Vulkan guide

Table of Contents

This guide is intended to fill the gaps between the Vulkan documentation and the rest of the GLFW documentation and is not a replacement for either. It assumes some familiarity with Vulkan concepts like loaders, devices, queues and surfaces and leaves it to the Vulkan documentation to explain the details of Vulkan functions.

To develop for Vulkan you should install an SDK for your platform, for example the LunarG Vulkan SDK. Apart from the headers and libraries, it also provides the validation layers necessary for development.

The GLFW library does not need the Vulkan SDK to enable support for Vulkan. However, any Vulkan-specific test and example programs are built only if the CMake files find a Vulkan SDK.

For details on a specific function in this category, see the **Vulkan reference**. There are also guides for the other areas of the GLFW API.

- **Introduction to the API**
- **Window guide**
- **Context guide**
- **Monitor guide**
- **Input guide**

Including the Vulkan and GLFW header files

To include the Vulkan header, define **GLFW_INCLUDE_VULKAN** before including the GLFW header.

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

If you instead want to include the Vulkan header from a custom location or use your own custom Vulkan header then do this before the GLFW header.

```
#include <path/to/vulkan.h>
#include <GLFW/glfw3.h>
```

Unless a Vulkan header is included, either by the GLFW header or above it, any GLFW functions that take or return Vulkan types will not be declared.

The `VK_USE_PLATFORM_*_KHR` macros do not need to be defined for the Vulkan part of GLFW to work. Define them only if you are using these extensions directly.

Querying for Vulkan support

If you are linking directly against the Vulkan loader then you can skip this section. The canonical desktop loader library exports all Vulkan core and Khronos extension functions, allowing them to be called directly.

If you are loading the Vulkan loader dynamically instead of linking directly against it, you can check for the availability of a loader with **glfwVulkanSupported**.

```
if (glfwVulkanSupported())
{
    // Vulkan is available, at least for compute
}
```

This function returns `GLFW_TRUE` if the Vulkan loader was found. This check is performed by **glfwInit**.

If no loader was found, calling any other Vulkan related GLFW function will generate a **GLFW_API_UNAVAILABLE** error.

Querying Vulkan function pointers

To load any Vulkan core or extension function from the found loader, call **glfwGetInstanceProcAddress**. To load functions needed for instance creation, pass `NULL` as the instance.

```
PFN_vkCreateInstance pfnCreateInstance = (PFN_vkCreateInstance)
    glfwGetInstanceProcAddress(NULL, "vkCreateInstance");
```

Once you have created an instance, you can load from it all other Vulkan core functions and functions from any instance extensions you enabled.

```
PFN_vkCreateDevice pfnCreateDevice = (PFN_vkCreateDevice)
    glfwGetInstanceProcAddress(instance, "vkCreateDevice");
```

This function in turn calls `vkGetInstanceProcAddr`. If that fails, the function falls back to a platform-specific query of the Vulkan loader (i.e. `dlsym` or `GetProcAddress`). If that also fails, the function returns `NULL`. For more information about `vkGetInstanceProcAddr`, see the Vulkan documentation.

Vulkan also provides `vkGetDeviceProcAddr` for loading device-specific versions of Vulkan function. This function can be retrieved from an instance with **glfwGetInstanceProcAddress**.

```
PFN_vkGetDeviceProcAddr pfnGetDeviceProcAddr = (PFN_vkGetDeviceProcAddr)
    glfwGetInstanceProcAddress(instance, "vkGetDeviceProcAddr");
```

Device-specific functions may execute a little bit faster, due to not having to dispatch internally based on the device passed to them. For more information about `vkGetDeviceProcAddr`, see the Vulkan documentation.

Querying required Vulkan extensions

To do anything useful with Vulkan you need to create an instance. If you want to use Vulkan to render to a window, you must enable the instance extensions GLFW requires to create Vulkan surfaces.

To query the instance extensions required, call **glfwGetRequiredInstanceExtensions**.

```
uint32_t count;
const char** extensions = glfwGetRequiredInstanceExtensions(&count);
```

These extensions must all be enabled when creating instances that are going to be passed to **glfwGetPhysicalDevicePresentationSupport** and **glfwCreateWindowSurface**. The set of extensions will vary depending on platform and may also vary depending on graphics drivers and other factors.

If it fails it will return `NULL` and GLFW will not be able to create Vulkan window surfaces. You can still use Vulkan for off-screen rendering and compute work.

The returned array will always contain `VK_KHR_surface`, so if you don't require any additional extensions you can pass this list directly to the `VkInstanceCreateInfo` struct.

```
VkInstanceCreateInfo ici;

memset(&ici, 0, sizeof(ici));
ici.enabledExtensionCount = count;
ici.ppEnabledExtensionNames = extensions;
...
```

Additional extensions may be required by future versions of GLFW. You should check whether any extensions you wish to enable are already in the returned array, as it is an error to specify an extension more than once in the `VkInstanceCreateInfo` struct.

Querying for Vulkan presentation support

Not every queue family of every Vulkan device can present images to surfaces. To check whether a specific queue family of a physical device supports image presentation without first having to create a window and surface, call **glfwGetPhysicalDevicePresentationSupport**.

```
if (glfwGetPhysicalDevicePresentationSupport(instance, physical_device,
        queue_family_index))
{
    // Queue family supports image presentation
}
```

The `VK_KHR_surface` extension additionally provides the `vkGetPhysicalDeviceSurfaceSupportKHR` function, which performs the same test on an existing Vulkan surface.

Creating the window

Unless you will be using OpenGL or OpenGL ES with the same window as Vulkan, there is no need to create a context. You can disable context creation with the **GLFW_CLIENT_API** hint.

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
GLFWwindow* window = glfwCreateWindow(640, 480, "Window Title", NULL, NULL);
```

See **Windows without contexts** for more information.

Creating a Vulkan window surface

You can create a Vulkan surface (as defined by the `VK_KHR_surface` extension) for a GLFW window with **glfwCreateWindowSurface**.

```
VkSurfaceKHR surface;
VkResult err = glfwCreateWindowSurface(instance, window, NULL, &surface);
if (err)
{
    // Window surface creation failed
}
```

It is your responsibility to destroy the surface. GLFW does not destroy it for you. Call `vkDestroySurfaceKHR` function from the same extension to destroy it.