

Validation layers

What are validation layers?

The Vulkan API is designed around the idea of minimal driver overhead and one of the manifestations of that goal is that there is very limited error checking in the API by default. Even mistakes as simple as setting enumerations to incorrect values or passing null pointers to required parameters are generally not explicitly handled and will simply result in crashes or undefined behavior. Because Vulkan requires you to be very explicit about everything you're doing, it's easy to make many small mistakes like using a new GPU feature and forgetting to request it at logical device creation time.

However, that doesn't mean that these checks can't be added to the API. Vulkan introduces an elegant system for this known as *validation layers*. Validation layers are optional components that hook into Vulkan function calls to apply additional operations. Common operations in validation layers are:

- Checking the values of parameters against the specification to detect misuse
- Tracking creation and destruction of objects to find resource leaks
- Checking thread safety by tracking the threads that calls originate from
- Logging every call and its parameters to the standard output
- Tracing Vulkan calls for profiling and replaying

Here's an example of what the implementation of a function in a diagnostics validation layer could look like:

```
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);  
}
```

These validation layers can be freely stacked to include all the debugging functionality that you're interested in. You can simply enable validation layers for debug builds and completely disable them for release builds, which gives you the best of both worlds!

Vulkan does not come with any validation layers built-in, but the LunarG Vulkan SDK provides a nice set of layers that check for common errors. They're also completely open source, so you can check which kind of mistakes they check for and contribute. Using the validation layers is the best way to avoid your application breaking on different drivers by accidentally relying on undefined behavior.

Validation layers can only be used if they have been installed onto the system. For example, the LunarG validation layers are only available on PCs with the Vulkan SDK installed.

There were formerly two different types of validation layers in Vulkan: instance and device specific. The idea was that instance layers would only check calls related to global Vulkan objects like instances, and device specific layers would only check calls related to a specific GPU. Device specific layers have now been deprecated, which means that instance validation layers apply to all Vulkan calls. The specification document still recommends that you enable validation layers at device level as well for compatibility, which is required by some implementations. We'll simply specify the same layers as the instance at logical device level, which we'll see later on.

Using validation layers

In this section we'll see how to enable the standard diagnostics layers provided by the Vulkan SDK. Just like extensions, validation layers need to be enabled by specifying their name. All of the useful standard validation is bundled into a layer included in the SDK that is known as `VK_LAYER_KHRONOS_validation`.

Let's first add two configuration variables to the program to specify the layers to enable and whether to enable them or not. I've chosen to base that value on whether the program is being compiled in debug mode or not. The `NDEBUG` macro is part of the C++ standard and means "not debug".

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

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

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

We'll add a new function `checkValidationLayerSupport` that checks if all of the requested layers are available. First list all of the available layers using the `vkEnumerateInstanceLayerProperties` function. Its usage is identical to that of `vkEnumerateInstanceExtensionProperties` which was discussed in the instance creation chapter.

```

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

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

    return false;
}

```

Next, check if all of the layers in `validationLayers` exist in the `availableLayers` list. You may need to include `<cstring>` for `strcmp`.

```

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;

```

We can now use this function in `createInstance`:

```

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

    ...
}

```

Now run the program in debug mode and ensure that the error does not occur. If it does, then have a look at the FAQ.

Finally, modify the `VkInstanceCreateInfo` struct instantiation to include the validation layer names if they are enabled:

```

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

```

If the check was successful then `vkCreateInstance` should not ever return a `VK_ERROR_LAYER_NOT_PRESENT` error, but you should run the program to make sure.

Message callback

The validation layers will print debug messages to the standard output by default, but we can also handle them ourselves by providing an explicit callback in our program. This will also allow you to decide which kind of messages you would like to see, because not all are necessarily (fatal) errors. If you don't want to do that right now then you may skip to the last section in this chapter.

To set up a callback in the program to handle messages and the associated details, we have to set up a debug messenger with a callback using the `VK_EXT_debug_utils` extension.

We'll first create a `getRequiredExtensions` function that will return the required list of extensions based on whether validation layers are enabled or not:

```

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;
}

```

The extensions specified by GLFW are always required, but the debug messenger extension is conditionally added. Note that I've used the `VK_EXT_DEBUG_UTILS_EXTENSION_NAME` macro here which is equal to the literal string "VK_EXT_debug_utils". Using this macro lets you avoid typos.

We can now use this function in `createInstance` :

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

Run the program to make sure you don't receive a `VK_ERROR_EXTENSION_NOT_PRESENT` error. We don't really need to check for the existence of this extension, because it should be implied by the availability of the validation layers.

Now let's see what a debug callback function looks like. Add a new static member function called `debugCallback` with the `PFN_vkDebugUtilsMessengerCallbackEXT` prototype. The `VKAPI_ATTR` and `VKAPI_CALL` ensure that the function has the right signature for Vulkan to call it.

```
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;
}
```

The first parameter specifies the severity of the message, which is one of the following flags:

- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT` : Diagnostic message
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT` : Informational message like the creation of a resource
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT` : Message about behavior that is not necessarily an error, but very likely a bug in your application
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT` : Message about behavior that is invalid and may cause crashes

The values of this enumeration are set up in such a way that you can use a comparison operation to check if a message is equal or worse compared to some level of severity, for example:

```
if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Message is important enough to show
}
```

The `messageType` parameter can have the following values:

- `VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT` : Some event has happened that is unrelated to the specification or performance

- `VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT` : Something has happened that violates the specification or indicates a possible mistake
- `VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT` : Potential non-optimal use of Vulkan

The `pCallbackData` parameter refers to a `VkDebugUtilsMessengerCallbackDataEXT` struct containing the details of the message itself, with the most important members being:

- `pMessage` : The debug message as a null-terminated string
- `pObjects` : Array of Vulkan object handles related to the message
- `objectCount` : Number of objects in array

Finally, the `pUserData` parameter contains a pointer that was specified during the setup of the callback and allows you to pass your own data to it.

The callback returns a boolean that indicates if the Vulkan call that triggered the validation layer message should be aborted. If the callback returns true, then the call is aborted with the `VK_ERROR_VALIDATION_FAILED_EXT` error. This is normally only used to test the validation layers themselves, so you should always return `VK_FALSE`.

All that remains now is telling Vulkan about the callback function. Perhaps somewhat surprisingly, even the debug callback in Vulkan is managed with a handle that needs to be explicitly created and destroyed. Such a callback is part of a *debug messenger* and you can have as many of them as you want. Add a class member for this handle right under `instance` :

```
VkDebugUtilsMessengerEXT debugMessenger;
```

Now add a function `setupDebugMessenger` to be called from `initVulkan` right after `createInstance` :

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

void setupDebugMessenger() {
    if (!enableValidationLayers) return;
}
```

We'll need to fill in a structure with details about the messenger and its callback:

```
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

```

The `messageSeverity` field allows you to specify all the types of severities you would like your callback to be called for. I've specified all types except for `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT` here to receive notifications about possible problems while leaving out verbose general debug info.

Similarly the `messageType` field lets you filter which types of messages your callback is notified about. I've simply enabled all types here. You can always disable some if they're not useful to you.

Finally, the `pfnUserCallback` field specifies the pointer to the callback function. You can optionally pass a pointer to the `pUserData` field which will be passed along to the callback function via the `pUserData` parameter. You could use this to pass a pointer to the `HelloTriangleApplication` class, for example.

Note that there are many more ways to configure validation layer messages and debug callbacks, but this is a good setup to get started with for this tutorial. See the [extension specification](#) for more info about the possibilities.

This struct should be passed to the `vkCreateDebugUtilsMessengerEXT` function to create the `VkDebugUtilsMessengerEXT` object. Unfortunately, because this function is an extension function, it is not automatically loaded. We have to look up its address ourselves using `vkGetInstanceProcAddr`. We're going to create our own proxy function that handles this in the background. I've added it right above the `HelloTriangleApplication` class definition.

```

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

```

The `vkGetInstanceProcAddr` function will return `nullptr` if the function couldn't be loaded. We can now call this function to create the extension object if it's available:

```

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

```

The second to last parameter is again the optional allocator callback that we set to `nullptr`, other than that the parameters are fairly straightforward. Since the debug messenger is specific to our Vulkan instance and its layers, it needs to be explicitly specified as first argument. You will also see this pattern with other *child* objects later on.

The `VkDebugUtilsMessengerEXT` object also needs to be cleaned up with a call to `vkDestroyDebugUtilsMessengerEXT`. Similarly to `vkCreateDebugUtilsMessengerEXT` the function needs to be explicitly loaded.

Create another proxy function right below `CreateDebugUtilsMessengerEXT`:

```

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

```

Make sure that this function is either a static class function or a function outside the class. We can then call it in the `cleanup` function:

```

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

    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}

```

Debugging instance creation and destruction

Although we've now added debugging with validation layers to the program we're not covering everything quite yet. The `vkCreateDebugUtilsMessengerEXT` call requires a valid instance to have been created and `vkDestroyDebugUtilsMessengerEXT` must be called before the instance is de-

stroyed. This currently leaves us unable to debug any issues in the `vkCreateInstance` and `vkDestroyInstance` calls.

However, if you closely read the [extension documentation](#), you'll see that there is a way to create a separate debug utils messenger specifically for those two function calls. It requires you to simply pass a pointer to a `VkDebugUtilsMessengerCreateInfoEXT` struct in the `pNext` extension field of `VkInstanceCreateInfo`. First extract population of the messenger create info into a separate function:

```
void populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT&
createInfo) {
    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;
}

...

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

    VkDebugUtilsMessengerCreateInfoEXT createInfo;
    populateDebugMessengerCreateInfo(createInfo);

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

We can now re-use this in the `createInstance` function:

```
void createInstance() {
    ...

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

    ...

    VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo{};
    if (enableValidationLayers) {
```

```

        createInfo.enabledLayerCount = static_cast<uint32_t>
(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();

        populateDebugMessengerCreateInfo(debugCreateInfo);
        createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*)
&debugCreateInfo;
    } else {
        createInfo.enabledLayerCount = 0;

        createInfo.pNext = nullptr;
    }

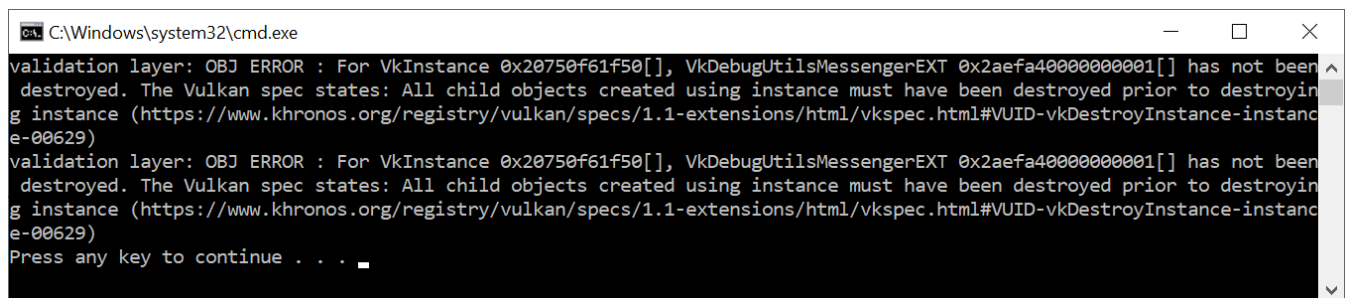
    if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create instance!");
    }
}

```

The `debugCreateInfo` variable is placed outside the if statement to ensure that it is not destroyed before the `vkCreateInstance` call. By creating an additional debug messenger this way it will automatically be used during `vkCreateInstance` and `vkDestroyInstance` and cleaned up after that.

Testing

Now let's intentionally make a mistake to see the validation layers in action. Temporarily remove the call to `DestroyDebugUtilsMessengerEXT` in the `cleanup` function and run your program. Once it exits you should see something like this:



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". It displays two identical Vulkan validation error messages. The first message is: "validation layer: OBJ ERROR : For VkInstance 0x20750f61f50[], VkDebugUtilsMessengerEXT 0x2aefa40000000001[] has not been destroyed. The Vulkan spec states: All child objects created using instance must have been destroyed prior to destroying instance (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#UID-vkDestroyInstance-instance-00629)". The second message is identical. Below the messages, it says "Press any key to continue . . .".

If you don't see any messages then [check your installation](#).

If you want to see which call triggered a message, you can add a breakpoint to the message callback and look at the stack trace.

Configuration

There are a lot more settings for the behavior of validation layers than just the flags specified in the `VkDebugUtilsMessengerCreateInfoEXT` struct. Browse to the Vulkan SDK and go to the `Config` directory. There you will find a `vk_layer_settings.txt` file that explains how to configure the layers.

To configure the layer settings for your own application, copy the file to the `Debug` and `Release` directories of your project and follow the instructions to set the desired behavior. However, for the remainder of this tutorial I'll assume that you're using the default settings.

Throughout this tutorial I'll be making a couple of intentional mistakes to show you how helpful the validation layers are with catching them and to teach you how important it is to know exactly what you're doing with Vulkan. Now it's time to look at [Vulkan devices in the system](#).

[C++ code](#)
