第 2 章 你的第一个 Vulkan 伪代码程序

> 在上一章中，我们提供了一个比较基本的介绍，以便可视化新一代的 Vulkan API。 我们通过这套 API 的高级生态系统设计进行了盘点，并了解内部模块的功能，以此来理解其执行模式。

> 在本章中，我们会了解一下 Vulkan 环境的安装过程，以便使用 Vulkan 伪代码进行编程做好准备。 Vulkan 的明确性会使编程代码变得更加冗长。 在 Vulkan 中，一个简单的 Hello World !!! 程序最终可能会有大约 1,500 行代码。 这意味着即使是一个简单的例子，对初学者来说也是一个挑战。 但我们先不管这些具体的代码；我们会将整个 Hello World！ 程序使用简单的伪代码编程的方式进行讲解。

初学者还将学习如何以一种用户友好的方式构建他们的第一个 Vulkan 应用程序，即分步骤操作的方法。 在本书接下来的章节中，我们会深入研究实际的编码过程，并利用 Vulkan 编程来解决问题。 所以学习过程分为几个模块、多个章节。

本章为其余的章节奠定了基础。 在这里，我们将构建一个非常简单的 Hello World！。这是一个 伪代码程序，我们将通过这份代码来了解使用 Vulkan 构建简单三色三角形的过程。 本章将涵盖以下主题：

- 安装 Vulkan 环境
- HelloWorld !!! 伪代码程序
- 整合在一起
- 安装 Vulkan

我们已经讨论了很多 Vulkan 的内容， 现在让我们深入研究一下 Vulkan 的安装过程，以及为了让 Vulkan 进行一些实际的作业任务需要掌握的内容。

提示

在继续安装之前，请仔细阅读本书提供的代码文件中的软硬件要求。 如果您的系统符合上述要求，那么您最好按照本章介绍的安装过程一起操作。

请按照以下说明安装 Vulkan：

1. **Vulkan 驱动程序**：大多数供应商现在都将 Vulkan 支持包含在常规的驱动程序包中。 首先，安装 Vulkan 驱动程序。 您可以选择安装位置；否则就会使用默认位置进行安装。 例如，如果您正在安装 NVIDIA 驱动程序，安装程序首先会检查系统的配置以扫描安装驱动程序的任何兼容性问题。同时它会升级系统上预装的驱动程序。
2. **安装 Python**：安装 Python 并确保将其添加到环境变量 PATH 中。 这可以通过简单地 勾选**Add Python to PATH**的复选框来实现。
3. **安装 CMake**：接下来安装 CMake。 确保选中 Add CMake to the system PATH for all users。 您可以使用默认位置进行安装。
4. **安装 SDK**：安装 LunarG SDK。 使用默认位置是没有什么问题的。

注意

LunarG SDK 包含 Vulkan 规范，手册以及有助于构建项目的必要库。 它还包含了可快速启动的演示示例，以检查安装状态。 如果您能够成功运行示例的可执行文件，这意味着 Vulkan 驱动程序和 SDK 已正确安装。 您可以在 / Bin 或 / Bin32（用于 32 位系统）下找到这些示例。

Hello World!!! 伪代码

在本节中，我们将构建我们的第一个 Hello World！ Vulkan 应用程序。 该应用程序使用伪代码编程模型构建，它具有以下优点：

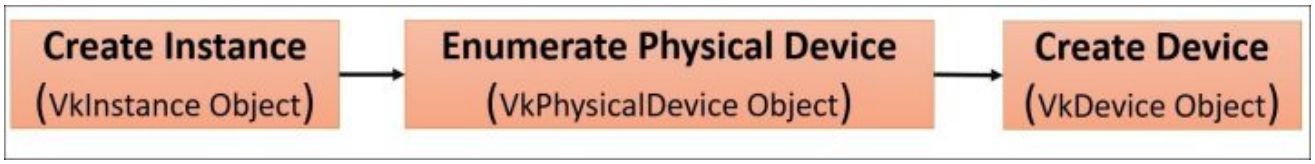
- 通过按步骤进行的过程学习如何构建 Vulkan 应用程序。
- 由于 Vulkan 编码冗长，因此初学者可能会迷失在细节中。 该伪代码仅仅强调了易于理解的必要细节。
- 这种紧凑形式的代码程序，对于首次使用 Vulkan 的读者来说更容易记忆。
- 每个伪代码都使用真正的 Vulkan API，并解释其控制流程。
- 在本章的最后，如果你是一个完完全全的初学者，你将能够理解 Vulkan 编程以及从零开始构建应用程序的所有必要线索。 此外，您将了解 Vulkan API 的高级概念及其职责和功能。
- 要详细了解 API，请使用 LunarG SDK 提供的 Vulkan 规范。 或者参考 **https://www.khronos.org/registry/vulkan/specs/1.0/apispec.html** 。

提示

鉴于本章的篇幅所限，不可能提供每个数据结构字段和 API 参数的详细描述。 伪代码仅限于为大多数重要的数据结构或 API 提供高级定义、概述和最多一到两行的相关功能描述。 在我们继续阅读本书后续章节时，会全面介绍 Vulkan 的所有 API 以及相关数据结构。

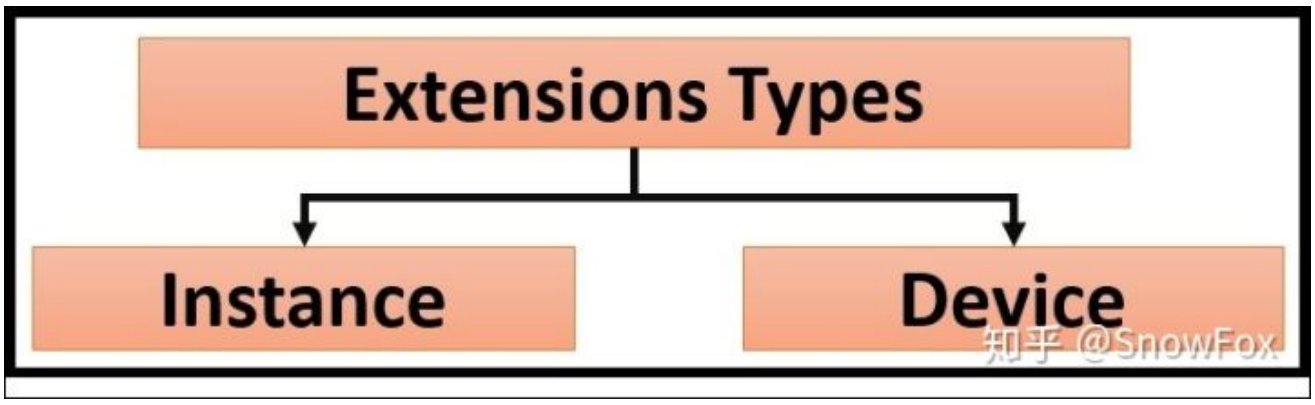
初始化 - 与设备握手

Vulkan 初始化包括验证层（validation layer）属性和实例对象（VkInstance）创建的初始化。 一旦创建实例，就要检查现有系统上可用的物理设备（VkPhysicalDevice）。 选择预期的物理设备，并借助实例对象创建相应的逻辑设备（VkDevice）。 在 Vulkan 编程中，逻辑设备用于大多数代表物理设备的逻辑表示的 API 中。



Vulkan 通过错误 (error) 和验证层 (validation layers) 提供调试功能。 有两种类型的扩展：

- 特定于实例 Instance-specific：提供了全局级的扩展
- 特定于设备 Device-specific：提供了物理设备特定的扩展



在开始时，会列举系统中的全局层 (global layers) 以及设备特定 (device-specific) 的扩展；这些由 Vulkan 驱动程序公开。 可以将全局层和扩展注入到要在全局级别（global level）启用的实例对象中。 但是，仅在设备级别启用扩展会使其仅在该特定设备上有效。

初始化负责创建实例 (instance) 和设备对象 (device objects)。 另外，还会查询全局层 / 扩展 (global layers/extensions) ，并在全局级别或实例级别启用它们。 同样，扩展在特定设备上启用。 以下是伪代码的初始化过程：

1. **枚举 Instance Layer 属性**：Vulkan 首先与加载程序通信并找到驱动程序。 该驱动程序公开了很多扩展 extensions 和层 layers，这些扩展和层可能随着每个新驱动的安装或不同的 GPU 供应商不同而有所差异。 vkEnumerateInstanceLayerProperties 检索层的数量及其属性。 每个层可能包含多个扩展，它们都是可以使用 vkEnumerateInstanceExtensionProperties 进行查询的：

```
/*** 1. Enumerate Instance Layer properties ***/


// Get number of instance layers
uint32_t instanceLayerCount;

// Use second parameter as NULL to return the layer count
vkEnumerateInstanceLayerProperties(&instanceLayerCount, NULL);

VkLayerProperties *layerProperty = NULL;
vkEnumerateInstanceLayerProperties(&instanceLayerCount,
layerProperty);

// Get the extensions for each available instance layer
foreach layerProperty{
VkExtensionProperties *instanceExtensions;
res = vkEnumerateInstanceExtensionProperties(layer_name, &instanceExtensionCount,
instanceExtensions);
}
```

1. **实例 Instance 的创建**：实例对象 (VkInstance) 是使用 vkCreateInstance ( ) API 创建的，参数指定了要启用的、用于验证或调试目的的层名和扩展名。 这些名称在 VkInstanceCreateInfo 结构中指定：

```
/*** 2. Instance Creation ***/


// Vulkan instance object
VkInstance instance;
VkInstanceCreateInfo instanceInfo   = {};

// Specify layer names that needs to be enabled on instance.
instanceInfo.ppEnabledLayerNames    = {
"VK_LAYER_LUNARG_standard_validation", "VK_LAYER_LUNARG_object_tracker" };

// Specify extensions that needs to be enabled on instance.
instanceInfo.ppEnabledExtensionNames = {
VK_KHR_SURFACE_EXTENSION_NAME, VK_KHR_WIN32_SURFACE_EXTENSION_NAME};

// Create the Instance object
vkCreateInstance(&instanceInfo, NULL, &instance);
```

1. **设备 Device 的创建**：枚举现有系统上的物理设备或 GPU 的数量 ------- 通过 vkEnumeratePhysicalDevices ( ) API：

```
/*** 3. Enumerate physical devices ***/

VkPhysicalDevice    gpu;        // Physical device
uint32_t gpuCount;       // Pysical device count
vector<VkPhysicalDevice>gpuList;    // List of physical devices
// Get number of GPU count
vkEnumeratePhysicalDevices(instance, &gpuCount, NULL);

// Get GPU information
vkEnumeratePhysicalDevices(instance, &gpuCount, gpuList);
```

对于每个物理设备，我们会按照实例创建期间使用的、同样方式枚举特定于设备的扩展。

注意

对于基于实例的枚举，请使用 vkEnumerateInstanceLayerProperties 和 vkEnumerateInstanceExtensionProperties API。 但是，基于设备的层的枚举已经弃用了；因此，可以使用扩展的 vkEnumerateDeviceExtensionProperties 进行枚举。

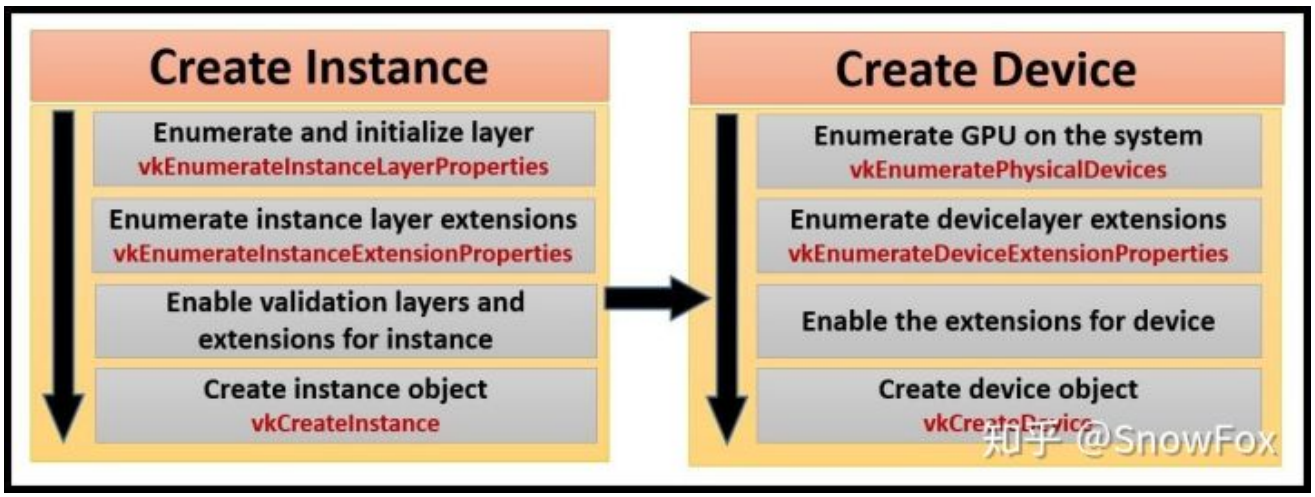
通过手中的物理设备列表，就可以查询以下信息：

- 队列 Queue 和队列类型 queue types：使用 `vkGetPhysicalDeviceQueueFamilyProperties()` API 查询可用物理设备的队列和队列属性。 在查询的队列中，搜索具有图形处理功能的队列并将其队列族索引 queue family index 存储在应用程序中供以后使用。 选择图形队列 graphics queue 是因为我们只对绘图操作感兴趣。
- 内存 Memory 信息：`vkGetPhysicalDeviceMemoryProperties()` API 检索目标物理设备上可用的内存类型。
- 物理设备属性 Physical device properties：或者，您可以存储物理设备的属性，以便在编程时检索某些特定的信息。 这可以使用 `vkGetPhysicalDeviceProperties()` API 来完成。

设备对象 device object 是使用 `vkCreateDevice()` API 创建的。 这是应用程序空间中物理设备的逻辑表示。 从现在开始，程序会在很多地方使用设备对象 device object ：

```
/*** 4. Create Device ***/


// Get Queue and Queue Type
vkGetPhysicalDeviceQueueFamilyProperties(gpu,
&queueCount, queueProperties);

// Get the memory properties from the physical device or GPU
vkGetPhysicalDeviceMemoryProperties(gpu, &memoryProperties);

// Get the physical device or GPU properties
vkGetPhysicalDeviceProperties(gpu, &gpuProps);

// Create the logical device object from physical device
VkDeviceCreateInfo deviceInfo = {}; vkCreateDevice(gpuList[0],&deviceInfo, NULL, &device);
```

下图以备忘表格的形式总结了创建 Vulkan 实例和设备的方法；你可以参考，将其作为这些过程的快速回顾：



Swapchain 初始化 - 查询 WSI 扩展

展示层 presentation 负责在输出窗口中显示渲染内容。 为此，我们需要一个空的窗口，这样就可以把我们绘图的图像填充到其中。 使用 CreateWindowEx（Windows）或 xcb_create_window（Linux）API 创建一个空窗口。

展示层 presentation 首先需要使用基于实例 instance 和基于设备 device 的 WSI 扩展 API 进行初始化。 这些 API 允许您使用各种表面属性 surface properties 创建展示表面 presentation surface。

注意

这些 API 必须动态链接并在应用程序中作为函数指针进行存储。 使用 `vkGetInstanceProcAddr()` API

- **创建一个抽象表面对象 abstract surface object**：表面创建的第一件事是创建 VkSurfaceKHR 对象。 该对象抽象了本机平台（Windows，Linux，Wayland，Android 等）窗口 / 表面机制（windowing/surface mechanisms）。 该对象是使用 vkCreate SurfaceKHR（ ）API 创建的。
- **使用带有展示能力的图形队列 Using a graphics queue with the presentation**：使用创建的抽象表面对象 abstract surface object 并通过 `vkGetPhysicalDeviceSurfaceSupportKHR()` API 搜索能够支持展示功能的图形队列 graphics queue。

注意

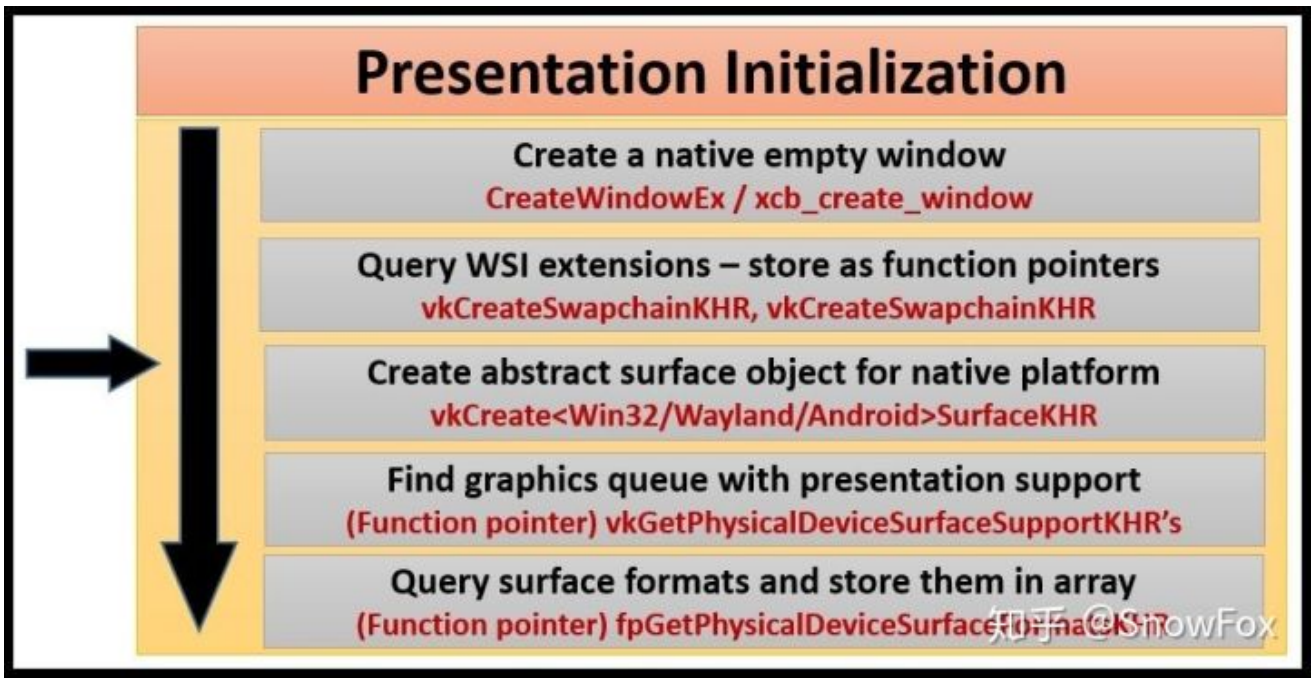
存储此搜索过的队列的句柄或索引。 稍后，它将用于查询其表面属性并创建此队列的逻辑对象（下一步）。

- **获取兼容队列 compatible queue**：在开始记录任何类型的命令缓冲区 command buffer 之前，必须获取队列 queue，用来提交命令缓冲区 command buffer。 使用 `vkGetDeviceQueue()` API 并指定我们在上一步中查询到的兼容队列的句柄或索引。
- **查询表面格式 surface formats**：使用 `vkGetPhysicalDeviceSurfaceFormatsKHR` API 检索物理设备支持的所有公开的表面格式：

```
/*** 5. Presentation Initialization ***/


// Create an empty Window CreateWindowEx(...);  /*Windows*/ xcb_create_window(...); /*Linux*/
// Query WSI extensions,store it as function pointers. For example:

// vkCreateSwapchainKHR, vkCreateSwapchainKHR .....



// Create an abstract surface object
```

```
VkWin32SurfaceCreateInfoKHR createInfo = {};
vkCreateWin32SurfaceKHR(instance, &createInfo, NULL, &surface);

// Among all queues, select a queue that supports presentation
foreach Queue in All Queues{ vkGetPhysicalDeviceSurfaceSupportKHR
(gpu, queueIndex, surface, &isPresentationSupported);
// Store this queue's index
if (isPresentationSupported) { graphicsQueueFamilyIndex = Queue.index; break;
}
}

// Acquire compatible queue supporting presentation

// and is also a graphics queue
vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &queue);


// Allocate memory for total surface format count
uint32_t formatCount; vkGetPhysicalDeviceSurfaceFormatsKHR
(gpu, surface, &formatCount, NULL);

VkSurfaceFormatKHR *surfaceFormats = allocate memory
(formatCount * VkSurfaceFormatKHR);

// Grab the surface format into VkSurfaceFormatKHR objects
vkGetPhysicalDeviceSurfaceFormatsKHR
(gpu, surface, &formatCount, surfaceFormats);
```

下图简要介绍了展示层 presentation 初始化的概况：



命令缓冲区 Command buffer 初始化 - 分配命令缓冲区 command buffers

在我们开始创建展示表面 presentation surface 之前，我们需要有命令缓冲区 command buffers。 命令缓冲区记录命令并将它们提交给兼容的队列进行处理。

命令缓冲区初始化包括以下内容：

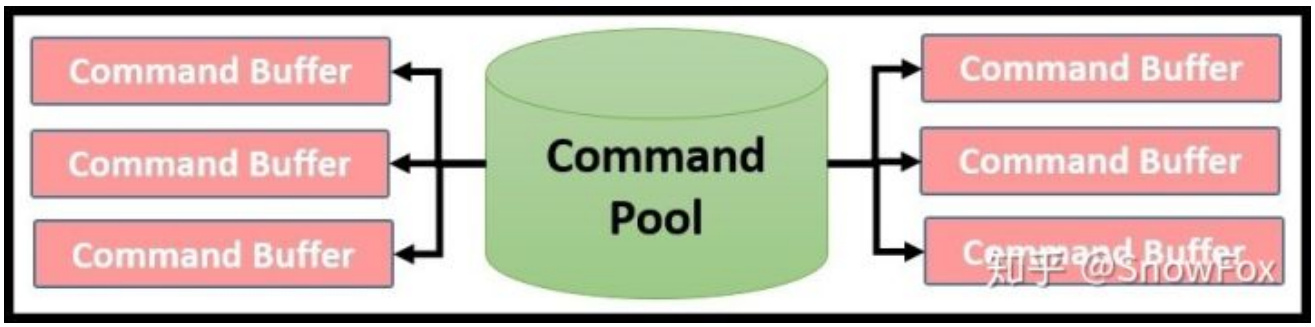
命令池 Command pool 的创建：请记住，我们保存了支持展示功能 presentation 的兼容图形队列 graphics queue 的句柄。 现在我们将通过 `vkCreateCommandPool()` API 并利用这个索引或句柄来创建一个与此队列族兼容的命令池 command pool。

- **分配命令缓冲区 command buffer**：命令缓冲区可以使用 `vkAllocateCommandBuffers()` API 从创建的命令池中进行简单的分配。

注意

如果要重复使用的话，则不必为每个帧都从命令池 command pool 分配命令缓冲区 command buffers。 如果不再需要现有的命令缓冲区，则可以高效地重用它们。

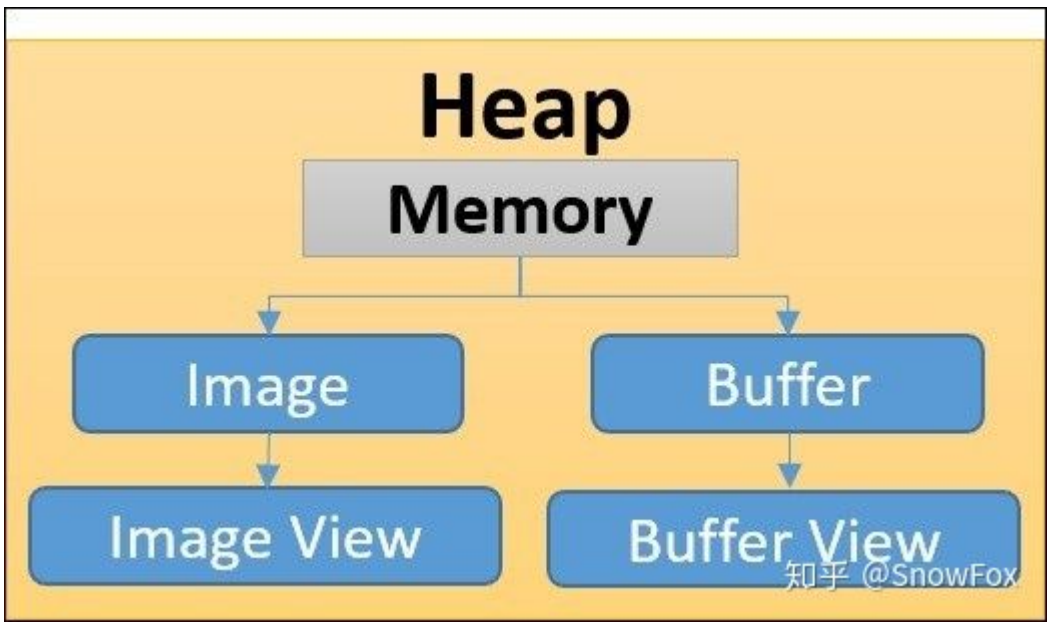
命令缓冲池 command buffer pool 用于分配内存区域以此来创建命令缓冲区 command buffer，而不用引入全局同步：



资源对象 Resource objects - 管理图像 images 和缓冲区 buffers

了解 Vulkan 中资源类型的概念是非常重要的。 从现在开始，我们会经常处理资源管理。 资源管理包括资源的创建、分配以及绑定。 例如，展示表面 presentation surface 本身就像处理其他普通的 Vulkan 资源类型一样处理绘图表面 drawing surface。

Vulkan 将资源分为两种类型，**缓冲区 Buffer** 和**图像 Image** ，如下图所示：



这些资源会进一步划分为视图 view; 让我们了解这些术语：

- **缓冲区 Buffer**：缓冲区对象 buffer object 使用用线性的数组类型表示资源。 缓冲区对象是 VkBuffer 类型的，并且使用 `vkCreateBuffer()` API 进行创建。 该 API 使用 VkBufferCreateInfo 结构作为参数输入，此参数指定了在创建对象期间可以使用的各种属性。 例如，您可以指定图像的平铺、图像的使用方式、尺寸大小、队列兼容性等。 现在我们来看看是什么构成了缓冲区视图：
- **缓冲区视图 Buffer view**：缓冲区视图 buffer view（VkBufferView）表示数据缓冲区 data buffer 本身。 它用于以连续的方式容纳数据，按照特定的数据解释格式排列。 它可以在 `vkCreateBufferView()` API 的帮助下进行创建。 接受一个 VkBufferViewCreateInfo 结构作为输入参数，其中可以指定各种缓冲区特定的属性，例如它的缓冲区对象 buffer object（VkBuffer）、格式、缓冲区视图的范围等。
- **图像 Image**：这是通过 VkImage 以编程方式来表示的。 该对象存储一维到三维的缓冲区数组 buffer arrays。 此对象是使用 `vkCreateImage()` API 创建的。 与缓冲区对象 buffer object 类似，此 API 使用 VkImageCreateInfo 结构在对象创建期间指定各种属性。 现在让我们看看图像视图是什么：
- **图像视图 Image view**：与缓冲区视图 buffer view 类似，图像视图对象 image view object 的类型为 VkImageView。 使用 `vkCreateImageView()` API 和 VkImageViewCreateInfo 结构来创建图像视图对象 image view object。

注意

应用程序并不会直接使用缓冲区（VkBuffer）和图像（VkImage）对象，相反，它依赖于它们各自的视图：VkBufferView 和 VkImageView。

让我们快速回顾一下。 到目前为止，我们已经创建了一个 Vulkan 实例 instance、一个表示我们物理设备 physical device 的逻辑设备 logical device，并且我们已经查询了队列属性 queue properties 并存储了支持展示功能 presentation 的队列族索引 queue family index。 我们为 WSI 扩展创建了函数指针并且理解了 Vulkan 资源类型。我们还从命令池 command pool 初始化并创建了我们的命令缓冲区 command buffers。

这其中涵盖了启动我们命令缓冲区记录 command buffer recording 过程所需的全部内容。

提示

命令缓冲区中应该记录些什么内容呢？

a）为交换链和深度 / 模板测试构建绘制图像和深度图。

b）创建着色器模块，用于与着色器程序相关联。

c）**利用描述符集和管线布局将资源绑定到着色器。**

d）**创建并管理 Render Pass 和 framebuffer 对象。**

e) 绘图操作。

使用 `vkBeginCommandBuffer()` API 开始命令缓冲区的记录。 它定义了命令缓冲区的起始范围；此后，任何指定的命令都会被记录在命令缓冲区中。

现在，我们来学习如何创建交换链。 在这一部分，我们将从交换链中获取绘制图像，以便进行渲染：

1. **获取表面的功能**：使用 `vkGetPhysicalDeviceSurfaceCapabilitiesKHR()` API 查询表面的功能，例如当前尺寸、可能的最小 / 最大尺寸、可能的转换功能等等。
2. **获取表面呈现模式 surface presentation modes**：呈现模式 presentation mode 告诉我们如何更新绘图表面 drawing surface，例如，它是以 即时模式更新 还是 垂直空白依赖模式更新 `vertical blank dependent` 等等。 呈现模式 presentation modes 可以使用 `vkGetPhysicalDeviceSurfacePresentModesKHR()` API 检索。
3. **创建交换链 swapchain**：使用表面的功能 surface capabilities 与呈现模式 presentation modes 一起创建交换链对象。 这些特性以及许多其他的参数（如尺寸大小、表面格式等）都在 VkSwapChainCreateInfo 结构中

指定，该结构被传递给 `vkCreateSwapchainKHR()` API 用来创建交换链对象。

4. **检索交换链图像**：查询交换链返回的图像表面的数量，并使用 `vkGetSwapchainImagesKHR()` API 检索相应的图像对象（VkImage）。 例如，**如果交换链支持双缓冲，那么它应该返回两个图像，并且可以对这两个图像进行绘图。**

注意

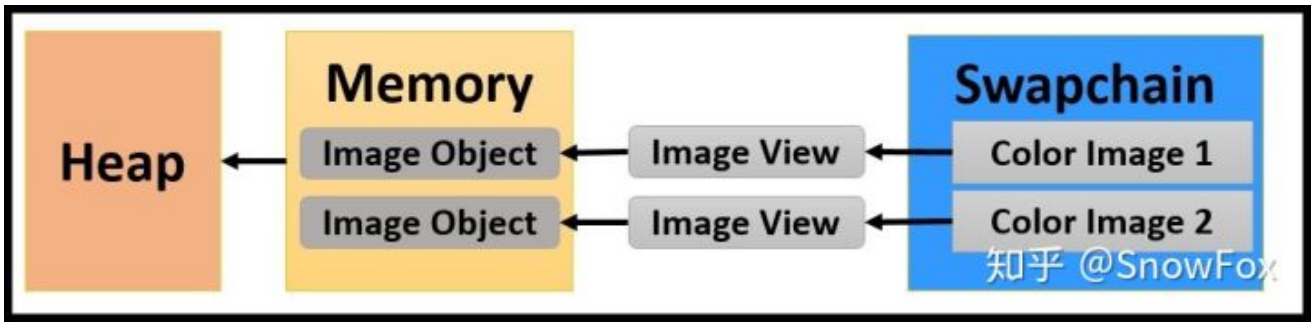
对于交换链图像，应用程序一端不需要进行内存分配。 在内部，交换链已经处理了内存的分配并返回烘焙后的对象。 应用程序只需要**指定如何通过图像视图使用图像**就可以了。 图像视图描述了图像的使用方式。

1. **设置图像布局**：对于每个图像，设置实现兼容的布局并添加**管线屏障。** 根据 Vulkan 规范，管线屏障会在一组命令之间插入执行依赖和一组内存依赖；首先插入命令缓冲区，然后是在命令缓冲区中插入若干个命令，这可以使用 `vkCmdPipelineBarrier()` API 完成这项操作。 通过插入屏障，可以保证图像视图在应用程序使用它之前可以在指定的布局中使用。
2. **创建图像视图**：由于应用程序只能使用 VkImageView 对象，因此使用 `vkCreateImageView()` 创建 VkImageView 对象。 保存视图对象以供应用程序后续使用：

```
/*** 6. Creating Swapchain ***/


//Start recording commands into command buffer
vkBeginCommandBuffer(cmd, &cmdBufInfo);

// Getting surface capabilities
vkGetPhysicalDeviceSurfaceCapabilitiesKHR
(gpu, surface, &surfCapabilities);

// Retrieve the surface presentation modes
vkGetPhysicalDeviceSurfacePresentModesKHR(gpu, surface, &presentModeCount, NULL);
VkPresentModeKHR presentModes[presentModeCount];
vkGetPhysicalDeviceSurfacePresentModesKHR(gpu, surface, &presentModeCount, presentModes);

// Creating the Swapchain
VkSwapchainCreateInfoKHR swapChainInfo = {};
fpCreateSwapchainKHR(device, &swapChainInfo, NULL, &swapChain);

// Create the image view of the retrieved swapchain images
vkGetSwapchainImagesKHR(device, swapChain, &swapchainImageCount, NULL);
VkImage swapchainImages[swapchainImageCount];
vkGetSwapchainImagesKHR(device, swapChain, &swapchainImageCount, swapchainImages);

// Retrieve the Swapchain images
foreach swapchainImages{

// Set the implementation compatible layout
SetImageLayout( . . .)

// Insert pipeline barrier
VkImageMemoryBarrier imgMemoryBarrier = { ... };
vkCmdPipelineBarrier(cmd,srcStages,destStages,0,0,NULL,0,NULL,1,&imgMemoryBarrier);

// Insert pipeline barrier

vkCreateImageView(device, &colorImageView, NULL, &scBuffer.view);
```

```
// Save the image view for application use
buffers.push_back(scBuffer);
}
```

下图显示了如何以图像视图对象（VkImageView）的形式使用 swapbuffer 图像对象（VkImage）：



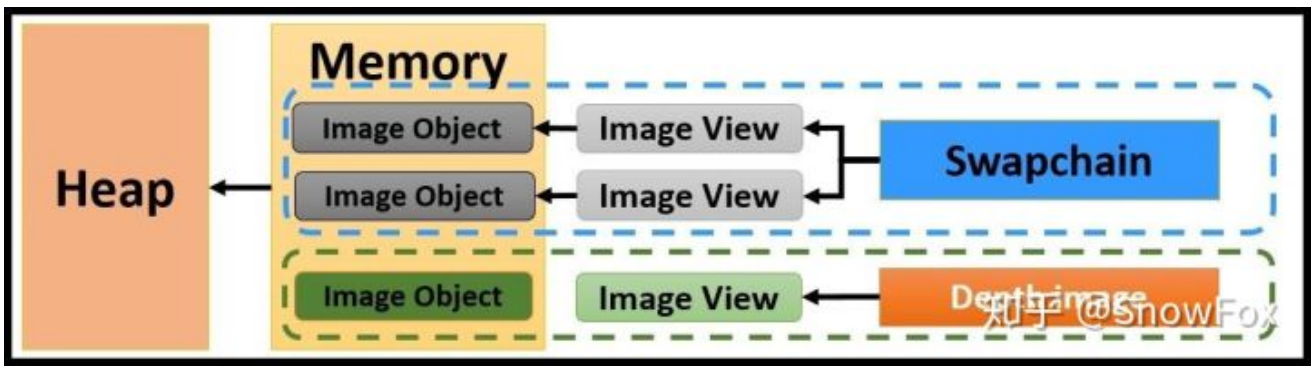
创建深度图

如果应用程序打算使用深度测试，则需要深度图。 对于 2D 绘图逻辑来说，只要有交换链图像就足够了。 深度图的创建过程与交换链图像相同。 但是有一个区别：不同于交换链图像，它们是现成的（由 `vkGetPhysicalDeviceSurfaceFormatsKHR()` 返回），而深度图对象（VkImage）由应用程序手动分配和创建的。

以下是深度图的创建过程：

1. 首先，使用 `vkGetPhysicalDeviceFormatProperties()` API 查询物理设备的格式属性 ，深度图会使用到这些属性。
2. 使用 `vkCreateImage()` API 创建一个图像对象，并使用 `vkGetImageMemoryRequirements()` API 获取所需资源的内存需求。
3. 接下来，使用检索到的内存需求属性、通过 `vkAllocateMemory()` API 分配内存。 通过调用 `vkBindImageMemory()` API 将分配的内存绑定到创建的图像对象。
4. 与交换链中的绘图图像类似，设置适当的图像布局并根据应用程序的要求创建图像视图。 有关设备内存分配的更多详细信息，请参阅下一节“资源分配 - 分配以及绑定设备内存”。

参考下图， 创建了一个新分配的深度图（VkImage）并将其连接到它对应的视图类型（VKImageView），其对象驻留在内存中：



以下伪代码说明深度图对象的创建过程，此深度图会用于深度测试目的：

```
/*** 7. Creating Depth image ***/

// Query supported format features of the physical device
```

```
vkGetPhysicalDeviceFormatProperties(gpus,depthFormat,&properties);

// Create an image object
vkCreateImage(device, &imageInfo, NULL, &imageObject);

// Get the memory requirements for an image resource
vkGetImageMemoryRequirements(device, image, &memRequirements);

// Allocate memory
vkAllocateMemory(device, &memAlloc, NULL, &memorys);

// Bind memory
vkBindImageMemory(device, imageObject, mem, 0);

// Set the implementation compatible layout
SetImageLayout(. . .)

// Insert a pipeline barrier to ensure that specified image
// layout are created before it being used further
vkCmdPipelineBarrier(cmd, srcStages, destStages, 0, 0, NULL,
0, NULL, 1, &imgPipelineBarrier);

// Create an Image View
vkCreateImageView(device, &imgViewInfo, NULL, &view);
```

资源分配 - 分配以及绑定设备内存

首次创建时，Vulkan 资源（**对于缓冲区来说就是 VkBuffer，对于图像来说就是 VkImage**）没有任何关联的后备内存，即没有分配实际的物理存储空间。 因此在使用资源之前，我们需要为其分配内存并将资源绑定到内存。

为了分配 Vulkan 资源对象，首先应用程序需要使用 `vkGetPhysicalDeviceMemory-Properties()` 来查询物理设备上的可用内存。 此 API 公开了一个或多个堆，并进一步暴露了这些堆中的一种或多种内存类型。 这些暴露的属性存储在内存控制结构（VkPhysicalDeviceMemoryProperties）中。 对于一个典型的 PC 用户来说，它会暴露两个堆：系统 RAM 和 GPU RAM。 此外，这些堆中的每一个都会根据其内存类型进行分类。

注意

特定于内存属性的查询，比如堆类型，可以在应用程序初始化期间进行，并在应用程序级对其进行缓存以供后续使用。

现在，这些内存类型中的每一种都可能拥有各种各样的需要从物理设备查询的属性。 例如，某些内存类型可能是 CPU 可见或不可见的；它们可以在 CPU 和 GPU 访问之间保持一致性，缓存的或未缓存的，等等。 这样的查询允许应用程序选择适合其需求的正确内存类型，以下是 Vulkan 中一般应用程序用于资源分配的典型过程：

- **内存要求**：资源对象（VkBuffer 和 VkImage）是根据其对象属性（例如平铺模式、使用标志等）来创建的。现在，每个对象都可能有不同的内存需求，这就需要通过调用 `vkGetBufferMemoryRequirements()` 或 `vkGetImageMemoryRequirements()` 来查询。 这有助于计算分配空间的大小；例如，返回的大小要处理填充对齐等。 它会考虑与该资源兼容的特定内存类型的位掩码。
- **分配**：内存使用 `vkAllocateMemory()` API 来分配。 它接受设备对象（VkDevice）以及内存控制结构（VkPhysicalDeviceMemoryProperties）作为输入参数。
- **绑定**：我们已经得到了能够帮助我们获得正确内存类型的内存需求，使用这个信息，我们就可以分配内存了。 现在我们可以使用 `vkBindBufferMemory()` 或 `vkBindImageMemory()` API 将资源对象绑定到分

配的内存。

- **内存映射**：内存映射简而言之，就是如何更新物理设备内存的内容。 首先，使用 `vkMapMemory( )` 将设备内存映射到主机内存。 更新此映射后的内存区域上的内容（在主机端）并调用 `vkUnmapMemory( )` API。 该 API 使用更新后的映射内存的内容更新设备内存的内容。

提供着色器 - 将着色器编译为 SPIR-V 格式

使用 glslangValidator.exe（LunarG SDK 中的工具）编译着色器文件，将它们从可读的文本格式转换为 SPIR-V 格式，这种格式是 Vulkan 可以理解的二进制中间格式：

```
// VERTEX SHADER
#version 450

layout (location = 0) in vec4 pos; layout (location = 1) in vec4 inColor; layout (location = 0)
out vec4 outColor;

out gl_PerVertex { vec4 gl_Position;
};

void main() {
outColor    = inColor; gl_Position      = pos; gl_Position.y = -gl_Position.y;
gl_Position.z = (gl_Position.z + gl_Position.w) / 2.0;
}

// FRAGMENT SHADER
#version 450

layout (location = 0) in vec4 color; layout (location = 0) out vec4 outColor;

void main() { outColor = color;
};
```

以下的伪代码显示了在应用程序中创建着色器模块的过程。 通过调用 vkCreateShaderModule（ ）API 创建给定着色器（顶点，片段，几何型，表面细分等）的着色器模块。 要想使用该 API，需要提供 SPIR-V 格式的中间二进制着色器代码， 这是在 VkShaderModuleCreateInfo 控制结构中进行指定的：

```
/*** 8. Building shader module ***/

VkPipelineShaderStageCreateInfo vtxShdrStages   = { };
VkShaderModuleCreateInfo moduleCreateInfo   = { };

// spvVertexShaderData contains binary form of vertex shader
moduleCreateInfo.pCode = spvVertexShaderData;

// Create Shader module on the device
vkCreateShaderModule
(device, &moduleCreateInfo, NULL, &vtxShdrStages.module);
```

绑定布局 - 描述符和管线布局

> 描述符通过布局绑定槽将资源与着色器连接起来。 它通常用于将 uniform 和 sampler 资源类型与着色器连接。

多个描述符布局绑定可以存在于单个描述符集中；它们会以块或数组的形式出现，如下面的伪代码所示。 然后将这些块绑定到单个控制结构 `VkDescriptorSetLayoutCreateInfo` 中，并利用这个结构，通过调用 `vkCreateDescriptorSetLayout()` API 创建描述符布局对象。 **描述符集布局表示描述符集包含的信息的类型。**

虽然创建了描述符布局，但是目前还不能被底层管线所访问。 为了提供访问权限，我们需要创建一个管线布局。管线布局是管线能够访问描述符集信息的手段。 它是通过调用 `vkCreatePipelineLayout()` API 创建的，该 API 使用到了 VkPipelineLayoutCreateInfo 控制结构对象，其中 *包含了上述的描述符布局* ：

```
/*** 9. Creating descriptor layout and pipeline layout ***/


// Descriptor layout specifies info type associated with shaders
VkDescriptorSetLayoutBinding layoutBind[2];////vertex阶段和fragment阶段

layoutBind[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
layoutBind[0].binding = 0;
layoutBind[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT; //vertex阶段

layoutBind[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
layoutBind[1].binding   = 0;
layoutBind[1].stageFlags    = VK_SHADER_STAGE_FRAGMENT_BIT; //fragmeent阶段

// Use layout bindings and create a descriptor set layout VkDescriptorSetLayoutCreateInfo
descriptorLayout = {}; descriptorLayout.pBindings = layoutBind;
VkDescriptorSetLayout descLayout[2];
vkCreateDescriptorSetLayout(device, &descriptorLayout, NULL, descLayout.data());

// Now use the descriptor layout to create a pipeline layout VkPipelineLayoutCreateInfo
pipelineLayoutCI = { ... }; pipelineLayoutCI.pSetLayouts = descLayout.data();
vkCreatePipelineLayout(device, &pipelineLayoutCI, NULL, &pipelineLayout);
```

注意

本章中的示例仅使用属性（顶点位置和颜色）。 它不使用任何 uniform 或采样器 sampler。 因此，在本章的这一点上，我们不需要定义描述符。 我们会在后面详细了解更多关于描述符集的内容，特别是第 10 章描述符和 push 常量。

创建渲染通道 - 定义通道属性

接下来，创建一个 Render Pass 对象。 渲染通道包含若干个子通道 `subpass` 和附件 `attachment` 。

> 它向驱动程序描述了绘图工作的结构，数据如何在各个附件之间流动或顺序要求是怎样的；以及运行时行为，比如每次加载时如何处理这些附件，或者是否需要清除或保存信息。

Render Pass 对象是通过调用 `vkCreateRenderPass()` API 创建的。 它接受 subpass 和附件控制结构作为参数。 有关更多信息，请参阅以下伪代码：

```
/*** 10. Render Pass ***/


// Define two attachment for color and depth buffer
VkAttachmentDescription attachments[2];
attachments[0].format = colorImageformat; attachments[0].loadOp = clear ?
VK_ATTACHMENT_LOAD_OP_CLEAR : VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[1].format = depthImageformat; attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;

VkAttachmentReference colorReference, depthReference = {...};

// Describe the subpass, use color image and depth image
VkSubpassDescription subpass = {};
subpass.pColorAttachments = &colorReference; subpass.pDepthStencilAttachment  =
&depthReference;

// Define RenderPass control structure
VkRenderPassCreateInfo rpInfo   = { &attachments,&subpass ...};

VkRenderPass renderPass; // Create Render Pass object
vkCreateRenderPass(device, &rpInfo, NULL, &renderPass);
```

帧缓冲区 - 将绘制图像连接到渲染通道

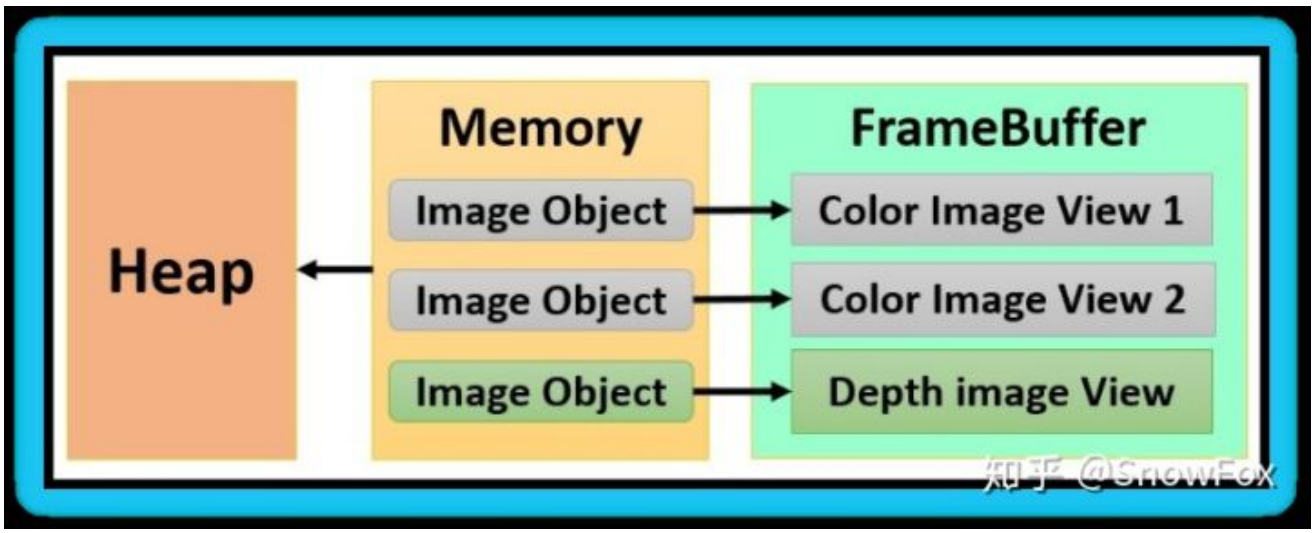
> 帧缓冲区是图像视图 (VKImageView) 的集合，对应于 Render Pass 中指定的附件 (Attachments)。 图像视图表示绘图图像 drawing image 或深度图。 Render Pass 对象用于控制这些附件，通过在创建 Render Pass 对象时指定的属性。

VkFramebufferCreateInfo 控制结构**接受 Render Pass 对象和附件以及其中的其他一些重要参数**，例如尺寸、附件数量、层等等。 该结构被传递给 `VkCreateFramebuffer()` API 来创建帧缓冲区对象。

注意

用于表示颜色和深度缓冲区的附件必须是图像视图（VKImageView），而不是图像对象（VkImage）。

下图显示了创建的帧缓冲区对象。 它包含用于绘制的、颜色缓冲区图像的图像视图以及用于深度测试的深度视图：



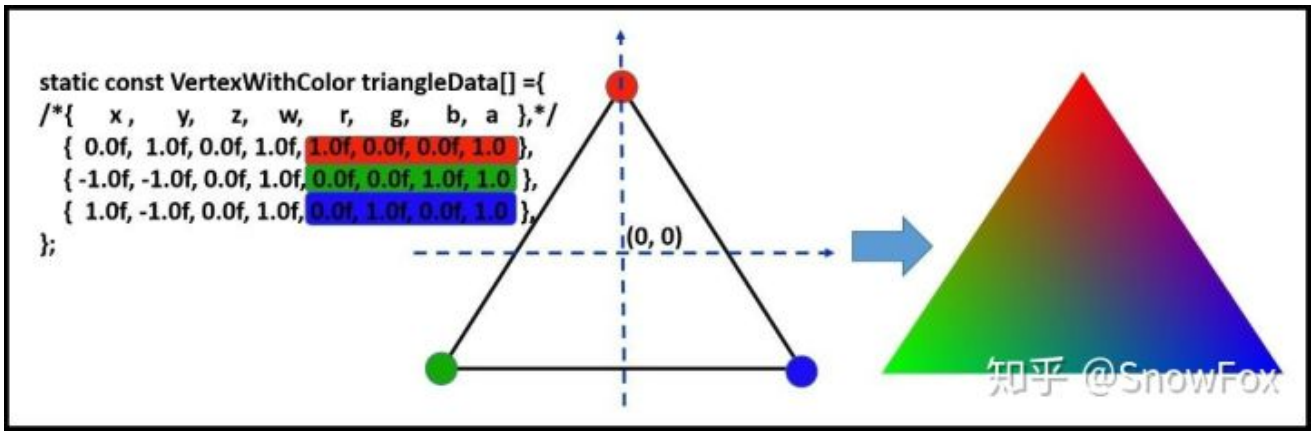
我们来看看创建帧缓冲区的过程，使用如下的伪代码来示范：

```
/*** 11. Creating Frame buffers ***/

VkImageView attachments[2]; // [0] for color, [1] for depth
attachments[1] = Depth.view; VkFramebufferCreateInfo fbInfo = {};
fbInfo.renderPass = renderPass;       // Pass render buffer object
fbInfo.pAttachments = attachments;  // Image view attachments
fbInfo.width = width;             // Frame buffer width
fbInfo.height = height;            // Frame buffer height
// Allocate memory for frame buffer objects, for each image

// in the swapchain, there is one frame buffer
VkFramebuffer framebuffers[number of draw imagein swap chain];

foreach (drawing buffer in swapchain) { attachments[0] = currentSwapChainDrawImage.view;
vkCreateFramebuffer(device, &fbInfo, NULL, &framebuffers[i]);
}
```

填充几何图形 - 将顶点存储到 GPU 内存中(vkMapMemory)

接下来，定义会出现在显示输出上的几何形状。 在本章中，我们使用了一个简单的三色三角形。

以下屏幕截图显示与此三角形关联的、交错存放的几何图形数据。 它包含了每个顶点的顶点位置，后面是颜色信息。 此数据数组需要通过 Vulkan 缓冲区对象（VkBuffer）提供给物理设备。



以下伪代码演示了缓冲区对象的分配，映射和绑定过程：

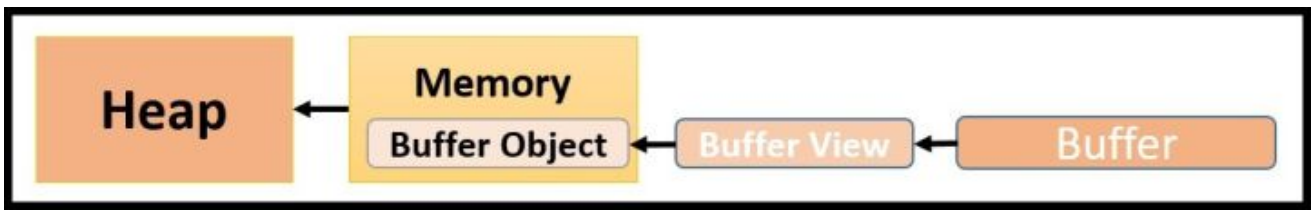
```
/***    12. Populate Geometry - storing vertex into GPU memory ***/

static const VertexWithColor triangleData[] ={
/*{ x,  y,  z,  w,  r,  g,  b,  a },*/
{   0.0f,   1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0 },
{ -1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0  },
{   1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0  },
};

VkBuffer  buffer;
VkMemoryRequirements  mem_requirement;
VkDeviceMemory  deviceMemory;
```

```
// Create buffer object, query required memory and allocate
VkBufferCreateInfo buffer_info = { ... }; vkCreateBuffer(device, &buffer_info, NULL, &buffer);

vkGetBufferMemoryRequirements(device, buffer, &mem_requirement);
VkMemoryAllocateInfo alloc_info = { ... }; vkAllocateMemory(device, &alloc_info, NULL, &
(deviceMemory));
// Copy the triangleData to GPU using mapping and unmapping.
uint8_t *pData;
vkMapMemory(device, deviceMemory, 0, mem_requirement.size, 0, &pData);
memcpy(pData, triangleData, dataSize); /**** Copying data ****/
vkUnmapMemory(device, deviceMemory);

// Bind the allocated memory
vkBindBufferMemory(device, buffer, deviceMemory, 0);
```

创建缓冲区资源的过程与图像对象的创建过程非常相似。 在这里，Vulkan 提供了用于分配，映射和绑定的、基于缓冲区的 API。 这与图像对象管理 API 非常相似。 下表显示了缓冲区和图像资源管理 API 及相关的数据结构：

缓冲区最初并不与任何类型的内存相关联。 应用程序必须先分配适当的设备内存并将其绑定到缓冲区，然后才能使用。 这一点与图像不同，必须使用图像视图强制创建图像才能够在应用程序中使用它们，而缓冲区对象则可以直接使用（如顶点属性，Uniform 等）。 如果需要在着色器阶段访问缓冲区对象，则必须以缓冲区视图对象的形式访问缓冲区对象。

VkBuffer	VkImageView
VkBufferCreateInfo vkCreateBuffer	vkCreateImageVkImageCreateInfo vkCreateImage
vkGetBufferMemoryRequirements	vkGetImageMemoryRequirements
vkBindBufferMemory	vkBindImageMemory
vkCreateBufferView	vkCreateImageView



> 一旦顶点数据上传到设备内存中，就必须通知管线，并对这些数据进行说明。 这将有助于检索和解释数据。例如，前面的几何图形顶点数据包括以交错方式存储的位置和颜色信息，并且每个属性是 16 字节宽。 这些信息需要通过**顶点输入绑定**（ `VkVertexInputBindingDescription` ）和**顶点输入属性描述符**（ `VkVertexInputAttributeDescription` ）控制结构来传递给底层的管线。

- `VkVertexInputBindingDescription` 包含了帮助管线读取缓冲区资源数据的一些属性，例如，考虑到要读取的信息的速率（无论是基于顶点还是基于多个实例），每个信息单元之间的跨度。
- `VkVertexInputAttributeDescription` 解释缓冲区资源数据。

在下面的伪代码中，位置和颜色属性在顶点着色器中的第 0 和第 1 位置处进行读取。 **由于数据是交错的形式，因此偏移量分别为 0 和 16：**

```
/***    13. Vertex binding ***/

VkVertexInputBindingDescription viBinding;
viBinding.binding    = 0;
viBinding.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
viBinding.stride = sizeof(triangleData) /*data Stride*/;

VkVertexInputAttributeDescriptionviAttribs[2];
viAttribs[0].binding = 0;
viAttribs[0].location   = 0;
viAttribs[0].format = VK_FORMAT_R32G32B32A32_SFLOAT; viAttribs[0].offset = 0;
viAttribs[1].binding    = 0;
viAttribs[1].location   = 1;
viAttribs[1].format = VK_FORMAT_R32G32B32A32_SFLOAT; viAttribs[1].offset = 16;
```

注意

控制结构对象 viAttribs 和 viBinding 会在管线创建时使用。 管线对象包含多个状态，其中顶点输入状态会消费有助于读取和解释缓冲区资源的对象。

管线状态管理 - 创建管线

> 管线是多个状态的集合。 每个状态都包含一组属性，它们定义了该状态的执行协议。 所有的这些状态共同生产了一条管线。

有两种类型的管线：

- **图形管线**(Graphics Pipeline)：该管线可以包含多个着色阶段，包括顶点着色、片段着色、镶嵌着色、几何图形着色等。 它有一个管线布局和多个固定功能的管线阶段。
- **计算管线**：用于计算操作。 它由单个静态的计算着色阶段和管线布局组成。

> 管线状态管理可以分为两个步骤。 第一步就是定义各种状态对象，其中包含一些重要的状态控制属性。 第二步，使用这些状态对象创建一个管线对象。

定义状态

管线可能会消耗多个状态，这些状态的定义如下所示：

- **动态状态**：动态状态用来通知管线在运行时期望更改的有关状态。 **这允许管线授权特定的例程更新相应的状态，而不是使用初始值。** 例如，视口和剪切是动态状态。 `VkPipelineDynamicStateCreateInfo` 结构指定应用程序中的所有动态状态及其属性。
- **顶点输入状态**：该状态有助于管线了解数据的读取和解析。 使用 `VkPipelineVertexInputStateCreateInfo` 对象并指定顶点输入绑定对象（VkVertexInputBindingDescription）和顶点输入属性描述符（VkVertexInputAttributeDescription）。
- **光栅化状态**：这是将图元转换为包含重要信息（例如颜色，深度和其他属性）的、二维图像的过程。 它由 `VkPipelineRasterizationStateCreateInfo` 结构表示；这个结构可以用剔除模式、正面方向、图元类型、线宽等指定。
- **颜色混合附件状态**：混合是源颜色和目标颜色的组合；这可以使用不同的属性和混合方程按照各种方式进行组合。 这是使用 `VkPipelineColorBlendStateCreateInfo` 结构表示的。

- **视口状态**：此状态有助于控制视口转换。 视口属性可以使用 `VkPipelineViewportStateCreateInfo` 来指定。 可能有各种各样的视口。 此状态有助于确定所选视口的重要属性，例如尺寸、起点、深度范围等。 对于每个视口，都有一个相应的裁剪矩形，用于定义裁剪测试的矩形边界。
- **深度模板状态**： `VkPipelineDepthStencilStateCreateInfo` 控制结构用于控制深度范围测试、模板测试和深度测试。
- **多重采样状态**：多采样状态包含了一些控制光栅化 Vulkan 图元（例如点，线和多边形）抗锯齿行为的重要属性。 `VkPipelineMultisampleStateCreateInfo` 控制结构可以用来指定这样的控制属性。
- 以下伪代码定义了各种管线状态对象，用于创建图形管线：

```
/*** 14. Defining states ***/


// Vertex Input state
VkPipelineVertexInputStateCreateInfo vertexInputStateInfo= {...};
vertexInputStateInfo.vertexBindingDescriptionCount = 1;
vertexInputStateInfo.pVertexBindingDescriptions       = &viBinding;
vertexInputStateInfo.vertexAttributeDescriptionCount  = 2;
vertexInputStateInfo.pVertexAttributeDescriptions  = viAttribs;

// Dynamic states
VkPipelineDynamicStateCreateInfo dynamicState   = { ... };

// Input assembly state control structure
VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo= { ... };

// Rasterization state control structure
VkPipelineRasterizationStateCreateInfo rasterStateInfo  = { ... };

// Color blend Attachment state control structure
VkPipelineColorBlendAttachmentState colorBlendSI    = { ... };

// Color blend state control structure
VkPipelineColorBlendStateCreateInfo colorBlendStateInfo = { ... };

// View port state control structure
VkPipelineViewportStateCreateInfo viewportStateInfo = { ... };

// Depth stencil state control structure
VkPipelineDepthStencilStateCreateInfo depthStencilStateInfo={..};

// Multisampling state control structure
VkPipelineMultisampleStateCreateInfo multiSampleStateInfo = {..};
```

创建图形管线

管线状态对象被打包到 `VkGraphicsPipelineCreateInfo` 控制结构中。 该结构提供了访问图形管线对象内部管线状态信息的手段。

管线状态对象的创建可能是一项昂贵的操作。 这是性能关键点之一。 因此，管线状态对象是从管线缓存（ `VkPipelineCache` ）创建的，以提供最大的性能。 **这允许驱动程序使用现有的基础管线创建新的管线。**

图形管线对象是使用 `vkCreateGraphicsPipelines()` API 创建的。 此 API 接受管线缓存对象，用于从中分配 VkPipeline 对象，并接受 VkGraphicsPipelineCreateInfo 对象指定与此管线连接的所有状态：

```
/***    15. Creating Graphics Pipeline ***/

// Create the pipeline objects
VkPipelineCache pipelineCache; VkPipelineCacheCreateInfo pipelineCacheInfo;
vkCreatePipelineCache(device, &pipelineCacheInfo, NULL, &pipelineCache);

// Define the control structure of graphics pipeline
VkGraphicsPipelineCreateInfo pipelineInfo;
pipelineInfo.layout = pipelineLayout; // 管线布局为了读取相应的描述符布局
pipelineInfo.pVertexInputState = &vertexInputStateInfo;
pipelineInfo.pInputAssemblyState = &inputAssemblyInfo;
pipelineInfo.pRasterizationState = &rasterStateInfo;
pipelineInfo.pColorBlendState  = &colorBlendStateInfo;
pipelineInfo.pMultisampleState  = &multiSampleStateInfo;
pipelineInfo.pDynamicState = &dynamicState;
pipelineInfo.pViewportState = &viewportStateInfo;
pipelineInfo.pDepthStencilState = &depthStencilStateInfo;
pipelineInfo.pStages = shaderStages;
pipelineInfo.stageCount = 2;
pipelineInfo.renderPass = renderPass;

// Create graphics pipeline
vkCreateGraphicsPipelines(device, pipelineCache, 1, &pipelineInfo, NULL, &pipeline);
```

执行渲染通道 - 绘制 Hello World！！！

在这个阶段，我们将在 Render Pass 阶段的帮助下在绘图表面渲染我们的简单三角形。 Render Pass 阶段的执行需要一个绘图表面和一组命令集的一个记录，该记录定义一个 Render Pass 的运行行为。

获取绘图表面

在开始渲染任何东西之前，我们需要的第一个东西就是绘图帧缓冲区。 我们已经创建了帧缓冲区对象并把交换链图形图像与其（它包含交换链图像视图）关联了起来。 现在，我们将使用 `vkAcquireNextImageKHR()` API 来确定绘图操作当前可用的绘图图像的索引。 使用获取到的索引，我们就可以引用相应的帧缓冲区并将其提供给 Render Pass 阶段用于渲染目的：

```
/***    16. Acquiring drawing image ***/


// Define semaphore for synchronizing the acquire of draw image.

// Only acquire draw image when drawing is completed
VkSemaphore imageAcquiredSemaphore;
VkSemaphoreCreateInfo imageAcquiredSemaphoreCI = {...};
imageAcquiredSemaphoreCI.sType=VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vkCreateSemaphore(device, &imageAcquiredSemaphoreCI, NULL, &imageAcquiredSemaphore);

// Get the index of the next available swapchain image:
vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAcquiredSemaphore, NULL,
&swapChainObjCurrentBuffer);
```

当使用两个或多个交换链绘制图像时，需要使用同步机制。 只有在绘制的图像已经在显示输出上呈现，并准备好接受下一项作业时才能获取绘图图像， 此状态由 `vkAcquireNextImageKHR()` 指示。 信号量对象可用于同步绘图图像的获取。 信号量（VkSemaphore）可以使用 `vkCreateSemaphore()` API 创建；这个对象将会在命令缓冲区提交中使用。

准备 Render Pass 控制结构

渲染通道需要一些特定的信息，例如帧缓冲区、渲染通道对象、渲染区域尺寸、清除颜色、深度模板值等。 这些信息使用 `VkRenderPassBeginInfo` 控制结构来指定。 此结构稍后用于定义渲染通道的执行。 以下伪代码会帮助读者详细了解此结构的用法：

```
/***    17. Preparing render pass control structure ***/


// Define clear color value and depth stencil values
const VkClearValue clearValues[2] = {
[0] = { .color.float32 = { 0.2f, 0.2f, 0.2f, 0.2f } },
[1] = { .depthStencil = { 1.0f, 0 } },
};

// Render pass execution data structure for a frame buffer
VkRenderPassBeginInfo beginPass;
beginPass.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO; beginPass.pNext= NULL;
beginPass.renderPass = renderPass;
beginPass.framebuffer =framebuffers[currentSwapchainImageIndex];
beginPass.renderArea.offset.x  = 0;
beginPass.renderArea.offset.y   = 0;
beginPass.renderArea.extent.width  = width; beginPass.renderArea.extent.height = height;
beginPass.clearValueCount = 2;
beginPass.pClearValues  = clearValues;
```

渲染通道的执行

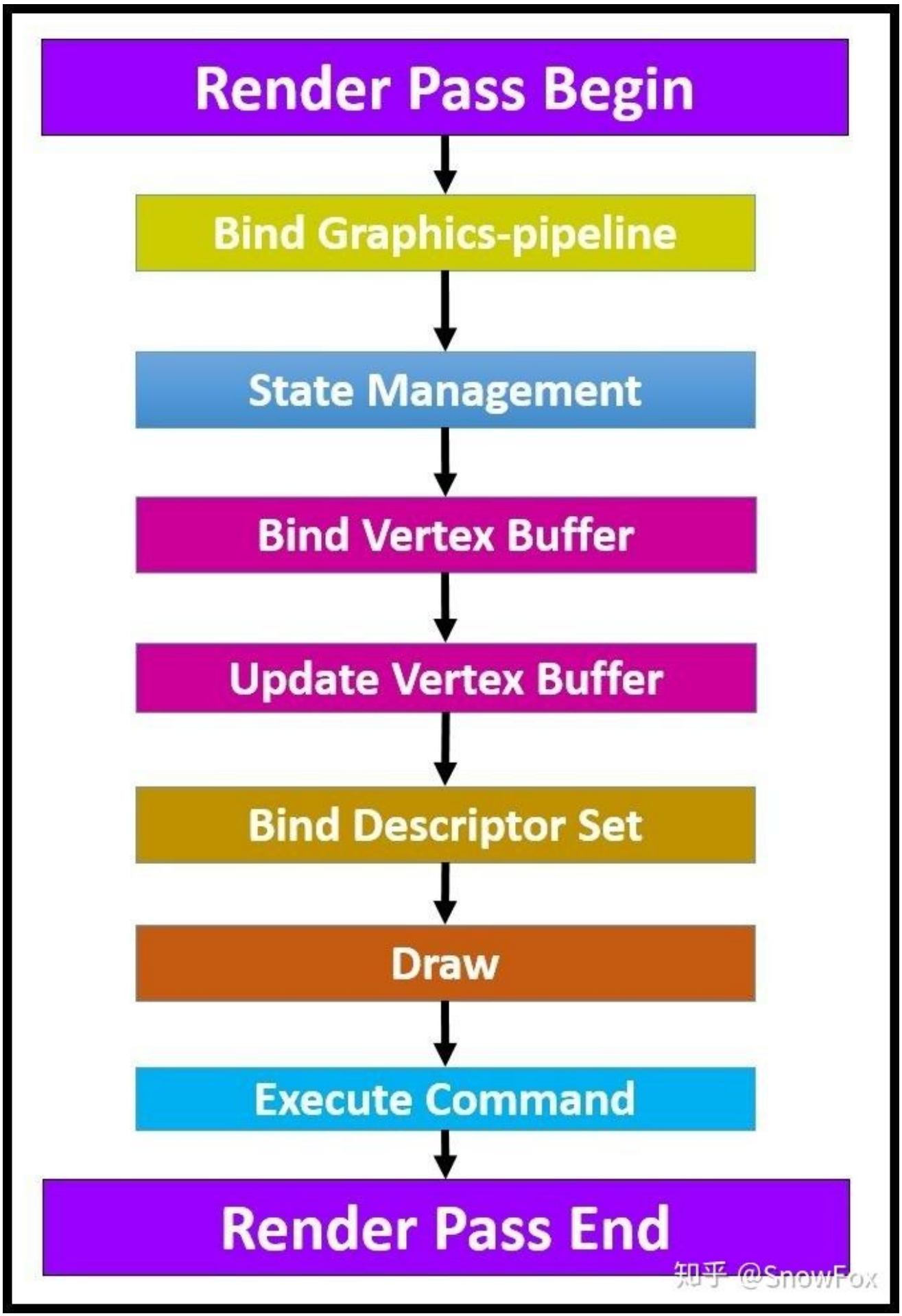
Render Pass 的执行在用户指定的范围内进行定义。 此范围分别使用由 `vkCmdBeginRenderPass()` 和 `vkCmdEndRenderPass()` API 定义的开始标记和结束标记来解释。 在该范围内，指定了以下命令，并自动链接到当前的 Render Pass：

1. 绑定管线：使用 `vkCmdBindPipeline()` 绑定图形管线。
2. 绑定几何图形缓冲区：使用 `vkCmdBindVertexBuffers()` API 将顶点数据缓冲区对象（类型为 VkBuffer）提供给 Render Pass。
3. 视口和裁剪：通过调用 `vkCmdSetViewport()` 和 `vkCmdSetScissor()` API 指定视口以及裁剪尺寸
4. Draw 对象： `vkCmdDraw()` API指定绘图命令，其中包含诸如需要从起始索引读取的顶点数量，实例数量等信息。

在完成命令缓冲区记录之前，要设置与实现兼容的图像布局并通过调用 `vkEndCommandBuffer()` 结束命令缓冲区的记录：

```
/**** START RENDER PASS ****/
vkCmdBeginRenderPass(cmd, &beginPass, VK_SUBPASS_CONTENTS_INLINE);

// Bind the pipeline
vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline); const VkDeviceSize
offsets[1] = { 0 };

// Bind the triangle buffer data
vkCmdBindVertexBuffers(cmd, 0, 1, &buffer, offsets);
// viewport = {0, 0, 500, 500, 0 ,1}
vkCmdSetViewport(cmd, 0, NUM_VIEWPORTS, &viewport);

// scissor  = {0, 0, 500, 500}
vkCmdSetScissor(cmd, 0, NUM_SCISSORS, &scissor);

// Draw command - 3 vertices, 1 instance, 0th first index
vkCmdDraw(cmd, 3, 1, 0, 0);

/**** END RENDER PASS ****/
vkCmdEndRenderPass(cmd);

// Set the swapchain image layout
setImageLayout(VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL . .);

/**** COMMAND BUFFER RECORDING ENDS HERE ****/
vkEndCommandBuffer(cmd);
```

下图显示了渲染通道的执行过程。 它突出显示了 Render Pass 范围内执行的操作。


Render Pass Begin
Bind Graphics-pipeline
State Management
Bind Vertex Buffer
Update Vertex Buffer
Bind Descriptor Set
Draw
Execute Command
Render Pass End
知乎 @SnowFox

队列提交和同步 - 发送作业

最后，我们使用若干命令（其中包括 Render Pass 信息和图形管线）成功记录了命令缓冲区。 命令缓冲区会被提交到队列中进行处理。 驱动程序会读取命令缓冲区并对其进行调度安排。

注意

> 通常会把命令缓冲区打包成批处理，以便进行高效渲染；因此，如果存在多个命令缓冲区，则需要将它们打包到一个 VkCommandBuffer 数组中。

在提交命令缓冲区之前，了解以前提交的批处理的状态很重要。 如果处理成功，那么将新的批处理压入队列才有意义。 Vulkan 提供栏栅（ `VkFence` ）作为同步机制，**以了解以前发送的作业是否已完成。** 使用 `vkCreateFence()` API 创建栏栅对象（VkFence）。 该 API 接受一个 VkFenceCreateInfo 控制结构。

命令缓冲区在提交对象（VkSubmitInfo）中进行指定。 该对象包含命令缓冲区列表以及一个 VkSemaphore 对象，用于同步带有若干交换链绘图图像的帧缓冲区。 这些信息会被输入到 `vkQueueSubmit()` API 中；其中包含一个 VkQueue 对象（命令缓冲区会被提交到其中）和一个 VkFence 对象（确保在每个命令缓冲区提交之间进行同步）：

```
VkFenceCreateInfo fenceInfo = { ... }; VkFence drawFence;
// Create fence forensuring completion of cmdBuffer processing
vkCreateFence(device, &fenceInfo, NULL, &drawFence);

// Fill the command buffer submission control sturctures
VkSubmitInfo submitInfo[1] = { ... }; submitInfo[0].pNext    = NULL;
submitInfo[0].sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo[0].pWaitSemaphores  = &imageAcquiredSemaphore;
submitInfo[0].commandBufferCount = 1;
submitInfo[0].pCommandBuffers  = &cmd;

// Queue the command buffer for execution
vkQueueSubmit(queue, 1, submitInfo, NULL);
```

使用展示层显示 --- 渲染三角形

一旦命令缓冲区被提交给队列，就会被物理设备异步处理。 因此，就会在交换链的绘图表面上渲染三色三角形。现在，该表面对用户是不可见的，并且需要在显示窗口上呈现出来。 绘图表面在 VkPresentInfoKHR 控制结构的帮助下显示出来。 绘图表面包含展示信息，例如应用程序中的交换链数量、需要检索的绘图图像的索引等。 此控制结构对象用作 `vkQueuePresentKHR` 中的参数。 这会把绘图表面图像翻转到显示窗口。

注意

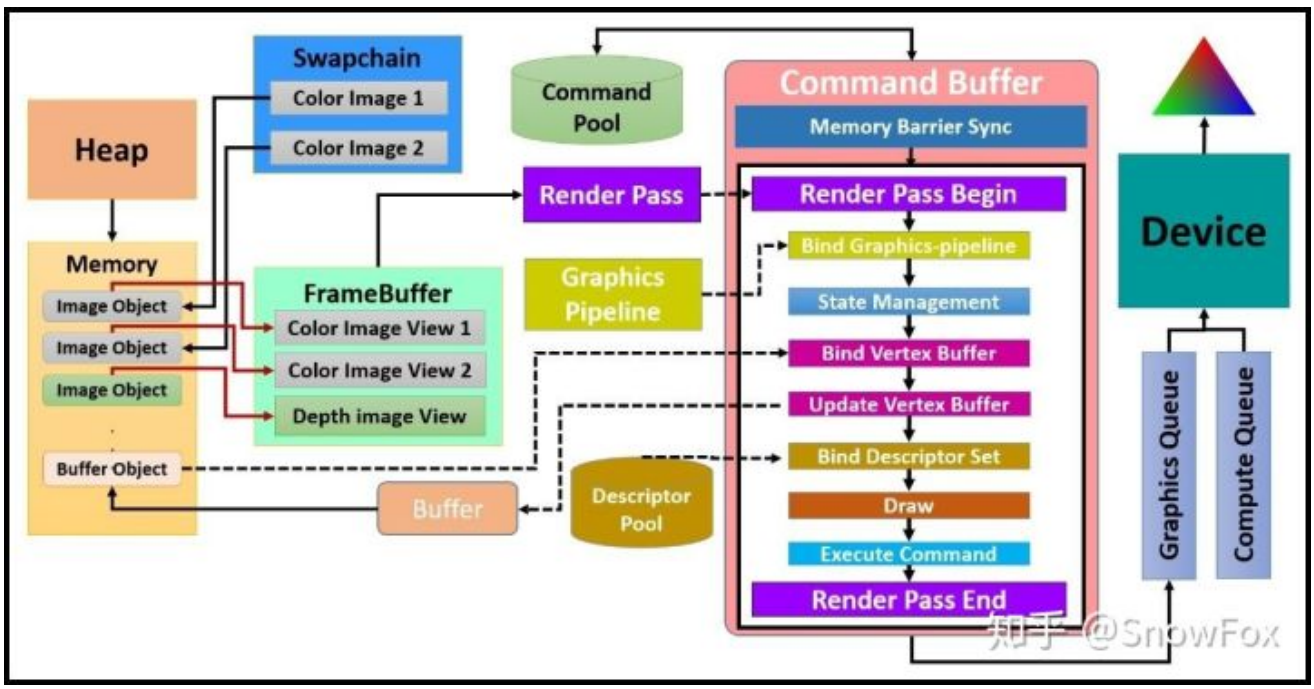
> 一旦调用 vkQueueSubmit，展示队列就可以在它执行展示操作之前等待，直到最后一次提交发出的 imageAcquiredSemaphore 信号量。

```
// Define the presentation control structure
VkPresentInfoKHR present    = { ... };
present.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
present.pNext = NULL;
present.swapchainCount = 1; present.pSwapchains = &swapChain;
present.pImageIndices = &swapChainObjCurrent_buffer;
```

```
// Check if all the submitted command buffers are processed
do { res=vkWaitForFences(device,1,&drawFence,VK_TRUE,FENCE_TIMEOUT);
} while (res == VK_TIMEOUT);

// Handover the swapchain image to presentation queue

// for presentation purpose
vkQueuePresentKHR(queue, &present);

// Destroy Synchronization objects
vkDestroySemaphore(device, imageAcquiredSemaphore, NULL); vkDestroyFence(device, drawFence,
NULL);
```

整合到一起

本节简要介绍我们的第一个 Vulkan 伪应用程序的工作原理。 下图是工作模型的快照：



首先，应用程序在初始阶段创建 Vulkan 实例和设备，启用必要的层并创建需要的扩展。 该设备暴露了各种队列（图形队列或计算队列），如上图所示。 这些队列会收集命令缓冲区并将它们提交给物理设备进行处理。

使用 WSI 扩展，准备绘制表面，用来渲染图形内容。 交换链会将这些绘图表面暴露为图像，这些图像以图像视图的形式使用。 类似地，准备深度图视图。 这些图像视图对象被帧缓冲区 framebuffer 所使用。 渲染通道使用这个帧缓冲区 framebuffer 定义一个渲染单元的操作。

命令缓冲区是从命令缓冲池中分配的，用于记录各种命令以及 Render Pass 执行过程。 如上图所示，Render Pass 的执行需要一些重要的 Vulkan 对象，例如图形管线、描述符集、着色器模块、管线对象和几何数据。

最后，命令缓冲区被提交给支持展示功能的队列，比如图形队列。 一旦提交，GPU 就会以异步方式进行处理。 可能需要一些同步机制和内存屏障才能使渲染输出无问题。

总结

在本章中，我们探讨了在系统上安装 Vulkan 的步骤。 然后我们使用伪代码编程“Hello World !!!”，在这个程序中，我们在显示窗口上渲染了一个三种颜色的三角形。

这个介绍性的章节已经将 Vulkan 的难度做了大大的简化，这样，了解这种图形 API 对于初学者来说也就是非常简单的事情了。 对于 Vulkan 编程来说，本章采用的是取巧代码；可以将其作为参考，用来帮助读者以正确的顺序记忆所有的编程步骤以及它们各自对应的 API。

亚里士多德说：“好的开始就是成功的一半！” 随着前两章的完成，我们为从 0 开始全面了解 Vulkan 机制奠定了坚实的基础；我们将会在后续的章节中继续完善其内容。

在下一章中，我们将深入研究核心编程，并开始构建我们的第一个 Vulkan 应用程序。 您将会了解到层和扩展，以及如何隐式和显式地启用它们。 我们还会研究 Vulkan 实例、设备以及队列的基本原理，这对于与 GPU 进行通信非常有用。 一旦我们开始编程，我们就需要查询 GPU 暴露的资源和设施。 我们还将学习如何获得队列及其暴露的属性。