Vulkanised 2026

The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

Vulkan Usability

Piers Daniell, NVIDIA

Overview

- Why do we Vulkan?
- The biggest complaints
- How we're improving
- What's available now
- What's coming
- Feedback

Why do we Vulkan?

- **Vulkan is for developers that:**
  - Need high-performance low-overhead access to the GPU
    - Games, game engines, virtual reality, etc.
  - Need to run on as many platforms as possible
    - Window, Linux, Android, consoles, embedded, etc.
    - New GPUs, old GPUs and everything in between
  - Need to research the latest graphics and compute
    - Academics need an open API with the latest innovations
    - Ray tracing, neural shading, gaussian splatting, generative 3D
- **It's an essential tool, but**
  - It's not always easy to use
  - It's not always a joy to use

The biggest complaints

- **Many things are difficult**
  - Descriptors
  - Pipeline layouts and descriptor set layouts
  - Pipelines
  - Synchronization
  - Render passes
  - Swapchains
  - Memory allocation
  - Instance and device creation
- **What's the right way to do something?**
  - API surface area is large and growing
- **How to get started?**
- **Debugging code and crashes**
- **Vulkan code is very verbose!**

Why make it better?

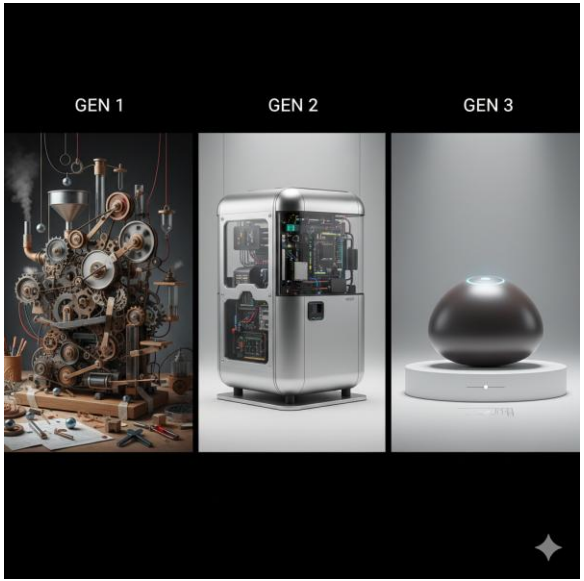
- Make it less painful for those that already do Vulkan
- Make it easier for new developers to get into Vulkan
- Make it approachable for researchers, educators and students
- Expand our market reach
- Our businesses depend on it
- Bring some joy!

The difficult things are getting easier

- **Descriptors: VK_EXT_descriptor_heap**
  - Replaces all other ways to do descriptors
  - Gets rid of descriptor set and pipeline layouts, descriptor sets and pools, image and buffer views, and sampler objects
- **Pipelines: VK_EXT_shader_object, VK_EXT_extended_dynamic_state3**
  - Makes shaders and state easier to work with
  - Eliminates combinatorial explosion
- **Synchronization: VK_KHR_unified_image_layouts**
  - Along with Vulkan 1.3 (VK_KHR_synchonization2) and simplified usage
- **Render passes: Dynamic rendering**
- **Swapchains: Latest maintenance extensions and present id/wait**
  - Usability is a primary focus of ongoing work
- **Memory allocation: VMA is a mature option for a simplified API**
  - But a lot of us need the full API surface

How is this possible?

- **Hardware is getting better and more unified**
  - More bindless – just pointers to memory
  - Allows us to reduce abstractions and API surface
  - More advancements coming in the pipe
- **We're learning from past mistakes**
- **Feedback from you folks!**



Other improvements

- **Debugging and crash tools improving**
  - VVL keeps improving
    - Vulkan Configurator makes it easy to enable and control
  - New API extensions making this possible
  - RenderDoc keeps getting better
  - VK_LAYER_LUNARG_crash_diagnostic helps
- **Legacy functionality being marked as such in the spec**
- **Docs and tutorials are improving**
  - Not only finding the right way to do this
  - But demonstrating it too: https://www.howtovulkan.com/

But what about code verbosity?

- Can we make this:

```
VkFenceCreateInfo fenceInfo;
fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceInfo.pNext = nullptr;
fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
vkCreateFence(device, &fenceInfo, nullptr, &fence);
```

- Look more like this:

```
vkCreateFence(device, { .flags{VK_FENCE_CREATE_SIGNALED_BIT} }, &fence);
```

- And this:

```
VkPipelineInputAssemblyStateCreateInfo inputAssemblyState;
inputAssemblyState.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssemblyState.pNext = nullptr;
inputAssemblyState.flags = 0U;
inputAssemblyState.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssemblyState.primitiveRestartEnable = VK_FALSE;
```

- Like this:

```
VkPipelineInputAssemblyStateCreateInfo inputAssemblyState {
    .topology{VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST} };
```

C++ syntactic sugar

- **Design goals:**
  - C++ façade that's part of vulkan_core.h, the Vulkan spec and man pages
  - Does not deviate far from the Vulkan API – still recognizable Vulkan
  - It does not change abstraction level – just makes existing level sweeter
  - Completely optional to use
  - 100% compatible with existing code
  - Keeps C++ usage very lite
    - Use vulkan.hpp if you want the full experience!
  - Does not change the ABI so no driver or tooling updates are required
  - Allows for ongoing improvements – always backwards compatible
  - Header only additions – no libraries to link or source to compile
  - App source code must target a specific VK_HEADER_VERSION or greater to get the functionality they need
  - Part of the standard Vulkan-Headers repo and Vulkan SDK

Vulkan structure defaulting

- **sType, pNext and other members default initialized (but not *everything*)**
  - Allow compiler to still catch uninitialized members that need app input
  - Opportunistic defaulting where it makes sense

```
typedef struct VkFenceCreateInfo {
    VkStructureType       sType VK_CPP11_DEFAULT( VK_STRUCTURE_TYPE_FENCE_CREATE_INFO );
    const void*           pNext VK_CPP11_DEFAULT( nullptr );
    VkFenceCreateFlags    flags; // We want app to be explicit about this one.
} VkFenceCreateInfo;
```

- **Compiler will still warn with this usage:**

```
VkFenceCreateInfo fenceInfo;
vkCreateFence(device, &fenceInfo, nullptr, &fence);
// warning: fenceInfo.flags used uninitialized
```

- **This zero-defaulting technique is still okay to use:**

```
VkFenceCreateInfo fenceInfo {};
vkCreateFence(device, &fenceInfo, nullptr, &fence);
```

- **This designated-initializer technique works too:**

```
VkFenceCreateInfo fenceInfo { .flags{VK_FENCE_CREATE_SIGNALED_BIT} };
vkCreateFence(device, &fenceInfo, nullptr, &fence);
```

What gets a default?

- Everything except choices that *have* to be made:

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                            sType                  VK_CPP11_DEFAULT(
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO );
    const void*                                pNext                  VK_CPP11_DEFAULT( nullptr );
    VkPipelineInputAssemblyStateCreateFlags    flags                  VK_CPP11_DEFAULT( 0U );
    VkPrimitiveTopology                        topology;
    VkBool32                                   primitiveRestartEnable VK_CPP11_DEFAULT( VK_FALSE );
} VkPipelineInputAssemblyStateCreateInfo;
```

- This one will be a crowd favorite:

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType                            sType                   VK_CPP11_DEFAULT(
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO );
    const void*                                pNext                   VK_CPP11_DEFAULT( nullptr );
    VkPipelineRasterizationStateCreateFlags    flags                   VK_CPP11_DEFAULT( 0U );
    VkBool32                                   depthClampEnable        VK_CPP11_DEFAULT( VK_FALSE );
    VkBool32                                   rasterizerDiscardEnable VK_CPP11_DEFAULT( VK_FALSE );
    VkPolygonMode                              polygonMode             VK_CPP11_DEFAULT( VK_POLYGON_MODE_FILL );
    VkCullModeFlags                            cullMode                VK_CPP11_DEFAULT( VK_CULL_MODE_NONE );
    VkFrontFace                                frontFace               VK_CPP11_DEFAULT( VK_FRONT_FACE_CLOCKWISE );
    VkBool32                                   depthBiasEnable         VK_CPP11_DEFAULT( VK_FALSE );
    float                                      depthBiasConstantFactor VK_CPP11_DEFAULT( 0.0F );
    float                                      depthBiasClamp          VK_CPP11_DEFAULT( 0.0F );
    float                                      depthBiasSlopeFactor    VK_CPP11_DEFAULT( 0.0F );
    float                                      lineWidth               VK_CPP11_DEFAULT( 1.0F );
} VkPipelineRasterizationStateCreateInfo;
```

  - A common source of VVL error!

How it’s done

- All defaults will come from the vk.XML and be documented in the spec
- Use of VK_CPP11_DEFAULT macro to ensure compatibility with C and older C++
- Addition will not affect existing apps

```
#if (defined(__cplusplus) && __cplusplus >= 201103L)
#define VK_CPP11_DEFAULT(DEFAULT) { DEFAULT }
#else
#define VK_CPP11_DEFAULT(DEFAULT)
#endif
```

- We could potentially add a C99 way to initialize objects

```
#define VK_PHYSICAL_DEVICE_VULKAN_1_4_PROPERTIES_DEFAULTS \
    .sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_4_PROPERTIES, \
    .pNext = NULL,

VkPhysicalDeviceVulkan14Properties vulkan14Properties{
        VK_PHYSICAL_DEVICE_VULKAN_1_4_PROPERTIES_DEFAULTS, };
```

- Draft implementation is here: https://github.com/KhronosGroup/Vulkan-Docs/pull/2669

Const ref function overloads

- **“const &” variant for all functions that take “const *”**
  - This facilitates using C++ temporary inline objects for brevity
  - From this:

```
VkFenceCreateInfo fenceInfo { .flags{VK_FENCE_CREATE_SIGNALED_BIT} };
vkCreateFence(device, &fenceInfo, nullptr, &fence);
```

  - To this:

```
vkCreateFence(device, { .flags{VK_FENCE_CREATE_SIGNALED_BIT} }, nullptr, &fence);
```

- **Optional create/destroy functions without pAllocator callbacks**
  - One less thing to worry about for those apps that don’t need it:

```
vkCreateFence(device, { .flags{VK_FENCE_CREATE_SIGNALED_BIT} }, &fence);
vkDestroyFence(device, fence);
```

- **The above takes advantage of designated initializers - new to C++20**
- ***Note: inline C++ overloads will affect Volk and similar tools that use VK_NO_PROTOTYPES, and define Vulkan functions as function pointers:***

```
extern PFN_vkCreateFence vkCreateFence; // <-- Can’t function overload this variable
```

  - This may require a new namespace or function prefixes to workaround

std::span parameters

- **Many functions take in (uint count, const Thing * pThings) pairs:**

```
VkFence myFences[] { fence0, fence1 };
vkWaitForFences(device, size(myFences), myFences, VK_TRUE, UINT64_MAX);
```

- **This would be nicer:**

```
VkFence myFences[] { fence0, fence1 };
vkWaitForFences(device, myFences, VK_TRUE, UINT64_MAX);
```

  - Or even:

```
vkWaitForFences(device, {{fence0, fence1}}, VK_TRUE, UINT64_MAX);
```

- **It can work with vectors, array, c-arrays, etc. naturally**
  - And it's safer
- **std::span was introduced with C++20**
  - *C++26 will make it even nicer (and safer) when using function parameter temporary arrays with span construction from initializer list (P2447R4):*

```
vkWaitForFences(device, {fence0, fence1}, VK_TRUE, UINT64_MAX);
```

How it’s done

- **The C++ overload functions are in the regular Vulkan headers**
  - And in the Vulkan spec and man pages too

```
VKAPI_ATTR VkResult VKAPI_CALL vkCreateFence(
    VkDevice                                    device,
    const VkFenceCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);

#if VK_CPP20_FEATURES
extern "C++" inline VkResult vkCreateFence(VkDevice device, const VkFenceCreateInfo& pCreateInfo, VkFence* pFence)
{
    return vkCreateFence(device, &pCreateInfo, nullptr, pFence);
}
#endif
```

- **The VK_CPP20_FEATURES guard hides them from pre-C++20 and allows them to be optional**
  - C++20 is needed for the std::span variants

```
#if VK_CPP20_FEATURES
extern "C++" inline VkResult vkResetFences(VkDevice device, std::span<const VkFence> pFences)
{
    return vkResetFences(device, uint32_t(pFences.size()), pFences.data());
}
#endif
```

Try it out today

- The Vulkan-Headers repo contains a draft PR you can test:
  - https://github.com/KhronosGroup/Vulkan-Headers/pull/591
- Please leave feedback in the PR or the Vulkan-Docs PRs
- This is based on the recent 1.4.342.0 headers

Instance creation

- **Many layers and extensions to enable**
  - Hooking up Vulkan Validation
  - Enabling WSI for presentation
- ***Lots of code for illustrative purposes only!***

```
    uint32_t layerCount = 0;
    result = vkEnumerateInstanceLayerProperties(&layerCount, 0);
    if (result != VK_SUCCESS) {
        printf("ERROR: vkEnumerateInstanceLayerProperties failed\n");
        return EXIT_FAILURE;
    }

    vector<VkLayerProperties> layerProperties(layerCount);
    result = vkEnumerateInstanceLayerProperties(&layerCount, layerProperties.data());
    if (result != VK_SUCCESS) {
        printf("ERROR: vkEnumerateInstanceLayerProperties failed\n");
        return EXIT_FAILURE;
    }

#if _DEBUG
    const char* optionalLayers[] = { "VK_LAYER_KHRONOS_validation" };
    vector<const char*> enabledLayers(size(optionalLayers));
    uint32_t enabledLayerCount = 0;
    for (uint32_t i = 0; i < size(optionalLayers); ++i) {
        for (uint32_t j = 0; j < layerCount; ++j) {
            if (strcmp(optionalLayers[i], layerProperties[j].layerName) == 0) {
                enabledLayers[enabledLayerCount++] = optionalLayers[i];
                break;
            }
        }
    }
#else
    uint32_t enabledLayerCount = 0;
    char const * * enabledLayers = nullptr;
#endif
```

Instance creation continued

```
uint32_t instanceExtensionsAvailableCount = 0;
    result = vkEnumerateInstanceExtensionProperties(nullptr,
        &instanceExtensionsAvailableCount, nullptr);
    if (result != VK_SUCCESS) {
        return EXIT_FAILURE;
    }
    vector<VkExtensionProperties>
        instanceExtensionsAvailable(instanceExtensionsAvailableCount);
    result = vkEnumerateInstanceExtensionProperties(nullptr,
        &instanceExtensionsAvailableCount, instanceExtensionsAvailable.data());
    if (result != VK_SUCCESS) {
        return EXIT_FAILURE;
    }

#if _DEBUG
    const char* requiredInstanceExtensions[] = { "VK_EXT_debug_utils", "VK_KHR_surface",
        "VK_KHR_surface_maintenance1", "VK_KHR_get_surface_capabilities2" };
    vector<const char*> instanceExtensions(size(requiredInstanceExtensions));
    uint32_t instanceExtensionCount = 0;
    for (uint32_t i = 0; i < size(requiredInstanceExtensions); ++i) {
        bool found = false;
        for (uint32_t j = 0; j < instanceExtensionsAvailableCount; ++j) {
            if (strcmp(requiredInstanceExtensions[i],
                    instanceExtensionsAvailable[j].extensionName) == 0) {
                found = true;
                instanceExtensions[instanceExtensionCount++] =
                    requiredInstanceExtensions[i];
                break;
            }
        }
        if (!found) {
            printf("ERROR: required extension %s not supported\n",
                requiredInstanceExtensions[i]);
            return EXIT_FAILURE;
        }
    }
#else
    uint32_t instanceExtensionCount = 0;
    char const * * instanceExtensions = nullptr;
#endif
```

```
    VkApplicationInfo appInfo = { VK_STRUCTURE_TYPE_APPLICATION_INFO };
    appInfo.pApplicationName = "temp-vulkan";
    appInfo.pEngineName = "temp-vulkan";
    appInfo.apiVersion = VK_API_VERSION_1_3;
    VkInstanceCreateInfo instanceCreateInfo = { VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO };
    instanceCreateInfo.pApplicationInfo = &appInfo;
    instanceCreateInfo.enabledLayerCount = enabledLayerCount;
    instanceCreateInfo.ppEnabledLayerNames = enabledLayers.data();
    instanceCreateInfo.enabledExtensionCount = instanceExtensionCount;
    instanceCreateInfo.ppEnabledExtensionNames = instanceExtensions.data();
    VkInstance instance;
    result = vkCreateInstance(&instanceCreateInfo, 0, &instance);
    if (result != VK_SUCCESS) {
        return EXIT_FAILURE;
    }

#if _DEBUG
    VkDebugUtilsMessengerCreateInfoEXT messengerInfo = {
        VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT };
    messengerInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
    messengerInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT;
    messengerInfo.pfnUserCallback = DebugUtilsMessengerCallbackEXT;
    VkDebugUtilsMessengerEXT messenger;
    result = vkCreateDebugUtilsMessengerEXT(instance, &messengerInfo, nullptr,
        &messenger);
    if (result != VK_SUCCESS) {
        return EXIT_FAILURE;
    }
#endif
```

Device creation

- **Device creation is also hard**
  - Query physical devices, extensions, features and queues
  - Determine if all needed extensions and features are available
  - Enable extensions, queues and features
  - Enable WSI extensions
  - Create device
- **Again, the following code is for illustrative purposes only…**

Device creation continued

```
uint32_t deviceExtensionsAvailableCount = 0;
result = vkEnumerateDeviceExtensionProperties(physicalDevice, 0,
    &deviceExtensionsAvailableCount, 0);
if (result != VK_SUCCESS) {
    printf("ERROR: vkEnumerateDeviceExtensionProperties failed\n");
    return EXIT_FAILURE;
}
vector<VkExtensionProperties>
    deviceExtensionsAvailable(deviceExtensionsAvailableCount);
result = vkEnumerateDeviceExtensionProperties(physicalDevice, 0,
    &deviceExtensionsAvailableCount, deviceExtensionsAvailable.data());
if (result != VK_SUCCESS) {
    printf("ERROR: vkEnumerateDeviceExtensionProperties failed\n");
    return EXIT_FAILURE;
}

const char* requiredDeviceExtensions[] = { "VK_EXT_line_rasterization",
    "VK_NV_linear_color_attachment", "VK_KHR_pipeline_library",
    "VK_EXT_graphics_pipeline_library" };
vector<const char*> deviceExtensions(size(requiredDeviceExtensions));
uint32_t deviceExtensionCount = 0;
for (uint32_t i = 0; i < size(requiredDeviceExtensions); ++i) {
    bool found = false;
    for (uint32_t j = 0; j < deviceExtensionsAvailableCount; ++j) {
        if (strcmp(requiredDeviceExtensions[i],
                deviceExtensionsAvailable[j].extensionName) == 0) {
            found = true;
            deviceExtensions[deviceExtensionCount++] = requiredDeviceExtensions[i];
            break;
        }
    }
    if (!found) {
        printf("ERROR: required extension %s not supported\n",
            requiredDeviceExtensions[i]);
        return EXIT_FAILURE;
    }
}

VkPhysicalDeviceVulkan12Features vulkan12Features = {
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES };
VkPhysicalDeviceVulkan13Features vulkan13Features = {
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES, &vulkan12Features };
VkPhysicalDeviceGraphicsPipelineLibraryFeaturesEXT graphicsPipelineLibraryFeatures = {
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GRAPHICS_PIPELINE_LIBRARY_FEATURES_EXT,
    &vulkan13Features };
VkPhysicalDeviceLinearColorAttachmentFeaturesNV colorAttachmentFeatures = {
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINEAR_COLOR_ATTACHMENT_FEATURES_NV,
    &graphicsPipelineLibraryFeatures };
VkPhysicalDeviceLineRasterizationFeaturesEXT lineRasterizationFeatures = {
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT,
    &colorAttachmentFeatures };
VkPhysicalDeviceFeatures2 features2 = { VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2,
    &lineRasterizationFeatures };
vkGetPhysicalDeviceFeatures2(physicalDevice, &features2);

uint32_t queueFamilyPropertyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyPropertyCount, 0);
vector<VkQueueFamilyProperties> queueFamilyProperties(queueFamilyPropertyCount);
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyPropertyCount,
    queueFamilyProperties.data());

VkDeviceQueueCreateInfo queueInfo = { VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO };
queueInfo.queueFamilyIndex = 0;
queueInfo.queueCount = 1;
float queuePriority = 1.0f;
queueInfo.pQueuePriorities = &queuePriority;
VkDeviceCreateInfo devInfo = { VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
devInfo.pNext = &features2;
devInfo.queueCreateInfoCount = 1;
devInfo.pQueueCreateInfos = &queueInfo;
devInfo.enabledExtensionCount = deviceExtensionCount;
devInfo.ppEnabledExtensionNames = deviceExtensions.data();
VkDevice device;
result = vkCreateDevice(physicalDevice, &devInfo, 0, &device);
if (result != VK_SUCCESS) {
    return EXIT_FAILURE;
}
```

Easier instance and device creation

- **Use Vulkan Configurator to hook up validation and other tools**
  - Part of the Vulkan SDK
- **Use Vulkan Profiles to enable extensions and features**

```cpp
#include <vulkan/vulkan_profiles.hpp>
#include <vulkan/vk_enum_string_helper.h>

int main(int argc, char const* const argv[])
{
    VkResult result;

    VpProfileProperties rm26Profile{ VP_KHR_ROADMAP_2026_NAME, VP_KHR_ROADMAP_2026_MIN_API_VERSION };

    VkApplicationInfo appInfo{ .apiVersion{rm26Profile.specVersion} };
    VkInstanceCreateInfo instanceInfo{ .pApplicationInfo{&appInfo} };
    VpInstanceCreateInfo rm26instanceInfo{ .pCreateInfo{&instanceInfo}, .enabledFullProfileCount{1u}, .pEnabledFullProfiles{&rm26Profile} };
    VkInstance instance;
    result = vpCreateInstance(&rm26instanceInfo, nullptr, &instance);
    if (result != VK_SUCCESS) {
        printf("ERROR: enable to create instance: %s\n", string_VkResult(result));
        return EXIT_FAILURE;
    }

    // Choose physicalDevice

    float queuePriorities[] = { 1.0f };
    VkDeviceQueueCreateInfo queueInfo{ .queueCount{std::size(queuePriorities)}, .pQueuePriorities{queuePriorities} };
    VkDeviceCreateInfo deviceInfo{ .queueCreateInfoCount{1u}, .pQueueCreateInfos{&queueInfo}, };
    VpDeviceCreateInfo rm26DeviceInfo{ .pCreateInfo{&deviceInfo}, .enabledFullProfileCount{1u}, .pEnabledFullProfiles{&rm26Profile} };
    VkDevice device;
    result = vpCreateDevice(physicalDevice, &rm26DeviceInfo, nullptr, &device);
    if (result != VK_SUCCESS) {
        printf("ERROR: unable to create device: %s\n", string_VkResult(result));
        return EXIT_FAILURE;
    }
```

Vulkan Profiles

- **Several profiles come with the Vulkan SDK**
  - Vulkan Roadmap profiles – 2022, 2024 and 2026
  - Android Vulkan profiles – 2021, 2022 and 2025
  - Android Vulkan requirements profiles – 15, 16, 17
  - LunarG desktop profiles
- **Pick one that supports the minimum functionality for your app and target market**
  - Not all are suitable for all developers, but one may work for you
- **Roadmap 2026 is new in latest Vulkan SDK**
  - Great starting point for developers targeting Vulkan 1.4 and modern GPUs
- **Or create your own .json profile!**
  - Arguably easier than lots of manual code
- **More info here: https://github.com/KhronosGroup/Vulkan-Profiles/blob/main/OVERVIEW.md**

vk-bootstrap

- An alternative to help with instance and device creation:
  - https://github.com/charles-lunarg/vk-bootstrap

Vulkan-Hpp

- **A rich C++ interface to the Vulkan API already exists**
  - Supports better type safety, STL, RAII, exception and many helpers
  - Has its own per-device dispatch implementation
- **I encourage C++ developers to take a look**
  - Be warned – don't be scared away by how it's implemented!
  - Just focus on the interface and what it can do for you
- **Already supports all the previously demonstrated C++ features**
- **Beginners and C/C++-lite developers may not like it**
  - The API is similar but not identical to the C API and Vulkan spec
  - It's an all-or-nothing commitment
    - But there are type conversion operators and other things that make it interoperate nicely with the C API
- **Docs are here: https://github.com/KhronosGroup/Vulkan-Hpp**

Future improvement ideas for API usability

- **Improve handles to help polymorphism and abstraction**
  - All handles know their object type and parent
  - **All** handles have private data
- **Add pNext helpers**
  - Ideally the core API should not need pNext chains!
- **Improve array and binary getter functions**
  - std::vector variants perhaps?
- **Borrow more ideas from vulkan.hpp**
- **Add more entry points to vulkan-1.lib**
  - Reduce friction for newbies and tutorials

Feedback

- **We want your feedback!**
- **Test and comment on the pending improvements:**
  - Structure defaulting: https://github.com/KhronosGroup/Vulkan-Docs/pull/2669
  - Inline helpers: https://github.com/KhronosGroup/Vulkan-Docs/pull/2671
- **Make new suggestions for API usability**
  - Make suggestions here: https://github.com/KhronosGroup/Vulkan-Docs/issues