目录

第一章 没有任何秘密的 API

简介第 0 部分：前言

作者：Pawel L. (Intel)，更新时间： 2016 年 9 月 2 日

关于作者

我成为软件开发人员已有超过 9 年的时间。 我最感兴趣的领域是图形编程，大部分工作主要涉及 3D 图形。 我在 OpenGL* 和着色语言（主要是 GLSL 和 Cg）方面拥有丰富的经验。三年来我还一直致力于开发 Unity* 软件。 我也曾有机会投身于涉及头盔式显示器（比如 Oculus Rift*） 或类似 CAVE 系统的 VR 项目。

最近，我正与英特尔的团队成员一起准备验证工具，为被称为 Vulkan 的新兴 API 提供显卡驱动程序支持。 图形编程接口及其使用方法对我来说非常新鲜。 在了解这些内容的时候我突然想到，我可以同时准备有关使用 Vulkan 编写应用的教程。 我可以像那些了解 OpenGL 并希望“迁移”至其后续产品的人一样，分享我的想法和经验。

关于 Vulkan

Vulkan 被视作是 OpenGL 的后续产品。 它是一种多平台 API，可支持开发人员准备游戏、CAD 工具、性能基准测试等高性能图形应用。 它可在不同的操作系统（比如 Windows*、Linux* 或 Android*）上使用。 Vulkan 由科纳斯组织创建和维护。 Vulkan 与 OpenGL 之间还有其他相似之处，包括图形管道阶段、OpenGL 着色器（一定程度上），或命名。

但也存在许多差异，但这进一步验证了新 API 的必要性。 20 多年来，OpenGL 一直处于不断变化之中。 自 90 年代以来，计算机行业发生了巨大的变化，尤其是显卡架构领域。 OpenGL 库非常适用，但仅依靠添加新功能以匹配新显卡功能并不能解决一切问题。 有时需要完全重新设计。 因此创建出了 Vulkan。

Vulkan 基于 Mantle* — 第一个全新的低级别图形 API。 Mantle 由 AMD 开发而成，专为 Radeon 卡架构而设计。 尽管是第一个公开发布的 API，但使用 Mantle 的游戏和基准测试均显著提升了性能。 后来陆续发布了其他低级别 API，比如 Microsoft 的 DirectX* 12、Apple 的 Metal*，以及现在的 Vulkan。

传统图形 API 和全新低级别 API 之间有何区别？ OpenGL 等高级别 API 使用起来非常简单。 开发人员只需声明操作内容和操作方式，剩下的都由驱动程序来完成。 驱动程序检查开发人员是否正确使用 API 调用、是否传递了正确的参数，以及是否充分准备了状态。 如果出现问题，将提供反馈。 为实现其易用性，许多任务必须由驱动程序在“后台”执行。

在低级别 API 中，开发人员需要负责完成大部分任务。 他们需要符合严格的编程和使用规则，还必须编写大量代码。 但这种做法是合理的。 开发人员知道他们的操作内容和希望实现的目的。 但驱动程序不知道，因此使用传统 API 时，驱动程序必须完成更多工作，以便程序正常运行。 采用 Vulkan 等 API 可避免这些额外的工作。 因此 DirectX 12、Metal 或 Vulkan 也被称为精简驱动程序/精简 API。 大部分时候它们仅将用户请求传输至硬件，仅提供硬件的精简抽象层。 为显著提升性能，驱动程序几乎不执行任何操作。

低级别 API 要求应用完成更多工作。 但这种工作是不可避免的， 必须要有人去完成。 因此由开发人员去完成更加合理，因为他们知道如何将工作分成独立的线程，图像何时成为渲染对象（颜色附件）或用作纹理/采样器等等。 开发人员知道管道处于何种状态，或哪些顶点属性变化的更频繁。 这样有助于提高显卡硬件的使用效率。 最重要的原因是它行之有效。 我们能够观察到显著的性能提升。

但“能够”一词非常重要。 它要求完成其他工作，但同时也是一种合适的方法。 在有一些场景中，我们将观察到，OpenGL 和 Vulkan 之间在性能方面没有任何差别。 如果不需要多线程化，或应用不是 CPU 密集型（渲染的场景不太复杂），使用 OpenGL 即可，而且使用 Vulkan 不会实现性能提升（但可能会降低功耗，这对移动设备至关重要）。 但如果我们想最大限度地发挥图形硬件的功能，Vulkan 将是最佳选择。

主要显卡引擎迟早会支持部分（如果不是所有）新的低级别 API。 如果希望使用 Vulkan 或其他 API，无需从头进行编写。 但通常最好对“深层”信息有所了解，因此我准备这一教程。

源代码说明

我是 Windows 开发人员 如果有选择，我选择编写面向 Windows 的应用。 因为我在其他操作系统方面没有任何经验。 但 Vulkan 是多平台 API，而且我希望展示它可用于不同的操作系统。 因此我们准备了一个示例项目，可在 Windows 和 Linux 上编译和执行。

关于本教程的源代码，请访问：

https://github.com/GameTechDev/IntroductionToVulkan

我曾尝试编写尽可能简单的代码示例，而且代码中不会掺杂不必要的“#ifdefs”。 但有时不可避免（比如在窗口创建和管理过程中），因此我们决定将代码分成几个小部分：

- **Tutorial** 文件，是这里最重要的一部分。 与 Vulkan 相关的所有代码都可放置在该文件中。 每节课都放在一个标头/源配对中。
- **OperatingSystem** 标头和源文件，包含依赖于操作系统的代码部分，比如窗口创建、消息处理和渲染循环。 这些文件包含面向 Linux 和 Windows 的代码，不过我试着尽可能地保持统一。
- **main.cpp** 文件，每节课的起点。 由于它使用自定义 Window 类，因此不包含任何特定于操作系统的代码。
- **VulkanCommon** 标头/源文件，包含面向从教程 3 之后各教程的基本课程。 该类基本上重复教程 1 和 2 — 创建 Vulkan 实例和渲染图像和其他所需的资源，以在屏幕上显示图像。 我提取了这一准备代码，因此其他章节的代码可以仅专注于所介绍的主题。
- **工具**，包含其他实用程序函数和类，比如读取二进制文件内容的函数，或用于自动破坏对象的包装程序类。

每个章节的代码都放置在单独的文件夹中。 有时可包含其他数据目录，其中放置了用于某特定章节的资源，比如着色器或纹理。 数据文件夹应拷贝至包含可执行文件的相同目录。 默认情况下可执行文件将编译成构建文件夹。
没错。 编译和构建文件夹。 由于示例项目可在 Windows 和 Linux 上轻松维护，因此我决定使用 CMakeLists.txt 文件和 CMake 工具。 Windows 上有一个 build.bat 文件，可创建 Visual Studio* 解决方案 — （默认情况下）Microsoft Visual Studio 2013 需要编译 Windows 上的代码。 我在 Linux 上提供了一个 build.sh 脚本，可使用 make 编译代码，但使用 Qt 等工具也可轻松打开 CMakeLists.txt。当然还需要 CMake。
生成解决方案与项目文件，而且可执行文件将编译至构建文件夹。 该文件夹也是默认的工作目录，因此数据文件夹应拷贝至该目录，以便课程正常运行。
执行过程中如果出现问题，cmd/terminal 中将“打印”其他信息。 如果出现问题，将通过命令行/终端运行课程，或检查控制台/终端窗口，以查看是否显示了消息。
我希望这些说明能够帮助大家了解并跟上 Vulkan 教程的节奏。 现在我们来重点学习 Vulkan！

请前往： [没有任何秘密的 API： Vulkan* 简介第 1 部分： 序言]

目录

教程 1：Vulkan* – 序言

我们从不显示任何内容的简单应用开始。 因为教程太长，因此本文不展示完整的源代码（以及窗口、渲染循环等）。 大家可以访问 https://github.com/GameTechDev/IntroductionToVulkan，在提供的示例中获取包含完整源代码的示例项目。 此处我仅展示与 Vulkan 相关的部分代码。

在应用中使用 Vulkan API 的方法有多种：

1. 可以动态加载驱动程序的库来提供 Vulkan API 实施，并自己获取它提供的函数指示器。
2. 可以使用 Vulkan SDK 并链接至提供的 Vulkan Runtime (Vulkan Loader) 静态库。
3. 可以使用 Vulkan SDK，在运行时动态加载 Vulkan Loader 库，并通过它加载函数指示器。

不建议使用第一种方法。 硬件厂商可以任意修改它们的驱动程序，从而可能影响与特定应用的兼容性。 甚至还会破坏应用，并要求开发人员编写支持 Vulkan 的应用，以覆写部分代码。 这就是为什么最好使用部分抽象层的原因。

建议使用 Vulkan SDK 的 Vulkan Loader。它能够提供更多配置功能和更高的灵活性，无需修改 Vulkan 应用源代码。有关灵活性的一个示例是层。Vulkan API 要求开发人员创建严格遵守 API 使用规则的应用。如果出现错误，驱动程序几乎不会提供反馈，仅报告部分严重且重大的错误（比如内存不足）。因为使用该方法，所以 API 本身能够尽可能的小、快。 但如果我们希望获得更多有关哪些地方出错的信息，那么必须启用调试/验证层。 不同的层级用途各不相同，比如内存使用、相应参数传递、对象寿命检查等等。 这些层级都会降低应用的性能，但会为我们提供更多信息。

我们还需选择是静态链接至 Vulkan Loader ，还是动态加载并在运行时由我们自己获取函数指示器。 选择哪一种只是个人喜好问题。 本文将重点介绍第三种使用 Vulkan 的访问，从 Vulkan Runtime 库动态加载函数指示器。 该方法与我们希望在 Windows* 系统上使用 OpenGL* 时的做法类似，采用该方法时，默认实施仅提供部分基础函数。 剩下的函数必须使用 wglGetProcAddress() 或标准窗口 GetProcAddress() 函数动态加载。 这就是创建 GLEW 或 GL3W 等 wrangler 库的对象。

加载 Vulkan Runtime 库并获取导出函数指示器

在本教程中，我们将逐步介绍如何自己获取 Vulkan 函数指示器。 我们从 Vulkan Runtime 库 (Vulkan Loader) 加载这些指示器，该运行时库应与支持 Vulkan 的显卡驱动程序一同安装。 面向 Vulkan 的动态库 (Vulkan Loader) 在 Windows* 和 Linux* 上分别命名为 vulkan-1.dll 和 libvulkan.so。

从现在起，我引用第一个教程的源代码，重点为 Tutorial01.cpp 文件。 因此在应用的初始化代码中，我们需要使用如下代码加载 Vulkan 库：

```
#if
defined(VK_USE_PLATFORM_WIN32_KHR)

VulkanLibrary = LoadLibrary( "vulkan-1.dll" );

#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)

VulkanLibrary = dlopen( "libvulkan.so", RTLD_NOW );

#endif
```

```
if( VulkanLibrary == nullptr ) {
  std::cout << "Could not load Vulkan library!" << std::endl;
  return false;
}
return true;
```

1.*Tutorial01.cpp, function LoadVulkanLibrary()*

VulkanLibrary 是 Windows 中类型 HMODULE 的变量，或者是 Linux 中 void* 的变量。 如果加载函数的库返回的值不是 0，那么我们可以加载所有导出的函数。 Vulkan 库和 Vulkan 实施（不同厂商的驱动程序）都要求仅展示一个可通过操作系统拥有的标准技巧（比如之前提到的 Windows 中的 GetProcAddress() 或 Linux 中的 dlsym()）加载的函数。 Vulkan API 的其他函数可能也能通过该方法获取，但无法保证（甚至不建议）。 必须导出的一个函数是 **vkGetInstanceProcAddr()**。

该函数用于加载其他所有 Vulkan 函数。 为减轻获取所有 Vulkan API 函数地址的工作负担，最便利的方法是将它们的名称放在宏中。 这样我们不用在多个位置（比如定义、声明或加载）复制函数名称，并且能够将它们放在一个标头文件中。 这种单一文件日后也可通过 #include 指令用于不同的用途。 我们可以像下面这样声明导出的函数：

```
#if !defined(VK_EXPORTED_FUNCTION)
#define VK_EXPORTED_FUNCTION( fun )
#endif
```

```
VK_EXPORTED_FUNCTION( vkGetInstanceProcAddr )

#undef VK_EXPORTED_FUNCTION
```

2.*ListOfFunctions.inl*

现在我们定义能够展示 Vulkan API 的函数的变量。 这可通过以下命令来实现：

```
#include "vulkan.h"

namespace ApiWithoutSecrets {

#define VK_EXPORTED_FUNCTION( fun ) PFN_##fun fun;
#define VK_GLOBAL_LEVEL_FUNCTION( fun ) PFN_##fun fun;
#define VK_INSTANCE_LEVEL_FUNCTION( fun ) PFN_##fun fun;
#define VK_DEVICE_LEVEL_FUNCTION( fun ) PFN_##fun fun;
```

```
#include "ListOfFunctions.inl"

}
```

3.*VulkanFunctions.cpp*

这里我们首先包含 vulkan.h 文件，它正式提供给希望在应用中使用 Vulkan API 的开发人员。 该文件与 OpenGL 库中的 gl.h 文件类似。 它定义开发 Vulkan 应用时所必须的所有枚举、结构、类型和函数类型。 接下来定义来自各“级”（稍后将具体介绍这些级）的函数的宏。 函数定义要求提供函数类型和函数名称。 幸运的是，Vulkan 中的函数类型可从函数名称轻松派生出来。 例如，**vkGetInstanceProcAddr()** 函数的类型定义如下：

```
typedef PFN_vkVoidFunction (VKAPI_PTR *PFN_vkGetInstanceProcAddr)(VkInstance instance, const char* pName);
```

4.*Vulkan.h*

展示该函数的变量定义如下：

```
PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr;
```

这是 VulkanFunctions.cpp 文件的宏进行扩展的目标。 它们提取函数名称（隐藏在“fun”参数中）并在开头部分添加“PFN_”。 然后，宏在类型后面放置一个空格，并在之后添加函数名称和分号。 函数“粘贴”至符合 #include “ListOfFunctions.inl” 指令的文件。

但我们必须牢记，如果希望自己定义 Vulkan 函数的原型，那么必须定义 VK_NO_PROTOTYPES 预处理器指令。 默认情况下，vulkan.h 标头文件包含所有函数的定义。 这将有助于静态链接至 Vulkan Runtime。 因此当我们添加自己的定义时，将会出现编译错误，声明特定变量（面向函数指示器）已定义多次（因为我们打破了“一种定义”规则）。 我们可以使用之前提到的预处理器宏禁用 vulkan.h 文件的定义。

同样，我们需要声明 VulkanFunctions.cpp 文件中定义的变量，以便它们显示在代码的其他部分。 这可通过相同的方法完成，但“extern”需要放在各函数的前面。 比较 VulkanFunctions.h 文件。

现在我们有了可用来保存从 Vulkan 库中获取的函数地址的变量。 为了只加载一个导出的函数，我们使用以下代码：

```
#if defined(VK_USE_PLATFORM_WIN32_KHR)
#define LoadProcAddress GetProcAddress
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
#define LoadProcAddress dlsym
#endif

#define VK_EXPORTED_FUNCTION( fun )                                   \
if( !(fun = (PFN_##fun)LoadProcAddress( VulkanLibrary, #fun )) ) {            \
  std::cout << "Could not load exported function: " << #fun << "!" << std::endl;   \
  return false;                                                       \
}

#include "ListOfFunctions.inl"
```

```
return true;
```

5.*Tutorial01.cpp, function LoadExportedEntryPoints()*

宏从“fun”参数中提取函数名称，将其转化成字符串（带 #）并从 VulkanLibrary 中获取它的地址。 地址可通过（Windows 上的） GetProcAddress() 或（Linux 上的） dlsym() 获取，并保存在 fun 呈现的变量中。 如果操作失败而且库没有显示函数，我们通过打印相应信息并返回假值来报告该问题。 宏在通过 ListOfFunctions.inl 包含的行上运行。 这样我们就不用多次写入函数名称。

由于我们有主函数加载程序，因此可以加载剩下的 Vulkan API 程序。 它们可分为三类：

- 全局级函数。 支持创建 Vulkan 实例。
- 实例级函数。 检查可用的支持 Vulkan 的硬件以及显示的 Vulkan 特性。
- 设备级函数。 负责执行通常在 3D 应用中完成的工作（比如绘制）。

我们从获取全局级的实例创建函数开始。

获取全局级函数指示器

在创建 Vulkan 实例之前，我们必须获取支持创建工作的函数地址。 以下是函数列表：

- vkCreateInstance
- vkEnumerateInstanceExtensionProperties
- vkEnumerateInstanceLayerProperties

最重要的函数是 **vkCreateInstance()**，它支持我们创建“Vulkan 实例”。 从应用视角来看，Vulkan 实例相当于 OpenGL 的渲染环境。 它保存按照应用状态（Vulkan 中没有全局状态），比如启用的实例级层和扩展功能。 其他两种函数支持我们检查有哪些实例层和实例扩展功能可用。 验证层可根据它们调试的功能分成实例级和设备级。 Vulkan 中的扩展功能与 OpenGL 中的扩展功能类似：展示核心规范不需要的附加功能，而且并非所有硬件厂商都会实施这些功能。 扩展功能（比如层）也可分为实例级扩展功能和设备级扩展功能，不同层级的扩展必须单独启用。 在 OpenGL 中，所有扩展功能（通常）都在创建的环境中提供；使用 Vulkan 时，必须在它们展示的功能能够使用之前启用它们。

我们调用函数 **vkGetInstanceProcAddr()** 获取实例级程序的地址。 它提取两个参数：实例和函数名称。 我们还没有实例，因此第一个参数为“null”。 这就是为什么这些函数有时调用 null 实例或非实例级函数的原因。通过 **vkGetInstanceProcAddr()** 函数获取的第二个参数是我们希望获取地址的程序名称。我们可以只加载没有实例的全局级函数。 不能加载其他第一个参数中未提供实例句柄的函数。

加载全局级函数的代码如下所示：

```
#define VK_GLOBAL_LEVEL_FUNCTION( fun )                                    \
if( !(fun = (PFN_##fun)vkGetInstanceProcAddr( nullptr, #fun )) ) {         \
  std::cout << "Could not load global level function: " << #fun << "!" << std::endl;   \
  returnfalse;                                                             \
}

#include "ListOfFunctions.inl"

return true;
```

6.*Tutorial01.cpp, function LoadGlobalLevelEntryPoints()*

该代码与用于加载导出函数（库展示的 **vkGetInstanceProcAddr()**）的代码之间唯一不同点在于，我们不使用操作系统提供的函数（比如 GetProcAddress()），而是调用第一个参数设为 null 的 **vkGetInstanceProcAddr()**。

如果您按照本教程自己编写代码，务必将包含在合理命名的宏中的全局级函数添加至 ListOfFunctions.inl 标头文件：

```
#if !defined(VK_GLOBAL_LEVEL_FUNCTION)
```

```
#define VK_GLOBAL_LEVEL_FUNCTION( fun )
#endif

VK_GLOBAL_LEVEL_FUNCTION( vkCreateInstance )
VK_GLOBAL_LEVEL_FUNCTION( vkEnumerateInstanceExtensionProperties )
VK_GLOBAL_LEVEL_FUNCTION( vkEnumerateInstanceLayerProperties )

#undef VK_GLOBAL_LEVEL_FUNCTION
```

7.*ListOfFunctions.inl*

创建 Vulkan 实例

加载全局级函数后，现在我们可以创建 Vulkan 实例。 可以通过调用拥有三个参数的 **vkCreateInstance()**函数完成。

- 第一个参数包含有关应用、请求的 Vulkan 版本，以及我们希望启用的实例级层和扩展功能的信息。 这都可以通过结构完成（结构在 Vulkan 中非常普遍）。
- 第二个参数为结构指示器提供与内存分配相关的函数列表。 它们可用于调试，但该特性是可选的，而且我们可以依赖内置的内存分配方法。
- 第三个参数是我们希望保存 Vulkan 实例句柄的变量地址。 在 Vulkan API 中，操作结果通常保存在我们提供地址的变量中。 返回值仅用于一些通过/未通过通知。 以下是有关实例创建的完整源代码：

```
VkApplicationInfo application_info = {
```

```
    VK_STRUCTURE_TYPE_APPLICATION_INFO,          // VkStructureType          sType
    nullptr,                                     // const void               *pNext
    "API without Secrets: Introduction to Vulkan",  // const char            *pApplicationName
    VK_MAKE_VERSION( 1, 0, 0 ),                  // uint32_t                 applicationVersion
    "Vulkan Tutorial by Intel",                  // const char               *pEngineName
    VK_MAKE_VERSION( 1, 0, 0 ),                  // uint32_t                 engineVersion
    VK_API_VERSION                               // uint32_t                 apiVersion
};

VkInstanceCreateInfo instance_create_info = {
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,      // VkStructureType          sType
    nullptr,                                     // const void*              pNext
    0,                                           // VkInstanceCreateFlags    flags
```

```
  &application_info,                    // const VkApplicationInfo    *pApplicationInfo
  0,                                    // uint32_t                   enabledLayerCount
  nullptr,                              // const char * const         *ppEnabledLayerNames
  0,                                    // uint32_t                   enabledExtensionCount
  nullptr                               // const char * const         *ppEnabledExtensionNames
};

if( vkCreateInstance( &instance_create_info, nullptr, &Vulkan.Instance ) != VK_SUCCESS ) {
  std::cout << "Could not create Vulkan instance!" << std::endl;
  return false;
}
return true;
```

8.*Tutorial01.cpp, function CreateInstance()*

大部分 Vulkan 结构的开头是描述结构类型的字段。 参数通过指示器提供给函数，以避免复制较大的内存块。 有时在结构内部，也会将指示器提供给其

他结构。 对于需要知道它应读取多少字节以及成员如何对齐的驱动程序来说，通常会提供结构类型。 那么这些参数到底有什么意义？

- sType – 结构类型。 在这种情况下，它通过提供 VK_STRUCTURE_TYPE_APPLICATION_INFO 的值，通知驱动程序我们将提供有关实例创建的信息。
- pNext – 未来版本的 Vulkan API 可能会提供有关实例创建的其他信息，该参数用于此目的。 目前它留作将来使用。
- flags – 另一个留作将来使用的参数；目前必须设为 0。
- pApplicationInfo – 包含应用信息（比如名称、版本、所需 Vulkan API 版本等）的另一结构的地址。
- enabledLayerCount – 定义我们希望启用的实例级验证层的数量。
- ppEnabledLayerNames – 包含我们希望启用的层级名称的 enabledLayerCount 要素阵列。
- enabledExtensionCount – 我们希望启用的实例级扩展功能数量。
- ppEnabledExtensionNames – 与层级一样，该参数必须指向至少包含我们希望使用的实例级扩展功能的名称的 enabledExtensionCount 要素的阵列。

大部分参数都可以设为 null 或 0。 最重要的一个参数（除结构类型信息外）是指向类型变量 VkApplicationInfo 的参数。 因此指定实例创建信息之前，我们还需要指定描述应用的另一变量。 该变量包含应用名称、正在使用的引擎名称，或所需的 Vulkan API 版本（与 OpenGL 版本类似，如果驱动程序不支持该版本，将无法创建实例）。 该信息可能对驱动程序非常有用。 请记住，一些显卡厂商会提供专门用于特定用途（比如特定游戏）的驱动程序。如果显卡厂商知道引擎游戏使用哪种显卡，将可以优化驱动程序的行为，从而加快游戏的速度。 该应用信息结构可用来实现这一目的。 VkApplicationInfo 结构的参数包括：

- sType – 结构类型。 此处为 VK_STRUCTURE_TYPE_APPLICATION_INFO，即有关应用的信息。
- pNext – 留作将来使用。
- pApplicationName – 应用名称。
- applicationVersion – 应用版本，使用 Vulkan 宏创建版本非常方便。 它包括主要版本和次要版本，并将数字合成一个 32 位的值。
- pEngineName – 应用使用的引擎名称。
- engineVersion – 我们在应用中使用的引擎版本。
- apiVersion – 我们希望使用的 Vulkan API 版本。 最好提供包含时在 Vulkan 标头中定义的版本，这就是我们为什么使用在 vulkan.h 标头文件中查找的 VK_API_VERSION。

定义了这两个结构后，现在我们可以调用 **vkCreateInstance()** 函数并检查实例是否已创建。 如果创建成功，实例句柄将保存在我们提供地址的变量中，并返回 VK_SUCCESS （0！）。

获取实例级函数指示器

我们已经创建了一个 Vulkan 实例。 接下来是获取函数指示器，以便创建逻辑设备，它可视作物理设备上的用户视图。 计算机上可能安装了许多支持

Vulkan 的不同设备。 每台设备都具备不同的特性、功能和性能，或者支持的功能也各不相同。 如果希望使用 Vulkan，那么必须指定用来执行操作的设备。 可以使用不同用途的设备（比如一台用于渲染 3D 图形、一台用于物理计算，另一台用于媒体解码）。 必须检查有多少设备，其中哪些可用，具备哪些功能，以及支持哪些操作。 这可通过实例级函数来完成。 我们使用之前用过的 **vkGetInstanceProcAddr()** 函数来获取这些函数的地址。 但这次要提供句柄，才能创建 Vulkan 实例。

使用 **vkGetInstanceProcAddr()** 函数加载每个 Vulkan 程序，且 Vulkan 实例句柄带有部分权衡。 将 Vulkan 用于数据处理时，必须创建一台逻辑设备并获取设备级函数。 但在运行应用的计算机上可能有许多支持 Vulkan 的设备。 确定使用哪台设备取决于前面提到的逻辑设备。 但 **vkGetInstanceProcAddr()** 无法认出逻辑设备，因为没有相应的参数。 使用该函数获取设备级程序时，事实上我们获取的是简单“jump”函数的地址。 这些函数提取逻辑设备的句柄，并跳至相应实施（为特定设备实施的函数）。 此次跳跃产生的开销是可以避免的。 建议使用其他函数单独加载每台设备的程序。 但仍然需要使用 **vkGetInstanceProcAddr()** 函数加载支持创建此类逻辑设备的函数。

部分实例级函数包括：

- vkEnumeratePhysicalDevices
- vkGetPhysicalDeviceProperties
- vkGetPhysicalDeviceFeatures
- vkGetPhysicalDeviceQueueFamilyProperties
- vkCreateDevice
- vkGetDeviceProcAddr
- vkDestroyInstance

这些函数是本教程用于创建逻辑设备所必需的。 但扩展功能也提供一些其他的实例级函数。 通过示例解决方案的源代码形成的标头文件中的列表将展开。 用于加载所有函数的源代码为：

```
#define VK_INSTANCE_LEVEL_FUNCTION( fun )                                    \
if( !(fun = (PFN_##fun)vkGetInstanceProcAddr( Vulkan.Instance, #fun )) ) {           \
  std::cout << "Could not load instance level function: " << #fun << "!" << std::endl;   \
```

```
  returnfalse;                                                \
}

#include "ListOfFunctions.inl"

return true;
```

9.*Tutorial01.cpp, function LoadInstanceLevelEntryPoints()*

用于加载实例级函数的代码与加载全局级函数的代码大体上相同。 我们只需将 **vkGetInstanceProcAddr()**函数的第一个参数从 null 改成创建 Vulkan 实例句柄。 当然我们还可以在实例级函数上运行，因此现在我们重新定义 VK_INSTANCE_LEVEL_FUNCTION() 宏，而非 VK_GLOBAL_LEVEL_FUNCTION() 宏。 我们还需定义实例级的函数。 像之前一样，最好通过共享标头中收集的包含宏的名称列表来完成，例如：

```
#if !defined(VK_INSTANCE_LEVEL_FUNCTION)
#define VK_INSTANCE_LEVEL_FUNCTION( fun )
#endif

VK_INSTANCE_LEVEL_FUNCTION( vkDestroyInstance )
```

```
VK_INSTANCE_LEVEL_FUNCTION( vkEnumeratePhysicalDevices )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceProperties )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceFeatures )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceQueueFamilyProperties )
VK_INSTANCE_LEVEL_FUNCTION( vkCreateDevice )
VK_INSTANCE_LEVEL_FUNCTION( vkGetDeviceProcAddr )
VK_INSTANCE_LEVEL_FUNCTION( vkEnumerateDeviceExtensionProperties )

#undef VK_INSTANCE_LEVEL_FUNCTION
```

10.*ListOfFunctions.inl*

实例级函数在物理设备上运行。 在 Vulkan 中可以看“物理设备”到和“逻辑设备”（简单称为设备）。 顾名思义，物理设备指安装在计算机上、运行支持 Vulkan 且能够执行 Vulkan 命令的应用的物理显卡（或其他硬件组件）。 如前所述，此类设备可以显示并实施不同（可选）Vulkan 特性，具备不同的功能（比如总内存，或能够处理不同大小的缓冲区对象），或提供不同的扩展功能。 此类硬件可以是专用（独立）显卡，也可以是内置（集成）于主处理器中的附加芯片。 甚至还可以是 CPU 本身。 实例级函数支持我们检查所有参数。 检查后，必须（根据检查结果和需求）决定使用哪台物理设备。 我们还希望使用多台设备（这也是可能的），但这种场景目前太过高级。 因此如果希望发挥物理设备的能力，那么我们必须创建一台逻辑设备，以呈现我们在应用中的选择（以及启用的层、扩展功能、特性等）。 创建设备（并获取队列）后，我们准备使用 Vulkan，方法与创建渲染环境后准备使用 OpenGL 的方法相同。

创建逻辑设备

创建逻辑设备之前，必须首先进行检查，看看系统中有多少可供执行应用的物理设备。 接下来获取所有可用物理设备的句柄：

```
uint32_t num_devices = 0;
if( (vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, nullptr ) != VK_SUCCESS) ||
   (num_devices == 0) ) {
  std::cout << "Error occurred during physical devices enumeration!" << std::endl;
  return false;
}

std::vector<VkPhysicalDevice> physical_devices( num_devices );
if( vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, &physical_devices[0] ) != VK_SUCCESS ) {
  std::cout << "Error occurred during physical devices enumeration!" << std::endl;
  return false;
}
```

11.*Tutorial01.cpp, function CreateDevice()*

如要检查有多少可用设备，可以调用 **vkEnumeratePhysicalDevices()** 函数。 调用两次，第一次调用时将最后一个参数设为 null。 这样驱动程序会知道我们仅要求知道可用物理设备的数量。 该数量将保存在我们在第二个参数中提供地址的变量中。

知道有多少可用物理设备后，我们可以准备保存它们的句柄。 我使用矢量，因此无需担心内存分配和取消分配问题。 再次调用 **vkEnumeratePhysicalDevices()** 时，所有参数不等于 null，我们将获取在最后一个参数中提供地址的阵列中的物理设备的句柄。 该阵列的大小可能与第一次调用后的返回值不同，但必须与在第二个参数中定义的要素数量相同。

例如，有 4 台可用物理设备，但我们只对第 1 台感兴趣。 因此在第一次调用后，在 num_devices 中设置一个为 4 的值。 这样我们将知道这里有任意兼容 Vulkan 的设备，然后继续。 我们将该值覆写成 1，因为无论有多少设备，我们只希望使用 1 台设备。 第二次调用后，我们将仅获取一个物理设备句柄。

提供的设备数量将由枚举的物理设备数量所取代（当然不会大于我们提供的值）。 例如，我们不希望两次调用这个函数。 我们的应用支持多达 10 台设备，并且我们提供该值和 10 要素静态阵列指示器。 驱动程序通常返回实际枚举的设备数量。 如果没有设备，我们提供的变量地址中将保存 0。 如果有这种设备，我们也会知道。 我们无法告知是否有超过 10 台设备。

由于我们有所有兼容 Vulkan 的物理设备的句柄，现在可以检查各设备的属性。 在示例代码中，这一过程在循环中完成：

```
VkPhysicalDevice selected_physical_device = VK_NULL_HANDLE;
uint32_t selected_queue_family_index = UINT32_MAX;
for( uint32_t i = 0; i < num_devices; ++i ) {
 if( CheckPhysicalDeviceProperties( physical_devices[i], selected_queue_family_index ) ) {
   selected_physical_device = physical_devices[i];
 }
```

```
}
```

12.*Tutorial01.cpp, function CreateDevice()*

设备属性

我创建了 CheckPhysicalDeviceProperties() 函数。 它提取物理设备的句柄，并检查特定设备是否具备足够的功能供应用正常运行。 如果是，返回真值，并将队列家族索引保存在第二个参数中提供在变量中。 队列和队列家族将在后续章节中介绍。

以下是 CheckPhysicalDeviceProperties() 函数的前半部分：

```
VkPhysicalDeviceProperties device_properties;
VkPhysicalDeviceFeatures    device_features;

vkGetPhysicalDeviceProperties( physical_device, &device_properties );
vkGetPhysicalDeviceFeatures( physical_device, &device_features );

uint32_t major_version = VK_VERSION_MAJOR( device_properties.apiVersion );
uint32_t minor_version = VK_VERSION_MINOR( device_properties.apiVersion );
uint32_t patch_version = VK_VERSION_PATCH( device_properties.apiVersion );
```

```
if( (major_version < 1) &&
   (device_properties.limits.maxImageDimension2D < 4096) ) {
  std::cout << "Physical device " << physical_device << " doesn't support required parameters!" << std::endl;
  return false;
}
```

13.*Tutorial01.cpp, function CheckPhysicalDeviceProperties()*

在函数的开头，查询物理设备的属性和特性。 属性包含的字段有：支持的 Vulkan API 版本、设备名称和类型（集成或专用/独立 GPU）、厂商 ID 和限制。 限制描述如何创建大纹理、anti-aliasing 中支持多少实例、或者特定着色器阶段可以使用多少缓冲区。

设备特性

特性是与扩展功能类似的附加硬件功能。 驱动程序也许没有必要支持这些，而且默认情况下不启用。 特性包含多个项目，比如几何体和镶嵌着色器多个视口、逻辑运算，或其他纹理压缩格式。 如果特定物理设备支持任意特性，那么我们将能够在逻辑设备创建期间启用该特性。 在 Vulkan 中默认不启用特性。 但 Vulkan 规范指出，部分特性可能会对性能（比如稳定性）造成影响。

查询硬件信息和功能后，我提供了一个有关如何使用这些查询的小示例。 我“保留” VK_MAKE_VERSION 宏并检索主要版本和次要版本，并修改了设备属性 apiVersion 字段的版本。 检查它是否高于我希望使用的版本，还检查我能否创建特定大小的 2D 纹理。 在本示例中，我没有使用任何特性，但如果希望使用特性（比如几何体着色器），必须检查它是否支持，并且在逻辑设备创建过程中必须（明确）启用它。 这就是我们为什么需要创建逻辑设备，不直接使用物理设备。 逻辑设备代表物理设备以及我们为其启用的所有特性和扩展功能。

检查物理设备的功能的下一部分 - 队列 - 需要另作解释。

队列、队列家族和命令缓冲区

如果我们希望处理数据（比如通过顶点数据和顶点属性绘制 3D 场景），要调用传递至驱动程序的 Vulkan 函数。 这些函数不直接传递，因为将每个请求

单独向下发送至通信总线的效率非常低。 最好是将它们集中起来，分组传递。 在 OpenGL 中，驱动程序自动完成该过程，用户是看不见的。 OpenGL API 调用在缓冲区中排队，如果该缓冲区已满（或我们请求刷新），整个缓冲区会传递至硬件以作处理。 在 Vulkan 中，该机制对用户是直接可见的，更重要的是，用户必须为命令专门创建并管理缓冲区。 这些是（方便）调用的命令缓冲区。

指令缓冲区（作为整个对象）被传递至硬件，以通过队列来执行。 然而，这些缓冲区包含不同的操作类型，比如图形命令（用于譬如在典型 3D 游戏中生成和显示图像）或计算命令（用于处理数据）。 特定命令类型可能由专用硬件处理，因此队列也可分成不同类型。 在 Vulkan 中，这些队列类型是调用的家族。 每个队列家族都可支持不同的操作类型。 因此我们还必须检查特定物理设备是否支持我们希望执行的操作类型。 另外，我们还可以在一台设备上执行一类操作，在另一台设备上执行另一类操作，但需要检查它的可行性。 这类检查由 CheckPhysicalDeviceProperties() 函数的后半部分完成：

```
uint32_t queue_families_count = 0;
vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count, nullptr );
if( queue_families_count == 0 ) {
 std::cout << "Physical device " << physical_device << " doesn't have any queue families!" << std::endl;
 return false;
}

std::vector<VkQueueFamilyProperties> queue_family_properties( queue_families_count );
vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count, &queue_family_properties[0] );
```

```
for( uint32_t i = 0; i < queue_families_count; ++i ) {
  if( (queue_family_properties[i].queueCount > 0) &&
      (queue_family_properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) ) {
    queue_family_index = i;
    return true;
  }
}

std::cout << "Could not find queue family with required properties on physical device " << physical_device << "!" << std::endl;
return false;
```

14.*Tutorial01.cpp, function CheckPhysicalDeviceProperties()*

我们必须首先检查特定物理设备中有多少可用的队列家族。其检查方式与枚举物理设备类似。首先我们调用 **vkGetPhysicalDeviceQueueFamilyProperties()**，其最后一个参数设为 null。 这样在“queue_count”中，将保存不同列队家族的可变数量。 接下来为该数量的队列家族的属性准备一个位置（如果想这样做 - 其机制与枚举物理设备类似）。 然后再次调用函数，各队列家族的属性将保存在提供的阵列中。

各队列家族的属性包含队列标记、家族中可用队列的数量、时间邮戳支持和图像传输粒度。 现在，最重要的部分是家族中的队列数量和标记。 （位字段）标记定义特定队列家族支持的操作类型（可能支持多种）。 它可以是图形、计算、传输（复制等内存操作），或（针对百万纹理等稀疏资源的）稀疏

绑定操作。 未来可能出现其他类型的操作。
在本示例中，我们检查图形操作支持，如果找到该支持，那么就可以使用特定物理设备。 请记住，我们还需牢记指定的家族索引。 选择物理设备后，我们可以创建将在应用其他部分代表该设备的逻辑设备，如下例所示：

```
if( selected_physical_device == VK_NULL_HANDLE ) {
  std::cout << "Could not select physical device based on the chosen properties!"<< std::endl;
  return false;
}

std::vector<float> queue_priorities = { 1.0f };

VkDeviceQueueCreateInfo queue_create_info = {
  VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,    // VkStructureType              sType
  nullptr,                                       // const void                   *pNext
  0,                                             // VkDeviceQueueCreateFlags     flags
```

```
    selected_queue_family_index,                // uint32_t                 queueFamilyIndex
    static_cast<uint32_t>(queue_priorities.size()), // uint32_t                 queueCount
    &queue_priorities[0]                        // const float              *pQueuePriorities
};

VkDeviceCreateInfo device_create_info = {
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,       // VkStructureType          sType
    nullptr,                                    // const void               *pNext
    0,                                          // VkDeviceCreateFlags      flags
    1,                                          // uint32_t                 queueCreateInfoCount
    &queue_create_info,                         // const VkDeviceQueueCreateInfo     *pQueueCreateInfos
    0,                                          // uint32_t                 enabledLayerCount
    nullptr,                                    // const char * const       *ppEnabledLayerNames
```

```
  0,                                // uint32_t                    enabledExtensionCount
  nullptr,                          // const char * const          *ppEnabledExtensionNames
  nullptr                           // const VkPhysicalDeviceFeatures     *pEnabledFeatures
};

if( vkCreateDevice( selected_physical_device, &device_create_info, nullptr, &Vulkan.Device ) != VK_SUCCESS ) {
  std::cout << "Could not create Vulkan device!" << std::endl;
  return false;
}

Vulkan.QueueFamilyIndex = selected_queue_family_index;
return true;
```

15.*Tutorial01.cpp, function CreateDevice()*

首先确保退出设备特性循环后，我们找到了可满足需求的设备。 然后通过调用 **vkCreateDevice()** 创建逻辑设备。 它提取物理设备的句柄和包含创建设备

所需的信息的结构地址。 该结构的类型为 VkDeviceCreateInfo 并包含以下字段：

- sType – 所提供结构的标准类型，此处的 VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO 表示我们为设备创建提供参数。
- pNext – 指向扩展特定结构的参数；此外我们设为 null。
- flags – 另一留作将来使用的参数，必须是 0。
- queueCreateInfoCount – 不同队列家族的数量，我们通过它创建队列和设备。
- pQueueCreateInfos – queueCreateInfoCount 要素（指定我们希望创建的队列）阵列的指示器。
- enabledLayerCount – 待启用的设备级验证层数量。
- ppEnabledLayerNames – 包含待启用设备级的 enabledLayerCount 名称的阵列的指示器。
- enabledExtensionCount – 为设备启用的扩展功能数量。
- ppEnabledExtensionNames – 包含 enabledExtensionCount 要素的指示器；各要素必须包含应该启用的扩展功能的名称。
- pEnabledFeatures – 结构指示器（表示为该设备启用的其他特性）（请参阅“设备”部分）。

特性（如前所述）是默认禁用的附加硬件功能。 如果希望启用所有可用特性，不能简单用 ones 填充该结构。 如果部分特性不支持，设备创建将失败。相反，我们应传递调用 **vkGetPhysicalDeviceFeatures()**时填充的结构。 这是启用所有支持特性的最简单的方法。 如果我们仅对部分特定特性感兴趣，那么查询面向可用特性的驱动程序，并清空所有不需要的字段。 如果不希望使用任何附加特性，可以清空该结构（用 0 填充），或为该参数传递一个 null 指示器（如本例所示）。

队列与设备一同自动创建。如要指定希望启用的队列类型，需要提供其他 VkDeviceQueueCreateInfo 结构阵列。该阵列必须包含 queueCreateInfoCount 要素。 该阵列中的每个要素都必须引用不同的队列家族；我们仅引用一次特定队列家族。

VkDeviceQueueCreateInfo 结构包含以下字段：

- sType – 结构类型，此处 VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO 表示它为队列创建信息。
- pNext – 为扩展功能预留的指示器。
- flags – 留作将来使用的值。
- queueFamilyIndex – 队列家族（通过它创建队列）的索引。
- queueCount – 我们希望在特定队列家族中启用的队列数量（希望通过该家族使用的队列数量）以及 pQueuePriorities 阵列中的要素数量。
- pQueuePriorities – 包含浮点值（描述通过该家族在各队列中执行的操作的优先级）的阵列。

如前所述，包含 VkDeviceQueueCreateInfo 要素的阵列中的各要素必须描述一个不同的队列家族。 索引是一个数，必须小于 **vkGetPhysicalDeviceQueueFamilyProperties()** 函数提供的值（必须小于可用队列家族的数量）。 在本示例中，我们仅对一个队列家族中的一个队列感兴

趣。 因此我们必须记住该队列家族索引。 它用于此处。 如果想准备一个更为复杂的场景，还应记住各家族的列队数量，因为各家族可能支持不同数量的队列。 而且创建的队列不能超过特定家族中的可用队列！

值得注意的一点是，不同队列家族可能有类似（甚至相同）的属性），这意味着它们可能支持类似的操作类型，即支持图形操作的队列家族不止一个。 此外，各家族可能包含不同数量的队列。

我们还必须向各队列分配一个浮点值（从 0.0 到 1.0，包括这两个值）。 我们为特定队列提供的值越大（相对于分配至其他队列的值），特定队列处理命令的时间越长（相对于其他队列）。 但这种关系并不绝对。 优先级也不会影响执行顺序。 它只是一个提示。

优先级仅在单台设备上有关系。 如果在多台设备上执行操作，优先级会影响各台设备（而非两台设备之间）的处理时间。 带有特定值的队列的重要性可能仅高于相同设备上优先级较低的队列。 不同设备的队列独立对待。 一旦我们填充了这些结构并调用 **vkCreateDevice()**，如果成功，创建的逻辑设备将保存在我们提供地址的变量中（在本示例中称为 VulkanDevice）。 如果该函数失败，将返回与 VK_SUCCESS 相反的值。

获取设备级函数指示器

我们创建了一台逻辑设备。 现在可以用它加载设备级函数。 正如我之前提到的在真实场景中，将有多家硬件厂商在单台计算机上为我们提供 Vulkan 实施的情况。 OpenGL 中现在就出现了这种情况。 许多计算机都有主要用于游戏的专用/独立显卡，但也有内置于处理器的英特尔显卡（当然也能用于游戏）。 因此未来将有更多设备支持 Vulkan。 而且借助 Vulkan。我们可以将处理分成任意硬件。 是否还记得什么时候出现了专门用于物理处理的扩展卡？ 或者往前回顾，带有附加显卡“加速器”的正常“2D”卡（是否还记得 Voodoo 卡）？ Vulkan 已准备好应对这种场景。

那么，如果有多台设备，我们该怎么处理设备级函数？ 我们可以加载通用程序。 这可通过 **vkGetInstanceProcAddr()** 函数来完成。 它返回派遣函数的地址，执行根据提供的逻辑设备句柄跳至相应实施的行为。 但我们可以通过分别加载各逻辑设备的函数，避免这种开销。 使用这种方法时，必须记住，只能基于供加载该函数的设备来调用特定函数。 因此如果在应用中使用较多设备，必须从各设备加载函数。 这并不是特别困难。 而且，尽管这样会导致保存较多函数（并根据供加载的设备对它们进行分组），但我们可以避免抽象层，并节约一部分处理器时间。 加载函数的方式与加载导出函数、全局级函数和实例级函数的方式类似：

```
#define VK_DEVICE_LEVEL_FUNCTION( fun )                                          \
if( !(fun = (PFN_##fun)vkGetDeviceProcAddr( Vulkan.Device, #fun )) ) {           \
  std::cout << "Could not load device level function: " << #fun << "!" << std::endl;  \
```

```
  returnfalse;                                                \
}

#include "ListOfFunctions.inl"

return true;
```

16.*Tutorial01.cpp, function LoadDeviceLevelEntryPoints()*

这次我们使用 **vkGetDeviceProcAddr()** 函数和逻辑设备句柄。 设备级函数放在共享标头中。 这次它们包含在 VK_DEVICE_LEVEL_FUNCTION() 宏中，如下所示：

```
#if !defined(VK_DEVICE_LEVEL_FUNCTION)
#define VK_DEVICE_LEVEL_FUNCTION( fun )
#endif

VK_DEVICE_LEVEL_FUNCTION( vkGetDeviceQueue )
```

```
VK_DEVICE_LEVEL_FUNCTION( vkDestroyDevice )
VK_DEVICE_LEVEL_FUNCTION( vkDeviceWaitIdle )

#undef VK_DEVICE_LEVEL_FUNCTION
```

17.*ListOfFunctions.inl*

所有函数都不是来自于导出、全局或实例级，而是来自设备级。第一个参数会造成另一个区别：对于设备级函数，列表中的第一个参数只能是类型 VkDevice、VkQueue 或 VkCommandBuffer。在接下来的教程中，如果出现新的函数，必须添加至 ListOfFunctions.inl 并进一步添加至 VK_DEVICE_LEVEL_FUNCTION 部分（有一些明显的例外情况，比如扩展功能）。

检索队列

创建设备后，我们需要能够为数据处理提交部分命令的队列。 队列通过逻辑设备自动创建，但为了使用这些队列，我们必须特别要求队列句柄。 这可通过 **vkGetDeviceQueue()** 完成，如下所示：

```
vkGetDeviceQueue( Vulkan.Device, Vulkan.QueueFamilyIndex, 0, &Vulkan.Queue );
```

18.*Tutorial01.cpp, function GetDeviceQueue()*

如要检索队列句柄，必须提供用于获取队列的逻辑设备。 还需要队列家族索引，该索引必须是我们在逻辑设备创建期间提供的索引之一（不能创建其他队列或使用我们没有请求的家族的队列）。 最后一个参数是来自特定家族的队列索引；它必须小于从特定家族请求的队列总数。 例如，如果设备支持 3 号家族的 5 个队列，而我们希望该家族提供 2 个队列，那么队列索引必须小于 2。 对于我们希望检索的各个队列来说，必须调用该函数并进行单独查询。 如果函数调用成功，请求队列的句柄会保存在我们在最后一个参数中提供地址的变量中。 从这时起，希望（使用命令缓冲区）执行的所有工作都可提交至获取的队列中以供处理。

Tutorial01 执行

如前所述，本教程提供的示例无法演示所有内容。 不过我们了解了足够多的信息。 那么，我们如何知道一切是否进展顺利？ 如果出现正常的应用窗口，控制台/终端没有打印任何内容，表示 Vulkan 设置成功。 从下一教程开始，操作结果将显示在屏幕上。

清空

我们还需牢记的一点是：清空和释放资源。 必须以特定的顺序（通常与创建顺序相反）进行清空。

应用关闭后，操作系统应释放内存及其他所有相关资源。 这应包含 Vulkan；驱动程序通常清空没有引用的资源。 遗憾的是，这种清空没有以相应的顺序执行，因此可能会导致应用在关闭过程中崩溃。 最佳实践是自己执行清理。 以下是释放在第一个教程中创建的资源所需的示例代码：

```
if( Vulkan.Device != VK_NULL_HANDLE ) {
 vkDeviceWaitIdle( Vulkan.Device );
 vkDestroyDevice( Vulkan.Device, nullptr );
}

if( Vulkan.Instance != VK_NULL_HANDLE ) {
 vkDestroyInstance( Vulkan.Instance, nullptr );
}

if( VulkanLibrary ) {
#if defined(VK_USE_PLATFORM_WIN32_KHR)
```

```
  FreeLibrary( VulkanLibrary );
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
  dlclose( VulkanLibrary );
#endif
}
```

19.*Tutorial01.cpp, destructor*

我们应该经常检查，看是否已创建任何特定资源。 没有逻辑设备就没有设备级函数指示器，也就无法调用相应的资源清理函数。 同样，如果没有实例，就无法获取 **vkDestroyInstance()** 函数的指示器。 一般来说，我们不能释放没有创建的资源。

必须确保对象在删除之前，没有经过设备的使用。 因此有一个等候函数，等特定设备的所有队列上的处理过程完成之后才进行拦截。 接下来，我们使用 **vkDestroyDevice()** 函数破坏逻辑设备。 与此相关的所有队列都会自动破坏掉，然后破坏实例。 在这之后我们就能够释放（或卸载）供获取所有函数的 Vulkan 库。

结论

本教程介绍了如何在应用中为使用 Vulkan 做准备。 首先“连接”Vulkan Runtime 库，并从中加载全局级函数。 然后创建 Vulkan 实例并加载实例级函数。 之后检查哪些物理设备可用，以及它们具备的特性、属性和功能。 接下来创建逻辑设备，并描述必须与设备一同创建的队列及其数量。 然后使用新创建的逻辑设备句柄检索设备级函数。 另外需要做的一件事是检索供我们提交待执行工作的队列。

目录

        - 检查看是否支持实例扩展
        - 启用实例层扩展
        - 创建演示平面
        - 检查是否支持设备扩展
        - 检查是否支持演示至特定平面
        - 借助启用的交换链扩展创建设备
    - 创建旗语（semaphore)
    - 创建交换链
        - 获取平面功能
        - 获取支持的平面格式
        - 获取支持的演示模式
        - 选择交换链图像数量
        - 选择交换链图像格式
        - 选择交换链图像大小
        - 选择交换链用法标记
        - 选择预转换
        - 选择演示模式
        - 创建交换链
    - 图像演示
    - 检查在交换链中创建的图像
    - 重新创建交换链
    - 快速了解命令缓冲区
        - 创建命令缓冲区内存池
        - 分配命令缓冲区
        - 记录命令缓冲区
        - 图像布局和布局过渡

教程 2： 交换链 — 集成 Vulkan 和操作系统

欢迎观看第 2 节 Vulkan 教程。 在第 1 节教程中，我们介绍了 Vulkan 基本设置：功能加载、实例创建、选择物理设备和队列，以及逻辑设备创建。 现在您一定希望绘制一些图像！ 很遗憾，我们需要等到下一节。 为什么？ 因为如果我们绘图，肯定希望能够看见它。 与 OpenGL* 类似，我们必须将 Vulkan 管道与操作系统提供的应用和 API 相集成。 不过很遗憾，使用 Vulkan 时，这项任务并不简单。 正如其他精简 API 一样，这样做是有目的的 — 实现高性能和灵活性。

那么如何集成 Vulkan 和应用窗口？ 它与 OpenGL 之间有何不同？ 在（Microsoft Windows* 上的）OpenGL 中，我们获取与应用窗口相关的设备环境 (Device Context)。 使用时，我们需要定义“如何”在屏幕上演示图像，用“哪种”格式在应用窗口上绘制，以及应支持哪些功能。 这可通过像素格式完成。 大多数时候，我们可通过 24 位深度缓冲区和双缓冲支持（这样我们可以在“隐藏的”后台缓冲区绘图，并图像完成后在屏幕上演示图像 — 交换前后缓冲区），创建 32 位色彩平面。 只有完成这些准备工作后，才能创建并激活渲染环境 (Rendering Context)。 在 OpenGL 中，所有渲染直接进入默认的后台缓冲区。 在 Vulkan 中没有默认帧缓冲区。 我们可以创建一个不显示任何内容的应用。 这种方法非常有效。 但是如果希望显示内容，我们可以创建支持渲染的缓冲区集。 这些缓冲区及其属性（与 Direct3D* 类似）称为交换链。 交换链可包含许多图像。 如果要显示其中一些图像，我们不必“交换”（顾名思义）图像，而是演示这些图像，这意味着我们要将它们返回至演示引擎。 因此在 OpenGL 中，首先需要定义平面格式并建立它与窗口（至少在 Windows 上）的相关性，然后创建渲染环境。 在 Vulkan 中，我们首先创建实例、设备，然后创建交换链。 但有意思的是，在一些情况下，需要破坏该交换链并重新创建它。 在工作流程中， 从头开始！

请求交换链扩展

在 Vulkan 中，交换链就是一种扩展。 为什么？ 我们希望在应用窗口中的屏幕中显示图像，这不是很明显吗？

并不明显。 Vulkan 可用于多种用途，包括执行数学运算、提升物理计算速度，以及处理视频流。 这些行为结果无需显示在常用显示器上，这就是核心 API 适用于操作系统的原因（与 OpenGL 类似）。

如果想创建游戏，并将渲染后的图像显示在显示器上，您可以（并应该）使用交换链。 但我们说交换链是扩展的原因还有第二个。 每种操作系统显示图像的方式都各不相同。 供您渲染图像的平面可能以不同的方式实施，格式不同，在操作系统的表现方式也不相同 — 没有一种通用的方法。 因此在 Vulkan 中，交换链还必须依赖编写应用所针对的操作系统。

在 Vulkan 中，交换链被视作一种扩展，原因如下：它可提供与操作系统特定代码相集成的渲染对象（在 OpenGL 中为 FBO 等缓冲区或图像）。 这是核心 Vulkan（独立于平台）所无法完成的。因此如果交换链创建和用法是一种扩展，那么我们需要在实例和设备创建过程中请求扩展。 我们必须在两个层面（至少在大多数操作系统（包括 Windows 和 Linux*）上）启用扩展，才能创建和使用交换链。 这意味着我们必须回到第 1 节教程并进行更改，以请求与交换链有关的相应扩展。 如果特定实例和设备不支持这些扩展，将无法创建该实例和/或设备。 当然我们还能采用其他方法显示图像，比如获取针对缓冲区（纹理）的内存指示器（映射它），或将数据拷贝至操作系统获取的窗口的平面指示器。 该流程（尽管并不困难）不在本教程讨论范围内。 但幸运的是，交换链似乎与 OpenGL 的核心扩展类似：并不在核心规范内，也不要求实施，但所有硬件厂商都会实施。 我认为所有硬件厂商都希望证明，他们支持 Vulkan，而且这样能够显著提升屏幕上显示的游戏的性能。而且还有一点能够支持该理论，即交换链扩展可集成至主要的“核心”vulkan.h 标头。
如果支持交换链，实际上会涉及到三种扩展：两种来源于实例层，一种来源于设备层。 从逻辑上来说，这些扩展将不同的功能分开来。 第一种是在实例层定义的 ***VK_KHR_surface*** 扩展。 它可描述“平面”对象，即应用窗口的逻辑表现形式。 该扩展支持我们查看平面的不同参数（功能、支持的格式、大小），并查询特定物理设备是否支持交换链（更确切的说，特定队列家族是否支持在特定平面上演示图像）。 这些信息非常实用，因为我们不想选择物理设备并尝试通过它创建逻辑设备，来了解它是否支持交换链。 该扩展还可定义破坏此类平面的方法。
第二种实例层扩展依赖于操作系统：在 Windows 操作系统家族中称为 ***VK_KHR_win32_surface***，在 Linux 中称为 ***VK_KHR_xlib_surface*** 或 ***VK_KHR_xcb_surface***。 该扩展支持我们创建在特定操作系统中展现应用窗口（并使用特定于操作系统的参数）的平面。

检查是否支持实例扩展

启用这两种实例层扩展之前，需要查看它们是否可用或受到支持。 我们一直在讨论实例扩展，还未创建过任何实例。 为确定 Vulkan 实例是否支持这些扩展，我们使用全局级函数 **vkEnumerateInstanceExtensionProperties()**。 它列举所有可用实例通用扩展，第一个参数是否为 null，或实例层扩展（似乎层级也可以有扩展），是否将第一个参数设置为任意特定层级的名称。 我们对层级不感兴趣，因此将第一个参数设为 null。 我们重新调用该函数两次。 在第一次调用中，我们希望获取支持的扩展总数，因此将第三个参数保留为 null。 接下来我们为所有扩展准备存储，并用第三个指向已分配存储再次调用该函数。

```
uint32_t extensions_count = 0;
```

```
if( (vkEnumerateInstanceExtensionProperties( nullptr, &extensions_count, nullptr ) != VK_SUCCESS) ||
```

```
  (extensions_count == 0) ) {
  std::cout << "Error occurred during instance extensions enumeration!" << std::endl;
  return false;
}

std::vector<VkExtensionProperties> available_extensions( extensions_count );
if( vkEnumerateInstanceExtensionProperties( nullptr, &extensions_count, &available_extensions[0] ) != VK_SUCCESS ) {
  std::cout << "Error occurred during instance extensions enumeration!" << std::endl;
  return false;
}

std::vector<const char*> extensions = {
  VK_KHR_SURFACE_EXTENSION_NAME,
```

```
#if defined(VK_USE_PLATFORM_WIN32_KHR)
  VK_KHR_WIN32_SURFACE_EXTENSION_NAME
#elif defined(VK_USE_PLATFORM_XCB_KHR)
  VK_KHR_XCB_SURFACE_EXTENSION_NAM
E
#elif defined(VK_USE_PLATFORM_XLIB_KHR)
  VK_KHR_XLIB_SURFACE_EXTENSION_NAME
#endif
};
for( size_t i = 0; i < extensions.size(); ++i ) {
  if( !CheckExtensionAvailability( extensions[i], available_extensions ) ) {
    std::cout << "Could not find instance extension named \"" << extensions[i] << "\"!" << std::endl;
    return false;
```

```
  }
}
```

1. *Tutorial02.cpp，函数 CreateInstance()*

我们可以为较少数量的扩展准备一个位置，然后 **vkEnumerateInstanceExtensionProperties()** 会返回 VK_INCOMPLETE，以让我们知道我们没有获取所有扩展。我们的阵列现在布满了所有可用（支持的）实例层扩展。 已分配空间的各要素均包含扩展名称及其版本。 第二个参数可能不常使用，但可帮助我们检查硬件是否支持特定版本的扩展。 例如，我们可能对部分特定扩展感兴趣，并为此下载了包含许多标头文件的 SDK。 每个标头文件都有自己的版本，与该查询返回的值相对应。 如果供应用执行的硬件支持旧版本扩展（不是我们下载了 SDK 的扩展），它可能不支持我们通过该特定扩展所使用的所有功能。 因此，有时验证版本非常实用，但对交换链来说无所谓 — 至少现在是这样。

现在我们可以搜索所有返回的扩展，检查该列表是否包含我们要寻找的扩展。 这里我使用两个方便的定义，分别为 VK_KHR_SURFACE_EXTENSION_NAME 和 VK_KHR_????_SURFACE_EXTENSION_NAME。 它们在 Vulkan 标头文件中定义，并包含扩展名称，因此我们无需拷贝或记住它们。 我们只需使用代码中的定义，如果出现错误，编译器会告诉我们。 我希望所有扩展都带有类似的定义。

第二个定义带有一个。 这两个提到的定义都位于 vulkan.h 标头文件中。 但第二个不是特定于操作系统定义，且 vulkan.h 标头独立于操作系统吗？ 两个问题都是对的，而且非常有效。 vulkan.h 文件独立于操作系统，且包含特定于操作系统的扩展的定义。 但这些均位于 #ifdef … #endif 预处理器指令之中。 如果想“启用”它们，需要在项目的某个地方添加相应的预处理器指令。 对 Windows 系统来说，需要添加 VK_USE_PLATFORM_WIN32_KHR 字符串。 在 Linux 上，根据我们是否希望使用 X11 或 XCB 库，需要添加 VK_USE_PLATFORM_XCB_KHR 或 VK_USE_PLATFORM_XLIB_KHR。 在提供的示例项目中，这些定义通过 CMakeLists.txt 文件默认添加。

但回到源代码。 CheckExtensionAvailability() 函数具有哪些功能？ 它循环所有可用扩展，并将它们的名称与所提供扩展的名称进行对比。 如果发现匹配，将返回“真”值。

```
for( size_t i = 0; i < available_extensions.size(); ++i ) {
  if( strcmp( available_extensions[i].extensionName, extension_name ) == 0 ) {
```

```
    return true;
  }
}
return false;
```

2.*Tutorial02.cpp，函数 CheckExtensionAvailability()*

启用实例层扩展

我们已经验证了这两种扩展均受支持。实例层扩展在实例创建中请求（启用）— 我们创建包含应启用的扩展列表的实例。以下代码负责完成这一步骤：

```
VkApplicationInfo application_info = {
  VK_STRUCTURE_TYPE_APPLICATION_INFO,          // VkStructureType          sType
  nullptr,                                     // const void               *pNext
  "API without Secrets: Introduction to Vulkan",  // const char            *pApplicationName
  VK_MAKE_VERSION( 1, 0, 0 ),                  // uint32_t                 applicationVersion
  "Vulkan Tutorial by Intel",                  // const char               *pEngineName
  VK_MAKE_VERSION( 1, 0, 0 ),                  // uint32_t                 engineVersion
```

```
    VK_API_VERSION                          // uint32_t                apiVersion
};

VkInstanceCreateInfo instance_create_info = {
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,     // VkStructureType          sType
    nullptr,                                    // const void               *pNext
    0,                                          // VkInstanceCreateFlags    flags
    &application_info,                          // const VkApplicationInfo  *pApplicationInfo
    0,                                          // uint32_t                 enabledLayerCount
    nullptr,                                    // const char * const       *ppEnabledLayerNames
    static_cast<uint32_t>(extensions.size()),   // uint32_t                 enabledExtensionCount
    &extensions[0]                              // const char * const       *ppEnabledExtensionNames
};
```

```
if( vkCreateInstance( &instance_create_info, nullptr, &Vulkan.Instance ) != VK_SUCCESS ) {
  std::cout << "Could not create Vulkan instance!" << std::endl;
  return false;
}
return true;
```

3.*Tutorial02.cpp，函数 CreateInstance()*

该代码与 Tutorial01.cpp 文件中的 **CreateInstance()** 函数类似。 要请求实例层扩展，必须为阵列准备我们希望启用的所有扩展的名称。 此处我们使用包含 “const char*” 要素，以及以定义形式提及的扩展名称的标准矢量。

在教程 1 中，我们声明零扩展，并将阵列地址的 nullptr 放在 VkInstanceCreateInfo 结构中。 这次我们必须提供阵列（包含请求扩展的名称）的第一个要素的地址。 而且还必须指定阵列所包含的要素数量（因此我们选择使用矢量：如果在未来版本中需要添加或删除扩展，该矢量的大小也会相应更改）。

接下来我们调用 **vkCreateInstance()** 函数。 如果不返回 VK_SUCCESS，表示（在本教程中）不支持这些扩展。 如果成功返回，我们可像之前一样加载实例层函数，但这次还需加载一些其他特定于扩展的函数。

这些函数附带了部分其他的函数。 而且因为它是实例层函数，因此必须将它们添加至实例层函数集（以便在有相应函数时能够适时加载）。 在本示例中，必须将以下函数添加至打包在 VK_INSTANCE_LEVEL_FUNCTION() 宏中的 ListOfFunctions.inl，如下所示：

```
// From extensions
#if defined(USE_SWAPCHAIN_EXTENSIONS)
```

```
VK_INSTANCE_LEVEL_FUNCTION( vkDestroySurfaceKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceSupportKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceCapabilitiesKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceFormatsKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfacePresentModesKHR )
#if defined(VK_USE_PLATFORM_WIN32_KHR)
VK_INSTANCE_LEVEL_FUNCTION( vkCreateWin32SurfaceKHR )
#elif defined(VK_USE_PLATFORM_XCB_KHR)
VK_INSTANCE_LEVEL_FUNCTION( vkCreateXcbSurfaceKHR )
#elif defined(VK_USE_PLATFORM_XLIB_KHR)
VK_INSTANCE_LEVEL_FUNCTION( vkCreateXlibSurfaceKHR )
#endif
#endif
```

4.*ListOfFunctions.inl*

还有一件事情： 我已将所有与交换链相关的函数打包在其他 #ifdef … #endif 配对中，这要求定义 USE_SWAPCHAIN_EXTENSIONS 预处理器指令。 我已完成这步，这样便能够按照教程 1 中的说明操作。 没有它，第一个应用（因为它使用相同的标头文件）将尝试加载所有函数。 但我们没有启用教程 1 中的交换链扩展，因此此操作会失败，应用也将在没有完全初始化 Vulkan 的情况下关闭。 如果没有启用特定扩展，它的函数将不可用。

创建演示平面

我们借助两个启用的扩展创建了 Vulkan 实例。 我们通过核心 Vulkan 规范和启用的扩展加载了实例层函数（得益于宏，这一过程自动完成）。 为创建平面，我们编写了如下代码：

```
#if defined(VK_USE_PLATFORM_WIN32_KHR)
VkWin32SurfaceCreateInfoKHR surface_create_info = {
  VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR,  // VkStructureType                sType
  nullptr,                                          // const void                    *pNext
  0,                                                // VkWin32SurfaceCreateFlagsKHR  flags
  Window.Instance,                                  // HINSTANCE                     hinstance
  Window.Handle                                     // HWND                          hwnd
};
```

```
if( vkCreateWin32SurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr, &Vulkan.PresentationSurface ) == VK_SUCCESS ) {
  return true;
}

#elif defined(VK_USE_PLATFORM_XCB_KHR)
VkXcbSurfaceCreateInfoKHR surface_create_info = {
  VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR,    // VkStructureType                sType
  nullptr,                                          // const void                    *pNext
  0,                                                // VkXcbSurfaceCreateFlagsKHR     flags
  Window.Connection,                                // xcb_connection_t*              connection
  Window.Handle                                     // xcb_window_t                   window
};

```

```
if( vkCreateXcbSurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr, &Vulkan.PresentationSurface ) == VK_SUCCESS ) {
  return true;
}

#elif defined(VK_USE_PLATFORM_XLIB_KHR)
VkXlibSurfaceCreateInfoKHR surface_create_info = {
  VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR,   // VkStructureType              sType
  nullptr,                                  // const void                  *pNext
  0,                                        // VkXlibSurfaceCreateFlagsKHR     flags
  Window.DisplayPtr,                        // Display                     *dpy
  Window.Handle                             // Window                       window
};
if( vkCreateXlibSurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr, &Vulkan.PresentationSurface ) == VK_SUCCESS ) {
```

```
  return true;
}

#endif

std::cout << "Could not create presentation surface!" << std::endl;
return false;
```

5.*Tutorial02.cpp，函数 CreatePresentationSurface()*

为创建演示平面，我们调用 **vkCreate????SurfaceKHR()** 函数，以接受 Vulkan 实例（借助启用的平面扩展）、特定于操作系统的结构指示器、可选内存分配处理函数的指示器，以及供保存已创建平面的句柄的变量指示器。

特定于操作系统的结构称为 **Vk????SurfaceCreateInfoKHR** 并包含以下字段：

- sType – 标准结构类型，应相当于 **VK_STRUCTURE_TYPE_????_SURFACE_CREATE_INFO_KHR**（其中 ???? 可以是 WIN32、XCB、XLIB 或其他）
- pNext – 其他部分结构的标准指示器
- flags – 留作将来使用的参数。
- hinstance/connection/dpy – 第一个特定于操作系统的参数
- hwnd/window – 应用窗口的句柄（同样特定于操作系统）

检查是否支持设备扩展

我们已经创建了实例和平面。 下一步是创建逻辑设备。 但我们希望创建支持交换链的设备。 因此我们还需检查特定物理设备是否支持交换链扩展 — 设备层扩展。 该扩展称为 ***VK_KHR_swapchain***，可定义交换链的实际支持、实施和用法。

要检查特定物理设备支持哪些扩展，我们必须创建与为实例层扩展准备的代码类似的代码。 这次我们只使用 **vkEnumerateDeviceExtensionProperties()** 函数。 它的运行模式与查询实例扩展的函数大体相同。 唯一的区别是它提取第一个参数中的附加物理设备句柄。 该函数的代码类似于以下示例。 在我们的示例源代码中，它是 CheckPhysicalDeviceProperties() 函数的一部分。

```
uint32_t extensions_count = 0;
if( (vkEnumerateDeviceExtensionProperties( physical_device, nullptr, &extensions_count, nullptr ) != VK_SUCCESS) ||
   (extensions_count == 0) ) {
 std::cout << "Error occurred during physical device " << physical_device << " extensions enumeration!" << std::endl;
 return false;
}

std::vector<VkExtensionProperties> available_extensions( extensions_count );
if( vkEnumerateDeviceExtensionProperties( physical_device, nullptr, &extensions_count, &available_extensions[0] ) != VK_SUCCESS ) {
 std::cout << "Error occurred during physical device " << physical_device << " extensions enumeration!" << std::endl;
 return false;
```

```
}

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

for( size_t i = 0; i < device_extensions.size(); ++i ) {
  if( !CheckExtensionAvailability( device_extensions[i], available_extensions ) ) {
    std::cout << "Physical device " << physical_device << " doesn't support extension named \"" << device_extensions[i] << "\"!" << std::endl;
    return false;
  }
}
```

6.*Tutorial02.cpp，函数 CheckPhysicalDeviceProperties()*

我们首先请求特定物理设备上可用的所有扩展的数量。 接下来获取它们的名称并检查设备层交换链扩展。 如果没有，那没有必要进一步查看设备的属

性、特性和队列家族的属性，因为特定设备根本不支持交换链。

检查是否支持演示至特定平面

现在我们回过头来看 CreateDevice() 函数。 创建完实例后，我们在教程 1 中循环了所有可用物理设备并查询了它们的属性。 根据这些属性，我们选择了我们所希望使用的设备和希望请求的队列家族。 该查询在循环所有可用物理设备的过程中完成。 既然我们希望使用交换链，那么必须对 CheckPhysicalDeviceProperties() 函数（通过 CreateDevice() 函数在所提到的循环中调用）进行更改，如下所示：

```
uint32_t selected_graphics_queue_family_index = UINT32_MAX;
uint32_t selected_present_queue_family_index = UINT32_MAX;

for( uint32_t i = 0; i < num_devices; ++i ) {
 if( CheckPhysicalDeviceProperties( physical_devices[i], selected_graphics_queue_family_index, selected_present_queue_family_index ) ) {
  Vulkan.PhysicalDevice = physical_devices[i];
 }
}
```

7.*Tutorial02.cpp，函数 CreateDevice()*

唯一的更改是添加了另外一个变量，它将包含支持交换链的队列家族的索引（更精确的图像演示）。 遗憾的是，仅检查是否支持交换链扩展远远不够，因为演示支持是一种队列家族属性。 物理设备可能支持交换链，但这不表示其所有队列家族都支持交换链。 我们是否针对需要另一队列或队列家族来显示图像？ 我们能不能仅使用我们在教程 1 中选择的图形队列？ 大多数时候，一个队列家族就能满足需求。 这意味着所选择的队列家族将支持图形操作和演示。 但遗憾的是，单个队列家族中还有可能存在不支持图形和演示的设备。 在 Vulkan 中，我们需要灵活应对任何情况。

vkGetPhysicalDeviceSurfaceSupportKHR() 函数用于查看特定物理设备的特定队列家族是否支持交换链，或更准确地说，是否支持在特定平面上演示图像。因此我们才需要提前创建平面。

假设我们已经检查了特定物理设备是否显示交换链扩展，并查询了一些特定物理设备支持的队列家族。 我们还请求了所有队列家族的属性。 现在可以检查特定队列家族是否支持在平面（窗口）上演示图像。

```
uint32_t graphics_queue_family_index = UINT32_MAX;
uint32_t present_queue_family_index = UINT32_MAX;

for( uint32_t i = 0; i < queue_families_count; ++i ) {
 vkGetPhysicalDeviceSurfaceSupportKHR( physical_device, i, Vulkan.PresentationSurface, &queue_present_support[i] );

 if( (queue_family_properties[i].queueCount > 0) &&
    (queue_family_properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) ) {
  // Select first queue that supports graphics
  if( graphics_queue_family_index == UINT32_MAX ) {
    graphics_queue_family_index = i;
```

```
    }

    // If there is queue that supports both graphics and present - prefer it
    if( queue_present_support[i] ) {
      selected_graphics_queue_family_index = i;
      selected_present_queue_family_index = i;
      return true;
    }
  }
}

// We don't have queue that supports both graphics and present so we have to use separate queues
for( uint32_t i = 0; i < queue_families_count; ++i ) {
```

```
  if( queue_present_support[i] ) {
    present_queue_family_index = i;
    break;
  }
}

// If this device doesn't support queues with graphics and present capabilities don't use it
if( (graphics_queue_family_index == UINT32_MAX) ||
    (present_queue_family_index == UINT32_MAX) ) {
  std::cout << "Could not find queue family with required properties on physical device " << physical_device << "!" << std::endl;
  return false;
}
```

```
selected_graphics_queue_family_index = graphics_queue_family_index;

selected_present_queue_family_index = present_queue_family_index;

return true;
```

8.*Tutorial02.cpp，函数 CheckPhysicalDeviceProperties()*

这里我们要迭代所有可用的队列家族。 在每次循环迭代中，我们调用一个负责查看特定队列家族是否支持演示图像的函数。 **vkGetPhysicalDeviceSurfaceSupportKHR()** 要求我们提供物理设备句柄、欲检查的队列家族，以及希望渲染（演示图像）的平面句柄。 如果支持，特定地址中将保存 VK_TRUE；否则将保存 VK_FALSE。

现在我们具有所有可用队列家族的属性。 我们知道哪种队列家族支持图形操作，哪种支持图像演示。 在本教程示例中，我更偏向于支持这两种功能的队列家族。 找到一个后，保存该队列家族的索引，并立即退出 CheckPhysicalDeviceProperties() 函数。 如果没有这种队列家族，我将使用支持图形的第一个队列家族和支持图像演示的第一个队列家族。 只有这样才能使该函数包含“成功”返回代码。

高级场景可能搜索所有可用设备，并尝试寻找包含支持图形操作和演示的队列家族的设备。 不过我还想象了没有设备支持这两种操作的场景。 那么我们必须用一台设备进行图形运算（类似于老式的“图形加速器”），用另一台设备在屏幕（连接“加速器”和显示器）上演示结果。 遗憾的是，在这种情况下，我们必须使用 Vulkan Runtime 的“通用”Vulkan 函数，或者需要保存适用于每台设备（设备实施 Vulkan 函数的方式各不相同 ）的设备层函数。 但我们希望这种场景不要经常出现。

借助启用的交换链扩展创建设备

现在我们回到 CreateDevice() 函数。 我们发现了支持图形和演示操作，但在单个队列家族中不一定支持的物理设备。 现在我们需要创建一台逻辑设备。

```
std::vector<VkDeviceQueueCreateInfo> queue_create_infos;

std::vector<float> queue_priorities = { 1.0f };

```

```
queue_create_infos.push_back( {
  VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,      // VkStructureType              sType
  nullptr,                                         // const void                   *pNext
  0,                                               // VkDeviceQueueCreateFlags     flags
  selected_graphics_queue_family_index,            // uint32_t                     queueFamilyIndex
  static_cast<uint32_t>(queue_priorities.size()),  // uint32_t                     queueCount
  &queue_priorities[0]                             // const float                  *pQueuePriorities
} );

if( selected_graphics_queue_family_index != selected_present_queue_family_index ) {
  queue_create_infos.push_back( {
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,    // VkStructureType              sType
    nullptr,                                       // const void                   *pNext
```

```
    0,                                  // VkDeviceQueueCreateFlags      flags
    selected_present_queue_family_index,          // uint32_t                 queueFamilyIndex
    static_cast<uint32_t>(queue_priorities.size()), // uint32_t                 queueCount
    &queue_priorities[0]                        // const float              *pQueuePriorities
  } );
}

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

VkDeviceCreateInfo device_create_info = {
  VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,         // VkStructureType                sType
```

```
    nullptr,                                // const void                   *pNext
    0,                                      // VkDeviceCreateFlags          flags
    static_cast<uint32_t>(queue_create_infos.size()), // uint32_t                   queueCreateInfoCount
    &queue_create_infos[0],                 // const VkDeviceQueueCreateInfo     *pQueueCreateInfos
    0,                                      // uint32_t                     enabledLayerCount
    nullptr,                                // const char * const           *ppEnabledLayerNames
    static_cast<uint32_t>(extensions.size()),       // uint32_t                     enabledExtensionCount
    &extensions[0],                         // const char * const           *ppEnabledExtensionNames
    nullptr                                 // const VkPhysicalDeviceFeatures     *pEnabledFeatures
};

if( vkCreateDevice( Vulkan.PhysicalDevice, &device_create_info, nullptr, &Vulkan.Device ) != VK_SUCCESS ) {
  std::cout << "Could not create Vulkan device!" << std::endl;
```

```
  return false;
}

Vulkan.GraphicsQueueFamilyIndex = selected_graphics_queue_family_index;
Vulkan.PresentQueueFamilyIndex = selected_present_queue_family_index;
return true;
```

9.*Tutorial02.cpp，函数 CreateDevice()*

像之前一样，我们需要填充 VkDeviceCreateInfo 类型的变量。为此，我们需要声明队列家族和欲启用的队列数量。我们通过包含 VkDeviceQueueCreateInfo 要素的独立阵列的指示器来完成这一步骤。 此处我声明一个矢量，并添加一个要素，定义支持图形操作的队列家族的某个队列。 使用矢量的原因是，如果不支持图形和演示操作，我们将需要定义两个单独的队列家族。 如果单个家族支持这两种操作，我们仅需定义一个成员，并声明仅需一个家族。 如果图形和演示家族的索引不同，我们需要声明面向矢量和 VkDeviceQueueCreateInfo 要素的其他成员。 在这种情况下，VkDeviceCreateInfo 结构必须提供这两个不同家族的信息。 因此矢量将再次派上用场（通过其 size() 成员函数）。

但我们还未完成设备创建。 我们需要请求第三个与交换链相关的扩展 — 设备层“***VK_KHR_swapchain***”扩展。 如前所述，该扩展定义交换链的实际支持、实施和用法。

请求该扩展时，与实例层一样，我们定义一个阵列（或矢量），其中我们欲启用的所有设备层扩展的名称。 我们提供该阵列第一个要素的地址，以及希望使用的扩展数量。 该扩展还以 #define VK_KHR_SWAPCHAIN_EXTENSION_NAME 的形式包含其名称定义。 我们可以在阵列（矢量）中使用该扩展，无需担心任何 typo 问题。

第三个扩展包含用于实际创建、破坏、或管理交换链的其他函数。 使用之前，我们需要将指示器加载至这些函数。 它们来自于设备层，因此我们可以使用 VK_DEVICE_LEVEL_FUNCTION() 宏将它们放在 ListOfFunctions.inl 文件中。

```
// From extensions
#if defined(USE_SWAPCHAIN_EXTENSIONS)
VK_DEVICE_LEVEL_FUNCTION( vkCreateSwapchainKHR )
VK_DEVICE_LEVEL_FUNCTION( vkDestroySwapchainKHR )
VK_DEVICE_LEVEL_FUNCTION( vkGetSwapchainImagesKHR )
VK_DEVICE_LEVEL_FUNCTION( vkAcquireNextImageKHR )
VK_DEVICE_LEVEL_FUNCTION( vkQueuePresentKHR )
#endif
```

10.*ListOfFunctions.inl*

你可再次看到，我们正在检查是否定义了 USE_SWAPCHAIN_EXTENSIONS 预处理器指令。 我只在启用交换链扩展的项目中定义。

由于我们创建了逻辑设备，所以需要接收图形队列和演示队列（如果分开）的句柄。 为方便起见，我使用两个单独的队列变量，但它们都包含相同的句柄。

加载设备层函数后，我们读取请求的队列句柄。 以下是相应的代码：

```
vkGetDeviceQueue( Vulkan.Device, Vulkan.GraphicsQueueFamilyIndex, 0, &Vulkan.GraphicsQueue );
vkGetDeviceQueue( Vulkan.Device, Vulkan.PresentQueueFamilyIndex, 0, &Vulkan.PresentQueue );
```

```
return true;
```

11.*Tutorial02.cpp，函数 GetDeviceQueue()*

创建旗语 (semaphore)

创建和使用交换链之前的最后一个步骤是创建旗语。 旗语指用于队列同步化的对象。 其中包含信号旗语和无信号旗语。 如果部分操作已完成，其中一个队列将发出旗语信号（将状态从无信号改成信号），另一队列将等待该旗语直至其变成信号旗语。 之后，队列重新执行通过命令缓冲区提交的操作。

```
VkSemaphoreCreateInfo semaphore_create_info = {
  VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,     // VkStructureType         sType
  nullptr,                                     // const void*             pNext
  0                                            // VkSemaphoreCreateFlags  flags
};

if( (vkCreateSemaphore( Vulkan.Device, &semaphore_create_info, nullptr, &Vulkan.ImageAvailableSemaphore ) != VK_SUCCESS) ||
    (vkCreateSemaphore( Vulkan.Device, &semaphore_create_info, nullptr, &Vulkan.RenderingFinishedSemaphore ) != VK_SUCCESS) ) {
  std::cout << "Could not create semaphores!" << std::endl;
  return false;
```

```
}

return true;
```

12.*Tutorial02.cpp，函数 CreateSemaphores()*

我们调用 **vkCreateSemaphore()** 函数，以创建旗语。 它要求提供包含三个字段的创建信息：

- sType – 标准结构类型，在此示例中必须设置为 VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO。
- pNext – 留作将来使用的标准参数。
- flags – 另一留作将来使用的参数，必须是 0。

旗语在绘制过程中（如果希望提高精确性，在演示过程中）使用。 稍后将详细介绍。

创建交换链

我们启用了交换链支持，但在屏幕上进行渲染之前，必须首先创建交换链，以获取图像，进行渲染（或在渲染至其他图像时进行拷贝）。

为创建交换链，我们调用 **vkCreateSwapchainKHR()** 函数。 它要求我们提供类型变量 VkSwapchainCreateInfoKHR 的地址，告知驱动程序将创建的交换链的属性。 为使用相应的值填充该结构，我们必须确定特定硬件和平台上的内容。 为此我们查询平台或窗口有关可用性和兼容不同特性的属性，即支持的图像格式或演示模式（如何在屏幕上演示图像）。 因此在创建交换链之前，我们必须查看特定平台的内容，以及如何创建交换链。

获取平面功能

首先必须查询平面功能。 为此，我们调用 **vkGetPhysicalDeviceSurfaceCapabilitiesKHR()** 函数，如下所示：

```
VkSurfaceCapabilitiesKHR surface_capabilities;

if( vkGetPhysicalDeviceSurfaceCapabilitiesKHR( Vulkan.PhysicalDevice, Vulkan.PresentationSurface, &surface_capabilities ) != VK_SUCCESS ) {

  std::cout << "Could not check presentation surface capabilities!" << std::endl;
```

```
  return false;
}
```

13.*Tutorial02.cpp，函数 CreateSwapChain()*

获取的功能包含有关交换链支持范围（限制）的重要信息，即图像的最大数和最小数、图像的最小尺寸和最大尺寸，或支持的转换格式（有些平台可能要求在演示图像之前首先进行图像转换）。

获取支持的平面格式

下一步我们需要查询支持的平面格式。 并非所有平台都兼容常见的图像格式，比如非线性 32 位 RGBA。 有些平台没有任何偏好，但有些仅支持少数几种格式。 我们仅选择其中一种适用于交换链的可用格式，否则创建将会失败。

要查询平面格式，我们必须调用 **vkGetPhysicalDeviceSurfaceFormatsKHR()** 函数。 我们可以像往常一样调用两次：第一次获取支持格式的数量，第二次获取阵列（为达到该目的而准备）中的支持格式。 可通过以下命令来实现：

```
uint32_t formats_count;
if( (vkGetPhysicalDeviceSurfaceFormatsKHR( Vulkan.PhysicalDevice, Vulkan.PresentationSurface, &formats_count, nullptr ) != VK_SUCCESS) ||
    (formats_count == 0) ) {
  std::cout << "Error occurred during presentation surface formats enumeration!" << std::endl;
  return false;
}
```

```
std::vector<VkSurfaceFormatKHR> surface_formats( formats_count );
if( vkGetPhysicalDeviceSurfaceFormatsKHR( Vulkan.PhysicalDevice, Vulkan.PresentationSurface, &formats_count, &surface_formats[0] ) != VK_SUCCESS ) {
  std::cout << "Error occurred during presentation surface formats enumeration!" << std::endl;
  return false;
}
```

14.*Tutorial02.cpp，函数 CreateSwapChain()*

获取支持的演示模式

我们还应请求可用的演示模式，告知我们如何在屏幕上演示（显示）图像。 演示模式定义应用是等待垂直同步，还是在可用时立即显示图像（可能会造成图像撕裂）。 稍后我将介绍不同的演示模式。

为查询特定平台支持的演示模式，我们调用 **vkGetPhysicalDeviceSurfacePresentModesKHR()** 函数。 我们可以创建与以下内容类似的代码：

```
uint32_t present_modes_count;
if( (vkGetPhysicalDeviceSurfacePresentModesKHR( Vulkan.PhysicalDevice, Vulkan.PresentationSurface, &present_modes_count, nullptr ) != VK_SUCCESS) ||
    (present_modes_count == 0) ) {
  std::cout << "Error occurred during presentation surface present modes enumeration!" << std::endl;
  return false;
```

```
}

std::vector<VkPresentModeKHR> present_modes( present_modes_count );

if( vkGetPhysicalDeviceSurfacePresentModesKHR( Vulkan.PhysicalDevice, Vulkan.PresentationSurface, &present_modes_count, &present_modes[0] ) != VK_SUCCESS ) {

 std::cout << "Error occurred during presentation surface present modes enumeration!" << std::endl;

 return false;

}
```

15.*Tutorial02.cpp，函数 CreateSwapChain()*

现在我们获取了所有相关数据，可帮助我们为创建交换链准备相应的数值。

选择交换链图像数量

交换链包含多个图像。 获取多个图像（通常超过一个）可帮助演示引擎正常运行，即一个图像在屏幕上演示，另一个图像等待查询下一垂直同步，而第三个图像用于应用渲染。

应用可能会请求更多图像。 如果希望，它可以一次使用多个图像，例如，为视频流编码时，第四个图像是关键帧，应用需要它来准备剩下的三个帧。 这种用法将决定在交换链中自动创建的图像数量：应用一次要求处理多少个图像，以及演示引擎要求多少个图像才能正常运行。

但我们必须确保请求的交换链图像数量不少于所需图像的最小值，也不超过支持图像的最大值（如果存在这种数量限制条件）。 而且，图像过多会要求使用更大的内存。 另一方面，图像过少，会导致应用中出现停顿（稍后详细介绍）。

交换链正常运行以及应用进行渲染所需的图像数量由平面功能定义。 以下部分代码可检查图像数量是否在允许的最小值和最大值之间：

```
// Set of images defined in a swap chain may not always be available for application to render to:
// One may be displayed and one may wait in a queue to be presented
// If application wants to use more images at the same time it must ask for more images
uint32_t image_count = surface_capabilities.minImageCount + 1;
if( (surface_capabilities.maxImageCount > 0) &&
    (image_count > surface_capabilities.maxImageCount) ) {
  image_count = surface_capabilities.maxImageCount;
}
return image_count;
```

16.*Tutorial02.cpp，函数 GetSwapChainNumImages()*

平面功能结构中的 minImageCount 值提供交换链正常运行所需的最少图像。 此处我们选择比要求多一个的图像数量，并检查是否请求太多。 多出的一个图像可用于类似三次缓冲的演示模式（如果可用）。 在高级场景中，我们还要求保存希望（一次）同时使用的图像数量。 我们想要编码之前提到过的视频流，因此需要一个关键帧（每第四个图像帧）和其他三个图像。 但交换链不允许应用一次操作四个图像 — 只能操作三个。 我们必须了解这种情况，因为我们仅通过关键帧准备了两个帧，然后我们需要释放它们（让其返回至演示引擎）并获取最后第三个非关键帧。 这一点稍后会更加清晰。

选择交换链图像格式

选择图像格式取决于我们希望执行的处理/渲染类型，即如果我们想混合应用窗口和桌面内容，可能需要一个阿尔法值。 我们还必须知道哪种色域可用，以及我们是否在线性或 sRGB 色域上操作。

平台支持的格式-色域配对数量各不相同。 如果希望使用特定数量，必须确保它们可用。

```
// If the list contains only one entry with undefined format
// it means that there are no preferred surface formats and any can be chosen
if( (surface_formats.size() == 1) &&
   (surface_formats[0].format == VK_FORMAT_UNDEFINED) ) {
 return{ VK_FORMAT_R8G8B8A8_UNORM, VK_COLORSPACE_SRGB_NONLINEAR_KHR };
}

// Check if list contains most widely used R8 G8 B8 A8 format
// with nonlinear color space
for( VkSurfaceFormatKHR &surface_format : surface_formats ) {
 if( surface_format.format == VK_FORMAT_R8G8B8A8_UNORM ) {
   return surface_format;
```

```
}
}

// Return the first format from the list
return surface_formats[0];
```

17.*Tutorial02.cpp，函数 GetSwapChainFormat()*

之前我们请求了放在阵列中的支持格式（在本示例中为矢量）。 如果该阵列仅包含一个值和一个未定义的格式，该平台将没有任何偏好。 我们可以使用任何图像格式。

在其他情况下，我们只能使用一种可用的格式。 这里我正在查找任意（线性或非线性）32 位 RGBA 格式。 如果可用，就可以选择。 如果没有这种格式，我将使用列表中的任意一种格式（希望第一个是最好的，同时也是精度最高的格式）。

选择交换链图像大小

交换链图像的大小通常与窗口大小相同。 我们可以选择其他大小，但必须符合图像大小限制。 符合当前应用窗口大小的图像大小以“currentExtent”成员的形式在平面功能结构中提供。

值得注意的是，特定值“-1”表示应用窗口大小由交换链大小决定，因此我们选择所希望的任意尺寸。 但必须确保所选尺寸不小于，也不大于定义的最小限值和最大限值。

选择交换链大小可能（通常）如下所示：

```
// Special value of surface extent is width == height == -1
// If this is so we define the size by ourselves but it must fit within defined confines
```

```
if( surface_capabilities.currentExtent.width == -1 ) {
  VkExtent2D swap_chain_extent = { 640, 480 };
  if( swap_chain_extent.width < surface_capabilities.minImageExtent.width ) {
    swap_chain_extent.width = surface_capabilities.minImageExtent.width;
  }
  if( swap_chain_extent.height < surface_capabilities.minImageExtent.height ) {
    swap_chain_extent.height = surface_capabilities.minImageExtent.height;
  }
  if( swap_chain_extent.width > surface_capabilities.maxImageExtent.width ) {
    swap_chain_extent.width = surface_capabilities.maxImageExtent.width;
  }
  if( swap_chain_extent.height > surface_capabilities.maxImageExtent.height ) {
    swap_chain_extent.height = surface_capabilities.maxImageExtent.height;
```

```
  }
  return swap_chain_extent;
}

// Most of the cases we define size of the swap_chain images equal to current window's size
return surface_capabilities.currentExtent;
```

18.*Tutorial02.cpp，函数 GetSwapChainExtent()*

选择交换链用法标记

用法标记定义如何在 Vulkan 中使用特定图像。 如果想对图像进行抽样（使用内部着色器），必须将它创建成“sampled”用法。 如果图像用作深度渲染对象，必须将它创建成“depth and stencil”用法。 没有“启用”相应用法的图像不能用于特定目的，否则不会定义此类操作结果。

对（大多数情况下）希望渲染其图像的交换链（用作渲染对象）来说，必须用 VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT enum 指定“color attachment”用法。 在 Vulkan 中，这种用法通常适用于交换链，因此通常在不进行其他检查的情况下对其进行设置。 但对于其他用法来说，我们必须确保它受到支持 — 可以通过平面功能结构的“supportedUsageFlags”成员来完成。

```
// Color attachment flag must always be supported
// We can define other usage flags but we always need to check if they are supported
if( surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_TRANSFER_DST_BIT ) {
```

```
  return VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT;
}
std::cout << "VK_IMAGE_USAGE_TRANSFER_DST image usage is not supported by the swap chain!" << std::endl
  << "Supported swap chain's image usages include:" << std::endl
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_TRANSFER_SRC_BIT             ? "    VK_IMAGE_USAGE_TRANSFER_SRC\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_TRANSFER_DST_BIT             ? "    VK_IMAGE_USAGE_TRANSFER_DST\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_SAMPLED_BIT                  ? "    VK_IMAGE_USAGE_SAMPLED\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_STORAGE_BIT                  ? "    VK_IMAGE_USAGE_STORAGE\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT         ? "    VK_IMAGE_USAGE_COLOR_ATTACHMENT\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT  ? "    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT      ? "    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT\n" : "")
  << (surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT          ? "    VK_IMAGE_USAGE_INPUT_ATTACHMENT" : "")
```

```
 << std::endl;

return static_cast<VkImageUsageFlags>(-1);
```

19.*Tutorial02.cpp，函数 GetSwapChainUsageFlags()*

在本示例中，我们定义图像清晰操作所需的其他“transfer destination”用法。

选择预转换

在有些平台上，我们可能希望对图像进行转换。 这种情况通常发生在朝向某一方位（而非默认方位）的平板电脑上。 在交换链创建期间，必须在图像演示之前指定应用于图像的转换方式。 当然，我们可以仅使用支持的转换，位于获取的平面功能的“supportedTransforms”成员之中。

如果选择的预转换不是（平面功能中的）当前转换，演示引擎将使用所选择的转换方式。 在部分平台上，这一操作可能会造成性能下降（可能不明显，但需要注意）。 在示例代码中，我不想进行任何转换，但必须检查是否支持转换。 如果不支持，我仅使用当前使用的相同转换。

```
// Sometimes images must be transformed before they are presented (i.e. due to device's orienation

// being other than default orientation)

// If the specified transform is other than current transform, presentation engine will transform image

// during presentation operation; this operation may hit performance on some platforms

// Here we don't want any transformations to occur so if the identity transform is supported use it

// otherwise just use the same transform as current transform

if( surface_capabilities.supportedTransforms & VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR ) {
```

```
  return VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
} else {
  return surface_capabilities.currentTransform;
}
```

20.*Tutorial02.cpp，函数 GetSwapChainTransform()*

选择演示模式

演示模式决定图像在演示引擎内部处理以及在屏幕上显示的方式。 过去，从头到尾仅显示单个缓冲区。 如果在上面绘图，绘制操作（整个图像创建流程）是可视的。

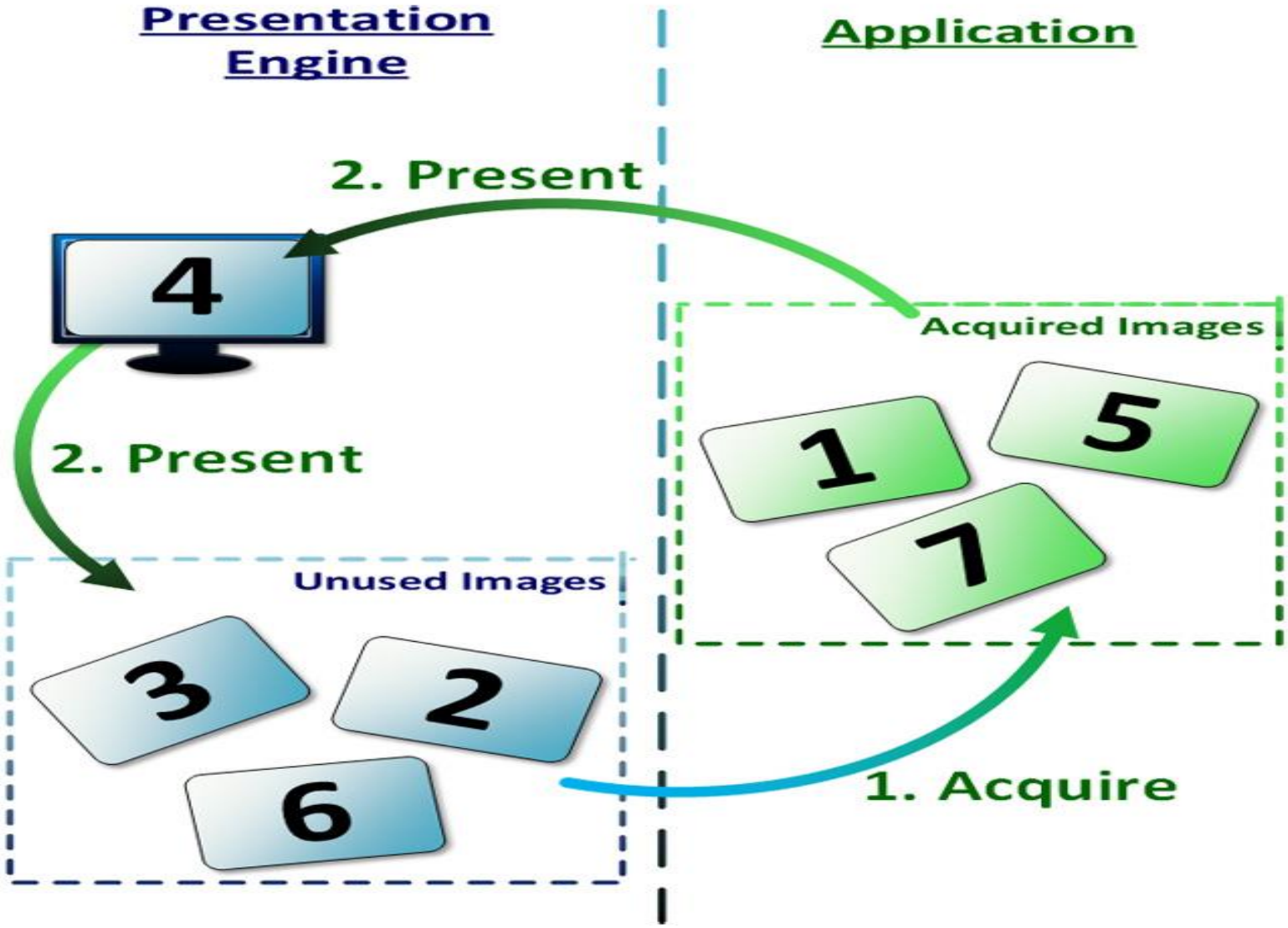
双缓冲的推出，可防止查看绘制操作：一个图像显示，第二个用于渲染。 在演示过程中，第二个图像的内容拷贝至第一个图像（之前），或（之后）两个图像互换（还记得 OpenGL 应用中使用的 SwapBuffers() 函数吗？），这意味着它们的指示器已经互换。

撕裂是图像显示遇到的另一个问题，我们想避免这一问题，因此推出了等待垂直回扫信号的功能。 但等待又引发了另一问题：输入延迟。 因此双缓冲换成了三次缓冲，我们可以轮流绘制进两个后台缓冲区，而且在垂直同步期间，最新的一个用于演示。

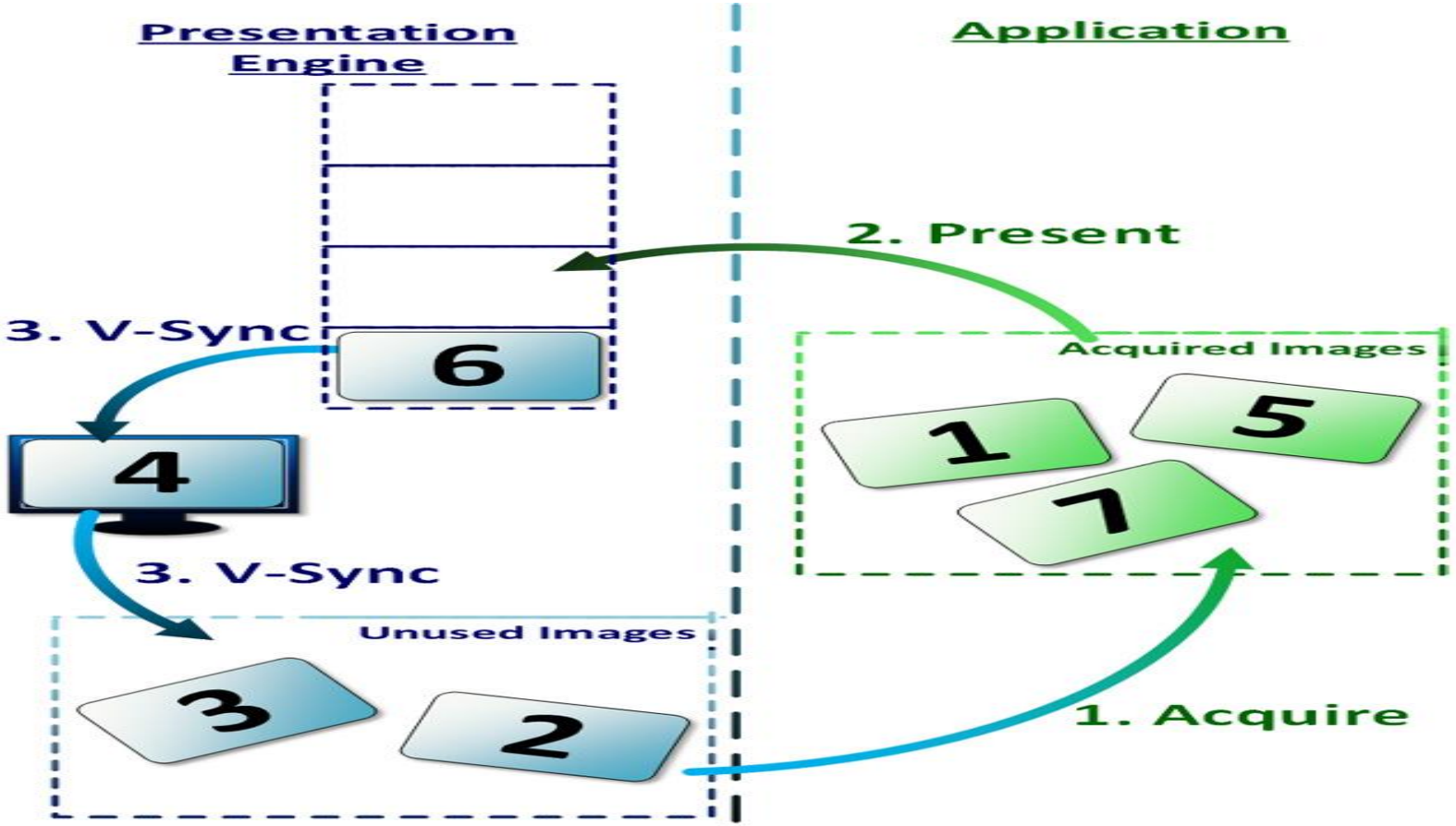
这就是演示的目的所在：如何处理这些问题、如何在屏幕上演示图像，以及是否希望使用垂直同步。

目前主要有四种演示模式：

- IMMEDIATE。 演示请求立即采用，可能会出现撕裂问题（取决于每秒帧速率）。 在内部，演示引擎不使用任何保持交换链图像的队列。


Presentation Engine
Application
2. Present
4
Acquired Images
1
5
7
2. Present
Unused Images
3
2
6
1. Acquire

- FIFO。 该模式与 OpenGL 的缓冲交换类似，交换间隔设为 1。 图像仅在垂直回扫期间显示（替换当前显示的图像），因此不会出现撕裂。 在内部，演示引擎使用带有“numSwapchainImages – 1”要素的队列。 演示请求附在队列的末尾。 回扫期间，队列开头的图像替换当前显示的图像，适用于应用。 如果所有图像都在队列中，那么应用必须等待垂直同步释放当前显示的图像。 只有这样，它才适用于渲染图像的应用和程序。 该模式必须始终适用于所有支持交换链扩展的 Vulkan 实施。



使用哪种演示模式取决于欲执行的操作类型。如果想解码和播放电影，我们希望以相应的顺序显示所有的帧。 因此在我看来，FIFO 模式是最佳选择。 但如果我们创建游戏，通常希望显示最新生成的帧。 在这种情况下，我建议使用 MAILBOX，因为该模式不会出现撕裂问题，而且其输入延迟最低。 显示最新生成的图像，应用不需要等待垂直同步。 但是为了实现这种行为，必须创建至少三个图像，而且并不总支持这种模式。

FIFO 模式始终可用，并且要求支持两个图像，但导致应用需等待垂直同步（无论请求多少交换链图像）。 即时模式的速度最快 据我所知，它也要求两个图像，但不会让应用等待显示器刷新率。 它的劣势是会造成图像撕裂。 如何选择在于您，像往常一样，我们必须确保所选的演示模式受到支持。

之前我们查询了可用演示模式，因此现在必须寻找最能满足我们需求的模式。 以下是用于寻找 MAILBOX 模式的代码：

Copy CodeDarkLight

```
// FIFO present mode is always available
// MAILBOX is the lowest latency V-Sync enabled mode (something like triple-buffering) so use it if available
for( VkPresentModeKHR &present_mode : present_modes ) {
   if( present_mode == VK_PRESENT_MODE_MAILBOX_KHR ) {
      return present_mode;
   }
}
for( VkPresentModeKHR &present_mode : present_modes ) {
   if( present_mode == VK_PRESENT_MODE_FIFO_KHR ) {
      return present_mode;
   }
}
std::cout << "FIFO present mode is not supported by the swap chain!" << std::endl;
return static_cast<VkPresentModeKHR>(-1);
```

21.Tutorial02.cpp，函数 GetSwapChainPresentMode()

创建交换链

现在我们拥有创建交换链需要的所有数据。 我们定义了所有所需的值，并确保它们满足特定平台的限制条件。

```
uint32_t                                desired_number_of_images = GetSwapChainNumImages( surface_capabilities );
VkSurfaceFormatKHR                      desired_format = GetSwapChainFormat( surface_formats );
VkExtent2D                              desired_extent = GetSwapChainExtent( surface_capabilities );
VkImageUsageFlags                       desired_usage = GetSwapChainUsageFlags( surface_capabilities );
VkSurfaceTransformFlagBitsKHR desired_transform = GetSwapChainTransform( surface_capabilities );
VkPresentModeKHR                        desired_present_mode = GetSwapChainPresentMode( present_modes );
VkSwapchainKHR                          old_swap_chain = Vulkan.SwapChain;

if( static_cast<int>(desired_usage) == -1 ) {
  return false;
}
if( static_cast<int>(desired_present_mode) == -1 ) {
  return false;
}

VkSwapchainCreateInfoKHR swap_chain_create_info = {
  VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR,    // VkStructureType                  sType
  nullptr,                                        // const void                      *pNext
  0,                                              // VkSwapchainCreateFlagsKHR        flags
  Vulkan.PresentationSurface,                     // VkSurfaceKHR                     surface
```

```
    desired_number_of_images,                       // uint32_t                         minImageCount
    desired_format.format,                         // VkFormat                         imageFormat
    desired_format.colorSpace,                     // VkColorSpaceKHR                  imageColorSpace
    desired_extent,                                 // VkExtent2D                       imageExtent
    1,                                                 // uint32_t                         imageArrayLayers
    desired_usage,                                 // VkImageUsageFlags                imageUsage
    VK_SHARING_MODE_EXCLUSIVE,                        // VkSharingMode                        imageSharingMode
    0,                                                 // uint32_t                         queueFamilyIndexCount
    nullptr,                                       // const uint32_t                   *pQueueFamilyIndices
    desired_transform,                             // VkSurfaceTransformFlagBitsKHR    preTransform
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR,                // VkCompositeAlphaFlagBitsKHR      compositeAlpha
    desired_present_mode,                          // VkPresentModeKHR                 presentMode
    VK_TRUE,                                          // VkBool32                         clipped
    old_swap_chain                                 // VkSwapchainKHR                   oldSwapchain
};

if( vkCreateSwapchainKHR( Vulkan.Device, &swap_chain_create_info, nullptr, &Vulkan.SwapChain ) != VK_SUCCESS ) {
    std::cout << "Could not create swap chain!" << std::endl;
    return false;
}
if( old_swap_chain != VK_NULL_HANDLE ) {
    vkDestroySwapchainKHR( Vulkan.Device, old_swap_chain, nullptr );
}

return true;
```

22.Tutorial02.cpp，函数 CreateSwapChain()

在本代码示例中，一开始我们收集了之前介绍的所有必要数据。 接下来创建类型 VkSwapchainCreateInfoKHR 的变量。 它包括以下成员：

sType – 标准结构类型，此处必须为 VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR。
pNext – 留作将来使用的指示器（用于指向此次扩展的部分扩展）。
flags – 留作将来使用的值，目前必须设为 0。
surface – 已创建的表示窗口系统（我们的应用窗口）的平面的句柄。
minImageCount – 应用为交换链请求的图像的最小数量（必须满足提供的限制条件）。
imageFormat – 应用针对交换链图像选择的格式；必须是受支持的平面格式。
imageColorSpace – 交换链图像的色域；仅列举的格式-色域配对的值可用于 imageFormat 和 imageColorSpace（不能使用一个配对的格式和另一配对的色域）。
imageExtent – 以像素形式定义的交换链图像的大小（尺寸）；必须满足提供的限制条件。
imageArrayLayers – 定义交换链图像的层数（即视图）；该数值通常为 1，但如果我们希望创建多视图或立体（立体 3D）图，可以将其设为更大的值。
imageUsage – 定义应用如何使用图像；可包含仅支持用法的数值；通常支持 color attachment 用法。
imageSharingMode – 在多个队列引用图像时描述图像共享模式（稍后将详细介绍）。
queueFamilyIndexCount – 供引用交换链图像的不同队列家族数量；该参数仅在使用 VK_SHARING_MODE_CONCURRENT 共享模式时重要。
pQueueFamilyIndices – 包含队列家族（将引用交换链图像）的索引阵列；必须至少包含 queueFamilyIndexCount 个要素，而且因为在 queueFamilyIndexCount 中，该参数仅在使用 VK_SHARING_MODE_CONCURRENT 共享模式时重要。
preTransform – 在演示图像时应用于交换链图像的转换；必须是支持的数值之一。
compositeAlpha – 该参数用于指示平面（图像）如何与相同窗口系统上的其他平面进行合成（混合？）；该数值必须是平面功能中返回的值（位），但看起来似乎始终支持不透明合成（没有混合，阿尔法忽略）（因为大多数游戏希望使用这种模式）。
presentMode – 交换链将使用的演示模式；只能选择支持的模式。
clipped – 连接像素所有权；一般来说，如果应用不希望读取交换链图像（比如 ReadPixels()），应设为 VK_TRUE，因为它将支持部分平台使用更好的演示方法；在特定场景下使用 VK_FALSE 值（我了解更多信息后再来介绍这些场景）。
oldSwapchain – 如果重新创建交换链，该参数将定义将之前的交换链替换为新创建的交换链。
那么使用这种共享模式会出现什么问题？ 队列可以引用 Vulkan 中的图像。 这意味着我们可以创建命令来使用这些图像。 这些命令保存在命令缓冲区

内，而这些缓冲区将提交至队列。 队列属于不同的队列家族。 Vulkan 要求我们声明队列家族的数量，以及其中哪些家族将通过提交至命令缓冲区中的命令引用这些图像。

如果希望，我们可以一次引用不同队列家族的图像。 在这种情况下，我们必须提供“并发”共享模式。 但这（可能）要求我们自己管理图像数据的一致性，即我们必须同步不同的阵列，同时确保图像中的数据合理而且不会出现任何不利影响 — 部分队列正在读取图像的数据，而其他队列还未完成写入。

我们可以不指定这些队列家族，只告诉 Vulkan 一次仅一个队列家族（一个家族的队列）引用图像。 这并不意味着其他队列不会引用这些图像。 仅意味着它们不会一次性同时进行。 因此如果我们希望引用一个家族的图像，然后引用另一家族的图像，那么必须明确告诉 Vulkan： “我的图像可在该队列家族内部使用，但从现在起其他家族（这个）将引用它。” 使用图像内存壁垒能够完成这种过渡。 当一次只有一个队列家族使用特定图像时，可以使用“专有”共享模式。

如果不满足这些要求，将出现未定义行为，而且我们无法信赖图像内容。

在本示例中，我们使用一个队列，因此不必指定“并发”共享模式，并使相关参数（queueFamilyCount 和 pQueueFamilyIndices）保持空白（或 null 或零值）状态。

现在我们可以调用 vkCreateSwapchainKHR() 函数创建交换链并查看该操作是否成功。 之后（如果我们重新创建交换链，表示这不是第一次创建）我们应该毁坏之前的交换链。 稍后将对此进行介绍。

图像演示

现在我们有包含多个图像的交换链。 为将这些图像用作渲染对象，我们可以获取用交换链创建的所有图像的句柄，但不允许那样使用。 交换链图像属于交换链。 这意味着应用不能直接使用这些图像，必须发出请求。 还意味着图像由平台和交换链（而非应用）一起创建和毁坏。

因此如果应用希望渲染交换链图像或以其他的方式使用，必须首先通过请求交换链以访问该图像。 如果交换链要求等待，那么我们必须等待。 应用使

用完图像后，应通过演示“返回”该图像。 如果忘记将图像返回至交换链，图像会很快用完，屏幕上将什么也不显示。

应用还可以请求一次访问更多图像，但它们必须处于可用状态。 获取访问可能要求等待。 在一些极端情况下，交换链中的图像不够，而应用希望访问多个图像，或者如果我们忘记将图像返回至交换链，应用甚至可能会无限期等待下去。

倘若（通常）至少有两个图像，我们还需要等待，这听起来似乎很奇怪，但这是有原因的。 并不是所有图像都可用于应用，因为演示引擎也要使用图像。 通常显示一个图像。 演示引擎可能需要使用其他图像，才能保持正常运行。 所以我们不能使用它们，因为这可能会在一定程度上阻碍演示引擎的运行。 我们不知道其内部机制和算法，也不知道供应用执行的操作系统的要求。 因此图像的可用性取决于多个因素：内部实施、操作系统、已创建图像数量、应用希望单次并以所选演示模式使用的图像数量（从本教程的角度来说，这是最重要的因素）。

在即时模式中，通常演示一个图像。 其他图像（至少一个）可用于应用。 当应用发布演示请求（“返回”图像），显示的图像将替换为新图像。 因此，如果创建两个图像，应用一次仅可使用一个图像。 如果应用请求其他图像，则必须“返回”之前的图像。 如果希望一次使用两个图像，则必须创建包含多个图像的交换链，否则将一直处于等待状态。 在即时模式中，如果我们请求多个图像，应用将一次请求（拥有）“imageCount － 1”个图像。

在 FIFO 模式中，显示一个图像，其余图像放在 FIFO 队列中。 该队列的长度通常等于“imageCount － 1”。 一开始，所有图像可能都可用于应用（因为队列是空的，没有任何图像）。 当应用演示图像（将其“返回”至交换链）时，该图像将附在队列末尾。 因此，队列变满后，应用需要等待其他图像，直至垂直回扫阶段释放出所显示的图像。 图像的显示顺序通常与应用的演示顺序相同。 如果出现垂直同步信号，该队列的第一个图像将替换显示的图像。 之前显示的图像（释放的图像）可用于应用，因为它成了未使用的图像（不演示，也不在队列中等待）。 如果所有的图像都在列队中，应用将等待下一个回扫期以访问其他图像。如果渲染时间长于刷新率，应用将不需要等待。如果有多个图像，该行为不会变化。内部交换链队列通常包含“imageCount － 1”个要素。

当前最后一个可用的模式是 MAILBOX。 如前所述，该模式与“传统”三次缓冲最为类似。 通常显示一个图像。 第二个图像在单要素队列中等待（通常仅有容纳一个要素的位置）。 其他图像可用于应用。 应用演示图像时，该图像将替换在队列中等待的那个图像。 队列中的图像仅在回扫期间显示，但应用无需等待下一个图像（如果有超过两个图像）。 只有两个图像时，MAILBOX 模式的运行方式与 FIFO 模式相同 — 应用必须等待垂直同步信号以获取下一个图像。 但如果有至少三个图像，它可立即获取由“演示”图像（队列中等待的图像）替换的那个图像。 这就是我所请求的图像比最小值多一个的原因。 如果 MAILBOX 模式可用，我希望以与三次缓冲相同的方式使用该模式（可能第一件事是检查哪种模式可用，然后根据所选的演示模式选择

交换链图像的数量）。

我希望这些示例可帮助您了解，如果应用希望使用任意图像时，为什么必须请求图像。 在 Vulkan 中，我们只能执行允许和要求的行为 — 不能太少，也不能太多。

```
uint32_t image_index;
VkResult result = vkAcquireNextImageKHR( Vulkan.Device, Vulkan.SwapChain, UINT64_MAX, Vulkan.ImageAvailableSemaphore, VK_NULL_HANDLE, &image_index );
switch( result ) {
  case VK_SUCCESS:
  case VK_SUBOPTIMAL_KHR:
    break;
  case VK_ERROR_OUT_OF_DATE_KHR:
    return OnWindowSizeChanged();
  default:
    std::cout << "Problem occurred during swap chain image acquisition!" << std::endl;
    return false;
}
```

23.Tutorial02.cpp，函数 Draw()

为获取图像，我们必须调用 vkAcquireNextImageKHR() 函数。 在调用期间，我们必须指定（除像其他函数中的设备句柄外）交换链，以通过其使用图像、超时 (timeout)、旗语和围栏对象。 如果成功，函数将图像索引保存在我们提供地址的变量中。 为何保存索引，而不保存图像（句柄）本身？ 这种行为非常方便（即在“预处理”阶段，当我们希望为渲染准备尽可能多的数据时，这样在典型帧渲染期间就不会浪费时间），不过这点我们稍后讨论。 只需记住，我们可以查看交换链中创建了哪些图像（需得到允许才能使用）。 进行这种查询时，会提供图像阵列。 而且 vkAcquireNextImageKHR() 函数将索引保存在该阵列中。

我们必须指定超时，因为有时候图像不能立即使用。 未得到允许就尝试使用图像会导致未定义行为。 指定超时可为演示引擎提供反应时间。 如果需要，可以等待下一垂直回扫期，而且我们提供时间。 因此如果没超过指定时间，该函数将会中断。 我们可提供最大可用值，因此该函数可能会无限期中断下去。 如果我们提供的超时为 0，该函数会立即返回。 如果调用时图像可用，将会立即提供图像。 如果没有可用图像，将会返回错误，提示图像尚未准备就绪。

获取图像后，我们可以任意使用该图像。 通过保存在命令缓冲区的命令，可以处理或引用图像。 我们可以事先准备命令缓冲区（以节省渲染的处理过程）并在此使用或提交。 或者，我们还可以现在准备命令，完成后进行提交。 在 Vulkan 中，创建命令缓冲区并将其提交至队列是支持设备执行操作的唯一方式。

命令缓冲区提交至队列后，所有命令开始处理。 但如果没有得到允许，队列不能使用图像，我们之前创建的旗语用于内部队列同步 — 队列开始处理引用特定图形的命令之前，应等待旗语（直到获得信号）。 但这种等待不会中断应用。 用于访问交换链图像的同步机制有两种： (1) 超时 — 可能会中断应用，但不会中止队列处理；(2) 旗语 — 不会中断应用，但会中断所选的队列。

现在我们（从理论上来说）了解了如何（通过命令缓冲区）进行渲染。 现在我们想像一下，我们正在命令缓冲区内部提交，部分渲染操作已在执行。 但开始处理之前，我们应让队列（执行渲染的地方）等待。 这一过程在提交操作中完成。

```
VkPipelineStageFlags wait_dst_stage_mask = VK_PIPELINE_STAGE_TRANSFER_BIT;
VkSubmitInfo submit_info = {
  VK_STRUCTURE_TYPE_SUBMIT_INFO,                  // VkStructureType              sType
  nullptr,                                        // const void                  *pNext
  1,                                              // uint32_t                     waitSemaphoreCount
  &Vulkan.ImageAvailableSemaphore,                // const VkSemaphore           *pWaitSemaphores
  &wait_dst_stage_mask,                           // const VkPipelineStageFlags  *pWaitDstStageMask;
  1,                                              // uint32_t                     commandBufferCount
  &Vulkan.PresentQueueCmdBuffers[image_index],    // const VkCommandBuffer       *pCommandBuffers
  1,                                              // uint32_t                     signalSemaphoreCount
```

```
    &Vulkan.RenderingFinishedSemaphore                  // const VkSemaphore                *pSignalSemaphores
};

if( vkQueueSubmit( Vulkan.PresentQueue, 1, &submit_info, VK_NULL_HANDLE ) != VK_SUCCESS ) {
    return false;
}
```

24.Tutorial02.cpp，函数 Draw()

首先我们准备一个结构，其中包含欲提交至队列的操作类型的信息。 这可通过 VkSubmitInfo 结构来完成。 它包含以下字段：

sType – 标准结构类型；此处必须设置为 VK_STRUCTURE_TYPE_SUBMIT_INFO。
pNext – 留作将来使用的标准指示器。
waitSemaphoreCount – 我们希望队列在开始处理命令缓冲区的命令之前所等待的旗语数量。
pWaitSemaphores – 包含队列应等待的旗语句柄的阵列指示器；该阵列必须包含至少 waitSemaphoreCount 个要素。
pWaitDstStageMask – 要素数量与 pWaitSemaphores 阵列相同的阵列指示器；它描述每个（相应）旗语等待将出现时所处的管道阶段；在本示例中，该队列可在开始使用交换链的图像之前执行部分操作，因此不会中断所有操作；该队列可开始处理部分绘制命令，而且必须等待管道进入使用图像的阶段。
commandBufferCount – 提交以待执行的命令缓冲区数量。
pCommandBuffers – 包含命令缓冲区句柄的阵列（必须包含至少 commandBufferCount 个要素）的指示器。
signalSemaphoreCount – 我们希望队列在处理完所有提交的命令缓冲区后发出信号的旗语数量。
pSignalSemaphores – 至少包含 signalSemaphoreCount 个要素和旗语句柄的阵列指示器；队列处理完在该提交信息内提交的命令后，将向这些旗语发出信号。
在本示例中，我们告诉队列仅等待一个旗语，其信号由演示引擎在队列安全开始处理引用交换链图像的命令时发出。

而且，我们仅提交一个简单的命令缓冲区。 该缓冲区是之前已经准备好的（稍后将介绍如何准备）。 它仅清空获取的图像。 但这已经足够我们查看在应用窗口中选择的颜色，以及交换链是否正常运行。

在上述代码中，命令缓冲区安排在阵列（更准确的说是矢量）中。 为简化提交相应命令缓冲区（引用当前获取的图像）的流程，我为每个交换链图像准备了一个单独的命令缓冲区。 此处可使用 vkAcquireNextImageKHR() 函数提供的图像索引。 （在类似场景中）使用图像句柄要求创建地图，以将句柄转换成特定命令缓冲区或索引。 另一方面，仅选择一个特定阵列要素时，我们可使用范数。 因此该函数为我们提供的是索引，而非图像句柄。

提交命令缓冲区后，所有处理过程在后台“硬件”上开始进行。 下一步我们希望演示渲染的图像。 演示表示我们希望显示图像并将其“交还”至交换链。 用于完成这一步骤的代码如下所示：

```
VkPresentInfoKHR present_info = {
   VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,              // VkStructureType              sType
   nullptr,                                         // const void                 *pNext
   1,                                               // uint32_t                     waitSemaphoreCount
   &Vulkan.RenderingFinishedSemaphore,              // const VkSemaphore          *pWaitSemaphores
   1,                                               // uint32_t                     swapchainCount
   &Vulkan.SwapChain,                               // const VkSwapchainKHR        *pSwapchains
   &image_index,                                    // const uint32_t              *pImageIndices
   nullptr                                          // VkResult                    *pResults
};
result = vkQueuePresentKHR( Vulkan.PresentQueue, &present_info );

switch( result ) {
   case VK_SUCCESS:
      break;
   case VK_ERROR_OUT_OF_DATE_KHR:
   case VK_SUBOPTIMAL_KHR:
      return OnWindowSizeChanged();
   default:
```

```
    std::cout << "Problem occurred during image presentation!" << std::endl;
    return false;
}

return true;
```

25.Tutorial02.cpp，函数 Draw()

通过调用 vkQueuePresentKHR() 函数演示图像。 这种感觉像是，提交缓冲区只进行一项操作：演示。

要演示图像，我们必须指定演示多少个图像，以及哪个交换链的哪些图像。 我们可一次性演示多个交换链的图像（即演示给多个窗口），但一次只可演示一个交换链的一个图像。 我们通过 VkPresentInfoKHR 结构提供这类信息，该结构包含以下字段：

sType – 标准结构类型；此处必须设置为 VK_STRUCTURE_TYPE_PRESENT_INFO_KHR。
pNext – 留作将来使用的参数。
waitSemaphoreCount – 演示图像之前希望队列等待的旗语数量。
pWaitSemaphores – 包含队列应等待的旗语句柄的阵列指示器；该阵列必须包含至少 waitSemaphoreCount 个要素。
swapchainCount – 我们希望演示其图像的交换链数量。
pSwapchains – 带有 swapchainCount 个要素，包含我们希望演示其图像的所有交换链的句柄的阵列；单个交换链仅在此阵列中出现一次。
imageIndices – 带有 swapchainCount 个要素，包含我们希望演示的图像的索引的阵列；阵列中的每个要素都对应 pSwapchains 阵列中的一个交换链；图像索引是阵列中每个交换链图像的索引（请参阅下一节）。
pResults – 包含至少 swapchainCount 个要素的阵列的指示器；该参数为可选项，可设置为 null，但如果我们提供此阵列，演示操作的结果将按照交换链分别保存在各要素中；整个函数返回的单个值与所有交换链的最差结果值相同。
该结构已准备好，现在我们可以用它来演示图像。 在本示例中，我只演示一个交换链的一个图像。

通过调用 vkQueue…() 函数执行（或提交）的每次操作都附在待处理队列的末尾。 按照提交的顺序依次处理各项操作。 我们在提交其他命令缓冲区后开始演示图像。 因此演示队列在处理完所有命令缓冲区后才开始演示图像。 这样可确保图像将在我们使用（渲染）后演示，而且内容正确的图像将显

示在屏幕上。 但在本示例中，我们向同一个队列 PresentQueue 提交绘制（清空）操作和演示操作。 我们只执行允许在演示队列上执行的简单操作。

如果想在队列上执行与演示操作不同的绘制操作，我们需要同步这些队列。 也可通过旗语完成，因此我们创建了两个旗语（本示例可能不需要使用第二个旗语，因为我们使用同一个队列渲染和演示图像，我是想展示如何正确地完成这一步骤）。

第一个旗语用于演示引擎告诉队列，它可以安全使用（引用/渲染）图像。 第二个旗语供我们使用。 图像操作（渲染）完成后向该旗语发出信号。 提交信息结构有一个名为 pSignalSemaphores 的字段。 它是旗语句柄阵列，在处理完所有提交的命令缓冲区后收到信号。 因此我们需要让第二个队列等待这第二个旗语。 我们将第二个旗语的句柄保存在 VkPresentInfoKHR 结构的 pWaitSemaphores 字段中。 这样由于有第二个旗语，我们提交演示操作的队列将等待特定图像渲染完成。

好了，就是这样。 我们使用 Vulkan 显示了第一个图像！

查看在交换链中创建的图像

之前我提到过交换链的图像索引。 在本代码实例中，我将对此具体介绍。

```
uint32_t image_count = 0;
if( (vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, nullptr ) != VK_SUCCESS) ||
      (image_count == 0) ) {
  std::cout << "Could not get the number of swap chain images!" << std::endl;
  return false;
}

std::vector<VkImage> swap_chain_images( image_count );
if( vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, &swap_chain_images[0] ) != VK_SUCCESS ) {
  std::cout << "Could not get swap chain images!" << std::endl;
```

```
    return false;
}
```

2 6. -

本代码示例是虚构出来，用于查看交换链中创建了多少以及哪些图像的函数的一个片段。 通过传统“两次调用”完成，这次使用的是 vkGetSwapchainImagesKHR() 函数。 首先我们调用该函数，其最后一个参数设为 null。 这样交换链中已创建图像数量将保存在“image_count”变量中，并且我们知道需要为所有图像的句柄准备多少存储。 第二次调用该函数时，我们在通过最后一个参数提供地址的阵列中取得句柄。

现在我们知道交换链正在使用的所有图像。 对 vkAcquireNextImageKHR() 函数和 VkPresentInfoKHR 结构来说，我引用的索引都是该阵列（通过 vkGetSwapchainImagesKHR() 函数返回的阵列）中的索引。 它称为交换链可演示图像的阵列。 在有交换链的情况下，如果函数希望提供或返回索引，该索引将是这一阵列中的图像索引。

重新创建交换链

之前我提到过多次，我们必须重新创建交换链，而且我还说过，之前的交换链必须毁坏。 vkAcquireNextImageKHR() 和 vkQueuePresentKHR() 函数返回的结果有时会导致调用 OnWindowSizeChanged() 函数。 该函数可重新创建交换链。

有时交换链会过期。 这意味着平面、平台或应用窗口的属性已发生了变化，当前交换链无法再使用。 最明显的（但可惜不是太好）示例是窗口大小的更改。 我们不能创建交换链图像，也不能改变大小。 唯一的可能就是毁坏并重新创建交换链。 还有一些我们仍能使用交换链的情况，但可能不再适用于为其创建的平面。

vkAcquireNextImageKHR() 和 vkQueuePresentKHR() 函数的返回代码会通知这些情况。

返回 VK_SUBOPTIMAL_KHR 值时，我们仍然可以将当前交换链用于演示。 它仍然可以使用，但不处于最佳状态（即色彩精度有所降低）。 如果有可能，建议重新创建交换链。 其中一个很好的示例就是执行性能严苛型渲染的时候，而且获取图像后，我们得知图像并不处于最佳状态。 我们不希望浪费这一处理过程，用户需要花很长时间等待另一个帧。 我们仅演示该图像并抓住机会重新创建交换链。

返回 VK_ERROR_OUT_OF_DATE_KHR 时，我们不能使用当前的交换链，必须立即重新创建。 我们不能使用当前交换链演示图像；该操作将失败。 我们必须尽快重新创建交换链。

我说过，关于平面属性更改，之后应重新创建交换链，更改窗口大小是最明显的（但不是最好的）的示例。 在这种情况下，我们应该重新创建交换链，但不会有之前提到的返回代码告知我们这一情况。 我们应该自己使用特定于操作系统的代码，监控窗口大小的变化。 而且这就是为什么在我们的资源中，该函数的名称为 OnWindowSizeChanged。 只要窗口大小发生变化，就要调用该函数。 但由于该函数仅重新创建交换链（和命令缓冲区），此处可调用相同的函数。

重新创建的方法与创建时相同。 有一个结构成员，我们在其中提供应被新交换链替换的交换链。 但创建新交换链后，我们必须毁坏之前的交换链。

快速了解命令缓冲区

现在您已了解了许多关于交换链的信息，但还有一点需要了解。 为解释这一点，我简要展示一下如何准备绘制命令。 关于交换链，最重要的一点是连接绘制和准备命令缓冲区。 我仅介绍如何清空图像，但这足以查看我们的交换链是否正常运行。

在教程 1 中，我介绍了队列和队列家族。 如果想在设备上执行命令，需通过命令缓冲区将它们提交至队列。 换句话说，命令封装在命令缓冲区内。 缓冲区提交至队列后，设备开始处理记录其中的命令。 还记得 OpenGL 的绘制列表吗？ 我们可准备命令列表，以便以绘制命令列表的形式绘制几何图形。 Vulkan 中的情况类似，但更加灵活和高级。

创建命令缓冲区内存池

命令缓冲区需要一部分存储保存命令。 如果要为命令提供空间，我们可创建一个池，缓冲区可向该池分配内存。 我们不必指定空间量 — 缓冲区建立（记录）时动态分配。

请记住，命令缓冲区只能提交至相应的队列家族，只有兼容特定家族的操作类型才能提交至特定队列。 此外，命令缓冲区本身不连接任何队列或队列家

族，但缓冲区分配内存的内存池连接。 因此每个从特定池获取内存的命令缓冲区只能提交至相应队列家族的队列 — 通过其（内部）创建内存池的家族。 如果通过特定家族创建多个队列，我们可将命令缓冲区提交至任意队列；此时家族索引最为重要。

```
VkCommandPoolCreateInfo cmd_pool_create_info = {
   VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,        // VkStructureType                  sType
   nullptr,                                           // const void*                      pNext
   0,                                                 // VkCommandPoolCreateFlags         flags
   Vulkan.PresentQueueFamilyIndex                     // uint32_t                         queueFamilyIndex
};

if( vkCreateCommandPool( Vulkan.Device, &cmd_pool_create_info, nullptr, &Vulkan.PresentQueueCmdPool ) != VK_SUCCESS ) {
   std::cout << "Could not create a command pool!" << std::endl;
   return false;
}
```

27.Tutorial02.cpp，函数 CreateCommandBuffers()

为创建命令缓冲区池，我们调用 vkCreateCommandPool() 函数。 它要求我们提供结构类型 VkCommandPoolCreateInfo 的（地址）变量。 它包含以下成员：

sType – 常用结构类型，此处必须相当于 VK_STRUCTURE_TYPE_CMD_POOL_CREATE_INFO。
pNext – 留作将来使用的指示器。
flags – 留作将来使用的值。
queueFamilyIndex – （为其创建池的队列家族的索引。
对测试应用来说，我们仅使用演示家族的一个队列，因此应该使用它的索引。 现在我们调用 vkCreateCommandPool() 函数并查看是否成功。 如果成功，命令池的句柄将保存在我们之前提供了地址的变量中。

分配命令缓冲区

接下来，我们需要分配命令缓冲区。 命令缓冲区不通过常见的方式创建，而是从池中分配。 从池对象获取内存的其他对象也分配（池自己创建）。 因此 vkCreate…() 和 vkAllocate…() 函数的名字相互分开。

如前所述，我分配一个以上的命令缓冲区 — 分别对应绘制命令引用的每个交换链图像。 因此每次获取交换链图像时，就可以提交/使用相应的命令缓冲区。

```
uint32_t image_count = 0;
if( (vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, nullptr ) != VK_SUCCESS) ||
      (image_count == 0) ) {
  std::cout << "Could not get the number of swap chain images!" << std::endl;
  return false;
}

Vulkan.PresentQueueCmdBuffers.resize( image_count );

VkCommandBufferAllocateInfo cmd_buffer_allocate_info = {
  VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO, // VkStructureType                sType
  nullptr,                                        // const void*                    pNext
  Vulkan.PresentQueueCmdPool,                     // VkCommandPool                  commandPool
  VK_COMMAND_BUFFER_LEVEL_PRIMARY,                // VkCommandBufferLevel           level
  image_count                                     // uint32_t                       bufferCount
};
if( vkAllocateCommandBuffers( Vulkan.Device, &cmd_buffer_allocate_info, &Vulkan.PresentQueueCmdBuffers[0] ) != VK_SUCCESS ) {
  std::cout << "Could not allocate command buffers!" << std::endl;
```

```
    return false;
}

if( !RecordCommandBuffers() ) {
    std::cout << "Could not record command buffers!" << std::endl;
    return false;
}
return true;
```

28.Tutorial02.cpp，函数 CreateCommandBuffers()

首先我们需要知道创建了多少个交换链图像（一个交换链创建的图像可超过我们指定的数量）。 这一点之前已有介绍。 我们调用 vkGetSwapchainImagesKHR() 函数，其中最后一个参数设为 null。 现在我们不需要图像句柄，只需要它们的总数。 然后我们为相应数量的命令缓冲区准备一个阵列（矢量），然后我们可以创建相应数量的命令缓冲区。 为实施该步骤，我们调用 vkAllocateCommandBuffers() 函数。 它要求我们准备类型 VkCommandBufferAllocateInfo 的结构化变量，其中包含以下字段：

sType – 结构类型，此时应为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO。
pNext – 留作将来使用的正态参数。
commandPool – 供缓冲区在命令记录期间分配内存的命令池。
level – 命令缓冲区的类型（级别）。 包含两个级别：主要和次要。 次要命令缓冲区仅通过主要命令缓冲区引用（使用）。 因为我们没有其他缓冲区，因此这里我们需要创建主要缓冲区。
bufferCount – 我们希望一次性创建的命令缓冲区数量。
调用 vkAllocateCommandBuffers() 函数后，需要查看缓冲器创建是否成功。 如果是，我们分配命令缓冲区，并准备记录一些（简单的）命令。

记录命令缓冲区

命令记录是我们在 Vulkan 中执行的最重要的操作。 记录本身要求我们提供许多信息。 信息越多，绘制命令越复杂。

（在本教程中）记录命令缓冲区要求以下变量：

```
uint32_t image_count = static_cast<uint32_t>(Vulkan.PresentQueueCmdBuffers.size());

std::vector<VkImage> swap_chain_images( image_count );
if( vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, &swap_chain_images[0] ) != VK_SUCCESS ) {
  std::cout << "Could not get swap chain images!" << std::endl;
  return false;
}

VkCommandBufferBeginInfo cmd_buffer_begin_info = {
  VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,    // VkStructureType                        sType
  nullptr,                                        // const void                             *pNext
  VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, // VkCommandBufferUsageFlags                flags
  nullptr                                         // const VkCommandBufferInheritanceInfo   *pInheritanceInfo
};

VkClearColorValue clear_color = {
  { 1.0f, 0.8f, 0.4f, 0.0f }
};

VkImageSubresourceRange image_subresource_range = {
  VK_IMAGE_ASPECT_COLOR_BIT,                      // VkImageAspectFlags                     aspectMask
  0,                                              // uint32_t                               baseMipLevel
  1,                                              // uint32_t                               levelCount
```

```
    0,                                                  // uint32_t                                 baseArrayLayer
    1                                                   // uint32_t                                 layerCount
};
```

29.Tutorial02.cpp，函数 RecordCommandBuffers()

首先我们获取所有交换链图像的句柄，用于绘制命令（我们仅将其清空至一种颜色，不过我们将要使用它们）。 我们知道了图像的数量，因此不必再次请求。 调用 vkGetSwapchainImagesKHR() 函数后，图像句柄保存在矢量中。

接下来我们需要准备结构化类型 VkCommandBufferBeginInfo 的变量。 它包含较多典型渲染场景（比如渲染通道）所需的信息。 这里不进行这些操作，因此我们将几乎全部参数都设为 0 或 null。 但为清楚起见，该结构包含以下字段：

sType – 标准类型，此次必须设为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO。
pNext – 留作将来使用的指示器，保留为 null。
flags – 定义命令缓冲区首选用法的参数。
pInheritanceInfo – 指示另一用于较多典型渲染场景的结构的参数。
命令缓冲区收集命令。 为将命令保存在命令缓冲区中，我们将它们记录下来。 以上结构提供一些必要的信息，供驱动程序准备和优化记录流程。

在 Vulkan 中，命令缓冲区分为主要缓冲区和次要缓冲区两种。 主要命令缓冲区是与绘制列表类似的常用命令缓冲区。 它们是独立的“个体”，（仅）提交至队列。 次要命令缓冲区还保存也可保存命令（我们也记录它们），但主要通过主要命令缓冲区引用（我们从主要命令缓冲区中调用次要命令缓冲区，就像从另一绘制列表调用 OpenGL 的绘制列表）。 不能将次要命令缓冲区直接提交至队列。

在下一节中，我们将详细介绍这些信息。

在这一简单示例中，我们希望用单个值清空图像。 因此接下来设置用于清空的颜色。 您可以选择任意值。 我使用淡橘色。

上述代码中的最后一个变量指定了我们执行操作的图像部分。 图像仅包含一个 mipmap 层和一个阵列层（没有立体缓冲区等）。 我们相应地设置

VkImageSubresourceRange 结构中的值。 该结构包含以下字段：

aspectMask – 将图像用作颜色渲染对象（它们有“颜色”格式）时，根据图像格式，指定此处的“color aspect”。
baseMipLevel – 将访问（修改）的第一个 mipmap 层。
levelCount – 执行操作的 mipmap 层（包括基础层）数量。
baseArrayLayer – 将访问（修改）的第一个阵列层。
arraySize – 执行操作的层级（包括基础层）数量。
我们准备记录一些缓冲区。

图像布局和布局过渡

（类型 VkImageSubresourceRange 的）上述代码示例所需的最后一个变量指定执行操作的图像部分。 在本课程中我们仅清空图像。 但我们还需执行资源过渡。 还记得创建交换链之前为交换链图像选择用法的代码吗？ 图像可用于不同的目的。 它们可用作渲染对象、在着色器中取样的纹理，或用于拷贝/blit 操作（数据传输）的数据源。 在为不同类型的操作创建图像时，我们必须指定不同的用法标记。 我们可以指定更多用法标记（如果支持；“color attachment”用法通常可用于交换链）。 但我们要做的不仅仅是指定图像用法。根据操作类型，图像可能以不同的方式分配，或在内存中呈现不同的布局。 每种图像操作类型可能与不同的“图像布局”有关。 我们可以支持所有操作支持的通用布局，但可能无法提供最佳性能。 对特定用法来说，我们应使用专用布局。

如果创建图像时考虑了不同的用法，并希望执行不同的操作，那么在执行每种操作之前，必须改变图像的当前布局。 为此，我们必须将当前局部过渡至兼容待执行操作的另一种布局。

我们创建的图像（通常）以未定义布局的形式创建，因此如果希望使用该图像，我们必须将其过渡至另一种布局。 但交换链创建的图像有 VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR 布局。 顾名思义，该布局针对演示引擎使用（演示）（即在屏幕上显示）的图像而设计。 因此，如果我们向在交换链图像上执行部分操作，需要将它们的布局更改成兼容所需操作的布局。 处理完图像（渲染图像）后，我们需要将布局过渡回 VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR。 否则演示引擎将无法使用这些引擎，而且会出现未定义行为。

进行布局过渡时，可使用图像内存壁垒。 我们用它们指定（当前）即将淘汰的旧布局和即将过渡到的新布局。 旧布局必须为当前或未定义的布局。 如果旧布局指定为未定义布局，那么在过渡过程中必须丢弃图像内容。 这样有助于驱动程序执行优化。 如果想保存图像内容，那么必须将布局指定为当前布局。

上述代码示例中类型 VkImageSubresourceRange 的最后一个变量也可用于图像过渡。 它可定义哪“部分”图像将改变布局，而且在准备图像内存壁垒时需要该变量。

记录命令缓冲区

最后一步是为每个交换链图像记录一个命令缓冲区。 我们希望将图像清空至任意一种颜色。 但首先需要改变图像布局，并在完成后恢复布局。 此处完成这一步的代码如下：

```
for( uint32_t i = 0; i < image_count; ++i ) {
  VkImageMemoryBarrier barrier_from_present_to_clear = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,         // VkStructureType                sType
    nullptr,                                        // const void                    *pNext
    VK_ACCESS_MEMORY_READ_BIT,                      // VkAccessFlags                  srcAccessMask
    VK_ACCESS_TRANSFER_WRITE_BIT,                   // VkAccessFlags                  dstAccessMask
    VK_IMAGE_LAYOUT_UNDEFINED,                      // VkImageLayout                  oldLayout
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,           // VkImageLayout                  newLayout
    Vulkan.PresentQueueFamilyIndex,                 // uint32_t                       srcQueueFamilyIndex
    Vulkan.PresentQueueFamilyIndex,                 // uint32_t                       dstQueueFamilyIndex
    swap_chain_images[i],                           // VkImage                        image
    image_subresource_range                         // VkImageSubresourceRange        subresourceRange
  };
```

```
  VkImageMemoryBarrier barrier_from_clear_to_present = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,         // VkStructureType                sType
    nullptr,                                        // const void                     *pNext
    VK_ACCESS_TRANSFER_WRITE_BIT,                   // VkAccessFlags                  srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,                      // VkAccessFlags                  dstAccessMask
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,           // VkImageLayout                  oldLayout
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                // VkImageLayout                  newLayout
    Vulkan.PresentQueueFamilyIndex,                 // uint32_t                       srcQueueFamilyIndex
    Vulkan.PresentQueueFamilyIndex,                 // uint32_t                       dstQueueFamilyIndex
    swap_chain_images[i],                           // VkImage                        image
    image_subresource_range                         // VkImageSubresourceRange        subresourceRange
  };

  vkBeginCommandBuffer( Vulkan.PresentQueueCmdBuffers[i], &cmd_buffer_begin_info );
  vkCmdPipelineBarrier( Vulkan.PresentQueueCmdBuffers[i], VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 0, nullptr, 1, &barrier_from_present_to_clear );

  vkCmdClearColorImage( Vulkan.PresentQueueCmdBuffers[i], swap_chain_images[i], VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, &clear_color, 1, &image_subresource_range );

  vkCmdPipelineBarrier( Vulkan.PresentQueueCmdBuffers[i], VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0, 0, nullptr, 0, nullptr, 1, &barrier_from_clear_to_present );
  if( vkEndCommandBuffer( Vulkan.PresentQueueCmdBuffers[i] ) != VK_SUCCESS ) {
    std::cout << "Could not record command buffers!" << std::endl;
    return false;
  }
```

}

return true;
30.Tutorial02.cpp，函数 RecordCommandBuffers()

该代码放在循环内。 我们为每个交换链图像记录一个命令缓冲区。 因此我们需要大量图像。 这里也需要图像句柄。 我们需要在图像清空期间为图像内存壁垒指定这些句柄。 不过大家回忆一下，我之前说过，必须得到允许，获取交换链图像后，才能使用该图像。 没错，但我们这里不使用这些图像。 我们仅准备命令。 将操作（命令缓冲区）提交至队列时执行用法。 这里我们仅告知 Vulkan，未来拿这张图片这样做，然后那样做.....等等。 这样，在我们开始主渲染循环之前，做尽可能多的准备工作，从而在真实渲染迁建避免切换、ifs、跳跃和其他分支。 在现实生活中这一场景并不简单，但我希望能够通过示例解释清除。

在上述代码中，我们首先准备两个图像内存壁垒。 内存壁垒用于改变图像中的三个不同元素。 从教程的角度来看，现在感兴趣的只有布局，但我们需要适当设置所有字段。 为设置内存壁垒，我们需要准备类型变量 VkImageMemoryBarrier，其中包含以下字段：

sType – 标准类型，此处必须设置为 VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER。
pNext – 保留为 null 状态，目前暂不使用的指示器。
srcAccessMask – 壁垒前在图像上完成的内存操作类型。
dstAccessMask – 壁垒后进行的内存操作类型。
oldLayout – 即将转换出的布局；并始终相当于当前布局（在本示例中面向第一个壁垒，应为 VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR）。或者我们还可以使用未定义布局，支持驱动程序执行部分优化，但图像内容可能会被丢弃。 既然我们不需要内容，那么我们可以使用此处的未定义布局。
newLayout – 兼容我们将在壁垒后执行的操作的布局；我们希望清空图像；为此我们需要指定 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 布局。 我们应一直使用特定的专用布局。
srcQueueFamilyIndex – 之前引用图像的队列家族索引。
dstQueueFamilyIndex – 将在壁垒后引用图像的队列的家族索引（指我之前所述的交换链共享模式）。
image – 图像句柄。
subresourceRange – 描述我们希望执行过渡的图像部分的结构；也就是之前代码示例的最后一个变量。

关于访问掩码和家族索引，需注意几点。 在本示例中，在第一个壁垒前和第二个壁垒后，仅演示引擎可访问图像。 演示引擎仅读取图像（不更改图像），因为我们将第一个壁垒中的 srcAccessMask 和第二个壁垒中的 dstAccessMask 设置为 VK_ACCESS_MEMORY_READ_BIT。 这表示以图像相关的图像为只读模式（在第一个壁垒前和第二个壁垒后，不更改能图像内容）。 在命令缓冲区中，我们仅清空图像。 该操作属于所谓的“转移”操作。 因此我们将第一个壁垒的 dstAccessMask 字段和第二个壁垒中的 srcAccessMask 字段设为 VK_ACCESS_TRANSFER_WRITE_BIT。

我不详细介绍队列家族索引，但如果用于图形操作的队列和演示相同，srcQueueFamilyIndex 和 dstQueueFamilyIndex 将相同，且硬件不会对图像访问队列的行为作出任何更改。 但请记住，我们已经指定，一次只有一个队列可以访问/使用图像。 因此如果队列不同，我们会将“所有权”更改通知给此处的硬件，即现在有不同的队列访问图像。 而且您此刻需要这些信息，以对壁垒进行相应的设置。

我们需要创建两个壁垒：一个将布局从“present source”（或未定义）改为“transfer dst”。 该壁垒用在命令缓冲区的开头，如果之前的演示引擎使用过图像，我们现在希望使用并更改它。 第二个壁垒用于当我们使用完图像并将其交还给交换链时，将布局恢复成“present source”。 该壁垒在命令缓冲区末尾设置。

现在我们已准备好通过调用 vkBeginCommandBuffer() 函数开始记录命令。 我们提供命令缓冲区句柄和类型变量 VkCommandBufferBeginInfo 的地址，并开始进行。 接下来设置壁垒，以更改图像布局。 我们调用函数，它包含多个参数，但在本示例中，相关的参数只有第一个（命令缓冲区句柄）和最后两个：阵列要素数量和包含类型 VkImageMemoryBarrier 的变量地址的阵列的第一个要素的指示器。 该阵列的要素描述图像、其组成部分，以及应进行的过渡类型。 我们可以在壁垒后安全执行任何有关交换链图像，且兼容已过渡图像的布局的操作。 通用布局兼容所有操作，但性能（可能）有所下降。

在本示例中，我们仅清空图像，因此调用 vkCmdClearColorImage() 函数。 它提取命令缓冲区句柄、图像句柄、图像的当前布局、带有清晰色彩值的变量的指示器、子资源数量（最后一个参数的数量的要素数量），以及类型 VkImageSubresourceRange 的变量的指示器。 最后阵列中的要素指定我们希望清空的图像部分（如果不想，我们可以不清空图像的所有 mipmap 或阵列层）。

在记录会话的结尾部分，我们设置另一个壁垒，将图像布局过渡回“present source”布局。 它是唯一一个兼容演示引擎执行的演示操作的布局。

现在我们调用 vkEndCommandBuffer()函数，通知我们已结束记录命令缓冲区。 如果在记录期间出错，该函数将通过返回值向我们通知出现错误。 如果出现错误，我们将无法使用命令缓冲器，并需要重新记录。 如果一切正常，我们将可以使用命令缓冲区，只需将缓冲区提交至阵列，就可告诉设备执行

保存在其中的操作。

教程 2 执行

在本示例中，如果一切正常，我们将看到一个显示淡橙色的窗口。 窗口内容将如下所示：

清空

现在您已知道如何创建交换链、在窗口中显示图像，并在设备上执行简单的操作。 我们已创建命令缓冲区、记录它们，并在屏幕上显示它们。 但在关

闭应用之前，我们需要清空所使用过的资源。 在本教程中我们将清空过程分成两个函数： 第一个函数仅清空（毁坏）间重新创建交换链时（即应用窗口改变大小后）应重新创建的资源。

```
if( Vulkan.Device != VK_NULL_HANDLE ) {
   vkDeviceWaitIdle( Vulkan.Device );

   if( (Vulkan.PresentQueueCmdBuffers.size() > 0) && (Vulkan.PresentQueueCmdBuffers[0] != VK_NULL_HANDLE) ) {
      vkFreeCommandBuffers( Vulkan.Device, Vulkan.PresentQueueCmdPool, static_cast<uint32_t>(Vulkan.PresentQueueCmdBuffers.size()),
&Vulkan.PresentQueueCmdBuffers[0] );
      Vulkan.PresentQueueCmdBuffers.clear();
   }

   if( Vulkan.PresentQueueCmdPool != VK_NULL_HANDLE ) {
      vkDestroyCommandPool( Vulkan.Device, Vulkan.PresentQueueCmdPool, nullptr );
      Vulkan.PresentQueueCmdPool = VK_NULL_HANDLE;
   }
}
```

31.Tutorial02.cpp，Clear()

首先我们必须确保设备阵列上没有执行操作（不能毁坏当前已处理命令所使用的资源）。 我们通过调用 vkDeviceWaitIdle() 函数进行检查。 直到所有操作完成后才中止。

接下来我们释放所有分配的命令缓冲区。 事实上，这里并不一定需要执行此操作。 毁坏命令池会暗中释放所有从特定池分配的命令缓冲区。 但我希望展示如何明确释放命令缓冲区。 接下来毁坏命令池。

以下代码负责毁坏本教程中创建的所有资源：

```
Clear();

if( Vulkan.Device != VK_NULL_HANDLE ) {
   vkDeviceWaitIdle( Vulkan.Device );

   if( Vulkan.ImageAvailableSemaphore != VK_NULL_HANDLE ) {
      vkDestroySemaphore( Vulkan.Device, Vulkan.ImageAvailableSemaphore, nullptr );
   }
   if( Vulkan.RenderingFinishedSemaphore != VK_NULL_HANDLE ) {
      vkDestroySemaphore( Vulkan.Device, Vulkan.RenderingFinishedSemaphore, nullptr );
   }
   if( Vulkan.SwapChain != VK_NULL_HANDLE ) {
      vkDestroySwapchainKHR( Vulkan.Device, Vulkan.SwapChain, nullptr );
   }
   vkDestroyDevice( Vulkan.Device, nullptr );
}

if( Vulkan.PresentationSurface != VK_NULL_HANDLE ) {
   vkDestroySurfaceKHR( Vulkan.Instance, Vulkan.PresentationSurface, nullptr );
}

if( Vulkan.Instance != VK_NULL_HANDLE ) {
   vkDestroyInstance( Vulkan.Instance, nullptr );
}
```

```
if( VulkanLibrary ) {
#if defined(VK_USE_PLATFORM_WIN32_KHR)
   FreeLibrary( VulkanLibrary );
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
   dlclose( VulkanLibrary );
#endif
}
```

32.Tutorial02.cpp，destructor

首先我们毁坏旗语（请记住，它们在使用的过程中不能毁坏，即队列等待特定旗语时）。 然后毁坏交换链。 与交换链一同创建的图像会自动毁坏，因此无需（也不允许）我们手动进行。 接下来毁坏设备。 我们还需毁坏代表应用窗口的平面。 最后，毁坏 Vulkan，并卸载图形驱动程序的动态库。 执行每步操作之前，我们还需检查是否相应地创建了特定资源。 不能毁坏不是相应创建的资源。

结论

在本教程中，您学习了如何在屏幕上显示用 Vulkan API 创建的图像。 步骤如下： 首先启用相应的实例层扩展。 接下来创建应用窗口的 Vulkan 表现形式（称为平面）。 然后选择带有家族阵列（支持演示并创建设备）的设备（不要忘记启用设备层扩展!）

之后创建交换链。 为此我们首先获取描述平面的参数集，然后选择适用于交换链创建的值。 这些值必须满足平面支持的限制条件。

为在屏幕上进行绘制，我们学习了如何创建和记录命令缓冲区，还包括图像使用内存壁垒（管道壁垒）所进行的布局过渡。 我们清空图像，以看到所选颜色显示在屏幕上。

我们还学习了如何在屏幕上演示特定图像，包括获取图像、提交命令缓冲区，以及演示流程。

教程 3： 第一个三角形 — 图形管道和绘制
关于源代码示例
创建渲染通道
渲染通道附件描述
子通道描述
渲染通道创建
创建帧缓冲器
创建图像视图
指定帧缓冲器参数
创建图形管道
创建着色器模块
准备着色器阶段描述
准备顶点输入描述
准备输入汇编描述
准备视口描述
准备光栅化状态描述
设置多点采样状态描述
设置混合状态描述
创建管道布局
创建图形管道
准备绘制命令
创建命令池
分配命令缓冲区
记录命令缓冲区
绘制
教程 3 执行

清空
结论
>> 前往第 4 部分

教程 3： 第一个三角形 — 图形管道和绘制

在本教程中我们将最后在屏幕上绘制一些图形。 简单的三角形就是 Vulkan 生成一个比较好的的“图像”。

一般来说，图形管道和绘制操作要求 Vulkan 做许多准备工作（以在许多结构中填充复杂字段的形式）。 我们在很多方面都有可能犯错误，而且在 Vulkan 中，即使简单的错误也会造成应用无法按预期运行、显示空白屏幕，而且我们无法得知到达哪里出现了错误。 在这种情况下，验证层可为我们提供帮助。 但我不希望深入探讨 Vulkan API 的细节。 因此，我尽可能准备小型、简单的代码。

这样我们创建的应用就可以按照预期正常运行并显示简单的三角形，但它还使用不建议使用、不灵活，而且可能不高效（尽管正确）的机制。 我不想讨论不建议使用的情况，但它可显著简化教程，并支持我们仅专注于所需的最小 API 用法集。 只要遇到了“有争议的”功能，我都会指出来。 在下一教程中，我将介绍一些绘制三角形的推荐方法。

要绘制第一个简单的三角形，我们需创建渲染通道、帧缓冲器和图形管道。 当然也需要命令缓冲器，但我们已对其有所了解。 我们将创建简单的 GLSL 着色器，并将其编译成 Khronos 的 SPIR*-V 语言 — Vulkan（官方）理解着色器的（目前）唯一形式。

如果您的电脑屏幕没有显示任何内容，请尝试尽可能地简化代码，或者回到教程 2。 检查命令缓冲区是否按预期仅清空了图形行为，而且图形清空的颜色是否显示在屏幕上。 如果是，请通过本教程修改代码并添加一些部分。 如果不是 VK_SUCCESS，请检查每个返回值。 如果这些方法没有用，请等待本教程的验证层。

关于源代码示例

为方便本教程及其随后的教程，我更换了示例项目。 之前教程中介绍的 Vulkan 准备阶段放在单独文件（标头和源）的“VulkanCommon”类。 面向特

定教程的类负责演示特定教程中介绍的主题、承袭“VulkanCommon”类并访问（所需）的 Vulkan，比如设备或交换链。 这样我们可以重新使用 Vulkan 创建代码，并准备仅专注于已演示主题的较小的类。 之前章节的代码能够正常运行，因此比较容易找出潜在错误。

我还为部分实用程序函数添加了独立的文件集。 此处我们将通过二进制文件读取 SPIR-V 着色器，因此我添加了一个函数，可检查二进制文件内容的加载。 它位于 Tools.cpp 和 Tools.h 文件。

创建渲染通道

为在屏幕上进行绘制，我们需要一个图形管道。 但现在创建这一管道需要其他结构的指示器，其中还可能需要另外其他结构的指示器。 因此我们从渲染通道开始。

什么是渲染通道？ 常见图片可为我们提供一个用于许多常用渲染技巧（比如延迟着色）的“逻辑”渲染通道。 该技巧包含许多子通道。 第一个子通道使用填充 G-Buffer 的着色器绘制几何图形：将漫射颜色保存在一种纹理中，将标准矢量保存在另一纹理中、将亮度保存在另一纹理中，而将深度（位置）保存在另一纹理中。 接下来是各个光源，执行的绘制包括读取数据（标准矢量、亮度、深度/位置）、计算照明，并将其保存在另一纹理中。 最后一个通道整合照明数据和漫射颜色。 这只是有关延期着色的（粗略）解释，但它介绍了渲染通道 — 执行部分绘制操作所需的数据集：将数据保存在纹理中，并从其他纹理读取数据。

在 Vulkan 中，渲染通道代表（或描述）执行绘制操作所需的帧缓冲区附件（图像）集，以及排列绘制操作的子通道集合。 它是一种收集所有颜色、深度与模板附件，以及操作的构造，对它们进行修改后，驱动程序无需自己推断这种信息，从而为部分 GPU 提供了重要的优化机会。 子通道包含使用（或多或少）相同附件的绘制操作。 每种绘制操作都从部分输入附件读取数据，并将数据渲染至其他（颜色、深度、模板）附件。 渲染通道还描述这些附件之间的相关性：我们在一个子通道中渲染纹理，而在另一子通道中该纹理将用作数据源（即通过其进行采样）。 所有这些数据都可帮助图形硬件优化绘制操作。

为在 Vulkan 中创建渲染通道，我们调用 vkCreateRenderPass() 函数，它要求具有结构指示器，该结构描述所有涉及渲染的附件和所有形成渲染通道的子通道。 像往常一样，使用的附件和子通道越多，所需的包含相应字段结构的阵列要素越多。 在这一简单示例中，我们仅通过单个通道绘制到单个纹理（颜色附件）。

渲染通道附件描述

```
VkAttachmentDescription attachment_descriptions[] = {
  {
    0,                                          // VkAttachmentDescriptionFlags     flags
    GetSwapChain().Format,                      // VkFormat                         format
    VK_SAMPLE_COUNT_1_BIT,                      // VkSampleCountFlagBits            samples
    VK_ATTACHMENT_LOAD_OP_CLEAR,                // VkAttachmentLoadOp                   loadOp
    VK_ATTACHMENT_STORE_OP_STORE,               // VkAttachmentStoreOp                  storeOp
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,            // VkAttachmentLoadOp                   stencilLoadOp
    VK_ATTACHMENT_STORE_OP_DONT_CARE,           // VkAttachmentStoreOp                  stencilStoreOp
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout                        initialLayout;
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR             // VkImageLayout                        finalLayout
  }
};
```

1.Tutorial03.cpp，函数 reateRenderPass()

为创建渲染通道，我们首先准备一个阵列，其中包含描述各个附件（无论附件类型）及其在渲染通道中的用法的要素。 该阵列要素的类型为 VkAttachmentDescription 并包含以下字段：

flags －描述附件的其他属性。 目前仅别名标记可用，它告知驱动程序附件与其他附件共享相同物理内存；这种情况不适用此处，因此我们将该参数设置为零。
format －用于附件的图像格式；此处我们直接渲染至交换链，因此需要采用这种格式。
samples － 图像样本数量；我们此处不使用任何多点采用，因此仅使用一个样本。
loadOp － 指定如何处理渲染通道开头的图像内容，是希望清空、保存，还是不管这些内容（我们将覆写这些内容）。 此处我们希望将图像清空至指定的

值。 该参数还表示深度/模板图像的深度部分。
storeOp – 告知驱动程序如何处理渲染通道后的图像内容（最后一次使用图像的子通道之后）。 此处我们希望在渲染通道后保存图像内容，因为我们想在屏幕上显示这些内容。 该参数还表示深度/模板图像的深度部分。
stencilLoadOp – 与 loadOp 相同，但面向深度/模板图像的模板部分；对颜色附件来说它已被忽略。
stencilStoreOp – 与 storeOp 相同，但面向深度/模板图像的模板部分；对颜色附件来说该参数已被忽略。
initialLayout – 渲染通道启动时特定附件将呈现的布局（应用为布局图像提供的内容）。
finalLayout – 渲染通道结束后驱动程序自动将特定图像过渡至的布局。
还需要一些其他信息亿完成加载和保存操作，以及初始和最终布局。

加载选项指渲染通道开头的附件内容。 该操作描述图形硬件如何处理附件：清空、在现有内容上操作（不触碰内容），或者不管它们，因为应用打算覆写这些内容。 这样硬件将有机会优化内存操作。 例如，如果我们希望覆写这些内容，硬件将不会打扰这些内容，而且如果速度加快，可能为附件分配所有新内存。

保存选项，顾名思义，用在渲染通道结尾部分，告知硬件我们是希望在渲染通道后使用附件内容，还是不关心并有可能舍弃这些内容。 在一些场景中（舍弃这些内容时）它将支持硬件在临时快速内存中创建图像，因为图像将仅在渲染通道期间“活跃”，而且实施操作可能会节省一些内存带宽，以避免在不需要的时候回写数据。

如果附件有深度格式（并可能有模板组件），加载和保存选项仅表示深度组件。 如果出现模板，将以模板加载和保存选项描述的方式处理模板值。 模板选项与颜色附件无关。

我们交换链教程中介绍过，布局指图像内部内存的安排形式。 图像数据整理后，相邻“图像像素”也是内存中的邻居，这样图像用作数据源时（即在纹理采样期间），可提高缓存命中率（加快内存读取速度）。 如果图像用作绘制操作的对象，并不一定要执行高速缓存，而且可能以完全不同的方法来整理用于该图像的内存。 图像可能呈现线性布局（支持 CPU 读取或填充图像的内存内容），也可能呈现最佳布局（面向性能优化，但依然依赖硬件/厂商）。因此一些硬件可能针对一些操作类型有特定的内存组织形式；其他硬件可能适用于任何操作类型。 部分内存布局可能更适合预期的图像“用法”。 或从另一角度来说，部分用法可能要求特定的内存布局。 同时也存在一种通用布局，兼容所有操作类型。 但从性能的角度来说，最好设置符合预期图像用法的布局，而且应用负责将布局过渡告知驱动程序。

可使用图像内存壁垒更改图像布局。 我们在交换链教程中这样做过，首先将布局演示源（演示引擎使用的图像）改成转移目标（希望使用特定颜色清空图像）。 但布局与图像内存壁垒不同，也可由渲染通道内的硬件自动更改。 如果我们指定不同的初始布局、子通道布局（稍后介绍）和最终布局，硬件将在适当时自动过渡。

初始布局将应用为特定附件“提供”（或“保留”）的布局告知硬件。 图像在渲染通道开头开始呈现这种布局（在本示例中，我们从演示引擎获取图像，因此图像呈现“演示源”布局）。 渲染通道的每个子通道都使用不同的布局，子通道之间的硬件自动进行过渡。 最终布局是特定附件将在渲染通道结尾时（渲染通道完成后）（自动）过渡至的布局。

必须为将用于渲染通道的每个附件准备这类信息。 图形硬件收到此类信息后，可能会在渲染通道期间优化操作和内存，以实现最佳性能。

子通道描述

```
VkAttachmentReference color_attachment_references[] = {
  {
    0,                                          // uint32_t                         attachment
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL    // VkImageLayout                    layout
  }
};

VkSubpassDescription subpass_descriptions[] = {
  {
    0,                                          // VkSubpassDescriptionFlags        flags
    VK_PIPELINE_BIND_POINT_GRAPHICS,            // VkPipelineBindPoint              pipelineBindPoint
    0,                                          // uint32_t                         inputAttachmentCount
    nullptr,                                    // const VkAttachmentReference      *pInputAttachments
```

```
        1,                                                  // uint32_t                         colorAttachmentCount
        color_attachment_references,                        // const VkAttachmentReference      *pColorAttachments
        nullptr,                                            // const VkAttachmentReference      *pResolveAttachments
        nullptr,                                            // const VkAttachmentReference      *pDepthStencilAttachment
        0,                                                  // uint32_t                         preserveAttachmentCount
        nullptr                                             // const uint32_t*                  pPreserveAttachments
    }
};
```

2.Tutorial03.cpp，函数 CreateRenderPass()

接下来我们指定渲染通道将包含的子通道描述。 该步骤可通过 VkSubpassDescription 结构完成，其中包含以下字段：

flags – 留作将来使用的参数。
pipelineBindPoint – 供子通道使用的管道类型（图形或计算）。 当然我们的示例使用图形管道。
inputAttachmentCount – pInputAttachments 阵列中的要素数量。
pInputAttachments – 包含描述哪些附件用作输入，并可从内部着色器读取的要素阵列。 此处我们不使用任何输入附件，因此将其设为 0。
colorAttachmentCount – pColorAttachments 和 pResolveAttachments 阵列中的要素数量。
pColorAttachments – 描述（指示）将用作颜色渲染对象（渲染图像）的附件阵列。
pResolveAttachments – 与颜色附件紧密相连的阵列。 该阵列的每个要素都分别对应颜色附件阵列中的一个要素；此类颜色附件将分解成特定分解附件（如果相同索引中的分解附件或整个指示器不是 null）。 该参数为可选项，而且可设为 null。
pDepthStencilAttachment – 将用于深度（和/或模板）数据的附件描述。 我们这里不使用深度信息，因此将其设为 null。
preserveAttachmentCount – pPreserveAttachments 阵列中的要素数量。
pPreserveAttachments – 描述将被保留的附件阵列。 如有多个子通道，并非所有通道都使用所有附件。 如果子通道不使用其中的附件，但在后续的通道中需要它们，那么我们必须在这里指定这些附件。
pInputAttachments、pColorAttachments、pResolveAttachments、pPreserveAttachments 和 pDepthStencilAttachment 参数的类型均为 VkAttachmentReference。该结构仅包含两个字段：

attachment – VkRenderPassCreateInfo 的 attachment_descriptions 阵列索引。
layout – 附件在特定子通道期间请求（所需）的布局。 在特定通道之前，硬件将帮助自动过渡至提供的布局。
该结果包含 VkRenderPassCreateInfo 的 attachment_descriptions 阵列的参考信息（索引）。 创建渲染通道时，我们必须提供有关用于渲染通道期间的所有附件的描述。 之前创建 attachment_descriptions 阵列时，我们已在“渲染通道附件描述”部分准备了该描述。 现在它仅包含一个要素，但在高级场景中将有多个附件。 因此这种所有渲染通道附件的“通用”集合将用作参考点。 在子通道描述中，当填充 pColorAttachments 或 pDepthStencilAttachment members 成员时，我们提供这种“通用”集合的索引，像这样：从渲染通道附件提取第一个附件，并将其用作颜色附件。 该阵列的第二个附件将用于深度数据。

整个渲染通道与其子通道是独立的，因为子通道可能以不同的方式使用多个附件，即我们在一个子通道中渲染颜色附件，而在下一个子通道中读取该附件。 这样我们准备用于整个渲染通道的附件列表，同时可以指定每个附件在通道中的使用方式。 由于各子通道可能以自己独有的方式使用特定附件，因此我们必须为各子通道指定每个图像的布局。

因此指定所有子通道（包含 VkSubpassDescription 类型要素的阵列）之前，必须为用于各子通道的附件创建引用。这就是创建 color_attachment_references 变量的目的所在。 编写纹理渲染教程时，该用法会变得更加明显。

渲染通道创建

现在我们有创建渲染通道需要的所有数据。

```
vkRenderPassCreateInfo render_pass_create_info = {
  VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO,       // VkStructureType                    sType
  nullptr,                                         // const void                        *pNext
  0,                                               // VkRenderPassCreateFlags            flags
  1,                                               // uint32_t                           attachmentCount
  attachment_descriptions,                         // const VkAttachmentDescription *pAttachments
```

```
    1,                                                  // uint32_t                             subpassCount
    subpass_descriptions,                               // const VkSubpassDescription       *pSubpasses
    0,                                                  // uint32_t                             dependencyCount
    nullptr                                             // const VkSubpassDependency        *pDependencies
};

if( vkCreateRenderPass( GetDevice(), &render_pass_create_info, nullptr, &Vulkan.RenderPass ) != VK_SUCCESS ) {
    printf( "Could not create render pass!\n" );
    return false;
}

return true;
```

3.Tutorial03.cpp，函数 CreateRenderPass()

我们首先填充 VkRenderPassCreateInfo 结构，其中包含以下字段：

sType – 结构类型（此处为 VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO）。
pNext – 目前不使用的参数。
flags – 留作将来使用的参数。
attachmentCount – 整个渲染通道（此处仅一个）期间使用的不同附件（pAttachments 阵列中的要素）数量。
pAttachments – 指定所有用于渲染通道的附件阵列。
subpassCount – 渲染通道包含的子通道数量（以及 pSubpasses 阵列（本示例中仅一个）中的要素数量）。
pSubpasses – 包含所有子通道描述的阵列。
dependencyCount – pDependencies 阵列中的要素数量（此处为 0）。
pDependencies – 描述子通道配对之间相关性的阵列。 我们的子通道不多，因此不存在相关性（此处设为 null）。
相关性描述图形管道的哪些部分以怎样的方式使用内存资源。 每个子通道使用资源的方式都各不相同。 资源布局不仅仅定义它们如何使用资源。 部分

子通道可能渲染图像或通过着色器图像保存数据。 其他子通道可能不使用图像，也可能在不同的图像管道阶段（即顶点或碎片）读取图像。

该信息可帮助驱动程序优化自动布局过渡，更常见的是优化子通道之间的壁垒。 仅在顶点着色器中写入图像时，等待碎片着色器执行（当前以已用图像的形式）的意义不大。 执行完所有顶点操作后，图像立即更改布局和内存访问类型，部分图形硬件甚至会开始执行后续操作（引用或读取特定图像），无需等待完成特定子通道的其他命令。 现在只需记住，相关性对性能非常重要。

现在我们已经准备了创建渲染通道需要的所有信息，可以安全地调用 vkCreateRenderPass() 函数。

创建帧缓冲器

我们创建了渲染通道。 它描述渲染通道期间使用的所有附件和子通道。 但这种描述非常抽象。 我们指定了所有附件（本示例中仅一个）的格式，并描述了子通道（同样只有一个）如何使用附件。 但我们没有指定使用哪些附件，换句话说，哪些图像将用作这些附件。 这一过程将通过帧缓冲器完成。

帧缓冲器描述供渲染通道操作的特定图像。 在 OpenGL*中，帧缓冲器是我们将渲染的纹理（附件）集。 在 Vulkan 中，该术语的意义更加广泛。 它描述渲染通道期间使用的所有纹理（附件），不仅包括即将渲染的图像（颜色和深度/模板附件），还包括用作数据源的图像（输入附件）。

渲染通道和帧缓冲器的分开为我们提高了灵活性。 特定渲染通道可用于不同的帧缓冲器，特定帧缓冲器也可用于不同的渲染通道，如果它们相互兼容，表示它们能在具有相同类型和用法的图像上以相同的方式操作。

创建帧缓冲器之前，我们必须为每个用作帧缓冲器和渲染通道附件的图像创建图像视图。 在 Vulkan 中，不仅在有帧缓冲器的情况下，一般情况下我们都不操作图像本身。 不能直接访问图像。 为此，我们使用图像视图。 图像视图代表图像，它们“包装”图像并提供其他（元）数据。

创建图像视图

在该简单应用中，我们想直接渲染交换链图像。 我们创建了包含多个图像的交换链，因此必须为每个图像创建图像视图。

```
const std::vector<VkImage> &swap_chain_images = GetSwapChain().Images;
Vulkan.FramebufferObjects.resize( swap_chain_images.size() );

for( size_t i = 0; i < swap_chain_images.size(); ++i ) {
  VkImageViewCreateInfo image_view_create_info = {
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO,     // VkStructureType              sType
    nullptr,                                      // const void                  *pNext
    0,                                            // VkImageViewCreateFlags       flags
    swap_chain_images[i],                         // VkImage                      image
    VK_IMAGE_VIEW_TYPE_2D,                        // VkImageViewType              viewType
    GetSwapChain().Format,                        // VkFormat                     format
    {                                             // VkComponentMapping           components
      VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle             r
      VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle             g
      VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle             b
      VK_COMPONENT_SWIZZLE_IDENTITY               // VkComponentSwizzle             a
    },
    {                                             // VkImageSubresourceRange      subresourceRange
      VK_IMAGE_ASPECT_COLOR_BIT,                  // VkImageAspectFlags             aspectMask
      0,                                          // uint32_t                       baseMipLevel
      1,                                          // uint32_t                       levelCount
      0,                                          // uint32_t                       baseArrayLayer
      1                                           // uint32_t                       layerCount
    }
  };
```

```
  if( vkCreateImageView( GetDevice(), &image_view_create_info, nullptr, &Vulkan.FramebufferObjects[i].ImageView ) != VK_SUCCESS ) {
    printf( "Could not create image view for framebuffer!\n" );
    return false;
  }
```

4.Tutorial03.cpp, function CreateFramebuffers()

为创建图像视图，必须首先创建类型变量 VkImageViewCreateInfo。 它包含以下字段：

sType – 结构类型，此处应设为 VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO。
pNext – 通常设为 null 的参数。
flags – 留作将来使用的参数。
image – 为其创建图像视图的图像的句柄。
viewType – 希望创建的视图类型。 视图类型必须兼容相应的图像。 （即我们可以为包含多个阵列层的图像创建 2D 视图，也可为包含 6 个层级的 2D 图像创建 CUBE 视图）。
format – 图像视图的格式；必须兼容图像的格式，不能为相同的格式（即可以是不同的格式，但没像素的位数必须相同）。
components – 将图像组件映射到通过纹理操作返回到着色器中的顶点。 这仅适用于读取操作（采样），但既然我们将图像用作颜色附件（渲染图像），那么必须设置身份映射（R 组件为 R，G -> G 等等）或仅使用“身份”值 (VK_COMPONENT_SWIZZLE_IDENTITY)。
subresourceRange – 描述视图可访问的 mipmap 层和阵列层集。 如果对图像进行 mipmap 处理，我们可以指定希望渲染的特定 mipmap 层（如果有渲染对象，必须精确指定某个阵列层的某个 mipmap 层）。
大家可以看这里，我们获取所有交换链图像的句柄，并在循环内引用它们。 这样我们填充创建图像视图所需的结构，这样我们前往 vkCreateImageView() 函数。 每个与交换链一起创建的图像都进行这样的处理。

指定帧缓冲器参数

现在我们可以创建帧缓冲器。 为此我们调用 vkCreateFramebuffer() 函数。

```
VkFramebufferCreateInfo framebuffer_create_info = {
      VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO,    // VkStructureType                sType
      nullptr,                                      // const void                    *pNext
      0,                                            // VkFramebufferCreateFlags       flags
      Vulkan.RenderPass,                            // VkRenderPass                   renderPass
      1,                                            // uint32_t                       attachmentCount
      &Vulkan.FramebufferObjects[i].ImageView,      // const VkImageView             *pAttachments
      300,                                          // uint32_t                       width
      300,                                          // uint32_t                       height
      1                                             // uint32_t                       layers
   };

   if( vkCreateFramebuffer( GetDevice(), &framebuffer_create_info, nullptr, &Vulkan.FramebufferObjects[i].Handle ) != VK_SUCCESS ) {
      printf( "Could not create a framebuffer!\n" );
      return false;
   }
}
return true;
```

5.Tutorial03.cpp，函数 CreateFramebuffers()

vkCreateFramebuffer() 函数要求我们提供类型变量 VkFramebufferCreateInfo 的指示器，因此我们必须首先准备。 它包含以下字段：

sType – 结构类型，此处设为 VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO。
pNext – 大多数时候设为 null 的参数。
flags – 留作将来使用的参数。
renderPass – 帧缓冲器将兼容的渲染通道。

attachmentCount – 帧缓冲器中的附件（pAttachments 阵列中要素）数量。
pAttachments – 图像视图阵列，表示用于帧缓冲器和渲染通道的所有附件。 阵列中的要素（图像视图）对应渲染通道中的附件。
width – 帧缓冲器的宽度。
height – 帧缓冲器的高度。
layers – 帧缓冲器中的层数（OpenGL 借助几何图形着色器进行分层渲染，可将层级指定为将渲染哪些通过特定多边形实现光栅化的碎片）。
帧缓冲器指定哪些图像用作供渲染通道操作的附件。 可以说，它可将图像（图像视图）过渡成特定附件。 为帧缓冲器指定的图像数量必须等于为其创建帧缓冲器的渲染通道中的附件数量。 而且，每个 pAttachments 阵列的要素直接对应渲染通道描述结构中的附件。 渲染通道和帧缓冲器紧密相连，因此我们必须在帧缓冲器创建期间指定渲染通道。 不过，我们不仅可以将帧缓冲器用于指定的渲染通道，还可用于兼容某一指定通道的渲染通道。 一般来说，兼容的渲染通道必须有相同数量的附件，而且对应附件的格式和样本数量必须相同。 但图像布局（初始布局、最终布局，以及面向各子通道的布局）可能各不相同，也不涉及渲染通道兼容性。

完成创建并填充 VkFramebufferCreateInfo 结构后，我们调用 vkCreateFramebuffer() 函数。

上述代码在循环中执行。 帧缓冲器引用图像视图。 此处图像视图针对各交换链图像创建。 因此我们要为交换链图像及其视图创建帧缓冲器。 我们这样做的目的是为了简化渲染循环中调用的代码。 在正常、真实的场景中，我们不（可能）为交换链图像创建帧缓冲器。 我假设了一种更好的解决方法，即渲染单个图像（纹理），然后使用命令缓冲区将该图像的渲染结构拷贝至特定交换链图像。 这样我们就只有三个连接至交换链的简单命令缓冲区。 其他渲染命令独立于交换链，因此更加易于维护。

创建图形管道

现在我们准备好创建图形管道。 管道是逐个处理数据的阶段集合。 Vulkan 中目前有计算管道和图形管道。 计算管道支持我们执行计算工作，比如对游戏中的对象执行物理计算。 图形管道用于绘制操作。

OpenGL 中有多个可编程阶段（顶点、镶嵌、碎片着色器等）和一些固定功能阶段（光栅器、深度测试、混合等）。 Vulkan 中的情况相似。 有些阶段比较类似（如果不相同）。 但整个管道的状态聚集在一个整体对象中。 OpenGL 支持我们随时更改影响渲染操作的状态，我们（大部分时候）可以独立更改各阶段的参数。 我们可以设置着色器程序、深度测试、混合，以及希望的各种状态，然后还可以渲染一些对象。 接下来我们可以仅更改一小部分状

态，并渲染另一对象。 在 Vulkan 中不能执行这类操作（可以说管道具有“免疫力”）。 我们必须准备整个状态，设置管道阶段的参数，并将它们分成管道对象组。 对我来说，一开始这是最令人震惊的信息。 我不能随时更改着色器程序？ 为什么?

最简单有效的解释是，因为这种状态会改变性能影响。 更改整个管道的单个状态可能导致图形硬件执行状态、错误检查等多项后台操作。 不同的硬件厂商可能（并通常）以不同的方式实施此功能。 如果在不同的图形硬件上执行，这样会导致应用以不同的方式执行（意味着会不可预测地影响性能）。 因此对开发人员来说，能够随时更改是一项非常方便的功能。 但遗憾的是，硬件因此会不太方便。

所以在 Vulkan 中，整个管道的状态聚集在一个单个对象中。 创建管道对象时执行所有相关的状态和错误检查。 如果出现问题（比如管道的不同部分设置的不兼容），管道对象创建将失败。 但我们提前了解了这点。 驱动程序无需担心，可以放心地使用毁坏的管道。 它会立即告诉我们这一问题。 但在真正使用期间，在应用的性能关键部分，一切都要正确设置和使用。

这种方法的缺点是，如果以不同的方式（一些不透明、一些半透明，一些启用深度测试等等）绘制对象，我们必须创建多个管道对象，管道对象的多个变量。 遗憾的是，由于着色器不同，不得不创建不同的管道对象。 如果想使用不同的着色器绘制对象，还必须创建多个管道对象，逐个整合着色器程序。 着色器还连接至整个管道状态。 它们使用不同的资源（比如纹理和缓冲区）、渲染不同的颜色附件，并读取不同的附件（可能是之前已渲染过的）。 必须初始化、准备并正确设置这些连接。 我们知道我们的目的，但驱动程序不知道。 因此由我们（而非驱动程序）进行是符合逻辑的最好办法。 一般来说这种方法比较有意义。

开始管道创建流程时，先从着色器开始。

创建着色器模块

创建图形管道要求我们以结构或结构阵列的形式准备大量数据。 第一个数据是所有着色器阶段和着色器程序（在渲染期间用于绑定的特定图形管道）的集合。

在 OpenGL 中我们用 GLSL 编写着色器。 它们经过编译，然后直接链接至应用中的着色器程序。 我们可以在应用中随时使用或停止着色器程序。

而 Vulkan 仅接收着色器的二进制形式 — 一种称为 SPIR-V 的中间语言。不能像在 OpenGL 中那样提供 GLSL 代码。 但有一种官方的独立编译器能够将用 GLSL 编写的着色器转换成二进制 SPIR-V 语言。 为使用该编译器，我们必须离线操作。 准备 SPIR-V 汇编后，我们可以通过它创建着色器模块。 然后将模块合成 VkPipelineShaderStageCreateInfo 结构阵列，从而与其他参数一起用于创建图形管道。

以下代码可通过包含二进制 SPIR-V 的指定文件创建着色器模块。

```
const std::vector<char> code = Tools::GetBinaryFileContents( filename );
if( code.size() == 0 ) {
   return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>();
}

VkShaderModuleCreateInfo shader_module_create_info = {
   VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO,       // VkStructureType                sType
   nullptr,                                           // const void                    *pNext
   0,                                                 // VkShaderModuleCreateFlags      flags
   code.size(),                                       // size_t                         codeSize
   reinterpret_cast<const uint32_t*>(&code[0])        // const uint32_t                *pCode
};

VkShaderModule shader_module;
if( vkCreateShaderModule( GetDevice(), &shader_module_create_info, nullptr, &shader_module ) != VK_SUCCESS ) {
   printf( "Could not create shader module from a %s file!\n", filename );
   return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>();
}

return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>( shader_module, vkDestroyShaderModule, GetDevice() );
```

6.Tutorial03.cpp，函数 CreateShaderModule()

首先准备包含以下字段的 VkShaderModuleCreateInfo 结构：

sType – 结构类型，本示例中设为 VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO。
pNext – 目前不使用的指示器。
flags – 留作将来使用的参数。
codeSize – 传递至 pCode 参数中的代码大小（字节）。
pCode – 包含源代码（二进制 SPIR-V 汇编）的阵列指示器。
为获取文件内容，我准备了一个简单的效用函数 GetBinaryFileContents()，可读取指定文件的所有内容。 它以字符矢量的形式返回内容。

准备该结构后，我们调用 vkCreateShaderModule() 函数并检查一切是否正常运行。

Tools 命名空间的 AutoDeleter<> 类是一种帮助类，可包装特定的 Vulkan 对象句柄，并提取用于删除该对象的函数。 该类与智能指示器类似，可在对象（智能指示器）超出范围时删除分配的内存。 AutoDeleter<> 可提取特定对象的句柄，并在这类对象的类型超出范围时运用提供的函数删除该对象。

```
template<class T, class F>
class AutoDeleter {
public:
  AutoDeleter() :
    Object( VK_NULL_HANDLE ),
    Deleter( nullptr ),
    Device( VK_NULL_HANDLE ) {
  }

  AutoDeleter( T object, F deleter, VkDevice device ) :
```

```
    Object( object ),
    Deleter( deleter ),
    Device( device ) {
}

AutoDeleter( AutoDeleter&& other ) {
    *this = std::move( other );
}

~AutoDeleter() {
    if( (Object != VK_NULL_HANDLE) && (Deleter != nullptr) && (Device != VK_NULL_HANDLE) ) {
        Deleter( Device, Object, nullptr );
    }
}

AutoDeleter& operator=( AutoDeleter&& other ) {
    if( this != &other ) {
        Object = other.Object;
        Deleter = other.Deleter;
        Device = other.Device;
        other.Object = VK_NULL_HANDLE;
    }
    return *this;
}

T Get() {
```

```
    return Object;
  }

  bool operator !() const {
    return Object == VK_NULL_HANDLE;
  }

private:
  AutoDeleter( const AutoDeleter& );
  AutoDeleter& operator=( const AutoDeleter& );
  T               Object;
  F               Deleter;
  VkDevice    Device;
};
```

7.Tools.h

为何我们如此费力地处理一个简单对象？ 着色器模块是创建图形管道所需的对象之一。 但创建管道后，我就不再需要这些着色器模块。 将它们保留下来有时会非常方便，因为我们可能需要创建其他类似的管道。 但在本示例中，创建完图形管道后，我们需要安全地毁坏它们。 通过调用 vkDestroyShaderModule() 函数毁坏着色器模块。 但在本示例中，我们需要在多个地方（多个“ifs”中和整个函数末尾）调用该函数。 因为我们不想忘记哪里需要调用该函数，同时不想出现内存泄漏情况，因此为了方便我准备了这个简单的类。 现在我们不需要记住删除创建的着色器模块，因为它会自动删除。

准备着色器阶段描述

知道如何创建和毁坏着色器模块后，现在我们可以创建着色器阶段数据，以组成图形管道。 正如我所写的，描述哪些着色器阶段应在绑定特定图形管道时处于活跃状态的数据，其形式是包含类型 VkPipelineShaderStageCreateInfo 的要素阵列。 以下代码可创建着色器模块并准备此类阵列：

```
Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule> vertex_shader_module = CreateShaderModule( "Data03/vert.spv" );
Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule> fragment_shader_module = CreateShaderModule( "Data03/frag.spv" );

if( !vertex_shader_module || !fragment_shader_module ) {
  return false;
}

std::vector<VkPipelineShaderStageCreateInfo> shader_stage_create_infos = {
  // Vertex shader
  {
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,      // VkStructureType                  sType
    nullptr,                                                  // const void                      *pNext
    0,                                                        // VkPipelineShaderStageCreateFlags flags
    VK_SHADER_STAGE_VERTEX_BIT,                               // VkShaderStageFlagBits            stage
    vertex_shader_module.Get(),                               // VkShaderModule                   module
    "main",                                                   // const char                       *pName
    nullptr                                                   // const VkSpecializationInfo       *pSpecializationInfo
  },
  // Fragment shader
  {
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,      // VkStructureType                  sType
    nullptr,                                                  // const void                      *pNext
    0,                                                        // VkPipelineShaderStageCreateFlags flags
    VK_SHADER_STAGE_FRAGMENT_BIT,                             // VkShaderStageFlagBits            stage
    fragment_shader_module.Get(),                             // VkShaderModule                   module
```

```
        "main",                                                 // const char                                   *pName
        nullptr                                                 // const VkSpecializationInfo                   *pSpecializationInfo
    }
};
```

8.Tutorial03.cpp，函数 CreatePipeline()

一开始我们创建两个面向顶点和碎片阶段的着色器模块。 用之前所述的函数创建。 如果出现错误，我们从 CreatePipeline() 函数返回，创建的模块将通过包装程序类和提供的删除函数自动删除。

面向着色器模块的代码从包含二进制 SPIR-V 汇编的文件中读取。 这些文件由应用“glslangValidator”生成。 它是一种通过 Vulkan SDK 正式发布的工具，设计目的是为了验证 GLSL 着色器。 但“glslangValidator”也具备编译或将 GLSL 着色器转换为 SPIR-V 二进制文件的功能。 官方 SDK 网站提供了有关该用法命令行的完整解释。 我用以下命令生成用于本教程的 SPIR-V 着色器：

```
glslangValidator.exe -V -H shader.vert > vert.spv.txt

glslangValidator.exe -V -H shader.frag > frag.spv.txt
```

“glslangValidator”提取指定文件并通过该文件生成 SPIR-V 文件。 输入文件的扩展文件（顶点着色器的“.vert”、几何图形着色器的“.geom”等）将自动检测着色器阶段的类型。 生成文件的名称可以指定，也可以默认为“<stage>.spv”形式。 因此在本示例中将生成“vert.spv”和“frag.spv”文件。

SPIR-V 文件为二进制格式，不容易读取和分析 — 但也有可能。 使用“-H”选项时，“glslangValidator”以易于读取的形式输出 SPIR-V。 这种形式打印为标准输出，因此我使用“> *.spv.txt”重定向运算符。

以下内容是为顶点阶段生成 SPIR-V 汇编的“shader.vert”文件内容：

```
#version 400
```

```
void main() {
	vec2 pos[3] = vec2[3]( vec2(-0.7, 0.7), vec2(0.7, 0.7), vec2(0.0, -0.7) );
	gl_Position = vec4( pos[gl_VertexIndex], 0.0, 1.0 );
}
```
9.shader.vert

大家看，我对用于渲染三角形的所有顶点位置进行了硬编码。 通过特定于 Vulkan 的“gl_VertexIndex”内置变量为它们编入了索引。 在最简单的场景中，当（此时）使用非索引绘制命令时，该数值从绘制命令的“firstVertex”参数的数值（在提供的示例中为 0）开始。

我之前写过这一具有争议的部分 — 这种方法可接受，也有效，但不太便于维护，也支持我们跳过创建图形管道所需的“结构填充”部分。 我选择使用这种方法，是为了尽可能地缩短和简化本教程。 在下一教程中，我将演示一种更常用的顶点数量绘制方法，类似于在 OpenGL 中使用顶点阵列和索引。

以下是“shader.frag”文件的碎片着色器的源代码，用于生成面向碎片阶段的 SPIRV-V 汇编：

```
#version 400

layout(location = 0) out vec4 out_Color;

void main() {
   out_Color = vec4( 0.0, 0.4, 1.0, 1.0 );
}
```
10.shader.frag

在 Vulkan 着色器中（当从 GLSL 转换成 SPIR-V 时），需要使用布局限定符。 这里我们指定哪些输出（颜色）附件保存碎片着色器生成的颜色值。 因为我们仅使用一个附件，所以必须指定第一个可用位置（零）。

现在了解如何为使用 Vulkan 的应用准备着色器后，就可以进行下一步了。 创建两个着色器模块后，检查这些操作是否成功。 如果成功，我们可以开始准备着色器阶段描述，以继续创建图形管道。

我们需要为每个启用的着色器阶段准备 VkPipelineShaderStageCreateInfo 结构实例。 这些结构阵列及其要素数量一起用于图形管道创建信息结构（提供给创建图形管道的函数）。 VkPipelineShaderStageCreateInfo 结构包含以下字段：

sType – 我们所准备的结构类型，此处必须等于 VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 留作将来使用的参数。
stage – 我们描述的着色器阶段类型（比如顶点、镶嵌控制等）。
module – 着色器模块句柄，包含用于特定阶段的着色器。
pName – 提供的着色器的切入点名称。
pSpecializationInfo – VkSpecializationInfo 结构指示器，留作现在使用，设为 null。
创建图形管道时，我们不创建太多 (Vulkan) 对象。 大部分数据只用此类结构的形式展示。

准备顶点输入描述

现在我们必须提供用于绘制的输入数据描述。 这类似于 OpenGL 的顶点数据：属性、组件数量、供数据提取的缓冲区、数据步长，或步进率。 当然，在 Vulkan 中准备这类数据的方式不同，但大体意思相同。 幸运的是，在本教程中由于顶点数据已硬编码成顶点着色器，因此我们几乎可以完全跳过这一步骤，并用 null 和零填充 VkPipelineVertexInputStateCreateInfo：

```
VkPipelineVertexInputStateCreateInfo vertex_input_state_create_info = {
  VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,      // VkStructureType                              sType
  nullptr,                                                        // const void                                  *pNext
  0,                                                              // VkPipelineVertexInputStateCreateFlags        flags;
```

```
    0,                                                                                                  //  uint32_t
vertexBindingDescriptionCount
    nullptr,                                                  // const VkVertexInputBindingDescription        *pVertexBindingDescriptions
    0,                                                                                                  //  uint32_t
vertexAttributeDescriptionCount
    nullptr                                                   // const VkVertexInputAttributeDescription      *pVertexAttributeDescriptions
};
```

11. Tutorial03.cpp，函数 CreatePipeline()

为清晰起见，以下为 VkPipelineVertexInputStateCreateInfo 结构的成员描述：

sType – 结构类型，此处为 VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO。
pNext – 特定于扩展的结构指示器。
flags – 留作将来使用的参数。
vertexBindingDescriptionCount – pVertexBindingDescriptions 阵列中的要素数量。
pVertexBindingDescriptions – 包含描述输入顶点数据（步长和步进率）的要素阵列。
vertexAttributeDescriptionCount – pVertexAttributeDescriptions 阵列中的要素数量。
pVertexAttributeDescriptions – 包含描述顶点属性（位置、格式、位移）的要素阵列。
准备输入汇编描述

下一步骤要求描述如何将顶点汇编成基元。 和 OpenGL 一样，我们必须指定欲使用的拓扑：点、线、三角形、三角扇等。

```
VkPipelineInputAssemblyStateCreateInfo input_assembly_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,    // VkStructureType                              sType
    nullptr,                                                        // const void                                   *pNext
    0,                                                              // VkPipelineInputAssemblyStateCreateFlags      flags
```

```
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,                    // VkPrimitiveTopology                    topology
    VK_FALSE                                                // VkBool32                               primitiveRestartEnable
};
```

12.Tutorial03.cpp，函数 CreatePipeline()

我们通过 VkPipelineInputAssemblyStateCreateInfo 结构完成该步骤，其中包含以下成员：

sType – 结构类型，此处设为 VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO。
pNext – 目前不使用的指示器。
flags – 留作将来使用的参数。
topology – 描述如何组织顶点以形成基元的参数。
primitiveRestartEnable – 告知特定索引值（执行索引绘制时）是否重新开始汇编特定基元的参数。
准备视口描述

我们已处理完输入数据。 现在必须指定输出数据的形式，图形管道连接碎片（比如光栅化、窗口（视口）、深度测试等）的所有部分。 这里必须准备的第一个数据集为视口状态，以指定我希望绘制哪部分图形（或纹理，或窗口）。

```
VkViewport viewport = {
    0.0f,                                                   // float                                  x
    0.0f,                                                   // float                                  y
    300.0f,                                                 // float                                  width
    300.0f,                                                 // float                                  height
    0.0f,                                                   // float                                  minDepth
    1.0f                                                    // float                                  maxDepth
};
```

```
VkRect2D scissor = {
    {                                                                   // VkOffset2D                                   offset
        0,                                                              // int32_t                                      x
        0                                                               // int32_t                                      y
    },
    {                                                                   // VkExtent2D                                   extent
        300,                                                            // int32_t                                      width
        300                                                             // int32_t                                      height
    }
};

VkPipelineViewportStateCreateInfo viewport_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO,              // VkStructureType                              sType
    nullptr,                                                            // const void                                   *pNext
    0,                                                                  // VkPipelineViewportStateCreateFlags           flags
    1,                                                                  // uint32_t                                     viewportCount
    &viewport,                                                          // const VkViewport                             *pViewports
    1,                                                                  // uint32_t                                     scissorCount
    &scissor                                                            // const VkRect2D                               *pScissors
};
```

13.Tutorial03.cpp，函数 CreatePipeline()

在本示例中，用法很简单：仅将视口坐标设置为预定义的值。 不用检查待渲染的交换链图像的大小。 但请记住，在真实生产应用中，必须执行这一操作，因为规范规定，视口的尺寸不能超过待渲染的附件尺寸。

为指定视口参数，我们填充包含以下字段的 VkViewport 结构：

x – 视口左侧。
y – 视口上侧。
width – 视口的宽度。
height – 视口的高度。
minDepth – 用于深度计算的最小深度值。
maxDepth – 用于深度计算的最大深度值。
指定视口坐标时，请记住，起点与 OpenGL 中的不同。 此处我们指定视口的左上角（而非左下角）。

另外一点值得注意的是，minDepth 和 maxDepth 值必须位于 0.0 - 1.0（包含 1.0）之间，但 maxDepth 可以小于 minDepth。 这样会以“相反”的顺序计算深度。

接下来必须指定用于 scissor 测试的参数。 scissor 测试与 OpenGL 类似，将碎片生成仅限制在指定的矩形区域。 但在 Vulkan 中，scissor 测试始终处于启用状态，无法关闭。 我们仅提供与为视口提供的相似的值。 尝试更改这些值，看看对生成的图像产生怎样的影响。

scissor 测试没有专用的结构。 为提供用于该测试的数据，我们填充 VkRect2D 结构，其中包含两个类似的结构成员。 第一个是 VkOffset2D，包含以下成员：

x – 用于 scissor 测试的矩形区域的左侧
y – 矩形区域的上侧
第二个成员的类型为 VkExtent2D，并包含以下字段：

width – scissor 矩形区域的宽度
height – scissor 区域的高度
一般来说，通过 VkRect2D 结构为 scissor 测试提供的数据与为视口准备的数据在意义上相似。

为视口和 scissor 测试准备数据后，我们最后可以填充用于创建管道的结构。 该结构称为 VkPipelineViewportStateCreateInfo，并包含以下字段：

sType – 结构类型，此处为 VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 留作将来使用的参数。
attachmentCount – pViewports 阵列中的要素数量。
pViewports – 描述绑定特定管道时所使用的视口参数的要素阵列。
scissorCount – pScissors 阵列中的要素数量。
pScissors – 描述针对各视口的 scissor 测试参数的要素阵列。
请记住，viewportCount 和 scissorCount 参数必须相等。 我们还允许指定更多视口，但之后必须启用 multiViewport 特性。

准备光栅化状态描述

图形管道创建的下一部分适用于光栅化状态。 我们必须指定如何对多边形进行光栅化（改成碎片），是希望为整个多边形生成碎片，还是仅为边缘生成碎片（多边形模式），或者希望看到多边形的正面或背面，还是同时看到这两面（背面剔除）。 我们还可以提供深度偏差参数，或指明是否希望启用深度夹紧 (depth clamp)。 整个状态将封装至 VkPipelineRasterizationStateCreateInfo。 它包含以下成员：

sType – 结构类型，本示例中为 VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 留作将来使用的参数。
depthClampEnable – 描述是希望将光栅化基元的深度值夹在视锥上（真值），还是希望进行正常裁剪（假值）的参数。
rasterizerDiscardEnable – 禁用碎片生成（在光栅化关闭碎片着色器之前舍弃基元）。
polygonMode – 控制如何为特定基元生成碎片（三角形模式）：为整个三角形生成，仅为其边缘生成，或是仅为其顶点生成。
cullMode – 选择用于剔除的三角形面（如果启用）。
frontFace – 选择将哪个面视作正面（取决于缠绕顺序）。
depthBiasEnable – 启用或禁用偏置碎片的深度值。

depthBiasConstantFactor – 启用偏置时添加至碎片深度值的常数因子。
depthBiasClamp – 适用于碎片深度的最大（或最小）偏差值。
depthBiasSlopeFactor – 启用偏置时在深度计算期间适用于碎片斜度的因子。
lineWidth – 光栅化线条的宽度。
以下源代码负责设置本示例中的光栅化状态：

```
VkPipelineRasterizationStateCreateInfo rasterization_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO,     // VkStructureType                              sType
    nullptr,                                                        // const void                                   *pNext
    0,                                                              // VkPipelineRasterizationStateCreateFlags      flags
    VK_FALSE,                                                       // VkBool32                                     depthClampEnable
    VK_FALSE,                                                       // VkBool32                                     rasterizerDiscardEnable
    VK_POLYGON_MODE_FILL,                                           // VkPolygonMode                                polygonMode
    VK_CULL_MODE_BACK_BIT,                                          // VkCullModeFlags                              cullMode
    VK_FRONT_FACE_COUNTER_CLOCKWISE,                                // VkFrontFace                                  frontFace
    VK_FALSE,                                                       // VkBool32                                     depthBiasEnable
    0.0f,                                                           // float                                        depthBiasConstantFactor
    0.0f,                                                           // float                                        depthBiasClamp
    0.0f,                                                           // float                                        depthBiasSlopeFactor
    1.0f                                                            // float                                        lineWidth
};
```

14.Tutorial03.cpp，函数 CreatePipeline()

在本教程中，我们禁用尽可能多的参数，以简化流程、代码和渲染操作。 这里的重要参数可设置适用于多边形光栅化、背面剔除，以及类似于 OpenGL 的逆时针正面的（典型）填充模式。 深度偏置和夹紧也处于禁用状态（要启用深度夹紧，我们首先需要在逻辑设备创建期间启用专用特定；同样，对多边形模式来说，我们也必须进行相同的操作，而非“填充”）。

设置多点采样状态描述

在 Vulkan 中，创建图形管道时，我们还必须指定与多点采用相关的状态。 该步骤可使用 VkPipelineMultisampleStateCreateInfo 结构来完成。 它包含以下成员：

sType – 结构类型，此处为 VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 留作将来使用的参数。
rasterizationSamples – 用于光栅化的每像素样本数量。
sampleShadingEnable – 指定是按照样本着色（启用），还是按照碎片着色（禁用）的参数。
minSampleShading – 指定应用于特定碎片着色期间的独有样本位置的最少数量。
pSampleMask – 静态覆盖范围样本掩码的阵列指示器；可设为 null。
alphaToCoverageEnable – 控制碎片的阿尔法值是否用于覆盖范围计算。
alphaToOneEnable – 控制是否替换碎片的阿尔法值。
在本示例中，我希望最大限度减少问题的发生，因此将参数设为通常禁用多点采样的值 — 每特定像素仅一个样本，其他参数均处于关闭状态。 请记住，如果我们希望启用样本着色或将阿尔法设为 1，还必须分别启用两个特性。 以下是用于准备 VkPipelineMultisampleStateCreateInfo 结构的源代码：

```
VkPipelineMultisampleStateCreateInfo multisample_state_create_info = {
  VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO,       // VkStructureType                          sType
  nullptr,                                                       // const void                              *pNext
  0,                                                             // VkPipelineMultisampleStateCreateFlags    flags
  VK_SAMPLE_COUNT_1_BIT,                                         // VkSampleCountFlagBits                    rasterizationSamples
  VK_FALSE,                                                      // VkBool32                                 sampleShadingEnable
  1.0f,                                                          // float                                    minSampleShading
  nullptr,                                                       // const VkSampleMask                      *pSampleMask
```

```
    VK_FALSE,                                                   // VkBool32                 alphaToCoverageEnable
    VK_FALSE                                                    // VkBool32                 alphaToOneEnable
};
```

15.Tutorial03.cpp，函数 CreatePipeline()

设置混合状态描述

在创建图形管道期间，我们另外还需准备混合状态（它还包括逻辑操作）。

```
VkPipelineColorBlendAttachmentState color_blend_attachment_state = {
    VK_FALSE,                                                   // VkBool32                 blendEnable
    VK_BLEND_FACTOR_ONE,                                        // VkBlendFactor            srcColorBlendFactor
    VK_BLEND_FACTOR_ZERO,                                       // VkBlendFactor            dstColorBlendFactor
    VK_BLEND_OP_ADD,                                            // VkBlendOp                colorBlendOp
    VK_BLEND_FACTOR_ONE,                                        // VkBlendFactor            srcAlphaBlendFactor
    VK_BLEND_FACTOR_ZERO,                                       // VkBlendFactor            dstAlphaBlendFactor
    VK_BLEND_OP_ADD,                                            // VkBlendOp                alphaBlendOp
    VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |       // VkColorComponentFlags    colorWriteMask
    VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT
};

VkPipelineColorBlendStateCreateInfo color_blend_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO,   // VkStructureType          sType
    nullptr,                                                    // const void               *pNext
    0,                                                          // VkPipelineColorBlendStateCreateFlags  flags
    VK_FALSE,                                                   // VkBool32                 logicOpEnable
```

```
    VK_LOGIC_OP_COPY,                                   // VkLogicOp                                    logicOp
    1,                                                  // uint32_t                                     attachmentCount
    &color_blend_attachment_state,                      // const VkPipelineColorBlendAttachmentState    *pAttachments
    { 0.0f, 0.0f, 0.0f, 0.0f }                          // float                                        blendConstants[4]
};
```

16.Tutorial03.cpp，函数 CreatePipeline()

VkPipelineColorBlendStateCreateInfo 结构可用于设置最终颜色操作。 它包含以下字段：

sType – 结构类型，在本示例中设置为 VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO。
pNext – 留作将来特定用于扩展的指示器。
flags – 同样留作将来使用的参数。
logicOpEnable – 指示是否希望启用有关像素的逻辑操作。
logicOp – 预执行的逻辑操作类型（比如拷贝、清空等）。
attachmentCount – pAttachments 阵列中的要素数量。
pAttachments – 包含颜色附件状态参数的阵列，其中这些颜色附件用于子通道，以便其绑定特定图形管道。
blendConstants – 包含四个要素以及用于混合操作的颜色值的阵列（使用专用混合因子时）。
attachmentCount 和 pAttachments 需要更多信息。 如果想执行绘制操作，我们需要设置参数，其中最重要的是图形管道、渲染通道和帧缓冲器。 显卡需要知道绘制的方式（描述渲染状态、着色器、测试等的图形管道）和绘制的位置（渲染通道提供通用设置；帧缓冲器指定使用哪些图像）。 我之前说过，渲染通道指定如何排列操作、具有哪些相关性、何时渲染特定附件，以及何时读取相同附件。 这些阶段以子通道的形式执行。 我们可以（但不是必须）为每项绘制操作启用/使用不同的管道。 我们进行绘制时，必须记住要将绘制成附件集。 该集合在渲染通道中定义，其中描述所有的颜色、输入和深度附件（帧缓冲器仅指定哪些图像用于这些附件）。 就混合状态而言，我们可以指定是否希望启用混合。 这一操作通过 pAttachments 阵列来完成。 每个要素都必须对应渲染通道中定义的颜色附件。 因此 pAttachments 阵列中 attachmentCount 要素的值必须等于渲染通道中定义的颜色附件数量。

还有一个限制条件。 在默认情况下，pAttachments 阵列中的所有要素都必须包含相同的值，必须以相同的方式指定，而且必须相同。 默认情况下，执行混合（和颜色掩码）的方式与所有附件相同。 为何是一个阵列？ 为什么只需指定一个值？ 因为有一项特性可以支持我们执行为每个活跃的颜色附件

独立、独特的混合。 如果在创建设备期间启用独立混合，我们可以为各颜色附件提供不同的值。

pAttachments 阵列要素的类型为 VkPipelineColorBlendAttachmentState。 该结构包含以下成员：

blendEnable – 指示是否希望启用混合。
srcColorBlendFactor – 面向源（入站）碎片颜色的混合因子。
dstColorBlendFactor – 面向目标颜色的混合因子（保存在帧缓冲器中，位置与入站碎片相同）。
colorBlendOp – 待执行操作的类型（乘法、加法等）。
srcAlphaBlendFactor – 面向源（入站）碎片阿尔法值的混合因子。
dstAlphaBlendFactor – 面向目标阿尔法值的混合因子（保存在帧缓冲器中）。
alphaBlendOp – 面向阿尔法混合执行的操作类型。
colorWriteMask – 选择（启用）编写哪个 RGBA 组件的位掩码。
在本示例中，我们禁用混合操作，这样其他所有参数都将处于不相关状态。 除 colorWriteMask 外，我们选择编写所有组件，但您可以自由检查该参数变成其他 R、G、B、A 组合后，将会发生什么。

创建管道布局

创建管道之前，我们需要做的最后一件事是创建相应的管道布局。 管道布局描述管道可访问的所有资源。 在本示例中，我们必须指定着色器将使用多少纹理，以及哪些着色器阶段将访问它们。 当然还会涉及到其他资源。 除着色器阶段外，我们还必须描述资源的类型（纹理、缓冲区）、总数量，以及布局。 该布局可以比作 OpenGL 的活跃纹理和 shader uniform。 在 OpenGL 中，我们将纹理绑定至所需的纹理图像单元，而且不为 shader uniform 提供纹理句柄，而是提供纹理图像单元（绑定实际纹理）的 ID（我们提供与特定纹理相关的单元编号）。

Vulkan 中的情况类似。 我们创建某种内存布局形式：首先是两个缓冲区，接下来是三个纹理和一个图像。 这种内存“结构”称为集，这些集的集合将提供给管道。 在着色器中，我们使用这些集（布局）中的内存“位置”访问指定的资源。 这可通过布局 (set = X, binding = Y) 分类符来完成，也可以解释为：从 Y 内存位置和 X 集提取资源。

管道布局可视作着色器阶段和着色器资源之间的交互，因为它提取这些资源组，并描述如何收集并向管道提供这些资源。

该流程比较复杂，我计划为其另外编写一节教程。 这里我们不使用其他资源，因为我展示的是关于创建“空”管道布局的示例。

```
VkPipelineLayoutCreateInfo layout_create_info = {
   VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,    // VkStructureType                sType
   nullptr,                                          // const void                    *pNext
   0,                                                // VkPipelineLayoutCreateFlags    flags
   0,                                                // uint32_t                       setLayoutCount
   nullptr,                                          // const VkDescriptorSetLayout   *pSetLayouts
   0,                                                // uint32_t                       pushConstantRangeCount
   nullptr                                           // const VkPushConstantRange     *pPushConstantRanges
};

VkPipelineLayout pipeline_layout;
if( vkCreatePipelineLayout( GetDevice(), &layout_create_info, nullptr, &pipeline_layout ) != VK_SUCCESS ) {
   printf( "Could not create pipeline layout!\n" );
   return Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout>();
}

return Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout>( pipeline_layout, vkDestroyPipelineLayout, GetDevice() );
```

17.Tutorial03.cpp，函数 CreatePipelineLayout()

为创建管道布局，必须首先创建类型变量 VkPipelineLayoutCreateInfo。 它包含以下字段：

sType – 结构类型，本示例中为 VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO。

pNext – 为扩展功能预留的参数。
flags – 留作将来使用的参数。
setLayoutCount – 本布局中包含的描述符集的数量。
pSetLayouts – 包含描述符布局描述的阵列指示器。
pushConstantRangeCount – push constant 范围（稍后介绍）的数量。
pPushConstantRanges – 描述（特定管道中）着色器中使用的所有 push constant 范围的阵列。
在本示例中，我们创建“空”布局，因此几乎所有字段都设置为 null 或零。

我们在这里不使用 push constant，不过这一概念值得介绍。 Vulkan 中的 push constant 支持我们修改用于着色器的常量变量的数据。 这里为 push constant 预留了少量的特殊内存。 我们通过 Vulkan 命令（而非内存更新）更新其数值，push constant 值的更新速度预计快于正常内存写入。

如上述示例所示，我还会将管道布局包装至“AutoDeleter”对象。管道创建、描述符集绑定（启用/激活着色器与着色器资源之间的交互），以及 push constant 设置期间都需要管道布局。 除管道创建外，本教程不执行任何其他操作。 因此在这里，创建管道后，不再需要使用布局。 为避免内存泄漏，离开创建图形管道的函数后，我立即使用了该帮助类毁坏布局。

创建图形管道

现在我们准备了创建图形管道需要的所有资源。 以下代码可帮助完成这一操作：

```
Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout> pipeline_layout = CreatePipelineLayout();
if( !pipeline_layout ) {
  return false;
}

VkGraphicsPipelineCreateInfo pipeline_create_info = {
  VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,                    // VkStructureType                                  sType
```

```
    nullptr,                                                    // const void                                   *pNext
    0,                                                          // VkPipelineCreateFlags                        flags
    static_cast<uint32_t>(shader_stage_create_infos.size()),    // uint32_t                                     stageCount
    &shader_stage_create_infos[0],                              // const VkPipelineShaderStageCreateInfo        *pStages
    &vertex_input_state_create_info,                            // const VkPipelineVertexInputStateCreateInfo   *pVertexInputState;
    &input_assembly_state_create_info,                          // const VkPipelineInputAssemblyStateCreateInfo *pInputAssemblyState
    nullptr,                                                    // const VkPipelineTessellationStateCreateInfo  *pTessellationState
    &viewport_state_create_info,                                // const VkPipelineViewportStateCreateInfo      *pViewportState
    &rasterization_state_create_info,                           // const VkPipelineRasterizationStateCreateInfo *pRasterizationState
    &multisample_state_create_info,                             // const VkPipelineMultisampleStateCreateInfo   *pMultisampleState
    nullptr,                                                    // const VkPipelineDepthStencilStateCreateInfo  *pDepthStencilState
    &color_blend_state_create_info,                             // const VkPipelineColorBlendStateCreateInfo    *pColorBlendState
    nullptr,                                                    // const VkPipelineDynamicStateCreateInfo       *pDynamicState
    pipeline_layout.Get(),                                      // VkPipelineLayout                             layout
    Vulkan.RenderPass,                                          // VkRenderPass                                 renderPass
    0,                                                          // uint32_t                                     subpass
    VK_NULL_HANDLE,                                             // VkPipeline                                   basePipelineHandle
    -1                                                          // int32_t                                      basePipelineIndex
};

if( vkCreateGraphicsPipelines( GetDevice(), VK_NULL_HANDLE, 1, &pipeline_create_info, nullptr, &Vulkan.GraphicsPipeline ) != VK_SUCCESS ) {
    printf( "Could not create graphics pipeline!\n" );
    return false;
}
return true;
```

18.Tutorial03.cpp，函数 CreatePipeline()

我们首先创建封装在“AutoDeleter” 对象中的管道布局。 接下来填充 VkGraphicsPipelineCreateInfo 类型的结构 。 它包含多个字段。 以下简要介绍它们：

sType – 结构类型，此处为 VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO。
pNext – 留作将来用于扩展的参数。
flags – 这次该参数不留做将来使用，而是用于控制如何创建管道：是创建衍生管道（如果承袭另一管道），还是支持从该管道创建衍生管道。 我们还可以禁用优化，这样可缩短创建管道的时间。
stageCount – pStages 参数中描述的阶段数据；必须大于 0。
pStages – 包含活跃着色器阶段（用着色器模块创建而成）描述的阵列；各阶段必须具有唯一性（指定特定阶段的数量不能超过一次）。 还必须展示一个顶点阶段。
pVertexInputState – 包含顶点输入状态描述的变量指示器。
pInputAssemblyState – 包含输入汇编描述的变量指示器。
pTessellationState – 镶嵌阶段描述指示器；如果镶嵌处于禁用状态，则可设置为 null。
pViewportState – 指定视口参数的变量指示器；如果光栅化处于禁用状态，则可设置为 null。
pRasterizationState – 指定光栅化行为的变量指示器。
pMultisampleState – 定义多点采样的变量指示器；如果镶嵌处于禁用状态，则可设置为 null。
pDepthStencilState – 深度/模板参数描述指示器；在两种情况下可设置为 null：光栅化处于禁用状态，或者我们不在渲染通道中使用深度/模板附件。
pColorBlendState – 包含颜色混合/写入掩码状态的变量指示器；也在两种情况下可设置为 null：光栅化处于禁用状态，或者我们不在渲染通道中使用任何颜色附件。
pDynamicState – 指定可动态设置哪部分图形管道的变量指示器；如果认为整个状态是静止状态（仅通过该创建信息结构定义），可设置为 null。
layout – 管道布局对象的句柄，该对象描述在着色器中访问的资源。
renderPass – 渲染通道对象句柄；管道可用于任何兼容已提供通道的渲染通道。
subpass – 供使用管道的子通道编号（索引）。
basePipelineHandle – 支持衍生该管道的管道句柄。
basePipelineIndex – 支持衍生该管道的管道索引。

创建新管道时，我们可以承袭其他管道的部分参数。 这意味着两个管道存在共同之处。 比较好的示例是着色器代码。 我们不指定哪些字段是相同的，但从其他管道继承的通用信息可显著加快管道创建速度。 但为什么有两个字段指示“父”管道？ 我们不能使用两个 — 一次仅使用一个。 我们使用句柄时，表示“父”管道已创建完成，我们正从提供句柄的管道衍生新管道。 但管道创建函数支持我们一次创建多个管道。 使用第二个参数“父”管道索引可以帮助我们用相同的调用方式创建“父”管道和“子”管道。 我们仅指定图形管道创建信息结构阵列，而且该阵列提供给管道创建函数。 因此“basePipelineIndex”是该阵列中管道创建信息的索引。 我们只需记住，“父”管道在该阵列中必须先创建（索引必须小），而且必须通过“allow derivatives”标记集创建。

在本示例中，我们创建整体处于静止状态的管道（“pDynamicState”参数设为 null）。 但什么是静止状态？ 为支持部分灵活性和减少管道对象的创建数量，我们引入了动态状态。 我们可以通过“pDynamicState”参数定义可通过其他 Vulkan 命令动态设置哪部分图形管道，以及在管道创建期间将哪部分一次设置为静态。 动态状态包括视口、线条宽度、混合常量等参数，或部分模板参数。 如果我们指定特定状态为动态，那么忽略管道创建信息结构中与该状态有关的参数。 在渲染期间，必须使用适当的命令设置特定状态，因为该状态的初始值可能没有定义。

因此经过这项繁重的准备工作后，我们可以创建图形管道。 这一过程通过调用 vkCreateGraphicsPipelines() 函数完成，提取管道创建信息结构的指示器阵列。 如果进展顺利，该函数将返回 VK_SUCCESS，图形管道句柄并保存在我们提供了地址的变量中。 现在我们可以开始进行绘制。

准备绘制命令

之前的教程介绍过命令缓冲区概念。 这里我简要介绍使用其中的哪些，以及如何使用。

命令缓冲区是 GPU 命令的容器。 如果想在设备上执行某项任务，我们可以通过命令缓冲区来完成。 这表示我们必须准备处理数据（即在屏幕上绘制图形）的命令集，并将这些命令记录在命令缓冲区中。 然后将整个缓冲区提交至设备的队列。 这种提交操作将告诉设备：我希望你现在替我执行一些任务。

为记录命令，我们必须首先分配命令缓冲区。 它们通过命令池进行分配，可视作内存块。 如果命令缓冲区希望变大（因为我们记录了许多复杂命令），它可以增长，并使用命令池（进行分配）的其他内存。 因此我们首先必须创建命令池。

创建命令池

创建命令池非常简单，如下所示：

```
VkCommandPoolCreateInfo cmd_pool_create_info = {
   VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,        // VkStructureType                sType
   nullptr,                                           // const void                    *pNext
   0,                                                 // VkCommandPoolCreateFlags       flags
   queue_family_index                                 // uint32_t                       queueFamilyIndex
};

if( vkCreateCommandPool( GetDevice(), &cmd_pool_create_info, nullptr, pool ) != VK_SUCCESS ) {
   return false;
}
return true;
```

19.Tutorial03.cpp，函数 CreateCommandPool()

首先准备类型变量 VkCommandPoolCreateInfo。 它包含以下字段：

sType – 标准结构类型，此处设置为 VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 指示命令池及其分配的命令缓冲区的使用场景；即我们告知驱动程序，从该命令池分配的命令缓冲区将存在较短时间；如果没有特定用法，可将其设置为零。
queueFamilyIndex – 队列家族（我们为其创建命令池）索引。
请记住，从特定命令池分配的命令缓冲区只能提交至命令池创建期间指定的队列家族的队列。

要创建命令池，我们只需调用 vkCreateCommandPool() 函数。

分配命令缓冲区

现在我们准备好命令池后，可以通过它分配命令缓冲区。

```
VkCommandBufferAllocateInfo command_buffer_allocate_info = {
  VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO, // VkStructureType                sType
  nullptr,                                        // const void                   *pNext
  pool,                                           // VkCommandPool                commandPool
  VK_COMMAND_BUFFER_LEVEL_PRIMARY,                // VkCommandBufferLevel         level
  count                                           // uint32_t                     bufferCount
};

if( vkAllocateCommandBuffers( GetDevice(), &command_buffer_allocate_info, command_buffers ) != VK_SUCCESS ) {
  return false;
}
return true;
```

20.Tutorial03.cpp，函数 AllocateCommandBuffers()

为分配命令缓冲区，我们指定一个结构类型变量。 这次的类型为 VkCommandBufferAllocateInfo，其中包含以下三个成员：

sType – 结构类型；此处为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO。
pNext – 为扩展功能预留的指示器。
commandPool – 我们希望命令缓冲区从中提取内存的池。
level – 命令缓冲区级别；包含两个级别：主缓冲区和次缓冲区；此时我们仅有兴趣使用主命令缓冲区。

bufferCount – 希望分配的命令缓冲区数量。
为分配命令缓冲区，我们调用 vkAllocateCommandBuffers() 函数并检查是否调用成功。 通过一次函数调用可以一次性分配多个缓冲区。

我准备了一个简单的缓冲区分配函数，以向大家展示如果包装 Vulkan 函数，以方便使用。 以下是两个此类包装程序函数的用法，分别用于创建命令池和分配命令缓冲区。

```
if( !CreateCommandPool( GetGraphicsQueue().FamilyIndex, &Vulkan.GraphicsCommandPool ) ) {
   printf( "Could not create command pool!\n" );
   return false;
}

uint32_t image_count = static_cast<uint32_t>(GetSwapChain().Images.size());
Vulkan.GraphicsCommandBuffers.resize( image_count, VK_NULL_HANDLE );

if( !AllocateCommandBuffers( Vulkan.GraphicsCommandPool, image_count, &Vulkan.GraphicsCommandBuffers[0] ) ) {
   printf( "Could not allocate command buffers!\n" );
   return false;
}
return true;
```

21.Tutorial03.cpp，函数 CreateCommandBuffers()

大家看，我们正在为显卡队列家族索引创建命令池。 所有图像状态过渡和绘制操作都将在显卡队列上执行。 演示操作在另一队列上执行（如果演示队列与显卡队列不同），但执行该操作时不需要使用命令缓冲区。

而且我们还为各交换链图像分配命令缓冲区。 这里我们提取图像数量，并将其提供给简单的“wrapper”函数，以便分配命令缓冲区。

记录命令缓冲区

从命令池分配到命令缓冲区后，最后我们可以记录在屏幕上进行绘制的操作。 首先必须准备执行记录操作所需的数据集。 一部分数据与所有命令缓冲区相同，另一部分引用特定的交换链图像。 以下是独立于交换链图像的代码：

```
VkCommandBufferBeginInfo graphics_commandd_buffer_begin_info = {
  VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,      // VkStructureType                        sType
  nullptr,                                          // const void                             *pNext
  VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT,     // VkCommandBufferUsageFlags              flags
  nullptr                                           // const VkCommandBufferInheritanceInfo   *pInheritanceInfo
};

VkImageSubresourceRange image_subresource_range = {
  VK_IMAGE_ASPECT_COLOR_BIT,                        // VkImageAspectFlags                     aspectMask
  0,                                                // uint32_t                               baseMipLevel
  1,                                                // uint32_t                               levelCount
  0,                                                // uint32_t                               baseArrayLayer
  1                                                 // uint32_t                               layerCount
};

VkClearValue clear_value = {
  { 1.0f, 0.8f, 0.4f, 0.0f },                       // VkClearColorValue                      color
};

const std::vector<VkImage>& swap_chain_images = GetSwapChain().Images;
```

22.Tutorial03.cpp，函数 RecordCommandBuffers()

执行命令缓冲区记录类似于 OpenGL 的绘制列表，其中我们通过调用 glNewList() 函数开始记录列表。 接下来准备绘制命令集，然后关闭列表并停止记录 (glEndList())。 因此我们首先要做的是准备类型变量 VkCommandBufferBeginInfo。 它用于开始记录命令缓冲区的时候，告知驱动程序有关命令缓冲区的类型、内容和用法等信息。 该类型变量包含以下成员：

sType – 标准结构类型，此处设置为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO。
pNext – 为扩展功能预留的指示器。
flags – 描述预期用法的参数（即我们是否想仅提交一次，并毁坏/重置该命令缓冲区，或者是否可以在之前的提交完成之前再次提交该缓冲区）。
pInheritanceInfo – 仅在记录次命令缓冲区时所使用的参数。
接下来描述为其设置图像内存壁垒的图像区域或部分。 此处我们设置壁垒以指定不同家族的列队将引用特定图像。 这一操作通过类型变量 VkImageSubresourceRange 完成，该变量包含以下成员：

aspectMask – 描述图像的”类型“，是否用于颜色、深度或模板数据。
baseMipLevel – 供我们执行操作的第一个 mipmap 层的编号。
levelCount – 供我们执行操作的 mipmap 层（包括基础层）数量。
baseArrayLayer – 参与操作的图像的第一个阵列层的编号。
layerCount – 将进行修改的层级（包括基础层）数量。
接下来设置面向图像的清空值。 进行绘制之前需要清空图像。 在之前的教程中，我们自己明确执行这一操作。 这里图像作为渲染通道附件加载操作的一部分进行清空。 我们要设置成“clear”，必须指定图像需清空的颜色。 这一操作通过类型变量 VkClearValue（其中我们提供了 R、G、B、A 四个值）完成。

到目前为止，我们创建的变量均独立于图像本身，因此我们在循环前完成了指定行为。 现在我们开始记录命令缓冲区：

```
for( size_t i = 0; i < Vulkan.GraphicsCommandBuffers.size(); ++i ) {
   vkBeginCommandBuffer( Vulkan.GraphicsCommandBuffers[i], &graphics_commandd_buffer_begin_info );
```

```
  if( GetPresentQueue().Handle != GetGraphicsQueue().Handle ) {
    VkImageMemoryBarrier barrier_from_present_to_draw = {
      VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,         // VkStructureType                sType
      nullptr,                                        // const void                    *pNext
      VK_ACCESS_MEMORY_READ_BIT,                      // VkAccessFlags                  srcAccessMask
      VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           // VkAccessFlags                  dstAccessMask
      VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                // VkImageLayout                  oldLayout
      VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                // VkImageLayout                  newLayout
      GetPresentQueue().FamilyIndex,                  // uint32_t                       srcQueueFamilyIndex
      GetGraphicsQueue().FamilyIndex,                 // uint32_t                       dstQueueFamilyIndex
      swap_chain_images[i],                           // VkImage                        image
      image_subresource_range                         // VkImageSubresourceRange        subresourceRange
    };
    vkCmdPipelineBarrier( Vulkan.GraphicsCommandBuffers[i], VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, 0, 0, nullptr, 0, nullptr, 1, &barrier_from_present_to_draw );
  }

  VkRenderPassBeginInfo render_pass_begin_info = {
    VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,       // VkStructureType                sType
    nullptr,                                        // const void                    *pNext
    Vulkan.RenderPass,                              // VkRenderPass                   renderPass
    Vulkan.FramebufferObjects[i].Handle,            // VkFramebuffer                  framebuffer
    {                                               // VkRect2D                       renderArea
      {                                             // VkOffset2D                     offset
        0,                                          // int32_t                         x
        0                                           // int32_t                         y
```

```
      },
      {                                                   // VkExtent2D                   extent
        300,                                              // int32_t                        width
        300,                                              // int32_t                        height
      }
    },
    1,                                                    // uint32_t                     clearValueCount
    &clear_value                                          // const VkClearValue           *pClearValues
};

vkCmdBeginRenderPass( Vulkan.GraphicsCommandBuffers[i], &render_pass_begin_info, VK_SUBPASS_CONTENTS_INLINE );

vkCmdBindPipeline( Vulkan.GraphicsCommandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, Vulkan.GraphicsPipeline );

vkCmdDraw( Vulkan.GraphicsCommandBuffers[i], 3, 1, 0, 0 );

vkCmdEndRenderPass( Vulkan.GraphicsCommandBuffers[i] );

if( GetGraphicsQueue().Handle != GetPresentQueue().Handle ) {
  VkImageMemoryBarrier barrier_from_draw_to_present = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,           // VkStructureType              sType
    nullptr,                                          // const void                   *pNext
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,             // VkAccessFlags                srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,                        // VkAccessFlags                dstAccessMask
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                  // VkImageLayout                oldLayout
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                  // VkImageLayout                newLayout
```

```
      GetGraphicsQueue().FamilyIndex,                 // uint32_t                 srcQueueFamilyIndex
      GetPresentQueue( ).FamilyIndex,                 // uint32_t                 dstQueueFamilyIndex
      swap_chain_images[i],                           // VkImage                  image
      image_subresource_range                         // VkImageSubresourceRange  subresourceRange
    };
    vkCmdPipelineBarrier( Vulkan.GraphicsCommandBuffers[i], VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0, 0, nullptr, 0, nullptr, 1, &barrier_from_draw_to_present );
  }

  if( vkEndCommandBuffer( Vulkan.GraphicsCommandBuffers[i] ) != VK_SUCCESS ) {
    printf( "Could not record command buffer!\n" );
    return false;
  }
}
return true;
```

23.Tutorial03.cpp，函数 RecordCommandBuffers()

通过调用 vkBeginCommandBuffer() 函数开始记录命令缓冲区。 开始时设置一个壁垒，告知驱动程序之前某个家族的队列引用了特定图像，但现在不同家族的队列将引用该图像（这么做的原因是在交换链创建期间，我们指定了专用共享模式）。 该壁垒仅在显卡队列不同于演示队列时使用。 该步骤可通过调用 vkCmdPipelineBarrier() 函数进行。 我们必须指定何时将壁垒放在管道中 (VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT)，以及如何设置该壁垒。 通过 VkImageMemoryBarrier 结构准备壁垒参数：

sType – 结构类型，此处设置为 VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER。
pNext – 为扩展功能预留的指示器。
srcAccessMask – 壁垒前执行的与特定图像有关的内存操作类型。
dstAccessMask – 连接至特定图像，在壁垒之后执行的内存操作类型。

oldLayout – 当前图像内存布局。
newLayout – 壁垒之后应该拥有的内存布局图像。
srcQueueFamilyIndex – 在壁垒之前引用图像的队列家族索引。
dstQueueFamilyIndex – 在壁垒之后将通过其引用图像的队列家族索引。
image – 图像本身的句柄。
subresourceRange – 我们希望进行过渡的图像部分。
在本示例中，我们不更改图像的布局，原因有两点： (1) 壁垒可以不用设置（如果图形和演示队列相同），(2) 布局过渡将作为渲染通道操作自动进行（在第一个 — 唯一 — 子通道开头）。

接下来启动渲染通道。 我们调用 vkCmdBeginRenderPass() 函数，而且必须为该函数提供 VkRenderPassBeginInfo 类型变量指示器。 它包含以下成员：

sType – 标准结构类型。 在本示例中，必须将其设置为 VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO 的值。
pNext – 留作将来使用的指示器。
renderPass – 我们希望启动的渲染通道的句柄。
framebuffer – 帧缓冲器的句柄，指定在该渲染通道中用作附件的图像。
renderArea – 受渲染通道内执行的操作所影响的图形区域。 它指定左上角（通过抵消成员 (offset member) 的 x 和 y 参数），以及渲染区域的宽度和高度（通过扩展成员 (extent member)）。
clearValueCount – pClearValues 阵列中的要素数量。
pClearValues – 有关各附件的清空值阵列。
指定渲染通道的渲染区域时，必须确保渲染操作不会修改该区域外的像素。 这只是给驱动程序的提示，但可以优化其行为。 如果不使用相应的 scissor 测试将操作限制在提供的区域内，该区域外的像素可能变成未定义像素（不能依靠内容）。 而且，我们不能指定大于帧缓冲区尺寸的渲染区域（超出帧缓冲器）。

就 pClearValues 而言，它必须包含各渲染通道附件的要素。 loadOp 设置为清空时，每个要素均指定特定附件必须清空的颜色。 对于 loadOp 不是清空的附件来说，将忽略提供给它们的值。 但不能为阵列提供数量较少的要素。

我们已经开始创建命令缓冲区、设置壁垒（如有必要），并启动渲染通道。 启动渲染通道时，我们还将启动其第一个子通道。 我们可以通过调用 vkCmdNextSubpass() 函数切换至下一个子通道。 执行这些操作期间，可能会出现布局过渡和清空操作。 清空操作在首先使用（引用）图像的子通道内进行。 如果子通道布局与之前的通道或（如果是第一个子通道或第一个引用图像时）初始布局（渲染通道之前的布局）不同，将出现布局过渡。 因此在本示例中，启动渲染通道时，交换链图像的布局将从“presentation source”布局自动变成 “color attachment optimal”布局。

现在我们绑定图形管道。 该步骤可通过调用 vkCmdBindPipeline() 函数完成。 这样可“激活”所有着色器程序（类似于 glUseProgram() 函数，并设置必要的测试、混合操作等。

绑定管道后，我们可以通过调用 vkCmdDraw() 函数，进行最终的绘制操作。 在本函数中，我们指定希望绘制的顶点数量（3 个）、应绘制的实例数量（仅 1 个），以及第一个顶点和第一个实例的索引编号（均为 0）。

接下来调用 vkCmdEndRenderPass() 函数，结束特定的渲染通道。 这里，如果为渲染通道指定的最终布局与引用特定图像的最后一个子通道所使用的布局不同，所有最终布局都将进行过渡。

之后将设置壁垒，其中我们告知驱动程序，显卡队列已使用完特定图像，而且从现在开始演示队列将使用该图像。 仅在显卡队列和演示队列不同的情况下，再次执行该步骤。 在壁垒之后，我们停止为特定图像记录命令缓冲区。 所有这些操作都会为每个交换链图像重复一次。

绘制

绘制函数与教程 2 中的 Draw() 函数相同。 我们获取图形索引、提交相应的命令缓冲区，并演示图像。 使用旗语的方式与之前的相同：一个旗语用于获取图像，并告知显卡队列等待可用的图像。 第二个命令缓冲区用于指示显卡队列上的绘制操作是否已经完成。 演示图像之前，演示队列需等待该旗语。 以下是 Draw() 函数的源代码：

Copy CodeDarkLight

```
VkSemaphore image_available_semaphore = GetImageAvailableSemaphore();
VkSemaphore rendering_finished_semaphore = GetRenderingFinishedSemaphore();
```

```
VkSwapchainKHR swap_chain = GetSwapChain().Handle;
uint32_t image_index;

VkResult result = vkAcquireNextImageKHR( GetDevice(), swap_chain, UINT64_MAX, image_available_semaphore, VK_NULL_HANDLE, &image_index );
switch( result ) {
   case VK_SUCCESS:
   case VK_SUBOPTIMAL_KHR:
      break;
   case VK_ERROR_OUT_OF_DATE_KHR:
      return OnWindowSizeChanged();
   default:
      printf( "Problem occurred during swap chain image acquisition!\n" );
      return false;
}

VkPipelineStageFlags wait_dst_stage_mask = VK_PIPELINE_STAGE_TRANSFER_BIT;
VkSubmitInfo submit_info = {
   VK_STRUCTURE_TYPE_SUBMIT_INFO,                       // VkStructureType                sType
   nullptr,                                             // const void                   *pNext
   1,                                                   // uint32_t                       waitSemaphoreCount
   &image_available_semaphore,                          // const VkSemaphore            *pWaitSemaphores
   &wait_dst_stage_mask,                                // const VkPipelineStageFlags   *pWaitDstStageMask;
   1,                                                   // uint32_t                       commandBufferCount
   &Vulkan.GraphicsCommandBuffers[image_index],         // const VkCommandBuffer        *pCommandBuffers
   1,                                                   // uint32_t                       signalSemaphoreCount
   &rendering_finished_semaphore                        // const VkSemaphore            *pSignalSemaphores
```

```
};

if( vkQueueSubmit( GetGraphicsQueue().Handle, 1, &submit_info, VK_NULL_HANDLE ) != VK_SUCCESS ) {
   return false;
}

VkPresentInfoKHR present_info = {
   VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,                // VkStructureType              sType
   nullptr,                                           // const void                  *pNext
   1,                                                 // uint32_t                     waitSemaphoreCount
   &rendering_finished_semaphore,                     // const VkSemaphore           *pWaitSemaphores
   1,                                                 // uint32_t                     swapchainCount
   &swap_chain,                                       // const VkSwapchainKHR        *pSwapchains
   &image_index,                                      // const uint32_t              *pImageIndices
   nullptr                                            // VkResult                    *pResults
};
result = vkQueuePresentKHR( GetPresentQueue().Handle, &present_info );

switch( result ) {
   case VK_SUCCESS:
      break;
   case VK_ERROR_OUT_OF_DATE_KHR:
   case VK_SUBOPTIMAL_KHR:
      return OnWindowSizeChanged();
   default:
      printf( "Problem occurred during image presentation!\n" );
```

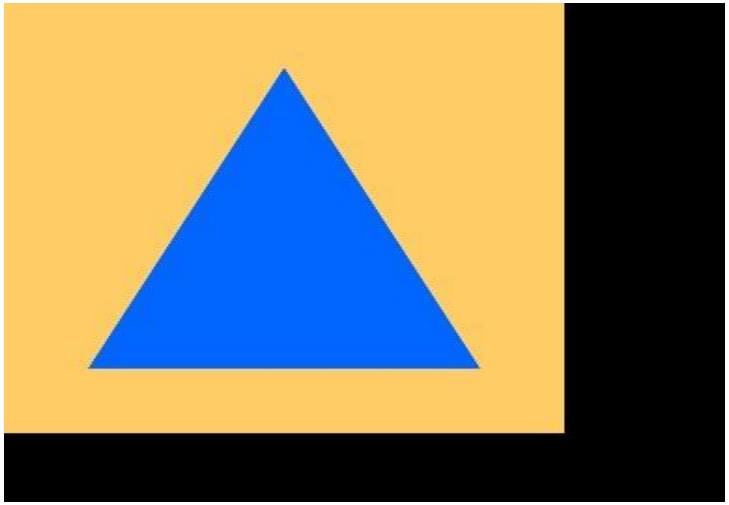
```
    return false;
}

return true;
```

24.Tutorial03.cpp，函数 Draw()

教程 3 执行

在本教程中我们执行了“真正的”绘制操作。 简单的三角形似乎没有太大的说服力，但对于 Vulkan 创建的第一个图像来说，它是一个良好的开端。 该三角形如下所示：



如果您想知道图像中为什么会出现黑色部分，原因如下： 为简化整个代码，我们创建了大小固定（宽度和高度均为 300 像素）的帧缓冲区。 但窗口尺寸（和交换链图像的尺寸）可能大于 300 x 300 像素。 超出帧缓冲区尺寸的图像部分没有被应用清空和修改。 这可能包含部分“人为因素”，因为供驱

动程序分配交换链图像的内存之前可能用于其他目的，可能包含一些数据。 正确的行为是创建大小与交换链图像相同的帧缓冲区，并在窗口尺寸大小变化后重新创建。 但如果在橙色/金色背景上渲染蓝色三角形，表示该代码可正常运行。

清空

本教程结束之前，我们最后需要学习的是如何使用在本教程中创建的资源。 释放资源所需的代码已在之前章节中创建，这里不再叙述。 只需查看 VulkanCommon.cpp 文件。 以下代码可毁坏特定于本章节的资源：

```
if( GetDevice() != VK_NULL_HANDLE ) {
   vkDeviceWaitIdle( GetDevice() );

   if( (Vulkan.GraphicsCommandBuffers.size() > 0) && (Vulkan.GraphicsCommandBuffers[0] != VK_NULL_HANDLE) ) {
      vkFreeCommandBuffers( GetDevice(), Vulkan.GraphicsCommandPool, static_cast<uint32_t>(Vulkan.GraphicsCommandBuffers.size()),
&Vulkan.GraphicsCommandBuffers[0] );
      Vulkan.GraphicsCommandBuffers.clear();
   }

   if( Vulkan.GraphicsCommandPool != VK_NULL_HANDLE ) {
      vkDestroyCommandPool( GetDevice(), Vulkan.GraphicsCommandPool, nullptr );
      Vulkan.GraphicsCommandPool = VK_NULL_HANDLE;
   }

   if( Vulkan.GraphicsPipeline != VK_NULL_HANDLE ) {
      vkDestroyPipeline( GetDevice(), Vulkan.GraphicsPipeline, nullptr );
      Vulkan.GraphicsPipeline = VK_NULL_HANDLE;
   }
```

```
  if( Vulkan.RenderPass != VK_NULL_HANDLE ) {
    vkDestroyRenderPass( GetDevice(), Vulkan.RenderPass, nullptr );
    Vulkan.RenderPass = VK_NULL_HANDLE;
  }

  for( size_t i = 0; i < Vulkan.FramebufferObjects.size(); ++i ) {
    if( Vulkan.FramebufferObjects[i].Handle != VK_NULL_HANDLE ) {
      vkDestroyFramebuffer( GetDevice(), Vulkan.FramebufferObjects[i].Handle, nullptr );
      Vulkan.FramebufferObjects[i].Handle = VK_NULL_HANDLE;
    }

    if( Vulkan.FramebufferObjects[i].ImageView != VK_NULL_HANDLE ) {
      vkDestroyImageView( GetDevice(), Vulkan.FramebufferObjects[i].ImageView, nullptr );
      Vulkan.FramebufferObjects[i].ImageView = VK_NULL_HANDLE;
    }
  }
  Vulkan.FramebufferObjects.clear();
}
```

25.Tutorial03.cpp，函数 ChildClear()

像往常一样，首先检查是否有设备。 如果没有设备，就没有资源。 接下来等待设备空闲下来，并删除所有已创建的资源。 我们首先通过调用 vkFreeCommandBuffers() 函数，开始删除命令缓冲区。 接下来通过 vkDestroyCommandPool() 函数毁坏命令池，然后破坏图形管道。 该步骤可通过 vkDestroyPipeline() 函数完成。然后调用 vkDestroyRenderPass() 函数，以释放渲染通道的句柄。最后删除与交换链图像相关的所有帧缓冲区和图像视图。

破坏对象之前，首先检查是否创建了特定资源。 如果没有，我们则跳过破环资源这一流程。

结论

在本教程中，我们创建了包含一个子通道的渲染通道。 接下来创建交换链图像图像视图和帧缓冲区。 其中一个最重要的部分是创建图形管道，因为这一过程要求我们准备大量数据。 我们需要创建着色器模块，并描述应该绑定图形管道时处于活跃状态的着色器阶段。 需要准备与输入顶点、布局，以及将其汇编成拓扑等相关的信息。 还需要准备视口、光栅化、多点采样和颜色混合信息。 然后创建简单的管道布局，之后才能创建管道。 接下来我们创建了命令池，并为各交换链图像分配了命令缓冲区。 记录在命令缓冲区中的操作涉及设置图像内存壁垒、启动渲染通道、绑定图形通道，以及绘制。 接下来结束渲染通道，并设置另一图像内存壁垒。 执行绘制的方式与之前教程 (2) 中所述的相同。

在接下来的教程中，我们将学习顶点属性、图像和缓冲区等相关知识。

教程 4： 顶点属性 – 缓冲区、图像和栅栏

指定渲染通道相关性
图形管道创建
编写着色器
顶点属性指定
输入汇编状态指定
视口状态指定
动态指定
管道对象创建
顶点缓冲区创建
缓冲区内存分配
绑定缓冲区内存
上传顶点数据

渲染资源创建
命令池创建
命令缓冲区分配
旗语创建
栅栏创建
绘制
记录命令缓冲区
教程 04 执行
清空
结论

教程 4： 顶点属性 – 缓冲区、图像和栅栏

我们在之前的教程中学些了一些基本知识。 教程篇幅比较长而且（我希望）介绍得比较详细。 这是因为 Vulkan* API 的学习曲线非常陡峭。 而且大家看，即使是准备最简单的应用，我们也需要了解大量的知识。

但现在我们已经打好了基础。 因此本教程篇幅会短一些，并重点介绍与 Vulkan API 相关的一些话题。 本节我们将通过从顶点缓冲区提供顶点属性，介绍绘制任意几何图形的推荐方法。 由于本课程的代码与“03 – 第一个三角形”教程的代码类似，因此我仅介绍与之不同的部分。

并介绍另外一种整理渲染代码的方法。 之前我们在主渲染循环前记录了命令缓冲区。 但在实际情况中，每帧动画都各不相同，因此我们无法预先记录所有渲染命令。 我们应该尽可能地晚一些记录和提交命令缓冲区，以最大限度地降低输入延迟，并获取最新的输入数据。 我们将正好在提交至队列之前记录命令缓冲区。 不过，一个命令缓冲区远远不够。 必须等到显卡处理完提交的命令缓冲区后，才记录相同的命令缓冲区。 此时将通过栅栏发出信号。 但每一帧都等待栅栏实在非常浪费时间，因此我们需要更多的命令缓冲区，以便交换使用。 命令缓冲区越多，所需的栅栏越多，情况也将越复杂。本教程将展示如何组织代码，尽可能地实现代码的轻松维护性、灵活性和快速性。

指定渲染通道相关性

我们从创建渲染通道开始，方法与之前教程中所介绍的相同。 但这次我们还需要提供其他信息。 渲染通道描述渲染资源（图像/附件）的内部组织形式，使用方法，以及在渲染流程中的变化。 通过创建图像内存壁垒，可明确更改图像的布局。 如果指定了合适的渲染通道描述（初始布局、子通道布局和最终图像布局），这种操作也可以隐式地进行。 我们首选隐式过渡，因为驱动程序可以更好地执行此类过渡。

在本教程的这一部分，与之前相同，我们将初始和最终图像布局指定为“transfer src”，而将渲染通道指定为“color attachment optimal”子通道布局。 但之前的教程缺乏其他重要信息，尤其是如何使用图像（即执行哪些与图像相关的操作），以及何时使用图像（渲染管道的哪些部分使用图像）。 此类信息可在图像内存壁垒和渲染通道描述中指定。 创建图像内存壁垒时，我们指定与特定图像相关的操作类型（壁垒之前和之后的内存访问类型），而且我们还指定何时放置壁垒（壁垒之前和之后使用图像的管道阶段）。

创建渲染通道并为其提供描述时，通过子通道相关性指定相同的信息。 其他数据对驱动程序也至关重要，可更好地准备隐式壁垒。 以下源代码可创建渲染通道并准备子通道相关性。

```
std::vector<VkSubpassDependency> dependencies = {
  {
    VK_SUBPASS_EXTERNAL,                                    // uint32_t                srcSubpass
    0,                                                      // uint32_t                dstSubpass
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,                   // VkPipelineStageFlags    srcStageMask
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,    // VkPipelineStageFlags    dstStageMask
    VK_ACCESS_MEMORY_READ_BIT,                              // VkAccessFlags           srcAccessMask
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,                   // VkAccessFlags           dstAccessMask
    VK_DEPENDENCY_BY_REGION_BIT                             // VkDependencyFlags       dependencyFlags
  },
  {
    0,                                                      // uint32_t                srcSubpass
    VK_SUBPASS_EXTERNAL,                                    // uint32_t                dstSubpass
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,    // VkPipelineStageFlags    srcStageMask
```

```
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,          // VkPipelineStageFlags          dstStageMask
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,          // VkAccessFlags                 srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,                     // VkAccessFlags                 dstAccessMask
    VK_DEPENDENCY_BY_REGION_BIT                    // VkDependencyFlags             dependencyFlags
  }
};

VkRenderPassCreateInfo render_pass_create_info = {
  VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO,       // VkStructureType               sType
  nullptr,                                         // const void                    *pNext
  0,                                               // VkRenderPassCreateFlags       flags
  1,                                               // uint32_t                      attachmentCount
  attachment_descriptions,                         // const VkAttachmentDescription *pAttachments
  1,                                               // uint32_t                      subpassCount
  subpass_descriptions,                            // const VkSubpassDescription    *pSubpasses
  static_cast<uint32_t>(dependencies.size()),      // uint32_t                      dependencyCount
  &dependencies[0]                                 // const VkSubpassDependency     *pDependencies
};

if( vkCreateRenderPass( GetDevice(), &render_pass_create_info, nullptr, &Vulkan.RenderPass ) != VK_SUCCESS ) {
  std::cout << "Could not create render pass!" << std::endl;
  return false;
}
```

1.Tutorial04.cpp，函数 CreateRenderPass()

子通道相关性描述不同子通道之间的相关性。 当附件以特定方式用于特定子通道（例如渲染）时，会以另一种方式用于另一个子通道（取样），因此我

们可创建内存壁垒或提供子通道相关性，以描述附件在这两个子通道中的预期用法。 当然，我们推荐使用后一种选项，因为驱动程序能够（通常）以最佳的方式准备壁垒。 而且代码本身也会得到完善 — 了解代码所需的一切都收集在一个位置，一个对象中。

在我们的简单示例中，我们仅有一个子通道，但我们指定了两种相关性。 因为我们能够（而且应该）指定渲染通道（通过提供特定子通道的编号）与其外侧操作（通过提供 VK_SUBPASS_EXTERNAL 值）之间的相关性。这里我们为渲染通道与其唯一的子通道之前执行的操作之间的颜色附件提供相关性。 第二种相关性针对子通道内和渲染通道之后执行的操作定义。

我们将讨论哪些操作。 我们仅使用一个附件，即从演示引擎（交换链）获取的图像。 演示引擎将图像用作可演示数据源。 它仅显示一个图像。 因此涉及图像的唯一操作是在“present src”布局的图像上进行“内存读取”。 该操作不会在任何正常的管道阶段中进行，而在“管道底部”阶段再现。

在渲染通道内部唯一的子通道（索引为 0）中，我们将渲染用作颜色附件的图像。 因此在该图像上进行的操作为“颜色附件写入”，在“颜色附件输入”管道阶段中（碎片着色器之后）执行。 演示并将图像返回至演示引擎后，会再次将该图像用作数据源。 因此在本示例中，渲染通道之后的操作与之前的一样： “内存读取”。

我们通过 VkSubpassDependency 成员阵列指定该数据。 创建渲染通道和 VkRenderPassCreateInfo 结构时，我们（通过 dependencyCount 成员）指定相关性阵列中的要素数量，并（通过 pDependencies）提供第一个要素的地址。 在之前的教程中，我们将这两个字段设置为 0 和 nullptr。 VkSubpassDependency 结构包含以下字段：

srcSubpass – 第一个（前面）子通道的索引或 VK_SUBPASS_EXTERNAL（如果希望指示子通道与渲染通道外的操作之间的相关性）。
dstSubpass – 第二个（后面）子通道的索引（或 VK_SUBPASS_EXTERNAL）。
srcStageMask – 之前（src 子通道中）使用特定附件的管道阶段。
dstStageMask – 之后（dst 子通道中）将使用特定附件的管道阶段。
srcAccessMask – src 子通道中或渲染通道前所发生的内存操作类型。
dstAccessMask – dst 子通道中或渲染通道后所发生的内存操作类型。
dependencyFlags – 描述相关性类型（区域）的标记。
图形管道创建

现在我们将创建图形管道对象。 （我们应创建面向交换链图像的帧缓冲器，不过这一步骤在命令缓冲区记录期间进行）。 我们不想渲染在着色器中进行过硬编码的几何图形， 而是想绘制任意数量的顶点，并提供其他属性（不仅仅是顶点位置）。 我们首先应该做什么？

编写着色器

首先查看用 GLSL 代码编写的顶点着色器：

```
#version 450

layout(location = 0) in vec4 i_Position;
layout(location = 1) in vec4 i_Color;

out gl_PerVertex
{
   vec4 gl_Position;
};

layout(location = 0) out vec4 v_Color;

void main() {
      gl_Position = i_Position;
      v_Color = i_Color;
}
```

2.shader.vert

尽管比教程 03 的复杂，但该着色器非常简单。

我们指定两个输入属性（named i_Position 和 i_Color）。 在 Vulkan 中，所有属性必须有一个位置布局限定符。 在 Vulkan API 中指定顶点属性描述时，属性名称不重要，重要的是它们的索引/位置。 在 OpenGL* 中，我们可请求特定名称的属性位置。 在 Vulkan 中不能这样做。 位置布局限定符是唯一的方法。

接下来我们重新声明着色器中的 gl_PerVertex 模块。 Vulkan 使用着色器 I/O 模块，所以我们应该重新声明 gl_PerVertex 以明确指定该模块使用哪些成员。如果没有指定，将使用默认定义。但我们必须记住该默认定义 contains gl_ClipDistance[]，它要求我们启用特性 shaderClipDistance（而且在 Vulkan 中，不能使用创建设备期间没有启用的特性，或可能无法正常运行的应用）。 这里我们仅使用 gl_Position 成员，因此不要求启用该特性。

然后我们指定一个与变量 v_Color 不同的附加输入，以保存顶点颜色。 在主函数中，我们将应用提供的值拷贝至相应的输入变量：position to gl_Position 和 color to v_Color。

现在查看碎片着色器，以了解如何使用属性。

```
#version 450

layout(location = 0) in vec4 v_Color;

layout(location = 0) out vec4 o_Color;

void main() {
   o_Color = v_Color;
}
```

3.shader.frag

在碎片着色器中，将与变量 v_Color 不同的输入仅拷贝至输出变量 o_Color。 两个变量都有位置布局说明符。 在顶点着色器中变量 v_Color 的位置与输出变量相同，因此它将包含定点之间插值替换的颜色值。

着色器能够以与之前相同的方式转换成 SPIR-V 汇编。 这一步骤可通过以下命令完成：

```
glslangValidator.exe -V -H shader.vert > vert.spv.txt
```

```
glslangValidator.exe -V -H shader.frag > frag.spv.txt
```

因此现在，了解哪些是我们希望在着色器中使用的属性后，我们将可以创建相应的图形管道。

顶点属性指定

在本教程中，我们将对顶点输入状态创建进行最重要的改进，为此我们指定类型变量 VkPipelineVertexInputStateCreateInfo。 在该变量中我们提供结构指示器，定义顶点输入数据类型，以及属性的数量和布局。

我们希望使用两个属性：顶点位置和顶点颜色，前者由四个浮点组件组成，后者由四个浮点值组成。 我们以交错属性布局的形式将所有顶点数据放在缓冲器中。 这表示我们将依次放置第一个顶点的位置，相同顶点的颜色，第二个顶点的位置，第二个顶点的颜色，第三个顶点的位置和颜色，依此类推。我们借助以下代码完成这种指定：

```
std::vector<VkVertexInputBindingDescription> vertex_binding_descriptions = {
  {
    0,                                                  // uint32_t                 binding
    sizeof(VertexData),                                 // uint32_t                 stride
    VK_VERTEX_INPUT_RATE_VERTEX                         // VkVertexInputRate        inputRate
  }
```

```
};

std::vector<VkVertexInputAttributeDescription> vertex_attribute_descriptions = {
    {
        0,                                                  // uint32_t                                 location
        vertex_binding_descriptions[0].binding,             // uint32_t                                 binding
        VK_FORMAT_R32G32B32A32_SFLOAT,                      // VkFormat                                 format
        offsetof(struct VertexData, x)                      // uint32_t                                 offset
    },
    {
        1,                                                  // uint32_t                                 location
        vertex_binding_descriptions[0].binding,             // uint32_t                                 binding
        VK_FORMAT_R32G32B32A32_SFLOAT,                      // VkFormat                                 format
        offsetof( struct VertexData, r )                    // uint32_t                                 offset
    }
};

VkPipelineVertexInputStateCreateInfo vertex_input_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,      // VkStructureType                          sType
    nullptr,                                                        // const void                               *pNext
    0,                                                              // VkPipelineVertexInputStateCreateFlags    flags;
    static_cast<uint32_t>(vertex_binding_descriptions.size()),      // uint32_t                                 vertexBindingDescriptionCount
    &vertex_binding_descriptions[0],                                // const VkVertexInputBindingDescription    *pVertexBindingDescriptions
    static_cast<uint32_t>(vertex_attribute_descriptions.size()),    // uint32_t                                 vertexAttributeDescriptionCount
    &vertex_attribute_descriptions[0]                               // const VkVertexInputAttributeDescription  *pVertexAttributeDescriptions
};
```

4.Tutorial04.cpp，函数 CreatePipeline()

首先通过 VkVertexInputBindingDescription 指定顶点数据绑定（通用内存信息）。 它包含以下字段：

binding – 与顶点数据相关的绑定索引。
stride – 两个连续要素（两个相邻顶点的相同属性）之间的间隔（字节）。
inputRate – 定义如何使用数据，是按照顶点还是按照实例使用。
步长和 inputRate 字段不言而喻。 绑定成员可能还要求提供其他信息。 创建顶点缓冲区时，在执行渲染操作之前我们将其绑定至所选的插槽。 插槽编号（索引）就是这种绑定，此处我们描述该插槽中的数据如何与内存对齐，以及如何使用数据（按顶点或实例）。 不同的顶点缓冲区可绑定至不同的绑定。 而且每个绑定都可放在内存中的不同位置。

接下来定义所有顶点属性。 我们必须指定各属性的位置（索引）（与着色器源代码相同，以位置布局限定符的形式）、数据源（从哪个绑定读取数据）、格式（数据类型和组件数量），以及查找特定属性数据的偏移（从特定顶点数据的开头，而非所有顶点数据的开头）。 这种情况与 OpenGL 几乎相同，我们创建顶点缓冲区对象（VBO，可视作等同于“绑定”），并使用 glVertexAttribPointer() 函数（通过该函数指定属性索引（位置）、大小和类型（组件数量和格式）、步长和偏移）定义属性。 可通过 VkVertexInputAttributeDescription 结构提供这类信息。 它包含以下字段：

location – 属性索引，与着色器源代码中由位置布局说明符定义的相同。
binding – 供数据读取的插槽编号（与 OpenGL 中的 VBO 等数据源），与 VkVertexInputBindingDescription 结构和 vkCmdBindVertexBuffers() 函数（稍后介绍）中的绑定相同。
format – 数据类型和每个属性的组件数量。
offset – 特定属性数据的开头。
准备好后，我们可以通过填充类型变量 VkPipelineVertexInputStateCreateInfo 准备顶点输入状态描述，该变量包含以下字段：

sType – 结构类型，此处应等于 VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。 目前将该数值设为 null。
flags – 留作将来使用的参数。

vertexBindingDescriptionCount – pVertexBindingDescriptions 阵列中的要素数量。
pVertexBindingDescriptions – 描述为特定管道（支持读取所有属性的缓冲区）定义的所有绑定的阵列。
vertexAttributeDescriptionCount – pVertexAttributeDescriptions 阵列中的要素数量。
pVertexAttributeDescriptions – 指定所有顶点属性的要素阵列。
它包含创建管道期间的顶点属性指定。 如要使用它们，我们必须创建顶点缓冲区，并在发布渲染命令之前将其绑定至命令缓冲区。

输入汇编状态指定

之前我们使用三角形条拓扑绘制了一个简单的三角形。 现在我们绘制一个四边形，通过定义四个顶点（而非两个三角形和六个顶点）绘制起来非常方便。
为此我们必须使用三角形条带拓扑。 我们通过 VkPipelineInputAssemblyStateCreateInfo 结构定义该拓扑，其中包含以下成员：

sType – 结构类型，此处等于 VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO。
pNext – 为扩展功能预留的指示器。
flags – 留作将来使用的参数。
topology – 用于绘制顶点的拓扑（比如三角扇、带、条）。
primitiveRestartEnable – 该参数定义是否希望使用特定顶点索引值重新开始汇编基元。
以下简单代码可用于定义三角条带拓扑：

```
VkPipelineInputAssemblyStateCreateInfo input_assembly_state_create_info = {
   VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,    // VkStructureType                              sType
   nullptr,                                                        // const void                                   *pNext
   0,                                                              // VkPipelineInputAssemblyStateCreateFlags      flags
   VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,                           // VkPrimitiveTopology                          topology
   VK_FALSE                                                        // VkBool32                                     primitiveRestartEnable
};
```

5.Tutorial04.cpp，函数 CreatePipeline()

视口状态指定

本教程引进了另外一个变化。 之前为了简单起见，我们对视口和 scissor 测试参数进行了硬编码，可惜导致图像总保持相同的大小，无论应用窗口多大。 这次我们不通过 VkPipelineViewportStateCreateInfo 结构指定这些数值， 而是使用动态。 以下代码负责定义静态视口状态参数：

```
VkPipelineViewportStateCreateInfo viewport_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO,      // VkStructureType                          sType
    nullptr,                                                    // const void                               *pNext
    0,                                                          // VkPipelineViewportStateCreateFlags       flags
    1,                                                          // uint32_t                                 viewportCount
    nullptr,                                                    // const VkViewport                         *pViewports
    1,                                                          // uint32_t                                 scissorCount
    nullptr                                                     // const VkRect2D                           *pScissors
};
```

6.Tutorial04.cpp，函数 CreatePipeline()

定义静态视口参数的结构包含以下成员：

sType – 结构类型，此处为 VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO。
pNext – 为特定于扩展的参数预留的指示器。
flags – 留作将来使用的参数。
viewportCount – 视口数量。
pViewports – 定义静态视口参数的结构指示器。
scissorCount – scissor 矩形的数量（数值必须与 viewportCount 参数相同）。
pScissors – 定义视口的静态 scissor 测试参数的 2D 矩形阵列指示器。

如果希望通过动态定义视口和 scissor 参数，则无需填充 pViewports 和 pScissors 成员。 因此在上述示例中将其设置为 null。 但我们必须定义视口和 scissor 测试矩形的数量。 通常通过 VkPipelineViewportStateCreateInfo 结构指定这些值，无论是否希望使用动态或静态视口和 scissor 状态。

动态指定

创建管道时，我们可以指定哪部分始终保持静态 — 在管道创建期间通过结构定义，哪部分保持动态 — 在命令缓冲区记录期间通过调用相应的函数来指定。 这可帮助我们减少仅在细节方面有所差异（比如线条宽度、混合常量、模板参数或之前提到的视口大小）的管道对象的数量。 以下代码用于定义管道应保持动态的部分：

```
std::vector<VkDynamicState> dynamic_states = {
   VK_DYNAMIC_STATE_VIEWPORT,
   VK_DYNAMIC_STATE_SCISSOR,
};

VkPipelineDynamicStateCreateInfo dynamic_state_create_info = {
   VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO,              // VkStructureType                              sType
   nullptr,                                                           // const void                                  *pNext
   0,                                                                 // VkPipelineDynamicStateCreateFlags            flags
   static_cast<uint32_t>(dynamic_states.size()),                      // uint32_t                                    dynamicStateCount
   &dynamic_states[0]                                                 // const VkDynamicState                        *pDynamicStates
};
```

7.Tutorial04.cpp，函数 CreatePipeline()

该步骤通过类型结构 VkPipelineDynamicStateCreateInfo 完成，其中包含以下字段：

sType – 定义特定结构类型的参数，此处等于 VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO。

pNext – 为扩展功能预留的参数。
flags – 留作将来使用的参数。
dynamicStateCount – pDynamicStates 阵列中的要素数量。
pDynamicStates – 包含 enum 的阵列，指定将哪部分管道标记为动态。 该阵列的要素类型为 VkDynamicState。
管道对象创建

定义完图形管道的所有必要参数后，将可以开始创建管道对象。 以下代码可帮助完成这一操作：

```
VkGraphicsPipelineCreateInfo pipeline_create_info = {
  VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,                  // VkStructureType                                  sType
  nullptr,                                                          // const void                                       *pNext
  0,                                                                // VkPipelineCreateFlags                            flags
  static_cast<uint32_t>(shader_stage_create_infos.size()),          // uint32_t                                         stageCount
  &shader_stage_create_infos[0],                                    // const VkPipelineShaderStageCreateInfo            *pStages
  &vertex_input_state_create_info,                                  // const VkPipelineVertexInputStateCreateInfo       *pVertexInputState;
  &input_assembly_state_create_info,                                // const VkPipelineInputAssemblyStateCreateInfo     *pInputAssemblyState
  nullptr,                                                          // const VkPipelineTessellationStateCreateInfo      *pTessellationState
  &viewport_state_create_info,                                      // const VkPipelineViewportStateCreateInfo          *pViewportState
  &rasterization_state_create_info,                                 // const VkPipelineRasterizationStateCreateInfo     *pRasterizationState
  &multisample_state_create_info,                                   // const VkPipelineMultisampleStateCreateInfo       *pMultisampleState
  nullptr,                                                          // const VkPipelineDepthStencilStateCreateInfo      *pDepthStencilState
  &color_blend_state_create_info,                                   // const VkPipelineColorBlendStateCreateInfo        *pColorBlendState
  &dynamic_state_create_info,                                       // const VkPipelineDynamicStateCreateInfo           *pDynamicState
  pipeline_layout.Get(),                                            // VkPipelineLayout                                 layout
  Vulkan.RenderPass,                                                // VkRenderPass                                     renderPass
  0,                                                                // uint32_t                                         subpass
```

```
    VK_NULL_HANDLE,                                 // VkPipeline                   basePipelineHandle
    -1                                              // int32_t                      basePipelineIndex
};

if( vkCreateGraphicsPipelines( GetDevice(), VK_NULL_HANDLE, 1, &pipeline_create_info, nullptr, &Vulkan.GraphicsPipeline ) != VK_SUCCESS ) {
    std::cout << "Could not create graphics pipeline!" << std::endl;
    return false;
}
return true;
```

8.Tutorial04.cpp，函数 CreatePipeline()

最重要的变量（包含对所有管道参数的引用）的类型为 VkGraphicsPipelineCreateInfo。 与之前教程相比，唯一的变化是添加了 pDynamicState 参数，以指出 VkPipelineDynamicStateCreateInfo 结构的类型，如上所示。 每个指定为动态的管道状态均在命令缓冲区记录期间通过相应的函数调用进行设置。

通过调用 vkCreateGraphicsPipelines() 函数创建管道对象。

顶点缓冲区创建

如要使用顶点属性，除了在创建管道期间指定它们外，还需准备包含所有这些属性数据的缓冲区。 我们将从该缓冲区读取属性值并将其提供给顶点着色器。

在 Vulkan 中，缓冲区和图像创建包含至少两个阶段： 首先创建对象本身。 然后，我们需要创建内存对象，该对象之后将绑定至缓冲区（或图像）。 缓冲区将从该内存对象中提取存储空间。 这种方法有助于我们指定针对内存的其他参数，并通过更多细节对其进行控制。

我们调用 vkCreateBuffer() 创建（通用）缓冲区对象。 它从其他参数中接受类型变量 VkBufferCreateInfo 的指示器，以定义已创建缓冲区的参数。 以下代码负责创建用于顶点属性数据源的缓冲区：

```
VertexData vertex_data[] = {
    {
        -0.7f, -0.7f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 0.0f
    },
    {
        -0.7f, 0.7f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 0.0f
    },
    {
        0.7f, -0.7f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 0.0f
    },
    {
        0.7f, 0.7f, 0.0f, 1.0f,
        0.3f, 0.3f, 0.3f, 0.0f
    }
};

Vulkan.VertexBuffer.Size = sizeof(vertex_data);

VkBufferCreateInfo buffer_create_info = {
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO,                   // VkStructureType              sType
    nullptr,                                                // const void                   *pNext
    0,                                                      // VkBufferCreateFlags          flags
```

```
    Vulkan.VertexBuffer.Size,                                   // VkDeviceSize             size
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,                          // VkBufferUsageFlags       usage
    VK_SHARING_MODE_EXCLUSIVE,                                  // VkSharingMode            sharingMode
    0,                                                          // uint32_t                 queueFamilyIndexCount
    nullptr                                                     // const uint32_t           *pQueueFamilyIndices
};

if( vkCreateBuffer( GetDevice(), &buffer_create_info, nullptr, &Vulkan.VertexBuffer.Handle ) != VK_SUCCESS ) {
    std::cout << "Could not create a vertex buffer!" << std::endl;
    return false;
}
```

9.Tutorial04.cpp，函数 CreateVertexBuffer()

我们在 CreateVertexBuffer() 函数开头定义了大量用于位置和颜色属性的数值。 首先为第一个顶点定义四个位置组件，然后为相同的顶点定义四个颜色组件，之后为第二个顶点定义四个有关位置属性的组件，然后为相同顶点定义四个颜色值，之后依次为第三个和第四个顶点定义位置和颜色值。 阵列大小用于定义缓冲区大小。 但请记住，内部显卡驱动程序要求缓冲区的存储大于应用请求的大小。

接下来定义 VkBufferCreateInfo 类型变量。 该结构包含以下字段：

sType – 结构类型，应设置为 VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO 。
pNext – 为扩展功能预留的参数。
flags – 定义其他创建参数的参数。 此处它支持创建通过稀疏内存备份的缓冲区（类似于宏纹理）。 我们不想使用稀疏内存，因此将该参数设为 0。
size – 缓冲区的大小（字节）。
usage – 定义将来打算如何使用该缓冲区的参数。 我们可以指定为将该缓冲区用作统一缓冲区、索引缓冲区、传输（拷贝）操作数据源等。 这里我们打算将该缓冲区用作顶点缓冲区。 请记住，我们不能将该缓冲区用于缓冲器创建期间未定义的目的。
sharingMode – 共享模式，类似于交换链图像，定义特定缓冲区能否供多个队列同时访问（并发共享模式），还是仅供单个队列访问（专有共享模式）。 如

果指定为并发共享模式，那么必须提供所有将访问缓冲区的队列的索引。 如果希望定义为专有共享模式，我们仍然可以在不同队列中引用该缓冲区，但一次仅引用一个。 如果希望在不同的队列中使用缓冲区（提交将该缓冲区引用至另一队列的命令），我们需要指定缓冲区内存壁垒，以将缓冲区的所有权移交至另一队列。

queueFamilyIndexCount – pQueueFamilyIndices 阵列中的队列索引数量（仅指定为并发共享模式时）。

pQueueFamilyIndices – 包含所有队列（将引用缓冲区）的索引阵列（仅指定为并发共享模式时）。

为创建缓冲区，我们必须调用 vkCreateBuffer() 函数。

缓冲区内存分配

我们接下来创建内存对象，以备份缓冲区存储。

```
VkMemoryRequirements buffer_memory_requirements;
vkGetBufferMemoryRequirements( GetDevice(), buffer, &buffer_memory_requirements );

VkPhysicalDeviceMemoryProperties memory_properties;
vkGetPhysicalDeviceMemoryProperties( GetPhysicalDevice(), &memory_properties );

for( uint32_t i = 0; i < memory_properties.memoryTypeCount; ++i ) {
  if( (buffer_memory_requirements.memoryTypeBits & (1 << i)) &&
    (memory_properties.memoryTypes[i].propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT) ) {

    VkMemoryAllocateInfo memory_allocate_info = {
      VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,       // VkStructureType                sType
      nullptr,                                      // const void                    *pNext
      buffer_memory_requirements.size,              // VkDeviceSize                   allocationSize
      i                                             // uint32_t                       memoryTypeIndex
```

```
      };

      if( vkAllocateMemory( GetDevice(), &memory_allocate_info, nullptr, memory ) == VK_SUCCESS ) {
        return true;
      }
    }
  }
  return false;
```

10.Tutorial04.cpp，函数 AllocateBufferMemory()

首先必须检查创建缓冲区需满足哪些内存要求。 为此我们调用 vkGetBufferMemoryRequirements() 函数。 它将供内存创建的参数保存在我们在最后一个参数中提供了地址的变量中。 该变量的类型必须为 VkMemoryRequirements，并包含与所需大小、内存对齐，以及支持内存类型等相关的信息。 内存类型有哪些？

每种设备都可拥有并展示不同的内存类型 — 属性不同的尺寸堆。 一种内存类型可能是设备位于 GDDR 芯片上的本地内存（因此速度极快）。 另一种可能是显卡和 CPU 均可见的共享内存。 显卡和应用都可以访问该内存，但这种内存的速度比（仅供显卡访问的）设备本地内存慢。

为查看可用的内存堆和类型，我们需要调用 vkGetPhysicalDeviceMemoryProperties() 函数，将相关内存信息保存在类型变量 VkPhysicalDeviceMemoryProperties 中。 它包含以下信息：

memoryHeapCount – 特定设备展示的内存堆数量。
memoryHeaps – 内存堆阵列。 每个堆代表大小和属性各不相同的内存。
memoryTypeCount – 特定设备展示的内存类型数量。
memoryTypes – 内存类型阵列。 每个要素描述特定的内存属性，并包含拥有这些特殊属性的内存堆索引。
为特定缓冲区分配内存之前，我们需要查看哪种内存类型满足缓冲区的内存要求。 如果还有其他特定的需求，我们也需要检查。 为此，我们迭代所有可用的内存类型。 缓冲区内存要求中有一个称为 memoryTypeBits 的字段，如果在该字段中设置特定索引的位，表示我们可以为特定缓冲区分配该索引

代表的类型的内存。 但必须记住，尽管始终有一种内存类型满足缓冲区的内存要求，但可能不支持其他特定需求。 在这种情况下，我们需要查看另一种内存类型，或更改其他要求。

这里我们的其他要求指内存需要对主机是可见的， 表示应用可以映射并访问该内存 — 读写数据。 这种内存的速度通常比设备本地内存慢，但这样我们可以轻松为顶点属性上传数据。 接下来的教程将介绍如何使用设备本地内存，以提升性能。

幸运的是，主机可见要求非常普遍，因此我们能够轻松找到一种内存类型既能满足缓冲区的内存需求，也具备主机可见性。 然后我们准备类型变量 VkMemoryAllocateInfo，并填充其中的字段：

sType – 结构类型，此处设置为 VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO。
pNext – 为扩展功能预留的指示器。
allocationSize – 要求分配的最小内存。
memoryTypeIndex – 希望用于已创建内存对象的内存类型索引。 该索引的其中一位在缓冲区内存要求中设置（值为 1）。
填充该结构后，我们调用 vkAllocateMemory() 并检查内存对象分配是否成功。

绑定缓冲区内存

创建完内存对象后，必须将其绑定至缓冲区。 如果不绑定，缓冲区中将没有存储空间，我们将无法保存任何数据。

```
if( !AllocateBufferMemory( Vulkan.VertexBuffer.Handle, &Vulkan.VertexBuffer.Memory ) ) {
  std::cout << "Could not allocate memory for a vertex buffer!" << std::endl;
  return false;
}

if( vkBindBufferMemory( GetDevice(), Vulkan.VertexBuffer.Handle, Vulkan.VertexBuffer.Memory, 0 ) != VK_SUCCESS ) {
  std::cout << "Could not bind memory for a vertex buffer!" << std::endl;
```

```
    return false;
}
```

11.Tutorial04.cpp，函数 CreateVertexBuffer()

函数 AllocateBufferMemory() 可分配内存对象， 之前已介绍过。 创建内存对象时，我们通过调用 vkBindBufferMemory() 函数，将其绑定至缓冲区。 在调用期间，我们必须指定缓冲区句柄、内存对象句柄和偏移。 偏移非常重要，需要我们另加介绍。

查询缓冲区内存要求时，我们获取了有关所需大小、内存类型和对齐方面的信息。 不同的缓冲区用法要求不同的内存对齐。 内存对象的开头（偏移为 0）满足所有对齐要求。 这表示所有内存对象将在满足所有不同用法要求的地址创建。 因此偏移指定为 0 时，我们完全不用担心。

但我们可以创建更大的内存对象，并将其用作多个缓冲区（或图像）的存储空间。 事实上我们建议使用这种方法。 创建大型内存对象表示我们只需创建较少的内存对象。 这有利于驱动程序跟踪较少数量的对象。 出于操作系统要求和安全措施，驱动程序必须跟踪内存对象。 大型内存对象不会造成较大的内存碎片问题。 最后，我们还应分配较多的内存数量，保持对象的相似性，以提高缓存命中率，从而提升应用性能。

但当我们分配大型内存对象并将其绑定至多个缓冲区（或图像）时，并非所有对象都能绑定在 0 偏移的位置。 只有一种可以如此，其他必须绑定得远一些，在第一个缓冲区（或图像）使用的空间之后。 因此第二个，以及绑定至相同内存对象的缓冲区的偏移必须满足查询报告的对齐要求。 而且我们必须牢记这点。 因此对齐成员至关重要。

创建缓冲区，并为其分配和绑定内存时，我们可以用顶点属性数据填充该缓冲区。

上传顶点数据

我们创建了缓冲区，并绑定了主机可见的内存。 这表示我们可以映射该内存、获取该内存的指示器，并使用该指示器将数据从应用拷贝至该缓冲区（与 OpenGL 的 glBufferData() 函数类似）：

```
void *vertex_buffer_memory_pointer;
```

```
if( vkMapMemory( GetDevice(), Vulkan.VertexBuffer.Memory, 0, Vulkan.VertexBuffer.Size, 0, &vertex_buffer_memory_pointer ) != VK_SUCCESS ) {
  std::cout << "Could not map memory and upload data to a vertex buffer!" << std::endl;
  return false;
}

memcpy( vertex_buffer_memory_pointer, vertex_data, Vulkan.VertexBuffer.Size );

VkMappedMemoryRange flush_range = {
  VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE,          // VkStructureType        sType
  nullptr,                                        // const void            *pNext
  Vulkan.VertexBuffer.Memory,                     // VkDeviceMemory         memory
  0,                                              // VkDeviceSize           offset
  VK_WHOLE_SIZE                                   // VkDeviceSize           size
};
vkFlushMappedMemoryRanges( GetDevice(), 1, &flush_range );

vkUnmapMemory( GetDevice(), Vulkan.VertexBuffer.Memory );

return true;
```

12.Tutorial04.cpp，函数 CreateVertexBuffer()

我们调用 vkMapMemory() 函数映射内存。 在该调用中，必须指定我们希望映射哪个内存对象以及访问区域。 区域表示内存对象的存储与大小开头的偏移。 调用成功后我们获取指示器。 我们可以使用该指示器将数据从应用拷贝至提供的内存地址。 这里我们从包含顶点位置和颜色的阵列拷贝顶点数据。

内存拷贝操作之后，取消内存映射之前（无需取消内存映射，可以保留指示器，不会影响性能），我们需要告知驱动程序我们的操作修改了哪部分内存。该操作称为 flushing。 尽管如此，我们指定所有内存范围，以支持应用将数据拷贝至该范围。 范围无需具备连续性。 通过包含以下字段的

VkMappedMemoryRange 要素阵列可定义该范围：

sType – 结构类型，此处等于 VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE。
pNext – 为扩展功能预留的指示器。
memory – 已映射并修改的内存对象句柄。
offset – 特定范围开始的偏移（从特定内存对象的存储开头）。
size – 受影响区域的大小（字节）。 如果从偏移到结尾的整个内存均受影响，我们可以使用 VK_WHOLE_SIZE 的特定值。
定义应闪存的所有内存范围时，我们可调用 vkFlushMappedMemoryRanges() 函数。 之后，驱动程序将知道哪些部分已修改，并重新加载它们（即刷新高速缓存）。 重新加载通常在壁垒上执行。 修改缓冲区后，我们应设置缓冲区内存壁垒，告知驱动程序部分操作对缓冲区造成了影响，应进行刷新。 不过幸运的是，在本示例中，驱动程序隐式地将壁垒放在命令缓冲区（引用特定缓冲区且不要求其他操作）的提交操作上。 现在我们可以在渲染命令记录期间使用该缓冲区。

渲染资源创建

现在必须准备命令缓冲区记录所需的资源。 我们在之前的教程中为每个交换链图像记录了一个静态命令缓冲区。 这里我们将重新整理渲染代码。 我们仍然展示一个比较简单的静态场景，不过此处介绍的方法可以用于展示动态场景的真实情况。

要想有效地记录命令缓冲区并将其提交至队列，我们需要四类资源：命令缓冲区、旗语、栅栏和帧缓冲器。 旗语我们之前介绍过，用于内部队列同步。而栅栏支持应用检查是否出现了特定情况，例如命令缓冲区提交至队列后是否已执行完。 如有必要，应用可以等待栅栏，直到收到信号。 一般来说，旗语用于同步队列 (GPU)，栅栏用于同步应用 (CPU)。

如要渲染一帧简单的动画，我们（至少）需要一个命令缓冲区，两个旗语 — 一个用于获取交换链图像（图像可用的旗语），另一个用于发出可进行演示的信号（渲染已完成的旗语） —，一个栅栏和一个帧缓冲器。 栅栏稍后将用于检查我们是否重新记录了特定命令缓冲区。 我们将保留部分渲染资源，以调用虚拟帧。 虚拟帧（包含一个命令缓冲区、两个旗语、一个栅栏和一个帧缓冲器）的数量与交换链图像数量无关。

渲染算法进展如下： 我们将渲染命令记录至第一个虚拟帧，然后将其提交至队列。 然后记录另一帧（命令缓冲区）并将其提交至队列。 直到记录并提

交完所有虚拟帧。 这时我们通过提取并再次重新记录最之前（最早提交）的命令缓冲区，开始重复使用帧。 然后使用另一命令缓冲区，依此类推。

此时栅栏登场。我们不允许记录已提交至队列，但在队列中的没有执行完的命令缓冲区。 在记录命令缓冲区期间，我们可以使用“simultaneous use”标记，以记录或重新提交之前已提交过的命令缓冲区。 不过这样会影响性能。 最好的方法是使用栅栏，检查命令缓冲区是否不能再使用。 如果显卡仍然在处理命令缓冲区，我们可以等待与特定命令缓冲区相关的栅栏，或将这额外的时间用于其他目的，比如改进 AI 计算，并在这之后再次检查以查看栅栏是否收到了信号。

我们应准备多少虚拟帧？ 一个远远不够。 记录并提交单个命令缓冲区时，我们会立即等待，直到能够重新记录该缓冲区。 这样会导致 CPU 和 GPU 浪费时间。 GPU 的速度通常更快，因此等待 CPU 会造成 GPU 等待的时间更长。 我们应该保持 GPU 的繁忙状态。 因此我们创建了 Vulkan 等瘦 API。使用两个虚拟帧将会显著提升性能，因为这样会大大缩短 CPU 和 GPU 的等待时间。 添加第三个虚拟帧能够进一步提升性能，但提升空间不大。 使用四组以上渲染资源没有太大意义，因为其性能提升可忽略不计（当然这取决于已渲染场景的复杂程度和类似 CPU 的物理组件或 AI 所执行的计算）。 增加虚拟帧数量时，会提高输入延迟，因为我们在 CPU 背后演示的帧为 1-3 个。 因此两个或三个虚拟帧似乎是最合理的组合，能够使性能、内存使用和输入延迟之间达到最佳动平衡。

您可能想知道虚拟帧的数量为何与交换链图像数量无关。 这种方法会影响应用的行为。 创建交换链时，我们请求所需图像的最小数量，但驱动程序允许创建更多图像。 因此不同的硬件厂商可能实施提供不同数量交换链图像的驱动程序，甚至要求相同（演示模式和最少图像数量）时也如此。 建立虚拟帧数量与交换链数量的相关性后，应用将在一个显卡上仅使用两个虚拟帧，而在另一显卡上使用四个虚拟帧。 这样会影响性能和之前提到的输入延迟。因此我们不希望出现这种行为。 通过保持固定数量的虚拟帧，我们可以控制渲染算法，并对其进行调优以满足需求，即渲染时间与 AI 或物理计算之间的平衡。

命令池创建

分配命令缓冲区之前，我们首先需要创建一个命令池。

```
VkCommandPoolCreateInfo cmd_pool_create_info = {
   VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,        // VkStructureType                sType
```

```
    nullptr,                                                    // const void                   *pNext
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | // VkCommandPoolCreateFlags        flags
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT,
    queue_family_index                                          // uint32_t                     queueFamilyIndex
};

if( vkCreateCommandPool( GetDevice(), &cmd_pool_create_info, nullptr, pool ) != VK_SUCCESS ) {
    return false;
}
return true;
```

13.Tutorial04.cpp，函数 CreateCommandPool()

通过调用 vkCreateCommandPool() 创建命令池，其中要求我们提供类型变量 VkCommandPoolCreateInfo 的指示器。 与之前的教程相比，代码基本保持不变。 但这次添加两个其他的标记以创建命令池：

VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT – 表示通过命令池分配的命令缓冲区可单独重新设置。 正常来说，如果没有这一标记，我们将无法多次重新记录相同的命令缓冲区。 它必须首先重新设置。 而且从命令池创建的命令缓冲区只能一起重新设置。 指定该标记可支持我们单独重新设置命令缓冲区，而且（甚至更好）通过调用 vkBeginCommandBuffer() 函数隐式地完成这一操作。
VK_COMMAND_POOL_CREATE_TRANSIENT_BIT – 该标记告知驱动程序通过该命令池分配的命令缓冲区将仅短时间内存在，需要经常重新记录和重新设置。该信息可帮助优化命令缓冲区分配并以更好地方式执行。
命令缓冲区分配

命令缓冲区分配与之前的相同。

```
for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if( !AllocateCommandBuffers( Vulkan.CommandPool, 1, &Vulkan.RenderingResources[i].CommandBuffer ) ) {
```

```
      std::cout << "Could not allocate command buffer!" << std::endl;
      return false;
    }
  }
  return true;
```

14.Tutorial04.cpp，函数 CreateCommandBuffers()

唯一的变化是命令缓冲区收集在渲染资源矢量中。 每个渲染资源结构均包含一个命令缓冲区、图像可用旗语、渲染已完成旗语、一个栅栏和一个帧缓冲器。 命令缓冲区循环分配。 渲染资源矢量中的要素数量可随意选择。 在本教程中，该数量为 3。

旗语创建

负责创建旗语的代码非常简单，与之前的相同：

```
VkSemaphoreCreateInfo semaphore_create_info = {
  VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,          // VkStructureType                sType
  nullptr,                                          // const void*                    pNext
  0                                                 // VkSemaphoreCreateFlags         flags
};

for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
  if( (vkCreateSemaphore( GetDevice(), &semaphore_create_info, nullptr, &Vulkan.RenderingResources[i].ImageAvailableSemaphore ) != VK_SUCCESS) ||
    (vkCreateSemaphore( GetDevice(), &semaphore_create_info, nullptr, &Vulkan.RenderingResources[i].FinishedRenderingSemaphore ) != VK_SUCCESS) ) {
    std::cout << "Could not create semaphores!" << std::endl;
    return false;
  }
```

```
}
return true;
```

15.Tutorial04.cpp，函数 CreateSemaphores()

栅栏创建

以下代码负责创建栅栏对象：

```
VkFenceCreateInfo fence_create_info = {
   VK_STRUCTURE_TYPE_FENCE_CREATE_INFO,                  // VkStructureType                  sType
   nullptr,                                              // const void                      *pNext
   VK_FENCE_CREATE_SIGNALED_BIT                          // VkFenceCreateFlags               flags
};

for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
   if( vkCreateFence( GetDevice(), &fence_create_info, nullptr, &Vulkan.RenderingResources[i].Fence ) != VK_SUCCESS ) {
      std::cout << "Could not create a fence!" << std::endl;
      return false;
   }
}
return true;
```

16.Tutorial04.cpp，函数 CreateFences()

我们调用 vkCreateFence() 函数，以创建栅栏对象。 它从其他参数中接受类型变量 VkFenceCreateInfo 的指示器，其中包含以下成员：

sType – 结构类型。 此处应设置为 VK_STRUCTURE_TYPE_FENCE_CREATE_INFO。

pNext – 为扩展功能预留的指示器。
flags – 目前该参数支持创建已收到信号的栅栏。
栅栏包含两种状态：收到信号和未收到信号。 该应用检查特定栅栏是否处于收到信号状态，否则将等待栅栏收到信号。 提交至队列的所有操作均处理完后，由 GPU 发出信号。 提交命令缓冲区时，我们可以提供一个栅栏，该栅栏将在队列执行完提交操作发布的所有命令后收到信号。 栅栏收到信号后，将由应用负责将其重新设置为未收到信号状态。

为何创建收到信号的栅栏？ 渲染算法将命令记录至第一个命令缓冲区，然后是第二个命令缓冲器，之后是第三个，然后（队列中的执行结束后）再次记录至第一个命令缓冲区。 我们使用栅栏检查能否再次记录特定命令缓冲区。 那么第一次记录会怎样呢？ 我们不希望第一次命令缓冲区记录和接下来的记录操作为不同的代码路径。 因此第一次发布命令缓冲区记录时，我们还检查栅栏是否已收到信号。 但因为我们没有提交特定命令缓冲区，因此与此相关的栅栏在执行完成后无法变成收到信号状态。 因此需要以已收到信号状态创建栅栏。 这样第一次记录时我们无需等待它变成已收到信号状态（因为它已经是已收到信号状态），但检查之后，我们要重新设置并立即前往记录代码。 之后我们提交命令缓冲区并提供相同的栅栏，这样在完成操作后将收到队列发来的信号。 下一次当我们希望将渲染命令记录至相同的命令缓冲区时，我们可以执行同样的操作：等待栅栏，重新设置栅栏，然后开始记录命令缓冲区。

绘制

现在我们准备记录渲染操作。 我们将正好在提交至队列之前记录命令缓冲区。 记录并提交一个命令缓冲区，然后记录并提交下一个命令缓冲区，然后记录并提交另一个。 在这之后我们提取第一个命令缓冲区，检查是否可用，并记录并将其提交至队列。

```
static size_t                    resource_index = 0;
RenderingResourcesData &rt_rendering_resource = Vulkan.RenderingResources[resource_index];
VkSwapchainKHR                   swap_chain = GetSwapChain().Handle;
uint32_t                         image_index;

resource_index = (resource_index + 1) % VulkanTutorial04Parameters::ResourcesCount;
```

```
if( vkWaitForFences( GetDevice(), 1, ¤t_rendering_resource.Fence, VK_FALSE, 1000000000 ) != VK_SUCCESS ) {
  std::cout << "Waiting for fence takes too long!" << std::endl;
  return false;
}
vkResetFences( GetDevice(), 1, ¤t_rendering_resource.Fence );

VkResult result = vkAcquireNextImageKHR( GetDevice(), swap_chain, UINT64_MAX, current_rendering_resource.ImageAvailableSemaphore, VK_NULL_HANDLE,
&image_index );
switch( result ) {
  case VK_SUCCESS:
  case VK_SUBOPTIMAL_KHR:
    break;
  case VK_ERROR_OUT_OF_DATE_KHR:
    return OnWindowSizeChanged();
  default:
    std::cout << "Problem occurred during swap chain image acquisition!" << std::endl;
    return false;
}

if( !PrepareFrame( current_rendering_resource.CommandBuffer, GetSwapChain().Images[image_index], current_rendering_resource.Framebuffer ) ) {
  return false;
}

VkPipelineStageFlags wait_dst_stage_mask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
VkSubmitInfo submit_info = {
  VK_STRUCTURE_TYPE_SUBMIT_INFO,                    // VkStructureType                sType
```

```
    nullptr,                                                    // const void                     *pNext
    1,                                                          // uint32_t                        waitSemaphoreCount
    ¤t_rendering_resource.ImageAvailableSemaphore,       // const VkSemaphore              *pWaitSemaphores
    &wait_dst_stage_mask,                                       // const VkPipelineStageFlags     *pWaitDstStageMask;
    1,                                                          // uint32_t                        commandBufferCount
    ¤t_rendering_resource.CommandBuffer,                 // const VkCommandBuffer          *pCommandBuffers
    1,                                                          // uint32_t                        signalSemaphoreCount
    ¤t_rendering_resource.FinishedRenderingSemaphore     // const VkSemaphore              *pSignalSemaphores
};

if( vkQueueSubmit( GetGraphicsQueue().Handle, 1, &submit_info, current_rendering_resource.Fence ) != VK_SUCCESS ) {
    return false;
}

VkPresentInfoKHR present_info = {
    VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,                         // VkStructureType                sType
    nullptr,                                                    // const void                     *pNext
    1,                                                          // uint32_t                        waitSemaphoreCount
    ¤t_rendering_resource.FinishedRenderingSemaphore, // const VkSemaphore              *pWaitSemaphores
    1,                                                          // uint32_t                        swapchainCount
    &swap_chain,                                                // const VkSwapchainKHR           *pSwapchains
    &image_index,                                               // const uint32_t                 *pImageIndices
    nullptr                                                     // VkResult                       *pResults
};
result = vkQueuePresentKHR( GetPresentQueue().Handle, &present_info );
```

```
switch( result ) {
   case VK_SUCCESS:
      break;
   case VK_ERROR_OUT_OF_DATE_KHR:
   case VK_SUBOPTIMAL_KHR:
      return OnWindowSizeChanged();
   default:
      std::cout << "Problem occurred during image presentation!" << std::endl;
      return false;
}

return true;
```

17.Tutorial04.cpp，函数 Draw()

首先提取最近使用的渲染资源。 然后等待与该组相关的栅栏收到信号。 如果收到了信号，表示我们可以安全提取并记录命令缓冲区。 不过它还表示我们可以提取用于获取并演示在特定命令缓冲区中引用的旗语。 不能将同一个旗语用于不同的目的或两项不同的提交操作，必须等待之前的提交操作完成。栅栏可防止我们修改命令缓冲区和旗语。 而且大家会看到，帧缓冲器也是如此。

栅栏完成后，我们重新设置栅栏并执行与正常绘制相关的操作：获取图像，记录渲染已获取图像的操作，提交命令缓冲区，并演示图像。

然后提取另一渲染资源集并执行相同的操作。 由于保留了三组渲染资源，三个虚拟帧，我们可缩短等待栅栏收到信号的时间。

记录命令缓冲区

负责记录命令缓冲区的函数很长。 此时会更长，因为我们使用顶点缓冲区和动态视口及 scissor 测试。 而且我们还创建临时帧缓冲器！

帧缓冲器的创建非常简单、快速。 同时保留帧缓冲器对象和交换链意味着，需要重新创建交换链时，我们需要重新创建这些对象。 如果渲染算法复杂，我们将有多个图像以及与之相关的帧缓冲器。 如果这些图像的大小必须与交换链图像的大小相同，那么我们需要重新创建所有图像（以纳入潜在的大小变化）。 因此最好按照需求创建帧缓冲器，这样也更方便。 这样它们的大小将始终符合要求。 帧缓冲器在面向特定图像创建的图像视图上运行。 交换链重新创建时，旧的图像将无效并消失。 因此我们必须重新创建图像视图和帧缓冲器。

在“03-第一个三角形”教程中，我们有大小固定的帧缓冲器，而且需要与交换链同时重新创建。 现在我们的帧缓冲器对象在每个虚拟帧资源组中。 记录命令缓冲器之前，我们要为将渲染的图像创建帧缓冲器，其大小与图像相同。 这样当我们重新创建交换链时，将立即调整下一帧的大小，而且新交换链图像的句柄及其图像视图将用于创建帧缓冲器。

记录使用渲染通道和帧缓冲器对象的命令缓冲区时，在队列处理命令缓冲区期间，帧缓冲器必须始终保持有效。 创建新的帧缓冲器时，命令提交至队列的操作完成后我们才开始破坏它。 不过由于我们使用栅栏，而且等待与特定命令缓冲区相关的栅栏，因为能够确保安全地破坏帧缓冲器。 然后我们创建新的帧缓冲器，以纳入潜在的大小和图像句柄变化。

```
if( framebuffer != VK_NULL_HANDLE ) {
  vkDestroyFramebuffer( GetDevice(), framebuffer, nullptr );
}

VkFramebufferCreateInfo framebuffer_create_info = {
  VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO,          // VkStructureType                sType
  nullptr,                                            // const void                    *pNext
  0,                                                  // VkFramebufferCreateFlags       flags
  Vulkan.RenderPass,                                  // VkRenderPass                   renderPass
  1,                                                  // uint32_t                       attachmentCount
  &image_view,                                        // const VkImageView             *pAttachments
  GetSwapChain().Extent.width,                        // uint32_t                       width
  GetSwapChain().Extent.height,                       // uint32_t                       height
```

```
    1                                                       // uint32_t                                 layers
};

if( vkCreateFramebuffer( GetDevice(), &framebuffer_create_info, nullptr, &framebuffer ) != VK_SUCCESS ) {
    std::cout << "Could not create a framebuffer!" << std::endl;
    return false;
}

return true;
```

18.Tutorial04.cpp，函数 CreateFramebuffer()

创建帧缓冲器时，我们提取当前的交换链扩展，以及已获取交换链图像的图像视图。

接下来开始记录命令缓冲区：

```
if( !CreateFramebuffer( framebuffer, image_parameters.View ) ) {
    return false;
}

VkCommandBufferBeginInfo command_buffer_begin_info = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,        // VkStructureType                          sType
    nullptr,                                            // const void                               *pNext
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,        // VkCommandBufferUsageFlags                flags
    nullptr                                             // const VkCommandBufferInheritanceInfo     *pInheritanceInfo
};
```

```
vkBeginCommandBuffer( command_buffer, &command_buffer_begin_info );

VkImageSubresourceRange image_subresource_range = {
  VK_IMAGE_ASPECT_COLOR_BIT,                    // VkImageAspectFlags          aspectMask
  0,                                            // uint32_t                    baseMipLevel
  1,                                            // uint32_t                    levelCount
  0,                                            // uint32_t                    baseArrayLayer
  1                                             // uint32_t                    layerCount
};

if( GetPresentQueue().Handle != GetGraphicsQueue().Handle ) {
  VkImageMemoryBarrier barrier_from_present_to_draw = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,     // VkStructureType             sType
    nullptr,                                    // const void                  *pNext
    VK_ACCESS_MEMORY_READ_BIT,                  // VkAccessFlags               srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,                  // VkAccessFlags               dstAccessMask
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout               oldLayout
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout               newLayout
    GetPresentQueue().FamilyIndex,              // uint32_t                    srcQueueFamilyIndex
    GetGraphicsQueue().FamilyIndex,             // uint32_t                    dstQueueFamilyIndex
    image_parameters.Handle,                    // VkImage                     image
    image_subresource_range                     // VkImageSubresourceRange     subresourceRange
  };
  vkCmdPipelineBarrier( command_buffer, VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, 0,
0, nullptr, 0, nullptr, 1, &barrier_from_present_to_draw );
}
```

```
VkClearValue clear_value = {
   { 1.0f, 0.8f, 0.4f, 0.0f },                    // VkClearColorValue                    color
};

VkRenderPassBeginInfo render_pass_begin_info = {
   VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,      // VkStructureType                      sType
   nullptr,                                       // const void                          *pNext
   Vulkan.RenderPass,                             // VkRenderPass                         renderPass
   framebuffer,                                   // VkFramebuffer                        framebuffer
   {                                              // VkRect2D                             renderArea
      {                                           // VkOffset2D                           offset
         0,                                       // int32_t                               x
         0                                        // int32_t                               y
      },
      GetSwapChain().Extent,                      // VkExtent2D                           extent;
   },
   1,                                             // uint32_t                             clearValueCount
   &clear_value                                   // const VkClearValue                  *pClearValues
};

vkCmdBeginRenderPass( command_buffer, &render_pass_begin_info, VK_SUBPASS_CONTENTS_INLINE );
```

19.Tutorial04.cpp，函数 PrepareFrame()

首先定义类型变量 VkCommandBufferBeginInfo，并指定命令缓冲区只能提交一次。指定 VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT 标记时，不能多次提交特定命令缓冲区。每次提交后，必须重新设置。但记录操作重新设置它的原因是

VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT 标记用于命令池创建。

接下来我们定义有关图像内存壁垒的子资源范围。 交换链图像布局过渡在渲染通道中隐式执行，但如果显卡队列和演示队列不同，则必须手动执行队列过渡。

然后启动包含临时帧缓冲器对象的渲染通道。

```
vkCmdBindPipeline( command_buffer, VK_PIPELINE_BIND_POINT_GRAPHICS, Vulkan.GraphicsPipeline );

VkViewport viewport = {
   0.0f,                                                  // float                                  x
   0.0f,                                                  // float                                  y
   static_cast<float>(GetSwapChain().Extent.width),       // float                                  width
   static_cast<float>(GetSwapChain().Extent.height),      // float                                  height
   0.0f,                                                  // float                                  minDepth
   1.0f                                                   // float                                  maxDepth
};

VkRect2D scissor = {
   {                                                      // VkOffset2D                             offset
      0,                                                  // int32_t                                x
      0                                                   // int32_t                                y
   },
   {                                                      // VkExtent2D                             extent
      GetSwapChain().Extent.width,                        // uint32_t                               width
      GetSwapChain().Extent.height                        // uint32_t                               height
```

```
    }
};

vkCmdSetViewport( command_buffer, 0, 1, &viewport );
vkCmdSetScissor( command_buffer, 0, 1, &scissor );

VkDeviceSize offset = 0;
vkCmdBindVertexBuffers( command_buffer, 0, 1, &Vulkan.VertexBuffer.Handle, &offset );

vkCmdDraw( command_buffer, 4, 1, 0, 0 );

vkCmdEndRenderPass( command_buffer );

if( GetGraphicsQueue().Handle != GetPresentQueue().Handle ) {
    VkImageMemoryBarrier barrier_from_draw_to_present = {
        VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,                 // VkStructureType                sType
        nullptr,                                                // const void                    *pNext
        VK_ACCESS_MEMORY_READ_BIT,                              // VkAccessFlags                  srcAccessMask
        VK_ACCESS_MEMORY_READ_BIT,                              // VkAccessFlags                  dstAccessMask
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                        // VkImageLayout                  oldLayout
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,                        // VkImageLayout                  newLayout
        GetGraphicsQueue().FamilyIndex,                         // uint32_t                       srcQueueFamilyIndex
        GetPresentQueue().FamilyIndex,                          // uint32_t                       dstQueueFamilyIndex
        image_parameters.Handle,                                // VkImage                        image
        image_subresource_range                                 // VkImageSubresourceRange        subresourceRange
    };
```

```
    vkCmdPipelineBarrier( command_buffer, VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0, 0, nullptr, 0, nullptr, 1, &barrier_from_draw_to_present );
  }

  if( vkEndCommandBuffer( command_buffer ) != VK_SUCCESS ) {
    std::cout << "Could not record command buffer!" << std::endl;
    return false;
  }
  return true;
```

20.Tutorial04.cpp，函数 PrepareFrame()

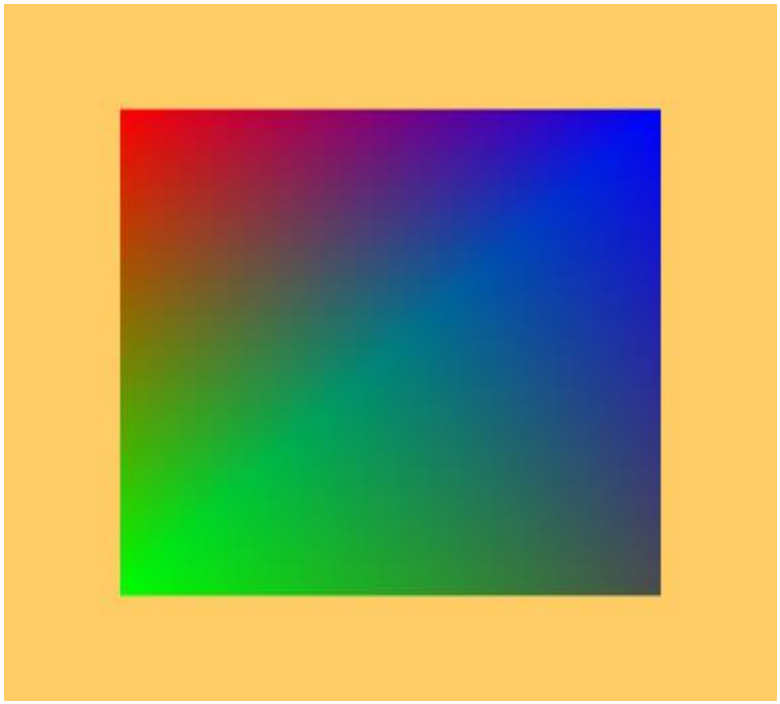
接下来我们绑定图像管道。 它包含两个标记为动态的状态：视口和 scissor 测试。 因此我们准备能够定义视口和 scissor 测试参数的结构。 通过调用 vkCmdSetViewport() 函数设置动态视口。 通过调用 vkCmdSetScissor() 函数设置动态 scissor 测试。 这样图像管道可用于渲染大小不同的图像。

进行绘制之前的最后一件事是绑定相应的顶点缓冲区，为顶点属性提供缓冲区数据。 此操作通过调用 vkCmdBindVertexBuffers() 函数完成。 我们指定一个绑定号码（哪个顶点属性集从该缓冲区提取数据）、一个缓冲区句柄指示器（或较多句柄，如果希望绑定多个绑定的缓冲区），以及偏移。 偏移指定应从缓冲区的较远部分提取有关顶点属性的数据。 但指定的偏移不能大于相应缓冲区（该缓冲区未绑定内存对象）的范围。

现在我们已经指定了全部所需要素：帧缓冲器、视口和 scissor 测试，以及顶点缓冲区。 我们可以绘制几何图形、完成渲染通道，并结束命令缓冲区。

教程 04 执行

以下是渲染操作的结果：



我们渲染了一个四边形，各个角落的颜色均不相同。 尝试更改窗口的大小；之前的三角形始终保持相同的大小，仅应用窗口右侧和底部的黑色方框变大或变小。 现在，由于是动态视口，因此四边形将随着窗口的变化而变大或变小。

清空

完成渲染后，关闭应用之前，我们需要破坏所有资源。 以下代码负责完成此项操作：

```
if( GetDevice() != VK_NULL_HANDLE ) {
   vkDeviceWaitIdle( GetDevice() );
```

```
for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if( Vulkan.RenderingResources[i].Framebuffer != VK_NULL_HANDLE ) {
        vkDestroyFramebuffer( GetDevice(), Vulkan.RenderingResources[i].Framebuffer, nullptr );
    }
    if( Vulkan.RenderingResources[i].CommandBuffer != VK_NULL_HANDLE ) {
        vkFreeCommandBuffers( GetDevice(), Vulkan.CommandPool, 1, &Vulkan.RenderingResources[i].CommandBuffer );
    }
    if( Vulkan.RenderingResources[i].ImageAvailableSemaphore != VK_NULL_HANDLE ) {
        vkDestroySemaphore( GetDevice(), Vulkan.RenderingResources[i].ImageAvailableSemaphore, nullptr );
    }
    if( Vulkan.RenderingResources[i].FinishedRenderingSemaphore != VK_NULL_HANDLE ) {
        vkDestroySemaphore( GetDevice(), Vulkan.RenderingResources[i].FinishedRenderingSemaphore, nullptr );
    }
    if( Vulkan.RenderingResources[i].Fence != VK_NULL_HANDLE ) {
        vkDestroyFence( GetDevice(), Vulkan.RenderingResources[i].Fence, nullptr );
    }
}

if( Vulkan.CommandPool != VK_NULL_HANDLE ) {
    vkDestroyCommandPool( GetDevice(), Vulkan.CommandPool, nullptr );
    Vulkan.CommandPool = VK_NULL_HANDLE;
}

if( Vulkan.VertexBuffer.Handle != VK_NULL_HANDLE ) {
    vkDestroyBuffer( GetDevice(), Vulkan.VertexBuffer.Handle, nullptr );
```

```
    Vulkan.VertexBuffer.Handle = VK_NULL_HANDLE;
  }

  if( Vulkan.VertexBuffer.Memory != VK_NULL_HANDLE ) {
    vkFreeMemory( GetDevice(), Vulkan.VertexBuffer.Memory, nullptr );
    Vulkan.VertexBuffer.Memory = VK_NULL_HANDLE;
  }

  if( Vulkan.GraphicsPipeline != VK_NULL_HANDLE ) {
    vkDestroyPipeline( GetDevice(), Vulkan.GraphicsPipeline, nullptr );
    Vulkan.GraphicsPipeline = VK_NULL_HANDLE;
  }

  if( Vulkan.RenderPass != VK_NULL_HANDLE ) {
    vkDestroyRenderPass( GetDevice(), Vulkan.RenderPass, nullptr );
    Vulkan.RenderPass = VK_NULL_HANDLE;
  }
}
```

21.Tutorial04.cpp，destructor

设备处理完所有提交至阵列的命令后，我们开始破坏所有资源。 资源破坏以相反的顺序进行。 首先破坏所有渲染资源：帧缓冲器、命令缓冲区、旗语和栅栏。 栅栏通过调用 vkDestroyFence() 函数破坏。 然后破坏命令池。 之后通过调用 vkDestroyBuffer() 函数和 vkFreeMemory() 函数分别破坏缓冲区和空闲内存对象。 最后破坏管道对象和渲染通道。

结论

本教程的编写以“03-第一个三角形”教程为基础。 我们通过在图像管道中使用顶点属性，并在记录命令缓冲区期间绑定顶点缓冲区，以此改进渲染过程。 我们介绍了顶点属性的数量和布局， 针对视口和 scissors 测试引入了动态管道状态， 并学习了如何创建缓冲区和内存对象，以及如何相互绑定。 我们还映射了内存，并将数据从 CPU 上传至 GPU。

我们创建了渲染资源集，以高效记录和发布渲染命令。 这些资源包括命令缓冲器、旗语、栅栏和帧缓冲器。 我们学习了如何使用栅栏，如何设置动态管道状态的值，以及如何在记录命令缓冲区期间绑定顶点缓冲区（顶点属性数据源）。

下节教程将介绍分期资源。 它们是用于在 CPU 和 GPU 之间拷贝数据的中间缓冲区。 这样应用无需映射用于渲染的缓冲区（或图像），它们也不必绑定至设备的本地（快速）内存。

Tutorial 5: Staging Resources – Copying Data Between Buffers

Tutorial 5: Staging Resources – Copying Data between Buffers

In this part of the tutorial we will focus on improving performance. At the same time, we will prepare for the next tutorial, in which we introduce images and descriptors (shader resources). Using the knowledge we gather here, it will be easier for us to follow the next part and squeeze as much performance as possible

from our graphics hardware.

What are “staging resources” or “staging buffers”? They are intermediate or temporary resources used to transfer data from an application (CPU) to a graphics card’s memory (GPU). We need them to increase our application’s performance.

In Part 4 of the tutorial we learned how to use buffers, bind them to a host-visible memory, map this memory, and transfer data from the CPU to the GPU. This approach is easy and convenient for us, but we need to know that host-visible parts of a graphics card’s memory aren’t the most efficient. Typically, they are much slower than the parts of the memory that are not directly accessible to the application (cannot be mapped by an application). This causes our application to execute in a sub-optimal way.

One solution to this problem is to always use device-local memory for all resources involved in a rendering process. But as device-local memory isn’t accessible for an application, we cannot directly transfer any data from the CPU to such memory. That’s why we need intermediate, or staging, resources.

In this part of the tutorial we will bind the buffer with vertex attribute data to the device-local memory. And we will use the staging buffer to mediate the transfer of data from the CPU to the vertex buffer.

Again, only the differences between this tutorial and the previous tutorial (Part 4) are described.

Creating Rendering Resources

This time I have moved rendering resources creation to the beginning of our code. Later we will need to record and submit a command buffer to transfer data from the staging resource to the vertex buffer. I have also refactored rendering resource creation code to eliminate multiple loops and replace them with only one loop. In this loop we can create all resources that compose our virtual frame.

```
bool Tutorial05::CreateRenderingResources() {
    if ( !CreateCommandPool( GetGraphicsQueue().FamilyIndex, &Vulkan.CommandPool ) ) {
```

```
      return false;
    }

    for ( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
      if ( !AllocateCommandBuffers( Vulkan.CommandPool, 1, &Vulkan.RenderingResources[i].CommandBuffer ) ) {
        return false;
      }

      if ( !CreateSemaphore( &Vulkan.RenderingResources[i].ImageAvailableSemaphore ) ) {
        return false;
      }

      if ( !CreateSemaphore( &Vulkan.RenderingResources[i].FinishedRenderingSemaphore ) ) {
        return false;
      }

      if ( !CreateFence( VK_FENCE_CREATE_SIGNALED_BIT, &Vulkan.RenderingResources[i].Fence ) ) {
        return false;
      }
    }
    return true;
}

bool Tutorial05::CreateCommandPool( uint32_t queue_family_index, VkCommandPool *pool ) {
    VkCommandPoolCreateInfo cmd_pool_create_info = {
      VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO, // VkStructureType sType
```

```
    nullptr, // const void *pNext
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | // VkCommandPoolCreateFlags flags
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT,
    queue_family_index // uint32_t queueFamilyIndex
  };

  if ( vkCreateCommandPool( GetDevice(), &cmd_pool_create_info, nullptr, pool ) != VK_SUCCESS ) {
    std::cout << "Could not create command pool!" << std::endl;
    return false;
  }
  return true;
}

bool Tutorial05::AllocateCommandBuffers( VkCommandPool pool, uint32_t count, VkCommandBuffer *command_buffers ) {
  VkCommandBufferAllocateInfo command_buffer_allocate_info = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO, // VkStructureType sType
    nullptr, // const void *pNext
    pool, // VkCommandPool commandPool
    VK_COMMAND_BUFFER_LEVEL_PRIMARY, // VkCommandBufferLevel level
    count // uint32_t bufferCount
  };

  if ( vkAllocateCommandBuffers( GetDevice(), &command_buffer_allocate_info, command_buffers ) != VK_SUCCESS ) {
    std::cout << "Could not allocate command buffer!" << std::endl;
    return false;
  }
```

```
    return true;
}

bool Tutorial05::CreateSemaphore( VkSemaphore *semaphore ) {
    VkSemaphoreCreateInfo semaphore_create_info = {
        VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO, // VkStructureType sType
        nullptr, // const void* pNext
        0 // VkSemaphoreCreateFlags flags
    };

    if ( vkCreateSemaphore( GetDevice(), &semaphore_create_info, nullptr, semaphore ) != VK_SUCCESS ) {
        std::cout << "Could not create semaphore!" << std::endl;
        return false;
    }
    return true;
}

bool Tutorial05::CreateFence( VkFenceCreateFlags flags, VkFence *fence ) {
    VkFenceCreateInfo fence_create_info = {
        VK_STRUCTURE_TYPE_FENCE_CREATE_INFO, // VkStructureType sType
        nullptr, // const void *pNext
        flags // VkFenceCreateFlags flags
    };

    if ( vkCreateFence( GetDevice(), &fence_create_info, nullptr, fence ) != VK_SUCCESS ) {
        std::cout << "Could not create a fence!" << std::endl;
```

```
        return false;
    }
    return true;
}
```

1. Tutorial05.cpp

1.Tutorial05.cpp

First we create a command pool for which we indicate that command buffers allocated from this pool will be short lived. In our case, all command buffers will be submitted only once before rerecording.

Next we iterate over the arbitrary chosen number of virtual frames. In this code example, the number of virtual frames is three. Inside the loop, for each virtual frame, we allocate one command buffer, create two semaphores (one for image acquisition and a second to indicate that frame rendering is done) and a fence. Framebuffer creation is done inside a drawing function, just before command buffer recording.

This is the same set of rendering resources used in Part 4, where you can find a more thorough explanation of what is going on in the code. I will also skip render pass and graphics pipeline creation. They are created in exactly the same way they were created previously. Since nothing has changed here, we will jump directly to buffer creation.

Buffer creation

Here is our general code used for buffer creation:

```
VkBufferCreateInfo buffer_create_info = {
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO, // VkStructureType sType
    nullptr, // const void *pNext
    0, // VkBufferCreateFlags flags
    buffer.Size, // VkDeviceSize size
```

```
    usage, // VkBufferUsageFlags usage
    VK_SHARING_MODE_EXCLUSIVE, // VkSharingMode sharingMode
    0, // uint32_t queueFamilyIndexCount
    nullptr // const uint32_t *pQueueFamilyIndices
};

if ( vkCreateBuffer( GetDevice(), &buffer_create_info, nullptr, &buffer.Handle ) != VK_SUCCESS ) {
    std::cout << "Could not create buffer!" << std::endl;
    return false;
}

if ( !AllocateBufferMemory( buffer.Handle, memoryProperty, &buffer.Memory ) ) {
    std::cout << "Could not allocate memory for a buffer!" << std::endl;
    return false;
}

if ( vkBindBufferMemory( GetDevice(), buffer.Handle, buffer.Memory, 0 ) != VK_SUCCESS ) {
    std::cout << "Could not bind memory to a buffer!" << std::endl;
    return false;
}

return true;
```

2.Tutorial05.cpp, function CreateBuffer()

The code is wrapped into a CreateBuffer() function, which accepts the buffer's usage, size, and requested memory properties. To create a buffer we need to prepare a variable of type VkBufferCreateInfo. It is a structure that contains the following members:

sType – Standard type of the structure. Here it should be equal to VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO.
pNext – Pointer reserved for extensions.
flags – Parameter describing additional properties of the buffer. Right now we can only specify that the buffer can be backed by a sparse memory.
size – Size of the buffer (in bytes).
usage – Bitfield indicating intended usages of the buffer.
sharingMode – Queue sharing mode.
queueFamilyIndexCount – Number of different queue families that will access the buffer in case of a concurrent sharing mode.
pQueueFamilyIndices – Array with indices of all queue families that will access the buffer when concurrent sharing mode is used.
Right now we are not interested in binding a sparse memory. We do not want to share the buffer between different device queues, so sharingMode, queueFamilyIndexCount, and pQueueFamilyIndices parameters are irrelevant. The most important parameters are size and usage. We are not allowed to use a buffer in a way that is not specified during buffer creation. Finally, we need to create a buffer that is large enough to contain our data.

To create a buffer we call the vkCreateBuffer() function, which when successful stores the buffer handle in a variable we provided the address of. But creating a buffer is not enough. A buffer, after creation, doesn't have any storage. We need to bind a memory object (or part of it) to the buffer to back its storage. Or, if we don't have any memory objects, we need to allocate one.

Each buffer's usage may have a different memory requirement, which is relevant when we want to allocate a memory object and bind it to the buffer. Here is a code sample that allocates a memory object for a given buffer:

```
VkMemoryRequirements buffer_memory_requirements;
vkGetBufferMemoryRequirements( GetDevice(), buffer, &buffer_memory_requirements );

VkPhysicalDeviceMemoryProperties memory_properties;
vkGetPhysicalDeviceMemoryProperties( GetPhysicalDevice(), &memory_properties );

for ( uint32_t i = 0; i < memory_properties.memoryTypeCount; ++i ) {
```

```
    if ( (buffer_memory_requirements.memoryTypeBits & (1 << i)) &&
         ((memory_properties.memoryTypes[i].propertyFlags & property) == property) ) {

      VkMemoryAllocateInfo memory_allocate_info = {
        VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO, // VkStructureType sType
        nullptr, // const void *pNext
        buffer_memory_requirements.size, // VkDeviceSize allocationSize
        i // uint32_t memoryTypeIndex
      };

      if ( vkAllocateMemory( GetDevice(), &memory_allocate_info, nullptr, memory ) == VK_SUCCESS ) {
        return true;
      }
    }
  }
  return false;
```

3.Tutorial05.cpp, function AllocateBufferMemory()

Similarly to the code in Part 4, we first check what the memory requirements for a given buffer are. After that we check the properties of a memory available in a given physical device. It contains information about the number of memory heaps and their capabilities.

Next we iterate over each available memory type and check if it is compatible with the requirement queried for a given buffer. We also check if a given memory type supports our additional, requested properties, for example, whether a given memory type is host-visible. When we find a match, we fill in a VkMemoryAllocateInfo structure and call a vkAllocateMemory() function.

The allocated memory object is then bound to our buffer, and from now on we can safely use this buffer in our application.

Vertex Buffer Creation

The first buffer we want to create is a vertex buffer. It stores data for vertex attributes that are used during rendering. In this example we store position and color for four vertices of a quad. The most important change from the previous tutorial is the use of a device-local memory instead of a host-visible memory. Device-local memory is much faster, but we can't copy any data directly from the application to device-local memory. We need to use a staging buffer, from which we copy data to the vertex buffer.

We also need to specify two different usages for this buffer. The first is a vertex buffer usage, which means that we want to use the given buffer as a vertex buffer from which data for the vertex attributes will be fetched. The second is transfer dst usage, which means that we will copy data to this buffer. It will be used as a destination of any transfer (copy) operation.

The code that creates a buffer with all these requirements looks like this:

```
const std::vector< float >& vertex_data = GetVertexData();

Vulkan.VertexBuffer.Size = static_cast <uint32_t>(vertex_data.size() * sizeof(vertex_data[0]));
if ( !CreateBuffer( VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, Vulkan.VertexBuffer ) ) {
  std::cout << "Could not create vertex buffer!" << std::endl;
  return false;
}

return true;
```

4.Tutorial05.cpp, function CreateVertexBuffer()

At the beginning we get the vertex data (hard-coded in a GetVertexData() function) to check how much space we need to hold values for all our vertices. After that we call a CreateBuffer() function presented earlier to create a vertex buffer and bind a device-local memory to it.

Staging Buffer Creation

Next we can create an intermediate staging buffer. This buffer is not used during the rendering process so it can be bound to a slower, host-visible memory. This way we can map it and copy data directly from the application. After that we can copy data from the staging buffer to any other buffer (or even image) that is bound to device-local memory. This way all resources that are used for rendering purposes are bound to the fastest available memory. We just need additional operations for the data transfer.

Here is a code that creates a staging buffer:

```
Vulkan.StagingBuffer.Size = 4000;
if ( !CreateBuffer( VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT, Vulkan.StagingBuffer ) ) {
   std::cout << "Could not staging buffer!" << std::endl;
   return false;
}

return true;
```

5.Tutorial04.cpp, function CreateStagingBuffer()

We will copy data from this buffer to other resources, so we must specify a transfer src usage for it (it will be used as a source for transfer operations). We would also like to map it to be able to directly copy any data from the application. For this we need to use a host-visible memory and that's why we specify this memory property. The buffer's size is chosen arbitrarily, but should be large enough to be able to hold vertex data. In real-life scenarios we should try to reuse the staging buffer as many times as possible, in many cases, so its size should be big enough to cover most of data transfer operations in our application. Of course, if we want to do many transfer operations at the same time, we have to create multiple staging buffers.

Copying Data between Buffers

We have created two buffers: one for the vertex attributes data and the other to act as an intermediate buffer. Now we need to copy data from the CPU to the GPU. To do this we need to map the staging buffer and acquire a pointer that we can use to upload data to the graphics hardware's memory. After that we need to record and submit a command buffer that will copy the vertex data from the staging buffer to the vertex buffer. And as all of our command buffers used for virtual frames and rendering are marked as short lived, we can safely use one of them for this operation.

First let's see what our data for vertex attributes looks like:

```
static const std::vector< float > vertex_data = {
   -0.7f, -0.7f, 0.0f, 1.0f,
   1.0f, 0.0f, 0.0f, 0.0f,
   //
   -0.7f, 0.7f, 0.0f, 1.0f,
   0.0f, 1.0f, 0.0f, 0.0f,
   //
   0.7f, -0.7f, 0.0f, 1.0f,
   0.0f, 0.0f, 1.0f, 0.0f,
   //
   0.7f, 0.7f, 0.0f, 1.0f,
   0.3f, 0.3f, 0.3f, 0.0f
};

return vertex_data;
```

6.Tutorial05.cpp, function GetVertexData()

It is a simple, hard-coded array of floating point values. Data for each vertex contains four components for position attribute and four components for color attribute. As we render a quad, we have four pairs of such attributes.

Here is the code that copies data from the application to the staging buffer and after that from the staging buffer to the vertex buffer:

```
// Prepare data in a staging buffer
const std::vector< float >& vertex_data = GetVertexData();

void *staging_buffer_memory_pointer;
if ( vkMapMemory( GetDevice(), Vulkan.StagingBuffer.Memory, 0, Vulkan.VertexBuffer.Size, 0, &staging_buffer_memory_pointer) != VK_SUCCESS ) {
  std::cout << "Could not map memory and upload data to a staging buffer!" << std::endl;
  return false;
}

memcpy ( staging_buffer_memory_pointer, &vertex_data[0], Vulkan.VertexBuffer.Size );

VkMappedMemoryRange flush_range = {
  VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE, // VkStructureType sType
  nullptr, // const void *pNext
  Vulkan.StagingBuffer.Memory, // VkDeviceMemory memory
  0, // VkDeviceSize offset
  Vulkan.VertexBuffer.Size // VkDeviceSize size
};
vkFlushMappedMemoryRanges( GetDevice(), 1, &flush_range );

vkUnmapMemory( GetDevice(), Vulkan.StagingBuffer.Memory );

// Prepare command buffer to copy data from staging buffer to a vertex buffer
```

```
VkCommandBufferBeginInfo command_buffer_begin_info = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO, // VkStructureType sType
    nullptr, // const void *pNext
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT, // VkCommandBufferUsageFlags flags
    nullptr // const VkCommandBufferInheritanceInfo *pInheritanceInfo
};

VkCommandBuffer command_buffer = Vulkan.RenderingResources[0].CommandBuffer;

vkBeginCommandBuffer( command_buffer, &command_buffer_begin_info);

VkBufferCopy buffer_copy_info = {
    0, // VkDeviceSize srcOffset
    0, // VkDeviceSize dstOffset
    Vulkan.VertexBuffer.Size // VkDeviceSize size
};
vkCmdCopyBuffer( command_buffer, Vulkan.StagingBuffer.Handle, Vulkan.VertexBuffer.Handle, 1, &buffer_copy_info );

VkBufferMemoryBarrier buffer_memory_barrier = {
    VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER, // VkStructureType sType;
    nullptr, // const void *pNext
    VK_ACCESS_MEMORY_WRITE_BIT, // VkAccessFlags srcAccessMask
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT, // VkAccessFlags dstAccessMask
    VK_QUEUE_FAMILY_IGNORED, // uint32_t srcQueueFamilyIndex
    VK_QUEUE_FAMILY_IGNORED, // uint32_t dstQueueFamilyIndex
    Vulkan.VertexBuffer.Handle, // VkBuffer buffer
```

```
    0, // VkDeviceSize offset
    VK_WHOLE_SIZE // VkDeviceSize size
};
vkCmdPipelineBarrier( command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, 0, 0, nullptr, 1, &buffer_memory_barrier, 0,
nullptr );

vkEndCommandBuffer( command_buffer );

// Submit command buffer and copy data from staging buffer to a vertex buffer
VkSubmitInfo submit_info = {
    VK_STRUCTURE_TYPE_SUBMIT_INFO, // VkStructureType sType
    nullptr, // const void *pNext
    0, // uint32_t waitSemaphoreCount
    nullptr, // const VkSemaphore *pWaitSemaphores
    nullptr, // const VkPipelineStageFlags *pWaitDstStageMask;
    1, // uint32_t commandBufferCount
    &command_buffer, // const VkCommandBuffer *pCommandBuffers
    0, // uint32_t signalSemaphoreCount
    nullptr // const VkSemaphore *pSignalSemaphores
};

if ( vkQueueSubmit( GetGraphicsQueue().Handle, 1, &submit_info, VK_NULL_HANDLE ) != VK_SUCCESS ) {
    return false;
}

vkDeviceWaitIdle( GetDevice() );
```

return true;

7.Tutorial05.cpp, function CopyVertexData()

At the beginning, we get vertex data and map the staging buffer's memory by calling the vkMapMemory() function. During the call, we specify a handle of a memory that is bound to a staging buffer, and buffer's size. This gives us a pointer that we can use in an ordinary memcpy() function to copy data from our application to graphics hardware.

Next we flush the mapped memory to tell the driver which parts of a memory object were modified. We can specify multiple ranges of memory if needed. We have one memory area that should be flushed and we specify it by creating a variable of type VkMappedMemoryRange and by calling a vkFlushMappedMemoryRanges() function. After that we unmap the memory, but we don't have to do this. We can keep a pointer for later use and this should not affect the performance of our application.

Next we start preparing a command buffer. We specify that it will be submitted only once before it will be reset. We fill a VkCommandBufferBeginInfo structure and provide it to a vkBeginCommandBuffer() function.

Now we perform the copy operation. First a variable of type VkBufferCopy is created. It contains the following fields:

srcOffset – Offset in bytes in a source buffer from which we want to copy data.
dstOffset – Offset in bytes in a destination buffer into which we want to copy data.
size – Size of the data (in bytes) we want to copy.

We copy data from the beginning of a staging buffer and to the beginning of a vertex buffer, so we specify zero for both offsets. The size of the vertex buffer was calculated based on the hard-coded vertex data, so we copy the same number of bytes. To copy data from one buffer to another, we call a vkCmdCopyBuffer() function.

Setting a Buffer Memory Barrier

We have recorded a copy operation but that's not all. From now on we will not use the buffer as a target for transfer operations but as a vertex buffer. We need to tell the driver that the type of buffer's memory access will change and from now on it will serve as a source of data for vertex attributes. To do this we set a memory barrier similarly to what we have done earlier in case of swapchain images.

First we prepare a variable of type VkBufferMemoryBarrier, which contains the following members:

sType – Standard structure type, here set to VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER.
pNext – Parameter reserved for extensions.
srcAccessMask – Types of memory operations that were performed on this buffer before the barrier.
dstAccessMask – Memory operations that will be performed on a given buffer after the barrier.
srcQueueFamilyIndex – Index of a queue family that accessed the buffer before.
dstQueueFamilyIndex – Queue family that will access the buffer from now on.
buffer – Handle to the buffer for which we set up a barrier.
offset – Memory offset from the start of the buffer (from the memory's base offset bound to the buffer).
size – Size of the buffer's memory area for which we want to setup a barrier.
As you can see, we can set up a barrier for a specific range of buffer's memory. But here we do it for the whole buffer, so we specify an offset of 0 and the VK_WHOLE_SIZE enum for the size. We don't want to transfer ownership between different queue families, so we use VK_QUEUE_FAMILY_IGNORED enum both for srcQueueFamilyIndex and dstQueueFamilyIndex.

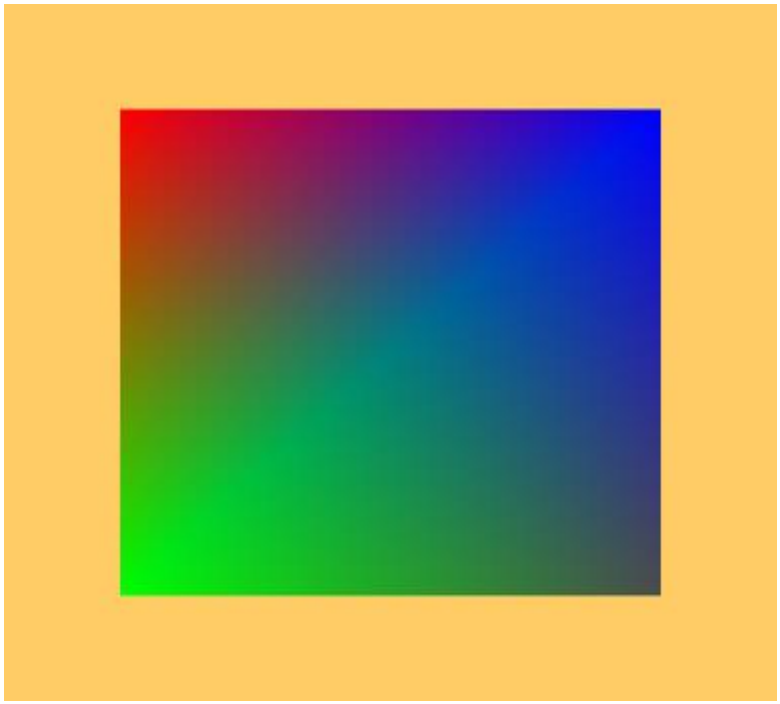
The most important parameters are srcAccessMask and dstAccessMask. We have copied data from the staging buffer to a vertex buffer. So before the barrier, the vertex buffer was used as a destination for transfer operations and its memory was written to. That's why we have specified VK_ACCESS_MEMORY_WRITE_BIT for a srcAccessMask field. But after that the barrier buffer will be used only as a source of data for vertex attributes. So for dstAccessMask field we specify VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT.

To set up a barrier we call a vkCmdPipelineBarrier() function. And to finish command buffer recording, we call vkEndCommandBuffer(). Next, for all of the above operations to execute, we submit a command buffer by calling vkQueueSubmit() function.

Normally during the command buffer submission, we should provide a fence. It is signaled once all transfer operations (and whole command buffer) are finished. But here, for the sake of simplicity, we call vkDeviceWaitIdle() and wait for all operations executed on a given device to finish. Once all operations complete, we have successfully transferred data to the device-local memory and we can use the vertex buffer without worrying about performance loss.

Tutorial05 Execution

The results of the rendering operations are exactly the same as in Part 4:



We render a quad that has different colors in each corner: red, green, dark gray, and blue. The quad should adjust its size (and aspect) to match window's size and shape.

Cleaning Up

In this part of the tutorial, I have also refactored the cleaning code. We have created two buffers, each with a separate memory object. To avoid code redundancy, I prepared a buffer cleaning function:

```
if ( buffer.Handle != VK_NULL_HANDLE ) {
   vkDestroyBuffer( GetDevice(), buffer.Handle, nullptr );
   buffer.Handle = VK_NULL_HANDLE;
}

if ( buffer.Memory != VK_NULL_HANDLE ) {
   vkFreeMemory( GetDevice(), buffer.Memory, nullptr );
   buffer.Memory = VK_NULL_HANDLE;
}
```

8.Tutorial05.cpp, function DestroyBuffer()

This function checks whether a given buffer was successfully created, and if so it calls a vkDestroyBuffer() function. It also frees memory associated with a given buffer through a vkFreeMemory() function call. The DestroyBuffer() function is called in a destructor that also releases all other resources relevant to this part of the tutorial:

```
if ( GetDevice() != VK_NULL_HANDLE ) {
   vkDeviceWaitIdle( GetDevice() );

   DestroyBuffer( Vulkan.VertexBuffer );
```

```
DestroyBuffer( Vulkan.StagingBuffer );

if ( Vulkan.GraphicsPipeline != VK_NULL_HANDLE ) {
    vkDestroyPipeline( GetDevice(), Vulkan.GraphicsPipeline, nullptr );
    Vulkan.GraphicsPipeline = VK_NULL_HANDLE;
}

if ( Vulkan.RenderPass != VK_NULL_HANDLE ) {
    vkDestroyRenderPass( GetDevice(), Vulkan.RenderPass, nullptr );
    Vulkan.RenderPass = VK_NULL_HANDLE;
}

for ( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if ( Vulkan.RenderingResources[i].Framebuffer != VK_NULL_HANDLE ) {
        vkDestroyFramebuffer( GetDevice(), Vulkan.RenderingResources[i].Framebuffer, nullptr );
    }
    if ( Vulkan.RenderingResources[i].CommandBuffer != VK_NULL_HANDLE ) {
        vkFreeCommandBuffers( GetDevice(), Vulkan.CommandPool, 1, &Vulkan.RenderingResources[i].CommandBuffer );
    }
    if ( Vulkan.RenderingResources[i].ImageAvailableSemaphore != VK_NULL_HANDLE ) {
        vkDestroySemaphore( GetDevice(), Vulkan.RenderingResources[i].ImageAvailableSemaphore, nullptr );
    }
    if ( Vulkan.RenderingResources[i].FinishedRenderingSemaphore != VK_NULL_HANDLE ) {
        vkDestroySemaphore( GetDevice(), Vulkan.RenderingResources[i].FinishedRenderingSemaphore, nullptr );
    }
```

```
      if ( Vulkan.RenderingResources[i].Fence != VK_NULL_HANDLE ) {
        vkDestroyFence( GetDevice(), Vulkan.RenderingResources[i].Fence, nullptr );
      }
    }

    if ( Vulkan.CommandPool != VK_NULL_HANDLE ) {
      vkDestroyCommandPool( GetDevice(), Vulkan.CommandPool, nullptr );
      Vulkan.CommandPool = VK_NULL_HANDLE;
    }
}
```

9.Tutorial05.cpp, destructor

First we wait for all the operations performed by the device to finish. Next we destroy the vertex and staging buffers. After that we destroy all other resources in the order opposite to their creation: graphics pipeline, render pass, and resources for each virtual frame, which consists of a framebuffer, command buffer, two semaphores, a fence, and a framebuffer. Finally we destroy a command pool from which command buffers were allocated.

Conclusion

In this tutorial we used the recommended technique for transferring data from the application to the graphics hardware. It gives the best performance for the resources involved in the rendering process and the ability to map and copy data from the application to the staging buffer. We only need to prepare an additional command buffer recording and submission to transfer data from one buffer to another.

Using staging buffers is recommended for more than just copying data between buffers. We can use the same approach to copy data from a buffer to images. And the next part of the tutorial will show how to do this by presenting the descriptors, descriptor sets, and descriptor layouts, which are another big part of the Vulkan API.

翻译：黄威（横写、意气风发） 审校：周行健（Purple Tulip）

大家好，在本篇教程我们将会学习 Vulkan 相关知识，我将重点讲述在电脑上渲染一个三角形需要的步骤与代码。
首先我要告诉你们，这不是一个渲染 API 的新手教程。我认为对于 API，你们已经有一定经验，例如 OpenGL 或是 DirectX 的使用经验，而且，你读这篇文章只是为了了解 Vulkan 的特性。我写这篇“教程”的主要目标就是做一个简短而又完整的 C 语言程序，让这个程序在 Windows 上使用 Vulkan 来运行一个绘图管线 （如果我有时间的话，我也会写一篇关于在 linux 上运行相似程序的文章。如果你对于我在进行中的工作感兴趣，你可以点击这里查看进展） 。
好了，让我们开始吧。

准备工作
在这篇文章中，我会把所有的代码都列出来。我会随着学习的过程把代码一步步列出来，所以你可以在教程中看到每一段代码。如果你想续写并且编译这些代码，你可以复制以下链接：
git clone https://bitbucket.org/jose_henriques/vulkan_tutorial.git
我已经在 windows7 和 windows10 上用 Visual Studio 2013 成功编译并运行了每一段指令。由于我不使用集成开发环境（IDE）（我只使用编译程序），我提供了一个批处理文件（build.bat），你们可以用它来完成代码的编译。但如果你想要从你的控制台调用批处理文件，在你的路径里需要有 CL 编译器。[你需要寻找并运行用来设置的 vcvars*.bat 。对于我正在使用的设置，你可以在"C:\ProgramFiles(x86)\MicrosoftVisualStudio12.0\VC\bin\x86_amd64\vcvarsx86_amd64.bat"处找到]
以下每一步我都会指出操作指令，你可以检查并编译这些代码，也可以自行修改。例如，要想获得平台画面代码的最初操作指令，你可以按以下内容进行操作：
git checkout 39534dc
先说一件非常重要的事情，那就是我们不会在这篇教程中尝试完成的事情。首先，我不会创建一个让你使用并开始编写下一个引擎的“画面”…我甚至不会创建一个重复好几次的函数代码。我认为，相对于通过一些间接的方法获得全部代码，直接参与全部代码的编写过程更有价值，对于一个教程来说更是如此。
如果我们已经能够在屏幕上得到一个带有绘图管线，并且有一个顶点及片段着色器在运行的三角形，这个教程就可以告一段落了。我以后也许会写一些其它主题的教程，但不是在这里！
如果这个代码能够给你带来任何价值的话，你可以免费使用它。不过我认为只有对于学习 API 的人来说这些代码才有价值。

Windows 平台代码

以下就是一个典型的注册并打开新窗口的 Windows 平台代码。如果你对此很熟悉，那么你可以选择跳过这一段内容。在我们的渲染完成之前，我都将从这些最基础的设置和添加开始讲。[抱歉，在此我不会解释这些代码。]

```
1       #include <windows.h>
2
3       LRESULT CALLBACK WindowProc( HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam ) {
4          switch( uMsg ) {
5             case WM_CLOSE: {
6                PostQuitMessage( 0 );
7                break;
8             }
9             default: {
10               break;
11            }
12         }
13
14         // a pass-through for now. We will return to this callback
15         return DefWindowProc( hwnd, uMsg, wParam, lParam );
16      }
17
18      int CALLBACK WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow ) {
19
20         WNDCLASSEX windowClass = {};
21         windowClass.cbSize = sizeof(WNDCLASSEX);
22         windowClass.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
23         windowClass.lpfnWndProc = WindowProc;
```

```
24      windowClass.hInstance = hInstance;
25      windowClass.lpszClassName = "VulkanWindowClass";
26      RegisterClassEx( &windowClass );
27
28      HWND windowHandle = CreateWindowEx( NULL, "VulkanWindowClass", "Core",
29                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
30                              100,
31                              100,
32                              800,    // some random values for now.
33                              600,    // we will come back to these soon.
34                              NULL,
35                              NULL,
36                              hInstance,
37                              NULL );
38
39      MSG msg;
40      bool done = false;
41      while( !done ) {
42        PeekMessage( &msg, NULL, NULL, NULL, PM_REMOVE );
43        if( msg.message == WM_QUIT ) {
44          done = true;
45        } else {
46          TranslateMessage( &msg );
47          DispatchMessage( &msg );
48        }
49
```

```
50              RedrawWindow( windowHandle, NULL, NULL, RDW_INTERNALPAINT );
51          }
52
53          return msg.wParam;
54      }</windows.h>
```

如果你得到了这个链接并对指令进行检查，那么你应该能够调用批处理文件（build.bat）来编译代码。如果你只是想复制/粘贴这个代码并自行编译，以下就是这个 bat 文件的内容：

```
1       @echo off
2
3       mkdir build
4       pushd build
5       cl /Od /Zi ..\main.cpp user32.lib
6       popd
```

这将会编译我们的测试程序，并在你的<project>文件夹中（如果你的电脑没有，则会自动创建）创建一个叫“main.exe”的二进制文件。如果你运行了这个应用程序，你就会在（800，600）大小的窗口中（100，100）位置得到一个白色小窗口，你可以随意关闭它。这是用来编写平台代码的…不过…在我们着手平台代码之前我们仍需要做一些设置。

Vulkan 动态加载

好了，现在我们需要开始讨论如何才能把 Vulkan 装到我们的电脑上…Khronos 和 LunarG 并没有表明你是否需要他们的 SDK（软件开发工具包）。简单的来说，答案就是不要，你不需要用他们的软件开发工具包进行 Vulkan 应用程序的编写。在后面的章节中，我会向你展示，即使是对于验证层级，如果你想要这么做的的话，那你也可以跳过软件开发工具包。

所以我们需要两个东西：数据库和标头。你的系统中应该已经内置了库，因为它由你的显卡驱动程序提供。在 Windows 中它叫做 vulkan-1.dll(在 linux 中叫做 ibvulkan.so.1) 并且它应该就在你的系统文件夹中。

Khronos 说他们提供的有加载器或驱动程序的标头就已经足够了。我在我的电脑中没有找到它们，所以我从 Khronos 的注册表中 Vulkan 说明文件中的链接中得到了它们：

git clone https://github.com/KhronosGroup/Vulkan-Docs.git

我发现我还需要以下链接：

git clone https://github.com/KhronosGroup/Vulkan-LoaderAndValidationLayers.git

我们随后才会需要用到它。但是现在，只要把 vulkan.h 和 vk_platform.h 复制到你的应用程序文件夹。（如果你想用这些 GIT 的链接，我把它放到了这个 commit 中。）

我们已经有了 Vulkan.h，那么现在我们就要开始加载我们需要的 API 函数了。我们将会动态加载 Vulkan 函数，而且我们想要确保我们在使用 Windows 平台特有的定义。所以，我们将会加入以下代码：

```
1       #define VK_USE_PLATFORM_WIN32_KHR
2       #define VK_NO_PROTOTYPES
3       #include "vulkan.h"
```

对于每一个我们想要使用的 Vulkan 函数，我们都要先在动态数据库中对它们进行定义（declare）并加载（load）。这个过程依赖于平台来进行。让我们现在来做一个 **win32_LoadVulkan()**函数。

特别要注意的是，对于我们需要调用的每一个函数，我们都需要在其中加入类似于 **vkCreateInstance()** 的加载代码。

```
1       PFN_vkCreateInstance vkCreateInstance = NULL;
2
3       void win32_LoadVulkan( ) {
4
5          HMODULE vulkan_module = LoadLibrary( "vulkan-1.dll" );
6          assert( vulkan_module, "Failed to load vulkan module." );
7
8          vkCreateInstance = (PFN_vkCreateInstance) GetProcAddress( vulkan_module, "vkCreateInstance" );
9          assert( vkCreateInstance, "Failed to load vkCreateInstance function pointer." );
10
11      }
```

我还写了一个辅助函数 **assert()**来做一些你希望它们完成的工作。这将会是我们的调试工具**!:**）（对于这个函数，你随意使用你喜欢的版本就可以了。）

```
1       void assert( bool flag, char *msg = "" ) {
2
3          if( !flag ) {
4             OutputDebugStringA( "ASSERT: " );
5             OutputDebugStringA( msg );
6             OutputDebugStringA( "\n" );
7             int *base = 0;
8             *base = 1;
9          }
10
11      }
```

这就是我们要做的所有的 Windows 特定设置了。接下来我们开始讨论 Vulkan 本身以及它的一些特点。

创建一个 Vulkan 实例

用一句话总结 Vulkan 的数据结构和它们的用途：以通用机制对它们进行填充，它们的主要作用就是管理功能参数。接下来我给大家举个例子：

```
1       VkApplicationInfo applicationInfo;
2       applicationInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO; // sType is a member of all structs
3       applicationInfo.pNext = NULL;                       // as is pNext and flag
4       applicationInfo.pApplicationName = "First Test";          // The name of our application
5       applicationInfo.pEngineName = NULL;                    // The name of the engine
6       applicationInfo.engineVersion = 1;                     // The version of the engine
7       applicationInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);      // The version of Vulkan we're using
```

现在，如果我们注意到 **VkApplicationInfo** 的细节，我们会发现这些字段大多数都是零。在任何情况下，**.sType** 的值都是已知的（通常值为

VK_STRUCTURE_TYPE_<uppercase_structure_name>）。在本教程中，对于大多数我们用来填充数据结构的值，我都会尽量讲得清楚一些，我可能让某些值保持为 0，因为我总是会把事情这样处理：

```
VkApplicationInfo applicationInfo = { };    // notice me senpai!
applicationInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
applicationInfo.pApplicationName = "First Test";
applicationInfo.engineVersion = 1;
applicationInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);
```

接下来，几乎所有的函数都会回到 **VkResult** 枚举值表。所以，让我们用炫酷的调试工具来写一个简单的辅助：

```
void checkVulkanResult( VkResult &result, char *msg ) {
    assert( result == VK_SUCCESS, msg );
}
```

在我们创建绘图管线期间，我们会设置很多的状态，创建或是初始化许多的“上下文”（context）。为了帮助我们跟踪所有的 Vulkan 状态，我们编写了以下代码：

```
struct vulkan_context {

    uint32_t width;
    uint32_t height;

    VkInstance instance;
}

vulkan_context context;
```

上下文将会慢慢扩展…但是现在让我们继续。你可能会注意到，我在上下文中调用了一个实例。实际上 Vulkan 根本没有全局状态。当 Vulkan 需要一些应用程序状态时，你就需要发送你的 **VkInstance**。对于许多结构都是这样的，包括我们的绘图管线。这只是我们需要创建、初始化并保持的内容中的一个。

所以让我们继续吧。

由于这个过程会重复调用几乎所有的函数，我将会详细讲一下这个实例。所以，检查一下细节，然后来写出一个我们需要调用的 **VkInstance**：

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

注意分配器：从经验上来说，当函数请求 **pAllocator** 时，你可以将其定为 Null 值，这样 Vulkan 就会使用默认分配器。我在本教程中不会将自定义分配器作为一个主题。你只要学会用默认分配器并且知道 Vulkan 允许你的应用程序控制 Vulkan 的内存分配就已经够了。

现在我要谈的就是那些需要你来填数据结构的函数，通常来说就像 **Vk*CreateInfo**，把值提交给 Vulkan 函数，这样的话 **vkCreateInstance**，就会传回它最后的参数结果。

```
1       VkInstanceCreateInfo instanceInfo = { };
2       instanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
3       instanceInfo.pApplicationInfo = &applicationInfo;
4       instanceInfo.enabledLayerCount = 0;
5       instanceInfo.ppEnabledLayerNames = NULL;
6       instanceInfo.enabledExtensionCount = 0;
7       instanceInfo.ppEnabledExtensionNames = NULL;
8
9       result = vkCreateInstance( &instanceInfo, NULL, &context.instance );
10      checkVulkanResult( result, "Failed to create vulkan instance." );
```

你可以编译并运行这段代码，然而什么事情都不会发生…我们需要用我们可能想用的验证层，以及要求使用的一些扩展来填充实例信息。相对于一个白窗口，只有这样我们才能做更多有趣的事情。

验证层级

效率是 Vulkan 的核心原则之一。那么问题就在于此，Vulkan 基本不会进行任何的验证与错误检查！当你的操作出现错误，Vulkan 确实会发生崩溃或是发生未定义的行为。这些都还好，但是当我们开发应用程序时，我们想要知道为什么我们的应用程序没有显示预期的内容，或是发生崩溃时，它为什么会崩溃。

进入验证层级。
Vulkan 是一个分层型 API。我们调用时有一个核心层级，但是在调用 API 中和加载其他“层级”时可能会拦截 API 调用。在这里我们感兴趣的就是验证层级，它可以在我们使用 API 时帮助我们调试与追踪发生的问题。
在你开发应用程序时，你会想要启用这一层级，但是在运行应用程序时记得禁用它们。
为了找到我们的载入程序能够识别的层级，我们需要调用以下代码：

```
1       uint32_t layerCount = 0;
2       vkEnumerateInstanceLayerProperties( &layerCount, NULL );
3
4       assert( layerCount != 0, "Failed to find any layer in your system." );
5
6       VkLayerProperties *layersAvailable = new VkLayerProperties[layerCount];
7       vkEnumerateInstanceLayerProperties( &layerCount, layersAvailable );
```

（别忘了在顶部添加申明，以及给 **win32_LoadVulkan()**函数加载 **vkEnumerateInstanceLayerProperties**）
这是另外一个循环机制。我们调用这个函数两次。第一次我们把 Null 作为参数发送给 **VkLayerProperties** 来查询层数。接下来我们分配必要的空间来支持大量的项，然后我们第二次调用这个函数来填充我们的数据结构。
如果你运行这段代码，你会注意到你可能没有发现任何层级…这是因为在我系统的渠道中，加载程序不能够找到任何层级。为了得到验证层，我们需要软件开发工具包并且（或是）在 **Vulkan-LoaderAndValidationLayers.git** 中编译代码。
在我试图找出是否需要软件开发工具包的过程中，我发现了你只需要在你的程序文件夹中找出你想要层级的 ***.json** 和***.dll**，然后你可以用这些文件在文件夹路径安装 **Vulkan-LoaderAndValidationLayers.git**. 环境变量。相比用软件开发工具包在注册表项 **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Khronos\Vulkan\ExplicitLayers** 中设置层级信息的不明不白的方法，我更喜欢这种方法，因为这种方法你可以更好地控制你的应用程序加载哪些内容。（我在想这是否能够提高安全等级？）
接下来我们将要使用的层级是 **VK_LAYER_LUNARG_standard_validation**。这个层级相当于其它层级的一种超级集合。[它来源于软件开发工具包]。所以，我假定你已经安装了软件开发工具包，或是你已经将所有你想要用的 **VkLayer_*.dll** 和 **VkLayer_*.json** 文件移动到了层级文件夹和 **set VK_LAYER_PATH=/path/to/layers/folder** 中。
只要保证我们找到了层级并且用以下信息配置了实例，我们现在就可以完成验证层级部分了：

```
1       bool foundValidation = false;
```

```
2       for( int i = 0; i < layerCount; ++i ) {
3         if( strcmp( layersAvailable[i].layerName, "VK_LAYER_LUNARG_standard_validation" ) == 0 ) {
4             foundValidation = true;
5         }
6       }
7       assert( foundValidation, "Could not find validation layer." );
8       const char *layers[] = { "VK_LAYER_LUNARG_standard_validation" };
9
10
11
12      // update the VkInstanceCreateInfo with:
13      instanceInfo.enabledLayerCount = 1;
14      instanceInfo.ppEnabledLayerNames = layers;
```

不幸的是，这一操作还是会产生和之前相同的结果。我们需要处理一些扩展项来输出一些调试信息。

扩展

就像在 OpenGL 和其他 API 一样，扩展可以给 Vulkan 添加一些不属于 API 核心部分的新功能。

我们需要 **VK_EXT_debug_report** 扩展才能开始调试我们的应用程序。接下来的代码和层级加载代码很像，不过显著的区别就在于我们正在寻找三个特殊的扩展。我先写出我们之后需要用到的另外两个扩展，不过我们现在不必考虑它们。

```
1       uint32_t extensionCount = 0;
2       vkEnumerateInstanceExtensionProperties( NULL, &extensionCount, NULL );
3       VkExtensionProperties *extensionsAvailable = new VkExtensionProperties[extensionCount];
4       vkEnumerateInstanceExtensionProperties( NULL, &extensionCount, extensionsAvailable );
5
```

```
6       const char *extensions[] = { "VK_KHR_surface", "VK_KHR_win32_surface", "VK_EXT_debug_report" };
7       uint32_t numberRequiredExtensions = sizeof(extensions) / sizeof(char*);
8       uint32_t foundExtensions = 0;
9       for( uint32_t i = 0; i < extensionCount; ++i ) {
10        for( int j = 0; j < numberRequiredExtensions; ++j ) {
11          if( strcmp( extensionsAvailable[i].extensionName, extensions[j] ) == 0 ) {
12            foundExtensions++;
13          }
14        }
15      }
16      assert( foundExtensions == numberRequiredExtensions, "Could not find debug extension" );
```

扩展新增了三个新函数：**vkCreateDebugReportCallbackEXT()**，**vkDestroyDebugReportCallbackEXT()**和 **vkDebugReportMessageEXT()**。由于这些功能不是 Vulkan API 的核心部分，我们不能像加载其他函数一样加载它们。我们需要使用 **vkGetInstanceProcAddr()**。只要我们把那个函数添加到 **win32_LoadVulkan()**，我们就可以定义另外一个看起来很熟悉的函数：

```
1       PFN_vkCreateDebugReportCallbackEXT vkCreateDebugReportCallbackEXT = NULL;
2       PFN_vkDestroyDebugReportCallbackEXT vkDestroyDebugReportCallbackEXT = NULL;
3       PFN_vkDebugReportMessageEXT vkDebugReportMessageEXT = NULL;
4
5       void win32_LoadVulkanExtensions( vulkan_context &context ) {
6
7         *(void **)&vkCreateDebugReportCallbackEXT = vkGetInstanceProcAddr( context.instance,
8                                     "vkCreateDebugReportCallbackEXT" );
9         *(void **)&vkDestroyDebugReportCallbackEXT = vkGetInstanceProcAddr( context.instance,
10                                    "vkDestroyDebugReportCallbackEXT" );
11        *(void **)&vkDebugReportMessageEXT = vkGetInstanceProcAddr( context.instance,
```

```
12                                  "vkDebugReportMessageEXT" );
13      }
```

扩展能够给我们提供一个能够提供所有调试信息的回调函数。下面就是我们的回调函数：

```
1       VKAPI_ATTR VkBool32 VKAPI_CALL MyDebugReportCallback( VkDebugReportFlagsEXT flags,
2         VkDebugReportObjectTypeEXT objectType, uint64_t object, size_t location,
3         int32_t messageCode, const char* pLayerPrefix, const char* pMessage, void* pUserData ) {
4
5         OutputDebugStringA( pLayerPrefix );
6         OutputDebugStringA( " " );
7         OutputDebugStringA( pMessage );
8         OutputDebugStringA( "\n" );
9         return VK_FALSE;
10      }
```

没有什么复杂的部分，因为我们只需要知道层级的信息来源和信息本身。

我还没有谈论过这部分内容，但是我经常会对 Visual Studio 进行调试。我曾说过我不用 IDE（集成开发环境），但是对于调试来说，我别无选择。不过我只是用 **devenv .\build\main.exe** 开启调试会话。你可能会需要加载主要的.cpp 文件，然后你就要开始设置断点、观察等等。

要加载我们的 Vulkan 扩展函数、注册回调函数并在应用的最后停止它，唯一要做的事情就是添加调用语句。（注意，我们可以使用调用语句来控制我们想要的报告类型。我们还给我们的 vulkan_context 结构加入了 **VkDebugReportCallbackEXT** 部分。）

```
1       win32_LoadVulkanExtensions( context );
2
3       VkDebugReportCallbackCreateInfoEXT callbackCreateInfo = { };
4       callbackCreateInfo.sType = VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT;
5       callbackCreateInfo.flags =   VK_DEBUG_REPORT_ERROR_BIT_EXT |
6                                    VK_DEBUG_REPORT_WARNING_BIT_EXT |
7                                    VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT;
8       callbackCreateInfo.pfnCallback = &MyDebugReportCallback;
```

```
9       callbackCreateInfo.pUserData = NULL;
10
11      result = vkCreateDebugReportCallbackEXT( context.instance, &callbackCreateInfo,
12                                      NULL, &context.callback );
13      checkVulkanResult( result, "Failed to create degub report callback." );
```

结束之后，我们可以对数据进行清理：

```
vkDestroyDebugReportCallbackEXT( context.instance, context.callback, NULL );
```

所以，我们准备开始创建渲染界面了，但在那之前我还要再解释两个额外的扩展。

设备

对于设置后台窗口渲染，我们已经做好了一切准备。现在我们需要进行表面渲染，那么我们就需要找出我们机器的哪个物理设备可以支持表面渲染。因此我们使用了我们藏在实例中的两个额外扩展函数：**VK_KHR_surface** 和 **VK_KHR_win32_surface**。**VK_KHR_surface**，它们在每个系统中都可以运行，因为它以显示本地窗口的方式将每个平台都抽象化了。那么另一个扩展的功能是什么呢，它可以在一个特定系统中创建 **VkSurface** 文件。对于 Windows 系统来说，那个文件就是 **VK_KHR_win32_surface**。

在那之前，我要简单说一下关于物理和逻辑设备，还有工作队列。物理设备指你系统中的单一的图形处理器。你的电脑中可以有很多的物理设备。逻辑设备一般用于应用程序追踪物理设备的使用。每一个物理设备支持的队列数量及类型都由它本身决定。（可以参考计算和图形队列。）我们要做的就是枚举我们系统中的物理设备，然后从中选中我们想要用的一个物理设备。在本篇教程中，我们将会选择我们找到的第一个有一个图形队列的物理设备，它可以呈现出我们的效果图…如果我们找不到任何符合要求的物理设备，很遗憾，那我们就失败了！

为了做出我们的效果图，我们首先要创建一个已经连接到我们创建的窗口的表面。（注意 **vkCreateWin32SurfaceKHR()**就是一个由 **VK_KHR_win32_surface** 扩展提供的实例函数，你必须把它添加到 **win32_LoadVulkanExtensions()**函数中。）

```
1       VkWin32SurfaceCreateInfoKHR surfaceCreateInfo = {};
2       surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
3       surfaceCreateInfo.hinstance = hInstance;
4       surfaceCreateInfo.hwnd = windowHandle;
5
6       result = vkCreateWin32SurfaceKHR( context.instance, &surfaceCreateInfo, NULL, &context.surface );
```

```
7       checkVulkanResult( result, "Could not create surface." );
```

接下来，我们需要对所有的物理设备进行迭代，然后找到能够渲染这个表面并且含有一个图形队列的物理设备：

```
1       uint32_t physicalDeviceCount = 0;
2       vkEnumeratePhysicalDevices( context.instance, &physicalDeviceCount, NULL );
3       VkPhysicalDevice *physicalDevices = new VkPhysicalDevice[physicalDeviceCount];
4       vkEnumeratePhysicalDevices( context.instance, &physicalDeviceCount, physicalDevices );
5
6       for( uint32_t i = 0; i < physicalDeviceCount; ++i ) {
7
8           VkPhysicalDeviceProperties deviceProperties = {};
9           vkGetPhysicalDeviceProperties( physicalDevices[i], &deviceProperties );
10
11          uint32_t queueFamilyCount = 0;
12          vkGetPhysicalDeviceQueueFamilyProperties( physicalDevices[i], &queueFamilyCount, NULL );
13          VkQueueFamilyProperties *queueFamilyProperties = new VkQueueFamilyProperties[queueFamilyCount];
14          vkGetPhysicalDeviceQueueFamilyProperties( physicalDevices[i],
15                                      &queueFamilyCount,
16                                      queueFamilyProperties );
17
18          for( uint32_t j = 0; j < queueFamilyCount; ++j ) {
19
20              VkBool32 supportsPresent;
21              vkGetPhysicalDeviceSurfaceSupportKHR( physicalDevices[i], j, context.surface,
22                                      &supportsPresent );
23
24              if( supportsPresent && ( queueFamilyProperties[j].queueFlags & VK_QUEUE_GRAPHICS_BIT ) ) {
```

```
25              context.physicalDevice = physicalDevices[i];
26              context.physicalDeviceProperties = deviceProperties;
27              context.presentQueueIdx = j;
28              break;
29            }
30        }
31        delete[] queueFamilyProperties;
32
33        if( context.physicalDevice ) {
34          break;
35        }
36      }
37      delete[] physicalDevices;
38
39      assert( context.physicalDevice, "No physical device detected that can render and present!" );
```

这里有大量的代码，不过大部分我们都已经很熟悉了。首先，有很多需要你来进行动态加载的函数（你可以查看链接代码进行动态加载），我们的 **vulkan_context** 也得到了扩展。在有些物理设备上我们能够进行一些渲染工作，要注意的是，我们已经得到了这些物理设备的队列索引。

现在我们缺少的就是逻辑设备了，即我们与物理设备的连接。我同样在下面写了一些我们下一步将会用到的东西：**VK_KHR_swapchain** 设备扩展：

```
1       // info for accessing one of the devices rendering queues:
2       VkDeviceQueueCreateInfo queueCreateInfo = {};
3       queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
4       queueCreateInfo.queueFamilyIndex = context.presentQueueIdx;
5       queueCreateInfo.queueCount = 1;
6       float queuePriorities[] = { 1.0f };    // ask for highest priority for our queue. (range [0,1])
7       queueCreateInfo.pQueuePriorities = queuePriorities;
8
```

```
9       VkDeviceCreateInfo deviceInfo = {};
10      deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
11      deviceInfo.queueCreateInfoCount = 1;
12      deviceInfo.pQueueCreateInfos = &queueCreateInfo;
13      deviceInfo.enabledLayerCount = 1;
14      deviceInfo.ppEnabledLayerNames = layers;
15
16      const char *deviceExtensions[] = { "VK_KHR_swapchain" };
17      deviceInfo.enabledExtensionCount = 1;
18      deviceInfo.ppEnabledExtensionNames = deviceExtensions;
19
20      VkPhysicalDeviceFeatures features = {};
21      features.shaderClipDistance = VK_TRUE;
22      deviceInfo.pEnabledFeatures = &features;
23
24      result = vkCreateDevice( context.physicalDevice, &deviceInfo, NULL, &context.device );
25      checkVulkanResult( result, "Failed to create logical device!" );
```

别忘了在你停止调试的时候删掉图层信息。 **VkPhysicalDeviceFeatures** 给我们提供了许多可选的规格特性，我们的执行可能支持这些特性。它们的每一个功能都可以启用。你可以在说明中找到一系列组成成分。有了它我们的绘图管线才能正常工作。（顺便说一下，我从验证层级中得到了这一信息。所以它们很重要！）最后，我们会创建一个最终能让屏幕显示事物的交换链。

交换链

现在我们已经有了渲染表面，接下来我们需要一个图像处理缓冲区，我们接下来会在这个缓冲区中写入数据。为了得到图像处理缓冲区，我们用了交换链扩展。创建时，我们需要确定我们需要的缓冲区数量（考虑有一个/两个/n 个缓冲区）、分辨率、色彩格式、色彩空间以及展示模式。要想创建一个交换链，我们需要大量的设置，但是这都可以理解。

我们首先来解决我们将要使用的色彩格式和色彩空间：

```
uint32_t formatCount = 0;
vkGetPhysicalDeviceSurfaceFormatsKHR( context.physicalDevice, context.surface,
                        &formatCount, NULL );
VkSurfaceFormatKHR *surfaceFormats = new VkSurfaceFormatKHR[formatCount];
vkGetPhysicalDeviceSurfaceFormatsKHR( context.physicalDevice, context.surface,
                        &formatCount, surfaceFormats );

// If the format list includes just one entry of VK_FORMAT_UNDEFINED, the surface has
// no preferred format. Otherwise, at least one supported format will be returned.
VkFormat colorFormat;
if( formatCount == 1 && surfaceFormats[0].format == VK_FORMAT_UNDEFINED ) {
   colorFormat = VK_FORMAT_B8G8R8_UNORM;
} else {
   colorFormat = surfaceFormats[0].format;
}
VkColorSpaceKHR colorSpace;
colorSpace = surfaceFormats[0].colorSpace;
delete[] surfaceFormats;
```

接下来我们需要看看我们的表面能力再确定我们可以请求的缓冲区数量。我们还需要决定是否使用一些表面转换（比如旋转 90°…不过我们没有用到）。我们必须要保证我们从交换链得到的分辨率与 **surfaceCapabilities.currentExtent** 相匹配。如果宽度和高度都是-1（或者说它们都不是-1 之类的！）这就意味着表面大小还没有确定，它们实际上可以被设置为任何值。但是，如果你已经设定了表面的大小，交换链的大小必须与之匹配！

```
VkSurfaceCapabilitiesKHR surfaceCapabilities = {};
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( context.physicalDevice, context.surface,
                          &surfaceCapabilities );

// we are effectively looking for double-buffering:
```

```
6       // if surfaceCapabilities.maxImageCount == 0 there is actually no limit on the number of images!
7       uint32_t desiredImageCount = 2;
8       if( desiredImageCount < surfaceCapabilities.minImageCount ) {
9          desiredImageCount = surfaceCapabilities.minImageCount;
10      } else if( surfaceCapabilities.maxImageCount != 0 &&
11             desiredImageCount > surfaceCapabilities.maxImageCount ) {
12         desiredImageCount = surfaceCapabilities.maxImageCount;
13      }
14
15      VkExtent2D surfaceResolution =   surfaceCapabilities.currentExtent;
16      if( surfaceResolution.width == -1 ) {
17         surfaceResolution.width = context.width;
18         surfaceResolution.height = context.height;
19      } else {
20         context.width = surfaceResolution.width;
21         context.height = surfaceResolution.height;
22      }
23
24      VkSurfaceTransformFlagBitsKHR preTransform = surfaceCapabilities.currentTransform;
25      if( surfaceCapabilities.supportedTransforms & VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR ) {
26         preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
27      }
```

我们有一些演示模式可以选择。如果工作队列不是空的，**VK_PRESENT_MODE_MAILBOX_KHR** 就会在任意垂直同步移除一个入口，以保持显示的单一入口队列。但是，当显示一段新的画面时，很明显它会取代之前的画面。所以，从某种意义上说它不是垂直同步的，因为如果一个更新的图画在同步的同时已经出现了，那么旧的画面可能根本就不会显示，它也不会造成屏幕画面撕裂。由于它的低延迟以及无撕裂的演示模式，在各种条件都能够支持的情况下，我们更愿意选用这种演示模式。**VK_PRESENT_MODE_IMMEDIATE_KHR** 没有垂直同步，并且如果一个画面延迟，屏幕会出现撕裂。

VK_PRESENT_MODE_FIFO_RELAXED_KHR 始终保持一个工作队列并且垂直同步，但它在画面有延迟的情况下仍然会出现画面撕裂。
VK_PRESENT_MODE_FIFO_KHR 和上一个差不多，但是不会出现画面撕裂。这是唯一的满足同步要求的表现模式，而且它本身就是我们的默认选择。

```
1       uint32_t presentModeCount = 0;
2       vkGetPhysicalDeviceSurfacePresentModesKHR( context.physicalDevice, context.surface,
3                                   &presentModeCount, NULL );
4       VkPresentModeKHR *presentModes = new VkPresentModeKHR[presentModeCount];
5       vkGetPhysicalDeviceSurfacePresentModesKHR( context.physicalDevice, context.surface,
6                                   &presentModeCount, presentModes );
7
8       VkPresentModeKHR presentationMode = VK_PRESENT_MODE_FIFO_KHR;    // always supported.
9       for( uint32_t i = 0; i < presentModeCount; ++i ) {
10        if( presentModes[i] == VK_PRESENT_MODE_MAILBOX_KHR ) {
11          presentationMode = VK_PRESENT_MODE_MAILBOX_KHR;
12          break;
13        }
14      }
15      delete[] presentModes;
```

现在要做的就是把这些放在一起，然后创建出我们的交换链：

```
1       VkSwapchainCreateInfoKHR swapChainCreateInfo = {};
2       swapChainCreateInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
3       swapChainCreateInfo.surface = context.surface;
4       swapChainCreateInfo.minImageCount = desiredImageCount;
5       swapChainCreateInfo.imageFormat = colorFormat;
6       swapChainCreateInfo.imageColorSpace = colorSpace;
7       swapChainCreateInfo.imageExtent = surfaceResolution;
8       swapChainCreateInfo.imageArrayLayers = 1;
```

```
9       swapChainCreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
10      swapChainCreateInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;    // <--
11      swapChainCreateInfo.preTransform = preTransform;
12      swapChainCreateInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
13      swapChainCreateInfo.presentMode = presentationMode;
14      swapChainCreateInfo.clipped = true;      // If we want clipping outside the extents
15                                      // (remember our device features?)
16
17      result = vkCreateSwapchainKHR( context.device, &swapChainCreateInfo, NULL, &context.swapChain );
18      checkVulkanResult( result, "Failed to create swapchain." );
```

我们需要注意共享模式。在本教程的所有代码中，没有工作队列或是任何其他资源的共享。与其他 API，例如 OpenGL 相比，Vulkan 在管理多任务队列和在同步的执行方面有明显的优势，所以我们在另一篇教程中详细探讨了这个问题。

我们已经创建了交换链，那么现在可以准备使用了。但在那之前，我们要先讨论图像布局，这就意味着我们需要讨论内存屏障、信号量和栅栏分隔符这些 Vulkan 必不可少的要素，我们必须要使用和理解这些要素。

交换链可以告诉我们在 **desiredImageCount** 中我们请求的 **VkImages** 的数量。它可以分配和使用支持这个图像的资源。在 **VK_IMAGE_LAYOUT_UNDEFINED** 或 **VK_IMAGE_LAYOUT_PREINITIALIZED** 布局中会创建一个 **VkImage**。例如，为了能够渲染这个图像，布局必须改变为 **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** 或 **VK_IMAGE_LAYOUT_GENERAL**。

那么什么是布局，为什么它们如此重要呢？图形数据以一种实现相关的方式储存在内存中。事先知道特殊内存的使用，就有可能对于数据方面的操作种类进行限制，“实现”就可以在数据存储时做出决定，这样就能够使存取性能更好。图像布局的转变可能会非常麻烦，在转变布局时我们可能需要使用内存屏障来同步所有的存取。例如，要想将 **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** 布局转变为 **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** 布局，我们要先写入所有的颜色信息，然后才能把图像移动到现在的布局中。我们调用 **vkCmdPipelineBarrier()**函数来完成这一操作。以下就是函数定义：

```
1       void vkCmdPipelineBarrier( VkCommandBuffer commandBuffer,
2                                  VkPipelineStageFlags srcStageMask,
3                                  VkPipelineStageFlags dstStageMask,
4                                  VkDependencyFlags dependencyFlags,
5                                  uint32_t memoryBarrierCount,
```

```
6                   const VkMemoryBarrier* pMemoryBarriers,
7                   uint32_t bufferMemoryBarrierCount,
8                   const VkBufferMemoryBarrier* pBufferMemoryBarriers,
9                   uint32_t imageMemoryBarrierCount,
10                  const VkImageMemoryBarrier* pImageMemoryBarriers);
```

这个函数可以帮助我们在命令缓冲区的内存屏障之前或之后的命令之间在队列中插入一个执行依赖或是一系列的内存依赖。**vkCmdPipelineBarrier** 是函数集的一部分，它以 **vkCmd***的形式将工作记录到命令缓冲区，之后由我们的工作队列来处理这些数据。这里还有很多要说的…首先，你要意识到命令缓冲区的创建不是同步的，你必须要注意在处理或是提交时，你的命令是否是按照你预计的顺序来进行。现在让我们稍稍跑题一下，我们先讲一下工作队列、命令缓冲区和提交工作的内容。

工作队列与命令缓冲区

命令缓冲区的内容会被提交至工作队列。我们在创建逻辑设备的同时就创建出了工作队列。回头看看，你就会发现我们创建逻辑设备之前就已经填上了 **VkDeviceQueueCreateInfo**。这就创建了我们的图形队列，我们可以在此执行我们的渲染命令。现在要做的就是在我们的 **vulkan_context** 结构中处理与储存队列。

```
vkGetDeviceQueue( context.device, context.presentQueueIdx, 0, &context.presentQueue );
```

在创建命令缓冲区之前我们需要创建一个命令缓冲器组。在我们分配命令缓冲区时，命令池是不透明目标。它们能让 Vulkan 实现在多个命令缓冲区间分摊创建时的资源消耗。

```
1   VkCommandPoolCreateInfo commandPoolCreateInfo = {};
2   commandPoolCreateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
3   commandPoolCreateInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
4   commandPoolCreateInfo.queueFamilyIndex = context.presentQueueIdx;
5
6   VkCommandPool commandPool;
7   result = vkCreateCommandPool( context.device, &commandPoolCreateInfo, NULL, &commandPool );
```

```
8       checkVulkanResult( result, "Failed to create command pool." );
```
我们可以单独调整这个命令池分配的命令（而不是需要重置整个命令池），并且只能提交给我们的工作队列。最后，准备好创建一对命令缓冲区：我们将为我们的设置创建一个命令缓冲区，再创建一个缓冲区来来进行我们的渲染命令。
```
1       VkCommandBufferAllocateInfo commandBufferAllocationInfo = {};
2       commandBufferAllocationInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
3       commandBufferAllocationInfo.commandPool = commandPool;
4       commandBufferAllocationInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
5       commandBufferAllocationInfo.commandBufferCount = 1;
6
7       result = vkAllocateCommandBuffers( context.device, &commandBufferAllocationInfo,
8                                          &context.setupCmdBuffer );
9       checkVulkanResult( result, "Failed to allocate setup command buffer." );
10
11      result = vkAllocateCommandBuffers( context.device, &commandBufferAllocationInfo,
12                                         &context.drawCmdBuffer );
13      checkVulkanResult( result, "Failed to allocate draw command buffer." );
```
缓冲区开始和结束的记录如下：
```
1       VkResult vkBeginCommandBuffer( VkCommandBuffer commandBuffer,
2                                      const VkCommandBufferBeginInfo* pBeginInfo);
3
4       VkResult vkEndCommandBuffer( VkCommandBuffer commandBuffer);
```
在这两个函数之间我们可以调用 **vkCmd*()**类函数。
同样，要提交我们的命令缓冲区，我们需要调用：
```
1       VkResult vkQueueSubmit( VkQueue queue,
2                               uint32_t submitCount,
3                               const VkSubmitInfo* pSubmits,
```

```
4                   VkFence fence );
```

为了改变图像布局，我们将会在我们的代码中仔细研究 **VkCommandBufferBeginInfo** 和 **VkSubmitInfo**。但是，我们现在有一个更重要的主题要讨论：同步。对于本教程我们只担心我们工作队列提交的同步以及在一个命令缓冲区的命令之间的同步问题。Vulkan 提供了一系列包括栅栏、信号和事件等的同步原语。Vulkan 还提供了障碍来帮助进行缓存控制和流程（对于图像布局来说，这就是我们所需要的东西）。

我们在本教程中不使用事件。栅栏和信号是你的典型构念。它们可以处于“signaled”状态或是“unsignaled”状态。

通常来说，主机会使用栅栏来决定队列提交工作的完成（正如你看到的，这是 **vkQueueSubmit()**函数的一个参数）。事件通常用于协调工作队列之间或是内部队列提交之间的操作。工作队列向它们发送信号，它们可以在相同或是不同的队列之间等待。在我们的内存障碍中我们将会使用信号。

我说了好多次，但这的确很重要。要想更好地使用 Vulkan，你需要对同步有一定了解。我建议你阅读规范第六章。

好了，让我们开始图像布局的改变吧！

图像布局

[好吧，这是注定要发生的，不是吗？…尽管验证层级没有什么好说的，但是在这一节中，有一些 API 的错误用法。如果我们没有购买，我们就不能够进行交换链上的内存屏障和布局改变。然而这并不意味着本章没有什么用。尽管我返工了，为了以更好的、正确的实方式来现我们所做的事情，请点击这里。（感谢 **ratchet freak** 指出这一点！）]

我们现在要做的就是抓取交换链为我们创建出的图像，为了让它们能够呈现出来，我们需要将它们从初始值 **VK_IMAGE_LAYOUT_UNDEFINED** 调为 **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR**。（在这一点我们还没有谈及对它们进行渲染..我们会做到的…也许是最后……不要放弃希望！）

在某种程度上，我们需要对这些图像进行读写。**VkImages** 不能帮助我们进行这些操作。绘图管线并不能直接访问图形对象。我们需要创建一个，它代表着图像连续范围和额外的元数据，有了它我们就能够访问图像数据了。还有另一个重要的代码要输入，让我们继续吧：

```
1       uint32_t imageCount = 0;
2       vkGetSwapchainImagesKHR( context.device, context.swapChain, &imageCount, NULL );
3       context.presentImages = new VkImage[imageCount];
4       vkGetSwapchainImagesKHR( context.device, context.swapChain, &imageCount, context.presentImages );
5
6       VkImageViewCreateInfo presentImagesViewCreateInfo = {};
7       presentImagesViewCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
8       presentImagesViewCreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
```

```
9       presentImagesViewCreateInfo.format = colorFormat;
10      presentImagesViewCreateInfo.components = { VK_COMPONENT_SWIZZLE_R,
11                                      VK_COMPONENT_SWIZZLE_G,
12                                      VK_COMPONENT_SWIZZLE_B,
13                                      VK_COMPONENT_SWIZZLE_A };
14      presentImagesViewCreateInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
15      presentImagesViewCreateInfo.subresourceRange.baseMipLevel = 0;
16      presentImagesViewCreateInfo.subresourceRange.levelCount = 1;
17      presentImagesViewCreateInfo.subresourceRange.baseArrayLayer = 0;
18      presentImagesViewCreateInfo.subresourceRange.layerCount = 1;
```

我们要做的第一件事就是得到交换链图像，然后将它们存储在我们的上下文中。之后我们将会需要它们。接下来我们需要填一个可以重复使用的结构，你应该已经对它很熟悉了。是的，这就是我们需要传递给创建 **VkImageView** 的函数的。这仍然需要很多的初始化代码：

```
1       VkCommandBufferBeginInfo beginInfo = {};
2       beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3       beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4
5       VkFenceCreateInfo fenceCreateInfo = {};
6       fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
7       VkFence submitFence;
8       vkCreateFence( context.device, &fenceCreateInfo, NULL, &submitFence );
```

由于我们需要记录一些命令并将它们提交给我们的工作队列，我们需要 **VkCommandBufferBeginInfo** 和 **VkFence**。接下来我们就可以开始循环当前图像然后改变它们的布局了：

```
1       VkImageView *presentImageViews = new VkImageView[imageCount];
2       for( uint32_t i = 0; i < imageCount; ++i ) {
```

```
3
4       // complete VkImageViewCreateInfo with image i:
5       presentImagesViewCreateInfo.image = context.presentImages[i];
6
7       // start recording on our setup command buffer:
8       vkBeginCommandBuffer( context.setupCmdBuffer, &beginInfo );
9
10      VkImageMemoryBarrier layoutTransitionBarrier = {};
11      layoutTransitionBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
12      layoutTransitionBarrier.srcAccessMask = 0;
13      layoutTransitionBarrier.dstAccessMask = VK_ACCESS_MEMORY_READ_BIT;
14      layoutTransitionBarrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
15      layoutTransitionBarrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
16      layoutTransitionBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
17      layoutTransitionBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
18      layoutTransitionBarrier.image = context.presentImages[i];
19      VkImageSubresourceRange resourceRange = { VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1 };
20      layoutTransitionBarrier.subresourceRange = resourceRange;
21
22      vkCmdPipelineBarrier(    context.setupCmdBuffer,
23                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
24                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
25                               0,
26                               0, NULL,
27                               0, NULL,
28                               1, &layoutTransitionBarrier );
```

```
29
30          vkEndCommandBuffer( context.setupCmdBuffer );
31
32          // submitting code to the queue:
33          VkPipelineStageFlags waitStageMask[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
34          VkSubmitInfo submitInfo = {};
35          submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
36          submitInfo.waitSemaphoreCount = 0;
37          submitInfo.pWaitSemaphores = NULL;
38          submitInfo.pWaitDstStageMask = waitStageMask;
39          submitInfo.commandBufferCount = 1;
40          submitInfo.pCommandBuffers = &context.setupCmdBuffer;
41          submitInfo.signalSemaphoreCount = 0;
42          submitInfo.pSignalSemaphores = NULL;
43          result = vkQueueSubmit( context.presentQueue, 1, &submitInfo, submitFence );
44
45          // waiting for it to finish:
46          vkWaitForFences( context.device, 1, &submitFence, VK_TRUE, UINT64_MAX );
47          vkResetFences( context.device, 1, &submitFence );
48
49          vkResetCommandBuffer( context.setupCmdBuffer, 0 );
50
51          // create the image view:
52          result = vkCreateImageView( context.device, &presentImagesViewCreateInfo, NULL,
53                                      &presentImageViews[i] );
54          checkVulkanResult( result, "Could not create ImageView." );
```

```
55      }
```

别被这一大段代码给吓到了…这段代码分为三个部分。

第一个就是我们管线屏障命令的记录，屏障命令将图像布局由 **oldLayout** 布局变为 **newLayout** 布局。其中重要的部分就是 srcAccessMask 和 dst AccessMask，它们可以在 **vkCmdPipelineBarrier()**函数前后的命令之间放置一个内存访问屏障。我的意思大概就是在这个屏障之后需要读取这个图像存储器的命令必须要进行等待。既然这样就没有其它命令了，但我们会在我们的渲染函数上做同样的事情，但情况不是这样！

第二部分就是用 **vkQueueSubmit()**将实际工作提交给工作队列，然后通过等待栅栏处于标记状态来等待工作的完成。在我们刚刚完成的设置命令缓冲区中我们传递记录以及我们等待被标记的栅栏。

最后一部分就是图像视图的创建，我们使用我们在循环之外创建的结构进行图像的创建。注意，我们用相同的命令缓冲区记录 **vkCmdPipelineBarrier()**函数两次，但是我们最后只提交一次结果。

我们已经讲了好多的基础知识，但是我们至今都没有做一些演示…

在我们着手创建帧缓冲（Framebuffers）之前，我们先别急，因为我们要先让窗口颜色做一些改变。

渲染黑色

我们现在有一些图像，我们可以做一些 ping-pong 测试，然后将它们显示在窗口中。它们没有什么内容，但是我们已经可以设置我们的渲染循环了。实际上没有什么是纯正的黑色的，黑色与白色完全相反！所以，我们添加一些平台代码，接下来定义我们的函数：

```
1       void render( ) {
2
3          uint32_t nextImageIdx;
4          vkAcquireNextImageKHR( context.device, context.swapChain, UINT64_MAX,
5                                 VK_NULL_HANDLE, VK_NULL_HANDLE, &nextImageIdx );
6
7          VkPresentInfoKHR presentInfo = {};
8          presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
9          presentInfo.pNext = NULL;
10         presentInfo.waitSemaphoreCount = 0;
11         presentInfo.pWaitSemaphores = NULL;
```

```
12          presentInfo.swapchainCount = 1;
13          presentInfo.pSwapchains = &context.swapChain;
14          presentInfo.pImageIndices = &nextImageIdx;
15          presentInfo.pResults = NULL;
16          vkQueuePresentKHR( context.presentQueue, &presentInfo );
17
18      }
19
20
21
22      // add another case to our WindowProc() switch:
23      case WM_PAINT: {
24          render( );
25          break;
26      }
```

我们调用 **vkAcquireNextImageKHR()**函数来得到下一个可用的交换链图像。如果一个可用交换链图像通过 **UINT64_MAX** 超时，我们请求阻止它。一旦它返回，**nextImageIdx** 就会获得这个图像的指标，我们可以用它进行我们的渲染。

做完之后，如果我们希望能够呈现出我们的结果，我们就需要调用 **vkQueuePresentKHR()**函数，它可以将当前图像的渲染排进表面队列。

这非常简单，不是吗？…好，但是不幸的是，如果你仔细看我们的验证层级调试输出，你会发现我们没有用一个白色窗口而是用了一个黑色窗口，这里的代码有许多的问题。其实这是我故意的，我想要退回去解释一下剩余的交换链接口，而且我们在这个过程中不使用新的复杂的渲染函数。别担心，我们会解决所有的问题，但是我们可能需要潜水回来…我希望你有足够的氧气。

深度图像缓冲区

由交换链提供的缓冲区是图像缓冲区。交换链并不会为我们创建深度图像缓冲区。但我们确实需要深度图像缓冲区来创建帧缓冲并完成最终渲染。这意味着我们将需要经过创建图像缓冲区、分配和绑定内存的过程。要进行内存处理，我们需要回到物理设备的创建，得到物理设备的内存属性：

```
1       // Fill up the physical device memory properties:
```

```
2       vkGetPhysicalDeviceMemoryProperties( context.physicalDevice, &context.memoryProperties );
```

现在我们得到了一个新的 **VkImage**，它可以作为我们的深度图像缓冲区：

```
1       VkImageCreateInfo imageCreateInfo = {};
2       imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
3       imageCreateInfo.imageType = VK_IMAGE_TYPE_2D;
4       imageCreateInfo.format = VK_FORMAT_D16_UNORM;                   // notice me senpai!
5       imageCreateInfo.extent = { context.width, context.height, 1 };
6       imageCreateInfo.mipLevels = 1;
7       imageCreateInfo.arrayLayers = 1;
8       imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;                // notice me senpai!
9       imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
10      imageCreateInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;    // notice me senpai!
11      imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
12      imageCreateInfo.queueFamilyIndexCount = 0;
13      imageCreateInfo.pQueueFamilyIndices = NULL;
14      imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;          // notice me senpai!
15
16      result = vkCreateImage( context.device, &imageCreateInfo, NULL, &context.depthImage );
17      checkVulkanResult( result, "Failed to create depth image." );
```

有人会认为这是正确的吗？不，没有人会这样想。这没有为资源进行内存分配，也没有给资源绑定任何内存。我们必须分配我们自己设备上的一些内存，然后将内存绑定到这个图像上。问题就是我们必须找到与我们需求相匹配的堆的物理设备内存属性。我们向设备请求本地内存。**VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT**，在我们有了哪些信息后，给资源分配和绑定内存就变得非常简单了：

```
1       VkMemoryRequirements memoryRequirements = {};
2       vkGetImageMemoryRequirements( context.device, context.depthImage, &memoryRequirements );
3
4       VkMemoryAllocateInfo imageAllocateInfo = {};
```

```
5       imageAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
6       imageAllocateInfo.allocationSize = memoryRequirements.size;
7
8       // memoryTypeBits is a bitfield where if bit i is set, it means that
9       // the VkMemoryType i of the VkPhysicalDeviceMemoryProperties structure
10      // satisfies the memory requirements:
11      uint32_t memoryTypeBits = memoryRequirements.memoryTypeBits;
12      VkMemoryPropertyFlags desiredMemoryFlags = VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT;
13      for( uint32_t i = 0; i < 32; ++i ) {
14        VkMemoryType memoryType = context.memoryProperties.memoryTypes[i];
15        if( memoryTypeBits & 1 ) {
16          if( ( memoryType.propertyFlags & desiredMemoryFlags ) == desiredMemoryFlags ) {
17            imageAllocateInfo.memoryTypeIndex = i;
18            break;
19          }
20        }
21        memoryTypeBits = memoryTypeBits >> 1;
22      }
23
24      VkDeviceMemory imageMemory = {};
25      result = vkAllocateMemory( context.device, &imageAllocateInfo, NULL, &imageMemory );
26      checkVulkanResult( result, "Failed to allocate device memory." );
27
28      result = vkBindImageMemory( context.device, context.depthImage, imageMemory, 0 );
29      checkVulkanResult( result, "Failed to bind image memory." );
```

我们将图像创建于 **VK_IMAGE_LAYOUT_UNDEFINED** 层级，我们需要将它改为 **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL** 层级。你已经看

到相似的代码了，但是这里有一些与深度缓冲区相关而不是与颜色缓冲区相关的改动：

```
1       VkCommandBufferBeginInfo beginInfo = {};
2       beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3       beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4
5       vkBeginCommandBuffer( context.setupCmdBuffer, &beginInfo );
6
7       VkImageMemoryBarrier layoutTransitionBarrier = {};
8       layoutTransitionBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
9       layoutTransitionBarrier.srcAccessMask = 0;
10      layoutTransitionBarrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
11                                      VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
12      layoutTransitionBarrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
13      layoutTransitionBarrier.newLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
14      layoutTransitionBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
15      layoutTransitionBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
16      layoutTransitionBarrier.image = context.depthImage;
17      VkImageSubresourceRange resourceRange = { VK_IMAGE_ASPECT_DEPTH_BIT, 0, 1, 0, 1 };
18      layoutTransitionBarrier.subresourceRange = resourceRange;
19
20      vkCmdPipelineBarrier(    context.setupCmdBuffer,
21                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
22                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
23                               0,
24                               0, NULL,
25                               0, NULL,
```

```
                    1, &layoutTransitionBarrier );

    vkEndCommandBuffer( context.setupCmdBuffer );

    VkPipelineStageFlags waitStageMask[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.waitSemaphoreCount = 0;
    submitInfo.pWaitSemaphores = NULL;
    submitInfo.pWaitDstStageMask = waitStageMask;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &context.setupCmdBuffer;
    submitInfo.signalSemaphoreCount = 0;
    submitInfo.pSignalSemaphores = NULL;
    result = vkQueueSubmit( context.presentQueue, 1, &submitInfo, submitFence );

    vkWaitForFences( context.device, 1, &submitFence, VK_TRUE, UINT64_MAX );
    vkResetFences( context.device, 1, &submitFence );
    vkResetCommandBuffer( context.setupCmdBuffer, 0 );
```

我们几乎要完成了。只差 **VkImageView**，我们将我们的深度图像缓冲区初始化，准备进行使用：

```
    VkImageAspectFlags aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
    VkImageViewCreateInfo imageViewCreateInfo = {};
    imageViewCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    imageViewCreateInfo.image = context.depthImage;
    imageViewCreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    imageViewCreateInfo.format = imageCreateInfo.format;
```

```
7       imageViewCreateInfo.components = { VK_COMPONENT_SWIZZLE_IDENTITY, VK_COMPONENT_SWIZZLE_IDENTITY,
8                                          VK_COMPONENT_SWIZZLE_IDENTITY, VK_COMPONENT_SWIZZLE_IDENTITY };
9       imageViewCreateInfo.subresourceRange.aspectMask = aspectMask;
10      imageViewCreateInfo.subresourceRange.baseMipLevel = 0;
11      imageViewCreateInfo.subresourceRange.levelCount = 1;
12      imageViewCreateInfo.subresourceRange.baseArrayLayer = 0;
13      imageViewCreateInfo.subresourceRange.layerCount = 1;
14
15      result = vkCreateImageView( context.device, &imageViewCreateInfo, NULL, &context.depthImageView );
16      checkVulkanResult( result, "Failed to create image view." );
```

渲染通道与帧缓冲

我们现在开始设置我们的渲染管道了。管道将会将一切都结合到一起。但是我们首先需要创建一个渲染通道，那样我们就创建出了所有的帧缓冲。一个渲染通道封装了一组帧缓冲附件，一个或多个 subpass 以及 subpass 之间的相关性。我们将要创建一个有单个 subpass 和两个帧缓冲附件的渲染通道，这两个附件一个给颜色缓冲区、一个给我们的深度缓冲区。在我们设置渲染通道时，如果一个 subpass 渲染成了一个附件，那么它将会成为另一个 subpass 的输入，这样事情就变得很麻烦了…但我们不会走到那一步的，所以，我们先来创建我们的帧缓冲附件信息：

```
1       VkAttachmentDescription passAttachments[2] = { };
2       passAttachments[0].format = colorFormat;
3       passAttachments[0].samples = VK_SAMPLE_COUNT_1_BIT;
4       passAttachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
5       passAttachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
6       passAttachments[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
7       passAttachments[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
8       passAttachments[0].initialLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

```
9       passAttachments[0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
10
11      passAttachments[1].format = VK_FORMAT_D16_UNORM;
12      passAttachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
13      passAttachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
14      passAttachments[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
15      passAttachments[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
16      passAttachments[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
17      passAttachments[1].initialLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
18      passAttachments[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
19
20      VkAttachmentReference colorAttachmentReference = {};
21      colorAttachmentReference.attachment = 0;
22      colorAttachmentReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
23
24      VkAttachmentReference depthAttachmentReference = {};
25      depthAttachmentReference.attachment = 1;
26      depthAttachmentReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

接下来，我们创建一个 **VkRenderPass**，它包含了一个使用了我们两个帧缓冲附件的 subpass。对于我们对本篇教程的预期，这里讲的有些多（描述符和许多有趣的东西），就其本身来说，这就是我们现在需要的全部：

```
1       VkSubpassDescription subpass = {};
2       subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
3       subpass.colorAttachmentCount = 1;
4       subpass.pColorAttachments = &colorAttachmentReference;
5       subpass.pDepthStencilAttachment = &depthAttachmentReference;
6
```

```
7       VkRenderPassCreateInfo renderPassCreateInfo = {};
8       renderPassCreateInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
9       renderPassCreateInfo.attachmentCount = 2;
10      renderPassCreateInfo.pAttachments = passAttachments;
11      renderPassCreateInfo.subpassCount = 1;
12      renderPassCreateInfo.pSubpasses = &subpass;
13
14      result = vkCreateRenderPass( context.device, &renderPassCreateInfo, NULL, &context.renderPass );
15      checkVulkanResult( result, "Failed to create renderpass" );
```

这一段是写给渲染通道的。渲染通道本身基本上就已经决定了我们能够创建那种类型的帧缓冲区和绘图管线，这就是为什么我们会选择先创建渲染通道。现在就让我们开始创建与这个渲染通道兼容的帧缓冲区：

```
1       VkImageView frameBufferAttachments[2];
2       frameBufferAttachments[1] = context.depthImageView;
3
4       VkFramebufferCreateInfo frameBufferCreateInfo = {};
5       frameBufferCreateInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
6       frameBufferCreateInfo.renderPass = context.renderPass;
7       frameBufferCreateInfo.attachmentCount = 2;   // must be equal to the attachment count on render pass
8       frameBufferCreateInfo.pAttachments = frameBufferAttachments;
9       frameBufferCreateInfo.width = context.width;
10      frameBufferCreateInfo.height = context.height;
11      frameBufferCreateInfo.layers = 1;
12
13      // create a framebuffer per swap chain imageView:
14      context.frameBuffers = new VkFramebuffer[ imageCount ];
15      for( uint32_t i = 0; i < imageCount; ++i ) {
```

```
16          frameBufferAttachments[0] = presentImageViews[ i ];
17          result = vkCreateFramebuffer( context.device, &frameBufferCreateInfo,
18                                        NULL, &context.frameBuffers[i] );
19          checkVulkanResult( result, "Failed to create framebuffer.");
20      }
```

注意，在我们给交换链现有图像设置颜色附件时，我们创建了两个帧缓冲区，但是我们给它们使用了相同的深度缓冲区。

顶点缓冲区

是时候定义我们的顶点信息了。我们的目标是渲染出一个三角形，所以我们必须需要定义我们的顶点信息，为三个顶点分配足够的内存，然后将它们上传到缓冲区。让我们首先定义一个简单的结构，然后为三个顶点创建一个缓冲区：

```
1       struct vertex {
2          float x, y, z, w;
3       };
4
5       // create our vertex buffer:
6       VkBufferCreateInfo vertexInputBufferInfo = {};
7       vertexInputBufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
8       vertexInputBufferInfo.size = sizeof(vertex) * 3; // size in Bytes
9       vertexInputBufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
10      vertexInputBufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
11
12      result = vkCreateBuffer( context.device, &vertexInputBufferInfo, NULL,
13                               &context.vertexInputBuffer );
14      checkVulkanResult( result, "Failed to create vertex input buffer." );
```

现在缓冲区已经创建完了。就像我们为 **VkImage** 做的一样，我们现在需要做一些相同的事情来为 **VkBuffer** 分配内存。不过有一点不同，那就是我们现在确实需要在一个堆中的内存，我们要从主机向其写入内容：（**VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT**）

```
VkMemoryRequirements vertexBufferMemoryRequirements = {};
vkGetBufferMemoryRequirements( context.device, context.vertexInputBuffer,
                               &vertexBufferMemoryRequirements );

VkMemoryAllocateInfo bufferAllocateInfo = {};
bufferAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
bufferAllocateInfo.allocationSize = vertexBufferMemoryRequirements.size;

uint32_t vertexMemoryTypeBits = vertexBufferMemoryRequirements.memoryTypeBits;
VkMemoryPropertyFlags vertexDesiredMemoryFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT;
for( uint32_t i = 0; i < 32; ++i ) {
   VkMemoryType memoryType = context.memoryProperties.memoryTypes[i];
   if( vertexMemoryTypeBits & 1 ) {
      if( ( memoryType.propertyFlags & vertexDesiredMemoryFlags ) == vertexDesiredMemoryFlags ) {
         bufferAllocateInfo.memoryTypeIndex = i;
         break;
      }
   }
   vertexMemoryTypeBits = vertexMemoryTypeBits >> 1;
}

VkDeviceMemory vertexBufferMemory;
result = vkAllocateMemory( context.device, &bufferAllocateInfo, NULL, &vertexBufferMemory );
checkVulkanResult( result, "Failed to allocate buffer memory." );
```

即使我们由主机请求可用内存，这个内存也不会直接由主机使用。它所做的就是创建一个可映射内存。要想对这个内存进行写入，我们首先要通过调用**vkMapMemory()**来获取一个指向可映射内存对象的主机虚拟地址。所以让我们先对这个内存进行映射，这样我们才能对它进行写入和绑定：

```
1       void *mapped;
2       result = vkMapMemory( context.device, vertexBufferMemory, 0, VK_WHOLE_SIZE, 0, &mapped );
3       checkVulkanResult( result, "Failed to map buffer memory." );
4
5       vertex *triangle = (vertex *) mapped;
6       vertex v1 = { -1.0f, -1.0f, 0, 1.0f };
7       vertex v2 = {  1.0f, -1.0f, 0, 1.0f };
8       vertex v3 = {  0.0f,  1.0f, 0, 1.0f };
9       triangle[0] = v1;
10      triangle[1] = v2;
11      triangle[2] = v3;
12
13      vkUnmapMemory( context.device, vertexBufferMemory );
14
15      result = vkBindBufferMemory( context.device, context.vertexInputBuffer, vertexBufferMemory, 0 );
16      checkVulkanResult( result, "Failed to bind buffer memory." );
```

现在你就明白了，一个三角形将会闯过我们的管线。那么问题来了，我们还没有一个管线，我们给它配绘图管线了吗？我们已经做的差不多了…我们现在只需要讨论一下着色器了！

着色器

我们的目标是建立一个简单的顶点和片段着色器。Vulkan 默认着色器代码是以 SPIR-V 格式编写的，但这并不是一个大问题，因为我们可以使用一个免费工具来将我们的 GLSL 着色器转换为 SPIR-V 着色器： **glslangValidator**。你可以从下面的地址得到它。

git clone https://github.com/KhronosGroup/glslang

所以，举个例子，如果我们要为我们的 **simple.vert** 进行顶点着色，我们要输入以下代码：

```
1       #version 400
2       #extension GL_ARB_separate_shader_objects : enable
```

```
3       #extension GL_ARB_shading_language_420pack : enable
4
5       layout (location = 0) in vec4 pos;
6
7       void main() {
8          gl_Position = pos;
9       }
```

我们可以调用：

```
glslangValidator -V simple.vert
```

这将会在相同的文件夹创建一个 **vert.spv**，简单而又明了，不是吗？

同样，对于我们的 **simple.frag** 片段着色器：

```
1       #version 400
2       #extension GL_ARB_separate_shader_objects : enable
3       #extension GL_ARB_shading_language_420pack : enable
4
5       layout (location = 0) out vec4 uFragColor;
6
7       void main() {
8          uFragColor = vec4( 0.0, 0.5, 1.0, 1.0 );
9       }
```

```
glslangValidator -V simple.frag
```

最后，我们以 **frag.spv** 作为结束。

我们要坚持我们适当展示所有代码的原则，我们要将其加载到 Vulkan 中，我们需要以下代码：

```
1          uint32_t codeSize;
2          char *code = new char[10000];
3          HANDLE fileHandle = 0;
```

```
4
5       // load our vertex shader:
6       fileHandle = CreateFile( "..\\vert.spv", GENERIC_READ, 0, NULL,
7                               OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
8       if( fileHandle == INVALID_HANDLE_VALUE ) {
9          OutputDebugStringA( "Failed to open shader file." );
10         exit(1);
11      }
12      ReadFile( (HANDLE)fileHandle, code, 10000, (LPDWORD)&codeSize, 0 );
13      CloseHandle( fileHandle );
14
15      VkShaderModuleCreateInfo vertexShaderCreationInfo = {};
16      vertexShaderCreationInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
17      vertexShaderCreationInfo.codeSize = codeSize;
18      vertexShaderCreationInfo.pCode = (uint32_t *)code;
19
20      VkShaderModule vertexShaderModule;
21      result = vkCreateShaderModule( context.device, &vertexShaderCreationInfo, NULL, &vertexShaderModule );
22      checkVulkanResult( result, "Failed to create vertex shader module." );
23
24      // load our fragment shader:
25      fileHandle = CreateFile( "..\\frag.spv", GENERIC_READ, 0, NULL,
26                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
27      if( fileHandle == INVALID_HANDLE_VALUE ) {
28         OutputDebugStringA( "Failed to open shader file." );
29         exit(1);
```

```
30      }
31      ReadFile( (HANDLE)fileHandle, code, 10000, (LPDWORD)&codeSize, 0 );
32      CloseHandle( fileHandle );
33
34      VkShaderModuleCreateInfo fragmentShaderCreationInfo = {};
35      fragmentShaderCreationInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
36      fragmentShaderCreationInfo.codeSize = codeSize;
37      fragmentShaderCreationInfo.pCode = (uint32_t *)code;
38
39      VkShaderModule fragmentShaderModule;
40      result = vkCreateShaderModule( context.device, &fragmentShaderCreationInfo, NULL, &fragmentShaderModule );
41      checkVulkanResult( result, "Failed to create vertex shader module." );
```

注意，如果我们找不到着色器代码，那么我们就非常遗憾地失败了，我们希望能够在运行它的上级目录找到它。如果你能够在 Visual Studio devenv 中运行，那非常好，但是如果你从命令行运行，那么它会崩溃并且没有任何的报告。我建议你换一个更适合你的方式来运行。

粗略地看一下代码，你可能会以各种语言来“问候”我，但我会忍耐的。我知道你在抱怨什么，为了这篇教程的目标，我并不在意。相信我，这不是我在我自己内部引擎中用的代码。:）

我希望在你们骂完我之后，你应该知道你需要做的是加载你自己的着色器。

好了。我想我们终于可以开始建立我们的渲染通道了。

绘图管线

绘图管线会跟踪渲染所需的所有状态。绘图管线是多组分着色器阶段、多组分固定函数管线阶段和管线层级的集合。我们把所有创建出的东西都放到了这里，所以我们可以以这样那样的方式来配置管线。我们需要提前设置所有的一切。记住，Vulkan 没有状态，所以我们需要配置和储存所有我们想要或是需要的状态。所以我们通过创建 **VkPipeline** 来达到这一目的。

你知道的，如果不知道的话至少想象一下，一个绘图管线中有非常非常多的状态。从视窗到混合函数，从着色器阶段到绑定…所以，我们下面要做的就是设置所有的状态。（在这个例子中，我们不会使用一些大的构件，例如描述符设置，绑定等…）所以，让我们先创建一个空的图层：

```
1       VkPipelineLayoutCreateInfo layoutCreateInfo = {};
```

```
2       layoutCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
3       layoutCreateInfo.setLayoutCount = 0;
4       layoutCreateInfo.pSetLayouts = NULL;    // Not setting any bindings!
5       layoutCreateInfo.pushConstantRangeCount = 0;
6       layoutCreateInfo.pPushConstantRanges = NULL;
7
8       result = vkCreatePipelineLayout( context.device, &layoutCreateInfo, NULL,
9                                        &context.pipelineLayout );
10      checkVulkanResult( result, "Failed to create pipeline layout." );
```

我们可能在之后回到这一阶段，所以，举个例子，我们可以设置一个同意缓冲区的对象来通过我们着色器的一些统一的 value。但是作为第一篇教程，空值就已经不错了！接下来我们用我们加载的着色模块来进行着色阶段设置：

```
1       VkPipelineShaderStageCreateInfo shaderStageCreateInfo[2] = {};
2       shaderStageCreateInfo[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
3       shaderStageCreateInfo[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
4       shaderStageCreateInfo[0].module = vertexShaderModule;
5       shaderStageCreateInfo[0].pName = "main";        // shader entry point function name
6       shaderStageCreateInfo[0].pSpecializationInfo = NULL;
7
8       shaderStageCreateInfo[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
9       shaderStageCreateInfo[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
10      shaderStageCreateInfo[1].module = fragmentShaderModule;
11      shaderStageCreateInfo[1].pName = "main";        // shader entry point function name
12      shaderStageCreateInfo[1].pSpecializationInfo = NULL;
```

这里没什么特别的。要配置顶点输入处理，我们需要它们：

```
1       VkVertexInputBindingDescription vertexBindingDescription = {};
2       vertexBindingDescription.binding = 0;
```

```
3       vertexBindingDescription.stride = sizeof(vertex);
4       vertexBindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
5
6       VkVertexInputAttributeDescription vertexAttributeDescritpion = {};
7       vertexAttributeDescritpion.location = 0;
8       vertexAttributeDescritpion.binding = 0;
9       vertexAttributeDescritpion.format = VK_FORMAT_R32G32B32A32_SFLOAT;
10      vertexAttributeDescritpion.offset = 0;
11
12      VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo = {};
13      vertexInputStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
14      vertexInputStateCreateInfo.vertexBindingDescriptionCount = 1;
15      vertexInputStateCreateInfo.pVertexBindingDescriptions = &vertexBindingDescription;
16      vertexInputStateCreateInfo.vertexAttributeDescriptionCount = 1;
17      vertexInputStateCreateInfo.pVertexAttributeDescriptions = &vertexAttributeDescritpion;
18
19      // vertex topology config:
20      VkPipelineInputAssemblyStateCreateInfo inputAssemblyStateCreateInfo = {};
21      inputAssemblyStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
22      inputAssemblyStateCreateInfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
23      inputAssemblyStateCreateInfo.primitiveRestartEnable = VK_FALSE;
```

好的，这里我需要解释一下。在第一部分，我们将顶点位置（我们的 (x,y,z,w)）确定为 location = 0, binding = 0。接下来我们配置顶点拓扑结构以一个三角形列表的形式解释我们的定点缓冲区。

接下来，视窗和裁剪就已经配置完了。稍后我们会将其状态改为动态，这样我们就能在每一帧改变它们了。

```
1       VkViewport viewport = {};
2       viewport.x = 0;
```

```
3       viewport.y = 0;
4       viewport.width = context.width;
5       viewport.height = context.height;
6       viewport.minDepth = 0;
7       viewport.maxDepth = 1;
8
9       VkRect2D scissors = {};
10      scissors.offset = { 0, 0 };
11      scissors.extent = { context.width, context.height };
12
13      VkPipelineViewportStateCreateInfo viewportState = {};
14      viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
15      viewportState.viewportCount = 1;
16      viewportState.pViewports = &viewport;
17      viewportState.scissorCount = 1;
18      viewportState.pScissors = &scissors;
```

在这里我们可以设置我们的光栅化配置。其中大部分都能自行解释：

```
1       VkPipelineRasterizationStateCreateInfo rasterizationState = {};
2       rasterizationState.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
3       rasterizationState.depthClampEnable = VK_FALSE;
4       rasterizationState.rasterizerDiscardEnable = VK_FALSE;
5       rasterizationState.polygonMode = VK_POLYGON_MODE_FILL;
6       rasterizationState.cullMode = VK_CULL_MODE_NONE;
7       rasterizationState.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

```
8       rasterizationState.depthBiasEnable = VK_FALSE;
9       rasterizationState.depthBiasConstantFactor = 0;
10      rasterizationState.depthBiasClamp = 0;
11      rasterizationState.depthBiasSlopeFactor = 0;
12      rasterizationState.lineWidth = 1;
```

接下来是采样配置：

```
1     VkPipelineMultisampleStateCreateInfo multisampleState = {};
2     multisampleState.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
3     multisampleState.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
4     multisampleState.sampleShadingEnable = VK_FALSE;
5     multisampleState.minSampleShading = 0;
6     multisampleState.pSampleMask = NULL;
7     multisampleState.alphaToCoverageEnable = VK_FALSE;
8     multisampleState.alphaToOneEnable = VK_FALSE;
```

在这一阶段，我们启用深度测试，禁用模板。

```
1       VkStencilOpState noOPStencilState = {};
2       noOPStencilState.failOp = VK_STENCIL_OP_KEEP;
3       noOPStencilState.passOp = VK_STENCIL_OP_KEEP;
4       noOPStencilState.depthFailOp = VK_STENCIL_OP_KEEP;
5       noOPStencilState.compareOp = VK_COMPARE_OP_ALWAYS;
6       noOPStencilState.compareMask = 0;
7       noOPStencilState.writeMask = 0;
8       noOPStencilState.reference = 0;
9
10      VkPipelineDepthStencilStateCreateInfo depthState = {};
11      depthState.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
```

```
12      depthState.depthTestEnable = VK_TRUE;
13      depthState.depthWriteEnable = VK_TRUE;
14      depthState.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
15      depthState.depthBoundsTestEnable = VK_FALSE;
16      depthState.stencilTestEnable = VK_FALSE;
17      depthState.front = noOPStencilState;
18      depthState.back = noOPStencilState;
19      depthState.minDepthBounds = 0;
20      depthState.maxDepthBounds = 0;
```

对于本教程来说，配色是有缺陷的，你可以在这里进行配置：

```
1       VkPipelineColorBlendAttachmentState colorBlendAttachmentState = {};
2       colorBlendAttachmentState.blendEnable = VK_FALSE;
3       colorBlendAttachmentState.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
4       colorBlendAttachmentState.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR;
5       colorBlendAttachmentState.colorBlendOp = VK_BLEND_OP_ADD;
6       colorBlendAttachmentState.srcAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
7       colorBlendAttachmentState.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
8       colorBlendAttachmentState.alphaBlendOp = VK_BLEND_OP_ADD;
9       colorBlendAttachmentState.colorWriteMask = 0xf;
10
11      VkPipelineColorBlendStateCreateInfo colorBlendState = {};
12      colorBlendState.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
13      colorBlendState.logicOpEnable = VK_FALSE;
14      colorBlendState.logicOp = VK_LOGIC_OP_CLEAR;
15      colorBlendState.attachmentCount = 1;
16      colorBlendState.pAttachments = &colorBlendAttachmentState;
```

```
17      colorBlendState.blendConstants[0] = 0.0;
18      colorBlendState.blendConstants[1] = 0.0;
19      colorBlendState.blendConstants[2] = 0.0;
20      colorBlendState.blendConstants[3] = 0.0;
```

现在所有的这些配置对于管线的一生来说都是不变的。我们可能想要在每一帧改变这些状态中的一部分，例如我们的视窗或是裁剪视窗。要想将状态动态化，我们可以：

```
1       VkDynamicState dynamicState[2] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };
2       VkPipelineDynamicStateCreateInfo dynamicStateCreateInfo = {};
3       dynamicStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
4       dynamicStateCreateInfo.dynamicStateCount = 2;
5       dynamicStateCreateInfo.pDynamicStates = dynamicState;
```

最后，我们把所有的一切汇聚到一起，这样我们就创建出了我们的绘图管线：

```
1       VkGraphicsPipelineCreateInfo pipelineCreateInfo = {};
2       pipelineCreateInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
3       pipelineCreateInfo.stageCount = 2;
4       pipelineCreateInfo.pStages = shaderStageCreateInfo;
5       pipelineCreateInfo.pVertexInputState = &vertexInputStateCreateInfo;
6       pipelineCreateInfo.pInputAssemblyState = &inputAssemblyStateCreateInfo;
7       pipelineCreateInfo.pTessellationState = NULL;
8       pipelineCreateInfo.pViewportState = &viewportState;
9       pipelineCreateInfo.pRasterizationState = &rasterizationState;
10      pipelineCreateInfo.pMultisampleState = &multisampleState;
11      pipelineCreateInfo.pDepthStencilState = &depthState;
12      pipelineCreateInfo.pColorBlendState = &colorBlendState;
13      pipelineCreateInfo.pDynamicState = &dynamicStateCreateInfo;
14      pipelineCreateInfo.layout = context.pipelineLayout;
```

```
15      pipelineCreateInfo.renderPass = context.renderPass;
16      pipelineCreateInfo.subpass = 0;
17      pipelineCreateInfo.basePipelineHandle = NULL;
18      pipelineCreateInfo.basePipelineIndex = 0;
19
20      result = vkCreateGraphicsPipelines( context.device, VK_NULL_HANDLE, 1, &pipelineCreateInfo, NULL,
21                              &context.pipeline );
22      checkVulkanResult( result, "Failed to create graphics pipeline." );
```

这里有非常多的代码…但是这只是设置状态。好消息就是我们现在准备开始渲染我们的三角形了。我们将会更新我们的渲染方法来渲染三角形。

最终渲染

我们终于可以更新我们的渲染代码来将一个三角形呈现在屏幕上了。你能相信吗？好，让我告诉你该怎么做：

```
1       void render( ) {
2
3         vkSemaphore presentCompleteSemaphore, renderingCompleteSemaphore;
4         VkSemaphoreCreateInfo semaphoreCreateInfo = { VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO, 0, 0 };
5         vkCreateSemaphore( context.device, &semaphoreCreateInfo, NULL, &presentCompleteSemaphore );
6         vkCreateSemaphore( context.device, &semaphoreCreateInfo, NULL, &renderingCompleteSemaphore );
7
8         uint32_t nextImageIdx;
9         vkAcquireNextImageKHR(   context.device, context.swapChain, UINT64_MAX,
10                          presentCompleteSemaphore, VK_NULL_HANDLE, &nextImageIdx );
```

首先我们需要注意我们渲染调用的同步，所以我们创建了一对信号量，并且更新了我们 **vkAcquireNextImageKHR()**函数的调用。我们需要将展示图像从 **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** 图层变为 **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** 图层。我们已经知道该怎么做了，下面就是代码：

```
1       VkCommandBufferBeginInfo beginInfo = {};
```

```
2       beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3       beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4
5       vkBeginCommandBuffer( context.drawCmdBuffer, &beginInfo );
6
7       // change image layout from VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
8       // to VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
9       VkImageMemoryBarrier layoutTransitionBarrier = {};
10      layoutTransitionBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
11      layoutTransitionBarrier.srcAccessMask = VK_ACCESS_MEMORY_READ_BIT;
12      layoutTransitionBarrier.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
13                                      VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
14      layoutTransitionBarrier.oldLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
15      layoutTransitionBarrier.newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
16      layoutTransitionBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
17      layoutTransitionBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
18      layoutTransitionBarrier.image = context.presentImages[ nextImageIdx ];
19      VkImageSubresourceRange resourceRange = { VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1 };
20      layoutTransitionBarrier.subresourceRange = resourceRange;
21
22      vkCmdPipelineBarrier(    context.drawCmdBuffer,
23                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
24                               VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
25                               0,
26                               0, NULL,
27                               0, NULL,
```

```
28                      1, &layoutTransitionBarrier );
```
这个代码你现在应该很熟悉了，接下来我们将激活我们的渲染通道：
```
1       VkClearValue clearValue[] = { { 1.0f, 1.0f, 1.0f, 1.0f }, { 1.0, 0.0 } };
2        VkRenderPassBeginInfo renderPassBeginInfo = {};
3        renderPassBeginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
4        renderPassBeginInfo.renderPass = context.renderPass;
5        renderPassBeginInfo.framebuffer = context.frameBuffers[ nextImageIdx ];
6        renderPassBeginInfo.renderArea = { 0, 0, context.width, context.height };
7        renderPassBeginInfo.clearValueCount = 2;
8        renderPassBeginInfo.pClearValues = clearValue;
9        vkCmdBeginRenderPass( context.drawCmdBuffer, &renderPassBeginInfo,
10                      VK_SUBPASS_CONTENTS_INLINE );
```

这里没什么特别的。我要说的只有要明确为两个帧缓冲附件使用哪一个帧缓冲区并设置明确的值。接下来我们通过确定我们的绘图管线来确定所有的渲染状态：
```
1     vkCmdBindPipeline( context.drawCmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, context.pipeline );
2
3     // take care of dynamic state:
4     VkViewport viewport = { 0, 0, context.width, context.height, 0, 1 };
5     vkCmdSetViewport( context.drawCmdBuffer, 0, 1, &viewport );
6
7     VkRect2D scissor = { 0, 0, context.width, context.height };
8     vkCmdSetScissor( context.drawCmdBuffer, 0, 1, &scissor);
```
注意在这一阶段我们是如何设置动态状态量的。接下来我们确定我们的定点缓冲区，绘制出我们可爱的三角形，然后请求 Vulkan 画出一个它的实例：
```
1       VkDeviceSize offsets = { };
```

```
2       vkCmdBindVertexBuffers( context.drawCmdBuffer, 0, 1, &context.vertexInputBuffer, &offsets );
3
4       vkCmdDraw( context.drawCmdBuffer,
5               3,   // vertex count
6               1,   // instance count
7               0,   // first vertex
8               0 ); // first instance
9
10      vkCmdEndRenderPass( context.drawCmdBuffer );
```

我们差不多已经要做完了。那么还缺少什么呢？对了，我们需要将 **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** 图层改变为 **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** 图层，我们需要保证在我们改变图层之前已经做完了所有的渲染工作！

```
1       VkImageMemoryBarrier prePresentBarrier = {};
2       prePresentBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
3       prePresentBarrier.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
4       prePresentBarrier.dstAccessMask = VK_ACCESS_MEMORY_READ_BIT;
5       prePresentBarrier.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
6       prePresentBarrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
7       prePresentBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
8       prePresentBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
9       prePresentBarrier.subresourceRange = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1};
10      prePresentBarrier.image = context.presentImages[ nextImageIdx ];
11
12      vkCmdPipelineBarrier( context.drawCmdBuffer,
13                      VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
14                      VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
15                      0,
```

```
16                  0, NULL,
17                  0, NULL,
18                  1, &prePresentBarrier );
19
20      vkEndCommandBuffer( context.drawCmdBuffer );
```

就是这样了，现在只需要提交了：

```
1       VkFence renderFence;
2       VkFenceCreateInfo fenceCreateInfo = {};
3       fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
4       vkCreateFence( context.device, &fenceCreateInfo, NULL, &renderFence );
5
6       VkPipelineStageFlags waitStageMash = { VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT };
7       VkSubmitInfo submitInfo = {};
8       submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
9       submitInfo.waitSemaphoreCount = 1;
10      submitInfo.pWaitSemaphores = &presentCompleteSemaphore;
11      submitInfo.pWaitDstStageMask = &waitStageMash;
12      submitInfo.commandBufferCount = 1;
13      submitInfo.pCommandBuffers = &context.drawCmdBuffer;
14      submitInfo.signalSemaphoreCount = 1;
15      submitInfo.pSignalSemaphores = &renderingCompleteSemaphore;
16      vkQueueSubmit( context.presentQueue, 1, &submitInfo, renderFence );
17
18      vkWaitForFences( context.device, 1, &renderFence, VK_TRUE, UINT64_MAX );
19      vkDestroyFence( context.device, renderFence, NULL );
20
```

```
21          VkPresentInfoKHR presentInfo = {};
22          presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
23          presentInfo.waitSemaphoreCount = 1;
24          presentInfo.pWaitSemaphores = &renderingCompleteSemaphore;
25          presentInfo.swapchainCount = 1;
26          presentInfo.pSwapchains = &context.swapChain;
27          presentInfo.pImageIndices = &nextImageIdx;
28          presentInfo.pResults = NULL;
29          vkQueuePresentKHR( context.presentQueue, &presentInfo );
30
31          vkDestroySemaphore( context.device, presentCompleteSemaphore, NULL );
32          vkDestroySemaphore( context.device, renderingCompleteSemaphore, NULL );
33      }
```

我们做到了！我们现在已经有了 Vulkan 应用程序运行的基本骨架。希望你能从这篇文章中对 Vulkan 有足够了解并且知道如何继续进行下去。无论如何，这是一个在我开始学习 Vulkan 时想要有的代码链接…所以这会对某些人有一些帮助。
我正在写另一篇教程，在那一篇教程中我详细地讨论了我留下的话题（主要包括着色器统一、材质贴图和基础照明）。所以记得定期来看新的内容。如果我写完并将其发表于此，我也会在我的推特同步更新。如果有建议或是反馈的话请随时联系我，**jhenriques@gmail.com**。
祝你好运，JH。