第 1 章　3D 程序分析方法

3D 程序通常由两部分组成：非着色器部分和着色器(shader)部分。非着色器部分用 C/C++/JavaScript 等编写，运行在 CPU 上，负责给 GPU 传送数据，以及设置 GPU 的参数和状态。非着色器部分也称作 CPU 部分。着色器部分用着色语言(shading language)描述，运行在 GPU 上，也称作 GPU 部分。着色语言有多种实现：基于 OpenGL 的 OpenGL 着色语言，简称 GLSL(OpenGL/OpenGL ES/WebGL/Vulkan 都支持 GLSL)；基于微软公司 DirectX 的高级着色语言，简称 HLSL；苹果公司的 Metal 着色语言，简称 MSL。本书讨论 WebGL/Vulkan 使用的 GLSL。GLSL 按照功能分为多种。图形相关的有：处理 3D 模型的顶点着色器(vertex shader)和几何着色器(geometry shader)、处理光照和纹理的片元着色器(fragment shader)。计算相关的有计算着色器(compute shader)，计算着色器目前广泛应用于深度学习。最新的 Vulkan 标准还增加了任务着色器(task shader)、网格着色器(mesh shader)，以及用于光线追踪的多种着色器等。

3D 程序的非着色器部分，如果从 CPU 的角度看，它是普通的程序，可以用程序语言 C/C++/JavaScript 等来描述。但是如果从 GPU 的角度看，GPU 自身就是一个实现了完整渲染流水线的状态机，非着色器部分只是描述了 GPU 的输入输出，以及一些 GPU 参数状态的调整控制命令。所以如果仅从算法和流程的复杂度来看，GPU 核心的流程和算法，一部分在 GPU 流水线本身，一部分在用户提供的顶点着色器和片元着色器里，而不是非着色器部分。

从这一点来说，分析一个 3D 程序，重点应该分析流水线(本书讨论的投影和纹理映射就属于流水线)和着色器(模型的处理、光照的处理)，而不是非着色器部分。基于 GL(本书用 GL 来统称 OpenGL/OpenGL ES/WebGL)的程序，其非着色器部分调用的 GPU 相关接口数量少，分析起来比较容易。但是 Vulkan 的引入，将原来封装在 GL 底层的一些功能暴露了出来，同时增加了多线程的支持，因而增加了大量的接口。Vulkan 程序的非着色器部分，也比相应的 GL 实现要复杂很多。针对 Vulkan 的情况，流水线和着色器依然是理解整个 3D 程序的重点。不过非着色器部分的代码变得复杂了，也需要做些分析。换句话说，用 GL 编程的时候，理解 GPU 的行为就可以了。Vulkan 则要求开发者在理解 GPU 的同时增加一些对 CPU 编程的理解。

本书重点讨论 GPU 的流水线和着色器部分。但是针对某些复杂的 Vulkan 场景的 CPU 部分(非着色器部分)，例如涉及多次渲染的时候，CPU 部分的结构会影响到对流水线和着色器的理解，因而也会介绍其结构。本书分析这些示例结构的时候不使用流程图、序列图、类图等常用的分析方法，而是使用了通信专业的输入输出分析方法，即给出程序的输入数据→数据处理过程→输出数据框架，以帮助读者理解 3D 程序的 CPU 部分。

为什么选择输入数据输出数据来分析 Vulkan 程序的 CPU 部分(当然也可以用于 GL 的分析)？如果是刚接触 3D 编程,理解 GL、Vulkan 是有一定难度的。如果有一定的 GL 经验,希望通过 GL 的经验分析 Vulkan 的 CPU 部分的源代码,也是有些挑战的。这些挑战来自以下两方面。

(1) 与 GL 实现的接口不一样,Vulkan 提供了更多底层资源操作的接口,同时实现了对多线程的支持。因此在接口数量上比 GL 要多出很多。哪怕是绘制一个最简单的三角形,代码也比同样绘制三角形的 GL 程序多很多。

(2) 另一方面,即使有了初步的 GL 基础,但是 GL 到 Vulkan 的接口很难一一对应起来,虽然两者的主要功能是一样的。

综上两点,从接口层面去理解 Vulkan CPU 部分的源码并不直观。然而,虽然 GL、Vulkan 接口差别很大,但是两者的输入输出都是类似的：输入是顶点(以及法线光线等)、MVP 矩阵、纹理及坐标；输出则是帧缓冲(或者绑定到帧缓冲的纹理)。数据的处理都是通过绘图渲染来完成的。复杂一些的过程,譬如延迟渲染或者阴影的计算,由于需要两次渲染过程,有些数据是第一个过程的输出,同时作为第二个过程的输入。如果从输入数据、输出数据以及数据的处理过程来理解分析 Vulkan,由于将大量的 Vulkan 接口按照输入、处理过程、输出分为三类,这种分析方法可以简化 Vulkan 的分析,同时也是一种适用于其他 3D 编程接口例如 GL 的分析方法。具体的分析模型如图 1-1 所示。图中白色方框是输入,灰色方框是输出,圆角框是数据处理过程,灰色虚线方框表示这个模块在作为一个过程的输入的同时还作为另一个过程的输出。

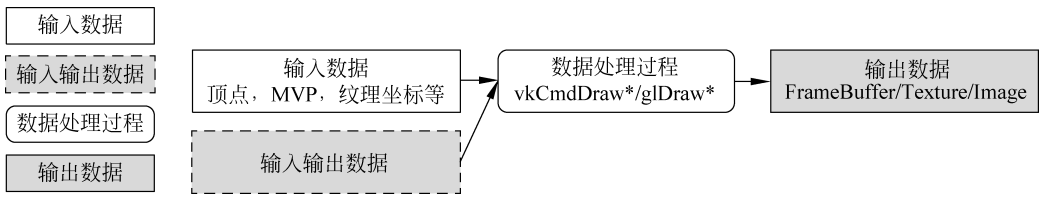

图 1-1　3D 程序的输入输出模型

对于 Vulkan,纹理资源是通过描述符(descriptor)来表示的。如果将着色器也当作一种输入数据,则通常将着色器作为 VkPipeline 的一部分传递给 GPU。在后文讲解 Vulkan 例子源码结构的时候,会在结构图里面标注描述符和着色器。至于 WebGL 的例子,因为结构本身就很简单,就没有对其结构做具体分析。

如果 Vulkan 程序包括多次绘图或者计算(vkCmdDraw * 发起绘图,vkCmdDispatch 发起计算),而且两个绘图或者计算过程之间还有资源的共享,例如绘图过程 1 的输出被当作绘图过程 2 的输入,那么会在结构图里面加一个箭头示意这里会发生资源共享,所以要留意资源读写时的同步与互斥。

本章将从输入输出的角度来分析 3D 程序的非着色器部分。读者会发现,如果将输入输出模型应用到本书的示例,理解 WebGL,以及更加复杂的 Vulkan 示例,就会简单很多。

本章分析输入顶点数据和纹理时会使用不同的示例。分析顶点数据和 MVP 数据使

用的是输出一个矩形的两个例子：WebGL/projection/projection_perspective_quad. html 和 Vulkan/examples/projection_perspective_quad。分析纹理使用的两个例子的输出都是纹理图片：WebGL/texturemapping/projection_perspective_texture_mapping. html 和 Vulkan/examples/projection_perspective_texture。当然，输出纹理的例子仍然需要输入顶点和 MVP。

WebGL 1.0 基于 OpenGL ES 2.0，WebGL 2.0 基于 OpenGL ES 3.0。除了 WebGL 之外，基于 Vulkan、Metal、Direct3D 的下一代 Web 3D 编程标准 WebGPU 目前正在讨论之中。本书的例子以 Vulkan 为主，部分章节还同时提供了 WebGL 1.0 的例子，主要是为了对比理解 3D 模型的通用性。选择 WebGL 的原因是，做简单的模型验证非常方便。但如果是要设计更加复杂的高性能 3D 程序，读者需要自己去调查了解 WebGL 是否满足性能要求。选择 Vulkan 而不是更接近 WebGL 的 OpenGL，则是因为 Vulkan 接口更复杂，能够很好地体现基于输入输出的分析方法优势。同时，Vulkan 是最新的标准，了解其应用很有必要。

1.1　输入顶点数据

顶点数据通常在 CPU 端描述，然后通过缓冲区传递给 GPU。最后在 GPU 流水线开始的时候，绑定相关的缓冲区，流水线就可以在不同的顶点着色器里面访问顶点数据。

1.1.1　描述顶点

WebGL 和 Vulkan 通常以三角形为单位进行渲染，所以四边形其实都是由两个三角形组成的。WebGL 的例子，仅指定了顶点的位置，颜色是在其他地方指定的，但是可以在指定顶点位置的同时指定顶点颜色。

WebGL 示例在代码里面指定四边形四个顶点的颜色，如程序清单 1-1 所示。

程序清单 1-1　WebGL 的顶点数据

```
// WebGL/projection/projection_perspective_quad.html
var scale = 1.0;
var zEye = -0.5;
var leftAtAnyZ = left * zEye/-near;
var rightAtAnyZ = right * zEye/-near;
var bottomAtAnyZ = bottom * zEye/-near;
var topAtAnyZ = topp * zEye/-near;
vertices = [
 leftAtAnyZ * scale, bottomAtAnyZ * scale, zEye,
 rightAtAnyZ * scale, bottomAtAnyZ * scale, zEye,
 rightAtAnyZ * scale, topAtAnyZ * scale, zEye,
 leftAtAnyZ * scale, topAtAnyZ * scale, zEye,
];
```

Vulkan 的例子，准备的数据和 WebGL 的类似，但是同时指定了顶点的位置和颜色，如程序清单 1-2 所示。

程序清单 1-2　Vulkan 顶点和顶点颜色数据

```
// Vulkan/examples/projection_perspective_quad/projection_perspective_quad.cpp
std::vector<Vertex> vertexBuffer =
{
 { {leftAtAnyZ, bottomAtAnyZ, zEye}, { 1.0f, 0.0f, 0.0f } },
 { {rightAtAnyZ, bottomAtAnyZ, zEye}, { 0.0f, 1.0f, 0.0f } },
 { {rightAtAnyZ, topAtAnyZ, zEye}, { 0.0f, 0.0f, 1.0f } },
 { {leftAtAnyZ, topAtAnyZ, zEye}, { 0.0f, 1.0f, 0.0f } }
};
```

1.1.2　传递顶点

所谓传递顶点数据，就是将 CPU 创建的顶点数据传递到 GPU 可见的缓冲区。

对于 WebGL，顶点数据通过 ARRAY_BUFFER 上传给 GPU，如程序清单 1-3 所示。

程序清单 1-3　WebGL 上传顶点数据

```
// WebGL/projection/projection_perspective_texture.html
cubeVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
// 变量 vertices 里面就是顶点数据
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

对于 Vulkan 而言，顶点数据是存储到 VkBuffer 中的。但是 VkBuffer 本身没有存储空间，需要通过 VkDeviceMemory 来存储数据。为 VkDeviceMemory 申请好存储空间之后，将程序清单 1-2 Vulkan 顶点和顶点颜色数据用 memcpy 复制到 VkDeviceMemory 里面去。本书大部分例子 VkDeviceMemory 申请的内存是 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT 类型，这类内存要先调用 vkMapMemory，CPU 才能对它进行读写。

具体 VulkanDevice::CreateBuffer 的实现，如程序清单 1-4 所示。

程序清单 1-4　创建并复制数据到 VkBuffer

```
// Vulkan/base/VulkanDevice.hpp
VkResult createBuffer(VkBufferUsageFlags usageFlags,
          VkMemoryPropertyFlags memoryPropertyFlags,
          VkDeviceSize size,
          VkBuffer* buffer,
          VkDeviceMemory* memory,
          void* data = nullptr) {
 // 调用 vkCreateBuffer 创建 VkBuffer
```

```
 VK_CHECK_RESULT(
   vkCreateBuffer(logicalDevice, &bufferCreateInfo, nullptr, buffer));
 // 为 VkDeviceMemory 申请存储空间
 VK_CHECK_RESULT(
   vkAllocateMemory(logicalDevice, &memAlloc, nullptr, memory));
 if (data != nullptr) {
  void * mapped;
  // VkDeviceMemory 经过 vkMapMemory 之后,CPU 可以直接对其进行读写了
  VK_CHECK_RESULT(vkMapMemory(logicalDevice, * memory, 0, size, 0, &mapped));
  // 复制数据到 VkDeviceMemory
  memcpy(mapped, data, size);
  // 结束后 vkUnmapMemory
  vkUnmapMemory(logicalDevice, * memory);
 }
 // 数据写到 VkDeviceMemory,要将 VkDeviceMemory 和 VkBuffer 绑定起来
 VK_CHECK_RESULT(vkBindBufferMemory(logicalDevice, * buffer, * memory, 0));
 return VK_SUCCESS;
}
```

WebGL 和 Vulkan 数据传递接口的主要差别是，Vulkan 提供了一个更底层的存储空间管理对象 VkDeviceMemory，WebGL 则封装了这部分细节。

1.1.3　绑定顶点缓冲区

绑定顶点缓冲区是一种 GPU 命令，所有 GPU 命令都是通过绘图命令或者提交命令提交给 GPU 的。对于 WebGL，绘图和提交命令都是 gl.Draw*（包括 gl.drawArrays 和 gl.drawElements）。对于 Vulkan，绘图命令是 vkCmdDraw*（包括 vkCmdDraw 和 vkCmdDrawIndexed），提交命令是 vkQueueSubmit。注意：CPU 里面调用 gl.Draw* 或者 vkCmdDraw* 等，仅仅是向 GPU 描述 CPU 的绘图意图，但并不等于 GPU 会立即去解释执行这些绘图命令，通常将 CPU 调用 gl.Draw* 和 vkCmdDraw* 的过程称为录制（record）GPU 命令的过程。

WebGL 的绑定如程序清单 1-5 所示。

程序清单 1-5　WebGL 绑定数据缓冲区

```
// WebGL/projection/projection_perspective_texture.html.
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
// vertexPositionAttribute 对应到顶点着色器里面的 attribute vec3 aVertexPosition;
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,cubeVertexPositionBuffer.itemSize,
  gl.FLOAT, false, 0, 0);
```

Vulkan 的绑定必须发生在 vkBeginCommandBuffer 和 vkEndCommandBuffer 之间，如程序清单 1-6 所示。

程序清单 1-6　Vulkan 绑定顶点数据相关的缓冲区

```
vkCmdBindVertexBuffers(drawCmdBuffers[i], VERTEX_BUFFER_BIND_ID, 1,
  &vertexBuffer.buffer, offsets);
```

1.2　输入 MVP 数据

MVP 数据分别是模型(model)、视图(view)、投影(projection)矩阵。

WebGL 可通过第三方库 glMatrix[①] 创建变换矩阵，接口是 mat4.create()。glMatrix 提供了丰富的接口，用来实现各种变换。例如 mat4.translate、mat4.rotate、mat4.scale 可以用来实现模型视图矩阵的位移、旋转、缩放变换。投影矩阵也可以通过 mat4.create 来创建，但更简便的方式则是通过 mat4.perspective 创建透视投影矩阵，mat4.ortho 创建正交投影矩阵。矩阵创建好了之后，可以通过 gl.uniformMatrix4fv 将矩阵数据传递给着色器进行下一步的处理。

Vulkan 可以使用封装了模型视图变换、透视投影、正交投影等的第三方库 glm[②]。其中，glm::mat4 用于创建模型视图矩阵，glm::translate、glm::rotate、glm::scale 则用来实现具体的模型视图变换。创建透视投影矩阵的 glm::perspective 和创建正交投影矩阵的 glm::ortho 则将创建矩阵和变换的过程封装到了一个接口。

Vulkan 的 MVP 数据可以通过 vkCmdPushConstants 在录制 GPU 命令的时候直接传递。Push Constants 是 Vulkan 提出的一种快速地向 GPU 提交小规模数据的方法。也可以像顶点索引数据一样，将 MVP 数据传递给 VkBuffer。不过和顶点索引数据不同的是，顶点索引数据传递给 VkBuffer 之后，可以直接在录制的时候通过 vkCmdBindVertexBuffers 来绑定。VkBuffer 还可以用来存储其他数据，例如 MVP 数据。但是除了顶点和索引数据之外，存储其他数据的 VkBuffer 需要先生成一个描述符，例如 VkDescriptorBufferInfo，并通过 vkUpdateDescriptorSets 将这个描述符追加到 VkDescriptorSet 里面。最后还是在录制的时候，通过 vkCmdBindDescriptorSets 告诉 GPU 本次绘图过程会用到这个缓冲区。

所以对于存储到 VkBuffer 的 MVP 数据，其使用过程分为以下三步。

(1) 复制数据到 VkBuffer。

(2) 为 VkBuffer 创建一个 VkDescriptorBufferInfo，通过 vkUpdateDescriptorSets 将其追加到 VkDescriptorSet。

(3) 在录制 GPU 命令的过程中，调用 vkCmdBindDescriptorSets 来告知 GPU 本次将使用的资源。

---

① glMatrix 项目，http://glmatrix.net/。

② OpenGL 数学库，https://glm.g-truc.net/。

1.3　输 入 纹 理

纹理可以用来存储从存储设备或者网络读取的图片，也可以绑定到输出缓冲区之后，用作 3D 过程的输出。这里讨论作为输入的图片纹理。用户将图片纹理传递给 GPU 可见的缓冲区，GPU 则通过采样器从这些缓冲区读取纹理的数据。

纹理包含图像数据，采样器包含影响纹理的采样过程的状态和控制信息，例如纹理的滤波模式(filter mode)、纹理坐标的环绕模式(wrap mode)等都受采样器的影响。纹理和采样器在不同的 GPU、不同的 GL 版本，以及 Vulkan 上的实现可能是有差别的。

在 GPU 里面，采样器的状态和纹理数据是分开的，两者不相关。同一个纹理，在一种情形下可以通过 VK_FILTER_LINEAR(GL_LINEAR)来采样，也可以在另一种情形下使用 VK_FILTER_NEAREST(GL_NEAREST)来采样。

OpenGL 3.2 之前是不区分纹理和采样器的，WebGL 也是如此。在这些版本的 GL 实现里面，纹理对象同时包含采样器的状态和控制信息。所以如果使用这些版本的 GL 创建一个纹理，调用 glGenTextures 就可以了。要配置纹理(其实是采样器)相关的滤波模式和环绕模式则需要调用 glTexParameter * 系列函数。OpenGL 3.2 开始引入 glGenSamplers 以解耦合纹理和采样器。Vulkan 里面，vkCreateImage 用于纹理的创建，vkCreateSampler 用于创建采样器，滤波模式和环绕模式是针对采样器的。本节分析的输入纹理将不包含采样器部分。

1.3.1　创建并传递纹理

WebGL 使用纹理是异步的，如程序清单 1-7 所示。

程序清单 1-7　WebGL 纹理创建

```
// WebGL/texturemapping/projection_perspective_texture_mapping.html
neheTexture = gl.createTexture();
neheTexture.image = new Image();
neheTexture.image.onload = function () {
 handleLoadedTexture(neheTexture)
}
neheTexture.image.src = "/resources/gorilla.png";
```

纹理创建好，并且图片数据加载完毕后，回调 handleLoadedTexture 来进行纹理的绑定，如程序清单 1-8 所示。

程序清单 1-8　WebGL 纹理绑定

```
// WebGL/texturemapping/projection_perspective_texture_mapping.html
gl.bindTexture(gl.TEXTURE_2D, texture);
```

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.bindTexture(gl.TEXTURE_2D, null);
```

Vulkan 对纹理提供了多种抽象，如 VkImage、VkImageView、VkSampler 等。一种直接使用纹理的方式是：从磁盘读取图片数据，并存储到 GPU 可见的存储空间，即复制到 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT（以下简称 HOST_VISIBLE）类型的存储空间中，然后 GPU 直接对这块存储空间进行采样。

由于硬件和驱动设计的不同，HOST_VISIBLE 类型的存储空间可能位于 CPU 的内存之中，因而对于独立显卡而言，HOST_VISIBLE 不是最优的直接给 GPU 输入纹理数据的方式。所以本书 Vulkan 的例子还提供了 staging 缓冲区的纹理加载方式。

通过 Staging 缓冲区加载资源是为了屏蔽 CPU 和 GPU 访问内存和显存的差异。这个差异来自两个方面：

（1）集成显卡的显存和内存是共享的，独立显卡使用自己的独立显存。

（2）内存通过内存控制器直接接入 SoC（System on Chip）环形总线；独立显卡和独立显存则通过 PCIe 总线间接接入环形总线。

针对集成显卡和独立显卡，这些差异导致的结果是：

（1）对于集成显卡，CPU、集成显卡从内存读取数据和显存读取数据都很快。

（2）对于独立显卡，独立显卡访问独立显存很快，访问内存很慢。CPU 访问内存很快，访问独立显存很慢。

总线、CPU、显卡、内存和显存之间的关系如图 1-2 所示（DRAM 是内存，VRAM 是独立显存，Intel HD GPU 是集成显卡，NV/AMD GPU 是独立显卡）。

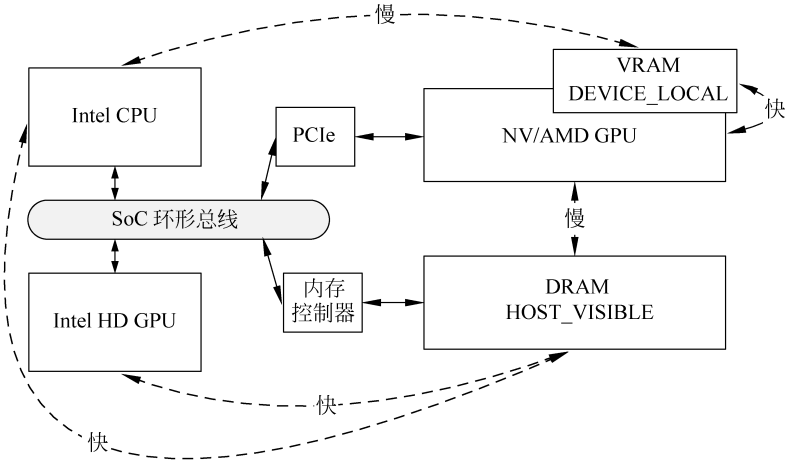

图 1-2　计算机系统的总线和存储结构

无论是集成显卡还是独立显卡，都可以直接访问内存的纹理数据。所以这里存在两种数据访问方式。

（1）非 Staging 访问：GPU 直接访问内存的纹理数据。集成显卡可以使用这个方式，独立显卡要避免 GPU 直接访问内存。

（2）Staging 访问：先将内存的纹理数据复制到显存，GPU 直接从显存里面读取数据。独立显卡应该使用这个方式。

对于 Vulkan，Staging 访问的方法是：先将纹理数据读入一个 HOST_VISIBLE 的内存空间，然后将这个 HOST_VISIBLE 的内存复制到 DEVICE_LOCAL 的显存中，如图 1-3 和图 1-4 所示。

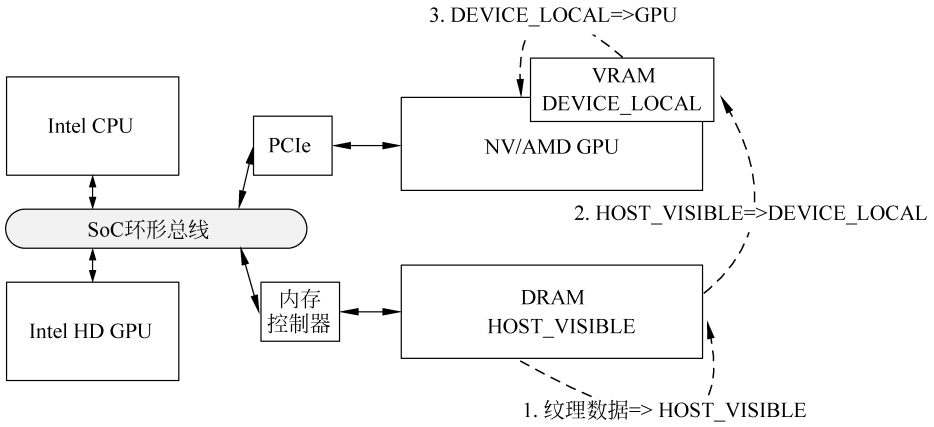

图 1-3　独立显卡的 Staging 方式

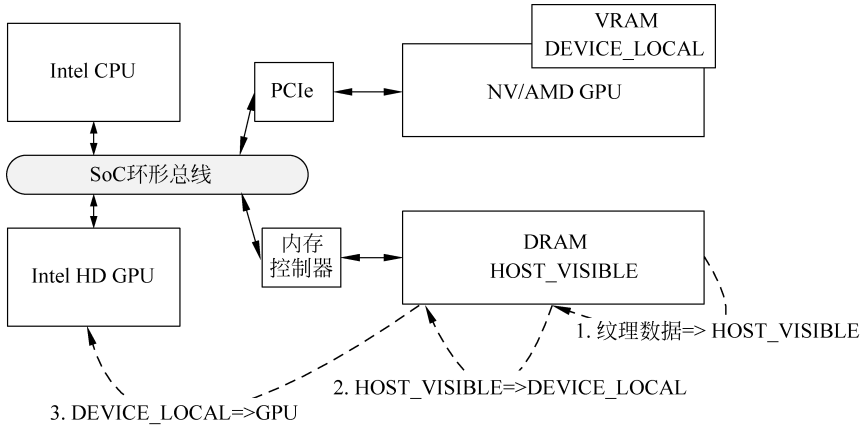

图 1-4　集成显卡的 Staging 方式

相应地，非 Staging 访问将纹理数据读入一个 HOST_VISIBLE 的内存空间，剩下的交给 GPU 完成。

这两种方式都实现在同一个 loadTexture 接口中。如图 1-5 所示为 Staging 缓冲器优化的纹理加载方式。步骤如下。

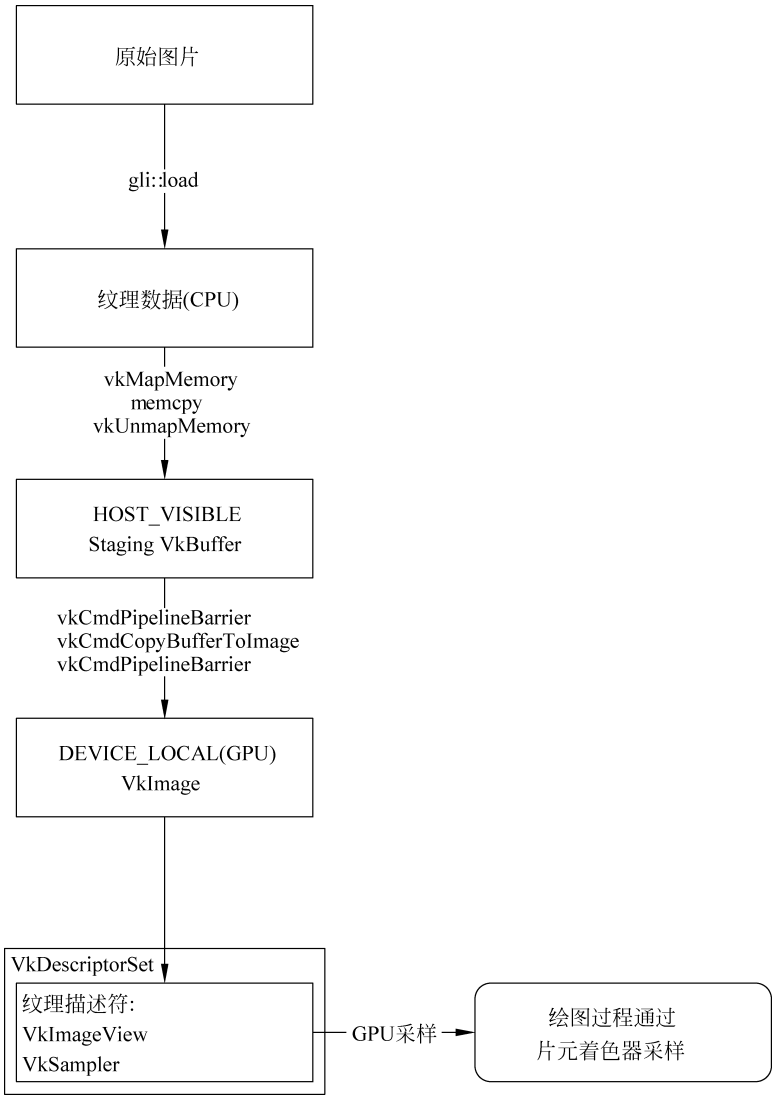

图 1-5　Staging 缓冲器优化的纹理加载方式

(1) 通过第三方库 gli::load 将 textures/gorilla.ktx 加载到 CPU 内存中,如程序清单 1-9 所示。

程序清单 1-9　从硬盘读取纹理

```
gli::texture2d tex2D(gli::load(filename));
```

(2) 创建一个 VkBuffer 以及相应的 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT 类型的 VkDeviceMemory,这个 VkDeviceMemory 通过映射(map)后 CPU 可见。于是可

以将程序清单 1-9 读进内存的 CPU 纹理数据复制到 VkDeviceMemory。这个过程和传递顶点数据是一样的。

(3) VkBuffer 里面的纹理数据无法给 GPU 直接读取采样,需要通过 vkCmdCopyBufferToImage 复制给一个 VkImage。该 VkImage 申请的存储空间类型是 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,位于 GPU 显存里面。

(4) GPU 读取数据的时候,需要 VkImageView/VkSampler 等辅助对象才能从 VkImage 里面读取数据。

1.3.2　绑定纹理

WebGL 通过程序清单 1-10 来绑定纹理。

程序清单 1-10　WebGL 绑定纹理

```
// WebGL/texturemapping/projection_perspective_texture_mapping.html
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, cubeVertexPositionBuffer.itemSize,
  gl.FLOAT, false, 0, 0);
```

Vulkan 并不直接绑定纹理。每个纹理会创建相应的描述符 VkDescriptorBufferInfo,并追加到 VkDescriptorSet。最后在录制 GPU 命令的时候,调用 vkCmdBindDescriptorSets 绑定 VkDescriptorSet 里面所有的描述符。

1.4　输出帧缓冲

3D 程序通常都有自己的输出,这个输出叫作帧缓冲(frame buffer)。

本书 WebGL 的例子,其输出帧缓冲都封装在浏览器引擎的 Canvas 元素里面。

Vulkan 提供了一个 VkFramebuffer 的帧缓冲对象,它的实现比 WebGL/OpenGL 复杂,它还和 VkRenderPass 等概念有关。但是对于本书而言,除非有特殊说明,读者仅需要理解 VkFramebuffer 及其绑定的 VkImage 会被用来作为 Vulkan 程序的输出。

1.5　数据处理过程

数据处理过程其实就是 3D 的流水线,是封装在 GPU 及其相关驱动里面的。不过由于 GPU 的绘图命令,例如 gl.Draw*(WebGL)和 vkCmdDraw*(Vulkan)会直接或者间接地触发 GPU 流水线开始工作,因此可以认为绘图命令封装了数据处理过程。

1.6 TensorFlow JS 的输入输出

本节用输入输出模型来分析 TensorFlow JS 的工作原理。

TensorFlow JS 是开源机器学习软件库 TensorFlow 的 JavaScript 实现，擅长各种感知和语言理解等任务。它支持用 GPU 的片元着色器来实现硬件加速。

用户以 Tensor 的形式输入数据。输入的数据被当作 GPU 的纹理传递给 GPU。GPU 对纹理数据进行运算后，将结果写入输出的纹理。由于输出的纹理是存在 GPU 里面的，CPU 需要用读回操作(gl.readPixels)将数据输出到输出 Tensor 里面。和本书要讨论的其他 GPU 处理模型不同的是，TensorFlow JS 的目标是为了计算，所以 TensorFlow JS 没有将结果输出到一个具体的窗口。其输入输出模型如图 1-6 所示。

图 1-6 TensorFlow 的输入输出模型

测试的例子，如程序清单 1-11 所示。

程序清单 1-11 TensorFlow JS 实现的多个数求乘积

```
// 输入数据 Tensor 1,通过 gl.texSubImage2D/gl.texImage2D 上传给 GPU
const xs = tf.tensor2d([-1, 0, 1, 2, 3, 4, -1, 0, 1, 2, 3, 4, -1, 0, 1, 2, 3, 4, -1, 0, 1,
  2, 3, 4, -1, 0, 1, 2, 3, 4, -1, 0, 1, 2, 3, 4],[36, 1]);
// 输入数据 Tensor 2,同样通过 gl.texSubImage2D/gl.texImage2D 上传给 GPU
const ys = tf.tensor2d([-3, -1, 1, 3, 5, 7, -3, -1, 1, 3, 5, 7, -3, -1, 1, 3, 5, 7, -3,
  -1, 1, 3, 5, 7, -3, -1, 1, 3, 5, 7, -3, -1, 1, 3, 5, 7], [36, 1]);
// 数据处理过程,会调用 gl.Draw*,最后通过片元着色器执行乘法运算
const sum = ys.mul(xs);
// 通过 gl.readPixels 从当前帧缓冲里面读出运算结果
sum.print();
```

1.7 Vulkan 的输入输出

前面通过输入输出方法分析了 GL 和 Vulkan 的差异，本节介绍 Vulkan 一些主要的输入输出对象。Vulkan 在描述输入输出对象的时候，在数据对象之外还抽象出了视图对象、布局对象。视图对象比较直观，它描述了数据对象的范围和格式等信息。布局对象则用于描述当前流水线会使用哪些数据、数据的类型和数目等，所以布局对象描述的其实是流水线的视图。根据 Vulkan 规范，布局对象(VkDescriptorSetLayout)用于描述流水线使用

VkBuffer 和 VkImage 等数据资源的情况。本节在这个基础上拓宽了布局对象的概念，将描述顶点绑定的 VkVertexInputBindingDescription 和 VkVertexInputAttributeDescription、描述 Push Constant 使用情况的 VkPushConstantRange 和输出帧缓冲使用情况的 VkRenderPass 都当作布局对象。

根据输入输出模型，Vulkan 主要提供了三类对象：输入数据对象以及相关的布局对象；输出帧缓冲对象以及相关的布局对象；GPU 命令相关的对象。

1. 输入数据对象及其相关的布局对象

输入对象主要用来描述和管理顶点纹理坐标、顶点索引、MVP、纹理等。相关的布局对象则描述了流水线使用这些对象的情况。

(1) 描述输入数据的实际存储资源。譬如 VkDeviceMemory、VkBuffer、VkImage，用于描述输入存储资源。有些小规模的数据，可以通过 VkBuffer 传递给 GPU，也可以存储到 std::array，然后通过 vkCmdPushConstants 传递给 GPU。

(2) 描述输入数据的布局。例如 VkVertexInputBindingDescription、VkDescriptorSetLayout、VkPushConstantRange 用于描述输入数据的布局。所有输入布局相关的信息，都被聚合到流水线对象 VkPipeline 里面。VkPipeline 里面包含所有输入数据的布局信息，这个布局信息本身也需要通过 vkCmdBindPipeline 来绑定。

2. 输出帧缓冲对象及相关的布局对象

输出对象主要用来描述输出帧缓冲及流水线的输出布局情况。

(1) VkImage、VkFrameBuffer 等用于描述输出帧缓冲的实际存储资源。

(2) VkRenderPass 用于描述帧缓冲的布局。这里的布局信息，指的是当前绘图过程会使用 VkFrameBuffer 的哪些资源。

3. GPU 命令相关的对象 VkCommandBuffer

输入数据对象以及相关的布局对象、输出帧缓冲对象以及相关的布局对象，要经过 GPU 命令进行绑定后，才能参与绘图过程。和绑定对象相关的 GPU 命令主要有以下几个。

(1) 输入数据绑定命令：vkCmdBindVertexBuffers、vkCmdBindIndexBuffer、vkCmdBindDescriptorSets、vkCmdPushConstants。

(2) 输出帧缓冲绑定命令：vkCmdBeginRenderPass、vkCmdEndRenderPass。

(3) 流水线 VkPipeline 绑定命令：vkCmdBindPipeline；VkPipeline 里面包含所有输入数据的布局信息。

GPU 命令管理对象 VkCommandBuffer 通过 vkBeginCommandBuffer、vkEndCommandBuffer 来管理命令的开始和结束。

综合输入数据对象和相关的布局对象、输出帧缓冲对象和相关的布局对象，以及这些对象的绑定命令和着色器的访问方式，得到表 1-1。

表 1-1　数据、布局、绑定命令和着色器访问(MVP 1：VkBuffer；MVP 2：Push Constant)

	数 据 对 象	布 局 对 象	绑 定 命 令	着色器访问

坐标	VkBuffer	VkVertexInputBinding Description	vkCmdBindVertexBuffers	layout (location = 0) in vec3 inPos;
				layout (location = 1) in vec2 inUV;
				layout (location = 2) in vec3 inNormal;

索引	VkBuffer	VkVertexInputBinding Description	vkCmdBindIndexBuffer	gl_VertexID
纹理	VkImage	VkDescriptorSetLayout	vkCmdBindDescriptorSets	layout(binding = 1) uniform sampler2D samplerColor;
MVP 1	VkBuffer	VkDescriptorSetLayout	vkCmdBindDescriptorSets	layout (binding = 0) uniform UBO { mat4 projection; mat4 model; vec4 viewPos; } ubo;

MVP 2	std::array	VkPushConstantRange	vkCmdPushConstants	layout(push_constant) uniform
				PushConsts{
				layout (offset = 0) mat4 mvp;
				} pushConsts;

帧缓冲	VkFrameBuffer VkImage	VkRenderPass	vkCmdBeginRenderPass vkCmdEndRenderPass	layout (location = 0) out vec4 outFragColor;

1.8　GL 和 Vulkan 的线程模型

从 CPU、GPU 硬件的角度看，CPU 硬件和 GPU 硬件都是多线程的。GPU 硬件的多线程指的是 GPU 能够通过多个执行单元来同时运行若干个着色器程序，但是所有这些 GPU 线程，都服务于同一个绘图任务（每次调用 vkCmdDraw * 对应一个绘图任务）。GPU 在同一时刻只能处理一个绘图请求，所以 GPU 硬件是单任务的。

相对于 GL 的单线程，Vulkan 的多线程，本质上是 CPU 的多线程，而不是 GPU 的多线程。所谓 CPU 的多线程，是指可以同时有多个 CPU 线程在录制多个绘图任务的 GPU 命令。但是，同一时刻的 GPU 流水线上面，只有一个绘图任务在执行。这意味着 GL 和 Vulkan 在 GPU 部分的执行模型并没有根本不同，都是单任务的。不同的是 CPU 部分：GL 只支持同一个时刻仅有一个线程录制 GPU 命令，而且录制命令和绘图任务提交命令必须位于同一个线程。Vulkan 则灵活了很多，Vulkan 支持多个线程同时录制 GPU 命令。Vulkan 命令的提交，也可以在其他的线程执行。

考虑一个场景里面的 N 个物体，每个物体有不同的顶点坐标、MVP 矩阵、纹理，分别通过 GL 和 Vulkan 来绘制。

对 GL 而言，只能在一个用户线程里面，按照特定的顺序，逐个录制这些物体的绘制命令并提交给 GPU（调用 glDraw * ）。GPU 线程按照用户提交绘图任务的顺序，逐个处理每个物体的绘图请求，如图 1-7 所示。注意图中的用户线程和 GPU 线程里面的命令，都是按从上到下、从左到右顺序执行的。其中，用户线程在 CPU 上运行，GPU 线程在 GPU 上运行。

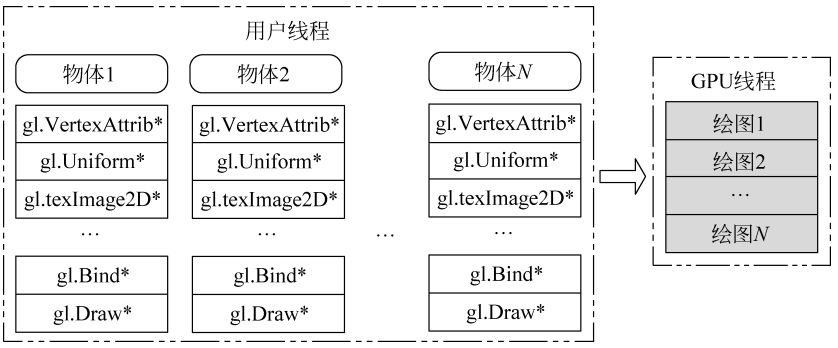

图 1-7　GL 的线程模型（* 表示有多个同类命令）

对于 Vulkan 而言，可以为每个物体创建一个线程来录制物体的绘制命令，每个线程录制好的绘图命令存储在 VkCommandBuffer 里面。在提交绘图任务（和 GL 不同，Vulkan 的 vkCmdDraw * 并不负责提交绘图任务，绘图任务的提交是通过某个线程调用另一个命令 vkQueueSubmit 实现的）的时候，可以一次将多个绘图任务（即多个

VkCommandBuffer)提交给 GPU 线程。GPU 线程按照用户绘图任务的顺序,逐个处理每个物体的绘图请求,如图 1-8 所示。注意图中的多个用户线程是并行执行的,用户线程里面里面的命令,都是按从上到下顺序执行的。其中,N 个用户线程在 CPU 上运行,GPU 线程在 GPU 上运行。

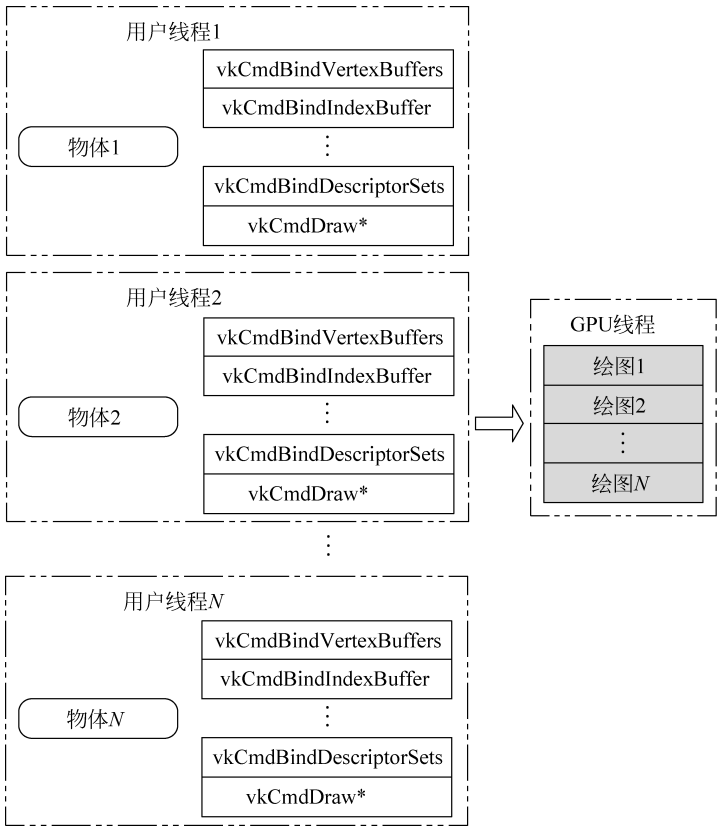

图 1-8　Vulkan 的线程模型(* 表示有多个同类命令)

针对 OpenGL 实现的应用,如果 CPU 占用的计算时间比 GPU 多,Vulkan 多线程可以提升性能,如图 1-9 所示。

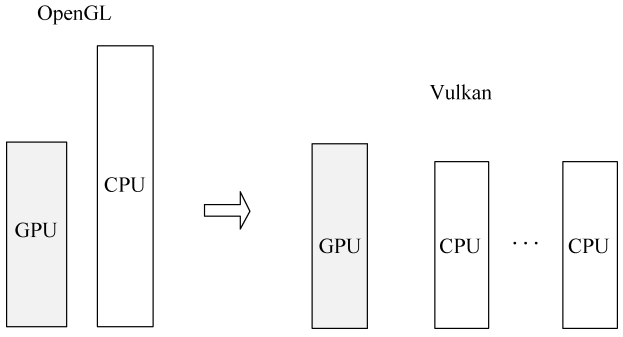

图 1-9　Vulkan 可以提升性能

对于 CPU 占用时间比 GPU 短的情况，Vulkan 可以降低功耗，如图 1-10 所示。

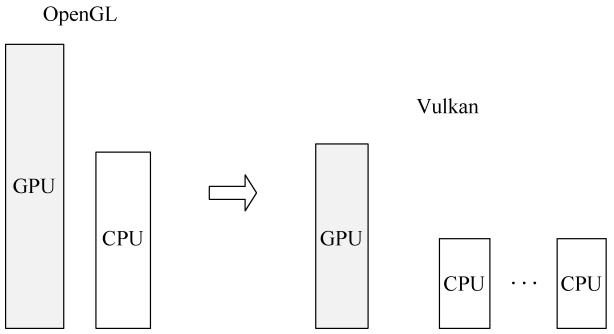

图 1-10　Vulkan 可以降低功耗

1.9　源码下载和编译

本书示例的 Vulkan 源代码 https://github.com/math3d/Vulkan/tree/projection_perspective，是基于开源示例程序 https://github.com/SaschaWillems/Vulkan 修改而来。可以运行在 Ubuntu 和 Windows 环境。

Vulkan 源码的获得：

```
$ git clone https://github.com/math3d/Vulkan.git
$ git submodule init
$ git submodule update
```

Vulkan 源码的编译 Ubuntu 18.04：

```
$ cmake CMakeLists.txt
$ make
```

Vulkan 源码的编译 Windows 10：

```
$ cmake -G "Visual Studio 15 2017 Win64"
```

用 Visual Studio 打开项目 vulkanExamples.sln，就可以编译了。

本书示例的 WebGL 源代码，根目录位于 https://github.com/math3d/WebGL。WebGL 源码可以运行在主流的 Web 服务器上面。

小　　结

虽然 GL(WebGL)难以和 Vulkan 的接口一一映射起来，但是两者的输入数据、输出数据等，却是非常类似的。给定一个 3D 场景，在大多数情况下，用 GL 或者 Vulkan 都可

以实现对该场景的渲染。而从 GL 或者 Vulkan 对输入数据的描述来看，两者在接口方面有很多是类似的，总结如表 1-2 所示。从这个角度来说，基于输入数据输出数据的 3D 程序分析方法有助于理解 3D 程序的数据模型。当然，这个方法主要用于程序分析，实际编程的时候还是需要查阅具体标准理解每个接口的含义。

表 1-2　WebGL Vulkan 数据操作接口对比

数据类型		准备数据	更新描述符（仅 Vulkan）	录制命令时绑定
Vertex/Index	WebGL	gl. createBuffer gl. bindBuffer gl. bufferData		gl. bindBuffer gl. vertexAttribPointer
	Vulkan	vkCreateBuffer vkAllocateMemory vkMapMemory vkBindBufferMemory vkUnmapMemory		vkCmdBindVertexBuffers vkCmdBindIndexBuffer
MVP 数据	WebGL			gl. uniformMatrix4fv
	Vulkan	vkCreateBuffer vkAllocateMemory vkMapMemory memcpy vkUnmapMemory	vkUpdateDescriptorSets	vkCmdBindDescriptorSets
纹理数据	WebGL	gl. createTexture gl. bindTexture gl. texImage2D		gl. activeTexture gl. bindTexture
	Vulkan	vkCreateImage vkAllocateMemory vkBindImageMemory vkMapMemory memcpy vkUnmapMemory vkCreateSampler vkCreateImageView	vkUpdateDescriptorSets	vkCmdBindDescriptorSets

本章的例子，都是由一次绘制过程完成的。实际上，为了达到更好的性能，如延迟渲染，实现某些特殊的效果（例如阴影），系统中可能使用了多次绘制。无论绘制多少次，都可以使用输入输出的分析方法。

本书重在分析 3D 编程的 3D 几何模型，方法是分析给定的输入数据，经过了什么样的流程得到输出数据。因此虽然本书使用的 Vulkan 例子源码冗长，但是如果使用输入数据输出数据的分析方法，背后的逻辑会简单很多。有了这个办法，读者就没必要担心数量庞大的 Vulkan 的编程接口了。当然，具体到每个 Vulkan 接口，读者还是要去查阅工具书理解每个接口背后的具体含义。

第 2 章　3D 图形学基础

本章介绍几何和图形编程的一些基本概念。

2.1　符号和约定

本书中涉及较多的数学公式。对于数学公式中的符号，以及正文对公式中符号的引用，约定如下。

标量用小写字母斜体表示：a，b。其中，坐标轴和坐标分量用 x，y，z，u，v，w 等表示。

向量用小写字母粗斜体表示：$\boldsymbol{a}$，$\boldsymbol{b}$。

几何意义上的点用大写字母斜体表示：A，B。

矩阵用大写字母粗体表示：$\boldsymbol{M}$。

在本书的部分章节，为了排版的需要，有时候不会刻意去区分整数和浮点数，例如 1 和 1.0、0 和 0.0 在本书都被当作是同一个数。

2.2　向量的基本运算

向量的模如下。

向量 $\boldsymbol{a}(x,y,z)$ 的模是向量的大小，表示为：

$$|\boldsymbol{a}|=\sqrt{x^2+y^2+z^2}$$

向量 $\boldsymbol{a}$ 的单位化向量 $\boldsymbol{a}'$ 为：

$$\boldsymbol{a}'=\frac{\boldsymbol{a}}{|\boldsymbol{a}|}$$

向量的点乘 $\boldsymbol{a}\cdot\boldsymbol{b}$ 为：

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\theta$$

点乘的几何意义是，如果 $\boldsymbol{b}$ 是单位向量的话，$\boldsymbol{a}\cdot\boldsymbol{b}$ 得到的就是向量 $\boldsymbol{a}$ 在向量 $\boldsymbol{b}$ 上的投影的长度，如图 2-1 所示。

向量加法 $\boldsymbol{a}+\boldsymbol{b}$，如图 2-2 所示。

向量减法 $\boldsymbol{a}-\boldsymbol{b}$，如图 2-3 所示。

在任意两个点坐标 A 和 B 之间做减法，可以用来确定两个点之间的向量，即 $A-B$ 得到的是 B 指向 A 的向量。

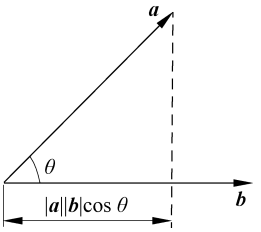

图 2-1　点乘

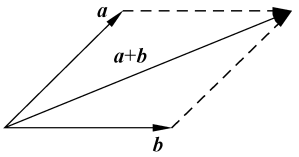

图 2-2　向量加法

三维空间中的两个向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 相乘，叫作叉乘 $\boldsymbol{a}\times\boldsymbol{b}$。叉乘具有下面的性质。

(1) $\boldsymbol{a}$，$\boldsymbol{b}$，$\boldsymbol{a}\times\boldsymbol{b}$ 的方向遵守右手法则。右手法则如图 2-4 所示，$\boldsymbol{a}$ 指向大拇指，$\boldsymbol{b}$ 指向食指，$\boldsymbol{a}\times\boldsymbol{b}$ 指向中指。

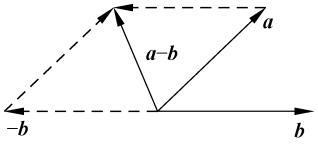

图 2-3　向量减法

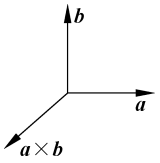

图 2-4　叉乘右手法则

(2) $\boldsymbol{a}\times\boldsymbol{b}$ 的模的长度，等于以 $\boldsymbol{a}$，$\boldsymbol{b}$ 为边的平行四边形的面积。

$\boldsymbol{a}\times\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\sin\theta\boldsymbol{n}$，其中，$\theta$ 代表了 $\boldsymbol{a}$，$\boldsymbol{b}$ 在平面上的夹角，且 $\theta\in[0°,180°]$。

2.3　齐 次 坐 标

德国数学家 August Ferdinand Mobius 提出的齐次坐标在透视投影中使用得非常普遍。齐次坐标是在原来笛卡儿坐标的维度上增加一个维度的坐标表达方式。如笛卡儿坐标(x',y',z')和齐次坐标(x,y,z,w)之间的关系如公式 2-1 所示。

$$\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{x'}{w}\\\frac{y'}{w}\\\frac{z'}{w}\end{pmatrix}$$

公式 2-1　笛卡儿坐标到齐次坐标

当 w 非 0 的时候，(x,y,z,w)是一个点；当 w 为 0 的时候，$(x,y,z,0)$是一个无穷远的点。我们更常用这个无穷远的点来表示具有大小和方向的向量(向量没有位置的概念，所以可以在空间里面平移)，因而有了 w 分量就可以用一个齐次坐标表示两种不同的量。

齐次坐标翻译自“homogeneous coordinates”。homogeneous 的英文解释是“同一种的、类似的”。中文对“齐次”的解释是“次数相等的意思”。此处英文解释更贴近齐次坐标