2025-10-09 14:32 | felixxfwang

# 零基础快速上手Vulkan：Android高性能图形开发的必修第一课！

## 1. Vulkan 概述

Vulkan是由**Khronos Group**开发的一种现代、高性能的图形和计算API。Vulkan号称下一代图形API，它的设计目标是取代OpenGL。Vulkan可以为开发者提供更直接、更细粒度的硬件控制。

Vulkan的前身是AMD公司的开发的**Mantle**。Mantle在2013年率先打破了传统图形API的限制，让开发者能够直接显式操作底层GPU硬件，显著提升了游戏性能。不过由 于AMD公司的影响力有限，Mantle并未能成为行业标准。

但是微软公司在Mantle的思路上，开发出了DirectX 12；Apple在其基础上，开发出了Metal。

2015年，AMD将Mantle捐献给了**Khronos Group**，并停止自身开发。**Khronos Group**以Mantle为基础，开发出了Vulkan，并扩展了其功能。

2016年，Vulkan 1.0正式发布，并成为了跨平台的行业标准。Android也在7.0上全面支持了Vulkan API。

---

## 2. Vulkan 与 OpenGL 的对比

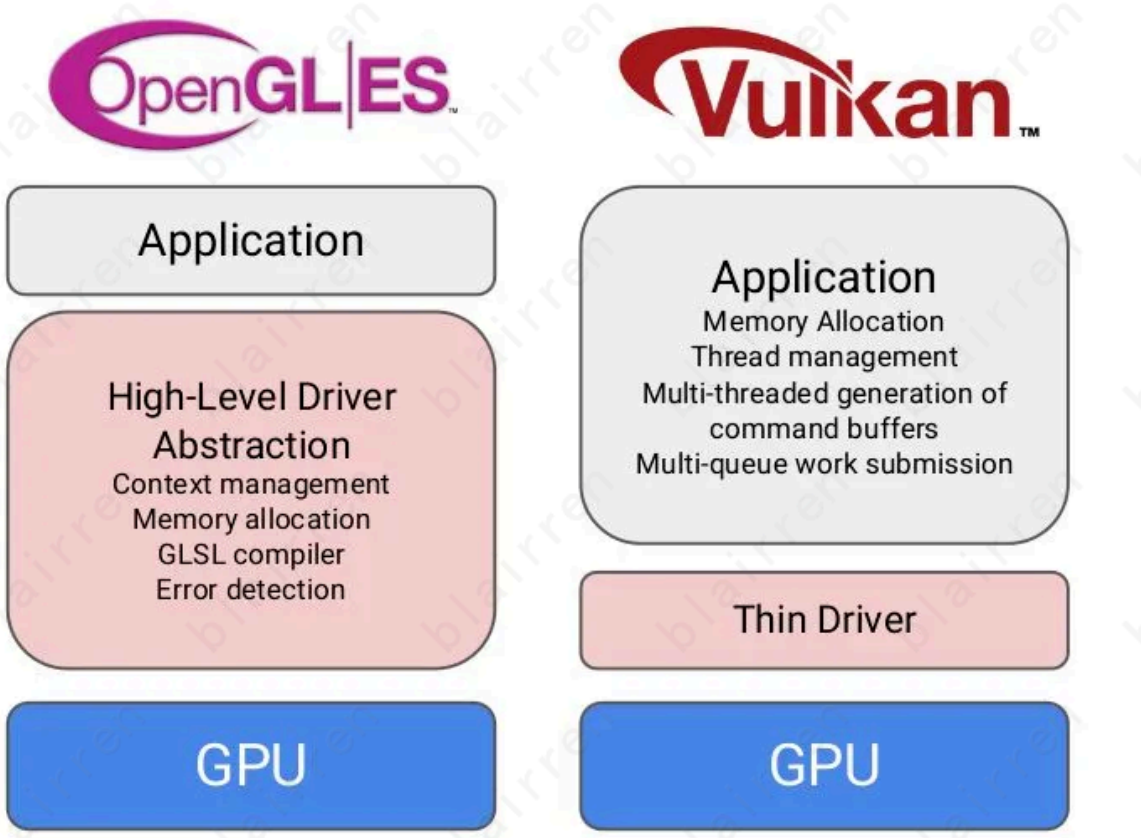
Vulkan作为下一代图形API，它与传统的OpenGL（ES）API有什么不同呢?

| 特性 | Vulkan | OpenGL |
| --- | --- | --- |
| GPU驱动层 | 显式GPU控制，开发者负责API验证、内存管理、多线程管理。 | 隐式GPU控制，驱动负责API验证、内存管理、线程管理。 |
| 多线程支持 | 引入Command Buffer，支持多线程并行提交渲染命令。 | 单线程模型，所有渲染操作在同一线程。 |
| Shader | 只支持SPV格式的二进制Shader，驱动内部编译，有缓存优化。 | 使用GLSL，运行时显式编译和链接。 |
| API设计 | 无状态，通过组合的模式实现高度解耦。 | 全局状态机，各模块紧密耦合。 |
| 代码复杂度 | 显式控制GPU，代码量大（绘制三角形可达上千行）。 | 隐式API，少量代码即可实现功能。 |
| 学习曲线 | 需开发者控制所有细节，使用复杂，学习成本高。 | 抽象层次高，使用简单。 |
| 性能表现 | CPU瓶颈时性能显著高于OpenGL，GPU瓶颈时表现相似。 | CPU瓶颈时性能受限，GPU瓶颈时表现相似。 |

1. GPU驱动层

Vulkan相比于OpenGL，有着更薄的驱动层。OpenGL是隐式GPU控制，驱动层负责了API验证、内存管理、线程管理等工作，这种大包大揽什么事情都管的做法，即使应用使用API出错，驱动也会帮忙解决处理，从而保证应用的正常运行。开发者使用起来非常简单。

而Vulkan把API验证、内存管理、多线程管理等工作全部交由开发者负责，号称没有任何秘密的API。一旦开发者使用API出错，程序就会出现crash。 这种方式无疑增加了API使用的复杂度和困难度，但换来的是性能上巨大的提升。据统计，单单是在驱动中去掉API验证的工作，就把性能提升了9倍。




2. 多线程支持

OpenGL是单线程模型，所有的操作都是在同一线程内完成，并且OpenGL Context是和线程绑定的一个概念，因此它对多线程的支持及其不友好。

而Vulkan通过引入CommandBuffer这一核心概念，支持多线程并行记录GPU指令到CommandBuffer上，最后将指令统一提交给GPU执行。

3. Shader

OpenGL使用的shader语言是GLSL，并且会在运行时进行显式的编译和链接。

而Vulkan只支持SPIR-V格式的二进制中间格式的shader，会在运行时在驱动内部进行编译，并且有编译缓存优化。

4. API设计

OpenGL有一个全局状态机，每进行一次接口调用都会改变这个状态机的状态，从而导致OpenGL各个模块之间是紧密耦合在一起的。

Vulkan是一种无状态的设计，通过组合的方式实现了各个模块间的高度解耦。两个API使用起来，有点像面向对象和面向过程这两种设计模式的区别。

5. 代码复杂度

OpenGL因为是隐式API，驱动层做了大量的隐式工作，所以使用OpenGL开发一个功能，代码量相对来说是比较少的。

Vulkan因为是显式API，所有的细节都需要开发者来控制，所以Vulkan程序的代码量一般偏大，一个简单的三角形绘制程序的代码量高达上千行。

6. 学习曲线

OpenGL的抽象层次高，学习起来相对简单。

Vulkan需要开发者控制所有细节，使用复杂，所以学习成本相对较高。

7. 性能

Vulkan的核心优势在于**显式控制和多线程功能**。这些特性使我们能够在更少的CPU时间内向GPU提交更多命令，从而在CPU成为性能瓶颈时，获得巨大的性能提升。当然如果你的应用程序的性能瓶颈在GPU的话，Vulkan和OpenGL的性能表现其实是差不多的。


总结来说，Vulkan相比于OpenGL的优势很大，但就目前形势而言，OpenGL (ES) 并不会被完全取代。Vulkan作为一个更为现代化和高效的解决方案，更适

合构建对渲染性能和效率要求高的大型软件，比如3D游戏。而在移动开发中，一些像图像视频特效、图形图像渲染等轻量级的场景，使用OpenGL ES，开发者可以在更短的时间内完成交付。

---

## 3. Vulkan开发基础概念

Vulkan的复杂性决定了其相比OpenGL拥有更多复杂的概念，掌握这些概念是进行Vulkan程序开发的前提。

本文将Vulkan的基础概念分为了7组进行介绍。

### 3.1 实例、设备和队列

- **Vulkan 实例（Instance）**：Vulkan API的基本概念，代表一个完整的Vulkan环境，连接Vulkan库与应用程序。应用程序必须先创建实例才能执行其他Vulkan操作。
- **物理设备（PhysicalDevice）**：指支持Vulkan的物理硬件，通常是显卡。我们可以从物理设备上获取其支持的Vulkan版本信息和功能特性。
- **队列（Queue）**：用于存储应用程序提交的GPU指令，物理设备会读取并执行队列中的指令。一个物理设备可能有多个队列，比如图形队列、计算队列和传输队列等。
- **逻辑设备（Logical Device）**：与物理设备交互的接口，抽象了对特定GPU的访问。允许应用程序提交命令和管理资源，无需直接与硬件交互。可根据需求关联物理设备的一个或多个队列。逻辑设备类似于OpenGL中的上下文，后续大多数Vulkan接口，都需要传递一个逻辑设备的参数。

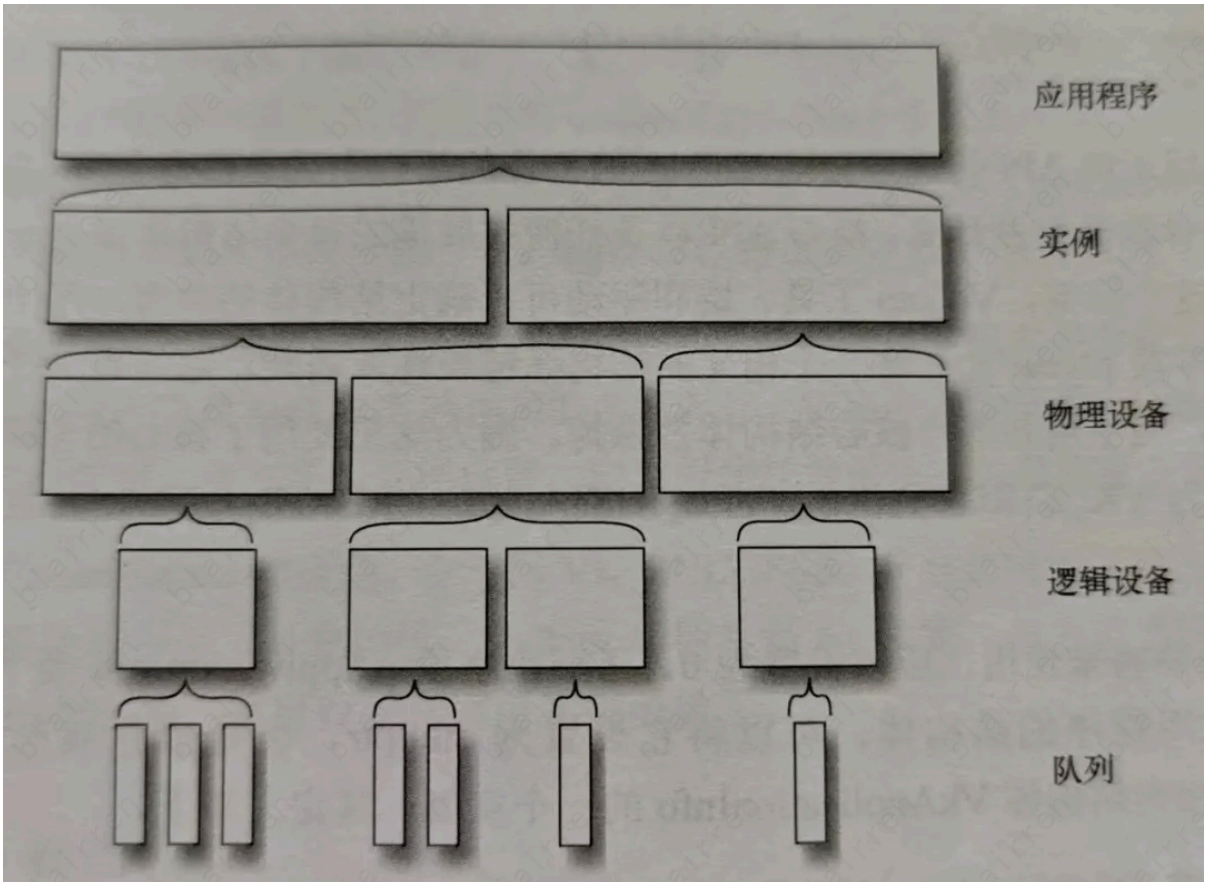



图 1.1 Vulkan 里关于实例、设备和队列的层级关系

## 3.2 内存管理、缓冲区和图像

在内存管理方面，OpenGL类似于Java，不需要开发者关心内存的分配和回收；Vulkan则更像是C，需要开发者显式分配和释放内存。

### 3.2.1 Vulkan内存类型

Vulkan将内存划分为两大类：

- 主机内存（**Host Memory**）
- 设备内存（**Device Memory**）

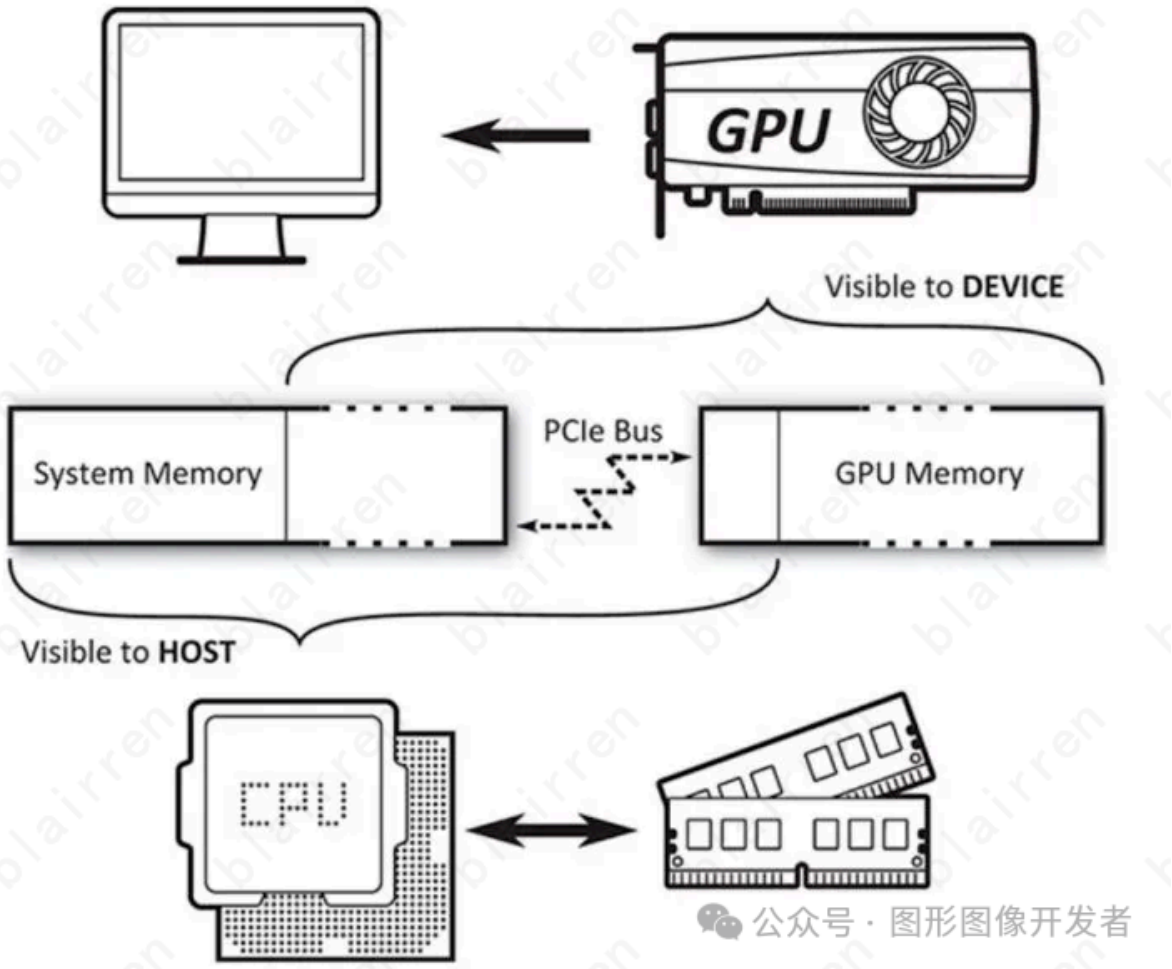
在移动设备上，主机内存指的是CPU内存，设备内存指的是GPU显存。

Vulkan提供了一种透明的机制，让开发者可以显式地控制内存的类型和布局，这在OpenGL中是无法想象的。

按照GPU访问速度由快到慢，Vulkan的内存可以细分为以下四类：

- **Device Local Memory**：仅对GPU可见，通常是显存，性能最高。
- **Device Local Host Memory**: 由GPU管理，对CPU可见的内存。

- **Host Local Device Memory**: 由CPU管理，对GPU可见的内存。
- **Host Local Memory**：仅对CPU可见，通常是普通内存。




具体选择什么类型的内存，需要进行性能、容量、访问频率等多方面综合考虑。比如对于较大的只在GPU侧访问的数据，应该优先考虑存储在Device Local Memory；对于需要CPU高频更新且在GPU侧访问的数据，就需要使用Device Local Host Memory了。

缓冲区和图像，是Vulkan提供的两种不同的数据结构。

### 3.2.2 缓冲区（Buffer）

缓冲区（**Buffer**） 是一种用于存储通用数据的线性内存块，内存布局连续，适合顺序访问。一般用于存储顶点数据、索引数据、Uniform数据等。与OpenGL中的Buffer类似。

创建Buffer时需要指定使用的内存类型和Buffer的用途。内存类型就是上一小节中提到的4中，而Buffer用途包括但不限于以下几种：

- `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`：顶点数据

- `VK_BUFFER_USAGE_INDEX_BUFFER_BIT`：顶点索引
- `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`：Uniform
- `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT`：SSBO

顶点数据和顶点索引数据，这个比较明显。Uniform和SSBO有什么区别呢？

| 特性 | Uniform Buffer (UBO) | Shader Storage Buffer Object (SSBO) |
| --- | --- | --- |
| 主要用途 | 传递常量和小量数据。 | 传递大量、可读写数据。 |
| 典型示例 | 变换矩阵、光照参数、时间。 | 顶点数据、粒子系统、骨骼、场景物体列表。 |
| 读写权限 | 只读。 | 可读可写。 |
| 内存布局 | `std140`（严格）。 | `std430`（宽松）。 |
| 大小限制 | 严格，通常在 64KB。 | 宽松，通常在 GB 级别。 |

### 3.2.3 图像（Image）

图像（Image） 用于存储多维图像数据，相当于OpenGL中的纹理。它比缓冲区更复杂，因为它包含独特的布局和格式信息。

图像的类型包括：

- 2D图像
- 3D图像
- 多级别图像

图像的用途包括：

- 纹理贴图
- 渲染目标
- 深度缓冲区
- 多层渲染目标

## 3.3 窗口表面与交换链

### 3.3.1 窗口表面（Surface）

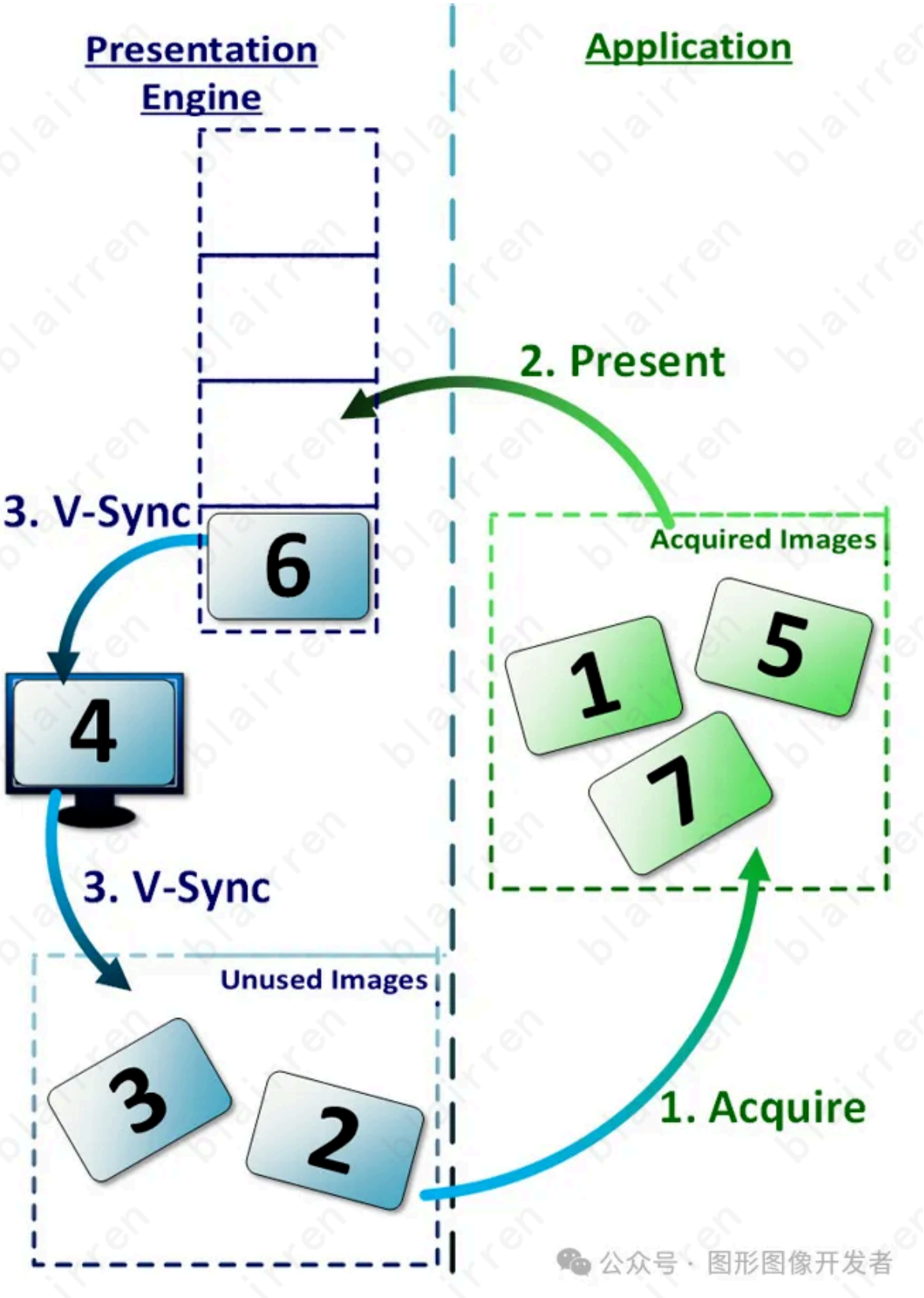
Vulkan 是一个平台无关的 API，它不能直接与窗口系统（如 Android 的窗口）交互。为了解决这个问题，Vulkan 引入了 **WSI（Window System Integration）**，使Vulkan应用程序可以与不同操作系统的窗口系统进行交互。

而窗口表面（**Surface**） 就是WSI机制通过特定硬件/操作系统创建的抽象对象，该对象是Vulkan连接外部窗口的接口。Surface本身并不存储你最终绘制的像素。它只是一个句柄（handle），链接了Vulkan实例和你选择的那个原生窗口（例如，Android上的ANativeWindow 或 Windows的HWND）。

### 3.3.2 交换链（SwapChain）

交换链（**Swapchain**） 是应用程序与窗口系统之间的桥梁，负责将渲染结果呈现给用户。它管理一组用于显示到屏幕的图像，让开发者能显式地控制双缓冲或三缓冲。

相比之下，OpenGL的双缓冲机制，通常通过 `glSwapBuffers`接口调用，由驱动层隐式管理。




这里额外补充一下，为什么会存在三缓冲这种技术，双缓冲有什么问题呢?

双缓冲解决了屏幕撕裂，但可能因为等待显示器而导致帧率不稳。

三缓冲在双缓冲的基础上，通过增加一个缓冲区，解决了GPU闲置的问题，从而在保持画面无撕裂的同时，提高了帧率和流畅性。

## 3.4 渲染通道和帧缓冲区

### 3.4.1 渲染通道（RenderPass）

**Vulkan 渲染通道（RenderPass）** 是对一个渲染流程的高级抽象，定义了渲染过程如何被组织和执行，用于优化渲染管线。

渲染通道定义了以下内容：

- 附件（**Attachments**）：定义渲染到哪里
- 加载/存储操作 **(Load/Store Operations)**：定义渲染开始结束时如何处理附件中的数据
- 子通道 **(Subpasses)**：定义渲染流程分为几个阶段

渲染通道是Vulkan的一个核心概念，OpenGL中并没有和其直接对应的概念。

### 3.4.2 帧缓冲区（Framebuffer）

帧缓冲区（**Framebuffer**）：一个容器，包含一组图像视图，作为渲染通道中的附件。它定义了渲染操作的实际内存目标。

Vulkan中的FrameBuffer是依赖于RenderPass而存在的，它在创建时必须指定一个RenderPass。而OpenGL中的FrameBuffer是一个更通用的对象，可以绑定纹理或者渲染缓冲区，glBindFrameBuffer后的绘制目标就会变为该FrameBuffer。

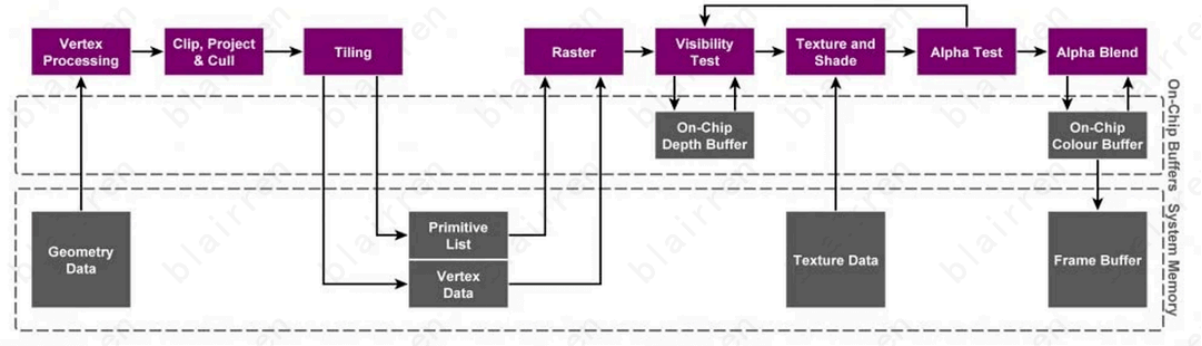
- **VkRenderPass** 相当于**渲染的结构**：定义做什么（渲染的步骤、依赖、附件操作）。
- **VkFramebuffer** 相当于**渲染的数据容器**：定义在哪儿做（具体的图像视图）。

### 3.4.3 渲染通道的存在意义

为什么Vulkan会有渲染通道这样一个高级抽象的概念呢？

RenderPass提供了Vulkan独有的优化机会，例如**子通道**。它允许开发者将复杂的渲染流程分解为多个子通道，子通道是渲染通道内部的一个阶段。前一个子通道的输出会成为下一个子通道的输入。

移动GPU一般是Tile-Base Rendering渲染架构，如下图所示：

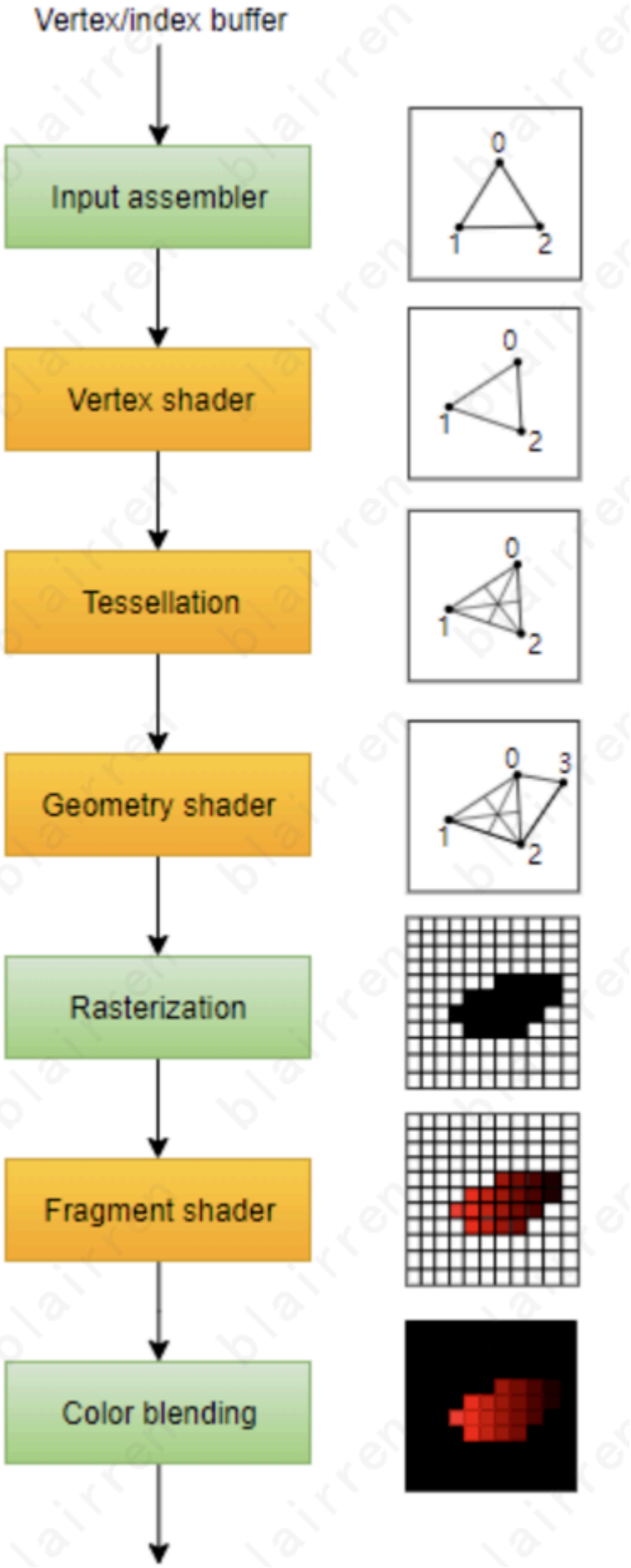



子通道让GPU能够在高速的片上缓存中传递数据，而无需将中间结果写回昂贵的主显存。这对于移动GPU尤为重要，因为它可以极大地减少内存带宽消耗。

## 3.5 图形管线和描述符集

### 3.5.1 图形管线（Pipeline）

现代GPU编程，都定义了一条可编程渲染管线。如下图所示：




其中最常见的可编程阶段就是顶点着色器和片元着色器。Vulkan中的图形管线对应的就是GPU编程中的可编程渲染管线。 图形管线（**Pipeline**）：可以看作是一条渲染流水线，定义了从输入顶点到最终输出图像的所有步骤。它是一个完整

的、不可变的状态集合。开发者必须提供一个顶点着色器和多个可选的其他着色器来构建Pipeline。

创建Pipeline依赖于一个RenderPass。在创建管线时，开发者必须指定它将用于哪个渲染通道的哪个子通道。所以说，Pipeline才是渲染通道里真正执行渲染操作的那个角色。

### 3.5.2 描述符集（DescriptorSet）

描述符集（**DescriptorSet**）：一个容器，用于将着色器使用的资源（比如如Uniform Buffer、Image等）与着色器中的绑定点进行关联。

这种数据绑定方式更加解耦，比OpenGL的直接绑定更灵活。

可以简单理解管线和描述符集之间的关系：

- **Pipeline** 负责定义渲染的规则。
- **DescriptorSet** 负责提供数据。

### 3.5.3 图形管线与渲染通道的关系

这里总结一下RenderPass、FrameBuffer和Pipeline三者之间的关系。我们以报纸印刷来类比：

- **RenderPass** 是报纸的版面设计：它定义了哪里是图片区，哪里是文字区，以及每一页的布局。
- **Framebuffer** 是具体的报纸纸张：它将版面设计与真实的纸张（图像）联系起来。
- **Pipeline** 是印刷机：它定义了用什么油墨、什么字体、什么印刷速度来将内容打印到纸张上。

## 3.6 指令缓冲区

CommandBuffer是OpenGL和Vulkan在API设计上最根本的区别之一，直接影响了两者在性能、多线程能力上的差异。

指令缓冲区（**CommandBuffer**）：用于记录和存储一系列绘图和计算指令。它在CPU上预先记录命令，然后一次性提交给设备队列，物理设备会从队列中读取指令并执行。这种方式极大地减少了CPU负载。

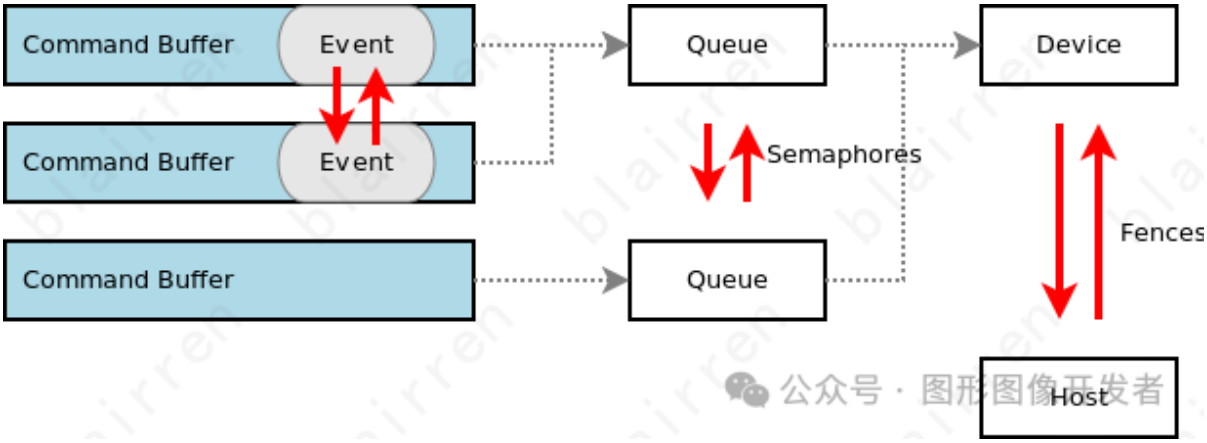

而OpenGL是即时模式的，即每次DrawCall命令会立即或在驱动程序的队列中排队执行。这种方式对CPU的开销较大，所以OpenGL有个常见的优化手段叫做减少DrawCall次数，以及DrawCall合批（比如使用实例化渲染技术）。

## 3.7 同步机制

Vulkan是高度并发的API，不可避免的存在资源竞争和同步。因此Vulkan开发者需要明确管理同步，防止读写冲突、数据损坏等问题。

Vulkan的同步机制用于控制GPU和CPU之间的任务执行顺序，确保资源的正确访问顺序和任务的按序执行。

Vulkan主要提供了以下四种同步机制，每种机制用于不同的场景。




### 3.7.1 栅栏 (Fences)

- 机制： CPU → GPU
- 用途： 阻塞 CPU 线程，直到 GPU 完成某个提交操作（`vkQueueSubmit`）。
- 使用场景：
  - 帧同步 (Frame Synchronization)： 这是最常见的用途。在渲染一帧后，使用栅栏等待该帧的所有工作完成，然后 CPU 可以安全地重置命令缓冲区、更新 Uniforms，并开始下一帧的录制。
  - 资源释放： 确保某个资源（如命令缓冲区）在被 CPU 重复使用或销毁之前，GPU 已经完成了对它的使用。

### 3.7.2 信号量 (Semaphores)

- **机制：** GPU → GPU
- **用途：** 协调 GPU 上的不同操作，确保它们按正确的顺序执行。它们是**不可重置**的，一旦发出信号就被消耗。
- **使用场景：**
  - **交换链同步：** 两个主要的信号量是必须的：
    - 图像可用信号量 (`Image Available`)：当 `vkAcquireNextImageKHR` 成功获取到下一个交换链图像时发出信号，告诉 GPU 渲染操作可以开始了。
    - 渲染完成信号量 (`Render Finished`)：当所有渲染命令完成时发出信号，告诉 呈现操作可以开始了（`vkQueuePresentKHR`）。
  - **跨队列依赖：** 确保一个队列（例如，传输队列）完成数据传输后，另一个队列（例如，图形队列）才能开始使用这些数据。

### 3.7.3 事件 (Events)

- **机制：** GPU ↔ GPU 或 GPU ↔ CPU
- **用途：** 提供比信号量更细粒度的同步控制，可以从 CPU 或 GPU 写入 (`vkCmdSetEvent`) 和等待 (`vkCmdWaitEvents`)。
- **使用场景：**
  - **命令缓冲区内部同步：** 在一个非常长的命令缓冲区中，如果只有一部分命令需要等待前面的命令完成，可以使用 Event 来精确地同步这部分命令，而无需等待整个命令缓冲区完成。
  - **异步计算：** 例如，在图形管线中等待一个计算着色器（Compute Shader）完成对某一资源的写入后，才开始相关的渲染读取操作。

### 3.7.4 屏障 (Barriers)

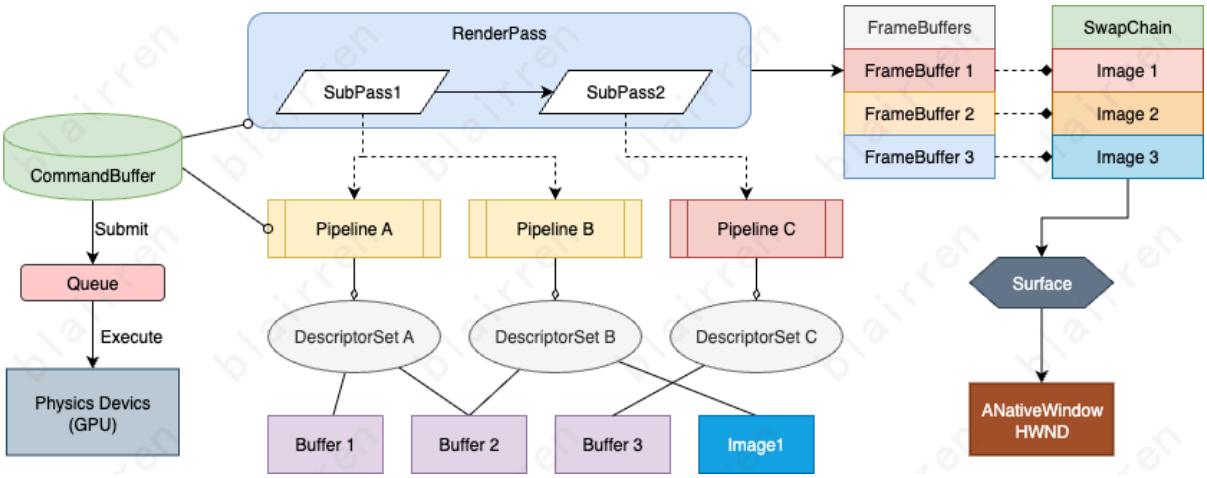
- **机制：** 内存/数据同步
- **用途：** 确保内存操作的可见性和依赖性。它强制 GPU 完成所有前一个操作的内存写入，并使所有后续操作可见。

- 核心组成： 屏障（`vkCmdPipelineBarrier`）需要定义三个关键信息：
  - **源阶段/访问掩码：** 完成什么工作（管线阶段）和什么操作（内存访问类型）。
  - **目标阶段/访问掩码：** 开始什么工作和什么操作。
  - **图像布局转换：** 针对图像资源，定义其从旧布局（`oldLayout`）到新布局（`newLayout`）的转换。
- 使用场景：
  - **图像布局转换：** 必须在图像从一种用途（例如，渲染附件 `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`）切换到另一种用途（例如，着色器读取 `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`）时使用。
  - **数据依赖：** 确保一个着色器（例如，计算着色器）对一个 SSBO 的写入完全完成并被缓存刷新后，下一个着色器（例如，顶点着色器）才能开始读取数据。

---

# 4. 总结

## 4.1 Vulkan概念总结




## 4.2 Vulkan Or OpenGL?

- Vulkan的优势在于其显式控制和多线程，这使其在CPU密集型渲染任务中表现卓越。
- 对于大型3D游戏等对性能要求极高的软件，Vulkan 是一个更现代化和高效的解决方案。

- 对于轻量级、快速开发的图形应用，OpenGL凭借其简洁性，仍有其不可替代的优势。

---