而成为人心与外界的界面，画出的作品立时升华成为艺术，进入一个更高的境界！

1985 年，我再次重新设计规划，采用人的智能原则，把人写字、认字的观念化为数据结构，程序只是用来阐释数据、控制计算机的界面。该字库的字形可做到无级次放大缩小，字体、字型皆能任意变化 (每字可以产生数亿种变形) 。而且除了现今各种字典已收的六万余字外，还可以组成完全符合中文规则的新字六百万个，足敷未来新时代新观念的发挥应用。

不仅如此，组字速度又提高了，每秒可以组成 30*30 的字形两千个！当然现在用的是 15MHZ 80286 ，比以往的 4.75 MHZ 的 Z80 已经快了近六倍。但是，改良后的新程序，其功能的增加，处理过程的繁杂性已远非当年可比。

这些成果，用了很多特殊的数据结构技巧，不可能经由高级语言来完成。既然用汇编语言所制作的程序能一再大幅度地改进，这就说明了汇编语言的弹性极大，效率相去千里。如不痛下苦功钻研，程序写完，能执行就算了事，又怎能领悟其中奥妙？

所以，我并不认为汇编语言只是一种程序语言而已，它是一种创造艺术品的工具，它能赋与无知无觉的电子机器一种「生命」，由无知进而有知，由有知而生智能。通过对汇编语言的研究探索，我整理出一些规律，写成这本书，以便于理解及应用。但是，要真正将汇编语言发展成为艺术，尚有待青年朋友们继续努力，在这个信息时代，开拓出一片崭新的天地。

无意义的音符能编成美妙的音乐，无规律的色彩可幻化为缤纷的世界，为什么计算机的机器指令，不能架构出信息的理性天地？

这就是艺术，作为艺术家，就必须奉献出自己的心血，以真、善、美为最高境界。

要达到这种目的，就要认真的作好准备动作，再一步一步地追求下去。

三、利人与利己

任何一种商业产品，当然是以利益为先，利己后而利人。如果是艺术品创造，则刚刚相反，唯有能忽视己利，沥血泣心地探索，虔诚狂热地奉献，才会迸发出人性的光辉，创造不朽的杰作。

艺术家之伟大，在于此，人性之可贵，在于此。

对组合程序语言，有人视为商品，将写作技巧当作专利，轻不示人。相信这也是迄今尚无一本象样的参考书籍之根本原因，我买了不少这类书，但书中除了指令介绍以及编程、侦错的手续外，完全没有技巧的说明，好象懂得指令就可以把程序写好一般。当我自己下了不少功夫，得到了一些心得，再回过头来看那些参考书，才发现连作者本人所举的例子，都是平铺直叙，毫无技巧可言。

（更正，在序言中我曾提到有本最近出版的”禅一汇编语言”，是唯一的例外，希望读者不要错过。）

多年来，我一直想写本有关汇编语言写作技巧的书，可惜都得不到机会。这次，为了实现「整合系统」革命性的计划，所有招收的工程师，一概从头训练。由于没有可用的教材，只好自己动手，于是初步有了讲义，再经修改，便成此书。

我认为，既然汇编语言是种艺术，我们不仅不应该藏私自珍，而且要相互探讨，交流切磋，以期发扬光大。

不过，技术本身与利用该技术所创造的产品却不能混为一谈，产品是借以谋生的工具，能够生存，大家才有研究发展的机会，也才能把成果贡献给社会。如果国人不尊重别人的产品权利，只是互相抄袭盗用，或能受惠于一时，但影响所及，人人贪图现成，不事发展，则观念停顿，技术落伍，其后果不堪设想。

前言

关于本书

本书主要介绍了 Vulkan 的图形和计算 API。Vulkan 是 Khronos 组织 (该组织以 OpenGL 闻名) 发布的新一代图形 API。这一新的 API 可以更好地描述应用程序所需的运算，并且相比于 OpenGL 和 Direct3D，拥有更好的性能，更轻便的驱动程序。Vulkan 在设计上类似于 Direct3D 12 和 Metal，但比之后两者，Vulkan 是跨平台的，可以在 Windows，Linux 和 Android 平台上使用。

使用 Vulkan 不是没有缺点，更精准的控制，意味着更繁琐的细节。我们需要在应用程序中做更多的工作，这包括设置初始时使用的帧缓冲，以及对缓冲和纹理图像的内存管理。

Vulkan 并非适合所有人。它是为追求计算机图形性能极限的狂热分子设计的。如果你对游戏开发更感兴趣，或许 OpenGL 和 Direct3D 更适合你，它们在短期内仍然会是主流，并且目前还有大量的设备尚未支持 Vulkan。除此之外，也可以使用 Unreal Engine 或 Unity 这类引擎，它们可以通过封装好的底层完全透明地使用 Vulkan。

下面是学习本书的一些先决条件：

- 支持 Vulkan 的显卡以及驱动程序 (NVIDIA，AMD，Intel)
- C++ 编程经验 (熟悉 RAII，初始化列表)

- 支持 C++11 的编译器 (Visual Studio 2013+，GCC 4.8+)
- 三维计算机图形学基础

本书不假设读者了解 OpenGL 和 Direct3D，但需要读者了解基本的三维计算机图形学知识。

读者可以使用 C 来替换 C++，但这样做需要读者对我们的代码进行大量修改。我们的代码使用了类，RAII 等 C++ 特性。

电子书

本书有两种格式的电子版提供：

- EPUB
- PDF

教程结构

我们首先简要介绍 Vulkan 的工作原理，然后介绍如何使用 Vulkan 在屏幕上绘制一个三角形。接着，我们介绍如何配置开发环境来使用 Vulkan SDK，在这里，我们还引入了 GLM 库来进行线性代数运算，引入了 GLFW 库来进行窗口的创建。教程包含了在 Windows 上使用 Visual Studio 的配置方法，在 Ubuntu 上使用 GCC 的配置方法。

之后，我们将会实现 Vulkan 绘制三角形的必要部分。每一章节大致遵循下面的结构：

- 介绍新概念，以及它的用途
- 将与之相关的 API 调用集成到我们的程序中去
- 封装辅助函数

尽管章节的组织有一定顺序，但对于部分章节，完全可以独立阅读，作为一个 Vulkan 特性的介绍。也就是说，除了作为教程外，本书可以作为 Vulkan 的一个参考手册，当作字典来查询。书中所有 Vulkan 函数和类型都被超链接到了 Vulkan 规范，可以通过鼠标点击，获取它们更加详细的信息。由于 Vulkan 是一个非常新的 API，它的规范文档可能存在许多不足，读者可以提交反馈给 Khronos 的 Github 仓库。

之前提到，为了更精准地控制硬件，使用 Vulkan 需要处理大量细节。这造成许多很基础的操作也需要编写很多代码才能完成。为了简化这类操作的处理，我们会编写一些辅助函数。

每一章节也都包含了总结，以及到此为止的完整代码的超链接。如果读者存在疑惑的地方，可以参考这些代码。这些代码经过了多个不同厂商的显卡测试。

Vulkan 是一个非常新的图形 API，有关它的最佳实践尚未建立。如果你对本书有任何建议，可以提交反馈到 Github 仓库。

完成使用 Vulkan 绘制三角形的程序后，我们将对其进行扩展，引入线性变换，纹理和三维模型。

如果读者在之前使用过图形 API，应该知道在几何体显示到屏幕之前，需要经过很多步骤。使用 Vulkan 同样是这样，但这些步骤很容易理解，并且每一步都不多余。绘制三维模型采取的步骤并不比绘制三角形所采取的步骤多很多。

如果在实践本书的过程中遇到问题，读者可以先查看本书的 FAQ，是否存在关于这一问题的解决方案，没有的话，欢迎在相关章节进行评论来寻求帮助。

准备好体验高性能次世代图形 API 了吗？让我们开始吧！

概述

本章节首先简要介绍了 Vulkan，以及它所解决的问题。然后，我们将会看到如何使用 Vulkan 来绘制一个三角形，建立 Vulkan 的基本使用思路。最后，我们将会介绍 Vulkan API 的基本结构和使用方式。

Vulkan 起源

Vulkan 是作为一个跨平台的图形 API 设计的。以往许多图形 API 采用固定功能渲染管线设计，应用程序按照一定格式提交顶点数据，配置光照和着色选项。

随着显卡架构逐渐成熟，提供了越来越多的可编程功能，这些功能被集成到原有的 API 中。造成驱动程序要做的工作越来越复杂，应用程序开发者要处理的兼容性问题也越来越多。随着移动浪潮到来，人们对移动 GPU 的要求也越来越高，但以往的图形 API 不能够进行更加精准地控制来提升效率，对多线程的支持也非常不足，导致没有发挥出图形硬件真正的潜力。

由于没有历史包袱，Vulkan 完全按照现代图形架构设计，提供了更加详细的 API 给开发者，大大减少了驱动程序的开销，允许多个线程并行创建和提交指令，使用标准化的着色器字节码，将图形和计算功能进行统一。

画一个三角形

现在，让我们来看下如何使用 Vulkan 绘制三角形。这里用到的所有概念会在下一章节进行详细地说明。

步骤 **1**：实例和物理设备选择

我们的应用程序是通过 VkInstance 来使用 Vulkan API 的。应用程序创建 VkInstance 后,就可以查询 Vulkan 支持的硬件,选择其中一个或多个 VkPhysicalDevices 进行操作。我们可以通过查询设备属性，选择一个适合我们的设备。

步骤 **2**：逻辑设备和队列族

选择完合适的硬件设备后，我们需要使用更详细的 VkPhysicalDevice 特性 (比如多视口，64 位浮点) 来创建一个逻辑设备 VkDevice。还需要指定我们想要使用的队列族。Vulkan 将诸如绘制指令、内存操作提交到 VkQueue 中，进行异步执行。队列由队列族分配，每个队列族支持一个特定操作集合。比如，图形，计算和内存传输操作可以使用独立的队列族。队列族可以作为物理设备选择时的一个参考。比如，一个

支持 Vulkan 的设备可能没有提供任何图形功能，但对于支持 Vulkan 的显卡设备而言，支持所有队列操作。

步骤 **3**：窗口表面和交换链

如果不是进行离屏渲染，通常我们需要创建一个窗口来显示渲染的图像。这一工作可以通过原生平台的窗口 API 或像 GLFW 或 SDL 这样的库来完成，在这里，我们使用的是 GLFW，有关 GLFW 的更多信息，我们会在下一章介绍。

我们还需要两个组件才能完成窗口渲染：窗口表面 (VkSurfaceKHR) 和交换链 (VkSwapChainKHR)。可以注意到这两个组件都有一个 KHR 后缀，这表示它们属于 Vulkan 扩展。Vulkan API 本身是完全平台无关的，需要我们使用 WSI(Window System Interface,窗口系统接口) 扩展与原生的窗口管理器进行交互。表面 (Surface) 是一个跨平台抽象，通常它是由原生窗口系统句柄作为参数实例化得到。不过，这一部分工作，GLFW 已经帮我们处理了，所以不用我们关心。

交换链是一个渲染目标集合。它可以保证我们正在渲染的图像和当前屏幕图像是两个不同的图像。这可以确保显示出来的图像是完整的。每次绘制一帧时，可以请求交换链提供一张图像。绘制完成后，图像被返回到交换链中，在之后某个时刻，图像被显示到屏幕上。渲染目标数量和图像显示到屏幕的时机依赖于显示模式。常用的显示模式有双缓冲 (vsync，垂直同步) 和三缓冲。我们将在创建交换链章节讨论这些问题。

步骤 **4**：图像视图和帧缓冲

从交换链获取图像后，还不能直接在图像上进行绘制，需要将图像先包装进 VkImageView 和 VkFramebuffer 中去。一个图像视图可以引用图像的特定部分，一个帧缓冲可以引用图像视图作为颜色，深度和模板目标。交换链中可能有多个不同的图像，我们可以预先为它们每一个都创建好图像视图和帧缓冲，然后在绘制时选择对应的那个。

步骤 **5**：渲染流程

渲染流程描述了渲染操作使用的图像类型，图像的使用方式，图像的内容如何处理。对于我们这个绘制三角形的程序，我们使用了一张图像作为颜色目标，在执行绘制操作前清除整个图像。渲染流程只描述了图像的类型，图像绑定是通过 VkFramebuffer 完成的。

步骤 **6**：图形管线

Vulkan 的图形管线可以通过 VkPipeline 对象建立。它描述了显卡的可配置状态，比如视口大小和深度缓冲操作，以及使用 VkShaderModule 对象的可编程状态。VkShaderModule 对象由着色器字节码创建而来。驱动程序知道哪些渲染目标被图形管线使用。

Vulkan 与之前的图形 API 的一个最大不同是几乎所有图形管线的配置都需要提前完成。这意味着如果我们想要使用另外一个着色器或者顶点布局，需要重新创建整个图形管线。显然效率很低，这迫使我们提前创建出所有我们需要的图形管线，在需要时直接使用已经创建好的图形管线。图形管线只有很少一部分配置可以动态修改，比如视口大小和清除颜色。图形管线的所有状态也需要显式地描述，比如，不存在默认的颜色混合状态。

这样做的好处类似于预编译相比于即时编译，驱动程序可以有更大的优化空间，并且以图形管线为切换单位，渲染效果的预期也变得十分容易，不用担心切换时，遗漏某个微小的设置，造成结果的巨大差异。

步骤 **7**：指令池和指令缓冲

之前提到，Vulkan 的许多操作需要提交到队列才能执行。这些操作首先被记录到一个 VkCommandBuffer 对象中，然后提交给队列。VkCommandBuffer 对象由一个关联了特定队列族的 VkCommandPool 分配而来。为了绘制三角形，我们需要记录下列操作到 VkCommandBuffer 对象中去：

- 开始渲染
- 绑定图形管线
- 绘制三角形
- 结束渲染

由于帧缓冲绑定的图像依赖于交换链给我们的图像，我们可以提前为每个图像建立指令缓冲，然后在绘制时，直接选择对应的指令缓冲使用。当然在每一帧记录指令缓冲也是可以的，但这样做效率很低。

步骤 **8**：主循环

将绘制指令包装进指令缓冲后，主循环变得非常直白。我们首先使用 vkAcquireNextImageKHR 函数从交换链获取一张图像。接着使用 vkQueueSubmit 函数提交图像对应的指令缓冲。最后，使用 vkQueuePresentKHR 函数将图像返回给交换链，显示图像到屏幕。

提交给队列的操作会被异步执行。我们需要采取同步措施比如信号量来确保操作按正确的顺序执行。绘制指令的执行必须在获取图像之后，否则，可能会出现读写冲突，屏幕正在读取图像数据的同时，绘制操作在进行绘制操作，造成屏幕读取显示的数据并非来自同一帧。同样，vkQueuePresentKHR 函数调用需要在绘制完成后进行。

总结

本章节通过绘制一个简单的三角形来使读者建立 Vulkan 的基本使用思路。通常，一个真正实用的程序会包含更多的步骤，比如分配顶点缓冲，创建 Uniform 缓冲，上传纹理图像等等。但为了降低学习难度，我们从最简单的形式开始，逐步复杂化。

对于绘制一个三角形，我们需要采取的步骤包括：

- 创建一个 VkInstance
- 选择一个支持 Vulkan 的图形设备 (VkPhysicalDevice)
- 为绘制和显示操作创建 VkDevice 和 VkQueue
- 创建一个窗口，窗口表面和交换链
- 将交换链图像包装进 VkImageView
- 创建一个渲染层指定渲染目标和使用方式

- 为渲染层创建帧缓冲
- 配置图形管线
- 为每一个交换链图像分配指令缓冲
- 从交换链获取图像进行绘制操作，提交图像对应的指令缓冲，返回图像到交换链

看起来步骤非常多，但其实每一步都非常简单。在接下来的章节，我们会对每一步进行非常详细地说明。如果你对程序中的某一步感到困惑，可以回来参考一下本章节。

API 概念

本小节对 Vulkan API 的结构进行简要的介绍。

编码约定

Vulkan 的所有函数、枚举和结构体都被定义在 vulkan.h 中，我们可以在 Vulkan SDK 中找到这一头文件。在下一章节，我们会介绍如何安装 Vulkan SDK。

Vulkan API 的函数都带有一个小写的 vk 前缀，枚举和结构体名带有一个 Vk 前缀，枚举值带有一个 VK_ 前缀。Vulkan 对结构体非常依赖，大量函数的参数由结构体提供。比如，Vulkan 创建对象的一般形式如下：

```
VkXXXCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr;
createInfo.foo = ...;
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

Vulkan 的许多结构体需要我们通过设置 sType 成员变量来显式指定结构体类型。结构体的 pNext 成员可以指向一个扩展的结构体，在本教程，我们不使用它，它被设置为 nullptr。Vulkan 中创建和销毁对象的函数都有一个 VkAllocationCallbacks 参数，可以被用来自定义内存分配器，在这里，我们不使用它，将其设置为 nullptr。

几乎所有 Vulkan 都会返回一个 VkResult 来表示调用的执行情况，它的值要么是 VK_SUCCESS，要么是一个错误代码。Vulkan 规范文档描述了这些函数返回的错误代码的意义。

校验层

之前提到，Vulkan 的设计目标是高性能、低驱动程序开销。所以，默认情况下，它提供的错误检测和调试功能非常有限。驱动程序会在发生错误时直接崩溃，而不是返

回一个错误代码。这可能导致对于某种显卡可以工作，不会崩溃，但对于其它显卡无法工作，驱动程序崩溃。

可以通过 Vulkan 的校验层 (Validation layers) 特性来进行一定的错误检查措施。校验层是一段被插入在 Vulkan API 和驱动程序之间的代码，可以对 Vulkan API 函数的参数进行检查，跟踪内存分配。我们可以在开发期开启校验层，然后在发布程序时关闭校验层，减少性能损失。校验层可以完全自己编写，但为了方便，我们的教程直接使用了 Vulkan SDK 提供的一组校验层。我们通过注册的回调函数来接受来自校验层的调试信息。

由于 Vulkan 的每个操作都要显式定义，加之校验层的使用，调试使用 Vulkan 的程序要比调试使用 OpenGL 和 Direct3D 的程序轻松太多。

接下来让我们配置开发环境，开始我们的 Vulkan 编程之旅吧！

开发环境

在本章节，我们将介绍如何配置 Vulkan SDK 和一些非常有用的库。所有在这里用到的工具除了编译器外，适用于 Windows，Linux 和 MacOS 三个平台，但它们的安装方法可能在不同平台会有所不同，所以在这里我们按平台分别描述如何配置它们。

Windows

我们这里使用 Visual Studio 2017 作为 Windows 平台的开发环境，当然使用 Visual Studio 2013 或 2015 应该也不会有任何问题，只是配置方法可能会略微不同。

Vulkan SDK

Vulkan SDK 是使用 Vulkan 开发应用程序必不可少的组件。它包含了 Vulkan API 的头文件，一个校验层实现，调试工具和 Vulkan 函数加载器。Vulkan 函数加载器类似 OpenGL 的 GLEW 可以在运行时查询驱动程序支持的 Vulkan API 函数。

Vulkan SDK 可以从 LunarG 的网站上免费下载。

Figure 1: image

安装 Vulkan SDK 后，我们需要验证下我们的显卡和驱动程序是否支持 Vulkan。这可以通过运行 Vulkan SDK 自带的 cube.exe 来完成，我们可以在 Vulkan SDK 安装目录下的 Bin 目录下找到它，运行后，可以看下到下面的窗口：

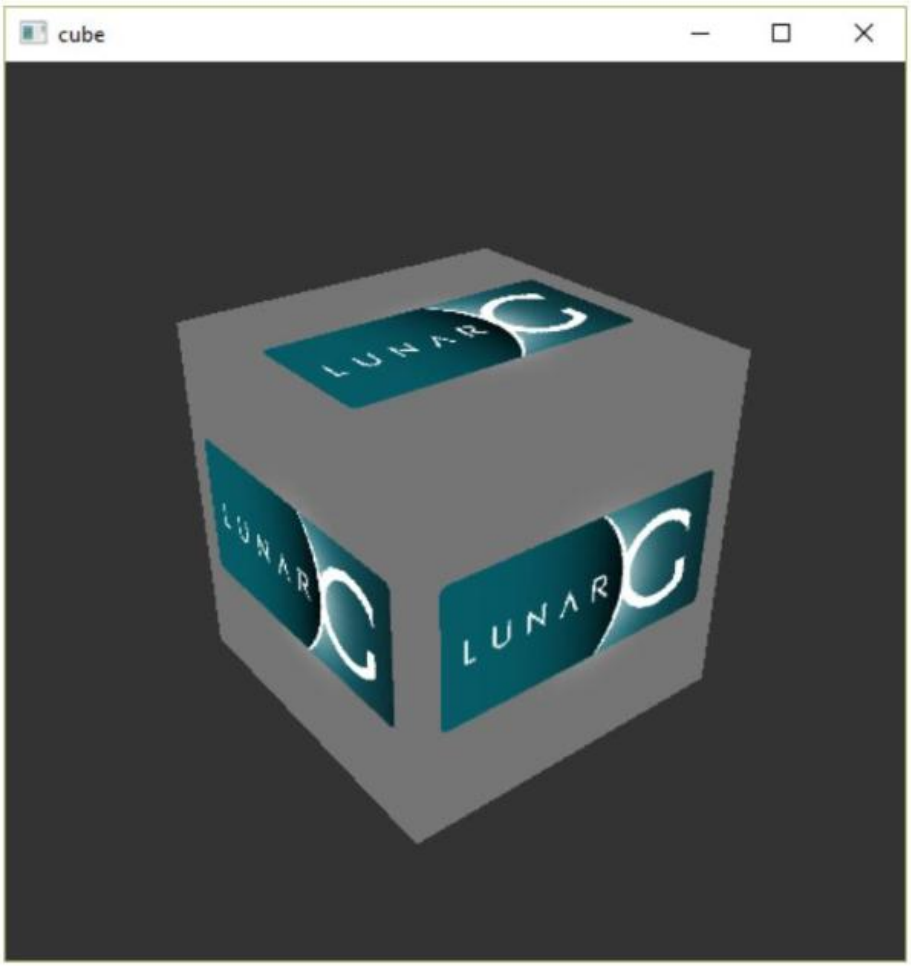

Figure 2: image

如果没有看到这个窗口，而是出现了一条错误消息，可以尝试更新显卡的驱动程序到最新版本，再次尝试，如果仍然出现错误消息，可以在显卡官网查询自己的显卡是否支持 Vukan。

在 Bin 目录下还有一个非常有用的程序：glslangValidator.exe。它可以将 GLSL 代码编译为字节码。我们会在着色器模块章节，对它进行更为详细地说明。除此之外，Bin 目录下还包含了 Vulkan 函数加载器和校验层的二进制文件，它们的库文件则位于 Vulkan SDK 的 Lib 目录下。

Vulkan SDK 的 Documentation 目录包含了 Vulkan SDK 的离线文档和完整的 Vulkan 规范文档。最后是 Vulkan SDK 的 Include 目录，它包含了 Vulkan API 的头文件。除此之外，还有很多文件和目录，但对于我们的教程来说，并没有直接用到它们，所以就不再一一介绍。

GLFW

之前提到，Vulkan 是一个平台无关的图形 API，它没有包含任何用于创建窗口的功能。为了跨平台和避免陷入 Win32 的窗口细节中去，我们使用 GLFW 库来完成窗口相关操作，GLFW 库支持 Windows，Linux 和 MacOS。当然，还有其它一些库可以完成类似功能，比如 SDL。但除了窗口相关处理，GLFW 库对于 Vulkan 的使用还有其它一些优点。

读者可以再 GLFW 的官方网站上免费下载到最新版本的 GLFW 库。在本教程，我们使用 64 位版本的 GLFW 库，但 32 位版本也是可以的，只是编译使用 Vulkan 的应用程序时也需要链接到 32 位版本的 Vulkan API，也就是链接到 Vulkan SDK 下 Lib32 目录下的库。下载 GLFW 后，将它解压缩到一个合适的位置。这里，我们将它解压到 Visual Studio 目录下的 Libraries 目录中。不要纠结于为什么解压后不存在 libvc-2017 目录，我们的 Visual Studio 2017 是完全兼容 lib-vc2015 的。

GLM

和 DirectX 12 不同，Vulkan 没有包含线性代数库，我们需要自己找一个。GLM 就是一个我们需要的线性代数库，它经常和 OpenGL 一块使用。

GLM 是一个只有头文件的库，我们只需要下载它的最新版，然后将它放在一个合适的位置，就可以通过包含头文件的方式使用它。

配置 Visual Studio

现在，我们可以创建一个基本 Visual Studio 工程来验证我们安装的依赖是否可以正常工作。

首先，启动 Visual Studio，然后选择 Windows Destop Wizard。

我们选择使用 Console Application (.exe) 应用程序类型，这样做我们就可以直接将调试信息输出到控制台窗口上。另外，我们将 Empty Project 选项打勾来阻止 Visual Studo 添加模板代码。

创建项目后，我们添加一个 C++ 源代码文件到项目中。

下面的代码是这个 C++ 源文件的内容。源代码的内容暂时不需要理解，我们现在只是为了验证我们的依赖是否配置正确，源代码的内容，我们会在后面的章节详细说明。

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
```

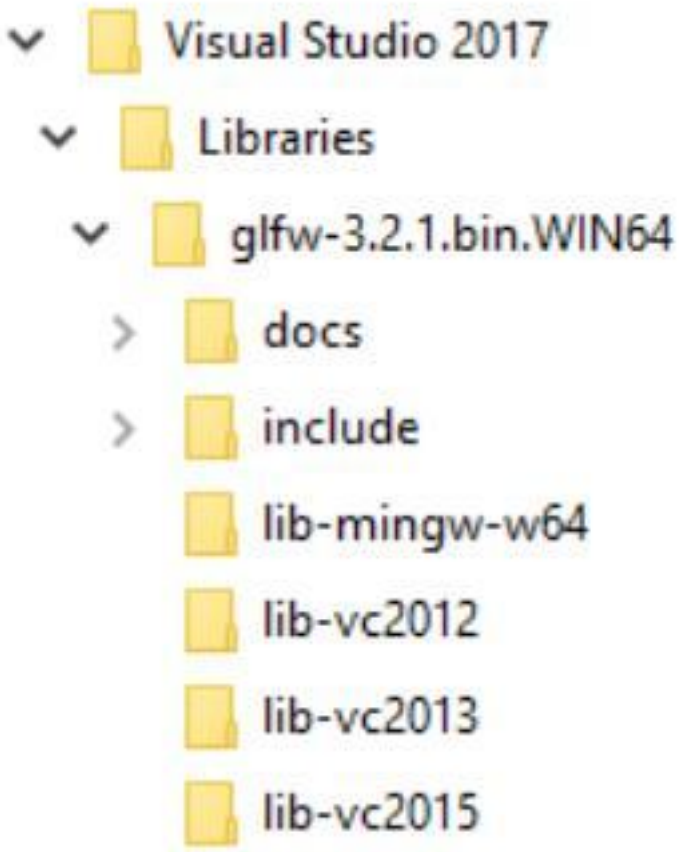

Figure 3: image

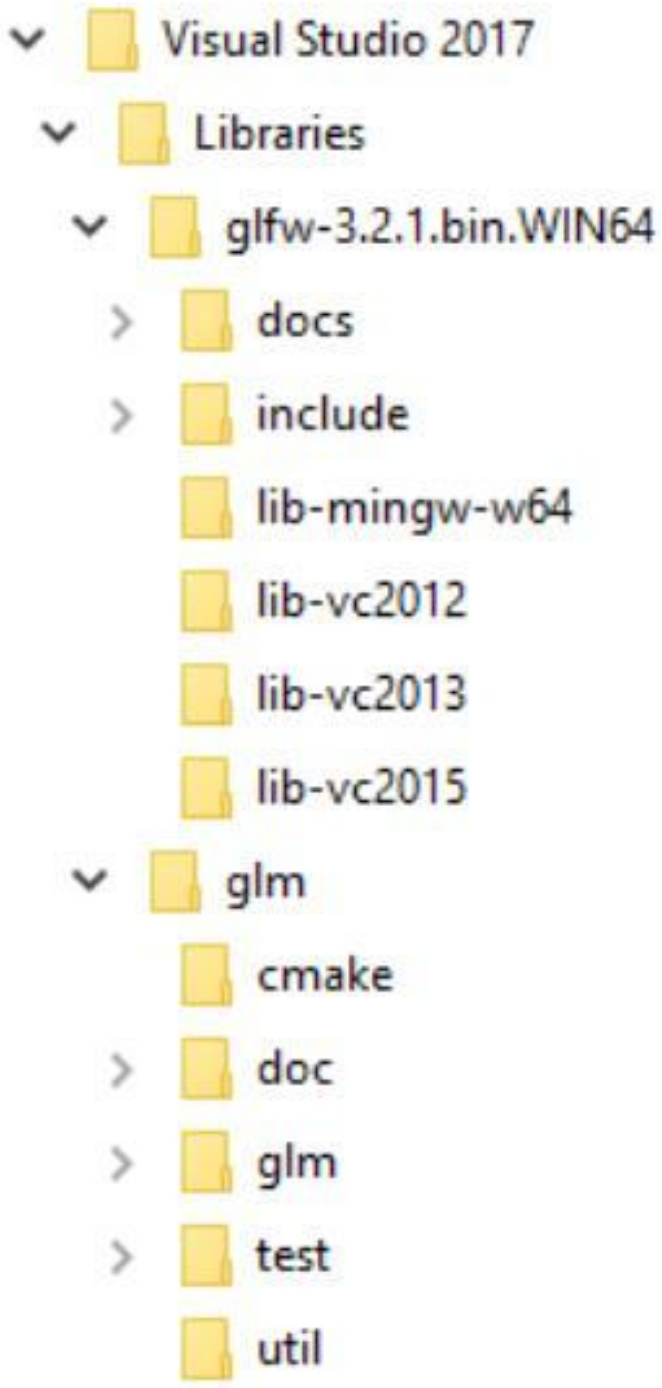

Figure 4: image

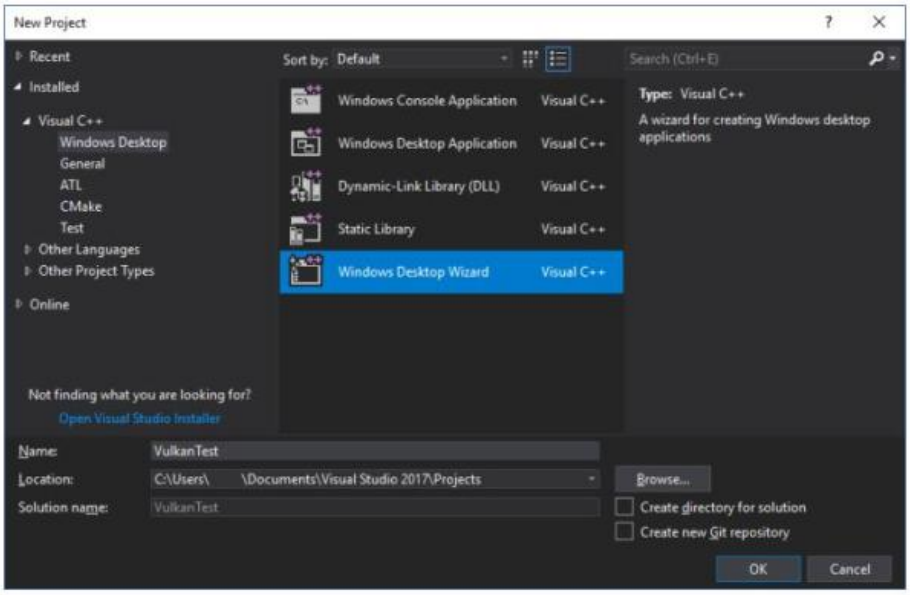

Figure 5: image

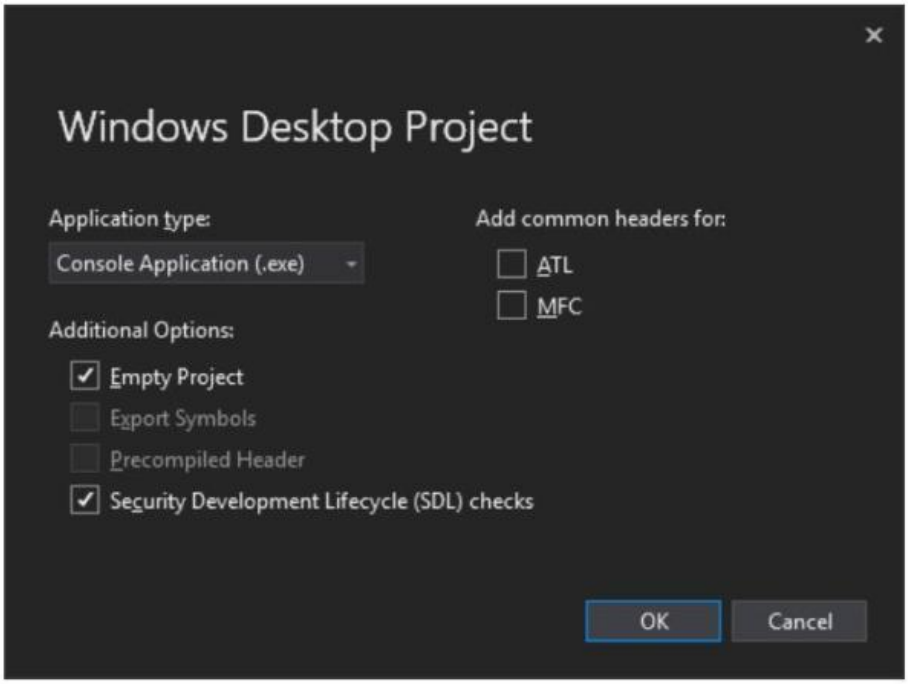

Figure 6: image

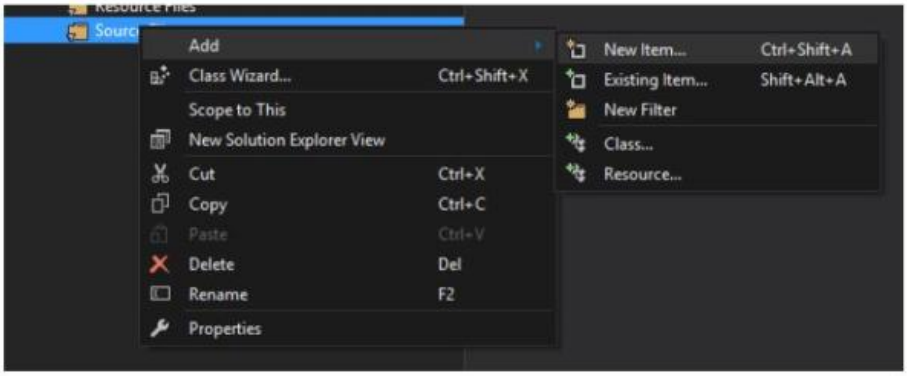

Figure 7: image

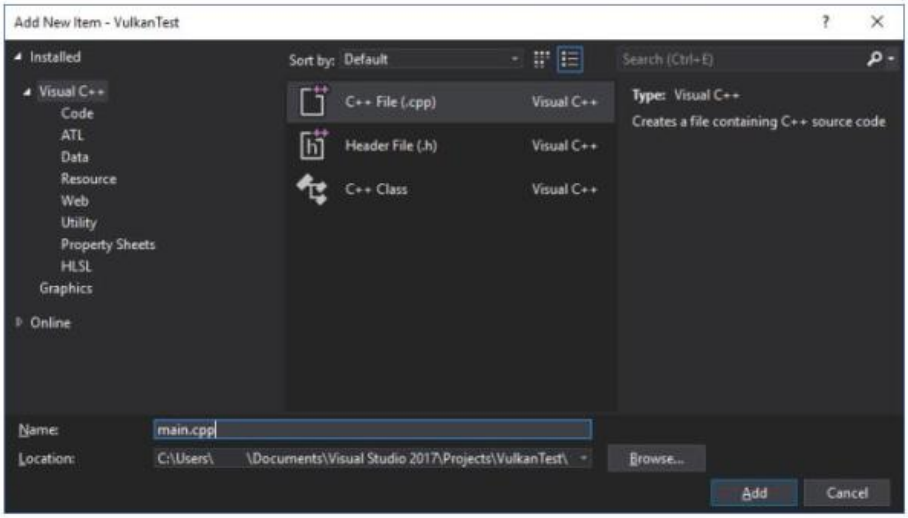

Figure 8: image

```cpp
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr,
                &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported" << std::endl;

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
```

现在，让我们开始配置项目属性，选择 All Configurations，让设置对 Debug 和 Release 模式都有效。

打开 C++ -> General -> Additional Include Directories，点击 Additional Include Directories 的 <Edit...> 下拉选项。

添加 Vulkan，GLFW 和 GLM 的头文件目录：

接着，打开 Linker -> General：

添加 Vulkan 和 GLFW 的库目录：

打开 Linker -> Input，点击 Additional Dependencies 的 <Edit...> 下拉选项：

添加 Vulkan 和 GLFW 的库文件：

现在可以关闭项目属性对话框了。如果一切顺利，我们的代码编辑器里已经没有任何高亮出的错误代码了。

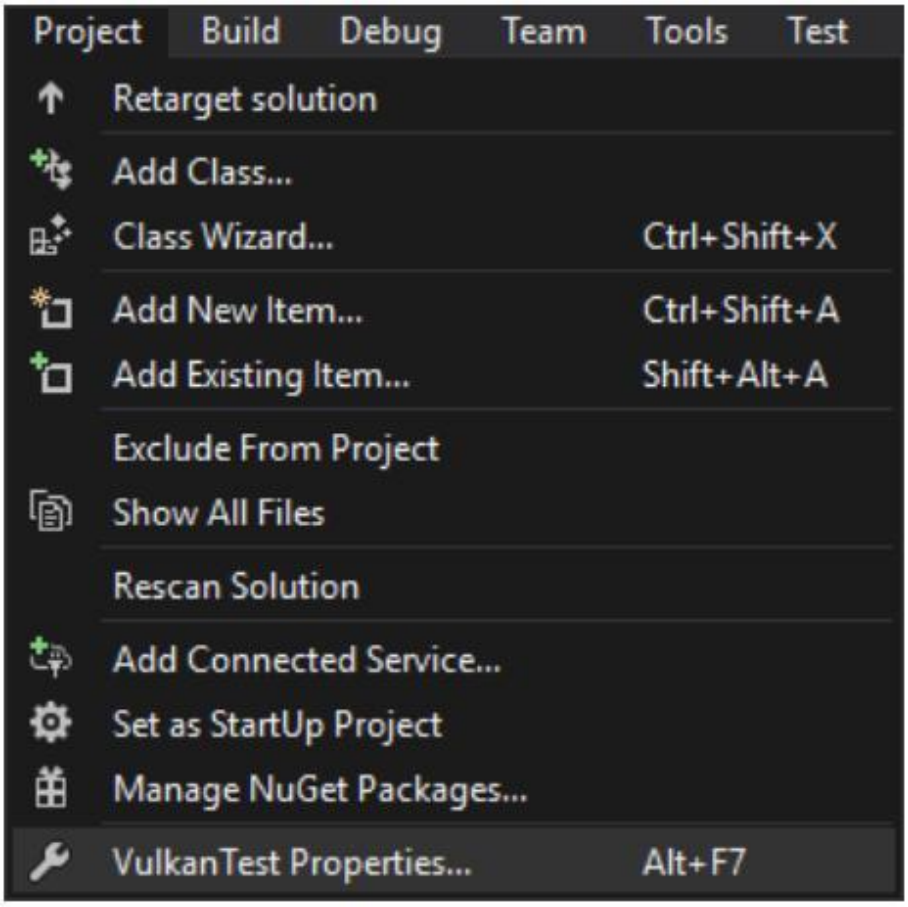

Figure 9: image

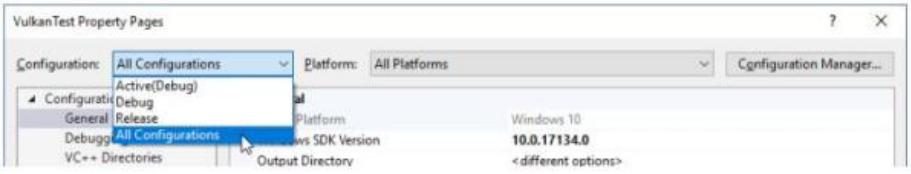

Figure 10: image

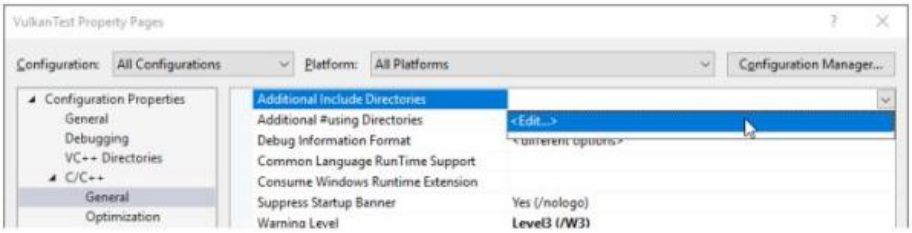

Figure 11: image

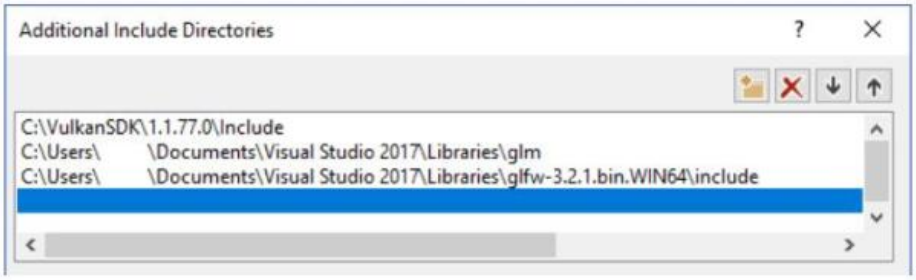

Figure 12: image

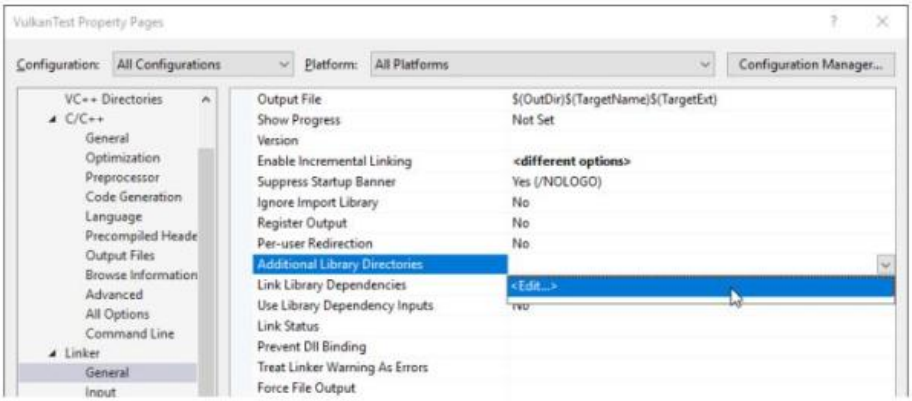

Figure 13: image

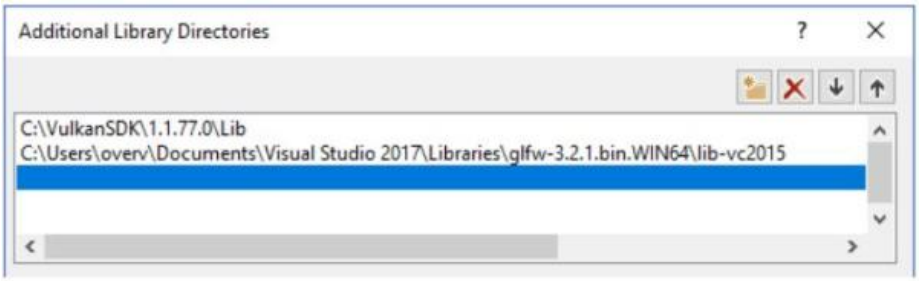

Figure 14: image

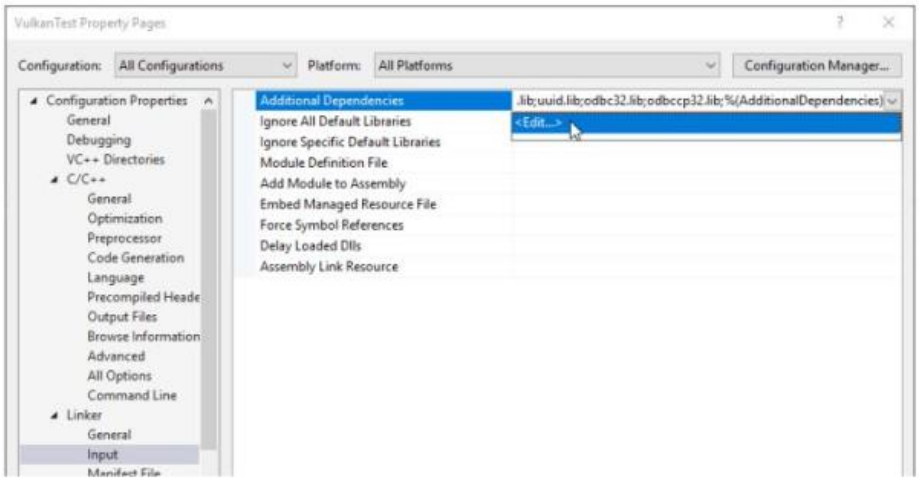

Figure 15: image

Figure 16: image

最后，确认我们的代码在 64 位模式下编译：

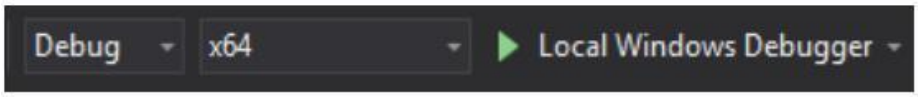

Figure 17: image

然后，按下 F5 编译运行，你就会看到下面的窗口：

Figure 18: image

控制台窗口显示的扩展数应该是非 0 的。至此，我们就配置好了 Vulkan 的开发环境！

Linux

对于 Linux 平台的配置，我们使用 Ubuntu 作为演示，其它 Linux 平台的配置方法应该是类似的。我们使用二进制包来安装 Vulkan SDK，使用 GCC 4.8 以上版本作为编译器，使用 CMake 和 make 作为构建系统。

Vulkan SDK

Vulkan SDK 是使用 Vulkan 开发应用程序必不可少的组件。它包含了 Vulkan API 的头文件，一个校验层实现，调试工具和 Vulkan 函数加载器。Vulkan 函数加载器类似 OpenGL 的 GLEW 可以在运行时查询驱动程序支持的 Vulkan API 函数。

Vulkan SDK 可以从 LunarG 的网站上免费下载。

Figure 19: image

打开终端，调整当前目录到我们下载的 Vulkan SDK 安装文件所在目录，然后使用下面的代码运行它：

```
chmod +x vulkansdk-linux-x86_64-xxx.run
./vulkansdk-linux-x86_64-xxx.run
```

安装文件运行后会将 Vulkan SDK 的所有文件导出到当前目录的 VulkanSDK 文件夹中。我们可以自己将 VulkanSDK 文件夹移动到合适的位置。

Vulkan SDK 的根目录有一个 build_examples.sh 脚本，执行它构建 Vulkan SDK 的示例程序需要我们安装 XCB 库，以及一些 X 窗口的开发文件，可以通过在终端运行下面的代码来安装这些所需的库：

```
sudo apt install libxcb1-dev xorg-dev
```

然后，我们就可以执行 build_examples.sh 了：

```
./build_examples.sh
```

如果编译成功，在./examples/build/下就会出现一个 cube 可执行文件，运行它，可以看到下面的画面：

如果没有看到，而是出现了一条错误消息，可以尝试更新显卡的驱动程序到最新版本，再次尝试，如果仍然出现错误消息，可以在显卡官网查询自己的显卡是否支持 Vukan。

GLFW

之前提到，Vulkan 是一个平台无关的图形 API，它没有包含任何用于创建窗口的功能。为了跨平台和避免陷入 X11 的窗口细节中去，我们使用 GLFW 库来完成窗口相关操作，GLFW 库支持 Windows，Linux 和 MacOS。当然，还有其它一些库可以完成类似功能，比如 SDL。但除了窗口相关处理，GLFW 库对于 Vulkan 的使用还有其它一些优点。

这里，由于 Vulkan 需要较新版本的 GLFW 才能支持。所以，我们使用源代码来编译安装 GLFW。读者可以从 GLFW 的官方网站免费下载到 GLFW 的最新源码包。下载完成后，我们将源码包解压，使用终端进入解压的源码所在的文件夹，执行下面的代码生成 makefile 文件，然后编译 GLFW：

```
cmake .
make
```

可能会出现 Could NOT find Vulkan 的警告信息，可以放心地忽略掉它。编译完成后，使用下面的代码将 GLFW 安装到系统的库目录中：

```
sudo make install
```

GLM

和 DirectX 12 不同，Vulkan 没有包含线性代数库，我们需要自己找一个。GLM 就是一个我们需要的线性代数库，它经常和 OpenGL 一块使用。

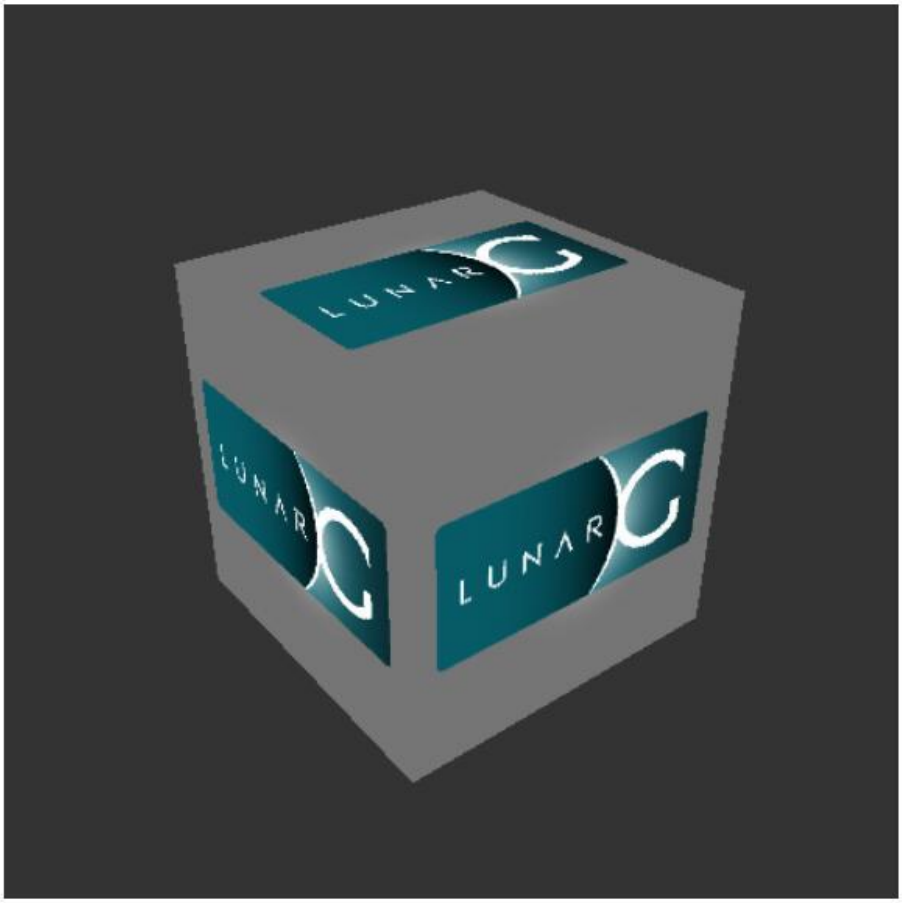

Figure 20: image

GLM 是一个只有头文件的库，我们只需要下载它的最新版，然后将它放在一个合适的位置，就可以通过包含头文件的方式使用它。

这里我们直接在终端使用下面的代码安装它：

```
sudo apt install libglm-dev
```

配置 **makefile** 文件

现在，我们已经安装完了所有的依赖项，可以开始配置应用程序的 makefile，验证安装是否正确。

在一个合适的位置新建一个叫做 VulkanTest 的文件夹，然后在文件夹里创建包含下面代码的 main.cpp 源代码文件。

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan
window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr,
                &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported" << std::endl;

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);
```

```
    glfwTerminate();

    return 0;
}
```

源代码的内容暂时不需要理解，我们现在只是为了验证我们的依赖是否配置正确，源代码的内容，我们会在后面的章节详细说明。

接着，我们需要编写 makefile 来编译源代码。这里假设读者具有一定 makefile 使用经验，知道 makefile 的变量和规则的用法。如果没有，也可以从本教程中快速学习这些知识。

我们首先定义一些变量来简化 makefile 编写。VULKAN_SDKPATH 变量存放了 Vulkan SDK 的 x86_64 目录的位置：

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
```

读者应该替换上面代码的路径为自己 Vulkan SDK 的实际路径。接着，我们定义 CFLAGS 变量来指定编译选项：

```
CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include
```

上面的代码表示使用 C++ 11 来编译源代码，将 Vulkan SDK 的包含目录加入编译器的包含目录搜索路径中。

然后，定义 LDFLAGS 变量来指定链接选项：

```
LDFLAGS = -L$(VULKAN_SDK_PATH)/lib `pkg-config --static --libs glfw3` -lvulkan
```

上面的代码将 Vulkan SDK 的库路径加入链接器的库搜索路径中，链接了 Vulkan SDK 的 vulkan 库，使用 pkg-config 命令取得了 glfw 静态链接选项。现在可以开始定义编译 VulkanTest 的规则了：

```
VulkanTest: main.cpp
    g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)
```

验证规则是否正确，可以将上面的代码保存为 Makefile 文件，然后使用终端在 Makefile 文件所在目录执行 make 命令。如果一切顺利，会生成一个 VulkanTest 可执行文件。

现在，我们定义另外两个规则，test 和 clean，前一个规则用于执行生成的可执行文件，后一个规则用于清除生成的可执行文件：

```
.PHONY: test clean

test: VulkanTest
    ./VulkanTest

clean:
    rm -f VulkanTest
```

验证规则能否执行后，读者可能会发现 make clean 工作的非常好，但 make test 却产生了下面的错误信息：

```
./VulkanTest: error while loading shared libraries:
libvulkan.so.1: cannot open shared object file: No such file or directory
```

这是因为 libvulkan.so 没有被安装在系统的库目录，无法被 VulkanTest 加载。我们可以通过 LD_LIBRARY_PATH 环境变量显式指定库目录来解决这个问题：

```
test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib ./VulkanTest
```

现在 make test 应该可以成功执行 VulkanTest 了。

```
test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib\
    VK_LAYER_PATH=$(VULKAN_SDK_PATH)/etc/explicit_layer.d\
    ./VulkanTest
```

至此，我们的 Makefile 文件已经编写完毕了，它的所有内容如下：

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include
LDFLAGS = -L$(VULKAN_SDK_PATH)/lib `pkg-config --static --libs glfw3` -lvulkan

VulkanTest: main.cpp
    g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)

.PHONY: test clean

test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib\
    VK_LAYER_PATH=$(VULKAN_SDK_PATH)/etc/ex plicit_layer.d\
    ./VulkanTest

clean:
    rm -f VulkanTest
```

读者可以将刚刚配置的 Makefile 文件作为一个模板在以后使用。

现在，让我们花点实践浏览下 Vulkan SDK 目录，在 x86_64/bin 目录下还有一个非常有用的程序：glslangValidator。它可以将 GLSL 代码编译为字节码。我们会在着色器模块章节，对它进行更为详细地说明。除此之外，Bin 目录下还包含了 Vulkan 函数加载器和校验层的二进制文件，它们的库文件则位于 Vulkan SDK 的 Lib 目录下。

Vulkan SDK 的 Doc 目录包含了 Vulkan SDK 的离线文档和完整的 Vulkan 规范文档。最后是 Vulkan SDK 的 Include 目录，它包含了 Vulkan API 的头文件。除此之外，还有很多文件和目录，但对于我们的教程来说，并没有直接用到它们，所以就不再一一介绍。

至此，我们已经做好开始 Vulkan 探险之旅的准备！

MacOS

这里假定读者使用 Xcode 和 Homebrew 包管理器。另外还需要我们的 MacOS 版本在 10.11 以上，显卡设备支持 Metal API。

Vulkan SDK

Vulkan SDK 是使用 Vulkan 开发应用程序必不可少的组件。它包含了 Vulkan API 的头文件，一个校验层实现，调试工具和 Vulkan 函数加载器。Vulkan 函数加载器类似 OpenGL 的 GLEW 可以在运行时查询驱动程序支持的 Vulkan API 函数。

Vulkan SDK 可以从 LunarG 的网站上免费下载。

Figure 21: image

Vulkan SDK 的 MacOS 版本是通过 MoltenVK 实现的，并非原生实现，也就是说 MoltenVK 作为一个中间层将 Vulkan API 调用转换为 Metal 调用。这也使得我们可以直接使用 Metal 的调试功能进行调试。

下载 Vulkan SDK 后，将它解压到一个合适的位置，在解压后的目录中的 Applications，可以找到一些 Vulkan SDK 的演示程序，运行其中的可执行文件 cube，你将会看到下面的窗口：

GLFW

之前提到，Vulkan 是一个平台无关的图形 API，它没有包含任何用于创建窗口的功能。为了跨平台和避免陷入窗口操作相关的细节中去，我们使用 GLFW 库来完成窗口相关操作，GLFW 库支持 Windows，Linux 和 MacOS。当然，还有其它一些库可以完成类似功能，比如 SDL。但除了窗口相关处理，GLFW 库对于 Vulkan 的使用还有其它一些优点。

我们使用 Homebrew 包管理器来安装 GLFW 库。GLFW 的 3.2.1 稳定版目前还尚未完全支持 Vulkan，所以我们使用下面的代码安装 glfw3 包的最新版本：

```
brew install glfw3 --HEAD
```

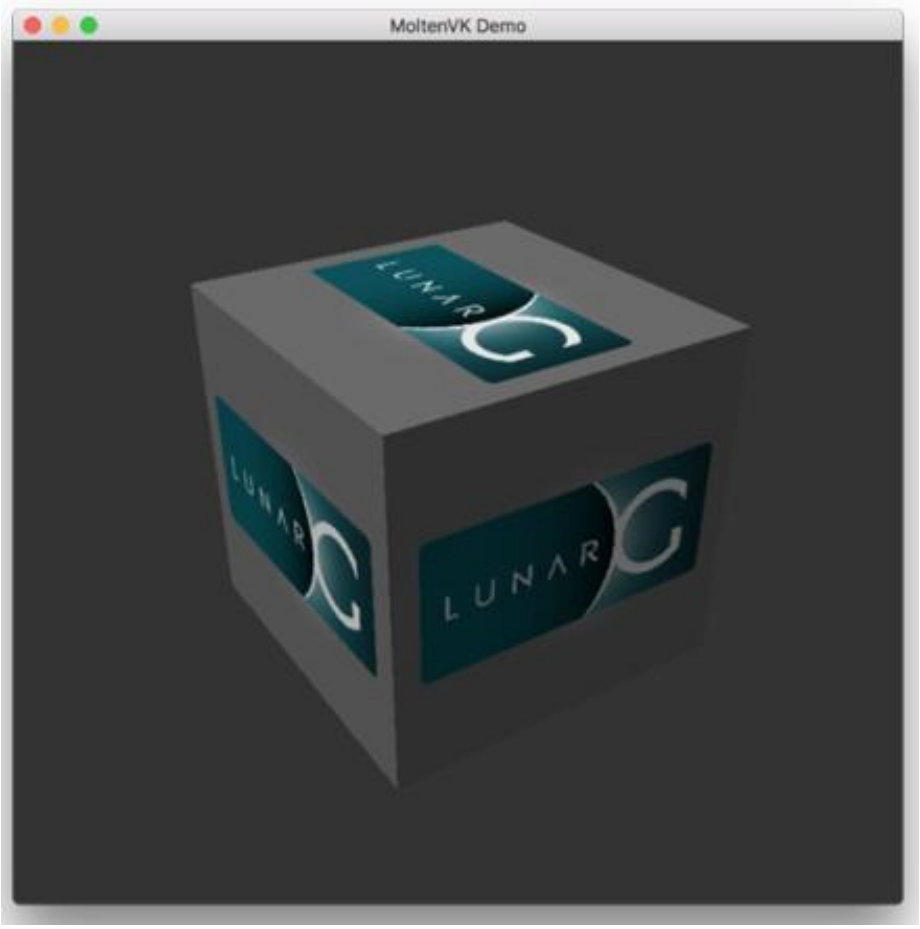

Figure 22: image

GLM

Vulkan 没有包含线性代数库，我们需要自己找一个。GLM 就是一个我们需要的线性代数库，它经常和 OpenGL 一块使用。

GLM 是一个只有头文件的库，我们只需要下载它的最新版，然后将它放在一个合适的位置，就可以通过包含头文件的方式使用它。

这里我们直接在终端使用下面的代码安装它：

```
brew install glm
```

配置 Xcode

现在所有的依赖项已经安装完毕，我们可以开始配置一个最基本的用于 Xcode 的 Vulkan 项目。

启动 Xcode，然后新建一个 Xcode 项目，选择 Application > Command Line Tool 项目类型：

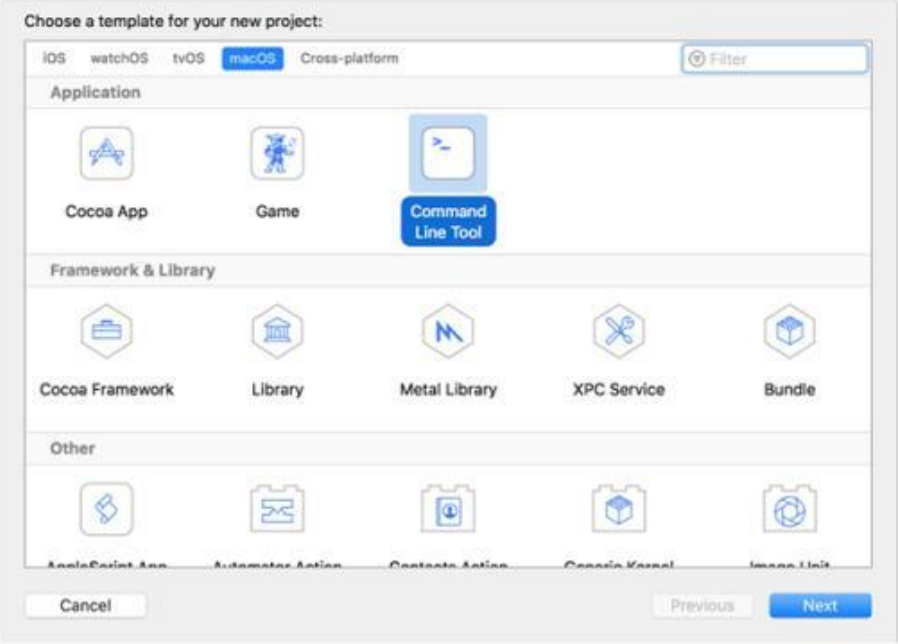

Figure 23: image

接着，选择 C++ 作为项目使用的语言：

现在将下面的代码作为项目的 main.cpp 源文件的内容：

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

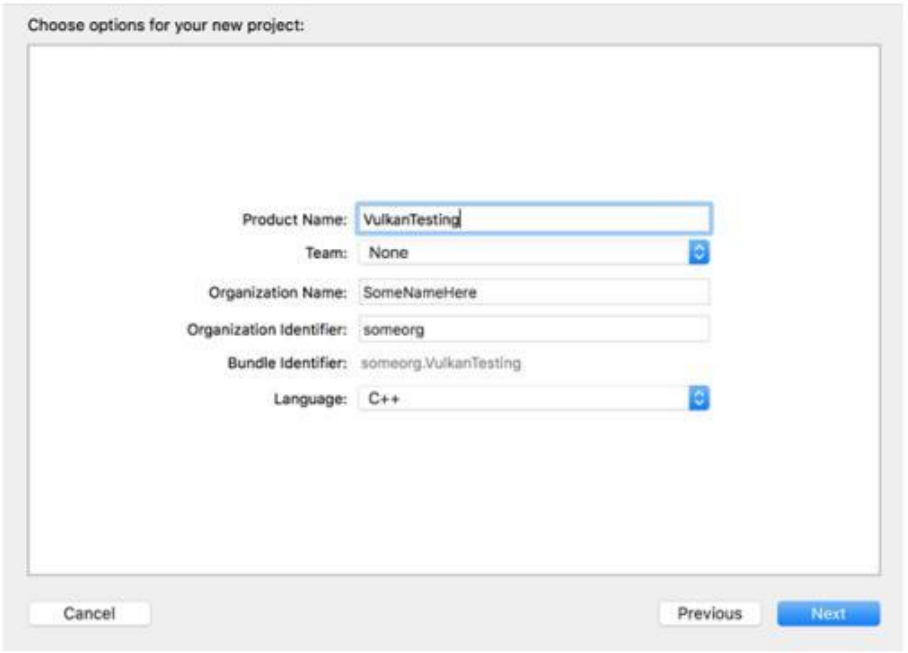

Figure 24: image

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan
window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr,
                &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported" << std::endl;

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
```

源代码的内容暂时不需要理解，我们现在只是为了验证我们的依赖是否配置正确，源代码的内容，我们会在后面的章节详细说明。

现在 Xcode 应该会显示一些诸如库未找到的错误。我们接下来的工作就是解决这些错误。在 Project Navigator 面板选择我们的项目，然后打开 Build Settings 标签页，进行下面的操作：

- 将/usr/local/include 加入 Header Search Paths，这是 Homebrew 安装头文件的路径，我们安装的 glm 和 glfw3 的头文件都在该文件夹下，然后将 vulkansdk/macOS/include 加入 Header Search Paths，vulkansdk 为我们安装的 Vulkan SDK 的目录。这样 Xcode 就可以找到我们使用的库的头文件。

- 将/usr/local/lib 加入 Library Search Paths，这是 Homebrew 安装库文件的路径，我们安装的 glm 和 glfw3 的库文件都在该文件夹下，然后将 vulkansdk/macOS/lib 加入 Library Search Paths，vulkansdk 为我们安装的 Vulkan SDK 的目录。这样 Xcode 就可以找到我们使用的库文件。

设置完成后，看起来像这样 (实际内容依赖于我们自己的文件所在的位置)：

Figure 25: image

现在，点击 Build Phases 标签页，添加 glfw3 和 vulkan 框架。这里，为了简便，我们添加的是动态库 (如果想要使用静态库，可以参考这些库的官方文档)。

- 对于 glfw，打开/usr/local/lib 目录，可以找到类似 libglfw.3.x.dylib 形式的文件 (x 是库的版本号，依赖于我们使用 Homebrew 下载安装的 glfw 的版本)。将这个文件拖拽到 Linked Frameworks and Libraries 标签页即可。
- 对于 Vulkan，打开 vulkansdk/macOS/lib 目录 (vulkansdk 是我们的 Vulkan SDK 所在目录)，拖拽 libvulkan.1.dylib 和 libvulkan.1.x.xx.dylib 文件到 Linked Frameworks and Libraries 标签页即可。

完成上面的操作后，更改 Copy Files 标签页下的 Destination 为 Frameworks，然后清空 Subpath 文本框，去掉勾选 Copy only when installing，点击 + 号，将所有三个动态库添加进去。

最后，我们需要配置环境变量。在 Xcode 的工具栏上通过 Product > Scheme > Edit Scheme... 打开 Arguments 标签页，添加下面的环境变量：

- VK_ICD_FILENAMES = vulkansdk/macOS/etc/vulkan/icd.d/MoltenVK_icd.json
- VK_LAYER_PATH = vulkansdk/macOS/etc/vulkan/explicit_layer.d

完成后，看起来像这样：

至此为止，我们完成了全部设置，可以编译运行项目了，效果如下：

程序输出的日志信息中的扩展数应该是非 0 的。现在，我们已经做好开始 Vulkan 探险之旅的准备！

基础代码

一般结构

在本章节，我们开始使用 Vulkan API 编写代码。

```
#include <vulkan/vulkan.h>
```

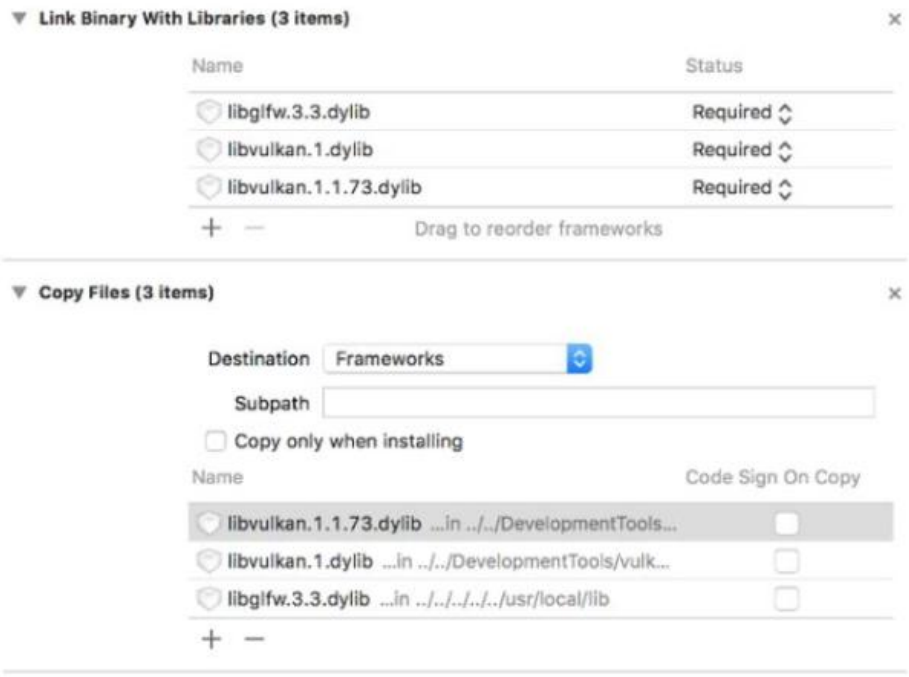

Figure 26: image

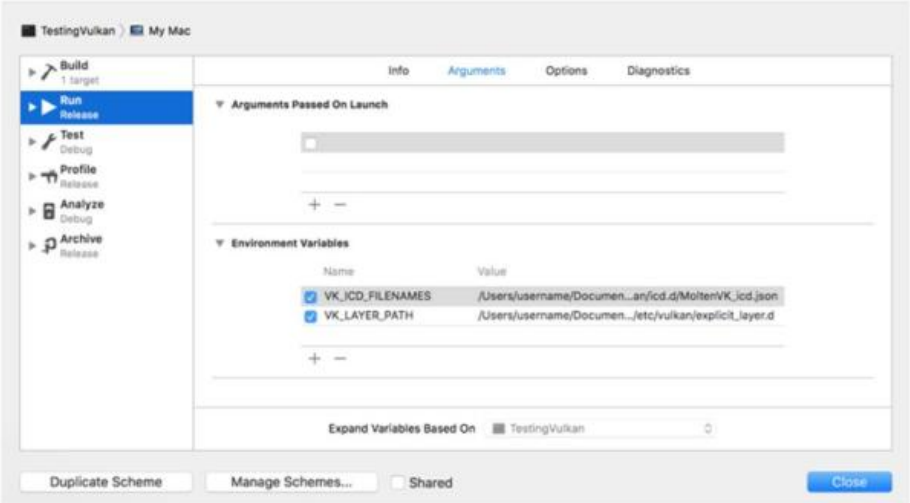

Figure 27: image

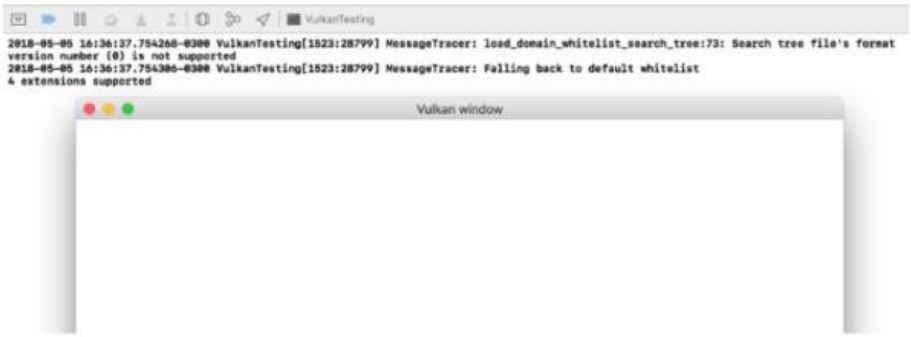

Figure 28: image

```
#include <iostream>
#include <stdexcept>
#include <functional>
#include <cstdlib>

class HelloTriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    void initVulkan() {

    }

    void mainLoop() {

    }

    void cleanup() {

    }
};

int main() {
    HelloTriangleApplication app;

    try {
```

```
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

代码中，我们首先包含了 Vulkan API 的头文件，它为我们提供了 Vulkan API 的函数，结构体和枚举。此外，包含 stdexcept 和 iostream 头文件用来报错。包含 functional 头文件用于资源管理。包含 cstdlib 头文件用来使用 EXITSUCCESS 和 EXIT_FAILURE 宏。

我们将程序本身包装为一个类，将 Vulkan 对象存储为类的私有成员。我们使用 initVulkan 函数来初始化 Vulkan 对象。初始化完成后，我们进入主循环进行渲染操作。mainLoop 函数包含了一个循环，直到窗口被关闭，才会跳出这个循环。mainLoop 函数返回后，我们使用 cleanup 函数完成资源的清理。

程序在执行过程中，如果发生错误，会抛出一个带有错误描述信息的 std::runtime_error 异常，我们在 main 函数捕获这个异常，并将异常的描述信息打印到控制台窗口。为了处理多种不同类型的异常，我们使用更加通用的 std::exception 来接受异常。一个比较常见的异常就是请求的扩展不被支持。

接下来的每一章，我们会添加新的成员到我们的类中，然后在 initVulkan 函数中初始化它们，在 cleanup 函数中清理它们。

资源管理

和使用 malloc 函数分配的内存块相同，使用 Vulkan API 创建的 Vulkan 对象也需要在不需要它们时显式地被清除。现代 C++ 可以通过 <memory> 头文件自动地进行资源管理，但在这里，为了让大家更加清晰地理解 Vulkan 对象地创建和清除，以及它们的生命周期，我们没有使用它，而是手动自己完成资源管理。除此之外，Vulkan 的一个核心思想就是通过显式地定义每一个操作来避免出现不一致的现象。

学完本教程后，读者可以通过重载 std::shared_ptr 来实现自动资源管理。将 RAII 应用到自己的程序中。但对于学习而言，最好是能清晰地理解每一个细节部分。

Vulkan 对象可以直接通过类似 vkCreateXXX 的函数，或是通过其它对象调用类似 vkAllocateXXX 的函数创建。当创建的对象不再使用时，使用对应的 vkDestroyXXX 或 vkFreeXXX 函数进行清除操作。这些函数的参数对于不同类型的对象通常是不同的，但都具有一个 pAllocator 参数。我们可以通过这个参数来指定回调函数编写自己的内存分配器。但在本教程，我们没有使用它，将它设置为 nullptr。

和 GLFW 交互

Vulkan 可以在完全没有窗口的情况下工作，通常，在离屏渲染时会这样做。但一般而言，还是需要一个窗口来显示渲染结果给用户。接下来，我们要完成的就是窗口相关操作。

首先替换代码中的 #include <vulkan/vulkan.h> 为：

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

上面的代码将 GLFW 库的定义包含进来，而 GLFW 库会自动包含 Vulkan 库的头文件。接着，我们添加一个叫做 initWindow 的函数来初始化 GLFW，并在 run 函数中调用它：

```
void run() {
        initWindow();
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    void initWindow() {

    }
```

initWIndow 函数首先调用了 glfwInit 函数来初始化 GLFW 库，由于 GLFW 库最初是为 OpenGL 设计的，所以我们需要显式地设置 GLFW 阻止它自动创建 OpenGL 上下文：

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
```

窗口大小变化地处理需要注意很多地方，我们会在以后介绍它，暂时我们先禁止窗口大小改变：

```
glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

接着，我们添加了一个 GLFWwindow* window 变量存储我们创建的窗口句柄：

```
window =glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

glfwCreateWindow 函数的前三个参数指定了要创建的窗口的宽度，高度和标题。第四个参数用于指定在哪个显示器上打开窗口，最后一个参数与 OpenGL 相关，对我们没有意义。

硬编码窗口大小不是一个好习惯，所以我们定义了两个常量，以便之后可以方便地修改它们：

```
const int WIDTH = 800;
const int HEIGHT = 600;
```

现在，我们地 initWindow 函数看起来应该像这样：

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

```
    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
}
```

为了确保我们的程序在没有发生错误和窗口没有被关闭的情况下可以一直运行，我们在 mainLoop 函数中添加了下面的事件循环：

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }
}
```

上面的代码应该非常直白，每次循环，检测窗口的关闭按钮是否被按下，如果没有被按下，就执行事件处理，否则结束循环。在之后的章节，我们会在这一循环中调用渲染函数来渲染一帧画面。

一旦窗口关闭，我们就可以开始结束 GLFW，然后清除我们自己创建的资源，这在 cleanup 函数中进行：

```
void cleanup() {
    glfwDestroyWindow(window);

    glfwTerminate();
}
```

至此，我们就编写完成了一个可以使用 Vulkan API 的窗口程序骨架。

本章节代码：

C++：

https://vulkan-tutorial.com/code/00_base_code.cpp

实例

创建一个实例

我们首先创建一个实例来初始化 Vulkan 库。这个实例指定了一些驱动程序需要使用的应用程序信息。

我们添加了一个 createInstance 函数调用到 initVulkan 函数中：

```
void initVulkan() {
    createInstance();
}
```

添加了一个存储实例句柄的私有成员：

```
private:
    VkInstance instance;
```

然后，填写应用程序信息，这些信息的填写不是必须的，但填写的信息可能会作为驱动程序的优化依据，让驱动程序进行一些特殊的优化。比如，应用程序使用了某个引擎，驱动程序对这个引擎有一些特殊处理，这时就可能有很大的优化提升：

```
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "Hello Triangle";
appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.pEngineName = "No Engine";
appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.apiVersion = VK_API_VERSION_1_0;
```

之前提到，Vulkan 的很多结构体需要我们显式地在 sType 成员变量中指定结构体的类型。此外，许多 Vulkan 的结构体还有一个 pNext 成员变量，用来指向未来可能扩展的参数信息，现在，我们并没有使用它，将其设置为 nullptr。

Vulkan 倾向于通过结构体传递信息，我们需要填写一个或多个结构体来提供足够的信息创建 Vulkan 实例。下面的这个结构体是必须的，它告诉 Vulkan 的驱动程序需要使用的全局扩展和校验层。全局是指这里的设置对于整个应用程序都有效，而不仅仅对一个设备有效，在之后的章节，我们会对此有更加清晰得认识。

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
```

上面代码中填写得两个参数非常直白，不用多解释。接下来，我们需要指定需要的全局扩展。之前提到，Vulkan 是平台无关的 API，所以需要一个和窗口系统交互的扩展。GLFW 库包含了一个可以返回这一扩展的函数，我们可以直接使用它：

```
uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

结构体的最后两个成员变量用来指定全局校验层。我们将在之后的章节更加深入地讨论它，在这里，我们将其设置为 0，不使用它：

```
createInfo.enabledLayerCount = 0;
```

填写完所有必要的信息，我们就可以调用 vkCreateInstance 函数来创建 Vulkan 实例：

```
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

如你所看到的，创建 Vulkan 对象的函数参数的一般形式就是：

- 一个包含了创建信息的结构体指针

- 一个自定义的分配器回调函数，在本教程，我们没有使用自定义的分配器，总是将它设置为 nullptr
- 一个指向新对象句柄存储位置的指针

如果一切顺利，我们创建的实例的句柄就被存储在了类的 VkInstance 成员变量中。几乎所有 Vulkan API 函数调用都会返回一个 VkResult 来反应函数调用的结果，它的值可以是 VK_SUCESS 表示调用成功，或是一个错误代码，表示调用失败。为了检测实例是否创建成功，我们可以直接将创建函数在条件语句中使用，不需要存储它的返回值：

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```

现在可以编译运行程序来确保实例创建成功。

检测扩展支持

如果读者看过 vkCreateInstance 函数的官方文档，可能会知道它返回的之中一个错误代码 VK_ERROR_EXTENSION_NOT_PRESENT。我们可以利用这个错误代码在扩展不能满足时直接结束我们的程序，这对于像窗口系统这种必要的扩展来说非常适合。但有时，我们请求的扩展可能是非必须的，有了很好，没有的话，程序仍然可以运行，这时，我们该怎么做呢？

实际上 Vulkan 提供了一个叫做 vkEnumerateInstanceExtensionProperties 可以在 Vulkan 实例创建之前返回支持的扩展列表。通过它，我们可以获取扩展的个数，以及扩展的详细信息，此外，它还允许我们指定校验层来对扩展进行过滤，但在这里，我们不使用它，将其设置为 nullptr。

我们首先需要知道扩展的数量，以便分配合适的数组大小来存储信息。可以通过下面的代码来获取扩展的数量：

```
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
```

知道了扩展的数量后，我们就可以分配数组来存储扩展信息：

```
std::vector<VkExtensionProperties> extensions(extensionCount);
```

我们使用下面的代码获取所有扩展信息：

```
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());
```

每个 VkExtensionProperties 结构体包含了扩展的名字和版本信息。我们可以使用下面的代码将这些信息打印在控制台窗口中 (代码中的
t 表示制表符)：

```
std::cout << "available extensions:" << std::endl;

for (const auto& extension : extensions) {
```

```
    std::cout << "\t" << extension.extensionName << std::endl;
}
```

读者可以将上面的代码加入 createInstance 函数，获取扩展支持信息。此外，我们可以编写一个函数来检测调用 glfwGetRequiredInstanceExtensions 函数返回的扩展是否全部包含在了扩展支持列表中。

清理

VkInstance 应该在应用程序结束前进行清除操作。我们可以在 cleanup 中调用 vkDestroyInstance 函数完成清除工作：

```
void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

vkDestroyInstance 函数的参数非常直白。之前提到，Vulkan 对象的分配和清除函数都有一个可选的分配器回调参数，在本教程，我们没有自定义的分配器，所以，将其设置为 nullptr。除了 Vulkan 实例，其余我们使用 Vulkan API 创建的对象也需要被清除，且应该在 Vulkan 实例清除之前被清除。

创建 Vulkan 实例后，在进行更复杂的操作之前，我们先熟悉一下校验层来帮助我们进行应用程序的调试。

本章节代码：

C++：

https://vulkan-tutorial.com/code/01_instance_creation.cpp

校验层

校验层是什么？

Vulkan API 的设计是紧紧围绕最小化驱动程序开销进行的，所以，默认情况下，Vulkan API 提供的错误检查功能非常有限。很多很基本的错误都没有被 Vulkan 显式地处理，遇到错误程序会直接崩溃或者发生未被明确定义的行为。Vukan 需要我们显式地定义每一个操作，所以就很容易在使用过程中产生一些小错误，比如使用了一个新的 GPU 特性，却忘记在逻辑设备创建时请求这一特性。

然而，这并不意味着我们不能将错误检查加入 API 调用。Vulkan 引入了校验层来优雅地解决这个问题。校验层是一个可选的可以用来在 Vulkan API 函数调用上进行附加操作的组件。校验层常被用来做下面的工作：

- 检测参数值是否合法

- 追踪对象的创建和清除操作，发现资源泄漏问题
- 追踪调用来自的线程，检测是否线程安全。
- 将 API 调用和调用的参数写入日志
- 追踪 API 调用进行分析和回放

下面的代码演示了 Vulkan 的校验层是如何工作的：

```
VkResult vkCreateInstance(
const VkInstanceCreateInfo* pCreateInfo,
const VkAllocationCallbacks* pAllocator,
VkInstance* instance) {

    if (pCreateInfo == nullptr || instance == nullptr) {
        log("Null pointer passed to required parameter!");
        return VK_ERROR_INITIALIZATION_FAILED;
    }

    return real_vkCreateInstance(pCreateInfo, pAllocator, instance);
}
```

校验层可以被自由堆叠包含任何读者感兴趣的调试功能。我们可以在开发时使用校验层，然后在发布应用程序时，禁用校验层来提高程序的运行表现。

Vulkan 库本身并没有提供任何内建的校验层，但 LunarG 的 Vulkan SDK 提供了一个非常不错的校验层实现。读者可以使用这个校验层实现来保证自己的应用程序在不同的驱动程序下能够尽可能得表现一致，而不是依赖于某个驱动程序的未定义行为。

校验层只能用于安装了它们的系统，比如，LunarG 的校验层只可以在安装了 Vulkan SDK 的 PC 上使用。

Vulkan 可以使用两种不同类型的校验层：实例校验层和设备校验层。实例校验层只检查和全局 Vulkan 对象相关的调用，比如 Vulkan 实例。设备校验层只检查和特定 GPU 相关的调用。设备校验层现在已经不推荐使用，也就是说，应该使用实例校验层来检测所有的 Vulkan 调用。Vulkan 规范文档为了兼容性仍推荐启用设备校验层。在本教程，为了简便，我们为实例和设备指定相同的校验层。

使用校验层

在本章节，我们将使用 LunarG 的 Vulkan SDK 提供的校验层。和使用扩展一样，使用校验层需要指定校验层的名称。LunarG 的 Vulkan SDK 允许我们通过 VK_LAYER_KHRONOS_validation 来隐式地开启所有可用的校验层。

首先，让我们添加两个变量到程序中来控制是否启用指定的校验层。这里，我们通过条件编译来设定是否启用校验层。代码中的 NDEBUG 宏是 C++ 标准的一部分，表示是否处于非调试模式下：

```
const int WIDTH = 800;
const int HEIGHT = 600;

const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif
```

接着，我们添加了一个叫做 checkValidationLayerSupport 的函数来请求所有可用的校验层。首先，我们调用 vkEnumerateInstanceLayerProperties 函数获取了所有可用的校验层列表。这一函数的用法和前面我们在创建 Vulkan 实例章节中使用的 vkEnumerateInstanceExtensionProperties 函数相同。

```
bool checkValidationLayerSupport() {
    uint32_t layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

    std::vector<VkLayerProperties> availableLayers(layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

    return false;
}
```

接着，检查是否所有 validationLayers 列表中的校验层都可以在 availableLayers 列表中找到：

```
for (const char* layerName : validationLayers) {
    bool layerFound = false;

    for (const auto& layerProperties : availableLayers) {
        if (strcmp(layerName, layerProperties.layerName) == 0) {
            layerFound = true;
            break;
        }
    }

    if (!layerFound) {
        return false;
    }
}
return true;
```

现在，我们在 createInstance 函数中调用它：

```
void createInstance() {
    if (enableValidationLayers && !checkValidationLayerSupport()) {
        throw std::runtime_error("validation layers requested, but not available!");
    }

        ...
}
```

现在，在调试模式下编译运行程序，确保没有错误出现。如果程序运行时出现错误，请确保正确安装了 Vulkan SDK。如果程序报告缺少可用的校验层，可以查阅 LunarG 的 Vulkan SDK 的官方文档寻找解决方法。

最后，修改我们之前的填写的 VkInstanceCreateInfo 结构体信息，在校验层启用时使用校验层：

```
if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
```

如果校验层检查成功,vkCreateInstance 函数调用就不会返回 VK_ERROR_LAYER_NOT_PRESENT 这一错误代码，但为了保险起见，读者应该运行程序来确保没有问题出现。

消息回调

仅仅启用校验层并没有任何用处，我们不能得到任何有用的调试信息。为了获得调试信息，我们需要使用 VK_EXT_debug_utils 扩展，设置回调函数来接受调试信息。

我们添加了一个叫做 getRequiredExtensions 的函数，这一函数根据是否启用校验层，返回所需的扩展列表：

```
std::vector<const char*> getRequiredExtensions() {
    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    std::vector<const char*> extensions(glfwExtensions,
    glfwExtensions + glfwExtensionCount);

    if (enableValidationLayers) {
        extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }

    return extensions;
}
```

GLFW 指定的扩展是必需的，调试报告相关的扩展根据校验层是否启用添加。代码

中我们使用了一个 VK_EXT_DEBUG_UTILS_EXTENSION_NAME，它等价于 VK_EXT_debug_utils，使用它是为了避免打字时的手误。

现在，我们在 createInstance 函数中调用这一函数：

```
auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
createInfo.ppEnabledExtensionNames = extensions.data();
```

接着,编译运行程序,确保没有出现 VK_ERROR_EXTENSION_NOT_PRESENT 错误。校验层的可用已经隐含说明对应的扩展存在，所以我们不需要额外去做扩展是否存在的检查。

现在，让我们来看接受调试信息的回调函数。我们在程序中以 vkDebugUtilsMessengerCallbackEXT 为原型添加一个叫做 debugCallback 的静态函数。这一函数使用 VKAPI_ATTR 和 VKAPI_CALL 定义，确保它可以被 Vulkan 库调用。

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback( VkDebugUtilsMessageSeverityFlagBitsEXT

    std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;

    return VK_FALSE;
}
```

函数的第一个参数指定了消息的级别，它可以是下面这些值：

- VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT：诊断信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT：资源创建之类的信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT：警告信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT：不合法和可能造成崩溃的操作信息

这些值经过一定设计，可以使用比较运算符来过滤处理一定级别以上的调试信息：

```
if (messageSeverity >=VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Message is important enough to show
}
```

messageType 参数可以是下面这些值：

- VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT：发生了一些与规范和性能无关的事件
- VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT：出现了违反规范的情况或发生了一个可能的错误
- VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT：进行了可能影响 Vulkan 性能的行为

pCallbackData 参数是一个指向 VkDebugUtilsMessengerCallbackDataEXT 结构体的指针，这一结构体包含了下面这些非常重要的成员：

- pMessage：一个以 null 结尾的包含调试信息的字符串
- pObjects：存储有和消息相关的 Vulkan 对象句柄的数组
- objectCount：数组中的对象个数

最后一个参数 pUserData 是一个指向了我们设置回调函数时，传递的数据的指针。

回调函数返回了一个布尔值，用来表示引发校验层处理的 Vulkan API 调用是否被中断。如果返回值为 true，对应 Vulkan API 调用就会返回 VK_ERROR_VALIDATION_FAILED_EXT 错误代码。通常，只在测试校验层本身时会返回 true，其余情况下，回调函数应该返回 VK_FALSE。

定义完回调函数，接下来要做的就是设置 Vulkan 使用这一回调函数。我们需要一个 VkDebugUtilsMessengerEXT 对象来存储回调函数信息，然后将它提交给 Vulkan 完成回调函数的设置：

```
VkDebugUtilsMessengerEXT callback;
```

现在，我们在 initVulkan 函数中，在 createInstance 函数调用之后添加一个 setupDebugCallback 函数调用：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
}

void setupDebugCallback() {
    if (!enableValidationLayers) return;

}
```

接着，我们需要填写 VkDebugUtilsMessengerCreateInfoEXT 结构体所需的信息：

```
VkDebugUtilsMessengerCreateInfoEXT createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
createInfo.pfnUserCallback = debugCallback;
createInfo.pUserData = nullptr; // Optional
```

messageSeverity 域可以用来指定回调函数处理的消息级别。在这里，我们设置回调函数处理除了 VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT 外的所有级别的消息，这使得我们的回调函数可以接收到可能的问题信息，同时忽略掉冗长的一般调试信息。

messageType 域用来指定回调函数处理的消息类型。在这里，我们设置处理所有类型的消息。读者可以根据自己的需要开启和禁用处理的消息类型。

pfnUserCallback 域是一个指向回调函数的指针。pUserData 是一个指向用户自定义数据的指针，它是可选的，这个指针所指的地址会被作为回调函数的参数，用来向回调函数传递用户数据。

有许多方式配置校验层消息和回调，更多信息可以参考扩展的规范文档。

填写完结构体信息后，我们将它作为一个参数调用 vkCreateDebugUtilsMessengerEXT 函数来创建 VkDebugUtilsMessengerEXT 对象。由于 vkCreateDebugUtilsMessengerEXT 函数是一个扩展函数，不会被 Vulkan 库自动加载，所以需要我们自己使用 vkGetInstanceProcAddr 函数来加载它。在这里，我们创建了一个代理函数，来载入 vkCreateDebugUtilsMessengerEXT 函数：

```
VkResult CreateDebugUtilsMessengerEXT(VkInstance instance, const VkDebugUtilsMessengerCreate
    auto func = (PFN_vkCreateDebugUtilsMessengerEXT)
    vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pCallback);
    } else {
        return VK_ERROR_EXTENSION_NOT_PRESENT;
    }
}
```

vkGetInstanceProcAddr 函数如果不能被加载，那么我们的代理函数就会发挥 nullptr。现在我们可以使用这个代理函数来创建扩展对象：

```
if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr,
&callback) != VK_SUCCESS) {
    throw std::runtime_error("failed to set up debug callback!");
}
```

函数的第二个参数是可选的分配器回调函数，我们没有自定义的分配器，所以将其设置为 nullptr。由于我们的调试回调是针对特定 Vulkan 实例和它的校验层，所以需要在第一个参数指定调试回调作用的 Vulkan 实例。现在，让我们编译运行程序，如果一切顺利，读者可以看到一个空白窗口，关闭空白窗口后，可以在控制台窗口看到下面的信息：

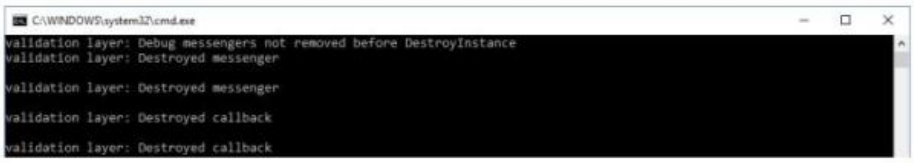

Figure 29: image

这说明，我们的程序还存在问题！VkDebugUtilsMessengerEXT 对象在程序结束前通过调用 vkDestroyDebugUtilsMessengerEXT 函数来清除掉。和 vkCreateDe-

bugUtilsMessengerEXT 函数相同，Vulkan 库没有自动加载这个函数，需要我们自己加载它。控制台窗口出现多次相同的错误信息是正常的，这是因为有多个校验层检查发现了这个问题。

现在，让我们创建 CreateDebugUtilsMessengerEXT 函数的代理函数：

```
void DestroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT callback, c
    auto func = (PFN_vkDestroyDebugUtilsMessengerEXT) vkGetInstanceProcAddr(instance, "vkDes
    if (func != nullptr) {
        func(instance, callback, pAllocator);
    }
}
```

这个代理函数需要被定义为类的静态成员函数或者被定义在类之外。我们在 cleanup 函数中调用这个函数：

```
void cleanup() {
    if (enableValidationLayers) {
        DestroyDebugUtilsMessengerEXT(instance, callback, nullptr);
    }

    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

现在，再次编译运行程序，如果一切顺利，错误信息这次就不会出现。如果读者想要了解到底是哪个函数调用引发了错误消息，可以在处理消息的回调函数设置断点，然后运行程序，观察程序在断点位置时的调用栈，就可以确定引发错误消息的函数调用。

配置

校验层除了 VkDebugUtilsMessengerCreateInfoEXT 结构体指定的标志外，还有大量可以决定校验层行为的设置。读者可以浏览 Vulkan SDK 的 Config 目录，里面有一个 vk_layer_settings.txt 解释了如何配置校验层。

读者可以将 vk_layer_settings.txt 复制到自己的项目的 Debug 和 Release 目录来使用它，并按照说明根据需要修改设置。在本教程，我们只使用 vk_layer_settings.txt 的默认设置。

在之后的章节，我们会故意造成一些错误，来演示如何使用校验层来发现这些错误，帮助读者理解校验层的重要性。现在，是时候来看一看系统中的 Vulkan 设备了。

本章节代码：

C++：

https://vulkan-tutorial.com/code/02_validation_layers.cpp

物理设备和队列族

选择一个物理设备

创建 VkInstance 后，我们需要查询系统中的显卡设备，选择一个支持我们需要的特性的设备使用。Vulkan 允许我们选择任意数量的显卡设备，并能够同时使用它们，但在这里，我们只使用第一个满足我们需求的显卡设备。

我们首先添加一个叫做 pickPhysicalDevice 的函数，然后在 initVulkan 函数中调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    pickPhysicalDevice();
}

void pickPhysicalDevice() {

}
```

我们使用 VkPhysicalDevice 对象来存储我们选择使用的显卡信息。这一对象可以在 VkInstance 进行清除操作时，自动清除自己，所以我们不需要再 cleanup 函数中对它进行清除。

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

请求显卡列表和请求扩展列表的操作类似，首先需要请求显卡的数量。

```
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
```

如果可用的显卡设备数量为 0，显然应用程序无法继续运行。

```
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
```

获取完设备数量后，我们就可以分配数组来存储 VkPhysicalDevice 对象。

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```

现在，让我们检查获取的设备能否满足我们的需求：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

我们检查设备，并选择使用第一个满足需求的设备：

```
for (const auto& device : devices) {
    if (isDeviceSuitable(device)) {
```

```
            physicalDevice = device;
            break;
        }
    }

    if (physicalDevice == VK_NULL_HANDLE) {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
```

下一节，我们开始具体说明 isDeviceSuitable 函数所进行的检查，随着我们使用的特性增多，这一函数所包含的检查也越来越多。

设备需求检测

为了选择合适的设备，我们需要获取更加详细的设备信息。对于基础的设备属性，比如名称，类型和支持的 Vulkan 版本的查询可以通过 vkGetPhysicalDeviceProperties 函数进行。

```
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(device, &deviceProperties);
```

纹理压缩，64 位浮点和多视口渲染 (常用于 VR) 等特性的支持可以通过 vkGetPhysicalDeviceFeatures 函数查询：

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
```

有关设备内存和队列族信息的获取，我们会在下一节说明。

现在，假设我们的应用程序只有在显卡支持几何着色器的情况下才可以运行，那么我们的 isDeviceSuitable 函数看起来会像这样：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    VkPhysicalDeviceProperties deviceProperties;
    VkPhysicalDeviceFeatures deviceFeatures;
    vkGetPhysicalDeviceProperties(device, &deviceProperties);
    vkGetPhysicalDeviceFeatures(device, &deviceFeatures);

    return deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU && deviceFeat
}
```

除了直接选择第一个满足需求的设备这种方法，一个更好的方法是给每一个满足需求的设备，按照特性加权打分，选择分数最高的设备使用。具体可以这样做：

```
#include <map>

        ...

void pickPhysicalDevice() {
        ...
```

```
    // Use an ordered map to automatically sort candidates by increasing score
    std::multimap<int, VkPhysicalDevice> candidates;

    for (const auto& device : devices) {
        int score = rateDeviceSuitability(device);
        candidates.insert(std::make_pair(score, device));
    }

    // Check if the best candidate is suitable at all
    if (candidates.rbegin()->first > 0) {
        physicalDevice = candidates.rbegin()->second;
    } else {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
}

int rateDeviceSuitability(VkPhysicalDevice device) {
        ...

    int score = 0;

    // Discrete GPUs have a significant performance advantage
    if (deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
        score += 1000;
    }

    // Maximum possible size of textures affects graphics quality
    score += deviceProperties.limits.maxImageDimension2D;

    // Application can't function without geometry shaders
    if (!deviceFeatures.geometryShader) {
        return 0;
    }

    return score;
}
```

此外，也可以显示满足需求的设备列表，让用户自己选择使用的设备。

由于我们的教程才刚刚开始，我们现在的唯一需求就是显卡设备需要支持 Vulkan，显然它对于我们使用 Vulkan API 获取的设备列表中的所有设备都永远满足：

队列族

之前提到，Vulkan 的几乎所有操作，从绘制到加载纹理都需要将操作指令提交给一个队列，然后才能执行。Vulkan 有多种不同类型的队列，它们属于不同的队列族，每

个队列族的队列只允许执行特定的一部分指令。比如，可能存在只允许执行计算相关指令的队列族和只允许执行内存传输相关指令的队列族。

我们需要检测设备支持的队列族，以及它们中哪些支持我们需要使用的指令。为了完成这一目的，我们添加了一个叫做 findQueueFamilies 的函数，这一函数会查找出满足我们需求的队列族。目前而言，我们需要的队列族只需要支持图形指令即可，但在之后的章节，我们可能会有更多的需求。

这一函数会返回满足需求得队列族的索引。这里，我们使用了一个结构体来作为函数返回结果的类型，索引-1 表示没有找到满足需求的队列族：

```
struct QueueFamilyIndices {
    int graphicsFamily = -1;

    bool isComplete() {
        return graphicsFamily >= 0;
    }
};
```

接下来，我们实现 findQueueFamilies 函数：

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

        ...

    return indices;
}
```

我们首先获取设备的队列族个数，然后分配数组存储队列族的 VkQueueFamilyProperties 对象：

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());
```

VkQueueFamilyProperties 结构体包含了队列族的很多信息，比如支持的操作类型，该队列族可以创建的队列个数。在这里，我们需要找到一个支持 VK_QUEUE_GRAPHICS_BIT 的队列族。

```
int i = 0;
for (const auto& queueFamily : queueFamilies) {
    if (queueFamily.queueCount > 0 && queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        indices.graphicsFamily = i;
    }

    if (indices.isComplete()) {
        break;
```

```
        }

        i++;
    }
```

现在，我们可以在 isDeviceSuitable 函数中调用它来确保我们选择的设备可以执行我们需要的指令：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    return indices.isComplete();
}
```

太棒了！我们已经完成了查找我们需要的物理设备这一工作！接下来，让我们创建逻辑设备来使用它！

本章节代码：

C++：

https://vulkan-tutorial.com/code/03_physical_device_selection.cpp

逻辑设备和队列

介绍

选择物理设备后，我们还需要一个逻辑设备来作为和物理设备交互的接口。逻辑设备的创建过程类似于我们之前描述的 Vulkan 实例的创建过程。我们还需要指定使用的队列所属的队列族。对于同一个物理设备，我们可以根据需求的不同，创建多个逻辑设备。

首先，我们添加一个逻辑设备对象作为类成员：

```
VkDevice device;
```

接着，添加一个叫做 createLogicalDevice 的函数，在 initVulkan 函数中调用它。

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createLogicalDevice() {

}
```

指定要创建的队列

逻辑设备创建需要填写 VkDeviceQueueCreateInfo 结构体。这一结构体描述了针对一个队列族我们所需的队列数量。目前而言，我们只使用了带有图形能力的队列族。

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily;
queueCreateInfo.queueCount = 1;
```

目前而言，对于每个队列族，驱动程序只允许创建很少数量的队列，但实际上，对于每一个队列族，我们很少需要一个以上的队列。我们可以在多个线程创建指令缓冲，然后在主线程一次将它们全部提交，降低调用开销。

Vulkan 需要我们赋予队列一个 0.0 到 1.0 之间的浮点数作为优先级来控制指令缓冲的执行顺序。即使只有一个队列，我们也要显式地赋予队列优先级：

```
float queuePriority = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriority;
```

指定使用的设备特性

接下来，我们要指定应用程序使用的设备特性。我们暂时先简单地定义它，之后再回来填写：

```
VkPhysicalDeviceFeatures deviceFeatures = {};
```

创建逻辑设备

填写好前面两个结构体后，我们可以开始填写 VkDeviceCreateInfo 结构体。

```
VkDeviceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

将 VkDeviceCreateInfo 结构体的 pQueueCreateInfos 指针指向 queueCreateInfo 的地址，pEnabledFeatures 指针指向 deviceFeatures 的地址：

```
createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.queueCreateInfoCount = 1;

createInfo.pEnabledFeatures = &deviceFeatures;
```

结构体的其余成员和 VkInstanceCreateInfo 类似，不同的是这次的设置是针对设备的。

VK_KHR_swapchain 就是一个设备特定扩展的例子，这一扩展使得我们可以将渲染的图像在窗口上显示出来。看起来似乎应该所有支持 Vulkan 的设备都应该支持这一扩展，然而，实际上有的 Vulkan 设备只支持计算指令，不支持这一图形相关扩展。在之后的章节，我们会对交换链进行更加深入地说明。

之前提到，我们可以对设备和 Vulkan 实例使用相同地校验层，不需要额外的扩展支持：

```
createInfo.enabledExtensionCount = 0;

if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
```

现在，我们可以调用 vkCreateDevice 函数创建逻辑设备了。

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical device!");
}
```

vkCreateDevice 函数的参数包括我们创建的逻辑设备进行交互的物理设备对象，我们刚刚在结构体中指定的需要使用的队列信息，可选的分配器回调，以及用来存储返回的逻辑设备对象的内存地址。和 Vulkan 实例对象的创建函数类似，这一函数调用在请求的设备需求不被满足时返回错误代码。

逻辑设备对象创建后，应用程序结束前，需要我们自己在 cleanup 函数中调用 vkDestroyDevice 函数来清除它：

```
void cleanup() {
    vkDestroyDevice(device, nullptr);
        ...
}
```

逻辑设备并不直接与 Vulkan 实例交互，所以创建逻辑设备时不需要使用 Vulkan 实例作为参数。

获取队列句柄

创建逻辑设备时指定的队列会随着逻辑设备一同被创建，为了方便，我们添加了一个 VkQueue 成员变量来直接存储逻辑设备的队列句柄：

```
VkQueue graphicsQueue;
```

逻辑设备的队列会在逻辑设备清除时，自动被清除，所以不需要我们在 cleanup 函数中进行队列的清除操作。

vkGetDeviceQueue 函数可以获取指定队列族的队列句柄。它的参数依次是逻辑设备对象，队列族索引，队列索引，用来存储返回的队列句柄的内存地址。因为，我们只创建了一个队列，所以，可以直接使用索引 0 调用函数：

```
vkGetDeviceQueue(device, indices.graphicsFamily, 0, &graphicsQueue);
```

创建完逻辑设备，我们就可以真正开始使用显卡来完成一些操作。在接下来的章节，我们将开始配置资源，进行一些绘制操作，将渲染结果显示在窗口上。

本章节代码：

C++：

https://vulkan-tutorial.com/code/04_logical_device.cpp

窗口表面

Vulkan 是一个平台无关的 API，它不能直接和窗口系统交互。为了将 Vulkan 渲染的图像显示在窗口上，我们需要使用 WSI(Window System Integration) 扩展。在本章节，我们首先介绍 VK_KHR_surface 扩展，它通过 VkSurfaceKHR 对象抽象出可供 Vulkan 渲染的表面。在本教程，我们使用 GLFW 来获取 VkSurfaceKHR 对象。

VK_KHR_surface 是一个实例级别的扩展，它已经被包含在使用 glfwGetRequiredInstanceExtensions 函数获取的扩展列表中，所以，我们不需要自己请求这一扩展。WSI 扩展同样也被包含在 glfwGetRequiredInstanceExtensions 函数获取的扩展列表中，也不需要我们自己请求。

由于窗口表面对物理设备的选择有一定影响，它的创建只能在 Vulkan 实例创建之后进行。

创建窗口表面

```
VkSurfaceKHR surface;
```

尽管 VkSurfaceKHR 对象是平台无关的，但它的创建依赖窗口系统。比如，在 Windows 系统上，它的创建需要 HWND 和 HMODULE。存在一个叫做 VK_KHR_win32_surface 的 Windows 平台特有扩展，用于处理与 Windows 系统窗口交互有关的问题，这一扩展也被包含在了 glfwGetRequiredInstanceExtensions 函数获取的扩展列表中。

接下来，我们将会演示如何使用这一 Windows 系统特有扩展来创建表面，但对于之后的章节，我们不会使用这一特定平台扩展，而是直接使用 GLFW 库来完成相关操作。我们可以使用 GLFW 库的 glfwCreateWindowSurface 函数来完成表面创建。这里演示如何使用这一平台特定扩展，是出于学习目的，让读者能明白我们使用的 GLFW 库在背后究竟做了什么。

我们需要填写 VkWin32SurfaceCreateInfoKHR 结构体来完成 VkSurfaceKHR 对象的创建。这一结构体包含了两个非常重要的成员：hwnd 和 hinstance。它们分别对应 Windows 系统的窗口句柄和进程实例句柄：

```
VkWin32SurfaceCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
createInfo.hwnd = glfwGetWin32Window(window);
createInfo.hinstance = GetModuleHandle(nullptr);
```

glfwGetWin32Window 函数可以获取 GLFW 窗口对象的 Windows 平台窗口句柄。GetModuleHandle 函数可以获取当前进程的实例句柄。

vkCreateWin32SurfaceKHR 函数需要我们自己加载。加载后使用 Vulkan 实例，要创建的表面信息，自定义内存分配器和要存储表面对象的内存地址为参数调用：

```
auto CreateWin32SurfaceKHR = (PFN_vkCreateWin32SurfaceKHR) vkGetInstanceProcAddr(instance, "

if (!CreateWin32SurfaceKHR || CreateWin32SurfaceKHR(instance, &createInfo, nullptr, &surface
    throw std::runtime_error("failed to create window surface!");
}
```

其它平台的处理方式与之类似，比如 Linux 平台，可以通过 vkCreateXcbSurfaceKHR 函数完成表面创建的工作。

GLFW 库的 glfwCreateWindowSurface 函数在不同平台的实现是不同的，可以跨平台使用。现在，我们将它集成到我们的应用程序中。添加一个叫做 createSurface 的函数，然后在 initVulkan 函数中在 Vulkan 实例创建和 setupDebugCallback 调用之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createSurface() {

}
```

glfwCreateWindowSurface 函数的参数非常直白：

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
        throw std::runtime_error("failed to create window surface!");
    }
}
```

它的参数依次是 VkInstance 对象，GLFW 窗口指针，自定义内存分配器，存储返回的 VkSurfaceKHR 对象的内存地址。调用后，它会返回 VkResult 来指示创建是否成功。表面在应用程序退出需要被清理，GLFW 并没有提供清除表面的函数，我们可以自己调用 vkDestroySurfaceKHR 函数完成这一工作：

```
void cleanup() {
        ...
    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestroyInstance(instance, nullptr);
        ...
}
```

需要注意，表面对象的清除需要在 Vulkan 实例被清除之前完成。

查询呈现支持

尽管，具体的 Vulkan 实现可能对窗口系统进行了支持，但这并不意味着所有平台的 Vulkan 实现都支持同样的特性。所以，我们需要扩展 isDeviceSuitable 函数来确保设备可以在我们创建的表面上显示图像。

实际上，支持绘制指令的队列族和支持表现的队列族并不一定重叠。所以，我们需要修改 QueueFamilyIndices 结构体，添加成员变量存储表现队列族的索引：

```
struct QueueFamilyIndices {
    int graphicsFamily = -1;
    int presentFamily = -1;

    bool isComplete() {
        return graphicsFamily >= 0 && presentFamily >= 0;
    }
};
```

接着，我们还需要修改 findQueueFamilies 函数，查找带有呈现图像到窗口表面能力的队列族。我们可以在检查队列族是否具有 VK_QUEUE_GRAPHICS_BIT 的同级循环调用 vkGetPhysicalDeviceSurfaceSupportKHR 函数来检查物理设备是否具有呈现能力：

```
VkBool32 presentSupport = false;
vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);
```

然后，根据队列族中的队列数量和是否支持表现确定使用的表现队列族的索引：

```
if (queueFamily.queueCount > 0 && presentSupport) {
    indices.presentFamily = i;
}
```

读者可能已经注意到，按照上面的方法最后选择使用的绘制指令队列族和呈现队列族很有可能是同一个队列族。但为了统一操作，即使两者是同一个队列族，我们也按照它们是不同的队列族来对待。实际上，读者可以显式地指定绘制和呈现队列族是同一个的物理设备来提高性能表现。

创建呈现队列

现在，我们可以修改逻辑设备的创建过程，创建呈现队列，并将队列句柄保存在成员变量中：

```
VkQueue presentQueue;
```

我们需要多个 VkDeviceQueueCreateInfo 结构体来创建所有使用的队列族。一个优雅的处理方式是使用 STL 的集合创建每一个不同的队列族：

```
#include <set>

        ...
```

```
    QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

    std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
    std::set<int> uniqueQueueFamilies = {indices.graphicsFamily, indices.presentFamily};

    float queuePriority = 1.0f;
    for (int queueFamily : uniqueQueueFamilies) {
        VkDeviceQueueCreateInfo queueCreateInfo = {};
        queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
        queueCreateInfo.queueFamilyIndex = queueFamily;
        queueCreateInfo.queueCount = 1;
        queueCreateInfo.pQueuePriorities = &queuePriority;
        queueCreateInfos.push_back(queueCreateInfo);
    }
```

修改 VkDeviceCreateInfo 结构体的 pQueueCreateInfos:

```
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
```

对于同一个队列族，我们只需要传递它的索引一次。最后，调用 vkGetDeviceQueue 函数获取队列句柄:

```
vkGetDeviceQueue(device, indices.presentFamily, 0, &presentQueue);
```

对于队列族相同的情况，我们获取的队列句柄也极有可能相同。在下一章节，我们将介绍交换链，以及如何使用它将图像显示到窗口表面上。

本章节代码:

C++:

https://vulkan-tutorial.com/code/05_window_surface.cpp

交换链

Vulkan 没有默认帧缓冲的概念，它需要一个能够缓冲渲染操作的组件。在 Vulkan 中，这一组件就是交换链。Vulkan 的交换链必须显式地创建，不存在默认的交换链。交换链本质上一个包含了若干等待呈现的图像的队列。我们的应用程序从交换链获取一张图像，然后在图像上进行渲染操作，完成后，将图像返回到交换链的队列中。交换链的队列的工作方式和它呈现图像到表面的条件依赖于交换链的设置。但通常来说，交换链被用来同步图像呈现和屏幕刷新。

检测交换链支持

并不是所有的显卡设备都具有可以直接将图像呈现到屏幕的能力。比如，被设计用于服务器的显卡是没有任何显示输出设备的。此外，由于图像呈现非常依赖窗口系统，以及和窗口系统有关的窗口表面，这些并非 Vulkan 核心的一部分。使用交换链，我们必须保证 VK_KHR_swapchain 设备扩展被启用。

为了确保 VK_KHR_swapchain 设备扩展被设备支持，我们需要扩展 VK_KHR_swapchain 函数检测该扩展是否被支持。之前，我们已经介绍了列出 VkPhysicalDevice 对象支持的扩展列表的方法，现在只需要在这个列表中检测是否存在 VK_KHR_swapchain 扩展即可。Vulkan 的头文件提供了一个叫做 VK_KHR_SWAPCHAIN_EXTENSION_NAME 的宏，它等价于 VK_KHR_swapchain。我们使用这个宏来做检测，而不直接使用 VK_KHR_swapchain，可以保证代码具有更好的兼容性。

首先，我们定义所需的的设备扩展列表，这类似于我们之前定义的要启用的校验层列表。

```
const std::vector<const char*> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};
```

接着，添加一个叫做 checkDeviceExtensionSupport 的函数，然后在 isDeviceSuitable 函数中调用它：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    bool extensionsSupported = checkDeviceExtensionSupport(device);

    return indices.isComplete() && extensionsSupported;
}

bool checkDeviceExtensionSupport(VkPhysicalDevice device) {
    return true;
}
```

修改 checkDeviceExtensionSupport 函数的函数体枚举设备扩展列表，检测所需的扩展是否存在：

```
bool checkDeviceExtensionSupport(VkPhysicalDevice device) {
    uint32_t extensionCount;
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);

    std::vector<VkExtensionProperties> availableExtensions(extensionCount);
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, availableExtensio

    std::set<std::string> requiredExtensions(deviceExtensions.begin(), deviceExtensions.end(

    for (const auto& extension : availableExtensions) {
        requiredExtensions.erase(extension.extensionName);
    }

    return requiredExtensions.empty();
}
```

在这里，我们将所需的扩展保存在一个集合中，然后枚举所有可用的扩展，将集合中的扩展剔除，最后，如果这个集合中的元素为 0，说明我们所需的扩展全部都被满足。实际上，如果设备支持呈现队列，那么它就一定支持交换链。但我们最好还是显式地进行交换链扩展的检测，然后显式地启用交换链扩展。

启用交换链扩展，只需要对逻辑设备的创建过程做很小地修改：

```
createInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();
```

查询交换链支持细节

只检查交换链是否可用还不够，交换链可能与我们的窗口表面不兼容。创建交换链所要进行的设置要比 Vulkan 实例和设备创建多得多，在进行交换链创建之前需要我们查询更多的信息。

有三种最基本的属性，需要我们检查：

- 基础表面特性 (交换链的最小/最大图像数量，最小/最大图像宽度、高度)
- 表面格式 (像素格式，颜色空间)
- 可用的呈现模式

和 findQueueFamilies 函数类似，我们使用结构体来存储我们查询得到的交换链细节信息：

```
struct SwapChainSupportDetails {
    VkSurfaceCapabilitiesKHR capabilities;
    std::vector<VkSurfaceFormatKHR> formats;
    std::vector<VkPresentModeKHR> presentModes;
};
```

现在，我们添加一个叫做 querySwapChainSupport 的函数用于填写上面的结构体：

```
SwapChainSupportDetails querySwapChainSupport(VkPhysicalDevice device) {
    SwapChainSupportDetails details;

    return details;
}
```

在本节，我们先介绍如何查询上面的结构体所包含的信息，在下一节再对它们的具体意义进行说明。

我们先查询基础表面特性。这一属性的查询非常简单，调用下面的函数即可：

```
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface, &details.capabilities);
```

这一函数以 VkPhysicalDevice 对象和 VkSurfaceKHR 作为参数来查询表面特性。与交换链信息查询有关的函数都需要这两个参数，它们是交换链的核心组件。

下一步，我们查询表面支持的格式。这一查询结果是一个结构体列表，所以它的查询与之前设备特性查询类似，首先查询格式数量，然后分配数组空间查询具体信息：

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);

if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data
}
```

确保向量的空间足以容纳所有格式结构体。最后，使用与调用 vkGetPhysicalDeviceSurfacePresentModesKHR 函数同样的方式查询支持的呈现模式：

```
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, nullptr);

if (presentModeCount != 0) {
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, details.pr
}
```

现在所有查询得到的信息已经存储在了结构体中，我可以再次扩展 isDeviceSuitable 函数检测交换链的能力是否满足需求。对于我们的教程而言，我们只需要交换链至少支持一种图像格式和一种支持我们的窗口表面的呈现模式即可：

```
bool swapChainAdequate = false;
if (extensionsSupported) {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(device);
    swapChainAdequate = !swapChainSupport.formats.empty() && !swapChainSupport.presentModes
}
```

我们只能在验证交换链扩展可用后查询交换链的细节信息。isDeviceSuitable 函数的最后一行需要修改为：

```
return indices.isComplete() && extensionsSupported && swapChainAdequate;
```

为交换链选择合适的设置

swapChainAdequate 为真，说明交换链的能力满足我们的需要，但仍有许多不同的优化模式需要设置。接下来，我们会编写一组函数来查找合适的设置。设置的内容如下：

- 表面格式 (颜色，深度)
- 呈现模式 (显示图像到屏幕的条件)
- 交换范围 (交换链中的图像的分辨率)

对于上面的设置，每一个我们都有一个理想的值，如果这个理想的值不能满足，我们会使用编写的逻辑查找一个尽可能好的替代值。

表面格式

我们添加了一个叫做 chooseSwapSurfaceFormat 的函数来选择合适的表面格式：

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR> &availableF

}
```

每一个 VkSurfaceFormatKHR 条目包含了一个 format 和 colorSpace 成员变量。format 成员变量用于指定颜色通道和存储类型。比如，如果 fomat 成员变量的值为 VK_FORMAT_B8G8R8A8_UNORM 表示我们以 B，G，R 和 A 的顺序，每个颜色通道用 8 位无符号整型数表示，总共每像素使用 32 位表示。colorSpace 成员变量用来表示 SRGB 颜色空间是否被支持，是否使用 VK_COLOR_SPACE_SRGB_NONLINEAR_KHR 标志。需要注意 VK_COLOR_SPACE_SRGB_NONLINEAR_KHR 在之前的 Vulkan 规范中叫做 VK_COLORSPACE_SRGB_NONLINEAR_KHR。

对于颜色空间，如果 SRGB 被支持，我们就使用 SRGB，使用它可以得到更加准确的颜色表示。直接使用 SRGB 颜色有很大挑战，所以我们使用 RGB 作为颜色格式，这一格式可以通过 VK_FORMAT_B8G8R8A8_UNORM 宏指定。

Vulkan 通过返回一个 format 成员变量值为 VK_FORMAT_UNDEFINED 的 VkSurfaceFormatKHR 表明表面没有自己的首选格式，这时，我们直接返回我们设定的格式：

```
if (availableFormats.size() == 1 && availableFormats[0].format == VK_FORMAT_UNDEFINED) {
    return {VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
}
```

如果 Vulkan 返回了一个格式列表，那么我们检查这个列表，看下我们想要设定的格式是否存在于这个列表中：

```
for (const auto& availableFormat : availableFormats) {
    if (availableFormat.format == VK_FORMAT_B8G8R8A8_UNORM && availableFormat.colorSpace ==
        return availableFormat;
    }
}
```

如果不能在列表中找到我们想要的格式，我们可以对列表中存在的格式进行打分，选择分数最高的那个作为我们使用的格式，当然，大多数情况下，直接使用列表中的第一个格式也是非常不错的选择：

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR> &availableF
    if (availableFormats.size() == 1 && availableFormats[0].format == VK_FORMAT_UNDEFINED) {
        return {VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
    }

    for (const auto& availableFormat : availableFormats) {
        if (availableFormat.format == VK_FORMAT_B8G8R8A8_UNORM && availableFormat.colorSpace
            return availableFormat;
```

```
        }
    }

    return availableFormats[0];
}
```

呈现模式

呈现模式可以说是交换链中最重要的设置。它决定了什么条件下图像才会显示到屏幕。Vulkan 提供了四种可用的呈现模式：

- VK_PRESENT_MODE_IMMEDIATE_KHR：应用程序提交的图像会被立即传输到屏幕上，可能会导致撕裂现象。
- VK_PRESENT_MODE_FIFO_KHR：交换链变成一个先进先出的队列，每次从队列头部取出一张图像进行显示，应用程序渲染的图像提交给交换链后，会被放在队列尾部。当队列为满时，应用程序需要进行等待。这一模式非常类似现在常用的垂直同步。刷新显示的时刻也被叫做垂直回扫。
- VK_PRESENT_MODE_FIFO_RELAXED_KHR：这一模式和上一模式的唯一区别是，如果应用程序延迟，导致交换链的队列在上一次垂直回扫时为空，那么，如果应用程序在下一次垂直回扫前提交图像，图像会立即被显示。这一模式可能会导致撕裂现象。
- VK_PRESENT_MODE_MAILBOX_KHR：这一模式是第二种模式的另一个变种。它不会在交换链的队列满时阻塞应用程序，队列中的图像会被直接替换为应用程序新提交的图像。这一模式可以用来实现三倍缓冲，避免撕裂现象的同时减小了延迟问题。

上面四种呈现模式，只有 VK_PRESENT_MODE_FIFO_KHR 模式保证一定可用，所以我们还需要编写一个函数来查找最佳的可用呈现模式：

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR> availablePresentM
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

作者个人认为三倍缓冲综合来说表现最佳。三倍缓冲避免了撕裂现象,同时具有较低的延迟。我们检查用于实现三倍缓冲的 VK_PRESENT_MODE_MAILBOX_KHR 模式是否可用，可用的话，就使用它：

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR> availablePresentM
    for (const auto& availablePresentMode : availablePresentModes) {
        if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR)
        {
            return availablePresentMode;
        }
    }
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

不幸的是，目前而言，还有许多驱动程序对 VK_PRESENT_MODE_FIFO_KHR 呈现模式的支持不够好，所以，如果 VK_PRESENT_MODE_MAILBOX_KHR 呈现模式不可用，我们应该使用 VK_PRESENT_MODE_IMMEDIATE_KHR 模式：

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR> availablePresentM
    VkPresentModeKHR bestMode = VK_PRESENT_MODE_FIFO_KHR;

    for (const auto& availablePresentMode : availablePresentModes) {
        if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR)
        {
            return availablePresentMode;
        } else if (availablePresentMode == VK_PRESENT_MODE_IMMEDIATE_KHR) {
            bestMode = availablePresentMode;
        }
    }

    return bestMode;
}
```

交换范围

现在只剩下一个属性需要设置了，我们添加一个叫做 chooseSwapExtent 的函数来设置它：

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR &capabilities) {

}
```

交换范围是交换链中图像的分辨率，它几乎总是和我们要显示图像的窗口的分辨率相同。VkSurfaceCapabilitiesKHR 结构体定义了可用的分辨率范围。Vulkan 通过 currentExtent 成员变量来告知适合我们窗口的交换范围。一些窗口系统会使用一个特殊值，uint32_t 变量类型的最大值，表示允许我们自己选择对于窗口最合适的交换范围，但我们选择的交换范围需要在 minImageExtent 与 maxImageExtent 的范围内。

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR &capabilities) {
    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    } else {
        VkExtent2D actualExtent = {WIDTH, HEIGHT};

        actualExtent.width = std::max(capabilities.minImageExtent.width, std::min(capabiliti
        actualExtent.height = std::max(capabilities.minImageExtent.height, std::min(capabili

        return actualExtent;
    }
}
```

代码中 max 和 min 函数用于在允许的范围内选择交换范围的高度值和宽度值，需要在源文件中包含 algorithm 头文件才能够使用它们。

创建交换链

现在，我们已经编写了大量辅助函数帮助我们在应用程序运行时选择最合适的设置，可以开始进行交换链的创建了。

我们添加一个叫做 createSwapChain 的函数，它会选择合适的交换链设置，然后，我们在 initVulkan 函数中在逻辑设备创建之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
}

void createSwapChain() {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(physicalDevice);

    VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
    VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes);
    VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);
}
```

除了上面这些，还有一些地方需要我们进行设置，但这些设置都很简单，没有必要为它们编写独立的设置函数。这些设置包括交换链中的图像个数，也就是交换链的队列可以容纳的图像个数。我们使用交换链支持的最小图像个数 +1 数量的图像来实现三倍缓冲：

```
uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount > swapChainSupport.capabil
    imageCount = swapChainSupport.capabilities.maxImageCount;
}
```

maxImageCount 的值为 0 表明，只要内存可以满足，我们可以使用任意数量的图像。

和其它 Vulkan 对象相同，创建交换链对象需要填写一个包含大量信息的结构体。这一结构体的一些成员我们已经非常熟悉：

```
VkSwapchainCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = surface;
```

指定交换链绑定的表面后，我们还需要指定有关交换链图像的信息：

```
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1;
createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

imageArrayLayers 成员变量用于指定每个图像所包含的层次。通常，来说它的值为 1。但对于 VR 相关的应用程序来说，会使用更多的层次。imageUsage 成员变量用于指定我们将在图像上进行怎样的操作。在本教程，我们在图像上进行绘制操作，也就是将图像作为一个颜色附着来使用。如果读者需要对图像进行后期处理之类的操作，可以使用 VK_IMAGE_USAGE_TRANSFER_DST_BIT 作为 imageUsage 成员变量的值，让交换链图像可以作为传输的目的图像。

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
uint32_t queueFamilyIndices[] = {(uint32_t) indices.graphicsFamily, (uint32_t) indices.prese

if (indices.graphicsFamily != indices.presentFamily) {
    createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
    createInfo.queueFamilyIndexCount = 2;
    createInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
    createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    createInfo.queueFamilyIndexCount = 0; // Optional
    createInfo.pQueueFamilyIndices = nullptr; // Optional
}
```

接着，我们需要指定在多个队列族使用交换链图像的方式。这一设置对于图形队列和呈现队列不是同一个队列的情况有着很大影响。我们通过图形队列在交换链图像上进行绘制操作，然后将图像提交给呈现队列来显示。有两种控制在多个队列访问图像的方式：

- VK_SHARING_MODE_EXCLUSIVE：一张图像同一时间只能被一个队列族所拥有，在另一队列族使用它之前，必须显式地改变图像所有权。这一模式下性能表现最佳。
- VK_SHARING_MODE_CONCURRENT：图像可以在多个队列族间使用，不需要显式地改变图像所有权。

如果图形和呈现不是同一个队列族，我们使用协同模式来避免处理图像所有权问题。协同模式需要我们使用 queueFamilyIndexCount 和 pQueueFamilyIndices 来指定共享所有权的队列族。如果图形队列族和呈现队列族是同一个队列族（大部分情况下都是这样），我们就不能使用协同模式，协同模式需要我们指定至少两个不同的队列族。

```
createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
```

我们可以为交换链中的图像指定一个固定的变换操作（需要交换链具有 supportedTransforms 特性），比如顺时针旋转 90 度或是水平翻转。如果读者不需要进行任何变换操作，指定使用 currentTransform 变换即可。

```
createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

compositeAlpha 成员变量用于指定 alpha 通道是否被用来和窗口系统中的其它窗口进行混合操作。通常,我们将其设置为 VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR 来忽略掉 alpha 通道。

```
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;
```

presentMode 成员变量用于设置呈现模式。clipped 成员变量被设置为 VK_TRUE 表示我们不关心被窗口系统中的其它窗口遮挡的像素的颜色，这允许 Vulkan 采取一定的优化措施，但如果我们回读窗口的像素值就可能出现问题。

```
createInfo.oldSwapchain = VK_NULL_HANDLE;
```

最后是 oldSwapchain 成员变量，需要指定它，是因为应用程序在运行过程中交换链可能会失效。比如，改变窗口大小后，交换链需要重建，重建时需要之前的交换链，具体细节，我们会在之后的章节详细介绍。现在，我们还没有创建任何一个交换链，将它设置为 VK_NULL_HANDLE 即可。添加一个 VkSwapchainKHR 成员变量来存储交换链：

```
VkSwapchainKHR swapChain;
```

调用 vkCreateSwapchainKHR 函数创建交换链：

```
if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
    throw std::runtime_error("failed to create swap chain!");
}
```

vkCreateSwapchainKHR 函数的参数依次是逻辑设备对象，交换链创建信息，可选的自定义内存分配器和用于存储返回的交换链对象的内存地址。接着，我们需要在 cleanup 函数中在逻辑设备被清除前调用 vkDestroySwapchainKHR 函数来清除交换链对象：

```
void cleanup() {
    vkDestroySwapchainKHR(device, swapChain, nullptr);
        ...
}
```

现在可以编译运行程序，确保我们成功地创建了交换链。如果 vkCreateSwapchainKHR 函数调用出现错误，那么，就移除 createInfo.imageExtent

extent; 这行代码，然后，启用校验层，编译运行程序，就可以捕获错误，得到一些有用地信息：

Figure 30: image

获取交换链图像

我们已经创建了交换链，接下来需要做地就是获取交换链图像的图像句柄。我们会在之后使用这些图像句柄进行渲染操作。现在，添加成员变量用于存储这些图像句柄：

```
std::vector<VkImage> swapChainImages
```

交换链图像由交换链自己负责创建，并在交换链清除时自动被清除，不需要我们自己进行创建和清除操作。

我们在 createSwapChain 函数的尾部，vkCreateSwapchainKHR 函数调用之后，添加代码来获取交换链图像句柄。获取它们的方法和获取其它 Vulkan 对象的方法类似，首先获取交换链图像的数量，然后分配数组空间，获取交换链图像句柄。

```
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, swapChainImages.data());
```

我们在创建交换链时指定了一个 minImageCount 成员变量来请求最小需要的交换链图像数量。Vulkan 的具体实现可能会创建比这个最小交换链图像数量更多的交换链图像，我们在这里，我们仍然需要显式地查询交换链图像数量，确保不会出错。

最后，在成员变量中存储我们设置的交换链图像格式和范围，我们会在之后的章节使用它们。

```
VkSwapchainKHR swapChain;
std::vector<VkImage> swapChainImages;
VkFormat swapChainImageFormat;
VkExtent2D swapChainExtent;

        ...

swapChainImageFormat = surfaceFormat.format;
swapChainExtent = extent;
```

现在，我们已经拥有了可以进行绘制操作的交换链图像，以及可以呈现图像的窗口表面。从下一章节开始，我们开始真正的图形管线部分。

本章节代码：

C++：

https://vulkan-tutorial.com/code/06_swap_chain_creation.cpp

图像视图

使用任何 VkImage 对象，包括处于交换链中的，处于渲染管线中的，都需要我们创建一个 VkImageView 对象来绑定访问它。图像视图描述了访问图像的方式，以及图像的哪一部分可以被访问。比如，图像可以被图像视图描述为一个没有细化级别的二维深度纹理，进而可以在其上进行与二维深度纹理相关的操作。

在本章节，我们编写了一个叫做 createImageViews 的函数来为交换链中的每一个图像建立图像视图。

首先，添加用于存储图像视图的成员变量：

```
std::vector<VkImageView> swapChainImageViews;
```

然后，添加 createImageViews 函数，并在 initVulkan 函数调用 createSwapChain 函数创建交换链之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
}

void createImageViews() {

}
```

接着，我们分配足够的数组空间来存储图像视图：

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

}
```

遍历所有交换链图像，创建图像视图：

```
for (size_t i = 0; i < swapChainImages.size(); i++) {

}
```

图像视图的创建需要我们填写 VkImageViewCreateInfo 结构体：

```
VkImageViewCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
createInfo.image = swapChainImages[i];
```

viewType 和 format 成员变量用于指定图像数据的解释方式。viewType 成员变量用于指定图像被看作是一维纹理、二维纹理、三维纹理还是立方体贴图。

```
createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
createInfo.format = swapChainImageFormat;
```

components 成员变量用于进行图像颜色通道的映射。比如，对于单色纹理，我们可以将所有颜色通道映射到红色通道。我们也可以直接将颜色通道的值映射为常数 0 或 1。在这里，我们只使用默认的映射：

```
createInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
```

subresourceRange 成员变量用于指定图像的用途和图像的哪一部分可以被访问。在这里，我们的图像被用作渲染目标，并且没有细分级别，只存在一个图层：

```
createInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
createInfo.subresourceRange.baseMipLevel = 0;
createInfo.subresourceRange.levelCount = 1;
createInfo.subresourceRange.baseArrayLayer = 0;
createInfo.subresourceRange.layerCount = 1;
```

如果读者在编写 VR 一类的应用程序，可能会使用支持多个层次的交换链。这时，读者应该为每个图像创建多个图像视图，分别用来访问左眼和右眼两个不同的图层。

调用 vkCreateImageView 函数创建图像视图：

```
if (vkCreateImageView(device, &createInfo, nullptr, &swapChainImageViews[i]) != VK_SUCCESS) {
    throw std::runtime_error("failed to create image views!");
}
```

和交换链图像不同，图像视图是由我们自己显式创建的，需要我们自己在 cleanup 函数中清除它们：

```
void cleanup() {
    for (auto imageView : swapChainImageViews) {
        vkDestroyImageView(device, imageView, nullptr);
    }

    ...
}
```

有了图像视图，就可以将图像作为纹理使用，但作为渲染目标，还需要帧缓冲对象。接下来的章节，我们会简要介绍渲染管线，然后介绍帧缓冲对象。

本章节代码：

C++：

https://vulkan-tutorial.com/code/07_image_views.cpp

图形管线概述

在接下来的章节，我们将开始配置图形管线来渲染我们的三角形。图形管线是一系列将我们提交的顶点和纹理转换为渲染目标上的像素的操作。它的简化过程如下：

input assembler 获取顶点数据，顶点数据的来源可以是应用程序提交的原始顶点数据，或是根据索引缓冲提取的顶点数据。

vertex shader 对每个顶点进行模型空间到屏幕空间的变换，然后将顶点数据传递给图形管线的下一阶段。

tessellation shaders 根据一定的规则对几何图形进行细分，从而提高网格质量。通常被用来使类似墙面这类不光滑表面看起来更自然。

geometry shader 可以以图元 (三角形，线段，点) 为单位处理几何图形，它可以剔除图元，输出图元。有点类似于 tessellation shader，但更灵活。但目前已经不推荐应用程序使用它，geometry shader 的性能在除了 Intel 集成显卡外的大多数显卡上表现不佳。

rasterization 阶段将图元离散为片段。片段被用来在帧缓冲上填充像素。位于屏幕外的片段会被丢弃，顶点着色器输出的顶点属性会在片段之间进行插值，开启深度测试后，位于其它片段之后的片段也会被丢弃。

fragment shader 对每一个未被丢弃的片段进行处理，确定片段要写入的帧缓冲，它可以使用来自 vertex shader 的插值数据，比如纹理坐标和顶点法线。

color blending 阶段对写入帧缓冲同一像素位置的不同片段进行混合操作。片段可以直接覆盖之前写入的片段，也可以基于之前片段写入的信息进行混合操作。

使用绿色标识的阶段也被叫做固定功能阶段。固定功能阶段允许通过参数对处理过程进行一定程度的配置。

使用橙色标识的阶段是可编程阶段，允许我们将自己的代码加载到显卡，进行我们想要的操作。这使得我们可以实现许多有趣的效果。我们加载到显卡的代码会被 GPU 并行处理。

如果读者使用了一些旧的图形 API，可能会对 glBlendFunc 和 OMSetBlendState 之类的函数比较熟悉，这些函数可以对图形管线进行一定的设置。而在 Vulkan 中，图形管线几乎完全不允许进行动态设置，如果我们想使用其它着色器，绑定其它帧缓冲，以及改变混合函数，都需要重新创建管线。这就迫使我们必须提前创建所有我们需要使用的图形管线，虽然这样看起来不太方便，但这给驱动程序带来了很大的优化空间。

图形管线的部分可编程阶段不是必需的。比如对于 tessellation 和 geometry shader 阶段，如果我们只是画一个简单的三角形，完全没有必要使用它们。如果我们只是需要生成阴影贴图的深度值，我们也可以不使用 fragment shader。

在下一章节，我们首先创建两个对于在屏幕上绘制三角形必需的可编程阶段：vertex shader 和 fragment shader。对于固定功能的设置，比如混合模式，视口，光栅化会在下一章节之后进行。最后，我们开始图形管线中帧缓冲的配置工作。

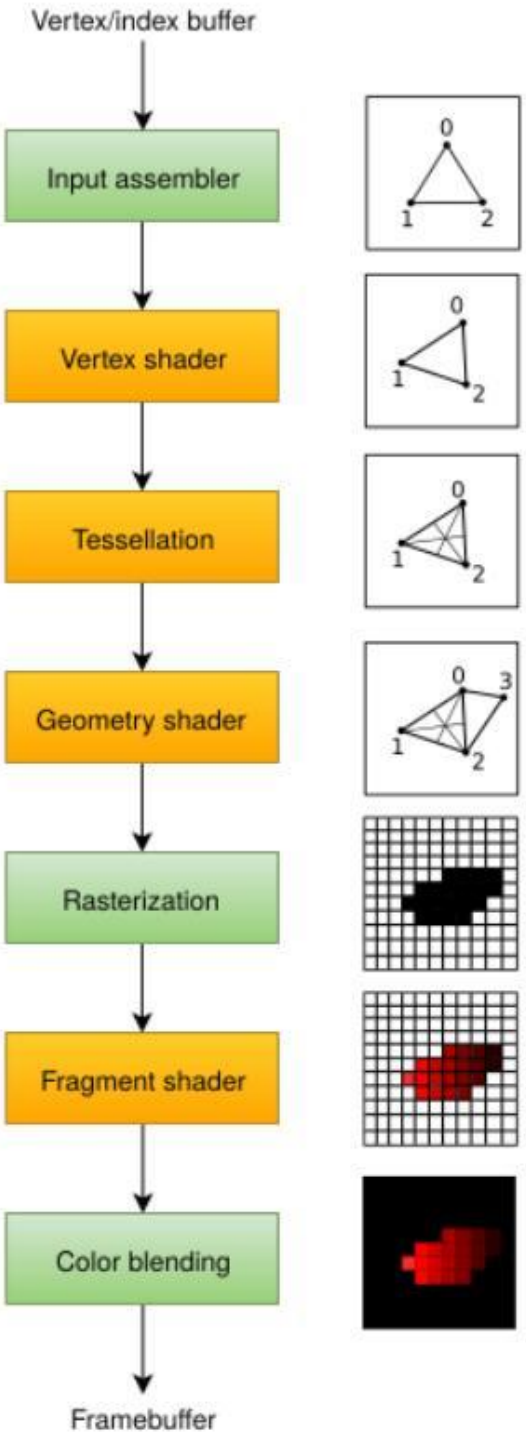

Figure 31: image

我们添加一个叫做 createGraphicsPipeline 的函数，并在 initVulkan 函数中的 createImageViews 函数调用之后调用它。接下来的章节我们主要在 createGraphicsPipeline 这个函数中进行代码编写。

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createGraphicsPipeline();
}

        ...

void createGraphicsPipeline() {

}
```

本章节代码：

C++：

https://vulkan-tutorial.com/code/08_graphics_pipeline.cpp

着色器模块

和之前的一些图形 API 不同，Vulkan 使用的着色器代码格式是一种叫做 SPIR-V 的字节码，这一字节码格式可以在 Vulkan 和 OpenCL 上使用。可以用它来编写图形和计算着色器，在本教程，我们将它用于编写图形管线的着色器。

GPU 厂商的编译器将字节码转换为原生代码的工作复杂度远远低于直接编译较高级的类 C 代码。过去的经验告诉我们使用类 C 代码，比如 GLSL 作为着色器代码，会因为不同 GPU 厂商对代码的不同解释而造成大量问题，并且类 C 代码的编译器实现要比字节码编译器复杂的多，GPU 厂商实现的编译器也极有可能存在错误，不同 GPU 厂商的实现也差异巨大。而使用字节码格式，上述的这些问题可以在极大程度上减少。

虽然，Vulkan 使用字节码格式作为着色器代码，但这并不意味着我们要直接书写字节码来编写着色器。Khronos 发布了一个独立于厂商的可以将 GLSL 代码转换为 SPIR-V 字节码的编译器。这个编译器可以验证我们的着色器代码是否完全符合标准，将 GLSL 代码转换为 SPIR-V 字节码。我们可以在应用程序运行时调用这个编译器，动态生成 SPIR-V 字节码，但在本教程，我们没有这样做。这一编译器已经被包含在了 LunarG 的 Vulkan SDK 中，编译器可执行文件名称为 glslangValidator.exe，不需要读者另外下载。

GLSL 是一个类 C 的着色器语言。使用 GLSL 编写的程序包含了一个 main 函数，这一函数完成具体的运算操作。GLSL 使用全局变量进行输入输出，它包含了许多用于图形编程的特性，比如向量和矩阵支持，用于计算叉积的函数，用于矩阵与向量相乘的函数，用于计算反射向量的函数等等。GLSL 中的向量类型叫做 vec，后跟一个表示向量元素数的数字。比如，用于表示一个三维空间位置的向量的类型为 vec3。GLSL 允许我们访问向量的分量比如.x，也允许我们使用表达式来创建新的向量值，比如 vec3(1.0, 2.0, 3.0).xy 会返回一个 vec2 类型的值。向量构造器也可以被组合使用，比如可以使用 vec3(vec2(1.0, 2.0), 3.0) 生成一个 vec3 类型的值。

之前的章节提到，我们需要编写顶点着色器和片段着色器才能完成在屏幕上绘制三角形的工作。接下来的两节的内容就是使用 GLSL 编写顶点着色器和片段着色器代码，然后使用编译器将它们转换为 SPIR-V 字节码。

顶点着色器

顶点着色器对输入的每个顶点进行处理。它可以接收顶点属性作为输入，比如世界坐标，颜色，法线和纹理坐标。它的输出包括顶点最终的裁剪坐标和需要传递给片段着色器的顶点属性，比如颜色和纹理坐标。这些值会被插值处理后传给顶点着色器。

裁剪坐标是一个来自顶点着色器的四维向量，它的四个成分会被除以第四个成分来完成规范化。规范化后的坐标被映射到帧缓冲的坐标空间，如下图所示：

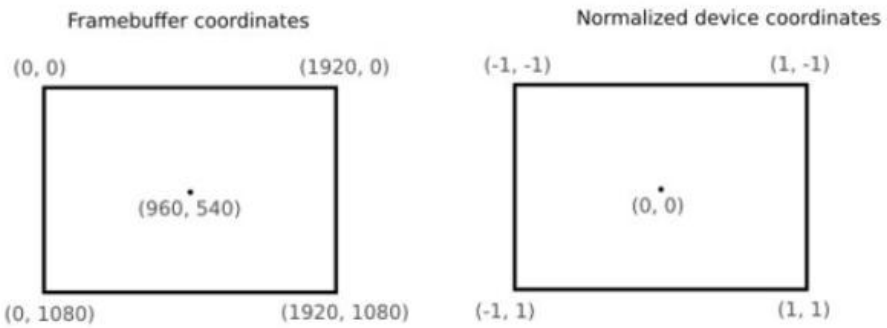

Figure 32: image

如果读者对计算机图形学有所了解，应该对此比较熟悉。如果读者之前使用过 OpenGL，可能会注意到这里的 Y 坐标和 OpenGL 的 Y 坐标是相反方向的，Z 坐标现在的范围和 Direct3D 相同，为 0 到 1。

对于我们的要绘制的三角形，不需要使用任何变换操作，我们直接将三角形的三个顶点的坐标作为规范化设备坐标来生成下图的三角形：

我们可以直接将顶点着色器输出的裁剪坐标的第四个成分设置为 1，然后作为规范设备坐标。这样裁剪坐标到规范设备坐标就不会对坐标进行任何变换。

通常，顶点坐标被存储在一个顶点缓冲中，但对于 Vulkan 来说，创建顶点缓冲，然后填入数据要进行很多操作。为了尽快让我们的三角形显示在屏幕上，我们暂时先直接将顶点坐标写入顶点着色器，就像这样：

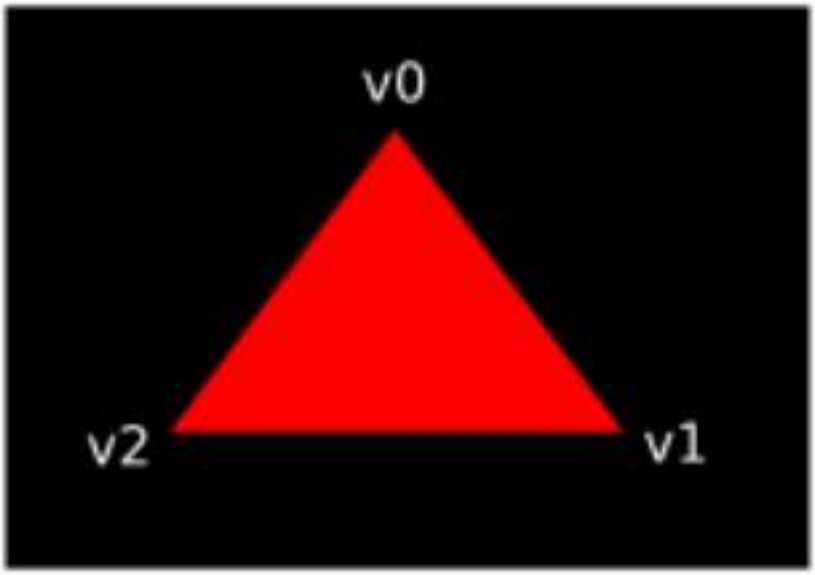

Figure 33: image

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4 gl_Position;
};

vec2 positions[3] = vec2[](
                vec2(0.0, -0.5),
                vec2(0.5, 0.5),
                vec2(-0.5, 0.5)
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
}
```

着色器的 main 函数对于每个顶点执行一次。GLSL 内建的 gl_VertexIndex 变量包含了当前顶点的索引。这一索引通常来说是用来引用顶点缓冲中的顶点数据，但在这里，我们用它来引用我们在着色器中硬编码的顶点数据。我们输出的裁剪坐标由代码中的 positions 数组给出了前两个成分，剩余两个成分被我们设置为了 0.0 和 1.0。为了让着色器代码可以在 Vulkan 下工作，我们需要使用 GL_ARB_separate_shader_objects 扩展。

片段着色器

我们的三角形由来自顶点着色器的三个顶点作为三角形的顶点构成，这一三角形范围内的屏幕像素会被使用片段着色器处理后的片段进行填充。一个非常简单的直接将片段颜色设置为红色的片段着色器代码如下所示：

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

对于每个片段着色器的 main 函数执行一次。GLSL 下的颜色是一个具有四个分量的向量，分别对应 R，G，B 和 Alpha 通道，分量的取值范围为 [0,1]。和顶点着色器不同，片段着色器没有类似 gl_Position 这样的内建变量可以用于输出当前处理的片段的颜色。我们必须自己为每个使用的帧缓冲指定对应的输出变量。上面代码中的 layout(location

0) 用于指定与颜色变量相关联的帧缓冲，颜色变量的颜色数据会被写入与它相关联的帧缓冲中。上面的代码，我们将红色写入和索引为 0 的帧缓冲相关联的颜色变量 outColor。

逐顶点着色

整个三角形都是红色，看上去一点都不好玩，接下来，让我们尝试把三角形变成下面这个样子：

我们需要对之前编写的顶点着色器和片段着色器进行修改才能得到上图的效果。首先，我们需要为三角形的每个顶点指定不同的颜色，我们在顶点着色器中添加一个颜色数组：

```
vec3 colors[3] = vec3[](
                 vec3(1.0, 0.0, 0.0),
                 vec3(0.0, 1.0, 0.0),
                 vec3(0.0, 0.0, 1.0)
);
```

现在，我们只需要将顶点的颜色传递给片段着色器，由片段着色器将颜色值输出到帧缓冲上。在顶点着色器中添加颜色输出变量，并在 main 函数中写入颜色值到颜色输出变量：

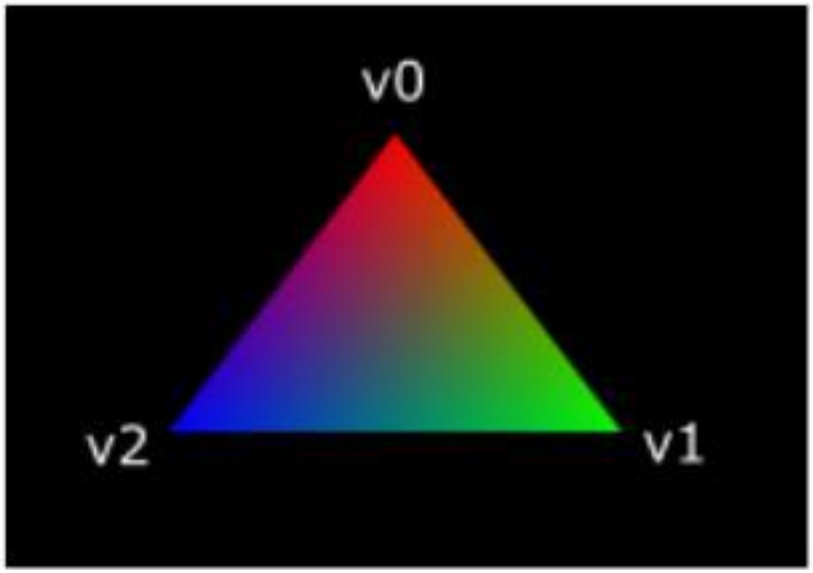

Figure 34: image

```
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

接着，我们在片段着色器中添加对应的输入变量：

```
layout(location = 0) in vec3 fragColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

一组对应的输入和输出变量可以使用不同的变量名称，编译器可以通过定义它们时使用的 location 将它们对应起来。片段着色器的 main 函数现在被我们修改为输出输入的颜色变量作为片段颜色。三角形除了顶点之外的片段颜色会被插值处理。

编译着色器

现在，让我们在项目目录下创建一个叫做 shaders 的文件夹，然后在其中添加一个叫做 shader.vert 的保存有顶点着色器代码的文件，以及一个叫做 shader.frag 的保存有片段着色器代码的文件。GLSL 并没有规定着色器文件应该使用的扩展名称，上面使用的扩展名名称来源于习惯。

shader.vert 文件的内容如下：

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4 gl_Position;
};

layout(location = 0) out vec3 fragColor;

vec2 positions[3] = vec2[](
                vec2(0.0, -0.5),
                vec2(0.5, 0.5),
                vec2(-0.5, 0.5)
);

vec3 colors[3] = vec3[](
                vec3(1.0, 0.0, 0.0),
                vec3(0.0, 1.0, 0.0),
                vec3(0.0, 0.0, 1.0)
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

shader.frag 文件的内容如下：

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

接下来，我们使用 glslangValidator 来将着色器代码编译为 SPIR-V 字节码格式。

Windows

创建一个 compile.bat 文件，它的内容如下：

```
C:/VulkanSDK/1.0.17.0/Bin32/glslangValidator.exe -V shader.vert
C:/VulkanSDK/1.0.17.0/Bin32/glslangValidator.exe -V shader.frag
pause
```

读者需要将上面代码中 glslangValidator.exe 的文件路径替换为自己的 glslangValidator.exe 所在的文件路径。双击运行这个文件，就可以完成着色器代码的编译。

Linux

创建一个 compile.sh 文件，它的内容如下：

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslangValidator -V shader.vert
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslangValidator -V shader.frag
```

读者需要将上面代码中 glslangValidator 的文件路径替换为自己的 glslangValidator 所在的文件路径。然后在终端中使用 chmod +x compile.sh 给予它可执行的权限，最后运行它。

以下部分与平台无关

上面我们使用两行代码使用-V 选项调用编译器，将 GLSL 着色器代码文件转换为 SPIR-V 字节码格式。运行脚本后，读者可以在当前文件夹下看到两个新的文件 vert.spv 和 frag.spv。这两个文件的文件名由编译器自动推导出来，读者可以使用自己喜欢的名称重命名这两个文件。编译脚本的执行过程中可能出现一些缺少特性的警告，读者可以放心地忽略掉这些警告。

如果着色器代码存在语法错误，编译器会报告语法错误所在的行，以及错误出现的原因。读者可以尝试去掉某行着色器代码的分号，让编译器检查这一语法错误，熟悉编译器的报错信息的。也可以尝试不使用任何选项调用编译器来查看编译器支持的选项种类。此外，编译器还支持将 SPIR-V 格式的字节码反向编译为便于人类阅读的代码格式。

载入着色器

我们已经得到了 SPIR-V 格式的着色器字节码文件，现在需要在应用程序中载入字节码文件。为了完成这项工作，我们首先编写一个用于载入二进制文件的辅助函数。

```
#include <fstream>

        ...

static std::vector<char> readFile(const std::string& filename) {
    std::ifstream file(filename, std::ios::ate | std::ios::binary);

    if (!file.is_open()) {
        throw std::runtime_error("failed to open file!");
    }
}
```

readFile 函数会读取指定文件的所有字节，然后将数据保存在 std::vector 数组返回给调用者。上面代码中我们使用了下面两个模式打开文件：

- ate：从文件尾部开始读取
- binary：以二进制的形式读取文件 (避免进行诸如行末格式是\n 还是\r\n 的转换)

使用 ate 模式，从文件尾部开始读取的原因是，我们可以根据读取位置确定文件的大小，然后分配足够的数组空间来容纳数据：

```
size_t fileSize = (size_t) file.tellg();
std::vector<char> buffer(fileSize);
```

分配好足够的数组空间后，我们可以跳到文件头部，读取整个文件：

```
file.seekg(0);
file.read(buffer.data(), fileSize);
```

最后，关闭文件，返回数据数组：

```
file.close();

return buffer;
```

现在，我们可以在 createGraphicsPipeline 函数中调用 readFile 函数来完成着色器字节码的读取：

```
void createGraphicsPipeline() {
    auto vertShaderCode = readFile("shaders/vert.spv");
    auto fragShaderCode = readFile("shaders/frag.spv");
}
```

我们可以通过比较打印出的分配的数组大小和实际的文件大小，确保着色器字节码被正确载入。

创建着色器模块

要将着色器字节码在管线上使用，还需要使用 VkShaderModule 对象。让我们添加一个叫做 createShaderModule 的函数来完成 VkShaderModule 对象的创建：

```
VkShaderModule createShaderModule(const std::vector<char>& code)
{

}
```

createShaderModule 函数使用我们读取的着色器字节码数组作为参数来创建 VkShaderModule 对象。

创建着色器模块非常简单，只需要通过 VkShaderModuleCreateInfo 结构体指定存储字节码的数组和数组长度即可。但需要注意一点，我们需要先将存储字节码的数组指针转换为 const uint32_t* 变量类型，来匹配结构体中的字节码指针的变量类型。这里，我们使用 C++ 的 reinterpret_cast 完成变量类型转换。此外，我们指

定的指针指向的地址应该符合 uint32_t 变量类型的内存对齐方式。我们这里使用的 std::vector，它的默认分配器分配的内存的地址符合这一要求。

```
VkShaderModuleCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
createInfo.codeSize = code.size();
createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
```

接着，我们调用 vkCreateShaderModule 函数创建 VkShaderModule 对象：

```
VkShaderModule shaderModule;
if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS) {
    throw std::runtime_error("failed to create shader module!");
}
```

和之前调用其它创建 Vulkan 对象的函数相同，调用 vkCreateShaderModule 函数需要逻辑设备对象，指向要创建对象信息的结构体，可选的自定义分配器以及用于存储返回的创建的对象句柄的内存地址。调用 vkCreateShaderModule 函数后，我们就可以立即释放掉存储着色器字节码的数组内存。最后，不要忘记返回创建的着色器模块对象：

```
return shaderModule;
```

着色器模块对象只在管线创建时需要，所以，不需要将它定义为一个成员变量，我们将它作为一个局部变量定义在 createGraphicsPipeline 函数中：

```
VkShaderModule vertShaderModule;
VkShaderModule fragShaderModule;
```

调用我们编写的辅助函数创建着色器模块对象：

```
vertShaderModule = createShaderModule(vertShaderCode);
fragShaderModule = createShaderModule(fragShaderCode);
```

我们需要在 createGraphicsPipeline 函数返回前，也就是 createGraphicsPipeline 函数的尾部，清除创建的着色器模块对象：

```
...
    vkDestroyShaderModule(device, fragShaderModule, nullptr);
    vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

创建着色器阶段

VkShaderModule 对象只是一个对着色器字节码的包装。我们还需要指定它们在管线处理哪一阶段被使用。指定着色器阶段需要使用 VkPipelineShaderStageCreateInfo 结构体。

我们首先在 createGraphicsPipeline 函数中为顶点着色器填写 VkPipelineShaderStageCreateInfo 结构体信息：

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo = {};
vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
```

上面代码，我们首先填写了 sType 成员变量，然后指定了着色器在管线的哪一阶段被使用。每个可编程阶段都有一个对应这一阶段的枚举值，我们使用这一枚举值指定着色器被使用的阶段。

```
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";
```

module 成员变量用于指定阶段使用的着色器模块对象，pName 成员变量用于指定阶段调用的着色器函数。我们可以通过使用不同 pName 在同一份着色器代码中实现所有需要的着色器，比如在同一份代码中实现多个片段着色器，然后通过不同的 pName 调用它们。但在本教程，我们使用 main 作为它的值。

VkPipelineShaderStageCreateInfo 还有一个可选的成员变量 pSpecializationInfo，在这里，我们没有使用它，但这一成员变量非常值得我们在这里对它进行说明，我们可以通过这一成员变量指定着色器用到的常量，我们可以对同一个着色器模块对象指定不同的着色器常量用于管线创建，这使得编译器可以根据指定的着色器常量来消除一些条件分支，这比在渲染时，使用变量配置着色器带来的效率要高得多。如果不使用着色器常量，可以将 pSpecializationInfo 成员变量设置为 nullptr。

接下来，填写用于片段着色器的 VkPipelineShaderStageCreateInfo 结构体：

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};
fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";
```

最后，我们定义包含上面定义的两个结构体的 VkPipelineShaderStageCreateInfo 数组，我们会在之后使用这一数组在管线创建时引用这两个结构体。

```
VkPipelineShaderStageCreateInfo shaderStages[] = {vertShaderStageInfo, fragShaderStageInfo};
```

我们已经完成了管线的可编程阶段的设置，接下来的章节，我们开始配置管线的固定功能阶段。

本章节代码：

C++：

https://vulkan-tutorial.com/code/09_shader_modules.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

固定功能

之前的许多图形 API 会为管线提供一些默认的状态。在 Vulkan 不存在默认状态，所有状态必须被显式地设置，无论是视口大小，还是使用的颜色混合函数都需要显式地指定。在本章节，我们开始对固定功能阶段进行配置。

顶点输入

我们可以使用 VkPipelineVertexInputStateCreateInfo 结构体来描述传递给顶点着色器的顶点数据格式。描述内容主要包括下面两个方面：

- 绑定：数据之间的间距和数据是按逐顶点的方式还是按逐实例的方式进行组织
- 属性描述：传递给顶点着色器的属性类型，用于将属性绑定到顶点着色器中的变量

由于我们直接在顶点着色器中硬编码顶点数据，所以我们填写结构体信息时指定不载入任何顶点数据。在后面的顶点缓冲章节，我们就会真正地载入顶点数据。

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vertexInputInfo.vertexBindingDescriptionCount = 0;
vertexInputInfo.pVertexBindingDescriptions = nullptr; // Optional
vertexInputInfo.vertexAttributeDescriptionCount = 0;
vertexInputInfo.pVertexAttributeDescriptions = nullptr; // Optional
```

pVertexBindingDescriptions 和 pVertexAttributeDescriptions 成员变量用于指向描述顶点数据组织信息地结构体数组。我们在 createGraphicsPipeline 函数中的 shaderStages 数组定义之后定义这一结构体。

输入装配

VkPipelineInputAssemblyStateCreateInfo 结构体用于描述两个信息：顶点数据定义了哪种类型的几何图元，以及是否启用几何图元重启。前一个信息通过 topology 成员变量指定，它的值可以是下面这些：

- VK_PRIMITIVE_TOPOLOGY_POINT_LIST：点图元
- VK_PRIMITIVE_TOPOLOGY_LINE_LIST：每两个顶点构成一个线段图元
- VK_PRIMITIVE_TOPOLOGY_LINE_STRIP：每两个顶点构成一个线段图元，除第一个线段图元外，每个线段图元使用上一个线段图元的一个顶点
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST：每三个顶点构成一个三角形图元
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP：每个三角形的第二个和第三个顶点被下一个三角形作为第一和第二个顶点使用

一般而言，我们会通过索引缓冲来更好地复用顶点缓冲中的顶点数据。如果将 primitiveRestartEnable 成员变量的值设置为 VK_TRUE，那么如果使用带有 _STRIP 结尾的图元类型，可以通过一个特殊索引值 0xFFFF 或 0xFFFFFFFF 达到重启图元的目的 (从特殊索引值之后的索引重置为图元的第一个顶点)。

我们的目的是绘制三角形，可以按照下面代码填写结构体：

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

视口和裁剪

视口用于描述被用来输出渲染结果的帧缓冲区域。一般而言，会将它设置为 (0，0) 到 (width，height)，在本教程，我们也采取这一设置：

```
VkViewport viewport = {};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float) swapChainExtent.width;
viewport.height = (float) swapChainExtent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
```

需要注意，交换链图像的大小可能与窗口大小不同。交换链图像在之后会被用作帧缓冲，所以这里我们设置视口大小为交换链图像的大小。

minDepth 和 maxDepth 成员变量用于指定帧缓冲使用的深度值的范围。它们的值必须在 [0.0f，1.0f] 之中，特别的，minDepth 的值可以大于 maxDepth 的值。如果读者没有特殊需要，一般将它们的值分别设置为 0.0f 和 1.0f。

视口定义了图像到帧缓冲的映射关系，裁剪矩形定义了哪一区域的像素实际被存储在帧缓存。任何位于裁剪矩形外的像素都会被光栅化程序丢弃。视口和裁剪的工作方式在下图中给出。

在本教程，我们在整个帧缓冲上进行绘制操作，所以将裁剪范围设置为和帧缓冲大小一样：

```
VkRect2D scissor = {};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
```

视口和裁剪矩形需要组合在一起，通过 VkPipelineViewportStateCreateInfo 结构体指定。许多显卡可以使用多个视口和裁剪矩形，所以指定视口和裁剪矩形的成员变量是一个指向视口和裁剪矩形的结构体数组指针。使用多个视口和裁剪矩形需要启用相应的特性支持 (参考逻辑设备创建章节)。

```
VkPipelineViewportStateCreateInfo viewportState = {};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportState.viewportCount = 1;
```

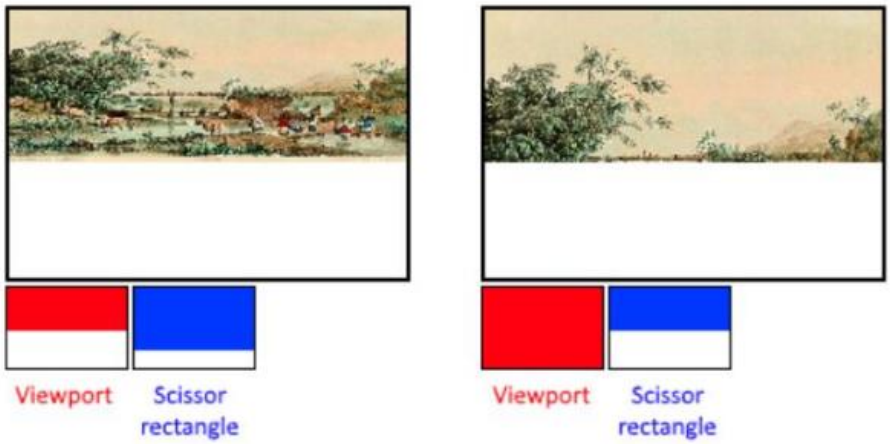

Figure 35: image

```
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

光栅化

光栅化程序将来自顶点着色器的顶点构成的几何图元转换为片段交由片段着色器着色。深度测试，背面剔除和裁剪测试如何开启了，也由光栅化程序执行。我们可以配置光栅化程序输出整个几何图元作为片段，还是只输出几何图元的边作为片段 (也就是线框模式)。光栅化程序的配置通过 VkPipelineRasterizationStateCreateInfo 结构体进行：

```
VkPipelineRasterizationStateCreateInfo rasterizer = {};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
```

depthClampEnable 成员变量设置为 VK_TRUE 表示在近平面和远平面外的片段会被截断为在近平面和远平面上，而不是直接丢弃这些片段。这对于阴影贴图的生成很有用。使用这一设置需要开启相应的 GPU 特性。

```
rasterizer.rasterizerDiscardEnable = VK_FALSE;
```

rasterizerDiscardEnable 成员变量设置为 VK_TRUE 表示所有几何图元都不能通过光栅化阶段。这一设置会禁止一切片段输出到帧缓冲。

polygonMode 成员变量用于指定几何图元生成片段的方式。它可以是下面这些值：

- VK_POLYGON_MODE_FILL：整个多边形，包括多边形内部都产生片段
- VK_POLYGON_MODE_LINE：只有多边形的边会产生片段

- VK_POLYGON_MODE_POINT：只有多边形的顶点会产生片段

使用除了 VK_POLYGON_MODE_FILL 外的模式，需要启用相应的 GPU 特性。

```
rasterizer.lineWidth = 1.0f;
```

lineWidth 成员变量用于指定光栅化后的线段宽度，它以线宽所占的片段数目为单位。线宽的最大值依赖于硬件，使用大于 1.0f 的线宽，需要启用相应的 GPU 特性。

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
```

cullMode 成员变量用于指定使用的表面剔除类型。我们可以通过它禁用表面剔除，剔除背面，剔除正面，以及剔除双面。frontFace 成员变量用于指定顺时针的顶点序是正面，还是逆时针的顶点序是正面。

```
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

光栅化程序可以添加一个常量值或是一个基于片段所处线段的斜率得到的变量值到深度值上。这对于阴影贴图会很有用，但在这里，我们不使用它，所以将 depthBiasEnable 成员变量设置为 VK_FALSE。

多重采样

我们使用 VkPipelineMultisampleStateCreateInfo 结构体来对多重采样进行配置。多重采样是一种组合多个不同多边形产生的片段的颜色来决定最终的像素颜色的技术，它可以一定程度上减少多边形边缘的走样现象。对于一个像素只被一个多边形产生的片段覆盖，只会对覆盖它的这个片段执行一次片段着色器，使用多重采样进行反走样的代价要比使用更高的分辨率渲染，然后缩小图像达到反走样的代价小得多。使用多重采样需要启用相应的 GPU 特性。

```
VkPipelineMultisampleStateCreateInfo multisampling = {};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampling.sampleShadingEnable = VK_FALSE;
multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optional
multisampling.pSampleMask = nullptr; // Optional
multisampling.alphaToCoverageEnable = VK_FALSE; // Optional
multisampling.alphaToOneEnable = VK_FALSE; // Optional
```

在之后的章节，我们会对多重采样进行更为详细地介绍，在这里，我们先禁用多重采样。

深度和模板测试

如果需要进行深度测试和模板测试，除了需要深度缓冲和模板缓冲外，还需要通过 VkPipelineDepthStencilStateCreateInfo 结构体来对深度测试和模板测试进行配置。在这里，暂时我们没有使用深度测试和模板测试，所以先将其设置为 nullptr，我们会在之后的章节再对它们进行更加详细地介绍。

颜色混合

片段着色器返回的片段颜色需要和原来帧缓冲中对应像素的颜色进行混合。混合的方式有下面两种：

- 混合旧值和新值产生最终的颜色
- 使用位运算组合旧值和新值

有两个用于配置颜色混合的结构体。第一个是 VkPipelineColorBlendAttachmentState 结构体，可以用它来对每个绑定的帧缓冲进行单独的颜色混合配置。第二个是 VkPipelineColorBlendStateCreateInfo 结构体，可以用它来进行全局的颜色混合配置。对于我们的教程，只使用了一个帧缓冲：

```
VkPipelineColorBlendAttachmentState colorBlendAttachment = {};
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
colorBlendAttachment.blendEnable = VK_FALSE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```

通过 VkPipelineColorBlendAttachmentState 结构体，我们可以对绑定的帧缓冲进行第一类混合方式的配置。第一类混合方式的运算过程类似下面的代码：

```
if (blendEnable) {
    finalColor.rgb = (srcColorBlendFactor * newColor.rgb) <colorBlendOp> (dstColorBlendFacto
    finalColor.a = (srcAlphaBlendFactor * newColor.a) <alphaBlendOp> (dstAlphaBlendFactor *
} else {
    finalColor = newColor;
}

finalColor = finalColor & colorWriteMask;
```

如果 blendEnable 成员变量被设置为 VK_FALSE,就不会进行混合操作。否则,就会执行指定的混合操作计算新的颜色值。计算出的新的颜色值会按照 colorWriteMask 的设置决定写入到帧缓冲的颜色通道。

通常，我们使用颜色混合是为了进行 alpha 混合来实现半透明效果。这时 finalColor 应该按照下面的方式计算：

```
finalColor.rgb = newAlpha * newColor + (1 - newAlpha) * oldColor;
finalColor.a = newAlpha.a;
```

上面的运算，可以通过下面的设置实现：

```
colorBlendAttachment.blendEnable = VK_TRUE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;
```

读者可以在 Vulkan 的规范文档中找到所有 VkBlendFactor 和 VkBlendOp 枚举可以进行的运算。

VkPipelineColorBlendStateCreateInfo 结构体使用了一个 VkPipelineColorBlendAttachmentState 结构体数组指针来指定每个帧缓冲的颜色混合设置，还提供了用于设置全局混合常量的成员变量。

```
VkPipelineColorBlendStateCreateInfo colorBlending = {};
colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlending.logicOpEnable = VK_FALSE;
colorBlending.logicOp = VK_LOGIC_OP_COPY; // Optional
colorBlending.attachmentCount = 1;
colorBlending.pAttachments = &colorBlendAttachment;
colorBlending.blendConstants[0] = 0.0f; // Optional
colorBlending.blendConstants[1] = 0.0f; // Optional
colorBlending.blendConstants[2] = 0.0f; // Optional
colorBlending.blendConstants[3] = 0.0f; // Optional
```

如果想要使用第二种混合方式 (位运算)，那么就需要将 logicOpEnable 成员变量设置为 VK_TRUE。然后使用 logicOp 成员变量指定要使用的位运算。需要注意，这样设置后会自动禁用第一种混合方式，就跟对每个绑定的帧缓冲设置 blendEnable 成员变量为 VK_FALSE 一样。colorWriteMask 成员变量的设置在第二种混合方式下仍然起作用，可以决定哪些颜色通道能够被写入帧缓冲。我们也可以禁用所有这两种混合模式，这种情况下，片段颜色会直接覆盖原来帧缓冲中存储的颜色值。

动态状态

只有非常有限的管线状态可以在不重建管线的情况下进行动态修改。这包括视口大小，线宽和混合常量。我们可以通过填写 VkPipelineDynamicStateCreateInfo 结构体指定需要动态修改的状态，比如像这样：

```
VkDynamicState dynamicStates[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};
```

```
VkPipelineDynamicStateCreateInfo dynamicState = {};
dynamicState.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.dynamicStateCount = 2;
dynamicState.pDynamicStates = dynamicStates;
```

这样设置后会导致我们之前对这里使用的动态状态的设置被忽略掉，需要我们在进行绘制时重新指定它们的值。有关这一问题的细节，我们会在之后的章节进行更加详细地说明。如果我们不需要管线创建后进行任何状态的动态修改，可以将设置这一结构体指针的成员变量设置为 nullptr。

管线布局

我们可以在着色器中使用 uniform 变量，它可以在管线建立后动态地被应用程序修改，实现对着色器进行一定程度的动态配置。uniform 变量经常被用来传递变换矩阵给顶点着色器，以及传递纹理采样器句柄给片段着色器。

我们在着色器中使用的 uniform 变量需要在管线创建时使用 VkPipelineLayout 对象定义。暂时，我们不使用 uniform 变量，但我们仍需要创建一个 VkPipelineLayout 对象，指定空的管线布局。

添加一个成员变量存储 VkPipelineLayout 对象，在后面我们会用到它。

```
VkPipelineLayout pipelineLayout;
```

createGraphicsPipeline 函数中创建 VkPipelineLayout 对象：

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optional
pipelineLayoutInfo.pSetLayouts = nullptr; // Optional
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optional

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCC
    throw std::runtime_error("failed to create pipeline layout!");
}
```

可以通过 VkPipelineLayout 结构体指定可以在着色器中使用的常量值。最后，VkPipelineLayout 对象需要我们在应用程序结束前自己清除它：

```
void cleanup() {
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
        ...
}
```

总结

恭喜！到此为止，我们完成了管线的固定功能阶段的设置。虽然很繁琐，但我们对一切细节都变得了如指掌，避免了由于默认状态导致的令人苦恼的问题，所有状态都是我们自己设置的。

在创建图形管线前，我们还要对渲染流程进行设置。

本章节代码：

C++：

https://vulkan-tutorial.com/code/10_fixed_functions.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

渲染流程

配置

在进行管线创建之前，我们还需要设置用于渲染的帧缓冲附着。我们需要指定使用的颜色和深度缓冲，以及采样数，渲染操作如何处理缓冲的内容。所有这些信息被 Vulkan 包装为一个渲染流程对象，我们添加了一个叫做 createRenderPass 的函数来创建这一对象，然后在 initVulkan 函数中 createGraphicsPipeline 函数调用之前调用 createRenderPass 函数：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
}

        ...

void createRenderPass() {

}
```

附着描述

在这里，我们只使用了一个代表交换链图像的颜色缓冲附着。

```
void createRenderPass() {
    VkAttachmentDescription colorAttachment = {};
```

```
    colorAttachment.format = swapChainImageFormat;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
}
```

format 成员变量用于指定颜色缓冲附着的格式。samples 成员变量用于指定采样数，在这里，我们没有使用多重采样，所以将采样数设置为 1。

```
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
```

loadOp 和 storeOp 成员变量用于指定在渲染之前和渲染之后对附着中的数据进行的操作。对于 loadOp 成员变量，可以设置为下面这些值：

- VK_ATTACHMENT_LOAD_OP_LOAD：保持附着的现有内容
- VK_ATTACHMENT_LOAD_OP_CLEAR：使用一个常量值来清除附着的内容
- VK_ATTACHMENT_LOAD_OP_DONT_CARE：不关心附着现存的内容

在这里,我们设置 loadOp 成员变量的值为 VK_ATTACHMENT_LOAD_OP_CLEAR，在每次渲染新的一帧前使用黑色清除帧缓冲。storeOp 成员变量可以设置为下面这些值：

- VK_ATTACHMENT_STORE_OP_STORE：渲染的内容会被存储起来，以便之后读取
- VK_ATTACHMENT_STORE_OP_DONT_CARE：渲染后，不会读取帧缓冲的内容

```
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

loadOp 和 storeOp 成员变量的设置会对颜色和深度缓冲起效。stencilLoadOp 成员变量和 stencilStoreOp 成员变量会对模板缓冲起效。在这里，我们没有使用模板缓冲，所以设置对模板缓冲不关心即可。

```
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Vulkan 中的纹理和帧缓冲由特定像素格式的 VkImage 对象来表示。图像的像素数据在内存中的分布取决于我们要对图像进行的操作。

下面是一些常用的图形内存布局设置：

- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL：图像被用作颜色附着
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR：图像被用在交换链中进行呈现操作
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL：图像被用作复制操作的目的图像

我们会在之后的章节对上述设置进行更为详细地说明。在这里，我们需要做的是指定适合我们在之后进行的渲染操作的图像布局即可。

initialLayout 成员变量用于指定渲染流程开始前的图像布局方式。finalLayout 成员变量用于指定渲染流程结束后的图像布局方式。将 initialLayout 成员变量设置为 VK_IMAGE_LAYOUT_UNDEFINED 表示我们不关心之前的图像布局方式。使用这一值后，图像的内容不保证会被保留，但对于我们的应用程序，每次渲染前都要清除图像，所以这样的设置更符合我们的需求。对于 finalLayout 成员变量，我们设置为 VK_IMAGE_LAYOUT_PRESENT_SRC_KHR，使得渲染后的图像可以被交换链呈现。

子流程和附着引用

一个渲染流程可以包含多个子流程。子流程依赖于上一流程处理后的帧缓冲内容。比如，许多叠加的后期处理效果就是在上一次的处理结果上进行的。我们将多个子流程组成一个渲染流程后，Vulkan 可以对其进行一定程度的优化。对于我们这个渲染三角形的程序，我们只使用了一个子流程。

每个子流程可以引用一个或多个附着，这些引用的附着是通过 VkAttachmentReference 结构体指定的：

```
VkAttachmentReference colorAttachmentRef = {};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

attachment 成员变量用于指定要引用的附着在附着描述结构体数组中的索引。在这里，我们的 VkAttachmentDescription 数组只包含了一个附着信息，所以将 attachment 指定为 0 即可。layout 成员变量用于指定进行子流程时引用的附着使用的布局方式，Vulkan 会在子流程开始时自动将引用的附着转换到 layout 成员变量指定的图像布局。我们推荐将 layout 成员变量设置为 VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL，一般而言，它的性能表现最佳。

我们使用 VkSubpassDescription 结构体来描述子流程：

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

Vulkan 在未来也可能会支持计算子流程，所以，我们还需要显式地指定这是一个图形渲染的子流程。接着，我们指定引用的颜色附着：

```
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
```

这里设置的颜色附着在数组中的索引会被片段着色器使用，对应我们在片段着色器中使用的 layout(location = 0) out vec4 outColor 语句。下面是其它一些可以被子流程引用的附着类型：

- pInputAttachments：被着色器读取的附着
- pResolveAttachments：用于多重采样的颜色附着

- pDepthStencilAttachment：用于深度和模板数据的附着
- pPreserveAttachments：没有被这一子流程使用，但需要保留数据的附着

渲染流程

现在，我们已经设置好了附着和引用它的子流程，可以开车创建渲染流程对象。首先，我们在 pipelineLayout 变量定义之前添加一个 VkRenderPass 类型的成员变量：

```
VkRenderPass renderPass;
VkPipelineLayout pipelineLayout;
```

创建渲染流程对象需要填写 VkRenderPassCreateInfo 结构体，

```
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = &colorAttachment;
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;

if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {
    throw std::runtime_error("failed to create render pass!");
}
```

和管线布局对象一样，我们需要在应用程序结束前，清除渲染流程对象：

```
void cleanup() {
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyRenderPass(device, renderPass, nullptr);
        ...
}
```

下一章节，我们开始创建图形管线对象。

本章节代码：

C++：

https://vulkan-tutorial.com/code/11_render_passes.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

阶段总结

现在，让我们回忆之下我们为了创建图形管线而创建的对象：

- 着色器阶段：定义了着色器模块用于图形管线哪一可编程阶段
- 固定功能状态：定义了图形管线的固定功能阶段使用的状态信息，比如输入装配，视口，光栅化，颜色混合
- 管线布局：定义了被着色器使用，在渲染时可以被动态修改的 uniform 变量
- 渲染流程：定义了被管线使用的附着附着的用途

上面所有这些信息组合起来完整定义了图形管线的功能，现在，我们可以开始填写 VkGraphicsPipelineCreateInfo 结构体来创建管线对象。我们需要在 createGraphicsPipeline 函数的尾部，vkDestroyShaderModule 函数调用之前添加下面的代码：

```
VkGraphicsPipelineCreateInfo pipelineInfo = {};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
```

上面代码我们通过指定 VkPipelineShaderStageCreateInfo 结构体数组来引用之前我们创建的两个着色器阶段。

```
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optional
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.pDynamicState = nullptr; // Optional
```

接着，我们引用了之前设置的固定功能阶段信息。

```
pipelineInfo.layout = pipelineLayout;
```

指定之前创建的管线布局。

```
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
```

最后，引用之前创建的渲染流程对象和图形管线使用的子流程在子流程数组中的索引。在之后的渲染过程中，仍然可以使用其它与这个设置的渲染流程对象相兼容的渲染流程。

```
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE; // Optional
pipelineInfo.basePipelineIndex = -1; // Optional
```

basePipelineHandle 和 basePipelineIndex 成员变量用于以一个创建好的图形管线为基础创建一个新的图形管线。当要创建一个和已有管线大量设置相同的管线时，使用它的代价要比直接创建小，并且，对于从同一个管线衍生出的两个管线，在它们之间进行管线切换操作的效率也要高很多。我们可以使用 basePipelineHandle 来指定已经创建好的管线，或是使用 basePipelineIndex 来指定将要创建的管线作为基础管线，用于衍生新的管线。目前，我们只使用一个管线，所以将这两个成员变量分别

设置为 VK_NULL_HANDLE 和-1，不使用基础管线衍生新的管线。这两个成员变量的设置只有在 VkGraphicsPipelineCreateInfo 结构体的 flags 成员变量使用了 VK_PIPELINE_CREATE_DERIVATIVE_BIT 标记的情况下才会起效。

现在，让我们添加一个 VkPipeline 成员变量来存储创建的管线对象：

```
VkPipeline graphicsPipeline;
```

创建管线对象：

```
if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphicsPi
    throw std::runtime_error("failed to create graphics pipeline!");
}
```

vkCreateGraphicsPipelines 函数的参数要比一般的 Vulkan 的对象创建函数的参数多一些。它被设计成一次调用可以通过多个 VkGraphicsPipelineCreateInfo 结构体数据创建多个 VkPipeline 对象。

vkCreateGraphicsPipelines 函数第二个参数可以用来引用一个可选的 VkPipelineCache 对象。通过它可以将管线创建相关的数据进行缓存在多个 vkCreateGraphicsPipelines 函数调用中使用，甚至可以将缓存存入文件，在多个程序间使用。使用它可以加速之后的管线创建，我们会在之后的章节对它进行更为详细地说明。

我们需要在应用程序结束前，清除我们创建的管线对象：

```
void cleanup() {
    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
        ...
}
```

现在，编译运行程序，确保一切正常。到此，我们就成功创建了一个图形管线，接下来只需要设置好帧缓冲，提交绘制指令，就可以在屏幕上看到渲染的图像了。

本章节代码：

C++：

https://vulkan-tutorial.com/code/12_graphics_pipeline_complete.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment：

https://vulkan-tutorial.com/code/09_shader_base.frag

帧缓冲

在之前的章节我们已经介绍了一些有关帧缓冲的内容，并且配置了渲染流程信息。接下来，我们对帧缓冲进行配置。

我们在创建渲染流程对象时指定使用的附着需要绑定在帧缓冲对象上使用。帧缓冲对象引用了用于表示附着的 VkImageView 对象。对于我们的程序，我们只使用了一个颜色附着。但这并不意味着我们只需要使用一张图像，每个附着对应的图像个数依赖于交换链用于呈现操作的图像个数。我们需要为交换链中的每个图像创建对应的帧缓冲，在渲染时，渲染到对应的帧缓冲上。

添加一个向量作为成员变量来存储所有帧缓冲对象：

```
std::vector<VkFramebuffer> swapChainFramebuffers;
```

添加一个叫做 createFramebuffers 的函数来完成所有帧缓冲对象的创建，并在 initVulkan 函数中创建图形管线的操作之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
}

        ...

void createFramebuffers() {

}
```

分配足够的空间来存储所有帧缓冲对象：

```
void createFramebuffers() {
    swapChainFramebuffers.resize(swapChainImageViews.size());
}
```

为交换链的每一个图像视图对象创建对应的帧缓冲：

```
for (size_t i = 0; i < swapChainImageViews.size(); i++) {
    VkImageView attachments[] = {
        swapChainImageViews[i]
    };

    VkFramebufferCreateInfo framebufferInfo = {};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = 1;
    framebufferInfo.pAttachments = attachments;
```

```
        framebufferInfo.width = swapChainExtent.width;
        framebufferInfo.height = swapChainExtent.height;
        framebufferInfo.layers = 1;

        if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, &swapChainFramebuffers[i]) !=
            throw std::runtime_error("failed to create framebuffer!");
        }
    }
```

如读者所看到的，创建帧缓冲的操作非常直白。我们首先指定帧缓冲需要兼容的渲染流程对象。之后的渲染操作，我们可以使用与这个指定的渲染流程对象相兼容的其它渲染流程对象。一般来说，使用相同数量，相同类型附着的渲染流程对象是相兼容的。

attachmentCount 和 pAttachments 成员变量用于指定附着个数，以及渲染流程对象用于描述附着信息的 pAttachment 数组。

width 和 height 成员变量用于指定帧缓冲的大小，layers 成员变量用于指定图像层数。我们使用的交换链图像都是单层的，所以将 layers 成员变量设置为 1。

我们需要在应用程序结束前，清除图像视图和渲染流程对象前清除帧缓冲对象：

```
void cleanup() {
    for (auto framebuffer : swapChainFramebuffers) {
        vkDestroyFramebuffer(device, framebuffer, nullptr);
    }

        ...
}
```

至此，我们就创建好了帧缓冲对象。下一章节，我们开始编写真正的绘制指令。

本章节代码：

C++：

https://vulkan-tutorial.com/code/13_framebuffers.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

指令缓冲

Vulkan 下的指令，比如绘制指令和内存传输指令并不是直接通过函数调用执行的。我们需要将所有要执行的操作记录在一个指令缓冲对象，然后提交给可以执行这些操作的队列才能执行。这使得我们可以在程序初始化时就准备好所有要指定的指令序

列，在渲染时直接提交执行。也使得多线程提交指令变得更加容易。我们只需要在需要指定执行的使用，将指令缓冲对象提交给 Vulkan 处理接口。

指令池

在创建指令缓冲对象之前，我们需要先创建指令池对象。指令池对象用于管理指令缓冲对象使用的内存，并负责指令缓冲对象的分配。我们添加了一个 VkCommandPool 成员变量到类中：

```
VkCommandPool commandPool;
```

添加一个叫做 createCommandPool 的函数，并在 initVulkan 函数中帧缓冲对象创建之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
}

        ...

void createCommandPool() {

}
```

指令池对象的创建只需要填写两个参数：

```
QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

VkCommandPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily;
poolInfo.flags = 0; // Optional
```

指令缓冲对象在被提交给我们之前获取的队列后，被 Vulkan 执行。每个指令池对象分配的指令缓冲对象只能提交给一个特定类型的队列。在这里，我们使用的是绘制指令，它可以被提交给支持图形操作的队列。

有下面两种用于指令池对象创建的标记，可以提供有用的信息给 Vulkan 的驱动程序进行一定优化处理：

- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT: 使用它分配的指令缓冲对象被频繁用来记录新的指令 (使用这一标记可能会改变帧缓冲对象的内存分配策略)。
- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT: 指令缓冲对象之间相互独立，不会被一起重置。不使用这一标记，指令缓冲对象会被放在一起重置。

对于我们的程序，我们只在程序初始化时记录指令到指令缓冲对象，然后在程序的主循环中执行指令，所以，我们不使用上面这两个标记。

```
if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
}
```

我们通过调用 vkCreateCommandPool 函数来完成指令池对象的创建。应用程序结束前我们需要清除创建的指令池对象：

```
void cleanup() {
vkDestroyCommandPool(device, commandPool, nullptr);

        ...
}
```

分配指令缓冲

现在，我们可以开始分配指令缓冲对象，使用它记录绘制指令。由于绘制操作是在帧缓冲上进行的，我们需要为交换链中的每一个图像分配一个指令缓冲对象。为此，我们添加了一个数组作为成员变量来存储创建的 VkCommandBuffer 对象。指令缓冲对象会在指令池对象被清除时自动被清除，不需要我们自己显式地清除它。

```
std::vector<VkCommandBuffer> commandBuffers;
```

添加一个叫做 createCommandBuffers 的函数为交换链中的每一个图像创建指令缓冲对象：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
}
```

```
    ...

void createCommandBuffers() {
    commandBuffers.resize(swapChainFramebuffers.size());
}
```

指令缓冲对象可以通过调用 vkAllocateCommandBuffers 函数分配得到。调用这一函数需要填写 VkCommandBufferAllocateInfo 结构体来指定分配使用的指令池和需要分配的指令缓冲对象个数：

```
VkCommandBufferAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = (uint32_t) commandBuffers.size();

if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate command buffers!");
}
```

level 成员变量用于指定分配的指令缓冲对象是主要指令缓冲对象还是辅助指令缓冲对象：

- VK_COMMAND_BUFFER_LEVEL_PRIMARY：可以被提交到队列进行执行，但不能被其它指令缓冲对象调用。
- VK_COMMAND_BUFFER_LEVEL_SECONDARY：不能直接被提交到队列进行执行，但可以被主要指令缓冲对象调用执行。

在这里，我们没有使用辅助指令缓冲对象，但辅助治理给缓冲对象的好处是显而易见的，我们可以把一些常用的指令存储在辅助指令缓冲对象，然后在主要指令缓冲对象中调用执行。

记录指令到指令缓冲

我们通过调用 vkBeginCommandBuffer 函数开始指令缓冲的记录操作，这一函数以 VkCommandBufferBeginInfo 结构体作为参数来指定一些有关指令缓冲的使用细节。

```
for (size_t i = 0; i < commandBuffers.size(); i++) {
    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;
    beginInfo.pInheritanceInfo = nullptr; // Optional

    if (vkBeginCommandBuffer(commandBuffers[i], &beginInfo) != VK_SUCCESS) {
        throw std::runtime_error("failed to begin recording command buffer!");
```

```
    }
}
```

flags 成员变量用于指定我们将要怎样使用指令缓冲。它的值可以是下面这些：

- VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT:
  指令缓冲在执行一次后，就被用来记录新的指令。
- VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT:
  这是一个只在一个渲染流程内使用的辅助指令缓冲。
- VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT:
  在指令缓冲等待执行时，仍然可以提交这一指令缓冲。

在这里,我们使用了 VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT 标记，这使得我们可以在上一帧还未结束渲染时，提交下一帧的渲染指令。pInheritanceInfo 成员变量只用于辅助指令缓冲，可以用它来指定从调用它的主要指令缓冲继承的状态。

指令缓冲对象记录指令后，调用 vkBeginCommandBuffer 函数会重置指令缓冲对象。

开始渲染流程

调用 vkCmdBeginRenderPass 函数可以开始一个渲染流程。这一函数需要我们使用 VkRenderPassBeginInfo 结构体来指定使用的渲染流程对象：

```
VkRenderPassBeginInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
```

renderPass 成员变量用于指定使用的渲染流程对象，framebuffer 成员变量用于指定使用的帧缓冲对象。

```
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;
```

renderArea 成员变量用于指定用于渲染的区域。位于这一区域外的像素数据会处于未定义状态。通常，我们将这一区域设置为和我们使用的附着大小完全一样。

```
VkClearValue clearColor = {0.0f, 0.0f, 0.0f, 1.0f};
renderPassInfo.clearValueCount = 1;
renderPassInfo.pClearValues = &clearColor;
```

clearValueCount 和 pClearValues 成员变量用于指定使用 VK_ATTACHMENT_LOAD_OP_CLEAR 标记后，使用的清除值。在这里，我们使用完全不透明的黑色作为清除值。

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

所有可以记录指令到指令缓冲的函数的函数名都带有一个 vkCmd 前缀，并且这些函数的返回值都是 void，也就是说在指令记录操作完全结束前，不用进行任何错误处理。

这类函数的第一个参数是用于记录指令的指令缓冲对象。第二个参数是使用的渲染流程的信息。最后一个参数是用来指定渲染流程如何提供绘制指令的标记，它可以是下面这两个值之一：

- VK_SUBPASS_CONTENTS_INLINE：所有要执行的指令都在主要指令缓冲中，没有辅助指令缓冲需要执行。
- VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS：有来自辅助指令缓冲的指令需要执行。

由于我们没有使用辅助指令缓冲,所以我们使用 VK_SUBPASS_CONTENTS_INLINE。

基础绘制指令

现在，绑定图形管线：

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
```

vkCmdBindPipeline 函数的第二个参数用于指定管线对象是图形管线还是计算管线。至此，我们已经提交了需要图形管线执行的指令，以及片段着色器使用的附着，可以开始调用指令进行三角形的绘制操作：

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

我们使用 vkCmdDraw 函数来提交绘制操作到指令缓冲，它的第一个参数是记录有要执行的指令的指令缓冲对象，它的剩余参数依次是：

- vertexCount：尽管这里我们没有使用顶点缓冲，但仍然需要指定三个顶点用于三角形的绘制。
- instanceCount：用于实例渲染，为 1 时表示不进行实例渲染。
- firstVertex：用于定义着色器变量 gl_VertexIndex 的值。
- firstInstance：用于定义着色器变量 gl_InstanceIndex 的值。

结束渲染流程

接着，我们调用 vkCmdEndRenderPass 函数结束渲染流程：

```
vkCmdEndRenderPass(commandBuffers[i]);
```

然后，结束记录指令到指令缓冲：

```
if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {
    throw std::runtime_error("failed to record command buffer!");
}
```

下一章节，我们开始编写主循环的代码，从交换链获取图像，提交指令缓冲执行渲染指令，将渲染结果呈现到屏幕上。

本章节代码：

C++：

https://vulkan-tutorial.com/code/14_command_buffers.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

渲染和呈现

配置

这一章，我们开始编写在主循环中调用的 drawFrame 函数，这一函数调用会在屏幕上绘制一个三角形：

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }
}

        ...

void drawFrame() {

}
```

同步

我们编写的 drawFrame 函数用于执行下面的操作：

- 从交换链获取一张图像
- 对帧缓冲附着执行指令缓冲中的渲染指令
- 返回渲染后的图像到交换链进行呈现操作

上面这些操作每一个都是通过一个函数调用设置的，但每个操作的实际执行却是异步进行的。函数调用会在操作实际结束前返回，并且操作的实际执行顺序也是不确定的。而我们需要操作的执行能按照一定的顺序，所以就需要进行同步操作。

有两种用于同步交换链事件的方式：栅栏 (fence) 和信号量 (semaphore)。它们都可以完成同步操作。

栅栏 (fence) 和信号量 (semaphore) 的不同之处是，我们可以通过调用 vkWaitForFences 函数查询栅栏 (fence) 的状态，但不能查询信号量 (semaphore) 的状态。通常，我们使用栅栏 (fence) 来对应用程序本身和渲染操作进行同步。使用信号量 (semaphore) 来对一个指令队列内的操作或多个不同指令队列的操作进行同步。这

里，我们想要通过指令队列中的绘制操作和呈现操作，显然，使用信号量 (semaphore) 更加合适。

信号量

在这里，我们需要两个信号量，一个信号量发出图像已经被获取，可以开始渲染的信号；一个信号量发出渲染已经结果，可以开始呈现的信号。我们添加了两个信号量对象作为成员变量：

```
VkSemaphore imageAvailableSemaphore;
VkSemaphore renderFinishedSemaphore;
```

添加 createSemaphores 函数用于创建上面这两个信号量对象：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
    createSemaphores();
}

        ...

void createSemaphores() {

}
```

创建信号量，需要填写 VkSemaphoreCreateInfo 结构体，但对于目前版本的 Vulkan 来说，这一结构体只有一个 sType 成员变量需要我们填写：

```
void createSemaphores() {
    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
}
```

未来版本的 Vulkan 或扩展可能会添加新的功能设置到这一结构体的 flags 和 pNext 成员变量。

```
if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphore) != VK_SUCCE
    throw std::runtime_error("failed to create semaphores!");
```

```
}
```

信号量 (semaphore) 需要我们在应用程序结束前，所有它所同步的指令执行结束后，对它进行清除：

```
void cleanup() {
    vkDestroySemaphore(device, renderFinishedSemaphore, nullptr);
    vkDestroySemaphore(device, imageAvailableSemaphore, nullptr);
```

从交换链获取图像

之前提到，我们在 drawFrame 函数中进行的第一个操作是从交换链获取一张图像。这可以通过调用 vkAcquireNextImageKHR 函数完成，可以看到 vkAcquireNextImageKHR 函数的函数名带有一个 KHR 后缀，这是因为交换链是一个扩展特性，所以与它相关的操作都会有 KHR 这一扩展后缀：

```
void drawFrame() {
    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::max(),
    imageAvailableSemaphore, VK_NULL_HANDLE, &imageIndex);
}
```

vkAcquireNextImageKHR 函数的第一个参数是使用的逻辑设备对象，第二个参数是我们要获取图像的交换链，第三个参数是图像获取的超时时间，我们可以通过使用无符号 64 位整型所能表示的最大整数来禁用图像获取超时。

接下来的两个函数参数用于指定图像可用后通知的同步对象，可以指定一个信号量对象或栅栏对象，或是同时指定信号量和栅栏对象进行同步操作。在这里，我们指定了一个叫做 imageAvailableSemaphore 的信号量对象。

vkAcquireNextImageKHR 函数的最后一个参数用于输出可用的交换链图像的索引，我们使用这个索引来引用我们的 swapChainImages 数组中的 VkImage 对象，并使用这一索引来提交对应的指令缓冲。

提交指令缓冲

我们通过 VkSubmitInfo 结构体来提交信息给指令队列：

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
```

VkSubmitInfo 结构体的 waitSemaphoreCount、pWaitSemaphores 和 pWaitDstStageMask 成员变量用于指定队列开始执行前需要等待的信号量，以及需要等待的管线阶段。这里，我们需要写入颜色数据到图像，所以我们指定等待图像管线到达可

以写入颜色附着的管线阶段。waitStages 数组中的条目和 pWaitSemaphores 中相同索引的信号量相对应。

```
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
```

commandBufferCount 和 pCommandBuffers 成员变量用于指定实际被提交执行的指令缓冲对象。之前提到，我们应该提交和我们刚刚获取的交换链图像相对应的指令缓冲对象。

```
VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

signalSemaphoreCount 和 pSignalSemaphores 成员变量用于指定在指令缓冲执行结束后发出信号的信号量对象。在这里，我们使用 renderFinishedSemaphore 信号量对象在指令缓冲执行结束后发出信号。

```
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}
```

现在我们可以调用 vkQueueSubmit 函数提交指令缓冲给图形指令队列。vkQueueSubmit 函数使用 vkQueueSubmit 结构体数组作为参数，可以同时大批量提交数据。vkQueueSubmit 函数的最后一个参数是一个可选的栅栏对象，可以用它同步提交的指令缓冲执行结束后要进行的操作。在这里，我们使用信号量进行同步，没有使用它，将其设置为 VK_NULL_HANDLE。

子流程依赖

渲染流程的子流程会自动进行图像布局变换。这一变换过程由子流程的依赖所决定。子流程的依赖包括子流程之间的内存和执行的依赖关系。虽然我们现在只使用了一个子流程，但子流程执行之前和子流程执行之后的操作也被算作隐含的子流程。

在渲染流程开始和结束时会自动进行图像布局变换，但在渲染流程开始时进行的自动变换的时机和我们的需求不符，变换发生在管线开始时，但那时我们可能还没有获取到交换链图像。有两种方式可以解决这个问题。一个是设置 imageAvailableSemaphore 信号量的 waitStages 为 VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT，确保渲染流程在我们获取交换链图像之前不会开始。一个是设置渲染流程等待 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT 管线阶段。在这里，为了让读者能够了解子流程依赖如何控制图像布局变换，我们使用第二种方式。

配置子流程依赖需要使用 VkSubpassDependency 结构体。让我们在 createRenderPass 函数添加下面的代码：

```
VkSubpassDependency dependency = {};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
```

srcSubpass 和 dstSubpass 成员变量用于指定被依赖的子流程的索引和依赖被依赖的子流程的索引。VK_SUBPASS_EXTERNAL 用来指定我们之前提到的隐含的子流程，对 srcSubpass 成员变量使用表示渲染流程开始前的子流程，对 dstSubpass 成员使用表示渲染流程结束后的子流程。这里使用的索引 0 是我们之前创建的子流程的索引。为了避免出现循环依赖，我们给 dstSubpass 设置的值必须始终大于 srcSubpass。

```
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
```

srcStageMask 和 srcAccessMask 成员变量用于指定需要等待的管线阶段和子流程将进行的操作类型。我们需要等待交换链结束对图像的读取才能对图像进行访问操作，也就是等待颜色附着输出这一管线阶段。

```
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_
```

dstStageMask 和 dstAccessMask 成员变量用于指定需要等待的管线阶段和子流程将进行的操作类型。在这里，我们的设置为等待颜色附着的输出阶段，子流程将会进行颜色附着的读写操作。这样设置后，图像布局变换直到必要时才会进行：当我们开始写入颜色数据时。

```
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

VkRenderPassCreateInfo 结构体的 dependencyCount 和 pDependencies 成员变量用于指定渲染流程使用的依赖信息。

呈现

渲染操作执行后，我们需要将渲染的图像返回给交换链进行呈现操作。我们在 drawFrame 函数的尾部通过 VkPresentInfoKHR 结构体来配置呈现信息：

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = signalSemaphores;
```

waitSemaphoreCount 和 pWaitSemaphores 成员变量用于指定开始呈现操作需要等待的信号量。

```
VkSwapchainKHR swapChains[] = {swapChain};
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;
```

接着，我们指定了用于呈现图像的交换链，以及需要呈现的图像在交换链中的索引。

```
presentInfo.pResults = nullptr; // Optional
```

我们可以通过 pResults 成员变量获取每个交换链的呈现操作是否成功的信息。在这里，由于我们只使用了一个交换链，可以直接使用呈现函数的返回值来判断呈现操作是否成功，没有必要使用 pResults。

```
vkQueuePresentKHR(presentQueue, &presentInfo);
```

调用 vkQueuePresentKHR 函数可以请求交换链进行图像呈现操作。在下一章节，我们会对 vkAcquireNextImageKHR 函数和 vkQueuePresentKHR 函数添加错误处理的代码，应对调用它们失败后的情况。

现在如果编译运行程序，当我们关闭应用程序窗口时，我们的程序直接就奔溃了。如果开启了校验层，我们可以从控制台窗口看到调试回调函数打印的错误信息。

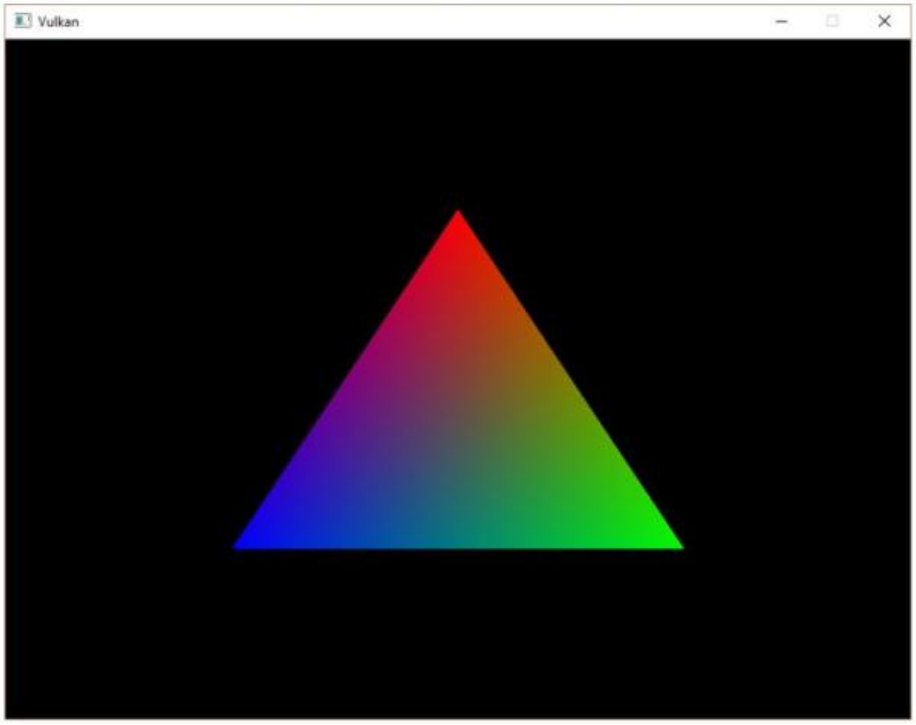

Figure 36: image

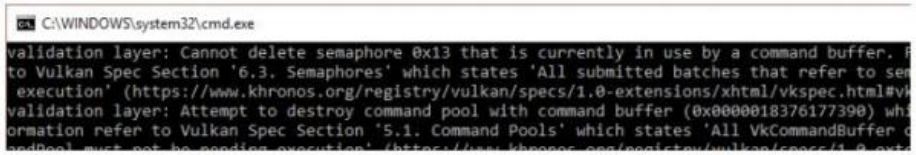

Figure 37: image

造成这一问题的原因是 drawFrame 函数中的操作是异步执行的。这意味着我们关闭应用程序窗口跳出主循环时，绘制操作和呈现操作可能仍在继续执行，这与我们紧接着进行的清除操作也是冲突的。

我们应该等待逻辑设备的操作结束执行才能销毁窗口：

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }

    vkDeviceWaitIdle(device);
}
```

我们可以使用 vkQueueWaitIdle 函数等待一个特定指令队列结束执行。现在再次编译运行程序，关闭应用程序窗口就不会造成程序直接崩溃了。

多帧并行渲染

如果读者开启校验层后运行程序，观察应用程序的内存使用情况，可以发现我们的应用程序的内存使用量一直在慢慢增加。这是由于我们的 drawFrame 函数以很快地速度提交指令，但却没有在下一次指令提交时检查上一次提交的指令是否已经执行结束。也就是说 CPU 提交指令快过 GPU 对指令的处理速度，造成 GPU 需要处理的指令大量堆积。更糟糕的是这种情况下，我们实际上对多个帧同时使用了相同的 imageAvailableSemaphore 和 renderFinishedSemaphore 信号量。

最简单的解决上面这一问题的方法是使用 vkQueueWaitIdle 函数来等待上一次提交的指令结束执行，再提交下一帧的指令：

```
void drawFrame() {
        ...

    vkQueuePresentKHR(presentQueue, &presentInfo);

    vkQueueWaitIdle(presentQueue);
}
```

但这样做，是对 GPU 计算资源的大大浪费。图形管线可能大部分时间都处于空闲状态。为了充分利用 GPU 的计算资源，现在我们扩展我们的应用程序，让它可以同时渲染多帧。

首先，我们在源代码的头部添加一个常量来定义可以同时并行处理的帧数：

```
const int MAX_FRAMES_IN_FLIGHT = 2;
```

为了避免同步干扰，我们为每一帧创建属于它们自己的信号量：

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
```

我们需要对 createSemaphores 函数进行修改来创建每一帧需要的信号量对象：

```
void createSemaphores() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);

    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i])
        vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphores[i]) !=

            throw std::runtime_error("failed to create semaphores for a frame!");
        }
    }
}
```

在应用程序结束前，我们需要清除为每一帧创建的信号量：

```
void cleanup() {
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
    }

        ...
}
```

我们添加一个叫做 currentFrame 的变量来追踪当前渲染的是哪一帧。之后，我们通过这一变量来选择当前帧应该使用的信号量：

```
size_t currentFrame = 0;
```

修改 drawFrame 函数，使用正确的信号量对象：

```
void drawFrame() {
    vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::max(), imageAvai

        ...

    VkSemaphore waitSemaphores[] = {imageAvailableSemaphores[currentFrame]};

        ...

    VkSemaphore signalSemaphores[] = {renderFinishedSemaphores[currentFrame]};

        ...
}
```

最后，不要忘记更新 currentFrame 变量：

```
void drawFrame() {
        ...

    currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
}
```

上面代码，我们使用模运算 (%) 来使 currentFrame 变量的值在 0 到 MAX_FRAMES_IN_FLIGHT-1 之间进行循环。我们还需要使用栅栏 (fence) 来进行 CPU 和 GPU 之间的同步，来防止有超过 MAX_FRAMES_IN_FLIGHT 帧的指令同时被提交执行。栅栏 (fence) 和信号量 (semaphore) 类似，可以用来发出信号和等待信号。我们为每一帧创建一个 VkFence 对象：

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;
```

将 createSemaphores 函数修改为 createSyncObjects 函数来在一个函数中创建所有同步对象：

```
void createSyncObjects() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);

    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

    VkFenceCreateInfo fenceInfo = {};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i])

            throw std::runtime_error("failed to create synchronization objects for a frame!"
        }
    }
}
```

在应用程序结束前，清除我们创建的 VkFence 对象：

```
void cleanup() {
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }
```

```
        ...
}
```

修改 drawFrame 函数使用我们创建的栅栏 (fence) 对象进行同步。vkQueueSubmit 函数有一个可选的参数可以用来指定在指令缓冲执行结束后需要发起信号的栅栏 (fence) 对象。我们通过它来发起一帧结束执行的信号。

```
void drawFrame() {
        ...

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUC
        throw std::runtime_error("failed to submit draw command buffer!");
    }
        ...
}
```

现在需要我们修改 drawFrame 函数来等待我们当前帧所使用的指令缓冲结束执行:

```
void drawFrame() {
    vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, std::numeric_limits<u
    vkResetFences(device, 1, &inFlightFences[currentFrame]);

        ...
}
```

vkWaitForFences 函数可以用来等待一组栅栏 (fence) 中的一个或全部栅栏 (fence) 发出信号。上面代码中我们对它使用的 VK_TRUE 参数用来指定它等待所有在数组中指定的栅栏 (fence)。我们现在只有一个栅栏 (fence) 需要等待，所以不使用 VK_TRUE 也是可以的。和 vkAcquireNextImageKHR 函数一样，vkWaitForFences 函数也有一个超时参数。和信号量不同，等待栅栏发出信号后，我们需要调用 vkResetFences 函数手动将栅栏 (fence) 重置为未发出信号的状态。

现在编译运行程序，读者可能会感到奇怪。应用程序没有呈现出我们渲染的三角形。启用校验层后运行程序，我们在控制台窗口看到下面这样的信息:

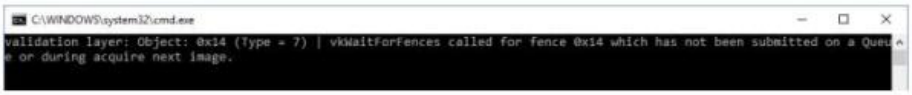

Figure 38: image

出现这一问题的原因是，默认情况下，栅栏 (fence) 对象在创建后是未发出信号的状态。这就意味着如果我们没有在 vkWaitForFences 函数调用之前发出栅栏 (fence) 信号，vkWaitForFences 函数调用将会一直处于等待状态。我们可以在创建栅栏 (fence) 对象时，设置它的初始状态为已发出信号来解决这一问题:

```
void createSyncObjects() {
        ...
```

```
    VkFenceCreateInfo fenceInfo = {};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

        ...
}
```

现在可以重新编译运行程序，内存泄漏问题应该已经不见了！我们已经通过同步机制确保不会有超过我们设定数量的帧会被异步执行。对于其它需要同步的地方，比如 cleanup 函数，使用 vkDeviceWaitIdle 函数的效果也足够好，不需要使用栅栏或信号量。

总结

到此为止，我们大概编写了 900 多行代码，终于可以在屏幕上看到自己用代码渲染的三角形。如此繁琐的设置跟 Vulkan 的一切都需要显式地指定有关。读者可以花点时间重新阅读自己编写的代码，对代码中我们设置的对象建立更加清晰的认知。在之后的章节，我们会在目前的基础上逐步扩展，让大家对 Vulkan 有更加深刻的认识。

下一章节，我们将对目前的程序进行一些细节上的改进。

本章节代码：

C++：

https://vulkan-tutorial.com/code/15_hello_triangle.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

重建交换链

介绍

我们已经成功地编写代码使用 Vulkan 在屏幕上绘制出了一个三角形，但这个程序还有许多细节问题我们没有处理。比如，窗口大小改变会导致交换链和窗口不再适配，我们需要重新对交换链进行处理。

重建交换链

我们添加一个叫做 recreateSwapChain 的函数，它会调用 createSwapChain 函数和其它一些依赖于交换链和窗口大小的对象的创建函数：

```
void recreateSwapChain() {
    vkDeviceWaitIdle(device);
```

```
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandBuffers();
}
```

上面代码，我们首先调用 vkDeviceWaitIdle 函数等待设备处于空闲状态，避免在对象的使用过程中将其清除重建。接着，我们重新创建了交换链。图形视图是直接依赖于交换链图像的，所以也需要被重建图像视图。渲染流程依赖于交换链图像的格式，虽然像窗口大小改变不会引起使用的交换链图像格式改变，但我们还是应该对它进行处理。视口和裁剪矩形在管线创建时被指定，窗口大小改变，这些设置也需要修改，所以我们也需要重建管线。实际上，我们可以通过使用动态状态来设置视口和裁剪矩形来避免重建管线。帧缓冲和指令缓冲直接依赖于交换链图像，也需要重建。

我们需要在重建前，清除之前使用的对象，所以我们将交换链相关的清除代码从 cleanup 函数中分离出来作为一个独立的 cleanupSwapChain 函数，然后在 recreateSwapChain 函数中调用它：

```
void cleanupSwapChain() {

}

void recreateSwapChain() {
    vkDeviceWaitIdle(device);

    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandBuffers();
}
```

将交换链相关的清除代码从 cleanup 中移出到 cleanupSwapChain 函数中：

```
void cleanupSwapChain() {
    for (size_t i = 0; i < swapChainFramebuffers.size(); i++) {
        vkDestroyFramebuffer(device, swapChainFramebuffers[i], nullptr);
    }

    vkFreeCommandBuffers(device, commandPool,
    static_cast<uint32_t>(commandBuffers.size()), commandBuffers.data());
```

```
    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyRenderPass(device, renderPass, nullptr);

    for (size_t i = 0; i < swapChainImageViews.size(); i++) {
        vkDestroyImageView(device, swapChainImageViews[i], nullptr);
    }

    vkDestroySwapchainKHR(device, swapChain, nullptr);
}

void cleanup() {
    cleanupSwapChain();

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }

    vkDestroyCommandPool(device, commandPool, nullptr);

    vkDestroyDevice(device, nullptr);

    if (enableValidationLayers) {
        DestroyDebugReportCallbackEXT(instance, callback, nullptr);
    }

    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

对于指令池对象，我们不需要重建，只需要调用 vkFreeCommandBuffers 函数清除它分配的指令缓冲对象即可。

窗口大小改变后，我们需要重新设置交换链图像的大小，这一设置可以通过修改 chooseSwapExtent 函数，让它设置交换范围为当前的帧缓冲的实际大小，然后我们在需要的地方调用 chooseSwapExtent 函数即可：

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR &capabilities) {
    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    } else {
```

```
        int width, height;
        glfwGetFramebufferSize(window, &width, &height);

        VkExtent2D actualExtent = {
            static_cast<uint32_t>(width),
            static_cast<uint32_t>(height)
        };

        ...
    }
}
```

至此，我们就完成了交换链重建的所有工作！但是，我们使用的这一重建方法需要等待正在执行的所有设备操作结束才能进行。实际上，是可以在渲染操作执行，原来的交换链仍在使用时重建新的交换链，只需要在创建新的交换链时使用 VkSwapchainCreateInfoKHR 结构体的 oldSwapChain 成员变量引用原来的交换链即可。之后，在旧的交换链结束使用时就可以清除它。

交换链不完全匹配和交换链过期

现在我们只需要在交换链必须被重建时调用 recreateSwapChain 函数重建交换链即可。我们可以根据 vkAcquireNextImageKHR 和 vkQueuePresentKHR 函数返回的信息来判定交换链是否需要重建：

- VK_ERROR_OUT_OF_DATE_KHR：交换链不能继续使用。通常发生在窗口大小改变后。
- VK_SUBOPTIMAL_KHR：交换链仍然可以使用，但表面属性已经不能准确匹配。

```
VkResult result = vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::ma
imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

if (result == VK_ERROR_OUT_OF_DATE_KHR) {
recreateSwapChain();
return;
} else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
throw std::runtime_error("failed to acquire swap chain image!");
}
recreateSwapChain();
return;
} else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
throw std::runtime_error("failed to acquire swap chain image!");
}
```

当交换链过期时，我们就不能再使用它，必须重建交换链。

但是，如果在获取交换链不可继续使用后，立即跳出这一帧的渲染，会导致我们使用的栅栏 (fence) 处于我们不能确定得状态。所以，我们应该在重建交换链时，重置栅

栏 (fence) 对象，这可以通过调用 vkResetFences 函数完成：

```
vkResetFences(device, 1, &inFlightFences[currentFrame]);

if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS
    throw std::runtime_error("failed to submit draw command buffer!");
}
```

我们可以在交换链不完全匹配时进行一些处理，在这里，我们没有选择在交换链不完全匹配时中断这一帧的渲染，毕竟，这种情况下，我们实际取得了可以用来绘制的交换链的图像。可以认为 VK_SUCCESS 和 VK_SUBOPTIMAL_KHR 都说明成功获取到了交换链图像：

```
result = vkQueuePresentKHR(presentQueue, &presentInfo);

if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR) {
    recreateSwapChain();
} else if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to present swap chain image!");
}

currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
```

vkQueuePresentKHR 函数的返回值与 vkAcquireNextImageKHR 函数的返回值有着相同的意义。在这里，为了保证最佳渲染效果，我们选择在交换链不完全匹配时也重建交换链。

显式处理窗口大小改变

尽管许多驱动程序会在窗口大小改变后触发 VK_ERROR_OUT_OF_DATE_KHR 信息，但这种触发并不可靠，所以我们最好添加一些代码来显式地在窗口大小改变时重建交换链。我们添加一个新的变量来标记窗口大小是否发生改变：

```
std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;

bool framebufferResized = false;
```

修改 drawFrame 函数，检测我们加入的标记：

```
if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR ||
framebufferResized) {
    framebufferResized = false;
    recreateSwapChain();
} else if (result != VK_SUCCESS) {
        ...
}
```

使用 glfwSetFramebufferSizeCallback 函数设置处理窗口大小改变的回调函数：

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
}

static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {

}
```

需要注意代码中，我们定义 framebufferResizeCallback 为静态函数，这样才能将其用作回调函数。framebufferResizeCallback 回调函数有一个 GLFWwindow 类型的参数，我们可以使用这一参数来引用 GLFW 窗口。GLFW 允许我们将任意的指针使用 glfwSetWindowUserPointer 函数存储在 GLFW 窗口相关的数据中：

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
glfwSetWindowUserPointer(window, this);
glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
```

我们可以通过 glfwGetWindowUserPointer 函数获取使用 glfwSetWindowUserPointer 函数存储的指针，在这里我们使用它们来存储应用程序类的 this 指针：

```
static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
    auto app = reinterpret_cast<HelloTriangleApplication*>(glfwGetWindowUserPointer(window));
    app->framebufferResized = true;
}
```

现在编译运行程序，改变应用程序窗口大小，观察帧缓冲是否随着窗口大小改变而被正确地重新设置。

处理窗口最小化

还有一种特殊情况需要处理，这就是窗口的最小化。这时窗口的帧缓冲实际大小为 0。在这里，我们设置应用程序在窗口最小化后停止渲染，直到窗口重新可见时重建交换链：

```
void recreateSwapChain() {
    int width = 0, height = 0;
    while (width == 0 || height == 0) {
        glfwGetFramebufferSize(window, &width, &height);
        glfwWaitEvents();
    }

    vkDeviceWaitIdle(device);
```

```
        ...
}
```

现在，我们已经将编写的程序修改得更加完善。下一章节，我们将使用顶点缓冲来替换我们之前在顶点着色器中硬编码的顶点数据。

本章节代码：

C++：

https://vulkan-tutorial.com/code/16_swap_chain_recreation.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/09_shader_base.vert

Fragment Shader：

https://vulkan-tutorial.com/code/09_shader_base.frag

顶点输入描述

介绍

在接下来的章节，我们会将之前在顶点着色器中直接硬编码的顶点数据替换为顶点缓冲来定义。我们首先创建一个 CPU 的缓冲，然后将我们要使用的顶点数据先复制到这个 CPU 缓冲中，最后，我们复制 CPU 缓冲中的数据到阶段缓冲。

顶点着色器

修改顶点着色器，去掉代码中包含的硬编码顶点数据。使用 in 关键字使用顶点缓冲中的顶点数据：

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

上面代码中的 inPosition 和 inColor 变量是顶点属性。它们代表了顶点缓冲中的每个顶点数据，就跟我们使用数组定义的顶点数据是一样的。重新编译顶点着色器，保证没有出现问题。

layout(location = x) 用于指定变量在顶点数据中的索引。特别需要注意，对于 64 位变量，比如 dvec3 类型的变量，它占用了不止一个索引位置，我们在定义这种类型的顶点属性变量之后的顶点变量，要注意索引号的增加并不是 1：

```
layout(location = 0) in dvec3 inPosition;
layout(location = 2) in vec3 inColor;
```

读者可以查阅 OpenGL 的官方文档获得关于 layout 修饰符的更多信息。

顶点数据

我们将顶点数据从顶点着色器代码移动到我们的应用程序的代码中。我们首先包含 GLM 库的头文件，它提供了我们需要使用的线性代数库：

```
#include <glm/glm.hpp>
```

创建一个叫做 Vertex 的新的结构体类型，我们使用它来定义单个顶点的数据：

```
struct Vertex {
        glm::vec2 pos;
        glm::vec3 color;
};
```

GLM 库提供了能够完全兼容 GLSL 的 C++ 向量类型：

```
const std::vector<Vertex> vertices = {
        {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
        {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
        {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

现在，我们可以使用 Vertex 结构体数组来定义我们的顶点数据。这次定义和之前在顶点着色器中进行的略有不同，之前在顶点着色器中，我们将顶点位置和顶点颜色数据定义在了不同的数组中，这次，我们将它们定义在了同一个结构体数组中。这种顶点属性定义方式定义的顶点数据也被叫做交叉顶点属性 (interleaving vertex attributes)。

绑定描述

定义好顶点数据，我们需要将 CPU 缓冲 (也就是我们定义的 Vertex 结构体数组) 中顶点数据的存放方式传递给 GPU，以便 GPU 可以正确地加载这些数据到显存中。我们给 Vertex 结构体添加一个叫做 getBindingDescription 的静态成员函数，在里面编写代码返回 Vertex 结构体的顶点数据存放方式：

本章节代码：

C++：

https://vulkan-tutorial.com/code/17_vertex_input.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.vert

Fragment Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.frag

创建顶点缓冲

介绍

Vulkan 的缓冲是可以存储任意数据的可以被显卡读取的内存。缓冲除了用来存储顶点数据，还有很多其它用途。和之前我们见到的 Vulkan 对象不同，缓冲对象并不自动地为它们自己分配内存。

创建缓冲

添加一个叫做 createVertexBuffer 地函数，然后在 initVulkan 函数中 createCommandBuffers 函数调用之后调用它：

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createVertexBuffer();
    createCommandBuffers();
    createSyncObjects();
}

        ...

void createVertexBuffer() {

}
```

填写 VkBufferCreateInfo 结构体：

```
VkBufferCreateInfo bufferInfo = {};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = sizeof(vertices[0]) * vertices.size();
```

size 成员变量用于指定要创建的缓冲所占字节大小。我们可以通过 sizeof 来计算顶点数据数组的字节大小。

```
bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
```

usage 成员变量用于指定缓冲中的数据的使用目的。可以使用位或运算来指定多个使用目的。在这里，我们将缓冲用作存储顶点数据，其它使用目的我们会在之后的章节进行介绍。

```
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

和交换链图像一样，缓冲可以被特定的队列族所拥有，也可以同时在多个队列族之前共享。在这里，我们只使用了一个队列，所以选择使用独有模式。

flags 成员变量用于配置缓冲的内存稀疏程度，我们将其设置为 0 使用默认值。

填写完结构体信息，我们就可以调用 vkCreateBuffer 函数来完成缓冲创建。我们添加一个类成员变量来存储创建的缓冲的句柄：

```
VkBuffer vertexBuffer;

        ...

void createVertexBuffer() {
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = sizeof(vertices[0]) * vertices.size();
    bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &vertexBuffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create vertex buffer!");
    }
}
```

缓冲不依赖交换链，所以我们不需要在交换链重建时重建缓冲。应用程序结束时，我们需要清除我们创建的缓冲对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, vertexBuffer, nullptr);

        ...
}
```

内存需求

缓冲创建后实际还没有给它分配任何内存。分配缓冲内存前，我们需要调用 vkGetBufferMemoryRequirements 函数获取缓冲的内存需求。

```
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
```

vkGetBufferMemoryRequirements 函数返回的 VkMemoryRequirements 结构体有下面这三个成员变量：

- size：缓冲需要的内存的字节大小，它可能和 bufferInfo.size 的值不同。
- alignment：缓冲在实际被分配的内存中的开始位置。它的值依赖于 bufferInfo.usage 和 bufferInfo.flags。
- memoryTypeBIts：指示适合该缓冲使用的内存类型的位域。

显卡可以分配不同类型的内存作为缓冲使用。不同类型的内存所允许进行的操作以及操作的效率有所不同。我们需要结合自己的需求选择最合适的内存类型使用。我们添加一个叫做 findMemoryType 的函数来做件事：

```
uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {

}
```

首先，我们使用 vkGetPhysicalDeviceMemoryProperties 函数查询物理设备可用的内存类型：

```
VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
```

vkGetPhysicalDeviceMemoryProperties 函数返回的 VkPhysicalDeviceMemoryProperties 结构体包含了 memoryTypes 和 memoryHeaps 两个数组成员变量。memoryHeaps 数组成员变量中的每个元素是一种内存来源，比如显存以及显存用尽后的位于主内存种的交换空间。现在，我们暂时不关心这些内存的来源。

遍历数组，查找缓冲可用的内存类型：

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if (typeFilter & (1 << i)) {
        return i;
    }
}

throw std::runtime_error("failed to find suitable memory type!");
```

typeFilter 参数用于指定我们需要的内存类型的位域。我们只需要遍历可用内存类型数组，检测每个内存类型是否满足我们需要即可 (相应位域为 1)。

我们需要位域满足 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT(用于从 CPU 写入数据) 和 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT(后面介绍原因) 的内存类型。

修改代码，检测我们满足我们需要的内存类型：

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].propertyFlags & properties)
        return i;
    }
}
```

由于我们不只一个需要的内存属性，所以仅仅检测位与运算的结果是否非 0 是不够的，还需要检测它是否与我们需要的属性的位域完全相同。

内存分配

确定好使用的内存类型后，我们可以填写 VkMemoryAllocateInfo 结构体来分配需要的内存：

```
VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryTyp(memRequirements.memoryTypeBits, VK_MEMORY_PROPERTY
```

内存分配只需要填写好需要的内存大小和内存类型，然后调用 vkAllocateMemory 函数分配内存即可：

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;

        ...

if (vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate vertex buffer memory!");
}
```

如果内存分配成功，我们就可以使用 vkBindBufferMemory 函数将分配的内存和缓冲对象进行关联：

```
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
```

vkBindBufferMemory 函数的前三个参数非常直白，第四个参数是偏移值。这里我们将内存用作顶点缓冲，可以将其设置为 0。偏移值需要满足能够被 memRequirements.alignment 整除。

和 C++ 的动态内存分配一样，这里分配的内存需要我们自己进行释放。通常我们在缓冲不再使用时，释放它所关联的内存：

```
void cleanup() {
cleanupSwapChain();

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
```

填充顶点缓冲

现在，我们可以开始将顶点数据复制到缓冲中。我们需要使用 vkMapMemory 函数将缓冲关联的内存映射到 CPU 可以访问的内存：

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
```

vkMapMemory 函数允许我们通过给定的内存偏移值和内存大小访问特定的内存资源。这里我们使用的偏移值和内存大小分别时 0 和 bufferInfo.size。还有一个特殊值 VK_WHOLE_SIZE 可以用来映射整个申请的内存。vkMapMemory 函数的倒数第二个参数可以用来指定一个标记，但对于目前版本的 Vulkan 来说，这个参数还没有可以使用的标记，必须将其设置为 0。最后一个参数用于返回内存映射后的地址。

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
memcpy(data, vertices.data(), (size_t) bufferInfo.size);
vkUnmapMemory(device, vertexBufferMemory);
```

现在可以使用 memcpy 将顶点数据复制到映射后的内存，然后调用 vkUnmapMemory 函数来结束内存映射。然而，驱动程序可能并不会立即复制数据到缓冲关联的内存中去，这是由于现代处理器都存在缓存这一设计，写入内存的数据并不一定在多个核心同时可见，有下面两种方法可以保证数据被立即复制到缓冲关联的内存中去：

- 使用带有 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT 属性的内存类型，保证内存可见的一致性
- 写入数据到映射的内存后，调用 vkFlushMappedMemoryRanges 函数，读取映射的内存数据前调用 vkInvalidateMappedMemoryRanges 函数

在这里，我们使用第一种方法，它可以保证映射的内存的内容和缓冲关联的内存的内容一致。但使用这种方式，会比第二种方式些许降低性能表现。

绑定顶点缓冲

扩展 createCommandBuffers 函数，使用顶点缓冲进行渲染操作：

```
vkCmdBindPipeline(commandBuffers[i],
VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdDraw(commandBuffers[i],
static_cast<uint32_t>(vertices.size()), 1, 0, 0);
```

我们使用 vkCmdBindVertexBuffers 函数来绑定顶点缓冲。它的第二和第三个参数用于指定偏移值和我们要绑定的顶点缓冲的数量。它的最后两个参数用于指定需要绑定的顶点缓冲数组以及顶点数据在顶点缓冲中的偏移值数组。我们还需要修改 vkCmdDraw 函数调用，使用顶点缓冲中的顶点个数替换之前硬编码的数字 3。

现在编译运行程序，我们可以看到下面的窗口：

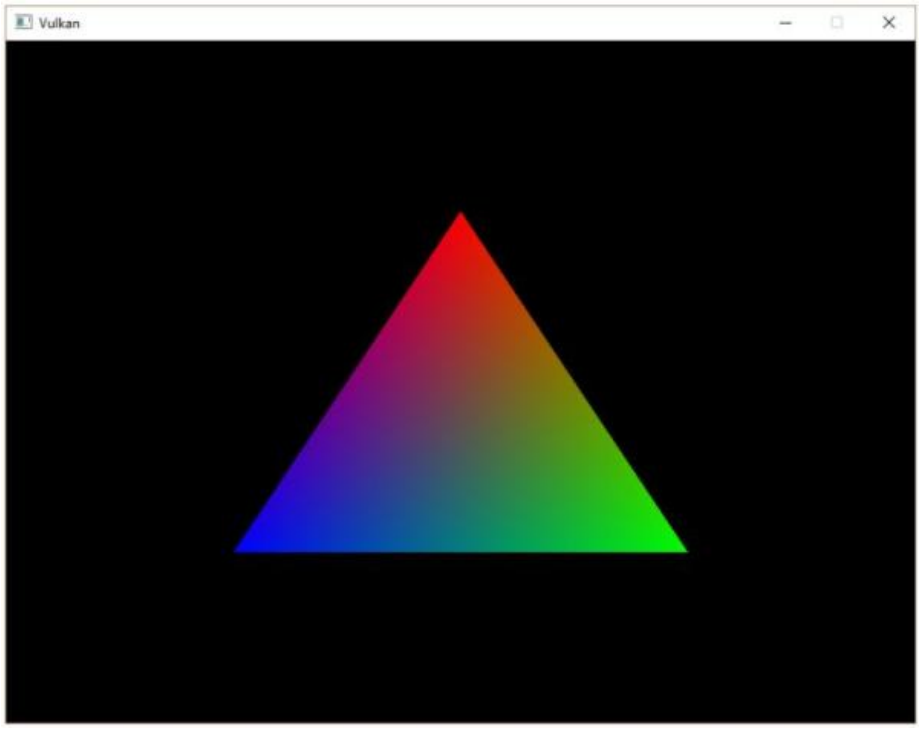

Figure 39: image

尝试修改顶点的颜色数据：

```
const std::vector<Vertex> vertices = {
        {{0.0f, -0.5f}, {1.0f, 1.0f, 1.0f}},
        {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
        {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

再次编译运行程序，可以看到：

下一章节，我们会介绍一种更高效的写入顶点数据到顶点缓冲的方式。

本章节代码：

C++：

https://vulkan-tutorial.com/code/18_vertex_buffer.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.vert

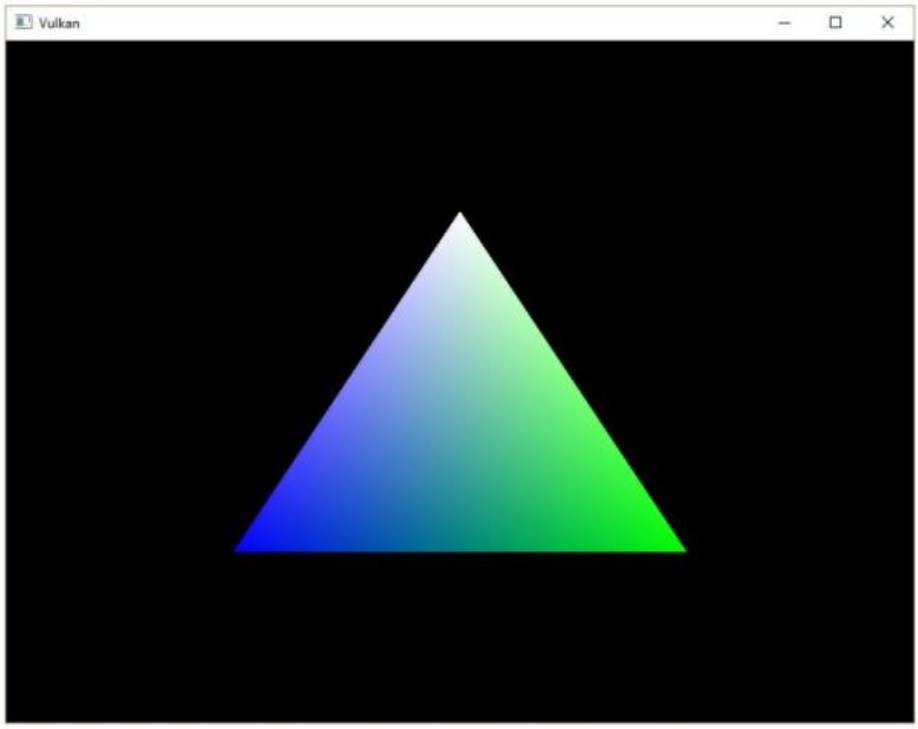

Figure 40: image

Fragment Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.frag

暂存缓冲

介绍

现在我们创建的顶点缓冲已经可以使用了，但我们的顶点缓冲使用的内存类型并不是适合显卡读取的最佳内存类型。最适合显卡读取的内存类型具有 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT 标记，含有这一标记的内存类型通常 CPU 无法直接访问。在本章节，我们会创建两个顶点缓冲。一个用于 CPU 加载数据，一个用于显卡设备读取数据。我们通过缓冲复制指令将 CPU 加载到的缓冲中的数据复制到显卡可以快速读取的缓冲中去。

传输队列

缓冲复制指令需要提交给支持传输操作的队列执行，我们可以查询队列族是否支持 VK_QUEUE_TRANSFER_BIT 特性，确定是否可以使用缓冲复制指令。对于支持 VK_QUEUE_GRAPHICS_BIT 或 VK_QUEUE_COMPUTE_BIT 特性的队列族，VK_QUEUE_TRANSFER_BIT 特性一定被支持，所以我们不需要显式地检测队列族是否支持 VK_QUEUE_TRANSFER_BIT 特性。

如果读者喜欢挑战，可以尝试使用其它支持传输操作的队列族。这需要读者按照下面对程序进行一定地修改：

- 修改 QueueFamilyIndices 和 findQueueFamilies 显式地查找具有 VK_QUEUE_TRANSFER_BIT 特性,但不具有 VK_QUEUE_GRAPHICS_BIT 特性的队列族。
- 修改 createLogicalDevice 函数，申请一个传输队列。
- 创建另外一个指令池对象用于分配用于传输队列的指令缓冲对象。
- 修改 sharingMode 为 VK_SHARING_MODE_CONCURRENT。指定使用图形和传输两个队列族。
- 提交传输指令 (比如 vkCmdCopyBuffer) 给传输队列而不是之前的图形队列。

看起来需要修改的地方还比较多，但复杂的修改也更容易让我们对资源在队列族之间的共享有更深刻的理解。

创建缓冲的辅助函数

我们添加一个叫做 createBuffer 的辅助函数来帮助我们创建缓冲：

```
void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags propert
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
```

```
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }

    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

我们添加了一些参数来方便地使用不同的缓冲大小，内存类型来创建我们需要的缓冲。createBuffer 函数的最后两个参数用于返回创建的缓冲对象和它关联的内存对象。

现在我们可以使用 createBuffer 函数替换之前 createVertexBuffer 函数的实现内容：

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
    createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_HOST_VIS

    void* data;
    vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, vertexBufferMemory);
}
```

编译运行程序，确保顶点缓冲工作正常。

使用暂存缓冲

修改 createVertexBuffer 函数，使用 CPU 可见的缓冲作为临时缓冲，使用显卡读取较快的缓冲作为真正的顶点缓冲：

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
```

```
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIB

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFE
    vertexBufferMemory);
}
```

现在我们可以使用新的关联 stagingBufferMemory 作为内存的 stagingBuffer 缓冲对象来存放 CPU 加载的顶点数据。在本章节，我们会使用下面两个缓冲使用标记：

- VK_BUFFER_USAGE_TRANSFER_SRC_BIT：缓冲可以被用作内存传输操作的数据来源。
- VK_BUFFER_USAGE_TRANSFER_DST_BIT：缓冲可以被用作内存传输操作的目的缓冲。

vertexBuffer 现在关联的内存是设备所有的，不能 vkMapMemory 函数对它关联的内存进行映射。我们只能通过 stagingBuffer 来向 vertexBuffer 复制数据。我们需要使用标记指明我们使用缓冲进行传输操作。

添加一个 copyBuffer 的辅助函数用于在缓冲之间复制数据：

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {

}
```

我们需要一个支持内存传输指令的指令缓冲来记录内存传输指令，然后提交到内存传输指令队列执行内存传输。通常，我们会为内存传输指令使用的指令缓冲创建另外的指令池对象，这是因为内存传输指令的指令缓存通常生命周期很短，为它们使用独立的指令池对象，可以进行更好的优化。我们可以在创建指令池对象时为它指定 VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 标记。

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
}
```

开始记录内存传输指令：

```
VkCommandBufferBeginInfo beginInfo = {};
beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

vkBeginCommandBuffer(commandBuffer, &beginInfo);
```

我们之前对绘制指令缓冲使用的 VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT 标记，在这里不是必须的，这是因为我们只使用这个指令缓冲一次，等待复制操作完成后才继续程序的执行。我们可以使用 VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT 标记告诉驱动程序我们如何使用这个指令缓冲，来让驱动程序进行更好的优化。

```
VkBufferCopy copyRegion = {};
copyRegion.srcOffset = 0; // Optional
copyRegion.dstOffset = 0; // Optional
copyRegion.size = size;
vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);
```

我们使用 vkCmdCopyBuffer 指令来进行缓冲的复制操作。它以源缓冲对象和目的缓冲对象，以及一个 VkBufferCopy 数组为参数。VkBufferCopy 数组指定了复制操作的源缓冲位置偏移，目的缓冲位置偏移，以及要复制的数据长度。和 vkMapMemory 指令不同，这里不能使用 VK_WHOLE_SIZE 来指定要复制的数据长度。

```
vkEndCommandBuffer(commandBuffer);
```

我们的内存传输操作使用的指令缓冲只包含了复制指令，记录完复制指令后，我们就可以结束指令缓冲的记录操作，提交指令缓冲完成传输操作的执行：

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
vkQueueWaitIdle(graphicsQueue);
```

和绘制指令不同，这一次我们直接等待传输操作完成。有两种等待内存传输操作完成的方法：一种是使用栅栏 (fence)，通过 vkWaitForFences 函数等待。另一种是通过 vkQueueWaitIdle 函数等待。使用栅栏 (fence) 可以同步多个不同的内存传输操作，给驱动程序的优化空间也更大。

```
vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
```

最后，传输操作完成后我们需要清除我们使用的指令缓冲对象。

接着，我们可以在 createVertexBuffer 函数中调用 copyBuffer 函数复制顶点数据到显卡读取较快的缓冲中：

```
createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BI

copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
```

最后，不要忘记清除我们使用的缓冲对象和它关联的内存对象：

```
...

    copyBuffer(stagingBuffer, vertexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

编译运行程序，确保一切正常。暂时由于我们使用的顶点数据过于简单，性能提升并不明显。当我们渲染更为复杂的对象时，可以看到更为明显的提升。

总结

实际上，很少有程序为每个缓冲对象都调用 vkAllocateMemory 函数分配关联内存。物体设备允许同时存在的内存分配次数是有限制的，它最大为 maxMemoryAllocationCount。即使在高端硬件上，比如 NVIDIA GTX 1080，maxMemoryAllocationCount 也只有 4096 这么大。所以，通常我们会一次申请一个很大块的内存，然后基于这个内存实现自己的内存分配器为我们创建的对象通过偏移参数分配内存。

我们可以自己实现内存分配器，也可以使用 GPUOpen 提供的 VulkanMemoryAllocator 内存分配器。在本教程，我们的内存分配次数实际上很小，所以我们为每个需要内存的对象调用 vkAllocateMemory 函数分配内存。

本章节代码：

C++：

https://vulkan-tutorial.com/code/19_staging_buffer.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.vert

Fragment Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.frag

索引缓冲

介绍

应用程序使用的三维网格的顶点通常是在多个三角形间共享的。对于绘制矩形这种简单的几何图元也可以共享顶点：

绘制一个矩形可以通过绘制两个三角形来实现，如果不共享顶点，就需要 6 个顶点。共享顶点的话，只需要 4 个顶点就可以。可以想象对于更加复杂的三维网格，通过共享顶点可以节约大量内存资源。

索引缓冲是一个包含了指向顶点缓冲中顶点数据的索引数组的缓冲。使用索引缓冲，我们可以对顶点数据进行复用。上图演示了索引缓冲的原理，顶点缓冲中包含每一个独一无二的顶点的数据，索引缓冲使用顶点缓冲中顶点数据的索引来引用顶点数据。

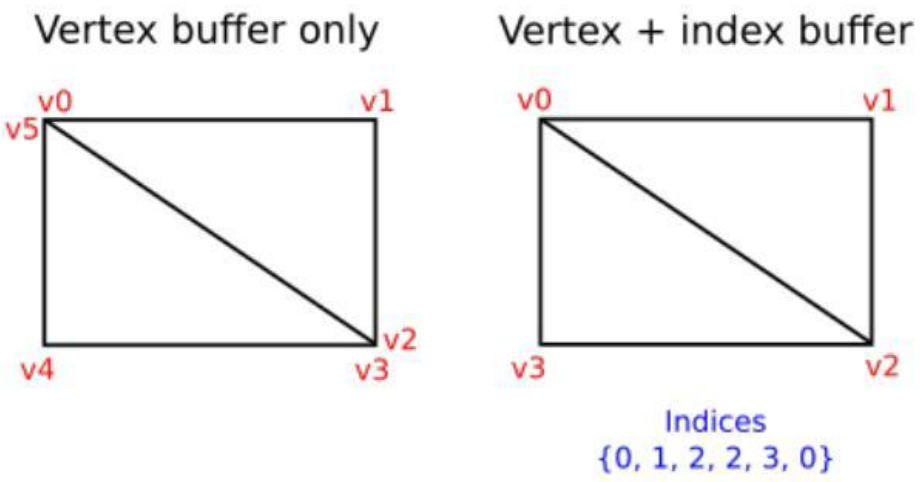

Figure 41: image

创建索引缓冲

在本章节我们通过绘制矩形来演示索引缓冲的使用。修改顶点数据定义矩形的 4 个顶点：

```
const std::vector<Vertex> vertices = {
        {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},
        {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},
        {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},
        {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}
};
```

代码中，我们将矩形左上角的顶点设置为红色，右上角的顶点设置为绿色，右下角的顶点设置蓝色，左下角的顶点设置为白色。我们添加一个新的数组 indices 来存储索引数据：

```
const std::vector<uint16_t> indices = {
            0, 1, 2, 2, 3, 0
};
```

我们可以使用 uint16_t 或 uint32_t 变量类型作为索引的类型，对于不重复的顶点数据小于 65535 的情况，使用 uint16_t 变量类型作为索引类型可以节约一半的内存空间。

和顶点数据一样，我们需要将索引数据加载到一个 VkBuffer 来让 GPU 可以访问它。我们定义了两个类成员变量来存储索引缓冲对象：

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
VkBuffer indexBuffer;
```

```
VkDeviceMemory indexBufferMemory;
```

添加 createIndexBuffer 函数用于索引缓冲创建，它和内容和 createVertexBuffer 函数的内容几乎一样：

```
void initVulkan() {
        ...
    createVertexBuffer();
    createIndexBuffer();
        ...
}

void createIndexBuffer() {
    VkDeviceSize bufferSize = sizeof(indices[0]) * indices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIB

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, indices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER

    copyBuffer(stagingBuffer, indexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

和 createVertexBuffer 函数相比，只有两处明显的不同。bufferSize 现在的值为顶点索引个数乘以索引变量类型所占字节大小。indexBuffer 的用法标记为 VK_BUFFER_USAGE_INDEX_BUFFER_BIT。除此之外的处理和顶点缓冲的创建是相同的。我们也需要创建一个暂存缓冲来存储 indices 数组中的索引数据，然后复制暂存缓冲中的索引数据到 GPU 能够快速访问的缓冲中。

应用程序退出前，我们需要清除创建的索引缓冲对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, indexBuffer, nullptr);
    vkFreeMemory(device, indexBufferMemory, nullptr);

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
```

```
        ...
}
```

使用索引缓冲

使用索引缓冲进行绘制操作，还需要对我们之前编写的 createCommandBuffers 函数进行修改。首先我们需要将索引缓冲对象绑定到指令缓冲对象上，这和绑定顶点缓冲基本类似，不同之处是我们只能绑定一个索引缓冲对象。我们不能为每个顶点属性使用不同的索引，所以即使只有一个顶点属性不同，也要在顶点缓冲中多出一个顶点的数据。

```
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT16);
```

索引缓冲通过调用 vkCmdBindIndexBuffer 函数来进行绑定。vkCmdBindIndexBuffer 函数以索引缓冲对象，索引数据在索引缓冲中的偏移，以及索引数据的类型作为参数。

仅仅绑定索引缓冲是不会起任何作用的，我们需要使用 vkCmdDrawIndexed 指令替换之前使用 vkCmdDraw 指令进行绘制操作：

```
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

vkCmdDrawIndexed 函数的使用和 vkCmdDraw 函数类似。除了指令缓冲对象之外的前两个参数用于指定索引的个数和实例的个数。在这里，我们没有使用实例渲染，所以将实例个数设置为 1。偏移值用于指定显卡开始读取索引的位置，偏移值为 1 对应索引数据中的第二个索引。倒数第二个参数是检索顶点数据前加到顶点索引上的数值。最后一个参数用于第一个被渲染的实例的 ID，在这里，我们没有使用它。

现在编译运行程序可以看到下面的画面：

现在我们已经知道了如何使用索引缓冲来复用顶点数据。在之后的章节，我们载入的三维模型也需要使用索引缓冲来进行渲染。

之前提到，我们应该申请一大块内存来分配给多个缓冲对象使用，实际上，可以更进一步，使用一个缓冲对象通过偏移值来存储多个不同的顶点缓冲和索引缓冲数据。这样做之后，由于数据之间非常紧凑，可以更好地被缓存。对于没有同时进行的操作使用的内存块可以供多个对象复用，这种复用方式也被叫做混叠，许多 Vulkan 函数包含有标记参数可以用于显式地指定混叠处理。

本章节代码：

C++：

https://vulkan-tutorial.com/code/20_index_buffer.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.vert

Fragment Shader：

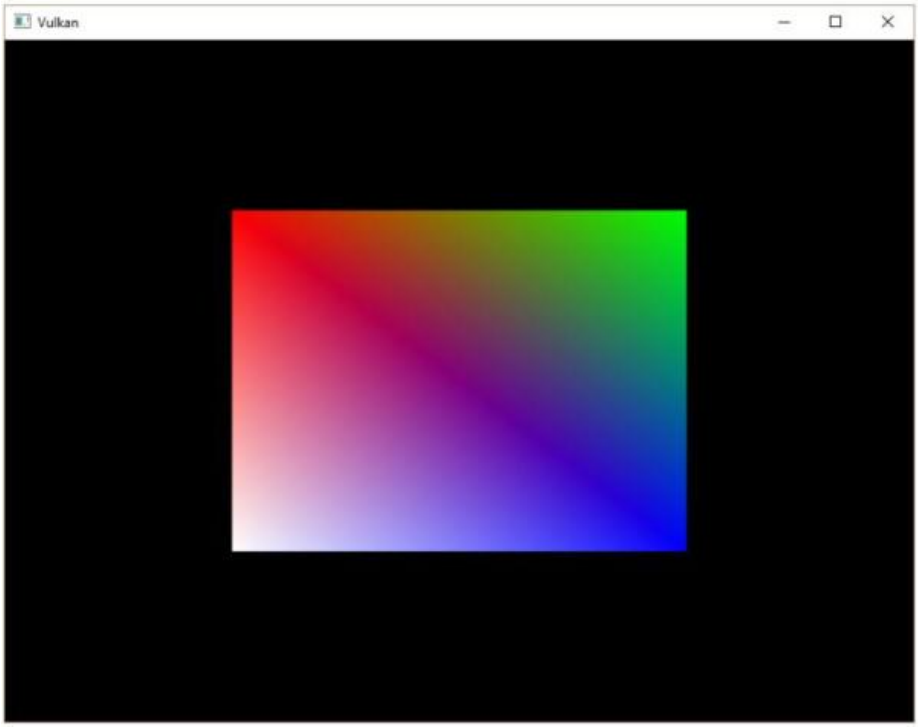

Figure 42: image

https://vulkan-tutorial.com/code/17_shader_vertexbuffer.frag

描述符布局和缓冲

介绍

现在，我们已经可以传递顶点属性给顶点着色器，但对于一些所有顶点都共享的属性，比如顶点的变换矩阵，将它们作为顶点属性为每个顶点都传递一份显然是非常浪费的。

Vulkan 提供了资源描述符来解决这一问题。描述符是用来在着色器中访问缓冲和图像数据的一种方式。我们可以将变换矩阵存储在一个缓冲中，然后通过描述符在着色器中访问它。使用描述符需要进行下面三部分的设置：

- 在管线创建时指定描述符布局
- 从描述符池分配描述符集
- 在渲染时绑定描述符集

描述符布局用于指定可以被管线访问的资源类型，类似渲染流程指定可以被访问的附着类型。描述符集指定了要绑定到描述符上的缓冲和图像资源，类似帧缓冲指定绑定到渲染流程附着上的图像视图。最后，将描述符集绑定到绘制指令上，类似绑定顶点缓冲和帧缓冲到绘制指令上。

有多种类型的描述符，但在本章节，我们只使用 uniform 缓冲对象 (UBO)。在之后的章节，我们会看到其它类型的描述符，它们的使用方式和 uniform 缓冲对象类似。现在，让我们先用结构体定义我们要在着色器中使用的 uniform 数据：

```
struct UniformBufferObject {
            glm::mat4 model;
            glm::mat4 view;
            glm::mat4 proj;
};
```

我们将要使用的 uniform 数据复制到一个 VkBuffer 中，然后通过一个 uniform 缓冲对象描述符在顶点着色器中访问它：

```
layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model *
    vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

接下来，我们在每一帧更新模型，视图，投影矩阵，来让上一章节渲染的矩形在三维空间中旋转。

顶点着色器

修改顶点着色器包含上面的 uniform 缓冲对象。这里，我们假定读者对 MVP 变换矩阵有一定了解。如果没有，读者可以查找资源学习一下。

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

uniform, in 和 out 定义在着色器中出现的顺序是可以任意的。上面代码中的 binding 修饰符类似于我们对顶点属性使用的 location 修饰符。我们会在描述符布局引用这个 binding 值。包含 gl_Position 的那行现在使用变换矩阵计算顶点最终的裁剪坐标。

描述符集布局

下一步，我们在应用程序的代码中定义 UBO：

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

通过 GLM 库我们可以准确地匹配我们在着色器中使用变量类型，可以放心地直接使用 memcpy 函数复制 UniformBufferObject 结构体的数据 VkBuffer 中。

我们需要在管线创建时提供着色器使用的每一个描述符绑定信息。我们添加了一个叫做 createDescriptorSetLayout 的函数，来完成这项工作。并在管线创建前调用它：

```
void initVulkan() {
        ...
    createDescriptorSetLayout();
    createGraphicsPipeline();
        ...
}

        ...

void createDescriptorSetLayout() {

}
```

我们需要使用 VkDescriptorSetLayoutBinding 结构体来描述每一个绑定：

```
void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding = {};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.descriptorCount = 1;
}
```

binding 和 descriptorType 成员变量用于指定着色器使用的描述符绑定和描述符类型。这里我们指定的是一个 uniform 缓冲对象。着色器变量可以用来表示 uniform 缓冲对象数组，descriptorCount 成员变量用来指定数组中元素的个数。我们可以使用数组来指定骨骼动画使用的所有变换矩阵。在这里，我们的 MVP 矩阵只需要一个 uniform 缓冲对象，所以我们将 descriptorCount 的值设置为 1。

```
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

我们还需要指定描述符在哪一个着色器阶段被使用。stageFlags 成员变量可以指定通过 VkShaderStageFlagBits 组合或 VK_SHADER_STAGE_ALL_GRAPHICS 指定描述符被使用的着色器阶段。在这里，我们只在顶点着色器使用描述符。

```
uboLayoutBinding.pImmutableSamplers = nullptr; // Optional
```

pImmutableSamplers 成员变量仅用于和图像采样相关的描述符。这里我们先将其设置为默认值，之后的章节会对它进行介绍。

所有的描述符绑定被组合进一个 VkDescriptorSetLayout 对象。我们在 pipelineLayout 成员变量的定义上面定义 descriptorSetLayout 成员变量：

```
VkDescriptorSetLayout descriptorSetLayout;
VkPipelineLayout pipelineLayout
```

调用 vkCreateDescriptorSetLayout 函数创建 VkDescriptorSetLayout 对象。vkCreateDescriptorSetLayout 函数以 VkDescriptorSetLayoutCreateInfo 结构体作为参数：

```
VkDescriptorSetLayoutCreateInfo layoutInfo = {};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = 1;
layoutInfo.pBindings = &uboLayoutBinding;

if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr,    &descriptorSetLayout) != VK
    throw std::runtime_error("failed to create descriptor set layout!");
}
```

我们需要在管线创建时指定着色器需要使用的描述符集布局。管线布局对象指定了管线使用的描述符集布局。修改 VkPipelineLayoutCreateInfo 结构体信息引用布局对象：

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
```

读者可能会有疑问，明明一个 VkDescriptorSetLayout 对象就包含了所有要使用的描述符绑定，为什么这里还可以指定多个 VkDescriptorSetLayout 对象，我们会在下一章节作出解释。

描述符布局对象可以在应用程序的整个生命周期使用，即使使用了新的管线对象。通常我们在应用程序退出前才清除它：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);

        ...
}
```

uniform 缓冲

在下一章节，我们将会为着色器指定包含 UBO 数据的缓冲对象。我们首先需要创建用于包含数据的缓冲对象，然后在每一帧将新的 UBO 数据复制到 uniform 缓冲。由于需要频繁的数据更新，在这里使用暂存缓冲并不会带来性能提升。

由于我们同时并行渲染多帧的缘故，我们需要多个 uniform 缓冲，来满足多帧并行渲染的需要。我们可以对并行渲染的每一帧或每一个交换链图像使用独立的 uniform 缓冲对象。由于我需要在指令缓冲中引用 uniform 缓冲，对于每个交换链图像使用独立的 uniform 缓冲对象相对来说更加方便。

添加两个新的类成员变量 uniformBuffers 和 uniformBuffersMemory：

```
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;
```

```
std::vector<VkBuffer> uniformBuffers;
std::vector<VkDeviceMemory> uniformBuffersMemory;
```

添加一个叫做 createUniformBuffers 的函数，在 createIndexBuffer 函数调用后调用它分配 uniform 缓冲对象：

```
void initVulkan() {
        ...
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffer();
        ...
}

        ...

void createUniformBuffer() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    uniformBuffers.resize(swapChainImages.size());
    uniformBuffersMemory.resize(swapChainImages.size());

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, VK_MEMORY_PROPERTY_HOST
    }
}
```

我们会在另外的函数中使用新的变换矩阵更新 uniform 缓冲，所以在这里没有出现 vkMapMemory 函数调用。应用程序退出前不要忘记清除申请的 uniform 缓冲对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }

        ...
}
```

更新 uniform 数据

添加一个叫做 updateUniformBuffer 的函数，然后在 drawFrame 函数中我们已经可以确定获取交换链图像是哪一个后调用它：

```
void drawFrame() {
        ...

    uint32_t imageIndex;
    VkResult result = vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>

        ...

    updateUniformBuffer(imageIndex);

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

        ...
}

        ...

void updateUniformBuffer(uint32_t currentImage) {

}
```

调用 updateUniformBuffer 函数可以在每一帧产生一个新的变换矩阵。updateUniformBuffer 函数的实现需要使用了下面这些头文件：

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <chrono>
```

包含 glm/gtc/matrix_transform.hpp 头文件是为了使用 glm::rotate 之类的矩阵变换函数。GLM_FORCE_RADIANS 宏定义用来使 glm::rotate 这些函数使用弧度作为参数的单位。

包含 chrono 头文件是为了使用计时函数。我们将通过计时函数实现每秒旋转 90 度的效果。

```
void updateUniformBuffer(uint32_t currentImage) {
    static auto startTime = std::chrono::high_resolution_clock::now();

    auto currentTime = std::chrono::high_resolution_clock::now();
    float time = std::chrono::duration<float,
    std::chrono::seconds::period>(currentTime - startTime).count();
}
```

我们在 uniform 缓冲对象中定义我们的 MVP 变换矩阵。模型的渲染被我们设计成绕 Z 轴渲染 time 弧度。

```
UniformBufferObject ubo = {};
ubo.model = glm::rotate(glm::mat4(1.0f), time * glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1
```

glm::rotate 函数以矩阵，旋转角度和旋转轴作为参数。glm::mat4(1.0f) 用于构造单位矩阵。这里，我们通过 time * glm::radians(90.0f) 完成每秒旋转 90 度的操作。

```
ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0
```

对于视图变换矩阵，我们使用上面代码中的定义。glm::lookAt 函数以观察者位置，视点坐标和向上向量为参数生成视图变换矩阵。

```
ubo.proj = glm::perspective(glm::radians(45.0f), swapChainExtent.width / (float) swapChainEx
```

对于透视变换矩阵，我们使用上面代码中的定义。glm::perspective 函数以视域的垂直角度，视域的宽高比以及近平面和远平面距离为参数生成透视变换矩阵。特别需要注意在窗口大小改变后应该使用当前交换链范围来重新计算宽高比。

```
ubo.proj[1][1] *= -1;
```

GLM 库最初是为 OpenGL 设计的，它的裁剪坐标的 Y 轴和 Vulkan 是相反的。我们可以通过将投影矩阵的 Y 轴缩放系数符号取反来使投影矩阵和 Vulkan 的要求一致。如果不这样做，渲染出来的图像会被倒置。

定义完所有的变换矩阵，我们就可以将最后的变换矩阵数据复制到当前帧对应的 uniform 缓冲中。复制数据的方法和复制顶点数据到顶点缓冲一样，除了没有使用暂存缓冲：

```
void* data;
vkMapMemory(device, uniformBuffersMemory[currentImage], 0, sizeof(ubo), 0, &data);
memcpy(data, &ubo, sizeof(ubo));
vkUnmapMemory(device, uniformBuffersMemory[currentImage]);
```

对于在着色器中使用的需要频繁修改的数据，这样使用 UBO 并非最佳方式。还有一种更加高效的传递少量数据到着色器的方法，我们会在之后的章节介绍它。

在下一章节，我们将会对绑定 VkBuffer 对象到 uniform 缓冲描述符的描述符集进行介绍。

本章节代码：

C++：

https://vulkan-tutorial.com/code/21_descriptor_layout.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.vert

Fragment Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.frag

描述符池和描述符集

介绍

上一章节我们介绍了用于描述可以绑定的描述符类型的描述符布局。在这一章节，我们为每一个 vkBuffer 创建描述符集，将其和 uniform 缓冲描述符进行绑定。

描述符池

描述符集不能被直接创建，需要通过描述符池来分配。我们添加一个叫做 createDescriptorPool 的函数来进行描述符池的创建：描述符集不能被直接创建，需要通过描述符池来分配。我们添加一个叫做 createDescriptorPool 的函数来进行描述符池的创建：

```
void initVulkan() {
        ...
    createUniformBuffer();
    createDescriptorPool();
        ...
}


        ...

void createDescriptorPool() {

}
```

我们通过 VkDescriptorPoolSize 结构体来对描述符池可以分配的描述符集进行定义：

```
VkDescriptorPoolSize poolSize = {};
poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSize.descriptorCount = static_cast<uint32_t>(swapChainImages.size());
```

我们会在每一帧分配一个描述符。描述符池的大小需要通过 VkDescriptorPoolCreateInfo 结构体定义：

```
VkDescriptorPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = 1;
poolInfo.pPoolSizes = &poolSize;
```

除了可用的最大独立描述符个数外，我们还需要指定可以分配的最大描述符集个数：

```
poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());
```

VkDescriptorPoolCreateInfo 结构体有一个用于优化的标记,它决定了独立的描述符集是否可以被清除掉:VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT。由于我们在描述符集创建后，就不再对其进行操作，所以我们不需要使用这一标记，将其设置为 0 来使用它的默认值。

```
VkDescriptorPool descriptorPool;

...

if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create descriptor pool!");
}
```

我们添加了一个新的类成员变量来存储使用 vkCreateDescriptorPool 函数创建的描述符池对象。应用程序退出前，我们需要清除我们创建的描述符池对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyDescriptorPool(device, descriptorPool, nullptr);

        ...
}
```

描述符集

有了描述符池，现在我们可以开始创建描述符集对象了。添加一个叫做 createDescriptorSets 的函数来完成描述符集对象的创建：

```
void initVulkan() {
        ...
    createDescriptorPool();
    createDescriptorSets();
        ...
}


        ...

void createDescriptorSets() {

}
```

描述符集的创建需要我们填写 VkDescriptorSetAllocateInfo 结构体。我们需要指定分配描述符集对象的描述符池，需要分配的描述符集数量，以及它们使用的描述符布局：

```
std::vector<VkDescriptorSetLayout>
layouts(swapChainImages.size(), descriptorSetLayout);
VkDescriptorSetAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = static_cast<uint32_t>(swapChainImages.size());
allocInfo.pSetLayouts = layouts.data();
```

在这里，我们为每一个交换链图像使用相同的描述符布局创建对应的描述符集。但由于描述符布局对象个数要匹配描述符集对象个数，所以，我们还是需要使用多个相同的描述符布局对象。

添加类成员变量来存储使用 vkAllocateDescriptorSets 函数创建的描述符集对象：

```
VkDescriptorPool descriptorPool;
std::vector<VkDescriptorSet> descriptorSets;

        ...

descriptorSets.resize(swapChainImages.size());
if (vkAllocateDescriptorSets(device, &allocInfo, &descriptorSets[0]) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate descriptor sets!");
}
```

描述符集对象会在描述符池对象清除时自动被清除，所以不需要我们自己显式地清除描述符对象。vkAllocateDescriptorSets 函数分配地描述符集对象，每一个都带有一个 uniform 缓冲描述符。

描述符集对象创建后，还需要进行一定地配置。我们使用循环来遍历描述符集对象，对它进行配置：

```
for (size_t i = 0; i < swapChainImages.size(); i++) {

}
```

我们通过 VkDescriptorBufferInfo 结构体来配置描述符引用的缓冲对象。VkDescriptorBufferInfo 结构体可以指定缓冲对象和可以访问的数据范围：

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkDescriptorBufferInfo bufferInfo = {};
    bufferInfo.buffer = uniformBuffers[i];
    bufferInfo.offset = 0;
    bufferInfo.range = sizeof(UniformBufferObject);
}
```

如果读者需要使用整个缓冲,可以将 range 成员变量的值设置为 VK_WHOLE_SIZE。更新描述符的配置可以使用 vkUpdateDescriptorSets 函数进行，它以 VkWriteDescriptorSet 结构体数组作为参数：

```
VkWriteDescriptorSet descriptorWrite = {};
descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrite.dstSet = descriptorSets[i];
descriptorWrite.dstBinding = 0;
descriptorWrite.dstArrayElement = 0;
```

dstSet 和 dstBinding 成员变量用于指定要更新的描述符集对象以及缓冲绑定。在这里，我们将 uniform 缓冲绑定到索引 0。需要注意描述符可以是数组，所以我们还需要指定数组的第一个元素的索引，在这里，我们没有使用数组作为描述符，将索引指定为 0 即可。

```
descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrite.descriptorCount = 1;
```

我们需要再次指定描述符的类型。可以通过设置 dstArrayElement 和 descriptorCount 成员变量一次更新多个描述符。

```
descriptorWrite.pBufferInfo = &bufferInfo;
descriptorWrite.pImageInfo = nullptr; // Optional
descriptorWrite.pTexelBufferView = nullptr; // Optional
```

pBufferInfo 成员变量用于指定描述符引用的缓冲数据。pImageInfo 成员变量用于指定描述符引用的图像数据。pTexelBufferView 成员变量用于指定描述符引用的缓冲视图。这里，我们的描述符使用来访问缓冲的，所以我们只使用 pBufferInfo 成员变量。

```
vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

vkUpdateDescriptorSets 函数可以接受两个数组作为参数：VkWriteDescriptorSet 结构体数组和 VkCopyDescriptorSet 结构体数组。后者被用来复制描述符对象。

使用描述符集

现在修改 createCommandBuffers 函数为每个交换链图像绑定对应的描述符集。这需要调用 cmdBindDescriptorSets 函数来完成，我们在调用 vkCmdDrawIndexed 函数之前调用这一函数：

```
vkCmdBindDescriptorSets(commandBuffers[i],
VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

和顶点缓冲、索引缓冲不同，描述符集并不是图形管线所独有的，所以需要我们指定我们要绑定的是图形管线还是计算管线。管线类型之后的参数是描述符所使用的布局。接着三个参数用于指定描述符集的第一个元素索引，需要绑定的描述符集个数，以及用于绑定的描述符集数组。最后两个参数用于指定动态描述符的数组偏移。在之后的章节，我们会对它们进行更为详细地讨论。

现在编译运行程序，读者会发现窗口一片漆黑，看不到我们渲染的矩形了。这是因为之前我们设置的投影矩阵将 Y 轴反转，导致顶点按照顺时针绘制构成背面，从而被背面剔除。我们可以修改 createGraphicsPipeline 函数中 VkPipelineRasterizationStateCreateInfo 结构体的 frontFace 成员变量对于正面的设置来解决这一问题：

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

重新编译运行程序，就可以看到下面的画面：

之前的矩形，现在变成了一个正方形，这是我们设置的投影矩阵的宽高比导致的。updateUniformBuffer 函数在每一帧被调用，可以响应窗口大小变化，不需要在 recreateSwapChain 函数中对描述符进行重建。

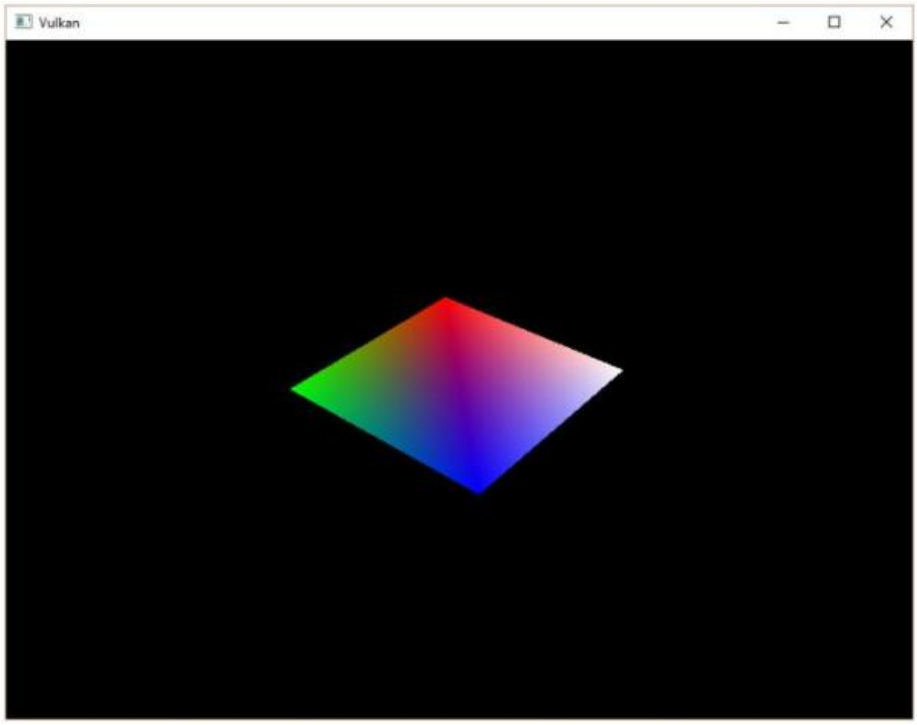

Figure 43: image

多个描述符集

实际上，如我们之前看到的这些函数，可以同时绑定多个描述符集。我们只需要在创建管线布局时为每个描述符集指定一个描述符布局即可。着色器就可以像下面这样引用特定的描述符集：

```
layout(set = 0, binding = 0) uniform UniformBufferObject { ... }
```

我们可以使用这一特性将共享的描述符放入独立的描述符集中。这样就可以避免在多个不同的绘制调用重新绑定描述符从而提高性能表现。

本章节代码：

C++：

https://vulkan-tutorial.com/code/22_descriptor_sets.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.vert

Fragment Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.frag

图像

介绍

之前我们使用顶点颜色来为几何图元进行着色，这样可以产生的效果相当有限。接下来我们将介绍使用纹理图像来为几何图元着色。在之后加载绘制简单的三维模型时，我们也会用纹理图像来对它进行着色。

在我们的程序中使用纹理，需要采取下面的步骤：

- 创建设备内存 (显存) 支持的图像对象
- 加载图像文件的像素数据
- 创建图像采样器
- 使用图像采样器描述符采样纹理数据

之前，我们对图像对象已经有所熟悉，我们的渲染操作是在我们获取的交换链图像上进行的，但我们还没有自己创建过图像对象。现在，我们将开始自己创建一个图像对象，这一过程有点类似顶点缓冲对象的创建。我们首先会创建一个暂存资源，然后使用像素数据填充它，接着将像素数据从暂存资源复制到我们用来渲染的图像对象。实际上也可以直接创建暂存图像来用于渲染，Vulkan 允许我们直接从 VkBuffer 中复制像素数据到图像对象，并且在一些硬件平台这样做效率确实要高很多。我们首先创建缓冲然后填充像素数据，接着创建一个图像对象，将缓冲中的数据复制到图像对象。创建图像对象和创建缓冲的方式区别不大，需要我们查询内存需求，分配并绑定设备内存。

但创建图像对象还是有一些地方需要我们注意。图像的布局会影响它的像素数据在内存中的组织方式。一般而言，对于现在的硬件设备，直接将像素数据按行进行存储并不能得到最佳的性能表现。不同的图像布局对图像操作也有一定的影响。在之前的章节，我们已经了解了一些图像布局：

- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR：适合呈现操作
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL：适合作为颜色附着，在片段着色器中写入颜色数据。
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL：适合作为传输操作的数据来源，比如 vkCmdCopyImageToBuffer
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL：适合作为传输操作的目的位置，比如 vkCmdCopyBufferToImage
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL：适合在着色器中进行采样操作

最常用的变换图像布局的方式是使用管线屏障 (pipeline barrier)。管线屏障 (pipeline barrier) 主要被用来同步资源访问，比如保证图像在被读取之前数据被写入。它也可以被用来变换图像布局。在本章节，我们使用它进行图像布局变换。如果队列的所有模式为 VK_SHARING_MODE_EXCLUSIVE，管线屏障 (pipeline barrier) 还可以被用来传递队列所有权。

图像库

有许多可以使用的加载图像资源的库，甚至读者也可以自己编写代码来加载一些格式比较简单的图像文件，比如 BMP 和 PPM 图像文件。在本教程，我们使用 stb_image 库来加载图像文件。这一图像库只有一个头文件 stb_image.h，读者可以下载它，放在一个方便的位置，然后将存放它的位置加入编译器的包含路径，就可以使用它了。

Visual Studio

添加包含 stb_image.h 文件的目录到 Additional Include Directories 中。

Makefile

添加包含 stb_image.h 文件的目录到包含路径：

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb

        ...

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH)
```

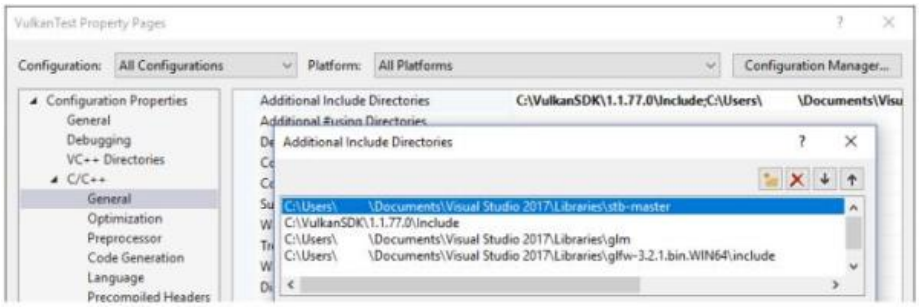

Figure 44: image

载入图像

包含图像库头文件:

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

默认情况下 stb_image.h 文件只定义了函数原型，我们需要在包含 stb_image.h 文件前定义 STB_IMAGE_IMPLEMENTATION 宏，来让它将函数实现包含进来。

```
void initVulkan() {
        ...
    createCommandPool();
    createTextureImage();
    createVertexBuffer();
        ...
}


        ...

void createTextureImage() {

}
```

添加一个叫做 createTextureImage 的函数用于加载图像数据到一个 Vulkan 图像对象。我们需要使用指令缓冲来完成加载，所以 createTextureImage 函数会在 createCommandPool 函数调用之后被调用。

创建一个叫做 textures 的和 shaders 目录同级的目录用于存放图像文件。我们会从 textures 目录载入一个叫做 texture.jpg 的图像文件。我们使用的图像大小为 512x512 像素。这里使用的图像库可以载入常见格式的图像文件，比如 JPEG, PNG, BMP 和 GIF 图像文件。

使用图像库载入文件非常简单:

```
void createTextureImage() {
```

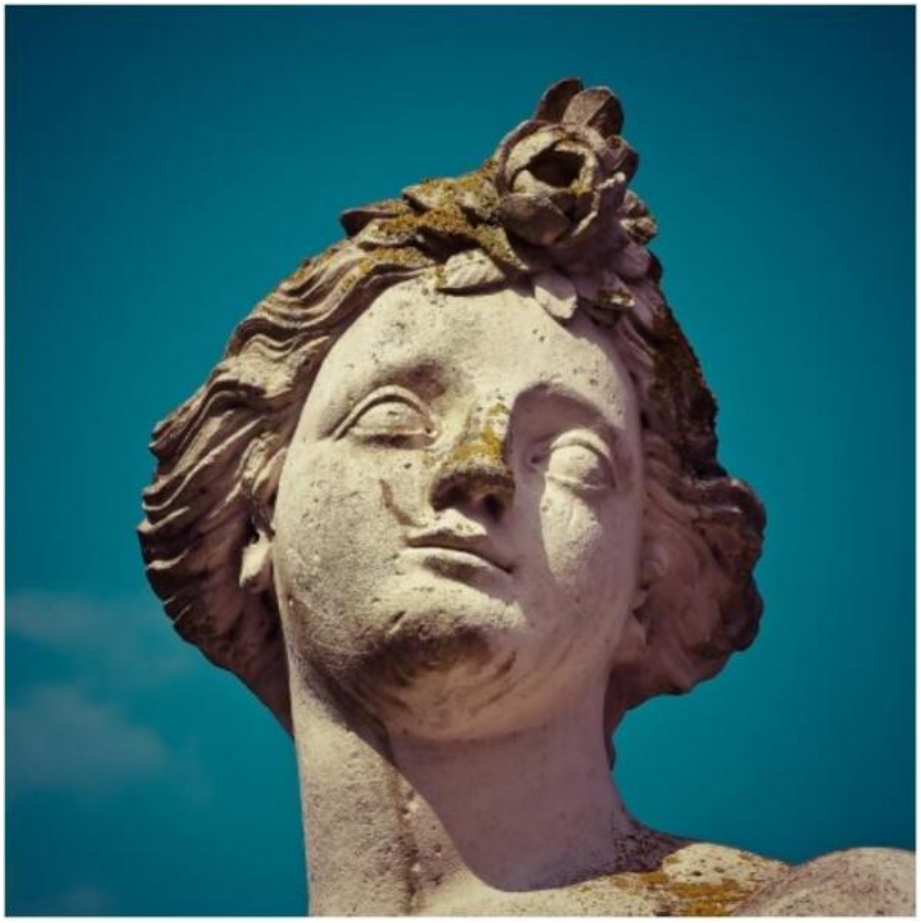

Figure 45: image

```
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }
}
```

上面代码中的 stbi_load 函数以图像文件路径和需要加载的颜色通道作为参数来加载图像文件。使用 STBI_rgb_alpha 通道参数可以强制载入 alpha 通道，即使图像数据不包含这一通道，也会被添加上一个默认的 alpha 值作为 alpha 通道的图像数据，这为我们的处理带来了方便。stbi_load 函数还可以返回图像的宽度，高度和图像数据实际存在的颜色通道。stbi_load 函数的返回值是一个指向解析后的图像像素数据的指针。使用 STBI_rgba_alpha 作为通道参数，每个像素需要 4 个字节存储，所有像素按照行的方式依次存储，总共整个图像需要 texWidth * texHeight * 4 字节来存储。

暂存缓冲

我们创建一个 CPU 可见的缓冲，调用 vkMapMemory 函数映射内存，将图像像素数据复制到其中。在 createTextureImage 函数中添加临时的缓冲变量：

```
VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;
```

我们使用的缓冲内存需要对 CPU 可见，这样，我们才能映射内存，将图像数据复制到其中：

```
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BI
```

接着，我们就可以映射内存，将图像数据复制到缓冲中：

```
void* data;
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
memcpy(data, pixels, static_cast<size_t>(imageSize));
vkUnmapMemory(device, stagingBufferMemory);
```

最后，不要忘记清除我们解析得到的图像像素数据：

```
stbi_image_free(pixels);
```

纹理图像

尽管，我们可以在着色器直接访问缓冲中的像素数据，但使用 Vulkan 的图像对象会更好。Vulkan 的图像对象允许我们使用二维坐标来快速获取颜色数据。图像对象的像素数据也被叫做纹素。现在，让我们添加新的类成员变量：

```
VkImage textureImage;
VkDeviceMemory textureImageMemory;
```

创建图像参数我们需要填写 VkImageCreateInfo 结构体：

```
VkImageCreateInfo imageInfo = {};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType = VK_IMAGE_TYPE_2D;
imageInfo.extent.width = static_cast<uint32_t>(texWidth);
imageInfo.extent.height = static_cast<uint32_t>(texHeight);
imageInfo.extent.depth = 1;
imageInfo.mipLevels = 1;
imageInfo.arrayLayers = 1;
```

imageType 成员变量用于指定图像类型，Vulkan 通过它来确定图像数据的坐标系。图像类型可以是一维，二维和三维图像。一维图像通常被用来存储数组数据或梯度数据。二维图像通常被用来存储纹理。三维图像通常被用类存储体素数据。extent 成员变量用于指定图像在每个维度的范围，也就是在每个坐标轴有多少纹素。我们在这里使用的是二维图像，所以 depth 的值被我们设置为 1，并且我们现在没有使用分级细化，所以将其设置为 1。

```
imageInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
```

Vulkan 支持多种格式的图像数据，这里我们使用的是图像库解析的像素数据格式。

```
imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
```

tiling 成员变量可以是下面这两个值之一：

- VK_IMAGE_TILING_LINEAR：纹素以行主序的方式排列
- VK_IMAGE_TILING_OPTIMAL：纹素以一种对访问优化的方式排列

tiling 成员变量的设置在之后不可以修改。如果读者需要直接访问图像数据，应该将 tiling 成员变量设置为 VK_IMAGE_TILING_LINEAR。由于这里我们使用暂存缓冲而不是暂存图像来存储图像数据，设置为 VK_IMAGE_TILING_LINEAR 是不必要的，我们使用 VK_IMAGE_TILING_OPTIMAL 来获得更好的访问性能。

```
imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
```

initialLayout 成员变量可以设置为下面这些值：

- VK_IMAGE_LAYOUT_UNDEFINED：GPU 不可用，纹素在第一次变换会被丢弃。
- VK_IMAGE_LAYOUT_PREINITIALIZED：GPU 不可用，纹素在第一次变换会被保留。

大多数情况下对于第一次变换，纹素没有保留的必要。但如果读者使用图像对象以及 VK_IMAGE_TILING_LINEAR 标记来暂存纹理数据，这种情况下，纹理数据作为数据传输来源不会被丢弃。但在这里，我们是将图像对象作为传输数据的接收方，将纹理数据从缓冲对象传输到图像对象，所以我们不需要保留图像对象第一次变换时的纹理数据，使用 VK_IMAGE_LAYOUT_UNDEFINED 更好。

```
imageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
```

usage 成员变量的用法和创建缓冲时使用的 usage 成员变量用法相同。这里，我们创建的图像对象被用作传输数据的接收方。并且图像数据需要被着色器采样，所以我们使用了 VK_IMAGE_USAGE_TRANSFER_DST_BIT 和 VK_IMAGE_USAGE_SAMPLED_BIT 这两个使用标记。

```
imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

我们的图像对象只被一个队列族使用：支持传输操作的队列族。所以这里我们使用独占模式。

```
imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageInfo.flags = 0; // Optional
```

samples 成员变量用于设置多重采样。这一设置只对用作附着的图像对象有效，我们的图像不用于附着，将其设置为采样 1 次。有许多用于稀疏图像的优化标记可以使用。稀疏图像是一种离散存储图像数据的方法。比如，我们可以使用稀疏图像来存储体素地形，避免为”空气”部分分配内存。在这里，我们没有使用 flags 标记，将其设置为默认值 0。

```
if (vkCreateImage(device, &imageInfo, nullptr, &textureImage) != VK_SUCCESS) {
    throw std::runtime_error("failed to create image!");
}
```

调用 vkCreateImage 函数创建图像对象，它的参数没有需要特别说明的地方。实际上，图形硬件也可能不支持 VK_FORMAT_R8G8B8A8_UNORM 格式，读者可以使用图形硬件支持的格式来替换它。我们这里跳过检测图形硬件是否支持这一格式是因为这一格式的支持已经十分普遍。使用其它格式还需要一些其它处理。我们会在之后的章节再详细讨论与之相关的问题。

```
VkMemoryRequirements memRequirements;
vkGetImageMemoryRequirements(device, textureImage, &memRequirements);

VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, VK_MEMORY_PROPERT

if (vkAllocateMemory(device, &allocInfo, nullptr, &textureImageMemory) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate image memory!");
}

vkBindImageMemory(device, textureImage, textureImageMemory, 0);
```

分配图像内存的方法和分配缓冲内存几乎一模一样。首先调用 vkGetImageMemoryRequirements 函数获取图像对象的内存需求，然后调用 vkAllocateMemory 函数分配内存，最后调用 vkBindImageMemory 函数将图像对象和内存进行关联即可。

为了简化图像对象的创建操作，我们编写了一个叫做 createImage 的辅助函数：

```
void createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling, Vk
    AVkImageCreateInfo imageInfo = {};
```

```
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = usage;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateImage(device, &imageInfo, nullptr, &image) != VK_SUCCESS) {
        throw std::runtime_error("failed to create image!");
    }

    VkMemoryRequirements memRequirements;
    vkGetImageMemoryRequirements(device, image, &memRequirements);

    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate image memory!");
    }

    vkBindImageMemory(device, image, imageMemory, 0);
}
```

上面代码我们将图像宽度、高度、格式、tiling 模式、使用标记、内存属性作为函数参数，之后的章节，我们将直接使用这一函数来创建图像对象。

现在 createTextureImage 函数可以简化为下面这个样子：

```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }
```

```
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBI

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
    memcpy(data, pixels, static_cast<size_t>(imageSize));
    vkUnmapMemory(device, stagingBufferMemory);

    stbi_image_free(pixels);

    createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_TILING_OPTIMAL, VK_I
}
```

布局变换

接下来我们开始记录传输指令到指令缓冲，我们为此编写了两个辅助函数：

```
VkCommandBuffer beginSingleTimeCommands() {
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);

    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    vkBeginCommandBuffer(commandBuffer, &beginInfo);

    return commandBuffer;
}

void endSingleTimeCommands(VkCommandBuffer commandBuffer) {
    vkEndCommandBuffer(commandBuffer);

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
```

```
    vkQueueWaitIdle(graphicsQueue);

    vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
}
```

上面的代码大部分来自 copyBuffer 函数，现在我们可以用它来简化 copyBuffer 函数的实现：

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    VkBufferCopy copyRegion = {};
    copyRegion.size = size;
    vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

    endSingleTimeCommands(commandBuffer);
}
```

如果我们使用的是缓冲对象而不是图像对象，那么就可以记录传输指令，然后调用 vkCmdCopyBufferToImage 函数结束工作，但这一指令需要图像满足一定的布局要求，所以需要我们编写一个新的函数来进行图像布局变换：

```
void transitionImageLayout(VkImage image, VkFormat format,
VkImageLayout oldLayout, VkImageLayout newLayout) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    endSingleTimeCommands(commandBuffer);
}
```

通过图像内存屏障 (image memory barrier) 我们可以对图像布局进行变换。管线屏障 (pipeline barrier) 主要被用来同步资源访问，比如保证图像在被读取之前数据被写入。它也可以被用来变换图像布局。在本章节，我们使用它进行图像布局变换。如果队列的所有模式为 VK_SHARING_MODE_EXCLUSIVE，管线屏障 (pipeline barrier) 还可以被用来传递队列所有权。对于缓冲对象也有一个可以实现同样效果的缓冲内存屏障 (buffer memory barrier)。

```
VkImageMemoryBarrier barrier = {};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.oldLayout = oldLayout;
barrier.newLayout = newLayout;
```

oldLayout 和 newLayout 成员变量用于指定布局变换。如果不需要访问之前的图像数据，可以将 oldLayout 设置为 VK_IMAGE_LAYOUT_UNDEFINED 来获得更好的性能表现。

```
barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
```

如果读者使用屏障来传递队列族所有权，那么就需要对 srcQueueFamilyIndex 和 dstQueueFamilyIndex 成员变量进行设置。如果读者不进行队列所有权传递，则必

须将这两个成员变量的值设置为 VK_QUEUE_FAMILY_IGNORED。

```
barrier.image = image;
barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
barrier.subresourceRange.baseMipLevel = 0;
barrier.subresourceRange.levelCount = 1;
barrier.subresourceRange.baseArrayLayer = 0;
barrier.subresourceRange.layerCount = 1;
```

image 和 subresourceRange 成员变量用于指定进行布局变换的图像对象，以及受影响的图像范围。这里，我们使用的图像不存在细分级别，所以将 level 和 layer 的值都设置为 1。

```
barrier.srcAccessMask = 0; // TODO
barrier.dstAccessMask = 0; // TODO
```

我们需要指定在屏障之前必须发生的资源操作类型，以及必须等待屏障的资源操作类型。虽然我们 已经使用 vkQueueWaitIdle 函数来手动地进行同步，但还是需要我们进行这一设置。但这一设置依赖旧布局和新布局，所以我们会在确定使用的布局变换后再来设置它。

```
vkCmdPipelineBarrier(
commandBuffer, 0 /* TODO */, 0 /* TODO */, 0, 0, nullptr, 0, nullptr, 1, &barrier);
```

提交管线屏障对象需要调用 vkCmdPipelineBarrier 函数。vkCmdPipelineBarrier 函数除了指令缓冲对象外的第一个参数用于指定发生在屏障之前的管线阶段，第二个参数用于指定发生在屏障之后的管线阶段。如果读者想要在一个屏障之后读取 uniform，应该指定 VK_ACCESS_UNIFORM_READ_BIT 使用标记和最早读取 uniform 的着色器阶段,比如 VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT。当指定和使用标记不匹配的管线阶段时校验层会发出警告信息。

第三个参数可以设置为 0 或 VK_DEPENDENCY_BY_REGION_BIT。设置为 VK_DEPENDENCY_BY_REGION_BIT 的话，屏障就变成了一个区域条件。这允许我们读取资源目前已经写入的那部分。

最后 6 个参数用于引用三种可用的管线屏障数组：内存屏障 (memory barriers)，缓冲内存屏障 (buffer memory barriers) 和图像内存屏障 (image memory barriers)。我们这里使用的是图像内存屏障 (image memory barriers)。需要注意这里我们没有使用 VkFormat 参数，但我们会在之后章节使用它进行特殊的变换操作。

复制缓冲到图像

在我们回到 createTextureImage 函数之前，先编写一个新的辅助函数：copyBufferToImage。

```
void copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t
width, uint32_t height) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();
```

```
    endSingleTimeCommands(commandBuffer);
}
```

和复制缓冲数据一样，我们需要使用 VkBufferImageCopy 结构体指定将数据复制到图像的哪一部分。

```
VkBufferImageCopy region = {};
region.bufferOffset = 0;
region.bufferRowLength = 0;
region.bufferImageHeight = 0;

region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
region.imageSubresource.mipLevel = 0;
region.imageSubresource.baseArrayLayer = 0;
region.imageSubresource.layerCount = 1;

region.imageOffset = {0, 0, 0};
region.imageExtent = { width, height, 1};
```

bufferOffset 成员变量用于指定要复制的数据在缓冲中的偏移位置。bufferRowLength 和 bufferImageHeight 成员变量用于指定数据在内存中的存放方式。通过这两个成员变量我们可以对每行图像数据使用额外的空间进行对齐。将这两个成员变量的值都设置为 0，数据将会在内存中被紧凑存放。imageSubresource、imageOffset 和 imageExtent 成员变量用于指定数据被复制到图像的哪一部分。

从缓冲复制数据到图像需要调用 vkCmdCopyBufferToImage 函数来记录指令到指令缓冲：

```
vkCmdCopyBufferToImage( commandBuffer, buffer, image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
```

vkCmdCopyBufferToImage 函数的第 4 个参数用于指定目的图像当前使用的图像布局。这里我们假设图像已经被变换为最适合作为复制目的的布局。我们只复制了一张图像，实际上是可以指定一个 VkBufferImageCopy 数组来一次从一个缓冲复制数据到多个不同的图像对象。

准备纹理图像

现在我们可以回到 createTextureImage 函数。复制暂存缓冲中的数据到纹理图像，我们需要进行下面两步操作：

- 变换纹理图像到 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
- 执行图像数据复制操作

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_UNDEFINED, VK_
copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth), static_cast<
```

这里我们创建的图像对象使用 VK_IMAGE_LAYOUT_UNDEFINED 布局，所以转换图像布局时应该将 VK_IMAGE_LAYOUT_UNDEFINED 指定为旧布局。需要注意的是我们之所以这样设置是因为我们不需要读取复制操作之前的图像内容。

为了能够在着色器中采样纹理图像数据，我们还需要进行一次图像布局变换：

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_TRANSFER_DST_O
```

变换屏障掩码

如果读者在开启校验层的情况下运行程序，会发现校验层报告 transitionImageLayout 中使用的访问掩码和管线阶段是无效的。我们需要根据布局变换设置 transitionImageLayout。

我们需要处理两种变换：

- 未定义-> 传输目的：传输操作的数据写入不需要等待
- 传输目的-> 着色器读取：着色器读取图像数据需要等待传输操作的写入结束。

我们使用下面的代码指定变换规则：

```
VkPipelineStageFlags sourceStage;
VkPipelineStageFlags destinationStage;

if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTI
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT
        barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
        barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

        sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
        destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else {
    throw std::invalid_argument("unsupported layout transition!");
}

vkCmdPipelineBarrier(commandBuffer, sourceStage, destinationStage, 0, 0, nullptr, 0, nullptr
```

传输的写入操作必须在管线传输阶段进行。这里因为我们的写入操作不需要等待任何对象,我们可以指定一个空的的访问掩码,使用 VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT 指定最早出现的管线阶段。需要注意 VK_PIPELINE_STAGE_TRANSFER_BIT 并非图形和计算管线中真实存在的管线阶段，它实际上是一个伪阶段，出现在传输操作发生时。更多信息可以参考官方文档中有关伪阶段的说明。

图像数据需要在片段着色器读取，所以我们指定了片段着色器管线阶段的读取访问掩码。

之后如果还需要进行更多的图像布局变换，我们会扩展 transitionImageLayout 函数。

指令缓冲的提交会隐式地进行 VK_ACCESS_HOST_WRITE_BIT 同步。因为 transitionImageLayout 函数执行的指令缓冲只包含了一条指令，如果布局转换依赖 VK_ACCESS_HOST_WRITE_BIT，我们可以通过设置 srcAccessMask 的值为 0 来使用这一隐含的同步。不过，我们最好显式地定义一切，依赖于隐含属性，不就变得和 OpenGL 一样容易出现错误。

有一个特殊的支持所有操作的图像布局类型:VK_IMAGE_LAYOUT_GENERAL。但它并不保证能为所有操作都带来最佳性能表现。对于一些特殊情况，比如将图像同时用作输入和输出对象，或读取一个已经改变初始化时的布局的图像时，需要使用它。

目前为止，我们编写的包含提交指令操作的辅助函数都被设置为通过等待队列空闲来进行同步。对于实用的程序，更推荐组合多个操作到一个指令缓冲对象，通过异步执行来增大吞吐量，特别对于 createTextureImage 函数中的变换和数据复制操作，这样做可以获得很大的性能提升。我们可以编写一个叫做 setupCommandBuffer 的辅助函数来记录指令，编写一个叫做 flushSetupCommands 来提交执行记录的指令，通过实验得到最佳的同步策略。

清理

最后，不要忘记在 createTextureImage 函数的结尾清除我们使用的暂存缓冲和它关联的内存:

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_TRANSFER_DST_O

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

纹理图像一直被使用到程序结束才被我们清除:

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);

        ...
}
```

图像数据被加载到图像对象后，还需要一定的设置才能被访问。下一章节，我们会介绍访问我们加载的图像数据的方法。

本章节代码:

C++:

https://vulkan-tutorial.com/code/23_texture_image.cpp

Vertex Shader:

https://vulkan-tutorial.com/code/21_shader_ubo.vert

Fragment Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.frag

图像视图和采样器

在本章节我们会创建两个资源来进行图像数据的采样。第一个资源我们在介绍交换链图像时已经有所了解，第二个资源和着色器采样图像数据有关，我们尚未介绍。

纹理图像视图

在之前的交换链图像和帧缓冲章节，我们已经介绍过访问图像需要通过图像视图进行。对于纹理图像同样需要通过图像视图来进行访问。

添加一个类成员变量来存储纹理图像的图像视图对象，编写 createTextureImageView 函数来创建纹理图像的图像视图对象：

```
VkImageView textureImageView;

        ...

void initVulkan() {
        ...
createTextureImage();
createTextureImageView();
createVertexBuffer();
        ...
}

        ...

void createTextureImageView() {

}
```

createTextureImageView 函数的实现可以由 createImageViews 函数修改得到。两者只有两处设置不同：format 和 image 成员变量的设置。

```
VkImageViewCreateInfo viewInfo = {};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = textureImage;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
```

```
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;
```

在这里，由于 VK_COMPONENT_SWIZZLE_IDENTITY 的值实际上是 0，所以我们可以省略对 viewInfo.components 的显式初始化，将 viewInfo.components 的成员变量都设置为 0。最后，调用 vkCreateImageView 函数创建图像视图对象：

```
if (vkCreateImageView(device, &viewInfo, nullptr, &textureImageView) != VK_SUCCESS) {
    throw std::runtime_error("failed to create texture image view!");
}
```

我们可以编写一个 createImageView 函数来简化图像视图对象的创建：

```
VkImageView createImageView(VkImage image, VkFormat format) {
    VkImageViewCreateInfo viewInfo = {};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;
    if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
        throw std::runtime_error("failed to create texture image view!");
    }

    return imageView;
}
```

现在我们可以使用 createImageView 函数来简化 createTextureImageView 的函数实现：

```
void createTextureImageView() {
    textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_UNORM);
}
```

createImageViews 的函数实现可以被简化为：

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

    for (uint32_t i = 0; i < swapChainImages.size(); i++) {
        swapChainImageViews[i] = createImageView(swapChainImages[i],
        swapChainImageFormat);
    }
}
```

应用程序结束前，我们需要在清除图像对象之前清除与之关联的图像视图对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyImageView(device, textureImageView, nullptr);

    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);
```

采样器

在着色器中是可以直接访问图像数据，但当图像被作为纹理时，我们通常不这样做。通常我们使用采样器来访问纹理数据，采样器可以自动地对纹理数据进行过滤和变换处理。

采样器进行的过滤操作可以很好地处理纹理采样过密的问题。考虑一个被映射到一个几何图元上的纹理，每个纹素占据了多个片段。如果我们直接采样与片段最近的纹素作为片段颜色，可能就会得到下面第一幅图的效果：

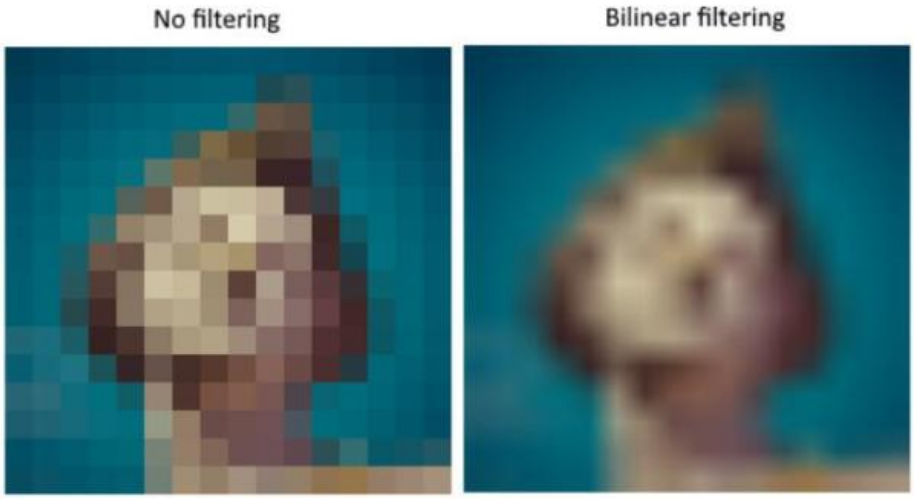

Figure 46: image

如果使用线性插值组合 4 个最近的纹素，我们可以得到上面第二幅图的效果。当然，有时候我们可能会喜欢第一幅图的效果 (比如在编写类似我的世界这种游戏时)，当通常来说我们更多的想要第二幅图的效果。采样器可以自动地为我们进行上面这样的纹理过滤。

与之相反的是纹理采样过疏的问题，这种情况发生在多个纹素被映射到一个片段时。这会造成幻影现象，如下面第一幅图：

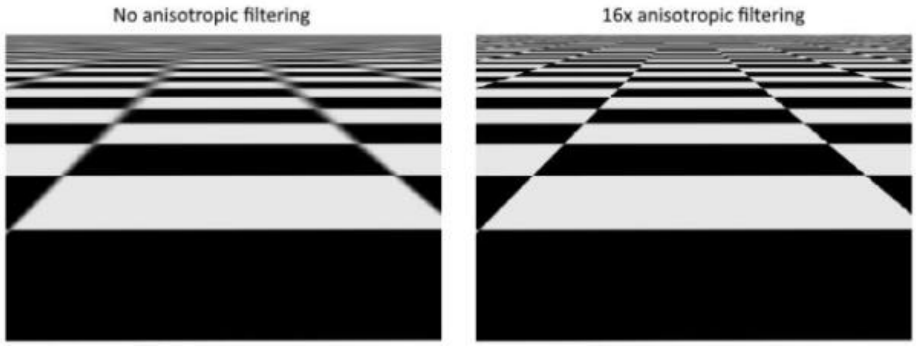

Figure 47: image

上面第二幅图，我们可以看到远处的纹理已经变得模糊不清。解决这一问题的方法是使用采样器进行各向异性过滤。

除了上面这些过滤器，采样器还可以进行变换操作。变换操作发生在采样超出纹理图像实际范围的数据时，下面这些图像就是使用采样器的变换操作产生的：

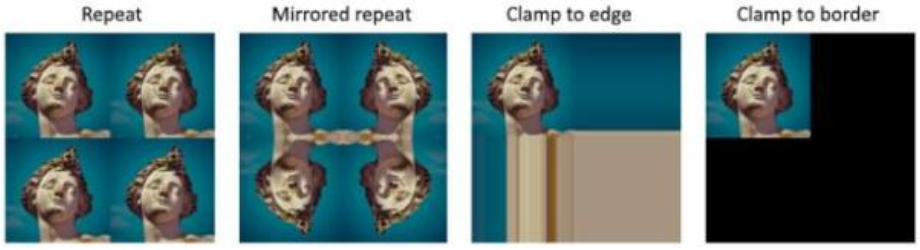

Figure 48: image

我们编写一个叫做 createTextureSampler 的函数来创建采样器对象。之后，我们会在着色器中使用创建的采样器对象采样纹理数据。

```
void initVulkan() {
        ...
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
        ...
}


        ...

void createTextureSampler() {
```

```
}
```

采样器对象的配置需要填写 VkSamplerCreateInfo 结构体，通过它可以指定采样器使用的过滤器和变换操作。

```
VkSamplerCreateInfo samplerInfo = {};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
```

magFilter 和 minFilter 成员变量用于指定纹理需要放大和缩小时使用的插值方法。纹理放大会出现采样过密的问题，纹理缩小会出现采样过疏的问题。对于这两个成员变量，设置其值为 VK_FILTER_NEAREST 或 VK_FILTER_LINEAR，分别对应上面我们讨论的两种过滤方式。

```
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

addressModeU、addressModeV 和 addressModeW 用于指定寻址模式。这里的 U、V、W 对应 X、Y 和 Z 轴。它们的值可以是下面这些：

- VK_SAMPLER_ADDRESS_MODE_REPEAT：采样超出图像范围时重复纹理
- VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT：采样超出图像范围时重复镜像后的纹理
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE：采样超出图像范围时使用距离最近的边界纹素
- VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE：采样超出图像范围时使用镜像后距离最近的边界纹素
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER：采样超出图像返回时返回设置的边界颜色

在这里，我们的采样不会超出图像范围，所以暂时不用关心使用的寻址模式。通常来说，VK_SAMPLER_ADDRESS_MODE_REPEAT 模式是最常用的，可以用它来实现平铺纹理的效果。

```
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
```

anisotropyEnable 和 maxAnisotropy 成员变量和各向异性过滤相关。通常来说，只要性能允许，我们都会开启各向异性过滤。maxAnisotropy 成员变量用于限定计算最终颜色使用的样本个数。maxAnisotropy 成员变量的值越小，采样的性能表现越好，但采样结果质量较低。目前为止，还没有图形硬件能够使用超过 16 个样本，同时即使可以使用超过 16 个样本，采样效果的增强也开始变得微乎其微。

```
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
```

borderColor 成员变量用于指定使用 VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER 寻址模式时采样超出图像范围时返回的边界颜色。边界颜色并非可以设置为任意颜色。它可以被设置为浮点或整型格式的黑色、白色或透明色。

```
samplerInfo.unnormalizedCoordinates = VK_FALSE;
```

unnormalizedCoordinates 成员变量用于指定采样使用的坐标系统。将其设置为 VK_TRUE 时，采样使用的坐标范围为 [0, texWidth) 和 [0, texHeight)。将其设置为 VK_FALSE，采样使用的坐标范围在所有轴都是 [0, 1)。通常使用 VK_FALSE 的情况更常见，这种情况下我们可以使用相同的纹理坐标采样不同分辨率的纹理。

```
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
```

通过 compareEnable 和 compareOp 成员变量，我们可以将样本和一个设定的值进行比较，然后将比较结果用于之后的过滤操作。通常我们在进行阴影贴图时会使用它。暂时我们不使用这一功能，在之后的章节，我们会对这一功能进行更为详细地介绍。

```
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = 0.0f;
```

mipmapMode、mipLodBias、minLod 和 maxLod 成员变量用于设置分级细化 (mipmap)，我们会在之后的章节对分级细化进行介绍，它可以看作是过滤操作的一种。

至此，我们就完成了 VkSamplerCreateInfo 结构体的填写。现在添加一个类成员变量来存储我们调用 vkCreateSampler 函数创建的采样器对象：

```
VkImageView textureImageView;
VkSampler textureSampler;

        ...

void createTextureSampler() {
        ...

    if (vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler) != VK_SUCCESS) {
        throw std::runtime_error("failed to create texture sampler!");
    }
}
```

需要注意，采样器对象并不引用特定的 VkImage 对象，它是一个用于访问纹理数据的接口。我们可以使用它来访问任意不同的图像，不管图像是一维的、二维的、还是三维的。这和一些旧的图形 API 将纹理图像和过滤设置绑定在一起进行采样是不同的。

最后，我们需要在应用程序结束前清除我们创建的采样器对象：

```
void cleanup() {
    cleanupSwapChain();

    vkDestroySampler(device, textureSampler, nullptr);
    vkDestroyImageView(device, textureImageView, nullptr);

        ...
}
```

各向异性设备特性

如果现在读者编译运行程序，就会看到下面的校验层信息：

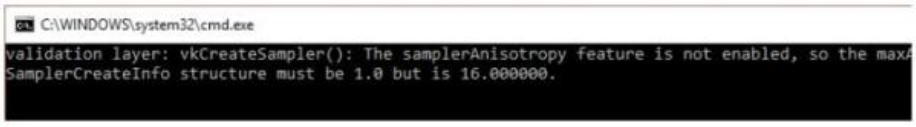

Figure 49: image

这是因为各向异性过滤实际上是一个非必需的设备特性。我们需要在创建逻辑设备时检查这一设备特性是否被设备支持才能使用它：

```
VkPhysicalDeviceFeatures deviceFeatures = {};
deviceFeatures.samplerAnisotropy = VK_TRUE;
```

尽管，现在大部分现代图形硬件都已经支持了这一特性，但我们最好还是以防万一对设备是否支持这一特性进行检测：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
        ...

    VkPhysicalDeviceFeatures supportedFeatures;
    vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

    return indices.isComplete() && extensionsSupported && swapChainAdequate && supportedFeat
}
```

我们通过调用 vkGetPhysicalDeviceFeatures 函数获取物理设备支持的特性信息，然后验证是否包含各向异性过滤特性。

如果不想使用各向异性过滤，可以按照下面这样设置：

```
samplerInfo.anisotropyEnable = VK_FALSE;
samplerInfo.maxAnisotropy = 1;
```

在下一章节，我们会在着色器中使用采样器访问纹理数据将纹理图像绘制到一个矩形上。

本章节代码：

C++：

https://vulkan-tutorial.com/code/24_sampler.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.vert

Fragment Shader：

https://vulkan-tutorial.com/code/21_shader_ubo.frag

组合图像采样器

介绍

在之前的章节，我们已经对描述符有所了解，这一章节，我们将介绍一种新的描述符类型：组合图像采样器。着色器可以通过这一类型的描述符访问图像资源。

我们首先会修改描述符布局，描述符池和描述符集来使用组合图像采样器描述符。然后，我们会添加纹理坐标信息到 Vertex 结构体。最后，我们修改片段着色器从纹理中读取颜色数据。

更新描述符

在 createDescriptorSetLayout 函数中填写用于组合图像采样器描述符的 VkDescriptorSetLayoutBinding 结构体信息。将下面的代码放在 uniform 缓冲绑定之后：

```
VkDescriptorSetLayoutBinding samplerLayoutBinding = {};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.pImmutableSamplers = nullptr;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

std::array<VkDescriptorSetLayoutBinding, 2> bindings = {uboLayoutBinding, samplerLayoutBindi
VkDescriptorSetLayoutCreateInfo layoutInfo = {};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
layoutInfo.pBindings = bindings.data();
```

设置 stageFlags 成员变量指明在片段着色器中使用组合图像采样器描述符。在顶点着色器也可以进行纹理采样，一个常见的用途是在顶点着色器中使用高度图纹理来对顶点进行变形。

现在在开启校验层的情况下编译运行程序，我们会得到描述符池不能分配该类型的描述符集这一信息。这一因为我们原来创建的描述符池对象并没有包含组合图像采样器描述符，为了解决这一问题，我们修改 createDescriptorPool 函数，添加一个用于组合图像采样器描述符的 VkDescriptorPoolSize 结构体信息：

```
std::array<VkDescriptorPoolSize, 2> poolSizes = {};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = static_cast<uint32_t>(swapChainImages.size());

VkDescriptorPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());
```

最后，我们在 createDescriptorSets 函数中绑定图像和采样器到描述符集中的描述符：

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkDescriptorBufferInfo bufferInfo = {};
    bufferInfo.buffer = uniformBuffers[i];
    bufferInfo.offset = 0;
    bufferInfo.range = sizeof(UniformBufferObject);

    VkDescriptorImageInfo imageInfo = {};
    imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    imageInfo.imageView = textureImageView;
    imageInfo.sampler = textureSampler;

        ...
}
```

需要使用 VkDescriptorImageInfo 结构体为组合图像采样器指定图像资源。

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites = {};

descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount = 1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;
```

```
vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()), descriptorWri
```

更新描述符需要使用图像资源信息。至此，我们就可以在着色器中使用描述符了。

纹理坐标

我们需要使用纹理坐标来将纹理映射到几何图元上。

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription = {};
        bindingDescription.binding = 0;
        bindingDescription.stride = sizeof(Vertex);
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3>
        attributeDescriptions = {};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        attributeDescriptions[1].binding = 0;
        attributeDescriptions[1].location = 1;
        attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[1].offset = offsetof(Vertex, color);

        attributeDescriptions[2].binding = 0;
        attributeDescriptions[2].location = 2;
        attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[2].offset = offsetof(Vertex, texCoord);

        return attributeDescriptions;
    }
};
```

修改 Vertex 结构体添加一个新的 vec2 类型变量来存储纹理坐标。添加一个新的

VkVertexInputAttributeDescription 对象用于在顶点着色器访问顶点的纹理坐标。这样我们才能将纹理坐标传递给片段着色器使用。

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
};
```

在本教程，我们为渲染的矩形的四个顶点指定纹理图像的四个顶点作为纹理坐标。

着色器

现在可以修改着色器从纹理中采样颜色数据。我们首先需要修改顶点着色器将纹理坐标传递给片段着色器：

```
layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 2) in vec2 inTexCoord;

layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

fragTexCoord 的值会被插值后传递给片段着色器。我们可以通过将纹理坐标输出为片段颜色来形象地认知这一插值过程：

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragTexCoord, 0.0, 1.0);
}
```

重新编译我们的着色器代码，然后编译运行应用程序，将会看到下面的画面：

我们这里使用绿色通道来表示 x 坐标，使用红色通道来表示 y 坐标。通过上图的黑色和黄色角落，我们可以确定矩形左上角顶点的纹理坐标为 $(0, 0)$，右下角顶点的纹

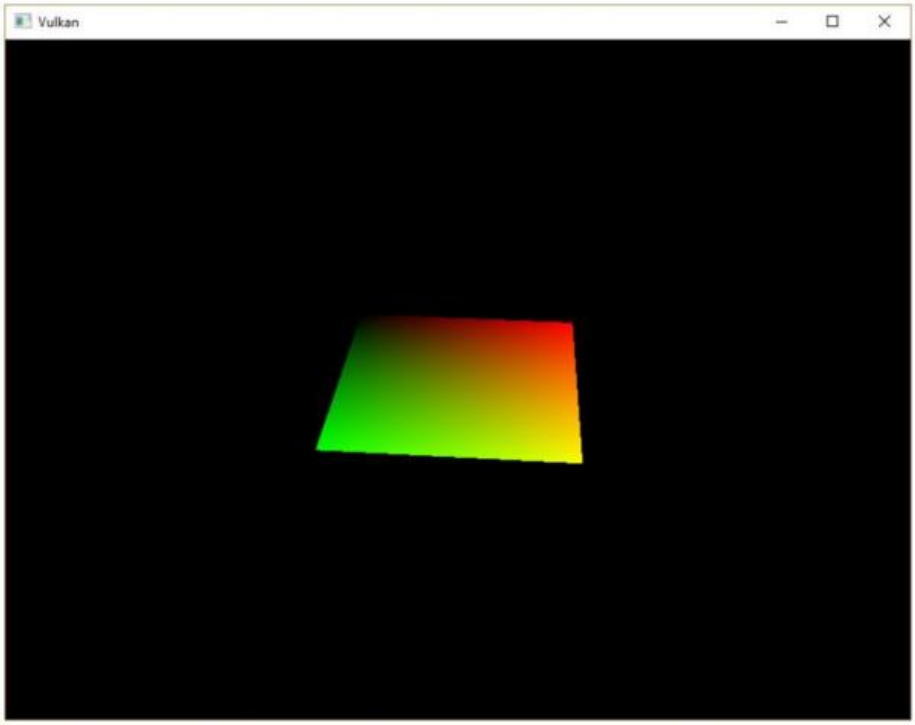

Figure 50: image

理坐标为 (1, 1)。这种通过将变量值作为片段颜色输出的方法是一个很有用的调试着色器的手段。

在 GLSL 中，使用 uniform 变量来表示组合图像采样器描述符。为了在片段着色器中使用描述符，我们需要添加下面的代码：

```
layout(binding = 1) uniform sampler2D texSampler;
```

对于一维图像和二维图像需要使用对应的 sampler1D 和 sampler3D 变量类型来绑定描述符。

```
void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

在 GLSL 纹理采样需要使用 GLSL 内建的 texture 函数。texture 函数使用一个采样器变量和一个纹理坐标作为参数对纹理进行采样。采样器会自动按照我们设定的过滤和变换方式对纹理进行采样。现在编译运行程序，可以看到下面的画面：

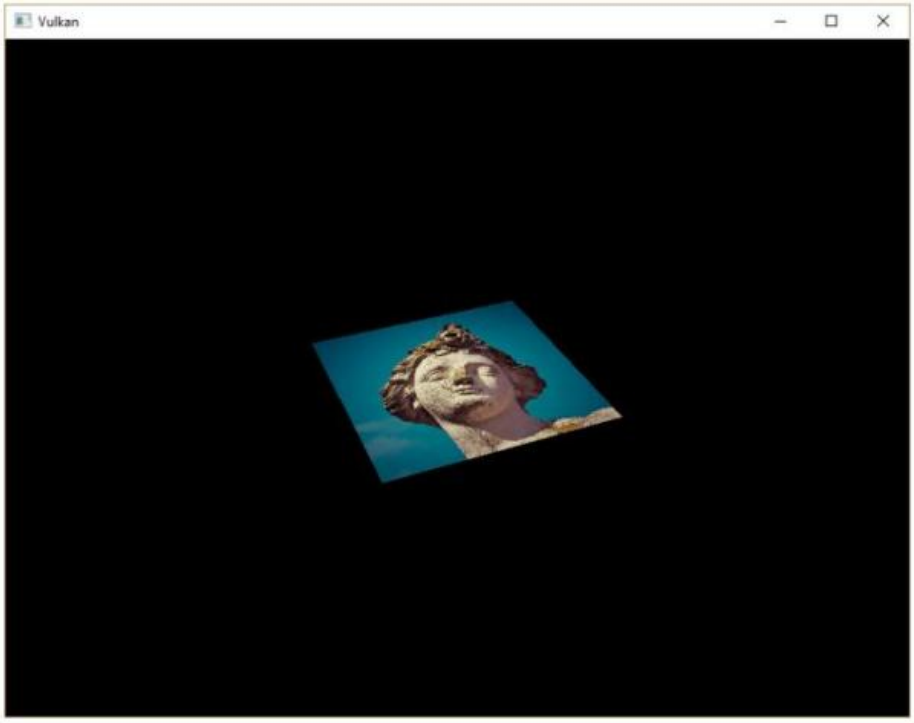

Figure 51: image

```
void main() {
    outColor = texture(texSampler, fragTexCoord * 2.0);
}
```

读者可以尝试通过将纹理坐标设置为大于 1.0 的值来观察寻址模式对纹理采样的影响。下图是使用 VK_SAMPLER_ADDRESS_MODE_REPEAT 寻址模式产生的效果：

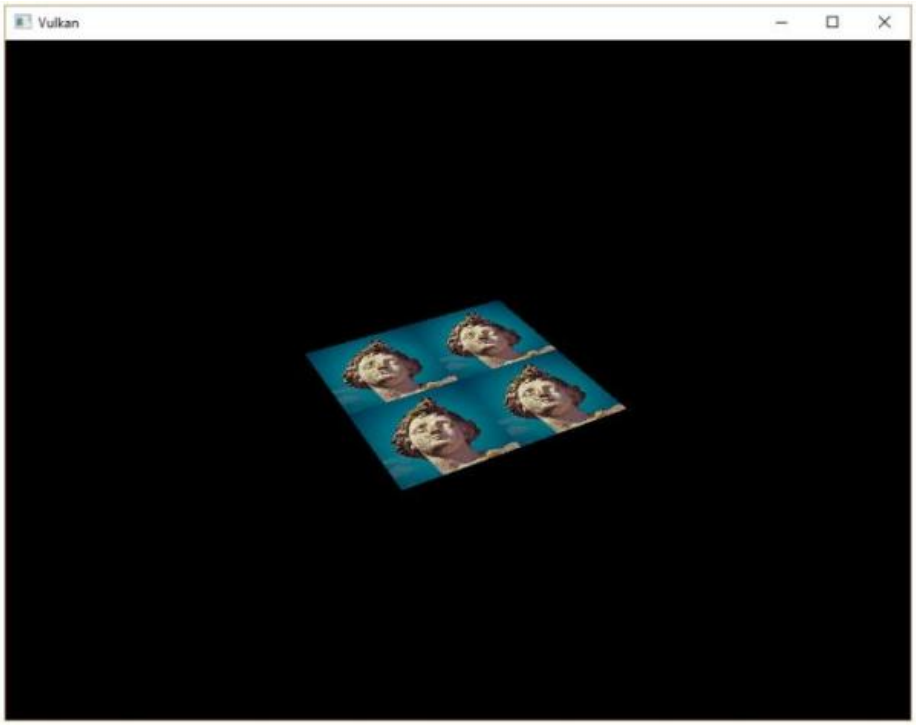

Figure 52: image

我们还可以同时使用顶点颜色和纹理颜色来产生最终的片段颜色：

```
void main() {
    outColor = vec4(fragColor * texture(texSampler, fragTexCoord).rgb, 1.0);
}
```

下图是这样做后产生的效果：

现在，我们已经明白了如何在着色器中访问图像数据。通过这项技术，我们可以实现许多复杂的效果，比如后期处理，延迟着色等等。

本章节代码：

C++：

https://vulkan-tutorial.com/code/25_texture_mapping.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/25_shader_textures.vert

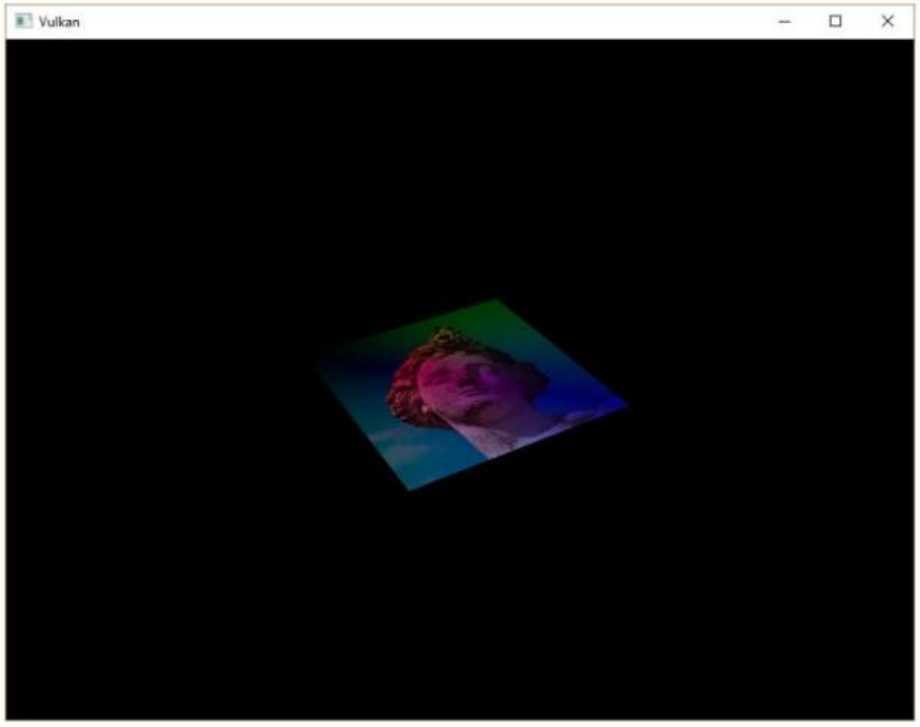

Figure 53: image

Fragment Shader：

https://vulkan-tutorial.com/code/25_shader_textures.frag

深度缓冲

介绍

在本章节，我们会开始使用三维的顶点坐标。我们会放置两个 z 坐标不同的矩形来表现深度缓冲解决的问题。

三维几何

修改 Vertex 结构体，使用三维向量来表示顶点的位置信息，更新对应的 VkVertexInputAttributeDescription 结构体的 format 成员变量设置：

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

        ...

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3>
        attributeDescriptions = {};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        ...
    }
};
```

接着，修改顶点着色器接收三维顶点位置输入。最后不要忘记重新编译修改后的顶点着色器代码：

```
layout(location = 0) in vec3 inPosition;

        ...

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
    fragColor = inColor;
```

```
    fragTexCoord = inTexCoord;
}
```

更新 vertices 中的顶点数据，添加 z 坐标信息：

```
const std::vector<Vertex> vertices = {
{{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
{{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
{{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
{{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};
```

现在如果编译运行程序，我们会看到和之前一样的渲染结果。为了显示深度缓冲解决的问题，我们再添加一个矩形的数据：

```
const std::vector<Vertex> vertices = {
{{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
{{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
{{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
{{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}},

{{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
{{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
{{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
{{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};

const std::vector<uint16_t> indices = {
            0, 1, 2, 2, 3, 0,
            4, 5, 6, 6, 7, 4
};
```

新添加的矩形和之前的矩形除了 z 坐标相差 0.5f 外，其它数据完全一样。另外我们将第二矩形使用的顶点索引数据添加到了 indices 数组中。

现在编译运行程序，应该可以看到类似上图的画面。

我们会发现较低的矩形却绘制在了较高的矩形上面，这是因为较低矩形的索引数据在较高矩形的索引数据之后出现，绘图调用在绘制完较高的矩形后才绘制较低的矩形。

有两种方式可以解决这个问题：

- 按照深度坐标对绘制调用进行排序
- 使用深度测试

通常我们使用第一种方式来绘制需要透明的对象。使用第二种方式来绘制一般对象。使用第二种方式需要用到深度缓冲。深度缓冲是用于存储片段深度值的缓冲对象。光栅化后生成的片段包含了一个深度值可以被深度测试检查是否可以使用这一片段覆盖之前的数据。我们可以在片段着色器对深度值进行处理。

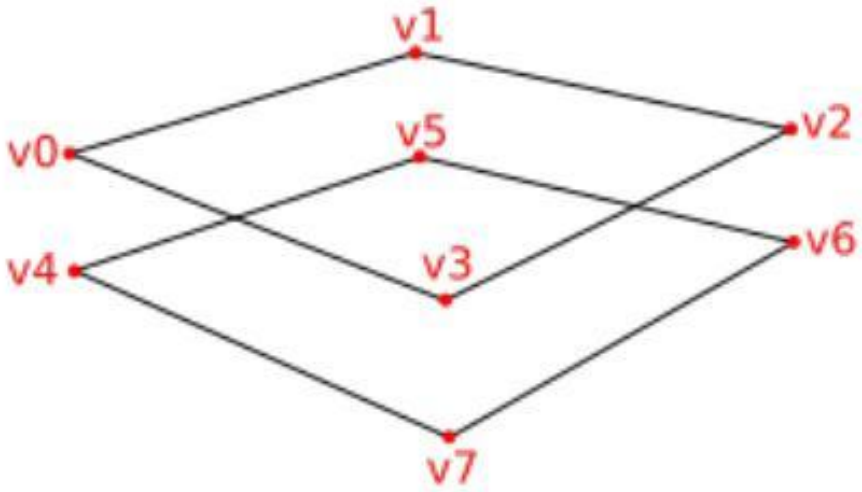

Figure 54: image

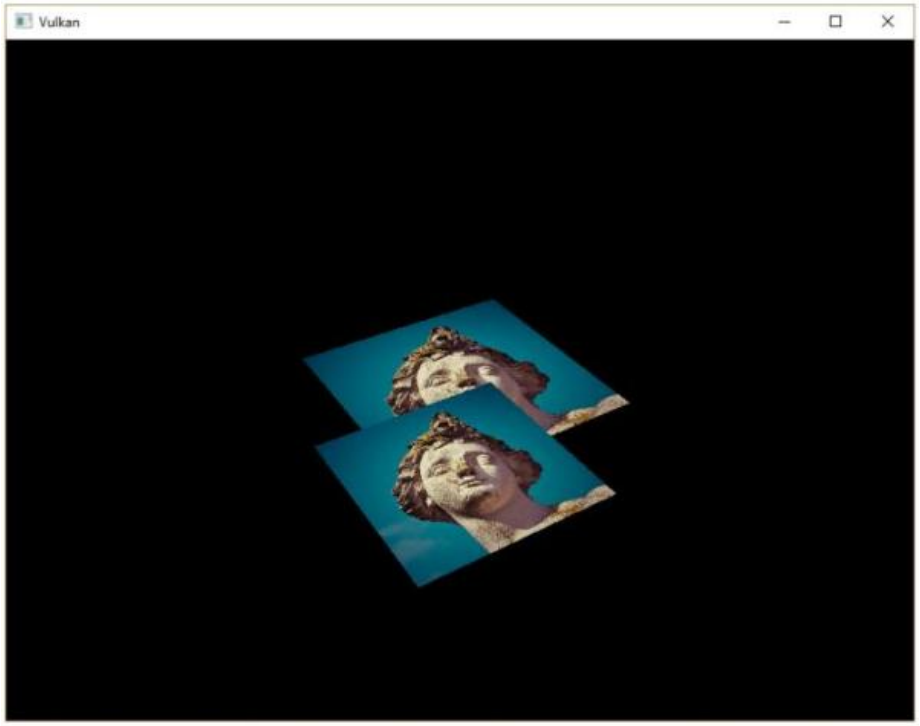

Figure 55: image

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

默认情况下，GLM 库的透视投影矩阵使用 OpenGL 的深度值范围 (-1.0，1.0)。我们需要定义 GLM_FORCE_DEPTH_ZERO_TO_ONE 宏来让它使用 Vulkan 的深度值范围 (0.0，1.0)。

深度图像和视图

深度附着和颜色附着一样都是基于图像对象。区别是，交换链不会自动地为我们创建深度附着使用的深度图像对象。我们需要自己创建深度图像对象。使用深度图像需要图像、内存和图像视图对象这三种资源。

```
VkImage depthImage;
VkDeviceMemory depthImageMemory;
VkImageView depthImageView;
```

添加一个叫做 createDepthResources 的函数来配置深度图像需要的资源：

```
void initVulkan() {
        ...
    createCommandPool();
    createDepthResources();
    createTextureImage();
        ...
}


        ...

void createDepthResources() {

}
```

深度图像的创建方法相当简单。设置和颜色附着完全一样的图像分辨率，以及用于深度附着的使用标记，优化的 tiling 模式和设备内存。对于深度图像，只需要使用一个颜色通道。

和纹理图像不同，深度图像不需要特定的图像数据格式，这是因为实际上我们不需要直接访问深度图像。只需要保证深度数据能够有一个合理的精度即可，通常这一合理精度是至少为深度数据提供 24 位的位宽，下面这些值满足 24 位的需求：

- VK_FORMAT_D32_SFLOAT：32 位浮点深度值
- VK_FORMAT_D32_SFLOAT_S8_UINT：32 位浮点深度值和 8 位模板值

- VK_FORMAT_D24_UNORM_S8_UINT：24 位浮点深度值和 8 位模板值

模板值用于模板测试，我们会在之后的章节介绍它。

我们可以直接使用 VK_FORMAT_D32_SFLOAT 作为图像数据格式，目前这一图像数据格式已经被广泛支持。但我们最好还是对设备是否支持这一图像数据格式进行检测。为此我们可以编写一个 findSupportedFormat 函数查找一个既符合我们需求又被设备支持的图像数据格式：

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,

}
```

可用的深度图像数据格式还依赖于 tiling 模式和使用标记，所以我们需要将这些信息作为函数参数，通过调用 vkGetPhysicalDeviceFormatProperties 函数查询可用的深度图像数据格式：

```
for (VkFormat format : candidates) {
    VkFormatProperties props;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
}
```

上面代码中的 VkFormatProperties 结构体包含了下面这些成员变量：

- linearTilingFeatures：数据格式支持线性 tiling 模式
- optimalTilingFeatures：数据格式支持优化 tiling 模式
- bufferFeatures：数据格式支持缓冲

目前我们只用到上面前两个成员变量，通过它们对应的 tiling 模式检测数据格式是否被支持：

```
if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features)
    return format;
} else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) ==
    return format;
}
```

如果没有合适的格式可用，我们可以返回一个特殊值或直接抛出一个异常：

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == f
        {
            return format;
        } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & featu
            return format;
```

```
        }
    }

    throw std::runtime_error("failed to find supported format!");
}
```

我们使用 findSupportedFormat 函数创建一个叫做 findDepthFormat 的辅助函数来查找适合作为深度附着的图像数据格式：

```
VkFormat findDepthFormat() {
    return findSupportedFormat( {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORM
}
```

需要注意这里使用的是 VK_FORMAT_FEATURE_ 类的标记而不是 VK_IMAGE_USAGE_ 类标记，这一类标记的格式都包含了深度颜色通道，有一些还包含了模板颜色通道，但在这里，我们没有用到模板颜色通道。但如果使用的格式包含模板颜色通道，在进行图像布局变换时就需要考虑这一点。这里我们添加一个可以用于检测格式是否包含模板颜色通道的函数：

```
bool hasStencilComponent(VkFormat format) {
    return format == VK_FORMAT_D32_SFLOAT_S8_UINT || format == VK_FORMAT_D24_UNORM_S8_UINT;
}
```

我们在 createDepthResources 函数中调用 findDepthFormat 函数来查找一个可用的深度图像数据格式：

```
VkFormat depthFormat = findDepthFormat();
```

至此，我们已经具有足够的信息来创建深度图像对象，可以开始调用 createImage 和 createImageView 函数来创建图像资源：

```
createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTI
depthImageView = createImageView(depthImage, depthFormat);
```

但我们的 createImageView 函数目前假设 aspectFlags 的设置总是 VK_IMAGE_ASPECT_COLOR_BIT，不符合我们的需求，我们需要进行修改将其作为一个函数参数：

```
VkImageView createImageView(VkImage image, VkFormat format,
VkImageAspectFlags aspectFlags) {
        ...
    viewInfo.subresourceRange.aspectMask = aspectFlags;
        ...
}
```

更新所有使用 createImageView 函数的地方，使用正确的 aspectFlags 标记：

```
swapChainImageViews[i] = createImageView(swapChainImages[i],
swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT);
        ...
depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);
        ...
textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_C
```

接着，就是创建深度图像。通常，我们在渲染流程开始后首先会清除深度附着中的数据，不需要复制数据到深度图像中。我们需要对图像布局进行变换，来让它适合作为深度附着使用。由于这一图像变换只需要进行一次，这里我们使用管线障碍来同步图像变换：

```
transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_DE
```

因为我们不需要深度图像之前的数据,所以我们使用 VK_IMAGE_LAYOUT_UNDEFINED 作为深度图像的初始布局。现在我们需要修改 transitionImageLayout 函数的部分代码来使用正确的 subresource aspect：

```
if (newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;

    if (hasStencilComponent(format)) {
        barrier.subresourceRange.aspectMask |= VK_IMAGE_ASPECT_STENCIL_BIT;
    }
} else {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
}
```

尽管这里我们没有使用模板颜色通道，我们还是要对它进行变换处理。

最后，设置正确的访问掩码和管线阶段。

```
if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTI
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
    destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_DEPTH_STEN
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_ACCESS_DEPTH_ST

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
} else {
    throw std::invalid_argument("unsupported layout transition!");
}
```

深度缓冲数据会在进行深度测试时被读取,用来检测片段是否可以覆盖之前的片段。这一读取过程发生在 VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT，

如果片段可以覆盖之前的片段,新的深度缓冲数据会在 VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_
写入。我们应该使用与要进行的操作相匹配的管线阶段进行同步操作。

渲染流程

现在让我们修改 createRenderPass 函数，通过 VkAttachementDescription 结构体设置深度附着信息:

```
VkAttachmentDescription depthAttachment = {};
depthAttachment.format = findDepthFormat();
depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

format 成员变量的值的设置应该和深度图像的图像数据格式相同。我们在绘制结束后不需要从深度缓冲中复制深度数据，所以将 storeOp 成员变量设置为 VK_ATTACHMENT_STORE_OP_DONT_CARE。这一设置可以让驱动进行一定程度的优化。同样，我们不需要读取之前深度图像数据，所以将 initialLayout 成员变量设置为 VK_IMAGE_LAYOUT_UNDEFINED。

```
VkAttachmentReference depthAttachmentRef = {};
depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

为我们仅有的子流程添加对深度附着的引用:

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;
```

和颜色附着不同，一个子流程只可以使用一个深度 (或深度模板) 附着。一般而言，也很少有需要多个深度附着的情况。

```
std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

最后，更新 VkRenderPassCreateInfo 结构体信息引用深度附着。

帧缓冲

下一步就是修改创建帧缓冲的代码，绑定深度图像作为帧缓冲的深度附着。在 createFramebuffers 函数中指定深度图像视图对象作为帧缓冲的第二个附着：

```
std::array<VkImageView, 2> attachments = {
    swapChainImageViews[i],
    depthImageView
};

VkFramebufferCreateInfo framebufferInfo = {};
framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
framebufferInfo.renderPass = renderPass;
framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
framebufferInfo.pAttachments = attachments.data();
framebufferInfo.width = swapChainExtent.width;
framebufferInfo.height = swapChainExtent.height;
framebufferInfo.layers = 1;
```

和每个交换链图像对应不同的颜色附着不同，使用我们这里的信号量设置，同时只会有一个子流程在执行，所以，这里我们只需要使用一个深度附着即可。

我们需要在 createFramebuffers 函数调用前创建深度图像相关的对象：

```
void initVulkan() {
        ...
    createDepthResources();
    createFramebuffers();
        ...
}
```

清除值

现在我们使用了多个使用 VK_ATTACHMENT_LOAD_OP_CLEAR 标记的附着，这也意味着我们需要设置多个清除值。在 createCommandBuffers 函数中添加一个 VkClearValue 结构体数组：

```
std::array<VkClearValue, 2> clearValues = {};
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};
clearValues[1].depthStencil = {1.0f, 0};

renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
renderPassInfo.pClearValues = clearValues.data();
```

Vulkan 的深度值范围是 [0.0，1.0]，1.0 对应视锥体的远平面，0.0 对应视锥体的近平面。深度缓冲的初始值应该设置为远平面的深度值，也就是 1.0。

深度和模板状态

至此，深度附着已经可以使用，只剩下开启图形管线的深度测试功能。我们需要通过 VkPipelineDepthStencilStateCreateInfo 结构体来设置深度测试：

```
VkPipelineDepthStencilStateCreateInfo depthStencil = {};
depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
```

depthTestEnable 成员变量用于指定是否启用深度测试。depthWriteEnable 成员变量用于指定片段通过深度测试后是否写入它的深度值到深度缓冲。使用这两个成员变量可以实现透明效果。透明对象的片段的深度值需要和之前不透明对象片段的深度值进行比较，但透明对象的片段的深度值不需要写入深度缓冲。

```
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

depthCompareOp 成员变量用于指定深度测试使用的比较运算。这里我们指定深度测试使用小于比较运算，这一设置下，新的片段只有在它的深度值小于深度缓冲中的深度值时才会被写入颜色附着。

```
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
```

depthBoundsTestEnable、minDepthBounds 和 maxDepthBounds 成员变量用于指定可选的深度范围测试。这一测试开启后只有深度值位于指定范围内的片段才不会被丢弃。这里我们不使用这一功能。

```
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```

stencilTestEnable、front 和 back 成员变量用于模板测试，在我们的教程中没有用到。如果读者想要使用模板测试，需要注意使用包含模板颜色通道的图像数据格式。

```
pipelineInfo.pDepthStencilState = &depthStencil;
```

更新之前创建图形管线时填写的 VkGraphicsPipelineCreateInfo 结构体信息，引用我们刚刚设置的深度模板缓冲状态信息。如果渲染流程包含了深度模板附着，那就必须指定深度模板状态信息。

现在编译运行程序，读者就可以看到被正确渲染的几何图元：

处理窗口大小变化

当窗口大小变化时，需要对深度缓冲进行处理，让深度缓冲的大小和新的窗口大小相匹配。为此我们扩展 recreateSwapChain 函数在窗口大小改变时重建深度缓冲：

```
void recreateSwapChain() {
    vkDeviceWaitIdle(device);
```

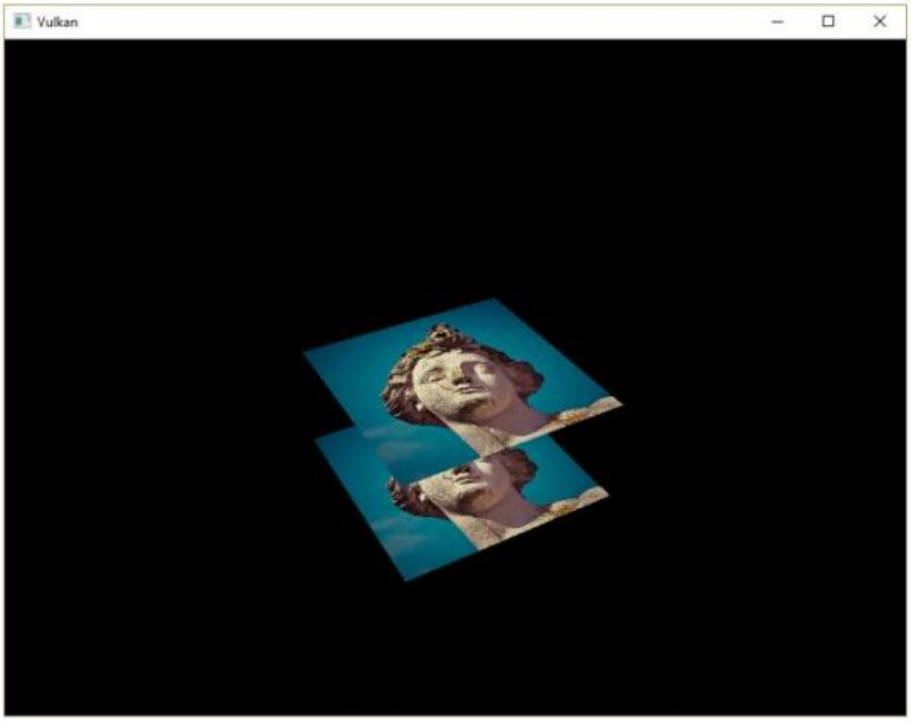

Figure 56: image

```
    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createDepthResources();
    createFramebuffers();
    createCommandBuffers();
}
```

最后，需要注意在交换链的清除函数 cleanupSwapChain 中添加对深度缓冲相关的清除操作：

```
void cleanupSwapChain() {
    vkDestroyImageView(device, depthImageView, nullptr);
    vkDestroyImage(device, depthImage, nullptr);
    vkFreeMemory(device, depthImageMemory, nullptr);

        ...
}
```

至此，我们就可以使用深度测试来渲染三维对象。在下一章节，我们将介绍如何绘制一个带有纹理的三维模型。

本章节代码：

C++：

https://vulkan-tutorial.com/code/26_depth_buffering.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/26_shader_depth.vert

Fragment Shader：

https://vulkan-tutorial.com/code/26_shader_depth.frag

载入模型

介绍

本章节我们将会渲染一个带有纹理的三维模型。

库

我们使用 tinyobjloader 库来从 OBJ 文件加载顶点数据。tinyobjloader 库是一个简单易用的单文件 OBJ 加载器，我们只需要下载 tiny_obj_loader.h 文件，然后在代码中包含这一头文件就可以使用它了。

Visual Studio

将 tiny_obj_loader.h 加入 Additional Include Directories paths 中：

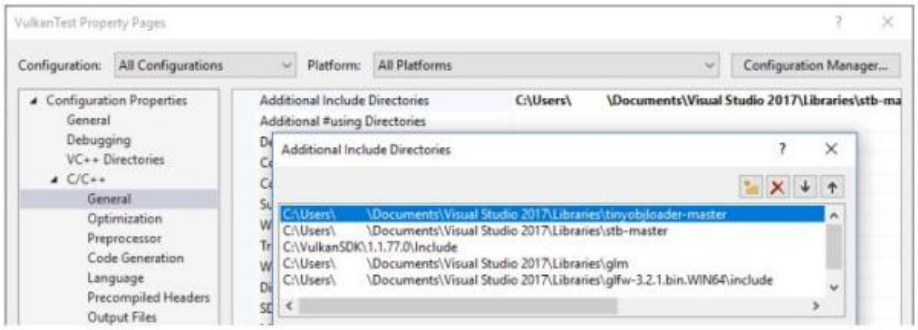

Figure 57: image

Makefile

将 tiny_obj_loader.h 所在目录加入编译器的包含目录中：

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb
TINYOBJ_INCLUDE_PATH = /home/user/libraries/tinyobjloader

        ...

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH) -I$(TINYOBJ_INCLUDE_P
```

网格样例

在本章节，我们暂时不使用光照，只简单地将纹理贴在模型上。读者可以从 Sketchfab 找到自己喜欢的 OBJ 模型来加载。

在这里，我们加载的模型叫做 Chalet Hippolyte Chassande Baroz。我们对它的大小和方向进行了调整：

- chalet.obj
- chalet.jpg

这一模型大概由 50 万面三角形构成。读者也可以使用自己的 OBJ 模型，但需要确保使用的模型给只包含了一个材质，并且模型的大小为 1.5x1.5x1.5。如果使用的模型大于这一尺寸，读者就需要对使用的视图矩阵进行修改。我们新建一个和 shaders 和 textures 文件夹同级的 models 文件夹，用于存放模型文件。

添加两个常量定义我们使用的模型文件路径和纹理文件路径：

```
const int WIDTH = 800;
const int HEIGHT = 600;

const std::string MODEL_PATH = "models/chalet.obj";
const std::string TEXTURE_PATH = "textures/chalet.jpg";
```

修改 createTextureImage 函数使用我们定义的纹理路径常量加载纹理图像：

```
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels, STBI_
```

载入顶点和索引

现在我们从模型文件加载顶点数据和索引数据，删除之前我们定义的 vertices 和 indices 这两个全局变量，使用大小可变的向量来重新定义它们：

```
std::vector<Vertex> vertices;
std::vector<uint32_t> indices;
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
```

由于我们使用的模型包含的顶点个数远远大于 65535，所以不能使用 uint16_t 作为索引的数据类型，而应该使用 uint32_t 作为索引的数据类型。更改索引数据类型后，还要修改我们调用 vkCmdBindIndexBuffer 函数时使用的参数：

```
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
```

tinyobjloader 库的使用和 STB 库类似,我们需要定义 TINYOBJLOADER_IMPLEMENTATION 宏来让它包含函数实现，不然就会在编译时出现链接错误：

```
#define TINYOBJLOADER_IMPLEMENTATION
#include <tiny_obj_loader.h>
```

现在编写用于载入模型文件的 loadModel 函数，它负责填充模型数据到 vertices 和 indices。我们在顶点缓冲和索引缓冲创建之前调用它来载入模型数据：

```
void initVulkan() {
        ...
    loadModel();
    createVertexBuffer();
    createIndexBuffer();
        ...
}


        ...

void loadModel() {

}
```

模型数据的载入是通过调用 tinyobj::LoadObj 完成的：

```
void loadModel() {
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    std::string err;

    if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &err, MODEL_PATH.c_str())) {
        throw std::runtime_error(err);
    }
}
```

一个 OBJ 模型文件包含了模型的位置、法线、纹理坐标和表面数据。表面数据包含了构成表面的多个顶点数据的索引。

我们在 loadModel 函数中使用 attrib 变量来存储载入的位置、法线和纹理坐标数据。使用 shapes 变量存储独立的对象和它们的表面数据。每个表面数据包含了一个顶点数组，顶点数组中的每个顶点数据包含了顶点的位置索引、法线索引和纹理坐标索引。OBJ 模型文件格式允许为模型的每个表面定义材质和纹理数据，但在这里，我们没有用到。

我们使用 err 变量来存储载入模型文件时产生的错误和警告信息，比如载入时没有找到引用的材质信息。如果载入模型文件失败，那么 tinyobj::LoadObj 函数就会返回 false。之前提到，OBJ 模型文件中的表面数据可以包含任意数量的顶点数据，但我们的程序只能渲染三角形表面，这就需要进行转换将 OBJ 模型文件中的表面数据都转换为三角形表面。tinyobj::LoadObj 函数有一个可选的默认参数，可以设置在加载 OBJ 模型数据时将表面数据转换为三角形表面。由于这一设置是默认的，所以，我们不需要自己设置它。

接着，我们将加载的表面数据复制到我们的 vertices 和 indices 向量中，这只需要遍历 shapes 向量即可：

```
for (const auto& shape : shapes) {

}
```

载入的表面数据已经被三角形化，所以我们可以直接将它们复制到 vertices 向量中：

```
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex = {};

        vertices.push_back(vertex);
        indices.push_back(indices.size());
    }
}
```

为了简化 indices 数组的处理，我们这里假定每个顶点都是独一无二的，可以直接使用 indices 数组的当前大小作为顶点索引数据。上面代码中的 index 变量的类型为 tinyobj::index_t，这一类型的变量包含了 vertex_index、normal_index 和

texcoord_index 三个成员变量。我们使用这三个成员变量来检索存储在 attrib 数组变量中的顶点数据：

```
vertex.pos = {
    attrib.vertices[3 * index.vertex_index + 0],
    attrib.vertices[3 * index.vertex_index + 1],
    attrib.vertices[3 * index.vertex_index + 2]
};

vertex.texCoord = {
    attrib.texcoords[2 * index.texcoord_index + 0],
    attrib.texcoords[2 * index.texcoord_index + 1]
};

vertex.color = {1.0f, 1.0f, 1.0f};
```

attrib.vertices 是一个浮点数组，并非 glm::vec3 数组，我们需要在使用索引检索顶点数据时首先要把索引值乘以 3 才能得到正确的顶点数据位置。对于纹理坐标数据，则乘以 2 进行检索。对于顶点位置数据，偏移值 0 对应 X 坐标，偏移值 1 对应 Y 坐标，偏移值 2 对应 Z 坐标。对于纹理坐标数据，偏移值 0 对应 U 坐标，偏移值 1 对应 V 坐标。

现在使用优化模式编译我们的程序 (使用 Visual Studio 的 Release 模式或 GCC 的-O3 编译选项)。这样做可以提高我们的模型加载速度。运行程序，应该可以看到下面这样的画面：

看起来，模型的几何形状是正确的，但纹理映射不对。这是因为 Vulkan 的纹理坐标的原点是左上角，而 OBJ 模型文件格式假设纹理坐标原点是左下角。我们可以通过反转纹理的 Y 坐标解决这一问题：

```
vertex.texCoord = {
    attrib.texcoords[2 * index.texcoord_index + 0], 1.0f - attrib.texcoords[2 * index.texcoo
};
```

现在再次编译运行程序，就可以看到被正确映射纹理的模型了：

顶点去重

按照之前的处理，我们没有达到索引缓冲节约空间的目的。三角形表面的顶点是被多个三角形表面共用的，而我们则是每个顶点都重新定义一次，vertices 向量包含了大量重复的顶点数据。我们可以将完全相同的顶点数据只保留一个，来解决空间。这一去重过程可以通过 STL 的 map 或 unordered_map 来实现：

```
#include <unordered_map>

        ...

std::unordered_map<Vertex, uint32_t> uniqueVertices = {};
```

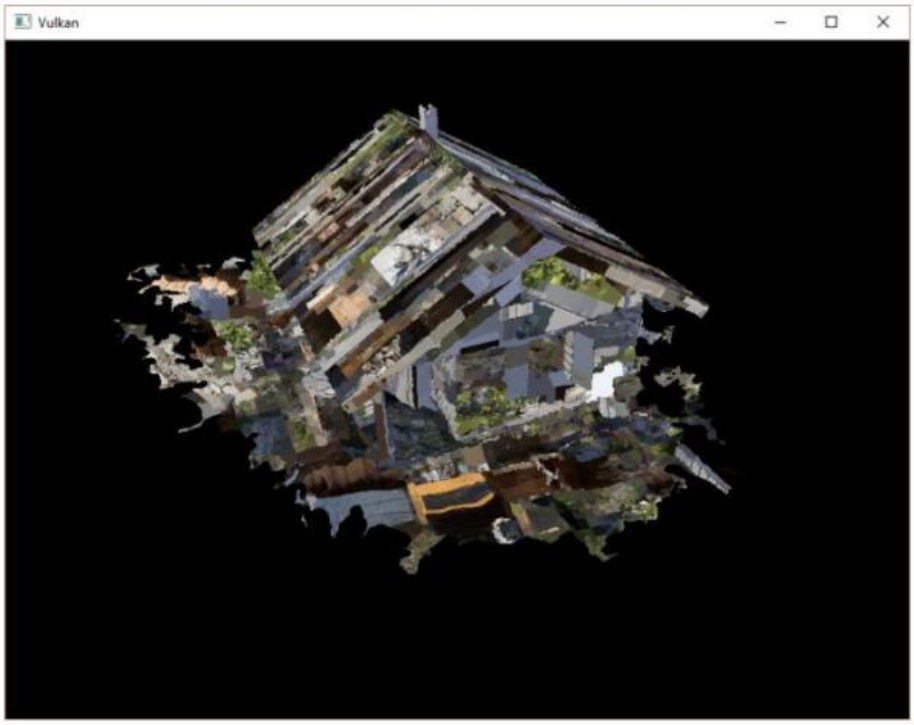

Figure 58: image

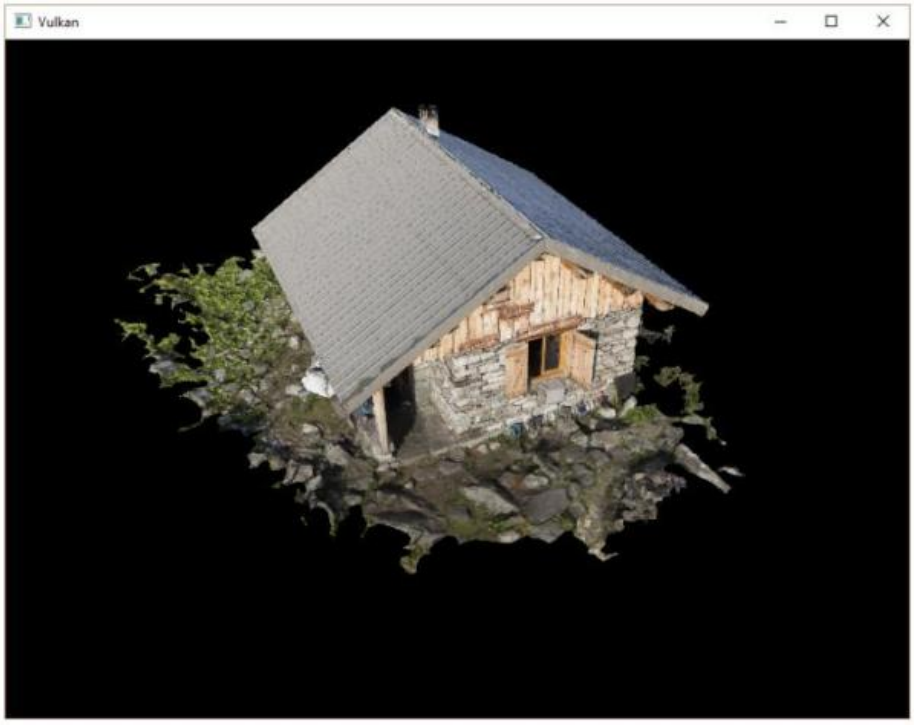

Figure 59: image

```
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex = {};

        ...

        if (uniqueVertices.count(vertex) == 0) {
            uniqueVertices[vertex] = static_cast<uint32_t>(vertices.size());
            vertices.push_back(vertex);
        }

        indices.push_back(uniqueVertices[vertex]);
    }
}
```

在从 OBJ 模型文件加载模型数据时，我们检查加载的顶点数据是否与已经加载的数据完全相同，如果相同，就不再将其加入 vertices 向量，将已经加载的顶点数据的索引存储到 indices 向量中。如果不同，将其加入 vertices 向量，并存储它对应的索引值到 uniqueVertices 容器中。然后将其索引存储在 indices 向量中。

我们需要实现两个函数来让 Vertex 结构体可以作为 map 变量的键值来检索 map 变量，首先是 == 函数：

```
bool operator==(const Vertex& other) const {
    return pos == other.pos && color == other.color && texCoord == other.texCoord;
}
```

然后是对 Vertex 结构体进行哈希的函数：

```
namespace std {
    template<> struct hash<Vertex> {
        size_t operator()(Vertex const& vertex) const {
            return ((hash<glm::vec3>()(vertex.pos) ^ (hash<glm::vec3>()(vertex.color) << 1)
        }
    };
}
```

上面这两个函数的代码需要放在 Vertex 结构体的定义外。GLM 库的变量类型的哈希函数可以通过下面的代码包含到我们的程序中：

```
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/gtx/hash.hpp>
```

GLM 库的哈希函数目前还是一个试验性的扩展，被定义在了 GLM 库的 gtx 目录下。所以需要我们定义 GLM_ENABLE_EXPERIMENTAL 宏来启用它。作为试验性扩展意味着在未来版本的 GLM 库有可能发生变化，但一般而言，我们可以认为变化不会太大。

现在重新编译运行程序，查看 vertices 向量的大小，可以发现 vertices 向量的大小从 1，500，000 下降到了 265，645。这也说明对于我们的模型数据，每个顶点数据

平均被 6 个三角形表面使用。

本章节代码：

C++：

https://vulkan-tutorial.com/code/27_model_loading.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/26_shader_depth.vert

Fragment Shader：

https://vulkan-tutorial.com/code/26_shader_depth.frag

生成细化贴图

介绍

在上一章节，我们成功将 OBJ 模型文件中的模型渲染出来。在这一章节，我们为我们的程序添加一个新的特性：细化贴图。细化贴图在游戏和渲染软件中的应用非常广泛，Vulkan 同样支持细化贴图这一特性。

细化贴图是预先计算的不同缩放层次的一组纹理图像。这一组纹理图像按照图像大小从大到小排列，每张图像的大小通常是上一张图像大小的一半。细化贴图经常被用来实现层次化细节效果 (LOD)。通过对离摄像机较远的对象使用较小的纹理图像，对于较近的对象使用较精细的纹理图像，从而达到既不损失效果，又节约计算资源的目的。

创建图像

对于 Vulkan 来说，一组细化贴图的每一张图像需要使用不同的细化级别设置存储在 VkImage 对象中。细化级别设置为 0 的 VkImage 对象存储的是源图像，细化级别设置大于 0 的 VkImage 对象会被细化链引用使用。

细化级别是在创建 VkImage 对象时设置的，之前，我们一直将其设置为 1。现在我们根据图像的尺寸来计算它。添加一个类成员变量来存储计算出的细化级别个数：

```
...
uint32_t mipLevels;
VkImage textureImage;
        ...
```

mipLevels 成员变量的值可以在 createTextureImage 函数载入纹理图像时计算得到：

```
int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels, STBI_
        ...
mipLevels = static_cast<uint32_t>(std::floor(std::log2(std::max(texWidth, texHeight)))) + 1;
```

Figure 60: image

上面代码我们通过 max 函数来选择纹理尺寸中较大的那个，然后对其对以 2 为底的对数，并将结果向下取整后 +1 得到细化级别的个数，也就是 mipLevels 的值。

我们需要对 createImage、createImageView 和 transitionImageLayout 函数进行修改为它们添加 mipLevels 参数：

```
void createImage(uint32_t width, uint32_t height, uint32_t mipLevels, VkFormat format, VkIma
        ...
    imageInfo.mipLevels = mipLevels;
        ...
}

VkImageView createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags,
        ...
    viewInfo.subresourceRange.levelCount = mipLevels;
        ...

void transitionImageLayout(VkImage image, VkFormat format,
VkImageLayout oldLayout, VkImageLayout newLayout, uint32_t mipLevels) {
        ...
    barrier.subresourceRange.levelCount = mipLevels;
        ...
```

更新对它们进行的调用，确保使用正确的参数进行调用：

```
createImage(swapChainExtent.width, swapChainExtent.height, 1,  depthFormat, VK_IMAGE_TILING_
        ...
createImage(texWidth, texHeight, mipLevels, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_TILING_OPTIMA

swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat, VK_IMAGE_
        ...
depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT, 1);
        ...
textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_C

transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED,
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, 1);
        ...
    transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_UNDEFINED,
```

生成细化贴图

我们的纹理图像现在有了多个细化级别，但我们目前还只有 0 级细化的图像数据 (也就是原始图像数据)，我们需要使用原始图像数据来生成其它细化级别的图像数据。这可以通过使用 vkCmdBlitImage 指令来完成。vkCmdBlitImage 指令可以进行复制，缩放和过滤图像的操作。我们会多次调用这一指令来生成多个不同细化级别的纹理图像数据。

vkCmdBlitImage 指令执行的是传输操作，使用这一指令，需要我们为纹理图像添加作为数据来源和接收目的使用标记。我们在 createTextureImage 函数中为创建的纹理图像添加这些使用标记：

```
...
createImage(texWidth, texHeight, mipLevels, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_TILING_OPTIMA
        ...
```

和其它图像操作一样，vkCmdBlitImage 指令对于它处理的图像的布局有一定的要求。我们可以将整个图像转换到 VK_IMAGE_LAYOUT_GENERAL 布局，这一图像布局满足大多数的指令需求，但使用这一图像布局进行操作的性能表现不能达到最佳。为了达到最佳性能表现，对于源图像，应该使用 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL 图像布局。对于目的图像，应该使用 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 图像布局。Vulkan 允许我们独立地对一张图像的不同细化级别进行布局变换。而每个传输操作一次只会对两个细化级别进行处理，所以，我们可以在使用传输指令前，将使用的两个细化级别的图像布局转换到最佳布局。

transitionImageLayout 函数会对整个图像进行布局变换，需要我们编写管线屏障指令。将 createTextureImage 函数图像布局变换到 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMA 布局的代码：

```
...
    transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_UNDEFINED,
    copyBufferToImage(stagingBuffer, textureImage,static_cast<uint32_t>(texWidth), static_ca
```

```
    //transitioned to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL while generating mipmaps
        ...
```

上面的代码会让纹理图像的每个细化级别变换为 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 布局。在传输指令读取完成后,每个细化级别会被变换到 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPT 布局。

接着，我们编写用于生成原始纹理图像不同细化级别的 generateMipmaps 函数:

```
void generateMipmaps(VkImage image, int32_t texWidth, int32_t texHeight, uint32_t mipLevels)
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    VkImageMemoryBarrier barrier = {};
    barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    barrier.image = image;
    barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    barrier.subresourceRange.baseArrayLayer = 0;
    barrier.subresourceRange.layerCount = 1;
    barrier.subresourceRange.levelCount = 1;

    endSingleTimeCommands(commandBuffer);
}
```

我们使用同一个 VkImageMemoryBarrier 对象对多次图像布局变换进行同步。上面代码中对于 barrier 的设置，只需设置一次，无需修改。subresourceRange.miplevel、oldLayout、srcAccessMask 和 dstAccessMask 这几个 barrier 的成员变量则需要在每次变换前根据需要进行修改。

```
int32_t mipWidth = texWidth;
int32_t mipHeight = texHeight;

for (uint32_t i = 1; i < mipLevels; i++) {

}
```

我们使用循环来遍历所有细化级别，记录每个细化级别使用的 VkCmdBlitImage 指令到指令缓冲中。需要注意，这里的循环变量 i 是从 1 开始，不是从 0 开始的。

```
barrier.subresourceRange.baseMipLevel = i - 1;
barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_TRANSF
0, nullptr, 1, &barrier);
```

上面代码,我们设置将细化级别为 i-1 的纹理图像变换到 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL

布局。这一变换会在细化级别为 i-1 的纹理图像数据被写入后 (上一次的传输指令写入或 vkCmdCopyBufferToImage 指令写入) 进行。当前的传输指令会等待这一变换结束才会执行。

```
VkImageBlit blit = {};
blit.srcOffsets[0] = { 0, 0, 0 };
blit.srcOffsets[1] = { mipWidth, mipHeight, 1 };
blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
blit.srcSubresource.mipLevel = i - 1;
blit.srcSubresource.baseArrayLayer = 0;
blit.srcSubresource.layerCount = 1;
blit.dstOffsets[0] = { 0, 0, 0 };
blit.dstOffsets[1] = { mipWidth > 1 ? mipWidth / 2 : 1, mipHeight > 1 ? mipHeight / 2 : 1, 1
blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
blit.dstSubresource.mipLevel = i;
blit.dstSubresource.baseArrayLayer = 0;
blit.dstSubresource.layerCount = 1;
```

接着,我们指定传输操作使用的纹理图像范围。这里我们将 srcSubresource.mipLevel 成员变量设置为 i-1, 也就是上一细化级别的纹理图像。将 dstSubresource.mipLevel 成员变量设置为 i, 也就是我们要生成的纹理图像的细化级别。srcOffsets 数组用于指定要传输的数据所在的三维图像区域。dstOffsets 数组用于指定接收数据的三维图像区域。dstOffsets[1] 的 X 分量和 Y 分量的值需要设置为上一细化级别纹理图像的一半。由于这里我们使用的是二维图像, 二维图像的深度值都为 1, 所以 srcOffsets[1] 和 dstOffsets[1] 的 Z 分量都必须设置为 1。

```
vkCmdBlitImage(commandBuffer, image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, image, VK_IMAGE_I
```

现在,我们可以开始记录传输指令到指令缓冲。可以注意到我们将 textureImage 变量同时作为 vkCmdBlitImage 指令的源图像和目的图像。这是因为我们的传输操作是在同一纹理对象的不同细化级别间进行的。传输指令开始执行时源细化级别图像布局刚刚被变换为 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,目的细化级别图像布局仍处于创建纹理时设置的 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 布局。

vkCmdBlitImage 指令的最后一个参数用于指定传输操作使用的 VkFilter 对象。这里我们使用和 VkSampler 一样的 VK_FILTER_LINEAR 过滤设置, 进行线性插值过滤。

```
barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
barrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGME
```

上面代码,我们设置屏障将细化级别为 i-1 的图像布局变换到 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_ 这一变换会等待当前的传输指令结束会才会进行。所有采样操作需要等待这一变换结束才能进行。

```
...
    if (mipWidth > 1) mipWidth /= 2;
    if (mipHeight > 1) mipHeight /= 2;
}
```

在遍历细化级别的循环结尾处，我们将 mipWidth 变量和 mipHeight 变量的值除以 2，计算出下一次循环要使用的细化级别的图像大小。这里的代码，可以处理图像的长宽不同的情况。当图像的长或宽为 1 时，就不再对其缩放。

```
barrier.subresourceRange.baseMipLevel = mipLevels - 1;
    barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FR

    endSingleTimeCommands(commandBuffer);
}
```

在我们开始执行指令缓冲前，我们需要插入一个管线障碍用于将最后一个细化级别的图像布局从 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 变换为 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL。这样做是因为最后一个细化级别的图像不会被作为传输指令的数据来源，所以就不会将布局变换为 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL，需要我们手动进行变换。

最后，在 createTextureImage 函数中添加对 generateMipmaps 函数的调用：

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_UNDEFINED, VK_
copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth),  static_cast
//transitioned to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL while generating mipmaps
        ...
generateMipmaps(textureImage, texWidth, texHeight, mipLevels);
```

至此，我们就生成了所有细化级别的纹理图像。

线性过滤支持

虽然使用内建的 vkCmdBlitImage 指令来生成纹理的所有细化级别非常方便，但并非所有平台都支持这一指令。它需要我们使用的纹理图像格式支持线性过滤特性。我们可以通过调用 vkGetPhysicalDeviceFormatProperties 函数来检查这一特性是否被支持。

首先，为 generateMipmaps 函数添加一个参数用于指定使用的纹理图像格式：

```
void createTextureImage() {
        ...

    generateMipmaps(textureImage, VK_FORMAT_R8G8B8A8_UNORM, texWidth, texHeight, mipLevels);
```

```
}

void generateMipmaps(VkImage image, VkFormat imageFormat, int32_t texWidth, int32_t texHeigh

        ...
}
```

在 generateMipmaps 函数中，调用 vkGetPhysicalDeviceFormatProperties 函数查询指定的纹理图像格式的属性：

```
void generateMipmaps(VkImage image, VkFormat imageFormat, int32_t texWidth, int32_t texHeigh

    // Check if image format supports linear blitting
    VkFormatProperties formatProperties;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, imageFormat, &formatProperties);

        ...
```

VkFormatProperties 结构体包含了 linearTilingFeatures、optimalTilingFeatures 和 bufferFeatures 三个成员变量，每个成员变量描述了在使用对应模式下，可以使用的特性。我们的纹理图像使用优化的 tiling 模式创建，所以我们需要检查 optimalTilingFeatures 成员变量来确定我们使用的纹理图像格式是否支持线性过滤特性：

```
if (!(formatProperties.optimalTilingFeatures & VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR
    throw std::runtime_error("texture image format does not support linear blitting!");
}
```

此外，我们还可以编写一个函数遍历常见的纹理图像格式，查找其中支持线性块传输的格式。或者使用可以生成纹理细化级别的库，比如 stb_image_resize 来生成纹理的所有细化级别，然后使用加载源图像数据一样的方法加载它们。

通常实践中很少在运行时动态生成纹理的细化级别，而是预先生成然后存储在文件中由程序直接加载所有细化级别的纹理图像。

采样器

我们需要使用 VkSampler 对象来控制读取细化级别的纹理图像数据。Vulkan 允许我们指定 minLod、maxLod、mipLodBias 和 mipmapMode 参数用于对纹理进行细化级别采样。它的采样方式可以用下面的代码表示：

```
lod = getLodLevelFromScreenSize(); //smaller when the object is close, may be negative
lod = clamp(lod + mipLodBias, minLod, maxLod);

//clamped to the number of mip levels in the texture
level = clamp(floor(lod), 0, texture.mipLevels - 1);

if (mipmapMode == VK_SAMPLER_MIPMAP_MODE_NEAREST) {
    color = sample(level);
```

```
} else {
    color = blend(sample(level), sample(level + 1));
}
```

如果 samplerInfo.mipmapMode 变量的值为 VK_SAMPLER_MIPMAP_MODE_NEAREST，就会根据计算出的细化级别选择对应的纹理图像进行采样。如果它的值为 VK_SAMPLER_MIPMAP_MODE_LINEAR，则会使用计算出的相邻两个级别的纹理图像进行线性混合采样。

采样操作同样受上面代码中的 lod 变量影响：

```
if (lod <= 0) {
    color = readTexture(uv, magFilter);
} else {
    color = readTexture(uv, minFilter);
}
```

如果对象离相机较近，就会使用 magFilter 的过滤设置。如果对象离相机较远，就会使用 minFilter 的过滤设置。通常，lod 的值为非负数，当 lod 的值为 0 时我们认为对象离相机较近。通过设置 mipLodBias，我们可以对 Vulkan 采样使用的 lod 和 level 值进行一定程度的偏移。

在这里，我们将 minFilter 和 magFilter 都指定为 VK_FILTER_LINEAR。然后使用下面代码指定其它的细化采样使用的参数：

```
void createTextureSampler() {
        ...
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    samplerInfo.minLod = 0; // Optional
    samplerInfo.maxLod = static_cast<float>(mipLevels);
    samplerInfo.mipLodBias = 0; // Optional
        ...
}
```

这里，我们将 minLod 设置为 0，maxLod 设置为纹理图像的最大细化级别。我们不需要对 lod 的值进行偏移，所以将 mipLodBias 设置为 0。

现在编译运行程序，可以看到下面的画面：

为了方便读者观察使用与不使用细化级别的差异，我们将它们放在下面进行对比：

可以看出，使用包含细化级别的纹理可以得到更加光滑的渲染结果。

读者可以对采样器的设置进行修改来观察它们的渲染结果的影响。比如修改 minLod 的值，读者可以强制采样器不使用较小细化级别的纹理图像：

```
samplerInfo.minLod = static_cast<float>(mipLevels / 2);
```

下面是这一设置产生的渲染效果：

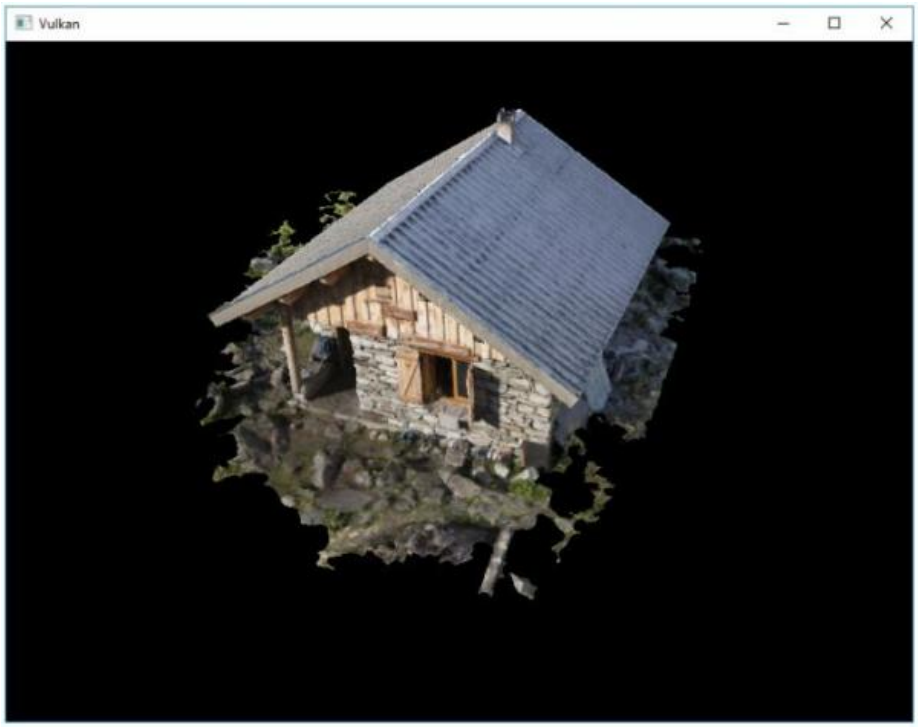

Figure 61: image

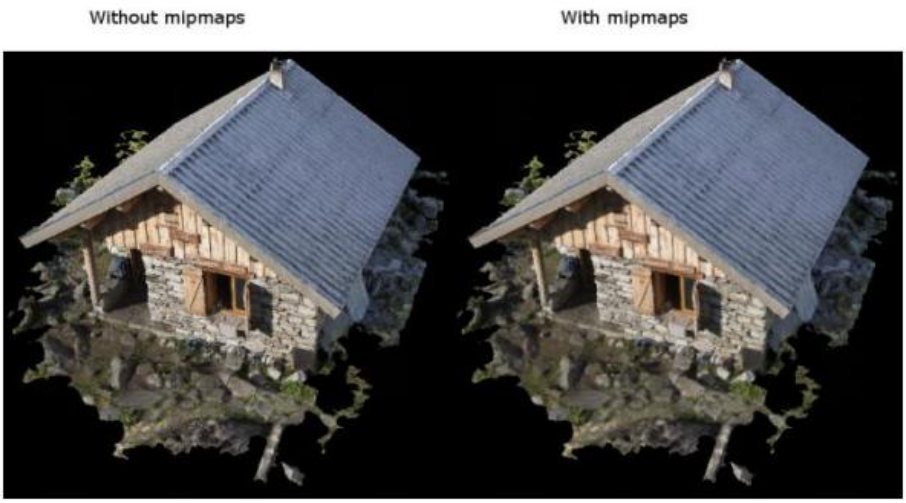

Figure 62: image

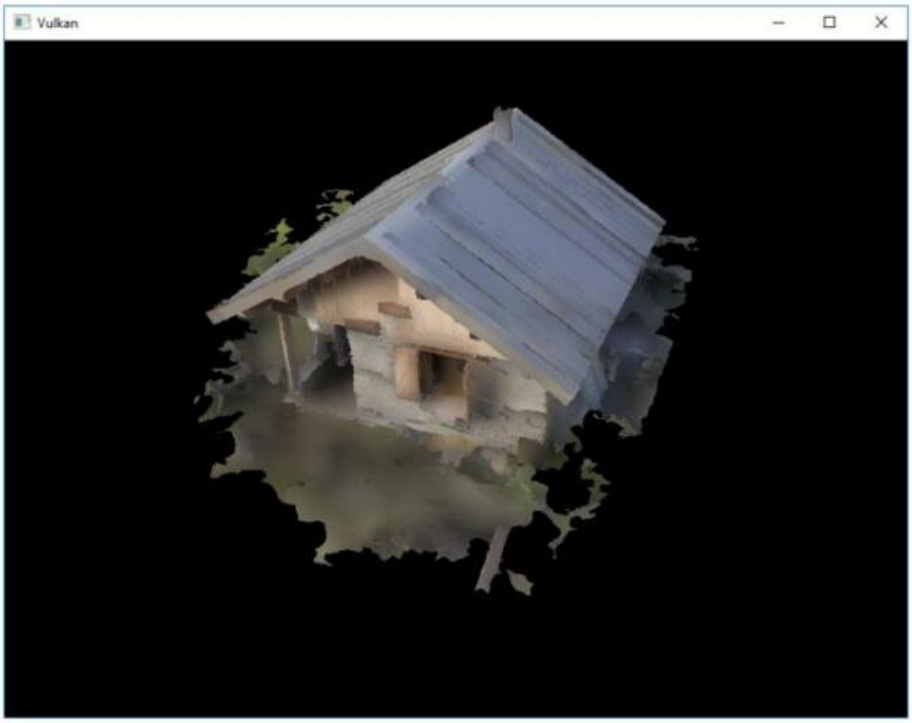

Figure 63: image

总结

至此，我们已经具有了不错的 Vulkan 使用基础。可以开始深入探索 Vulkan 的更多特性，比如：

- Push 常量
- 实例渲染
- 动态 uniform
- 分离图像和采样器描述符
- 管线缓存
- 多线程指令缓冲生成
- 多子流程
- 计算着色器

我们可以在现有的程序上扩展支持许多特性，比如 Blinn-Phong、后期处理、实时阴影。这些内容的实现方法可以在其它图形 API 的教程中找到，然后移植为 Vulkan 的 API 调用实现。

本章节代码：

C++：

https://vulkan-tutorial.com/code/28_mipmapping.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/26_shader_depth.vert

Fragment Shader：

https://vulkan-tutorial.com/code/26_shader_depth.frag

多重采样

介绍

现在我们的程序已经可以使用带有细化级别的纹理贴图，可以在一定程度上解决模型贴图的走样问题。但当绘制的几何图元的边接近水平或接近垂直时，还是会发现明显的锯齿现象。这一想象在我们最初绘制的矩形上看来十分明显：

这类现象也被我们叫做走样，造成它的原因是显示设备的分辨率有限，而几何图元本质上可以细分为无限的点，当构成几何图元的点粒度较大时，倾斜的边缘就比较容易出现明显的锯齿现象。有许多改善走样现象的技术，在本章节，我们介绍其中一种叫做多重采样的反走样技术 (MSAA)。

没有开启多重采样的情况下，像素的最终颜色由在像素中心的一次采样确定。如果一条线段穿过了一个像素，但没有覆盖采样点，那么这一像素的颜色仍然保持不变，否则使用线段颜色来填充像素。这就很容易导致锯齿现象的出现。

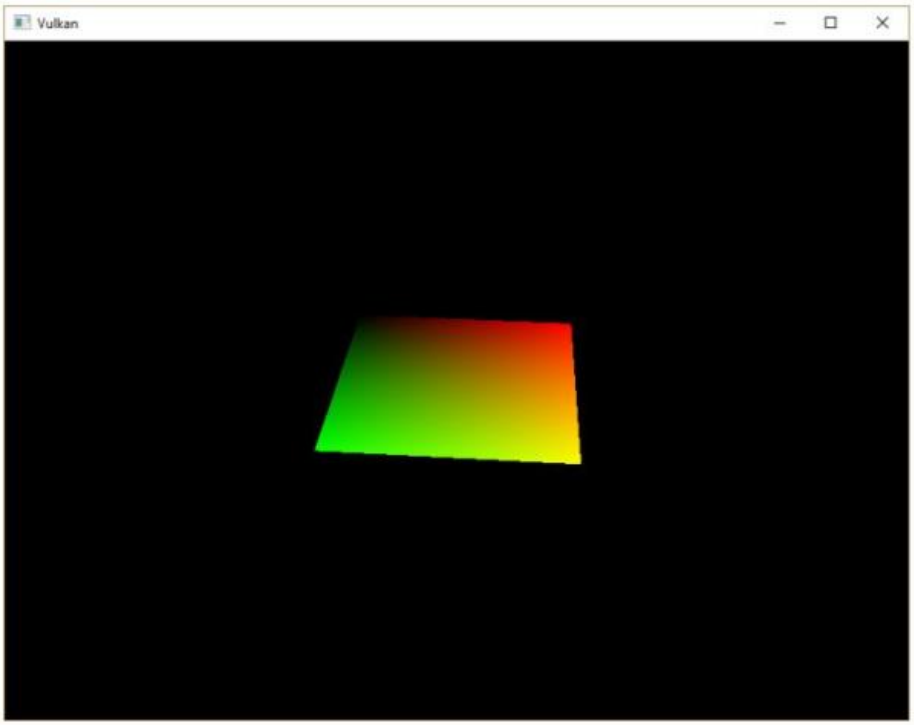

Figure 64: image

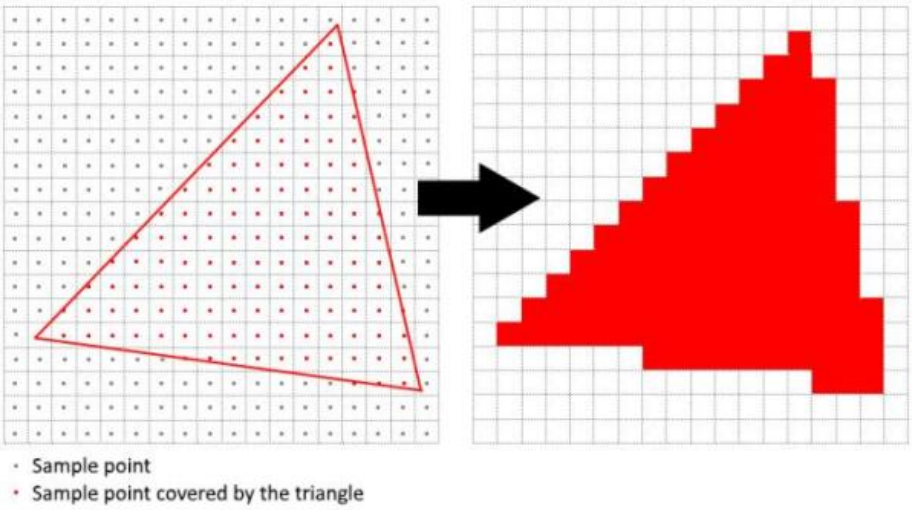

Figure 65: image

MSAA 技术对每个像素采样多次来确定最终的像素颜色。多次采样可以带来较好的渲染结果，但需要更多的计算量。

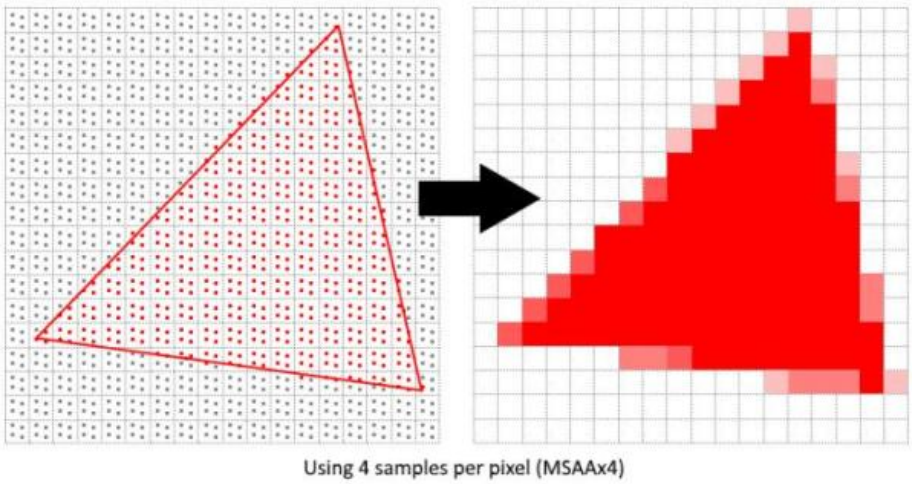

Figure 66: image

这里，我们采用最大可用的样本数实现多重采样。这一做法并非最佳实践，读者应该根据自己程序的实际需要确定多重采样的样本数。

获取可用样本数

我们首先需要确定硬件可以支持采样样本数。目前大多数现代 GPU 支持至少 8 个采样样本，我们添加一个类成员变量来存储涉笔的最大可用采样样本数：

```
...
VkSampleCountFlagBits msaaSamples = VK_SAMPLE_COUNT_1_BIT;
        ...
```

默认情况下，我们使用一个采样样本，这种情况下渲染的结果和不使用多重采样是一样的。最大可用采样个数可以通过调用 VkPhysicalDeviceProperties 函数来查询。由于我们还使用了一个深度缓冲，我们能使用的最大采样样本数也会小很多。添加一个叫做 getMaxUsableSampleCount 的函数查询可用采样数：

```
VkSampleCountFlagBits getMaxUsableSampleCount() {
    VkPhysicalDeviceProperties physicalDeviceProperties;
    vkGetPhysicalDeviceProperties(physicalDevice, &physicalDeviceProperties);

    VkSampleCountFlags counts = std::min(physicalDeviceProperties.limits.
    framebufferColorSampleCounts, physicalDeviceProperties. limits.framebufferDepthSampleCou
    if (counts & VK_SAMPLE_COUNT_64_BIT) { return VK_SAMPLE_COUNT_64_BIT; }
    if (counts & VK_SAMPLE_COUNT_32_BIT) { return VK_SAMPLE_COUNT_32_BIT; }
    if (counts & VK_SAMPLE_COUNT_16_BIT) { return VK_SAMPLE_COUNT_16_BIT; }
```

```
    if (counts & VK_SAMPLE_COUNT_8_BIT) { return VK_SAMPLE_COUNT_8_BIT; }
    if (counts & VK_SAMPLE_COUNT_4_BIT) { return VK_SAMPLE_COUNT_4_BIT; }
    if (counts & VK_SAMPLE_COUNT_2_BIT) { return VK_SAMPLE_COUNT_2_BIT; }

    return VK_SAMPLE_COUNT_1_BIT;
}
```

我们在选择使用的物理设备时调用 getMaxUsableSampleCount 函数来查询可用采样样本数。修改 pickPhysicalDevice 函数实现这一目的：

```
void pickPhysicalDevice() {
        ...
    for (const auto& device : devices) {
        if (isDeviceSuitable(device)) {
            physicalDevice = device;
            msaaSamples = getMaxUsableSampleCount();
            break;
        }
    }
        ...
}
```

设置渲染目标

多重采样是在一个离屏缓冲中进行的。这个离屏缓冲与我们之前进行渲染的一般图像对象略有不同。它需要对每个像素保存多个样本数据。多重采样缓冲的数据需要转换后，才能写入帧缓冲。和深度缓冲一样，我们只需要一个渲染目标用作多重采样缓冲即可，因为同时只会有一个绘制操作在进行。添加下面这些类成员变量：

```
...
VkImage colorImage;
VkDeviceMemory colorImageMemory;
VkImageView colorImageView;
        ...
```

我们需要修改 createImage 函数，添加 numSamples 参数用来指定采样个数：

```
void createImage(uint32_t width, uint32_t height, uint32_t mipLevels, VkSampleCountFlagBits
        ...
imageInfo.samples = numSamples;
        ...
```

现在，我们需要更新所有之前对 createImage 函数的调用，保证调用使用的参数正确：

```
createImage(swapChainExtent.width, swapChainExtent.height, 1, VK_SAMPLE_COUNT_1_BIT, depthFo
        ...
createImage(texWidth, texHeight, mipLevels, VK_SAMPLE_COUNT_1_BIT, VK_FORMAT_R8G8B8A8_UNORM,
VK_IMAGE_USAGE_SAMPLED_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageM
```

创建多重采样颜色缓冲。添加一个叫做 createColorResources 的函数，这里我们使用 msaaSamples 作为图像的采样个数。我们将图像的细化级别数目设置为 1。Vulkan 规范要求对于采样个数大于 1 的图像只能使用 1 个细化级别。实际上，对于多重采样颜色缓冲来说也只需要一个细化级别来存储原始图像数据，毕竟它不会被作为纹理贴图进行渲染。

```
void createColorResources() {
    VkFormat colorFormat = swapChainImageFormat;

    createImage(swapChainExtent.width, swapChainExtent.height, 1, msaaSamples, colorFormat,
    colorImageView = createImageView(colorImage, colorFormat, VK_IMAGE_ASPECT_COLOR_BIT, 1);

    transitionImageLayout(colorImage, colorFormat, VK_IMAGE_LAYOUT_UNDEFINED,VK_IMAGE_LAYOUT
}
```

我们在创建深度缓冲资源的函数调用之前调用 createColorResources 函数创建多重采样缓冲：

```
void initVulkan() {
        ...
    createColorResources();
    createDepthResources();
        ...
}
```

读者可能已经注意到，我们新创建的用于多重采样的缓冲图像使用了一个新的布局变换过程：从 VK_IMAGE_LAYOUT_UNDEFINED 到 VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL。我们需要修改 transitionImageLayout 函数来处理它：

```
void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageL
        ...
    else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_COLOR_AT
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACH
        sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
        destinationStage = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
    }
    else {
        throw std::invalid_argument("unsupported layout transition!");
    }
        ...
}
```

现在我们需要修改 createDepthResources 函数，更新深度缓冲使用的采样个数：

```
void createDepthResources() {
        ...
    createImage(swapChainExtent.width, swapChainExtent.height, 1, msaaSamples, depthFormat,
```

```
        ...
}
```

最后，不要忘记在清除交换链时，清除我们创建的与多重采样缓冲相关的资源：

```
void cleanupSwapChain() {
    vkDestroyImageView(device, colorImageView, nullptr);
    vkDestroyImage(device, colorImage, nullptr);
    vkFreeMemory(device, colorImageMemory, nullptr);
        ...
}
```

修改 recreateSwapChain 函数，在窗口大小变化时重建多重采样缓冲：

```
void recreateSwapChain() {
        ...
    createGraphicsPipeline();
    createColorResources();
    createDepthResources();
        ...
}
```

至此，我们就完成了多重采样缓冲的设置，可以在图形管线中使用它了。

添加附着

修改 createRenderPass 函数，更新对颜色附着和深度附着的设置：

```
void createRenderPass() {
        ...
    colorAttachment.samples = msaaSamples;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
        ...
    depthAttachment.samples = msaaSamples;
        ...
```

读者可能已经注意到,我们将 finalLayout 从 VK_IMAGE_LAYOUT_PRESENT_SRC_KHR 布局改为 VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL。这样做是因为多重采样缓冲不能直接用于呈现操作。我们需要先把它转换为普通的图形形式。对于深度缓冲，不需要进行呈现操作，也就不需要对其进行转换。添加一个新的颜色附着用于转换多重采样缓冲数据：

```
...
VkAttachmentDescription colorAttachmentResolve = {};
colorAttachmentResolve.format = swapChainImageFormat;
colorAttachmentResolve.samples = VK_SAMPLE_COUNT_1_BIT;
colorAttachmentResolve.loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachmentResolve.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
colorAttachmentResolve.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachmentResolve.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

```
colorAttachmentResolve.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachmentResolve.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
        ...
```

设置渲染流程引用这一新添加的附着。使用一个 VkAttachmentReference 结构体引用我们刚刚创建的颜色附着：

```
...
VkAttachmentReference colorAttachmentResolveRef = {};
colorAttachmentResolveRef.attachment = 2;
colorAttachmentResolveRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
        ...
```

设置 pResolveAttachments 成员变量指向刚刚添加的 VkAttachmentReference 结构体。

```
...
subpass.pResolveAttachments = &colorAttachmentResolveRef;
        ...
```

更新渲染流程结构体信息，包含新得颜色附着：

```
...
std::array<VkAttachmentDescription, 3> attachments = {colorAttachment, depthAttachment, colo
        ...
```

修改 createFrameBuffers 函数，添加新的图像视图到 attachments 中：

```
void createFrameBuffers() {
        ...
    std::array<VkImageView, 3> attachments = { colorImageView, depthImageView, swapChainImag
        ...
}
```

最后，修改 createGraphicsPipeline 函数，设置图像管线使用的采样样本数：

```
void createGraphicsPipeline() {
        ...
    multisampling.rasterizationSamples = msaaSamples;
        ...
}
```

现在编译运行程序就可以看到下面的画面：

我们与之前不使用多重采样的结果进行对比：

在模型的边缘处可以看到明显的不同：

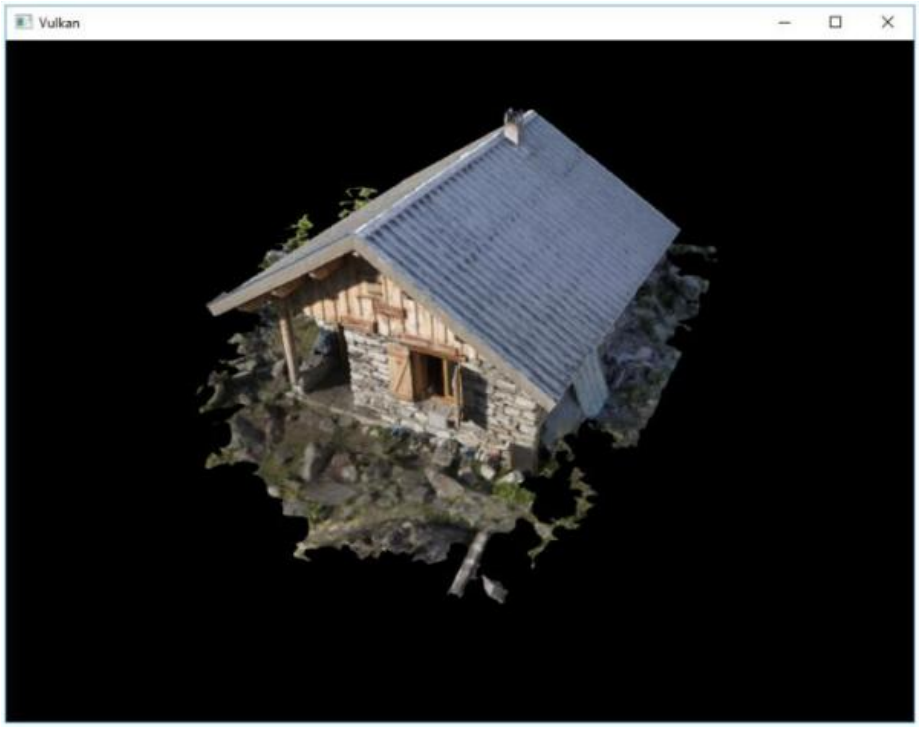

Figure 67: image

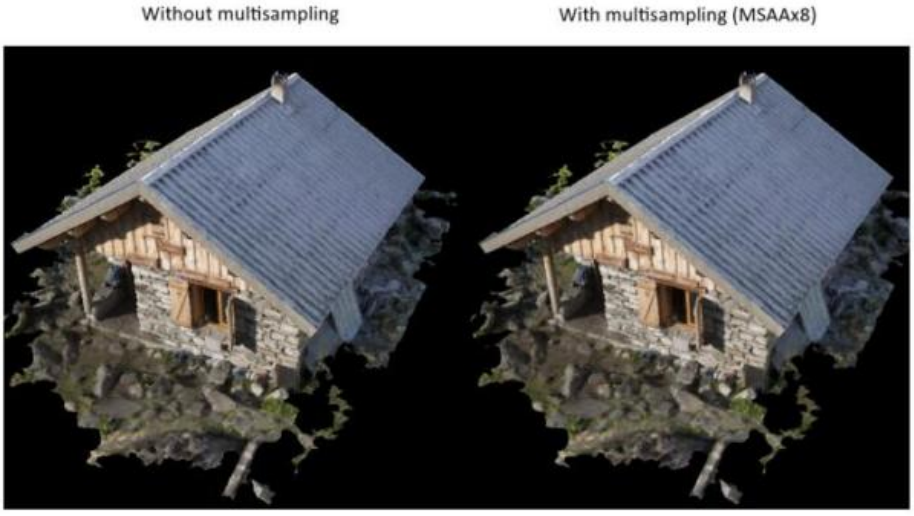

Figure 68: image

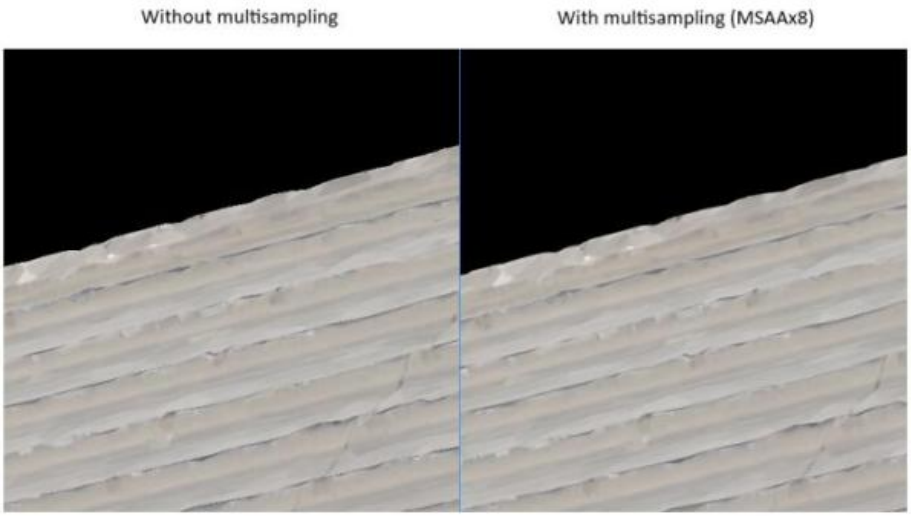

Figure 69: image

提升质量

我们目前实现的多重采样还有很大的提升空间。比如，由着色器造成的走样我们还没有处理。MSAA 只对几何图元的边缘进行平滑处理，但不会对图元的内部进行处理。这就造成几何图元上的贴图仍会出现走样现象。可以通过开启采样着色来解决这一问题，但这样做会造成一定的性能损失：

```
void createLogicalDevice() {
        ...
    deviceFeatures.sampleRateShading = VK_TRUE;
    // enable sample shading feature for the device
        ...
}

void createGraphicsPipeline() {
        ...
    multisampling.sampleShadingEnable = VK_TRUE; // enable sample shading in the pipeline
    multisampling.minSampleShading = .2f;
    // min fraction for sample shading; closer to one is smoother
        ...
}
```

在本例，我们关闭了采样着色。但在某些情况下，开启它后渲染质量的改善会很明显：

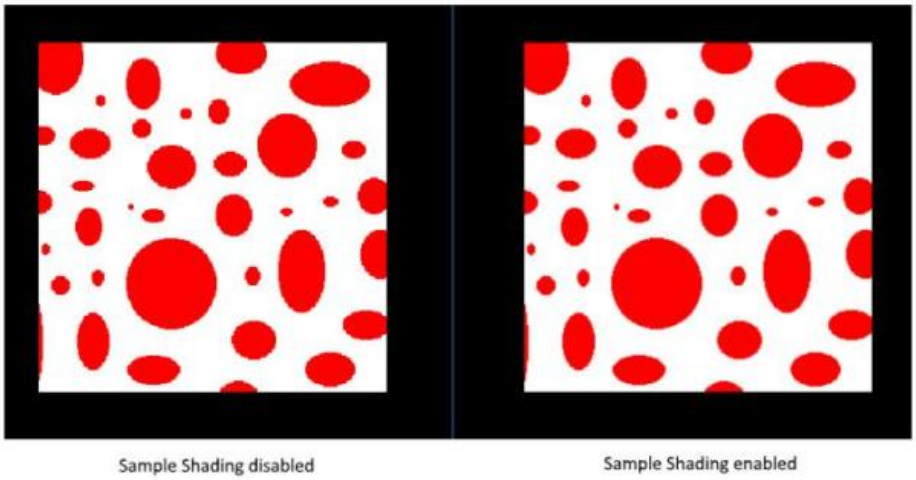

Figure 70: image

总结

至此，我们已经具有了不错的 Vulkan 使用基础。可以开始深入探索 Vulkan 的更多特性，比如：

- Push 常量
- 实例渲染
- 动态 uniform
- 分离图像和采样器描述符
- 管线缓存
- 多线程指令缓冲生成
- 多子流程
- 计算着色器

我们可以在现有的程序上扩展支持许多特性，比如 Blinn-Phong、后期处理、实时阴影。这些内容的实现方法可以在其它图形 API 的教程中找到，然后移植为 Vulkan 的 API 调用实现。

本章节代码：

C++：

https://vulkan-tutorial.com/code/29_multisampling.cpp

Vertex Shader：

https://vulkan-tutorial.com/code/26_shader_depth.vert

Fragment Shader：

https://vulkan-tutorial.com/code/26_shader_depth.frag

FAQ

本章节主要介绍使用 Vulkan 过程中的一些常见问题的解决方法：

- 校验层返回访问冲突错误：确保没有运行 MSI Afterburner / RivaTuner Statistics Server。这两个程序和 Vulkan 存在兼容问题。
- 校验层没有返回信息：默认情况下，校验层会将信息打印在终端上，所以为了看到校验层输出的信息，需要保证终端不被关闭。对于 Visual Studio 来说，使用 Ctrl+F5 编译运行程序，对于 Linux，使用终端运行程序，即使程序意外退出，也可以保证看到校验层输出到终端的信息。如果这样做后，仍然看不到校验层输出的信息，可以检查校验层是否真的被开启，以及 Vulkan SDK 是否被正确安装。
- SteamOverlayVulkanLayer64.dll 对 vkCreateSwapchainKHR 函数的调用触发了一个错误：这可能是由 beta 版 Steam 客户端造成的，可以尝试下面这些方法来解决这一问题：1. 不使用 beta 版 Steam 客户端 2. 设置 DISABLE_VK_LAYER_VALVE_steam_overlay_1 环境变量的值为 1：

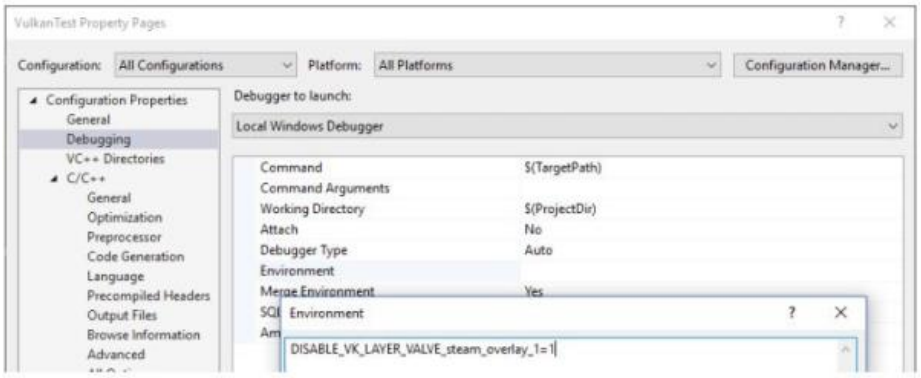

Figure 71: image

3. 删除注册表中的 HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\Vulkan\ImplicitLayers 条目