Vulkan 编程指南

Alexander Overvoorde 著

fangcun 译

日期：September 28, 2022

目录

1 基础代码

1.1 一般结构

在本章节，我们开始使用 Vulkan API 编写代码。

```
#include <vulkan/vulkan.h>

#include <iostream>
#include <stdexcept>
#include <functional>
#include <cstdlib>

class HelloTriangleApplication {
public:
  void run() {
    initVulkan();
    mainLoop();
    cleanup();
  }

private:
  void initVulkan() {

  }

  void mainLoop() {

  }

  void cleanup() {

  }
};

int main() {
  HelloTriangleApplication app;

  try {
    app.run();
  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

代码中，我们首先包含了 Vulkan API 的头文件，它为我们提供了 Vulkan API 的函数，结构体和枚举。此外，包含 stdexcept 和 iostream 头文件用来报错。包含 functional 头文件用于资源管理。包含 cstdlib 头文件用来使用 EXITSUCCESS 和 EXIT_FAILURE 宏。

我们将程序本身包装为一个类，将 Vulkan 对象存储为类的私有成员。我们使用 initVulkan 函数来初始化 Vulkan 对象。初始化完成后，我们进入主循环进行渲染操作。mainLoop 函数包含了一个循环，直到窗口被关闭，才会跳出这个循环。mainLoop 函数返回后，我们使用 cleanup 函数完成资源的清理。

程序在执行过程中，如果发生错误，会抛出一个带有错误描述信息的 std::runtime_error 异常，我们在 main 函数捕获这个异常，并将异常的描述信息打印到控制台窗口。为了处理多种不同类型的异常，我们使用更加通用的 std::exception 来接受异常。一个比较常见的异常就是请求的扩展不被支持。

接下来的每一章，我们会添加新的成员到我们的类中，然后在 initVulkan 函数中初始化它们，在 cleanup 函数中清理它们。

1.2 资源管理

和使用 malloc 函数分配的内存块相同，使用 Vulkan API 创建的 Vulkan 对象也需要在不需要它们时显式地被清除。现代 C++ 可以通过 <memory> 头文件自动地进行资源管理，但在这里，为了让大家更加清晰地理解 Vulkan 对象地创建和清除，以及它们的生命周期，我们没有使用它，而是手动自己完成资源管理。除此之外，Vulkan 的一个核心思想就是通过显式地定义每一个操作来避免出现不一致的现象。

学完本教程后，读者可以通过重载 std::shared_ptr 来实现自动资源管理。将 RAII 应用到自己的程序中。但对于学习而言，最好是能清晰地理解每一个细节部分。

Vulkan 对象可以直接通过类似 vkCreateXXX 的函数，或是通过其它对象调用类似 vkAllocateXXX 的函数创建。当创建的对象不再使用时，使用对应的 vkDestroyXXX 或 vkFreeXXX 函数进行清除操作。这些函数的参数对于不同类型的对象通常是不同的，但都具有一个 pAllocator 参数。我们可以通过这个参数来指定回调函数编写自己的内存分配器。但在本教程，我们没有使用它，将它设置为 nullptr。

1.3 和 GLFW 交互

Vulkan 可以在完全没有窗口的情况下工作，通常，在离屏渲染时会这样做。但一般而言，还是需要一个窗口来显示渲染结果给用户。接下来，我们要完成的就是窗口相关操作。

首先替换代码中的 #include <vulkan/vulkan.h> 为：

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

上面的代码将 GLFW 库的定义包含进来，而 GLFW 库会自动包含 Vulkan 库的头文件。接着，我们添加一个叫做 initWindow 的函数来初始化 GLFW，并在 run 函数中调用它：

```
  void run() {
```

```
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
  }

private:
  void initWindow() {

  }
```

initWIndow 函数首先调用了 glfwInit 函数来初始化 GLFW 库，由于 GLFW 库最初是为 OpenGL 设计的，所以我们需要显式地设置 GLFW 阻止它自动创建 OpenGL 上下文：

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
```

窗口大小变化地处理需要注意很多地方，我们会在以后介绍它，暂时我们先禁止窗口大小改变：

```
glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

接着，我们添加了一个 GLFWwindow* window 变量存储我们创建的窗口句柄：

```
window =glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

glfwCreateWindow 函数的前三个参数指定了要创建的窗口的宽度，高度和标题。第四个参数用于指定在哪个显示器上打开窗口，最后一个参数与 OpenGL 相关，对我们没有意义。

硬编码窗口大小不是一个好习惯，所以我们定义了两个常量，以便之后可以方便地修改它们：

```
const int WIDTH = 800;
const int HEIGHT = 600;
```

现在，我们地 initWindow 函数看起来应该像这样：

```
void initWindow() {
  glfwInit();

  glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
  glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

  window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
}
```

为了确保我们的程序在没有发生错误和窗口没有被关闭的情况下可以一直运行，我们在 mainLoop 函数中添加了下面的事件循环：

```
void mainLoop() {
  while (!glfwWindowShouldClose(window)) {
    glfwPollEvents();
```

```
  }
}
```

上面的代码应该非常直白，每次循环，检测窗口的关闭按钮是否被按下，如果没有被按下，就执行事件处理，否则结束循环。在之后的章节，我们会在这一循环中调用渲染函数来渲染一帧画面。

一旦窗口关闭，我们就可以开始结束 GLFW，然后清除我们自己创建的资源，这在 cleanup 函数中进行：

```
void cleanup() {
  glfwDestroyWindow(window);

  glfwTerminate();
}
```

至此，我们就编写完成了一个可以使用 Vulkan API 的窗口程序骨架。

本章节代码：

C++：

https://vulkan-tutorial.com/code/00_base_code.cpp

2 实例

2.1 创建一个实例

我们首先创建一个实例来初始化 Vulkan 库。这个实例指定了一些驱动程序需要使用的应用程序信息。

我们添加了一个 createInstance 函数调用到 initVulkan 函数中：

```
void initVulkan() {
  createInstance();
}
```

添加了一个存储实例句柄的私有成员：

```
private:
  VkInstance instance;
```

然后，填写应用程序信息，这些信息的填写不是必须的，但填写的信息可能会作为驱动程序的优化依据，让驱动程序进行一些特殊的优化。比如，应用程序使用了某个引擎，驱动程序对这个引擎有一些特殊处理，这时就可能有很大的优化提升：

```
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "Hello Triangle";
appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.pEngineName = "No Engine";
appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.apiVersion = VK_API_VERSION_1_0;
```

之前提到，Vulkan 的很多结构体需要我们显式地在 sType 成员变量中指定结构体的类型。此外，许多 Vulkan 的结构体还有一个 pNext 成员变量，用来指向未来可能扩展的参数信息，现在，我们并没有使用它，将其设置为 nullptr。

Vulkan 倾向于通过结构体传递信息，我们需要填写一个或多个结构体来提供足够的信息创建 Vulkan 实例。下面的这个结构体是必须的，它告诉 Vulkan 的驱动程序需要使用的全局扩展和校验层。全局是指这里的设置对于整个应用程序都有效，而不仅仅对一个设备有效，在之后的章节，我们会对此有更加清晰得认识。

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
```

上面代码中填写得两个参数非常直白，不用多解释。接下来，我们需要指定需要的全局扩展。之前提到，Vulkan 是平台无关的 API，所以需要一个和窗口系统交互的扩展。GLFW 库包含了一个可以返回这一扩展的函数，我们可以直接使用它：

```
uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;

```

```
glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

结构体的最后两个成员变量用来指定全局校验层。我们将在之后的章节更加深入地讨论它，在这里，我们将其设置为 0，不使用它：

```
createInfo.enabledLayerCount = 0;
```

填写完所有必要的信息，我们就可以调用 vkCreateInstance 函数来创建 Vulkan 实例：

```
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

如你所看到的，创建 Vulkan 对象的函数参数的一般形式就是：

- 一个包含了创建信息的结构体指针
- 一个自定义的分配器回调函数，在本教程，我们没有使用自定义的分配器，总是将它设置为 nullptr
- 一个指向新对象句柄存储位置的指针

如果一切顺利，我们创建的实例的句柄就被存储在了类的 VkInstance 成员变量中。几乎所有 Vulkan API 函数调用都会返回一个 VkResult 来反应函数调用的结果，它的值可以是 VK_SUCESS 表示调用成功，或是一个错误代码，表示调用失败。为了检测实例是否创建成功，我们可以直接将创建函数在条件语句中使用，不需要存储它的返回值：

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
  throw std::runtime_error("failed to create instance!");
}
```

现在可以编译运行程序来确保实例创建成功。

2.2 检测扩展支持

如果读者看过 vkCreateInstance 函数的官方文档，可能会知道它返回的之中一个错误代码 VK_ERROR_EXTENSION_NOT_PRESENT。我们可以利用这个错误代码在扩展不能满足时直接结束我们的程序，这对于像窗口系统这种必要的扩展来说非常适合。但有时，我们请求的扩展可能是非必须的，有了很好，没有的话，程序仍然可以运行，这时，我们该怎么做呢?

实际上 Vulkan 提供了一个叫做 vkEnumerateInstanceExtensionProperties 可以在 Vulkan 实例创建之前返回支持的扩展列表。通过它，我们可以获取扩展的个数，以及扩展的详细信息，此外，它还允许我们指定校验层来对扩展进行过滤，但在这里，我们不使用它，将其设置为 nullptr。

我们首先需要知道扩展的数量，以便分配合适的数组大小来存储信息。可以通过下面的代码来获取扩展的数量：

```
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
```

知道了扩展的数量后，我们就可以分配数组来存储扩展信息：

```
std::vector<VkExtensionProperties> extensions(extensionCount);
```

我们使用下面的代码获取所有扩展信息：

```
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data())
    ;
```

每个 VkExtensionProperties 结构体包含了扩展的名字和版本信息。我们可以使用下面的代码将这些信息打印在控制台窗口中 (代码中的
t 表示制表符)：

```
std::cout << "available extensions:" << std::endl;

for (const auto& extension : extensions) {
  std::cout << "\t" << extension.extensionName << std::endl;
}
```

读者可以将上面的代码加入 createInstance 函数，获取扩展支持信息。此外，我们可以编写一个函数来检测调用 glfwGetRequiredInstanceExtensions 函数返回的扩展是否全部包含在了扩展支持列表中。

2.3 清理

VkInstance 应该在应用程序结束前进行清除操作。我们可以在 cleanup 中调用 vkDestroyInstance 函数完成清除工作：

```
void cleanup() {
  vkDestroyInstance(instance, nullptr);

  glfwDestroyWindow(window);

  glfwTerminate();
}
```

vkDestroyInstance 函数的参数非常直白。之前提到，Vulkan 对象的分配和清除函数都有一个可选的分配器回调参数，在本教程，我们没有自定义的分配器，所以，将其设置为 nullptr。除了 Vulkan 实例，其余我们使用 Vulkan API 创建的对象也需要被清除，且应该在 Vulkan 实例清除之前被清除。

创建 Vulkan 实例后，在进行更复杂的操作之前，我们先熟悉一下校验层来帮助我们进行应用程序的调试。

本章节代码：

C++：

https://vulkan-tutorial.com/code/01_instance_creation.cpp

3 校验层

3.1 校验层是什么？

Vulkan API 的设计是紧紧围绕最小化驱动程序开销进行的，所以，默认情况下，Vulkan API 提供的错误检查功能非常有限。很多很基本的错误都没有被 Vulkan 显式地处理，遇到错误程序会直接崩溃或者发生未被明确定义的行为。Vukan 需要我们显式地定义每一个操作，所以就很容易在使用过程中产生一些小错误，比如使用了一个新的 GPU 特性，却忘记在逻辑设备创建时请求这一特性。

然而，这并不意味着我们不能将错误检查加入 API 调用。Vulkan 引入了校验层来优雅地解决这个问题。校验层是一个可选的可以用来在 Vulkan API 函数调用上进行附加操作的组件。校验层常被用来做下面的工作：

- 检测参数值是否合法
- 追踪对象的创建和清除操作，发现资源泄漏问题
- 追踪调用来自的线程，检测是否线程安全。
- 将 API 调用和调用的参数写入日志
- 追踪 API 调用进行分析和回放

下面的代码演示了 Vulkan 的校验层是如何工作的：

```
VkResult vkCreateInstance(
const VkInstanceCreateInfo* pCreateInfo,
const VkAllocationCallbacks* pAllocator,
VkInstance* instance) {

  if (pCreateInfo == nullptr || instance == nullptr) {
    log("Null pointer passed to required parameter!");
    return VK_ERROR_INITIALIZATION_FAILED;
  }

  return real_vkCreateInstance(pCreateInfo, pAllocator, instance);
}
```

校验层可以被自由堆叠包含任何读者感兴趣的调试功能。我们可以在开发时使用校验层，然后在发布应用程序时，禁用校验层来提高程序的运行表现。

Vulkan 库本身并没有提供任何内建的校验层，但 LunarG 的 Vulkan SDK 提供了一个非常不错的校验层实现。读者可以使用这个校验层实现来保证自己的应用程序在不同的驱动程序下能够尽可能得表现一致，而不是依赖于某个驱动程序的未定义行为。

校验层只能用于安装了它们的系统，比如，LunarG 的校验层只可以在安装了 Vulkan SDK 的 PC 上使用。

Vulkan 可以使用两种不同类型的校验层：实例校验层和设备校验层。实例校验层只检查和全局 Vulkan 对象相关的调用，比如 Vulkan 实例。设备校验层只检查和特定 GPU 相关的调用。设备校验层现在已经不推荐使用，也就是说，应该使用实例校验层来检测所有的 Vulkan 调用。

Vulkan 规范文档为了兼容性仍推荐启用设备校验层。在本教程，为了简便，我们为实例和设备指定相同的校验层。

3.2 使用校验层

在本章节，我们将使用 LunarG 的 Vulkan SDK 提供的校验层。和使用扩展一样，使用校验层需要指定校验层的名称。LunarG 的 Vulkan SDK 允许我们通过 VK_LAYER_KHRONOS_validation 来隐式地开启所有可用的校验层。

首先，让我们添加两个变量到程序中来控制是否启用指定的校验层。这里，我们通过条件编译来设定是否启用校验层。代码中的 NDEBUG 宏是 C++ 标准的一部分，表示是否处于非调试模式下：

```
const int WIDTH = 800;
const int HEIGHT = 600;

const std::vector<const char*> validationLayers = {
  "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif
```

接着，我们添加了一个叫做 checkValidationLayerSupport 的函数来请求所有可用的校验层。首先，我们调用 vkEnumerateInstanceLayerProperties 函数获取了所有可用的校验层列表。这一函数的用法和前面我们在创建 Vulkan 实例章节中使用的 vkEnumerateInstanceExtensionProperties 函数相同。

```
bool checkValidationLayerSupport() {
  uint32_t layerCount;
  vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

  std::vector<VkLayerProperties> availableLayers(layerCount);
  vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

  return false;
}
```

接着，检查是否所有 validationLayers 列表中的校验层都可以在 availableLayers 列表中找到：

```
for (const char* layerName : validationLayers) {
  bool layerFound = false;

  for (const auto& layerProperties : availableLayers) {
    if (strcmp(layerName, layerProperties.layerName) == 0) {
```

```
      layerFound = true;
      break;
    }
  }

  if (!layerFound) {
    return false;
  }
}
return true;
```

现在，我们在 createInstance 函数中调用它：

```
void createInstance() {
  if (enableValidationLayers && !checkValidationLayerSupport()) {
    throw std::runtime_error("validation layers requested, but not available!");
  }

    ...
}
```

现在，在调试模式下编译运行程序，确保没有错误出现。如果程序运行时出现错误，请确保正确安装了 Vulkan SDK。如果程序报告缺少可用的校验层，可以查阅 LunarG 的 Vulkan SDK 的官方文档寻找解决方法。

最后，修改我们之前的填写的 VkInstanceCreateInfo 结构体信息，在校验层启用时使用校验层：

```
if (enableValidationLayers) {
  createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
  createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
  createInfo.enabledLayerCount = 0;
}
```

如果校验层检查成功，vkCreateInstance 函数调用就不会返回 VK_ERROR_LAYER_NOT_PRESENT 这一错误代码，但为了保险起见，读者应该运行程序来确保没有问题出现。

3.3 消息回调

仅仅启用校验层并没有任何用处，我们不能得到任何有用的调试信息。为了获得调试信息，我们需要使用 VK_EXT_debug_utils 扩展，设置回调函数来接受调试信息。

我们添加了一个叫做 getRequiredExtensions 的函数，这一函数根据是否启用校验层，返回所需的扩展列表：

```
std::vector<const char*> getRequiredExtensions() {
  uint32_t glfwExtensionCount = 0;
  const char** glfwExtensions;
```

```
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    std::vector<const char*> extensions(glfwExtensions,
    glfwExtensions + glfwExtensionCount);

    if (enableValidationLayers) {
        extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }

    return extensions;
}
```

GLFW 指定的扩展是必需的，调试报告相关的扩展根据校验层是否启用添加。代码中我们使用了一个 VK_EXT_DEBUG_UTILS_EXTENSION_NAME，它等价于 VK_EXT_debug_utils，使用它是为了避免打字时的手误。

现在，我们在 createInstance 函数中调用这一函数：

```
auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
createInfo.ppEnabledExtensionNames = extensions.data();
```

接着，编译运行程序，确保没有出现 VK_ERROR_EXTENSION_NOT_PRESENT 错误。校验层的可用已经隐含说明对应的扩展存在，所以我们不需要额外去做扩展是否存在的检查。

现在，让我们来看接受调试信息的回调函数。我们在程序中以 vkDebugUtilsMessengerCallbackEXT 为原型添加一个叫做 debugCallback 的静态函数。这一函数使用 VKAPI_ATTR 和 VKAPI_CALL 定义，确保它可以被 Vulkan 库调用。

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageType, const
    VkDebugUtilsMessengerCallbackDataEXT* pCallbackData, void* pUserData) {

    std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;

    return VK_FALSE;
}
```

函数的第一个参数指定了消息的级别，它可以是下面这些值：

- VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT：诊断信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT：资源创建之类的信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT：警告信息
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT：不合法和可能造成崩溃的操作信息

这些值经过一定设计，可以使用比较运算符来过滤处理一定级别以上的调试信息：

```
if (messageSeverity >=VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
```

```
  // Message is important enough to show
}
```

messageType 参数可以是下面这些值：

- VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT：发生了一些与规范和性能无关的事件
- VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT：出现了违反规范的情况或发生了一个可能的错误
- VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT：进行了可能影响 Vulkan 性能的行为

pCallbackData 参数是一个指向 VkDebugUtilsMessengerCallbackDataEXT 结构体的指针，这一结构体包含了下面这些非常重要的成员：

- pMessage：一个以 null 结尾的包含调试信息的字符串
- pObjects：存储有和消息相关的 Vulkan 对象句柄的数组
- objectCount：数组中的对象个数

最后一个参数 pUserData 是一个指向了我们设置回调函数时，传递的数据的指针。

回调函数返回了一个布尔值，用来表示引发校验层处理的 Vulkan API 调用是否被中断。如果返回值为 true，对应 Vulkan API 调用就会返回 VK_ERROR_VALIDATION_FAILED_EXT 错误代码。通常，只在测试校验层本身时会返回 true，其余情况下，回调函数应该返回 VK_FALSE。

定义完回调函数，接下来要做的就是设置 Vulkan 使用这一回调函数。我们需要一个 VkDebugUtilsMessengerEXT 对象来存储回调函数信息，然后将它提交给 Vulkan 完成回调函数的设置：

```
VkDebugUtilsMessengerEXT callback;
```

现在，我们在 initVulkan 函数中，在 createInstance 函数调用之后添加一个 setupDebugCallback 函数调用：

```
void initVulkan() {
  createInstance();
  setupDebugCallback();
}

void setupDebugCallback() {
  if (!enableValidationLayers) return;

}
```

接着，我们需要填写 VkDebugUtilsMessengerCreateInfoEXT 结构体所需的信息：

```
VkDebugUtilsMessengerCreateInfoEXT createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
  VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
  VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
```

```
createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
  VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
  VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
createInfo.pfnUserCallback = debugCallback;
createInfo.pUserData = nullptr; // Optional
```

messageSeverity 域可以用来指定回调函数处理的消息级别。在这里，我们设置回调函数处理除了 VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT 外的所有级别的消息，这使得我们的回调函数可以接收到可能的问题信息，同时忽略掉冗长的一般调试信息。

messageType 域用来指定回调函数处理的消息类型。在这里，我们设置处理所有类型的消息。读者可以根据自己的需要开启和禁用处理的消息类型。

pfnUserCallback 域是一个指向回调函数的指针。pUserData 是一个指向用户自定义数据的指针，它是可选的，这个指针所指的地址会被作为回调函数的参数，用来向回调函数传递用户数据。

有许多方式配置校验层消息和回调，更多信息可以参考扩展的规范文档。

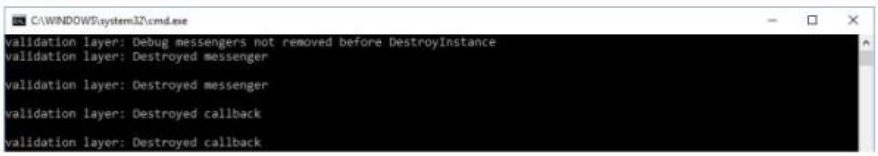
填写完结构体信息后，我们将它作为一个参数调用 vkCreateDebugUtilsMessengerEXT 函数来创建 VkDebugUtilsMessengerEXT 对象。由于 vkCreateDebugUtilsMessengerEXT 函数是一个扩展函数，不会被 Vulkan 库自动加载，所以需要我们自己使用 vkGetInstanceProcAddr 函数来加载它。在这里，我们创建了一个代理函数，来载入 vkCreateDebugUtilsMessengerEXT 函数：

```
VkResult CreateDebugUtilsMessengerEXT(VkInstance instance, const
    VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo, const VkAllocationCallbacks*
    pAllocator, VkDebugUtilsMessengerEXT* pCallback) {
  auto func = (PFN_vkCreateDebugUtilsMessengerEXT)
  vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
  if (func != nullptr) {
    return func(instance, pCreateInfo, pAllocator, pCallback);
  } else {
    return VK_ERROR_EXTENSION_NOT_PRESENT;
  }
}
```

vkGetInstanceProcAddr 函数如果不能被加载，那么我们的代理函数就会发挥 nullptr。现在我们可以使用这个代理函数来创建扩展对象：

```
if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr,
&callback) != VK_SUCCESS) {
  throw std::runtime_error("failed to set up debug callback!");
}
```

函数的第二个参数是可选的分配器回调函数，我们没有自定义的分配器，所以将其设置为 nullptr。由于我们的调试回调是针对特定 Vulkan 实例和它的校验层，所以需要在第一个参数指定调试回调作用的 Vulkan 实例。现在，让我们编译运行程序，如果一切顺利，读者可以看到一个空白窗口，关闭空白窗口后，可以在控制台窗口看到下面的信息：



这说明，我们的程序还存在问题！VkDebugUtilsMessengerEXT 对象在程序结束前通过调用 vkDestroyDebugUtilsMessengerEXT 函数来清除掉。和 vkCreateDebugUtilsMessengerEXT 函数相同，Vulkan 库没有自动加载这个函数，需要我们自己加载它。控制台窗口出现多次相同的错误信息是正常的，这是因为有多个校验层检查发现了这个问题。

现在，让我们创建 CreateDebugUtilsMessengerEXT 函数的代理函数：

```
void DestroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT
    callback, const VkAllocationCallbacks* pAllocator) {
  auto func = (PFN_vkDestroyDebugUtilsMessengerEXT) vkGetInstanceProcAddr(instance,
      "vkDestroyDebugUtilsMessengerEXT");
  if (func != nullptr) {
    func(instance, callback, pAllocator);
  }
}
```

这个代理函数需要被定义为类的静态成员函数或者被定义在类之外。我们在 cleanup 函数中调用这个函数：

```
void cleanup() {
  if (enableValidationLayers) {
    DestroyDebugUtilsMessengerEXT(instance, callback, nullptr);
  }

  vkDestroyInstance(instance, nullptr);

  glfwDestroyWindow(window);

  glfwTerminate();
}
```

现在，再次编译运行程序，如果一切顺利，错误信息这次就不会出现。如果读者想要了解到底是哪个函数调用引发了错误消息，可以在处理消息的回调函数设置断点，然后运行程序，观察程序在断点位置时的调用栈，就可以确定引发错误消息的函数调用。

3.4 配置

校验层除了 VkDebugUtilsMessengerCreateInfoEXT 结构体指定的标志外，还有大量可以决定校验层行为的设置。读者可以浏览 Vulkan SDK 的 Config 目录，里面有一个 vk_layer_settings.txt 解释了如何配置校验层。

读者可以将 vk_layer_settings.txt 复制到自己的项目的 Debug 和 Release 目录来使用它，并按照说明根据需要修改设置。在本教程，我们只使用 vk_layer_settings.txt 的默认设置。

在之后的章节，我们会故意造成一些错误，来演示如何使用校验层来发现这些错误，帮助读者理解校验层的重要性。现在，是时候来看一看系统中的 Vulkan 设备了。

本章节代码：

C++：

https://vulkan-tutorial.com/code/02_validation_layers.cpp

4 物理设备和队列族

4.1 选择一个物理设备

创建 VkInstance 后，我们需要查询系统中的显卡设备，选择一个支持我们需要的特性的设备使用。Vulkan 允许我们选择任意数量的显卡设备，并能够同时使用它们，但在这里，我们只使用第一个满足我们需求的显卡设备。

我们首先添加一个叫做 pickPhysicalDevice 的函数，然后在 initVulkan 函数中调用它：

```
void initVulkan() {
  createInstance();
  setupDebugCallback();
  pickPhysicalDevice();
}

void pickPhysicalDevice() {

}
```

我们使用 VkPhysicalDevice 对象来存储我们选择使用的显卡信息。这一对象可以在 VkInstance 进行清除操作时，自动清除自己，所以我们不需要再 cleanup 函数中对它进行清除。

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

请求显卡列表和请求扩展列表的操作类似，首先需要请求显卡的数量。

```
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
```

如果可用的显卡设备数量为 0，显然应用程序无法继续运行。

```
if (deviceCount == 0) {
  throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
```

获取完设备数量后，我们就可以分配数组来存储 VkPhysicalDevice 对象。

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```

现在，让我们检查获取的设备能否满足我们的需求：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
  return true;
}
```

我们检查设备，并选择使用第一个满足需求的设备：

```
for (const auto& device : devices) {
  if (isDeviceSuitable(device)) {
    physicalDevice = device;
```

```
    break;
  }
}

if (physicalDevice == VK_NULL_HANDLE) {
  throw std::runtime_error("failed to find a suitable GPU!");
}
```

下一节，我们开始具体说明 isDeviceSuitable 函数所进行的检查，随着我们使用的特性增多，这一函数所包含的检查也越来越多。

4.2 设备需求检测

为了选择合适的设备，我们需要获取更加详细的设备信息。对于基础的设备属性，比如名称，类型和支持的 Vulkan 版本的查询可以通过 vkGetPhysicalDeviceProperties 函数进行。

```
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(device, &deviceProperties);
```

纹理压缩，64 位浮点和多视口渲染 (常用于 VR) 等特性的支持可以通过 vkGetPhysicalDeviceFeatures 函数查询：

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
```

有关设备内存和队列族信息的获取，我们会在下一节说明。

现在，假设我们的应用程序只有在显卡支持集合着色器的情况下才可以运行，那么我们的 isDeviceSuitable 函数看起来会像这样：

```
bool isDeviceSuitable(VkPhysicalDevice device) {
  VkPhysicalDeviceProperties deviceProperties;
  VkPhysicalDeviceFeatures deviceFeatures;
  vkGetPhysicalDeviceProperties(device, &deviceProperties);
  vkGetPhysicalDeviceFeatures(device, &deviceFeatures);

  return deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU &&
      deviceFeatures.geometryShader;
}
```

除了直接选择第一个满足需求的设备这种方法，一个更好的方法是给每一个满足需求的设备，按照特性加权打分，选择分数最高的设备使用。具体可以这样做：

```
#include <map>

    ...

void pickPhysicalDevice() {
    ...
```

```
7
8     // Use an ordered map to automatically sort candidates by increasing score
9     std::multimap<int, VkPhysicalDevice> candidates;
10
11    for (const auto& device : devices) {
12      int score = rateDeviceSuitability(device);
13      candidates.insert(std::make_pair(score, device));
14    }
15
16    // Check if the best candidate is suitable at all
17    if (candidates.rbegin()->first > 0) {
18      physicalDevice = candidates.rbegin()->second;
19    } else {
20      throw std::runtime_error("failed to find a suitable GPU!");
21    }
22  }
23
24  int rateDeviceSuitability(VkPhysicalDevice device) {
25      ...
26
27    int score = 0;
28
29    // Discrete GPUs have a significant performance advantage
30    if (deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
31      score += 1000;
32    }
33
34    // Maximum possible size of textures affects graphics quality
35    score += deviceProperties.limits.maxImageDimension2D;
36
37    // Application can't function without geometry shaders
38    if (!deviceFeatures.geometryShader) {
39      return 0;
40    }
41
42    return score;
43  }
```

此外，也可以显示满足需求的设备列表，让用户自己选择使用的设备。

由于我们的教程才刚刚开始，我们现在的唯一需求就是显卡设备需要支持 Vulkan，显然它对于我们使用 Vulkan API 获取的设备列表中的所有设备都永远满足：

4.3 队列族

之前提到，Vulkan 的几乎所有操作，从绘制到加载纹理都需要将操作指令提交给一个队列，然后才能执行。Vulkan 有多种不同类型的队列，它们属于不同的队列族，每个队列族的队列只

允许执行特定的一部分指令。比如，可能存在只允许执行计算相关指令的队列族和只允许执行内存传输相关指令的队列族。

我们需要检测设备支持的队列族，以及它们中哪些支持我们需要使用的指令。为了完成这一目的，我们添加了一个叫做 findQueueFamilies 的函数，这一函数会查找出满足我们需求的队列族。目前而言，我们需要的队列族只需要支持图形指令即可，但在之后的章节，我们可能会有更多的需求。

这一函数会返回满足需求得队列族的索引。这里，我们使用了一个结构体来作为函数返回结果的类型，索引-1 表示没有找到满足需求的队列族：

```
struct QueueFamilyIndices {
  int graphicsFamily = -1;

  bool isComplete() {
    return graphicsFamily >= 0;
  }
};
```

接下来，我们实现 findQueueFamilies 函数：

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
  QueueFamilyIndices indices;

    ...

  return indices;
}
```

我们首先获取设备的队列族个数，然后分配数组存储队列族的 VkQueueFamilyProperties 对象：

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.
    data());
```

VkQueueFamilyProperties 结构体包含了队列族的很多信息，比如支持的操作类型，该队列族可以创建的队列个数。在这里，我们需要找到一个支持 VK_QUEUE_GRAPHICS_BIT 的队列族。

```
int i = 0;
for (const auto& queueFamily : queueFamilies) {
  if (queueFamily.queueCount > 0 && queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT)
      {
    indices.graphicsFamily = i;
  }
```

```
6
7     if (indices.isComplete()) {
8       break;
9     }
10
11    i++;
12  }
```

现在，我们可以在 isDeviceSuitable 函数中调用它来确保我们选择的设备可以执行我们需要的指令：

```
1  bool isDeviceSuitable(VkPhysicalDevice device) {
2    QueueFamilyIndices indices = findQueueFamilies(device);
3
4    return indices.isComplete();
5  }
```

太棒了！我们已经完成了查找我们需要的物理设备这一工作！接下来，让我们创建逻辑设备来使用它！

本章节代码：

C++：

https://vulkan-tutorial.com/code/03_physical_device_selection.cpp

5 逻辑设备和队列

5.1 介绍

选择物理设备后，我们还需要一个逻辑设备来作为和物理设备交互的接口。逻辑设备的创建过程类似于我们之前描述的 Vulkan 实例的创建过程。我们还需要指定使用的队列所属的队列族。对于同一个物理设备，我们可以根据需求的不同，创建多个逻辑设备。

首先，我们添加一个逻辑设备对象作为类成员：

```
VkDevice device;
```

接着，添加一个叫做 createLogicalDevice 的函数，在 initVulkan 函数中调用它。

```
void initVulkan() {
  createInstance();
  setupDebugCallback();
  pickPhysicalDevice();
  createLogicalDevice();
}

void createLogicalDevice() {

}
```

5.2 指定要创建的队列

逻辑设备创建需要填写 VkDeviceQueueCreateInfo 结构体。这一结构体描述了针对一个队列族我们所需的队列数量。目前而言，我们只使用了带有图形能力的队列族。

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily;
queueCreateInfo.queueCount = 1;
```

目前而言，对于每个队列族，驱动程序只允许创建很少数量的队列，但实际上，对于每一个队列族，我们很少需要一个以上的队列。我们可以在多个线程创建指令缓冲，然后在主线程一次将它们全部提交，降低调用开销。

Vulkan 需要我们赋予队列一个 0.0 到 1.0 之间的浮点数作为优先级来控制指令缓冲的执行顺序。即使只有一个队列，我们也要显式地赋予队列优先级：

```
float queuePriority = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriority;
```

5.3 指定使用的设备特性

接下来，我们要指定应用程序使用的设备特性。我们暂时先简单地定义它，之后再回来填写：

```
VkPhysicalDeviceFeatures deviceFeatures = {};
```

5.4 创建逻辑设备

填写好前面两个结构体后，我们可以开始填写 VkDeviceCreateInfo 结构体。

```
VkDeviceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

将 VkDeviceCreateInfo 结构体的 pQueueCreateInfos 指针指向 queueCreateInfo 的地址，pEnabledFeatures 指针指向 deviceFeatures 的地址：

```
createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.queueCreateInfoCount = 1;

createInfo.pEnabledFeatures = &deviceFeatures;
```

结构体的其余成员和 VkInstanceCreateInfo 类似，不同的是这次的设置是针对设备的。

VK_KHR_swapchain 就是一个设备特定扩展的例子，这一扩展使得我们可以将渲染的图像在窗口上显示出来。看起来似乎应该所有支持 Vulkan 的设备都应该支持这一扩展，然而，实际上有的 Vulkan 设备只支持计算指令，不支持这一图形相关扩展。在之后的章节，我们会对交换链进行更加深入地说明。

之前提到，我们可以对设备和 Vulkan 实例使用相同地校验层，不需要额外的扩展支持：

```
createInfo.enabledExtensionCount = 0;

if (enableValidationLayers) {
  createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
  createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
  createInfo.enabledLayerCount = 0;
}
```

现在，我们可以调用 vkCreateDevice 函数创建逻辑设备了。

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
  throw std::runtime_error("failed to create logical device!");
}
```

vkCreateDevice 函数的参数包括我们创建的逻辑设备进行交互的物理设备对象，我们刚刚在结构体中指定的需要使用的队列信息，可选的分配器回调，以及用来存储返回的逻辑设备对象的内存地址。和 Vulkan 实例对象的创建函数类似，这一函数调用在请求的设备需求不被满足时返回错误代码。

逻辑设备对象创建后，应用程序结束前，需要我们自己在 cleanup 函数中调用 vkDestroyDevice 函数来清除它：

```
void cleanup() {
  vkDestroyDevice(device, nullptr);
    ...
}
```

逻辑设备并不直接与 Vulkan 实例交互，所以创建逻辑设备时不需要使用 Vulkan 实例作为参数。

5.5 获取队列句柄

创建逻辑设备时指定的队列会随着逻辑设备一同被创建，为了方便，我们添加了一个 VkQueue 成员变量来直接存储逻辑设备的队列句柄：

```
VkQueue graphicsQueue;
```

逻辑设备的队列会在逻辑设备清除时，自动被清除，所以不需要我们在 cleanup 函数中进行队列的清除操作。

vkGetDeviceQueue 函数可以获取指定队列族的队列句柄。它的参数依次是逻辑设备对象，队列族索引，队列索引，用来存储返回的队列句柄的内存地址。因为，我们只创建了一个队列，所以，可以直接使用索引 0 调用函数：

```
vkGetDeviceQueue(device, indices.graphicsFamily, 0, &graphicsQueue);
```

创建完逻辑设备，我们就可以真正开始使用显卡来完成一些操作。在接下来的章节，我们将开始配置资源，进行一些绘制操作，将渲染结果显示在窗口上。

本章节代码：

C++：

https://vulkan-tutorial.com/code/04_logical_device.cpp