

COMP5822M – Exercise 1.1

Vulkan Initialization

Contents

1	Project overview	1
2	Vulkan Setup (VkInstance)	2
3	Enumerating Vulkan Devices	4
4	Validation (Layers & Ext.)	9
5	Logical Device (VkDevice)	14
A	Building the exercises	17
B	Using Premake	19

This first exercise guides you through the initial steps of working with Vulkan. Make sure you have the necessary software installed before continuing with the following. (See the *Software Setup* document for more information.)

First, the exercise provides a short overview of the exercise's project structure. You will then start work with a simple Vulkan program that loads the Vulkan API, creates a Vulkan instance and examines Vulkan-capable devices on your machine. The exercise will then cover the setup necessary to enable standard validation; this is a key step, as good diagnostics will aid in Vulkan development tremendously. Finally, you will create a logical Vulkan device (VkDevice) handle from one of the found physical devices.

Aside from introducing you to the Vulkan API, this exercise serves as an early check to ensure that you have the necessary software and hardware to complete the practical work in COMP5822M.

```
.
├── labutils
│   └── to_string.{hpp,cpp}
├── premake5, premake5.exe & premake5.apple
├── premake5.lua
├── exercisel
│   └── main.cpp
├── third_party
│   ├── volk
│   │   ├── LICENSE.md & README.md
│   │   ├── include
│   │   │   └── volk
│   │   │       └── volk.h
│   │   └── src
│   │       └── volk.c
│   ├── vulkan
│   │   ├── LICENSE.md & README.txt
│   │   ├── include
│   │   │   ├── ...
│   │   │   ├── vulkan
│   │   │   └── vulkan.h
│   │   └── ...
│   ├── premake5.lua
│   └── third_party.md
```

Listing 1: Project layout.

1 Project overview

Download and unpack the exercise sources. Take a brief look around the project; you can find the rough directory structure in Listing 1. The most important parts are:

exercisel Source code for this exercise. This is where most of your work will take place in Exercise 1.

labutils Code shared across the exercises. Right now this includes a single pair of headers and sources: `to_string.{cpp,hpp}`. These define a few functions that help us convert Vulkan constants to strings, such that we can print Vulkan constants in an easy-to-understand way. Later exercises will add more code to the “labutils”, and you will also end up implementing some of the functions there.

third_party This contains all the third party code that the exercises use. Right now that includes Volk (see below) and the official Vulkan headers. More third party code will be added as we go forward; you can always find brief summaries of and links to all third party code in the `third_party.md` file.

premake* Premake is a meta build-system, not unlike CMake. Premake does not require any installation, which makes it convenient to distribute. The generated project files do not depend on Premake, so they can also be distributed easily.

Third party software

Exercise 1 relies on Volk to load the Vulkan API. Volk was briefly introduced in Lecture 2. Additionally, the exercise includes a copy of the Vulkan headers.

Volk [↗Volk](#) is a meta Vulkan loader. The exercises use it to load the Vulkan API without linking against `vulkan-1.{lib,dll}` (Windows) or `libvulkan.so` (Linux). Instead, Volk locates the standard Vulkan loader at runtime and loads the Vulkan API from it on request. On one hand, this simplifies project setup (we do not need to link against the Vulkan loader, and consequently do not need to find where it was installed), and on the other hand, we can detect if the Vulkan loader is missing and give an appropriate error message.

Volk includes the standard Vulkan headers internally (after setting a few options via `#define`). The Vulkan headers are therefore still required. **Warning:** You should always include `<volk/volk.h>` instead of the standard `<vulkan/vulkan.h>`. If you include the latter without the configuration from Volk, you'll likely run into a number of problems (best case is that the build fails, but it is as likely that the program just crashes when calling into Vulkan).

Vulkan Headers A copy of the Vulkan headers is included in the project to make things a bit easier – by distributing the headers with the project, their location is known. This makes writing the project definitions quite a bit easier (for example, the include paths are set statically in `third_party/premake5.lua` via the `includedirs()` directive).

Why require installation of the Vulkan SDK if the project includes a copy of the standard Vulkan headers and uses Volk to load the Vulkan API at runtime? We still want access to a few other components installed by the SDK. The validation layer that is used in this exercise is one example.



2 Vulkan Setup (VkInstance)

In order to work with Vulkan, we first need to create a Vulkan instance. Take a look at the `main()` function in `exercisel/main.cpp`. You can compile and run the program out-of-the-box (see Appendix A for instructions). Do so. It will print something along the lines of

```
Vulkan loader version: 1.3.268 (variant 0)
```

and exit with an error condition.

The exact version depends on the version of the Vulkan loader that you have installed. (Devices can report a separate version; we'll discuss what the various versions imply later.)

Starting at the top of the `main()` functions, the existing code initializes Volk with `volkInitialize()`. Volk follows the Vulkan API conventions by returning a return code of type [↗VkResult](#). In `volkInitialize()`, Volk will try to locate the Vulkan loader DLL/SO and load a subset of the Vulkan API from it. This is the subset that we require to create a Vulkan instance.

One of loaded functions is [↗vkEnumerateInstanceVersion](#). This function reports the version of the Vulkan loader. Vulkan version numbers are packed into a single 32-bit unsigned integer as documented in section [↗Section 40.2.1](#) for the Vulkan specification. To print it, we need to decode it with a set of macros.

Originally, Vulkan did not have the *variant* field (`VK_API_VERSION_VARIANT`) in the version number; this was added in version 1.2.175. At that point, a new set of macros (`VK_API_VERSION_*`) were introduced to replace the original ones (`VK_VERSION_*`). You will likely still stumble across the latter in some code.



Next, the code attempts to create a Vulkan instance with `create_instance()`. Right now the implementation of `create_instance()` just returns [↗VK_NULL_HANDLE](#). `VK_NULL_HANDLE` is similar to C++'s `nullptr` in that it indicates an invalid object. However, when working with Vulkan handles, you should use `VK_NULL_HANDLE` as using `nullptr` is not guaranteed to work/compile.

Vulkan handles are implemented either as an opaque pointer type or a `std::uint64_t` value. The latter prevents the use of `nullptr` with Vulkan handles. (On 64-bit systems, all Vulkan handles are opaque pointer types, so using `nullptr` will work, but this is bad practice nevertheless.)



We will now implement `create_instance()`. The function is declared at the top of the `main.cpp` file and you can find its current definition just after the body of `main()`.

Vulkan instances are created with the [vkCreateInstance](#) function. Look at the documentation. It takes three arguments:

- `VkInstanceCreateInfo` `const*`
- `VkAllocationCallbacks` `const*`
- `VkInstance*`

and returns a `VkResult` to indicate success/failure.

The last argument (`VkInstance*`) is a pointer to a [VkInstance](#) handle. On success, `vkCreateInstance` will store the resulting instance in this handle.

The second argument ([VkAllocationCallbacks](#)) allows us to assume control over how Vulkan allocates host memory. For most purposes, the system's default allocator is more than sufficient, so we just pass `nullptr` to indicate that Vulkan should use the default allocator. Many of the Vulkan object construction functions have this `VkAllocationCallbacks` argument. The exercises will always set it to `nullptr`.

The first argument, [VkInstanceCreateInfo](#), holds all the parameters that we can pass to the instance creation. The structure is defined as follows (see documentation):

```
typedef struct VkInstanceCreateInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkInstanceCreateFlags flags;
    const VkApplicationInfo* pApplicationInfo;
    uint32_t             enabledLayerCount;
    const char* const*    ppEnabledLayerNames;
    uint32_t             enabledExtensionCount;
    const char* const*    ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

The first two members, `sType` and `pNext`, are found in many of the Vulkan structures. When these two members are present, we need to initialize `sType` to the correct [VkStructureType](#) value. For now, we can set `pNext` to `nullptr`.

As we will see a bit later, the `pNext` pointer is used in conjunction with extensions. The `pNext` can then be pointed to one or more additional Vulkan structures that provide further parameters to the function in question. (The additional Vulkan structure will also have the `pNext` member, which means that another structure can follow there. This creates a chain of additional structures, in the form of a singly linked list.)



The `pNext` pointer has type `void const*`. In order to know what type of structure `pNext` points to, Vulkan can inspect the `sType` field *which is always the first member of such structures*. Similarly, `pNext` is always the second member, meaning that unknown structures could even be skipped without breaking the chain.

The `flags` field is currently unused, and should be set to zero.

The `pApplicationInfo` member, a pointer to [VkApplicationInfo](#), lets the application provide information about itself to Vulkan.

The last four fields (`enabledLayerCount`, ...) are used to load Vulkan layers and to enable instance extensions. We will return to them a bit later in this exercise (Section 4). For now, we do not need them and can just set them to zero and `nullptr`, respectively.

We will start with defining the [VkApplicationInfo](#). Read the documentation (yes, really!). Declare a new `VkApplicationInfo` structure at the top of `create_instance()` and fill it in, for example, as follows:

```
VkInstance create_instance()
{
    // Most of the VkApplicationInfo fields we can choose freely. The only "important" field is the .apiVersion,
    // which specifies the highest version of Vulkan that the application is designed to use.
```

1
2
3
4
5

```

VkApplicationInfo appInfo{};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "COMP5822-exercise1";
appInfo.applicationVersion = 2023; // academic year of 2023/24
appInfo.apiVersion = VK_MAKE_API_VERSION( 0, 1, 3, 0 ); // Version 1.3

```

The `appInfo` structure is zero-initialized thanks to the pair of braces `{}`, which sets all fields in the structure to zero/`nullptr`. We then just specify values for a subset of the fields, leaving e.g. `pEngineName` and `engineVersion` zeroed out.

The most important field is the `apiVersion`. Here, we specify the highest possible Vulkan API version that our application is designed to support. If we were to specify version 1.2, we'd be limited to *at most* Vulkan 1.2 functions. However, it does not guarantee that Vulkan 1.2 will be available. A specific device might still be limited to (for example) Vulkan 1.1. Specifying version 1.3 means that we can use Vulkan functionality from all versions up to 1.3 (when supported), but not any new functions that a future version 1.4 might bring.

Next, directly following, we define the `VkInstanceCreateInfo` structure:

```

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

```

We set the mandatory `sType` and the pointer to the `VkApplicationInfo` that we just created. The other fields are not required at the moment and we can leave them zeroed out.

Now we are ready to call `vkCreateInstance`:

```

VkInstance instance = VK_NULL_HANDLE;

if( auto const res = vkCreateInstance( &instanceInfo, nullptr, &instance ); ▽
    ▷ VK_SUCCESS != res )
{
    std::fprintf( stderr, "Error: unable to create Vulkan instance\n" );
    std::fprintf( stderr, "vkCreateInstance(): %s\n", lut::to_string(res).c_str() );
    return VK_NULL_HANDLE;
}

```

Do not forget to change the final return statement of the `create_instance()` function to return the newly created `instance` instead of `VK_NULL_HANDLE`!

Many Vulkan functions return a `VkResult` return value. It is good practice to check it for errors. In this case, if the value is not `VK_SUCCESS`, we print an error message, and return `VK_NULL_HANDLE` (which then causes the program to exit with an error condition).

Back in `main()`, the program will call `volkLoadInstance()` with our new Vulkan instance. At this point, `Volk` loads the remainder of the Vulkan API (it uses a Vulkan function, [vkGetInstanceProcAddr](#), to do so – this function requires a valid instance handle).

We now have access to the full Vulkan API.

Like many Vulkan objects, we are required to destroy the Vulkan instance when we no longer need it. The code already does this – look for [vkDestroyInstance](#) in `main.cpp`. However, unlike C++, where it is legal to `delete` a `nullptr`-pointer, Vulkan does not allow passing of `VK_NULL_HANDLE` to its destructors.



3 Enumerating Vulkan Devices

Vulkan is designed with systems that have multiple devices (GPUs) in mind. We can list devices that Vulkan knows of (i.e., that have a properly installed Vulkan driver), inspect these, and then select one or more. While Vulkan uses the term *device*, this really refers to a Vulkan implementation. A software implementation would therefore also show up as a device.

Vulkan distinguishes between *physical devices* ([VkPhysicalDevice](#)) and *logical devices* ([VkDevice](#)). Physical devices represent a single Vulkan implementation. When we enumerate available implementations, these will be identified by a `VkPhysicalDevice` handle. Through the handle, we can query Vulkan for information about the underlying device/implementation. We then select one (or more) devices to work with, and create a logical device instance of the selected implementation. The logical device represents an instance of the selected Vulkan

implementation and encapsulates the implementation's state and resources. Each logical device is independent from other logical devices.

It is possible to create multiple logical device instances, each representing a different GPU. We use the logical device handle (`VkDevice`) to identify the device that we want to address with our Vulkan calls. It is even possible to create multiple logical device instances from a single physical device, if necessary. Finally, in some cases, multiple physical devices can form a *device group*, which can then be treated as a single logical device. Throughout these exercises, we will be working with a single logical device, though, and this is by far the most common setting.



In the `main()` method, you will find two calls: `enumerate_devices()` and `select_device()`. We will start with the first one, `enumerate_devices()`. The goal is to query Vulkan for available devices, and print a short list of these along with some of their properties.

Listing Vulkan devices and properties

Jump to the definition of `enumerate_devices()`. Right now it does not do much. We will change it such that it first queries Vulkan for a list of physical devices. Second, it should iterate over the devices, querying Vulkan for information about each.

To query available physical devices, Vulkan provides [vkEnumeratePhysicalDevice](#). We will need to call it twice - first to get the number of physical devices, and second to get a list of physical device handles.

The two steps are a somewhat common pattern in the C-based Vulkan API, and we will see it again later in this exercise. This way, the user is wholly responsible for providing (allocating) the memory in which the list is stored, and Vulkan does not have to concern itself with this.



In the first of the two calls, we pass a pointer to a `std::uint32_t` in the second argument of the function `vkEnumeratePhysicalDevices` and set the third argument to `nullptr`. Vulkan will then set the `std::uint32_t` to the number of devices. In the second call, we set both arguments - the second argument remains the same, but the third one now points at an array of `VkPhysicalDevice` handles with the length returned by the first call (the example below uses `std::vector` to allocate this array). The code could look something like the following:

```
std::uint32_t numDevices = 0;
if( auto const res = vkEnumeratePhysicalDevices( aInstance, &numDevices, nullptr );
    VK_SUCCESS != res )
{
    std::fprintf( stderr, "Error: unable to get physiscal device count\n" );
    std::fprintf( stderr, "vkEnumeratePhysicalDevices() returned error %s\n", lut::
    to_string(res).c_str() );
    return;
}

std::vector<VkPhysicalDevice> devices( numDevices, VK_NULL_HANDLE );
if( auto const res = vkEnumeratePhysicalDevices( aInstance, &numDevices, devices.
    data() ); VK_SUCCESS != res )
{
    std::fprintf( stderr, "Error: unable to get physiscal device list\n" );
    std::fprintf( stderr, "vkEnumeratePhysicalDevices() returned error %s\n", lut::
    to_string(res).c_str() );
    return;
}
```

Unlike most other Vulkan handles, we are not required/allowed to destroy the `VkPhysicalDevice` handles returned by `vkEnumeratePhysicalDevices`.

Next, we iterate over the physical devices:

```
std::printf( "Found %zu devices:\n", devices.size() );
for( auto const device : devices )
{
    //TODO: more code here, see below.
}
```

We can query information about each physical device with the device handle (type `VkPhysicalDevice`). We shall start with some of the device's properties, via

- [vkGetPhysicalDeviceProperties](#) and
- [VkPhysicalDeviceProperties](#).

Take a look at the documentation – in particular, follow links through to `VkPhysicalDeviceLimits` and briefly scroll through the documentation for it.

For now, we will print the device name (`.deviceName`), device type (`.deviceType`), and the two versions. See documentation of [VkPhysicalDeviceType](#) for a list of possible device types. The `to_string`.{hpp,cpp} has a few ready-made helper methods to convert some of the values to strings. Check the sources for additional information.

```

1 // Retrieve basic information (properties) about this device
2 VkPhysicalDeviceProperties props;
3 vkGetPhysicalDeviceProperties( device, &props );
4
5 auto const versionMajor = VK_API_VERSION_MAJOR(props.apiVersion);
6 auto const versionMinor = VK_API_VERSION_MINOR(props.apiVersion);
7 auto const versionPatch = VK_API_VERSION_PATCH(props.apiVersion);
8
9 // Note: while the driver version is stored in a std::uint32_t like the Vulkan API version, the encoding of
10 // the driver version is not standardized. (Some vendors stick to the Vulkan version coding, others do not.)
11 std::printf( "- %s (Vulkan: %d.%d.%d, Driver: %s)\n", props.deviceName,
12             versionMajor, versionMinor, versionPatch, lut::driver_version(props.vendorID,
13             props.driverVersion).c_str() );
14 std::printf( " - Type: %s\n", lut::to_string(props.deviceType).c_str() );

```

Each physical device provides its own Vulkan API version via the `.apiVersion` field – this is in addition to the Vulkan loader version that we queried and printed early on. The latter refers to the version of the Vulkan loader, whereas the per-device versions tell us which version of Vulkan the device’s implementation/driver supports. These two versions do not have to match. Different devices can also report different versions.

For example, the loader can report a higher Vulkan version than the device. In this case, we can only use device-related functions from the lower version, as reported by the device’s driver. The reverse can also occur (but is more rare, since drivers typically distribute a matching Vulkan loader) - however, in this case we can use the higher version API under some conditions (but some features, such as Vulkan layers, may not work). Generally, assume that you can use Vulkan features up to whichever version is lower: the one reported for the device or the one reported by the loader.

Similar to the properties, we can query support for optional device features via either

- [vkGetPhysicalDeviceFeatures](#) and
- [VkPhysicalDeviceFeatures](#)

or

- [vkGetPhysicalDeviceFeatures2](#) and
- [VkPhysicalDeviceFeatures2](#).

The second set of functions was added in Vulkan 1.1. If you look at the documentation, you will find that the new version simply adds the `.sType` and `.pNext` members. Through this we could query information about optional features in extensions (such as features related to the Vulkan ray-tracing implementation).

We will use the newer set of functions; if you for some reason are stuck with a very old Vulkan version, you can use the older API for now. (If this is the case, please get in touch with the module leader ASAP!).

```

1 // vkGetPhysicalDeviceFeatures2 is only available on devices with an API version ≥ 1.1. Calling it on
2 // “older” devices is an error.
3 if( versionMajor > 1 || (versionMajor == 1 && versionMinor >= 1) )
4 {
5     VkPhysicalDeviceFeatures2 features{};
6     features.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2;
7
8     vkGetPhysicalDeviceFeatures2( device, &features );
9
10    // See documentation for VkPhysicalDeviceFeatures2 and for the original VkPhysicalDeviceFeatures
11    // (which is the ‘features’ member of the new version) for a complete list of optional features.
12    //
13    // In addition, several extensions define additional vkPhysicalDevice*Features() structures that could
14    // be queried as well (assuming the corresponding extension is supported).

```



```

    // Anisotropic filtering is nice. You will use it in CW1 on devices that support it.
    std::printf( " - Anisotropic filtering: %s\n", features.features.▽
▷ samplerAnisotropy ? "true" : "false" );
}

```

The code only outputs whether anisotropic filtering (for textures) is available. Look at the various features listed in the `VkPhysicalDeviceFeatures` structure and print other features that you find interesting.

Next, we will briefly inspect the queue families of the device. In order to execute commands on the device, we need to create a queue to which the commands are sent. Each queue comes from a queue family - the queue family determines which types of commands the queue will accept.

We can fetch information about a physical device's queue families with

- [vkGetPhysicalDeviceQueueFamilyProperties](#) and
- [VkQueueFamilyProperties](#)

The most important field of `VkQueueFamilyProperties` is `.queueFlags` ([VkQueueFlagBits](#)). This tells us which types of commands queues from this family will accept. The `.queueCount` field tells us how many queues we can create from this queue family.

The exercises will mainly use a single queue from a queue family with `VK_QUEUE_GRAPHICS_BIT` set. This type of queue accepts graphics commands (as indicated by the flag), and is also required to accept transfer commands (regardless of whether `VK_QUEUE_TRANSFER_BIT` is set – though most drivers seem to set it as well). Transfer commands allow us to copy data to and from the device. There are a few more requirements on queues provided by devices (e.g., if a `GRAPHICS` queue family exists, there must be at least one queue family that supports both `GRAPHICS` and `COMPUTE` commands). See the documentation for `VkQueueFlagBits` for additional requirements.

In addition, a device can support a queue family that only supports a subset of commands. One example is a dedicated `TRANSFER` family. Devices with dedicated hardware (DMA) for transferring data would likely expose such a queue. This could be useful to stream data to the GPU while other processing - such as rendering - takes place. Note that this does not automatically mean that transfers on a dedicated `TRANSFER` queue are faster than on a generic `GRAPHICS` queue. (The transfer queue may not provide higher bandwidth than the graphics queue; instead, operations on the dedicated transfer queue might simply not consume shader compute time.)

Add the following to print the queue families and the commands that each queue family supports. Depending on your GPU, you might only have a single queue family.

```

std::uint32_t numQueues = 0;
vkGetPhysicalDeviceQueueFamilyProperties( device, &numQueues, nullptr );

std::vector<VkQueueFamilyProperties> families( numQueues );
vkGetPhysicalDeviceQueueFamilyProperties( device, &numQueues, families.data() );

for( auto const& family : families )
{
    std::printf( " - Queue family: %s (%u queues)\n", lut::queue_flags(family.▽
▷ queueFlags).c_str(), family.queueCount );
}

```

Finally, we will list information about the device's memory. To do so, we use

- [vkGetPhysicalDeviceMemoryProperties](#) and
- [VkPhysicalDeviceMemoryProperties](#).

```

// Each device has a number of memory types and memory heaps. Heaps correspond to a kind of memory,
// for example system memory or on-device memory (VRAM). Sometimes, there are a few additional
// special memory regions. Memory types use memory from one of the heaps. When allocating memory,
// we select a memory type with the desired properties (e.g., HOST-VISIBLE = must be accessible from
// the CPU).
VkPhysicalDeviceMemoryProperties mem;
vkGetPhysicalDeviceMemoryProperties( device, &mem );

std::printf( " - %u heaps\n", mem.memoryHeapCount );
for( std::uint32_t i = 0; i < mem.memoryHeapCount; ++i )

```

```

{
    std::printf( " - heap %2u: %6zu MBytes, %s\n", i, std::size_t(mem.memoryHeaps[12
    ▷ [i].size)/1024/1024, lut::memory_heap_flags(mem.memoryHeaps[i].flags).c_str() );
}
11
13
14
std::printf( " - %u memory types\n", mem.memoryTypeCount );
for( std::uint32_t i = 0; i < mem.memoryTypeCount; ++i )
15
16
{
    std::printf( " - type %2u: from heap %2u, %s\n", i, mem.memoryTypes[i].▽
17
18
    ▷ heapIndex, lut::memory_property_flags(mem.memoryTypes[i].propertyFlags).c_str() ▽
    ▷ );
}
19

```

There are more per-device values that we could consider. To find others, look for functions that take a `VkPhysicalDevice` handle as their first argument. One specific example is device extensions supported by the device, which we will return to in a later exercise.

For now, build and run the program (if you have not done so already). Look at the information that is being printed. Assuming you are working on a machine with a GPU with dedicated VRAM, can you spot which memory heap this corresponds to?

Selecting a device

We have now learned how to list and inspect available Vulkan devices. Time to select a device. There are many ways of selecting a device, and the correct selection process will ultimately depend on the application you are developing. You might chose to let the user select the device via some sort of interface. However, we will be aiming at an automatic method for the exercises (having to select a specific device each time we run one of the exercises would be extremely annoying!).

The method that we will use will inspect each device in turn and assign a score to the device. We then pick the device that has the highest score.

The score we compute is somewhat arbitrary. As you will see, the exercises will have a strong preference for dedicated GPUs. The idea here is that we want to use the most powerful device (this is not a foolproof way to do so – a very old dedicated GPU is not necessarily faster than a modern integrated GPU). Remember: this might not be the right choice for your application. Indeed, depending on the use case, you might prefer an integrated GPU under the assumption that it consumes less power than dedicated GPUs.

Locate the functions `select_device` and `score_device` in `main.cpp` and study them.

Right now, the function `select_device` operates on an empty vector of `VkPhysicalDevice` handles (see line marked with *//TODO – get list of physical devices!*). In the previous part, we fetched the list of available physical device from Vulkan. Replicate this code here (in this case, you may copy-paste your old code if you so wish; return `VK_NULL_HANDLE` if `vkEnumeratePhysicalDevices` fails).

Next, look at `score_device`. The idea is that the function returns a positive value for any eligible device and `-1` for devices that are not eligible. The latter devices are never selected. Right now, the function always returns `-1`, meaning that we discard all devices.

For now, we will implement `score_device` as follows:

- Check that the device supports at least Vulkan 1.2.
- Give dedicated GPUs a very high score, and integrated GPUs a medium score. All other devices/implementations will get a score of zero. (The exact numbers are arbitrary.)

We will update the method in later exercises with more conditions, but this will suffice for now.

First, inside of `score_device`, get the device's properties:

```

VkPhysicalDeviceProperties props;
vkGetPhysicalDeviceProperties( aPhysicalDev, &props );
1
2

```

Next, check the device's Vulkan version. If it is insufficient, we return `-1` to indicate that the device is ineligible:

```

// Only consider Vulkan 1.2 devices
auto const major = VK_API_VERSION_MAJOR( props.apiVersion );
auto const minor = VK_API_VERSION_MINOR( props.apiVersion );

if( major < 1 || (major == 1 && minor < 2) )
    return -1.f;
1
2
3
4
5
6

```


Finally, check the device type to assign a final score:

```
// Discrete GPU >> Integrated GPU > others
float score = 0.f;

if( VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU == props.deviceType )
    score += 500.f;
else if( VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU == props.deviceType )
    score += 100.f;

return score;
```

With this, compile and run the program. The program should print something along the lines of

```
...
Selected device: NVIDIA GeForce RTX 3080
```

at the end (where, of course, the name of the device will depend on what devices your machine has). Ensure that a device is found. (If not: check that you have at least one \geq Vulkan 1.2 device in your machine).

If a device is found, the application will for the first time exit with a zero status code, to indicate that it was successful. However, we are not quite done yet. The goal is to create a logical device instance from the physical device that was just selected. Before that, we will look at loading the Vulkan standard validation layers. During development, we should always use these, as it will inform us early of various errors (it is also a good idea to develop your programs in such a way that you can test your program at least intermittently – for example, you have been able to build and run the program after each step in these instructions and thereby verify that it is working correctly up to that point!).

4 Validation (Layers & Ext.)

The standard validation is performed by a Vulkan layer named [VK_LAYER_KHRONOS_validation](#). The layer works together with the [VK_EXT_debug_utils](#) instance extension. To enable validation, we need to load the layer, enable the extension and, finally, use functions from the extension to set up our application to receive validation messages.

Both layers and instance extensions are selected when we create the instance,. This is done by passing a list of the layers and extensions that we wish to enable to `vkCreateInstance` via fields in `VkInstanceCreateInfo`:

- Layers: `.enabledLayerCount` and `.ppEnabledLayerNames`
- Extensions: `.enabledExtensionCount` and `.ppEnabledExtensionNames`

(previously we had left these zeroed-out).

To set these fields, first change the declaration of `create_instance` (top of `main.cpp`) to take three arguments:

```
//VkInstance create_instance(); // old declaration
VkInstance create_instance(
    std::vector<char const*> const& aEnabledLayers = {},
    std::vector<char const*> const& aEnabledInstanceExtensions = {} ,
    bool aEnableDebugUtils = false
);
```

The first two arguments are arrays containing the names of all the layers and extensions that should be enabled. We will return to the third argument (`aEnableDebugUtils`) a bit later.

Next, change the definition of `create_instance` similarly:

```
//VkInstance create_instance() // old start of definition
VkInstance create_instance( std::vector<char const*> const& aEnabledLayers, std::vector<char const*> const& aEnabledInstanceExtensions, bool aEnableDebugUtils )
{
    //...
```

Providing the default arguments makes the new declaration mostly compatible with the old one. Will still be able to call `create_instance` without any arguments as before. In fact, after changing both the declaration and the definition, you should be able to compile and run the program (whose behaviour will be unchanged at this point).



Inside of `create_instance` we simply forward the lists of layer and extension names to `VkCreateInstance`. Add the following after the current initialization of the `VkInstanceCreateInfo` struct (but before the call to `vkCreateInstance`):

```
instanceInfo.enabledLayerCount      = std::uint32_t(aEnabledLayers.size());      1
instanceInfo.ppEnabledLayerNames    = aEnabledLayers.data();                    2
                                                                              3
instanceInfo.enabledExtensionCount  = std::uint32_t(aEnabledExtensions.size());  4
instanceInfo.ppEnabledExtensionNames = aEnabledExtensions.data();                5
```

We can now enable layers and extensions by passing them to `create_instance`. However, before we do so, we must check that the layers and extensions are available on the current system.

The processes for checking for available layers and extensions are very similar to each other. In both cases, we query Vulkan for the list of supported layers/extensions:

- [vkEnumerateInstanceLayerProperties](#) for layers, and
- [vkEnumerateInstanceExtensionProperties](#) for extensions

Both functions use the same pattern that we have seen before: we perform two calls, where the first one gives us the number of known layers and extensions, respectively. The second call then retrieves the list of layers/extensions into an array that we have allocated for this purpose.

First, declare two additional helper functions at the top of the file (make sure they are declared inside the anonymous namespace; I put them just above the declaration of `create_instance`).

```
std::unordered_set<std::string> get_instance_layers();      1
std::unordered_set<std::string> get_instance_extensions();  2
```

These helpers will query the list of layers and extensions, and insert them (their names, specifically) into an `std::unordered_set` (a hash table). We can then easily check the hash table to ascertain whether a specific layer or extension is available.

We will now define the function `get_instance_layers`. Again, make sure the definition resides inside of the anonymous namespace; for example, put it just below the definition of `create_instance` after `main()`:

```
std::unordered_set<std::string> get_instance_layers()      1
{                                                          2
    std::uint32_t numLayers = 0;                          3
    if( auto const res = vkEnumerateInstanceLayerProperties( &numLayers, nullptr ); ▽ 4
    ▷ VK_SUCCESS != res )
    {
        std::fprintf( stderr, "Error: unable to enumerate layers\n" );      5
        std::fprintf( stderr, "vkEnumerateInstanceLayerProperties() returned %s\n", ▽ 6
        ▷ lut::to_string(res).c_str() );
        return {};
    }
    std::vector<VkLayerProperties> layers( numLayers );    10
    if( auto const res = vkEnumerateInstanceLayerProperties( &numLayers, layers.data ▽ 12
    ▷ () ); VK_SUCCESS != res )
    {
        std::fprintf( stderr, "Error: unable to get layer properties\n" );    13
        std::fprintf( stderr, "vkEnumerateInstanceLayerProperties() returned %s\n", ▽ 14
        ▷ lut::to_string(res).c_str() );
        return {};
    }
    std::unordered_set<std::string> res;                  18
    for( auto const& layer : layers )                     19
        res.insert( layer.layerName );                   20
    return res;                                           23
}                                                         24
```

The implementation for `get_instance_extensions` is very similar - **implement it as well**. The main difference is the extra argument that `vkEnumerateInstanceExtensionProperties` takes (the first argument). An instance layer can expose additional instance extensions specific to that layer. However, the extensions that

we are looking for are provided by the Vulkan implementation itself, so you should just pass `nullptr` for the `pLayerName` argument.

Now, in `main()`, we can use the helpers that we just created to determine if `VK_LAYER_KHRONOS_validation` and `VK_EXT_debug_utils` are available. After the `std::printf()` that prints the Vulkan loader version, add

```
// Check instance layers and extensions
auto const supportedLayers = get_instance_layers();
auto const supportedExtensions = get_instance_extensions();

bool enableDebugUtils = false;
std::vector<char const*> enabledLayers, enabledExtensions;

# if !defined(NDEBUG) // debug builds only
if( supportedLayers.count( "VK_LAYER_KHRONOS_validation" ) )
{
    enabledLayers.emplace_back( "VK_LAYER_KHRONOS_validation" );
}

if( supportedExtensions.count( "VK_EXT_debug_utils" ) )
{
    enableDebugUtils = true;
    enabledExtensions.emplace_back( "VK_EXT_debug_utils" );
}
# endif // ~ debug builds

// Print the names of the layers and extensions that we are enabling:
for( auto const& layer : enabledLayers )
    std::printf( "Enabling layer: %s\n", layer );

for( auto const& extension : enabledExtensions )
    std::printf( "Enabling instance extension: %s\n", extension );
```

and change the call to `create_instance` to take `enabledLayers` and `enabledExtensions` as arguments:

```
//VkInstance instance = create_instance(); // old call
VkInstance instance = create_instance( enabledLayers, enabledExtensions, ▽
    ▷ enableDebugUtils );
```

In this case, we check if the `VK_LAYER_KHRONOS_validation` layer and the `VK_EXT_debug_utils` extension is supported. If not, we simply skip them. While we really want to use the validation when developing, our program can function perfectly without it. Therefore, if the layer or extension is not available, it makes sense to continue without enabling either. However, this is not always possible, so in some cases it might be necessary to terminate the program with an error if a certain extension (or layer) is not available.

In this case, we use the standard macro `NDEBUG` to determine if we are building a debug version of our program or not. The macro will be defined for release versions, in which case we will not enable the validation layer or the associated extension.

The macro `NDEBUG` also controls `assert()`, which is also disabled in release mode. (Windows developers frequently prefer `_DEBUG`, but this is nonstandard. However, the Premake project files make sure that both `NDEBUG` and `_DEBUG` are defined on all platforms under the right circumstances.)



Build and run the program in debug and release mode. In debug mode, the output should contain the lines

```
...
Enabling layer: VK_LAYER_KHRONOS_validation
Enabling instance extension: VK_EXT_debug_utils
...
```

and in release mode, it should not. (In Visual Studio, you can typically switch the build configuration between *debug* and *release* in the top menu bar. Make sure to switch back to debug once you are done with the test. With the makefiles, use `make config=release_x64` to build the release version.)

The final step is to configure a *debug messenger* (`VkDebugUtilsMessengerEXT`). The debug messenger is responsible for passing messages from e.g. the validation to our application.

At the top of `main.cpp`, in the anonymous namespace, declare the following:


```

VkDebugUtilsMessengerEXT create_debug_messenger( VkInstance );
1
VKAPI_ATTR VkBool32 VKAPI_CALL debug_util_callback( ▽
2
    ▽ VkDebugUtilsMessageSeverityFlagBitsEXT, VkDebugUtilsMessageTypeFlagsEXT, ▽
3
    ▽ VkDebugUtilsMessengerCallbackDataEXT const*, void* );

```

The first function is a helper function in which we will set up the debug messenger. The second function is a callback method that will receive the messages. The callback is documented at

- https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/PFN_vkDebugUtilsMessengerCallbackEXT.html.

The various “decorations” (VKAPI_ATTR and VKAPI_CALL) ensure that the callback uses the correct [calling conventions](#). 

The process of creating the VkDebugUtilsMessengerEXT should be somewhat familiar at this point. Create a function definition for create_debug_messenger in an appropriate place. The debug messenger object is created with [vkCreateDebugUtilsMessengerEXT](#) and [VkDebugUtilsMessengerCreateInfoEXT](#) struct. The key fields are:

- .messageSeverity a bitfield that defines which severity of events will cause our callback to be called
- .messageTypes a bitfield that defines which types of events will cause our callback to be called
- .pfnUserCallback pointer to our callback function

All put together, the function may look something like:


```

VkDebugUtilsMessengerEXT create_debug_messenger( VkInstance aInstance )
1
{
2
    assert( VK_NULL_HANDLE != aInstance );
3
4
    // Set up the debug messaging for the rest of the application
5
    VkDebugUtilsMessengerCreateInfoEXT debugInfo{};
6
    debugInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
7
    debugInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | ▽
8
    ▽ VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
9
    debugInfo.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;
10
    debugInfo.pfnUserCallback = &debug_util_callback;
11
    debugInfo.pUserData = nullptr;
12
13
    VkDebugUtilsMessengerEXT messenger = VK_NULL_HANDLE;
14
    if( auto const res = vkCreateDebugUtilsMessengerEXT( aInstance, &debugInfo, ▽
    ▽ nullptr, &messenger ); VK_SUCCESS != res )
15
    {
16
        std::fprintf( stderr, "Error: unable to set up debug messenger\n" );
17
        std::fprintf( stderr, "vkCreateDebugUtilsMessengerEXT() returned %s\n", lut:: ▽
    ▽ to_string(res).c_str() );
18
        return VK_NULL_HANDLE;
19
    }
20
21
    return messenger;
22
}

```

There are two additional severity levels that were not specified in the code above:

- VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT
- VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT

I tend to find that they produce a bit too much spam to be all that useful, so I typically leave those disabled. However, feel free to experiment with these and see what kind of output they produce. In contrast, the above code selects all message types. 

We also need to define the callback implementation. The only thing it does is print a message based on the information passed to it and return VK_FALSE.

```

VKAPI_ATTR VkBool32 VKAPI_CALL debug_util_callback( ▽
1
    ▽ VkDebugUtilsMessageSeverityFlagBitsEXT aSeverity, ▽
    ▽ VkDebugUtilsMessageTypeFlagsEXT aType, VkDebugUtilsMessengerCallbackDataEXT ▽
    ▽ const* aData, void* /*aUserPtr*/ )
2
{

```

```

    std::fprintf( stderr, "%s (%s): %s (%d)\n%s\n--\n", lut::to_string(aSeverity).▽ 3
    ▷ c_str(), lut::message_type_flags(aType).c_str(), aData->pMessageIdName, aData->▽
    ▷ messageIdNumber, aData->pMessage );
    return VK_FALSE;
}

```

The specification (see documentation for [the callback](#)) requires us to return `VK_FALSE`, mentioning that returning `VK_TRUE` is reserved for layer development.



Finally we need to call `create_debug_messenger` to actually perform the initialization. In `main()`, immediately after loading the instance API with `vkLoadInstance`, insert

```

// Setup debug messenger
VkDebugUtilsMessengerEXT debugMessenger = VK_NULL_HANDLE;
if( enableDebugUtils )
{
    debugMessenger = create_debug_messenger( instance );
}

```

Build and run the program (make sure to use debug mode such that the validation gets enabled). At this point, we should learn that our program contains an error:

```

...
SEVERITY_ERROR (VALIDATION): VUID-vkDestroyInstance-instance-00629 (-1958900200)
Validation Error: [ VUID-vkDestroyInstance-instance-00629 ] Object 0: handle = 0
x55d35f8f0fc0, type = VK_OBJECT_TYPE_INSTANCE; Object 1: handle = 0xfd5b260000000001,
type = VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT; | MessageID = 0x8b3d8e18 |
vkDestroyInstance(): OBJ ERROR : For VkInstance 0x55d35f8f0fc0[],
VkDebugUtilsMessengerEXT 0xfd5b260000000001[] 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/1.3-extensions/html/
vkspec.html#VUID-vkDestroyInstance-instance-00629)
--

```

The wall of text that is this error message can take a bit to decode, so spend some time parsing the output. Essentially, the message tells us that we forgot to destroy the debug messenger object that we just created, and then points us at the relevant portion of the Vulkan specification.

This is easily fixed. At the end of the `main()` function, just before the instance is destroyed, we need to destroy the debug messenger:

```

if( VK_NULL_HANDLE != debugMessenger )
    vkDestroyDebugUtilsMessengerEXT( instance, debugMessenger, nullptr );

```

Build and run the program again. There should be no more validation errors at this point.

The Linux machines in the 24h teaching lab might print a few *warnings*, i.e. with `SEVERITY_WARNING` instead of `SEVERITY_ERROR`. The warnings would say something about “Loader Message” and either mismatching loader interface versions or being unable to create an ICD instance. You can ignore these warnings for now. They are mostly related to having old driver versions installed on the University of Leeds systems.



Validation during instance creation

We create the debug messenger after we have created the instance. Because of this, we cannot receive any validation messages about the instance creation – we have not yet created the debug messenger object, so it is impossible for our application to receive debug messages. This is clearly not ideal.

`VK_EXT_debug_utils` provides a way to also receive messages about the instance creation. The idea is that we pass an instance of `VkDebugUtilsMessengerCreateInfoEXT` to `vkCreateInstance`. This way, the validation layer can find the debug messenger callback information already when the instance is being created (along with our specification for what types of messages we wish to receive).

To pass an instance of `VkDebugUtilsMessengerCreateInfoEXT` to `vkCreateInstance`, we use the `sType` and `pNext` mechanism: we point the `pNext` field of the `VkInstanceCreateInfo` structure to an instance of the `VkDebugUtilsMessengerCreateInfoEXT` structure.

In practice, this may look like the following (insert just after filling in the `instanceInfo` structure and before calling `vkCreateInstance` in `create_instance`):

```
// If we wish to receive validation information on instance creation, we need to provide additional information 1
// to vkCreateInstance(). This is done by linking an instance of VkDebugUtilsMessengerCreateInfoEXT into 2
// the pNext chain of VkInstanceCreateInfo. 3
// 4
// Note: this is identical to the one that we used when initializing the debug utils ordinarily 5
// (see create_debug_messenger()). 6
VkDebugUtilsMessengerCreateInfoEXT debugInfo{}; 7

8
if( aEnableDebugUtils ) 9
{ 10
    debugInfo.sType = ▽ 11
    ▽ VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
    debugInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | ▽ 12
    ▽ VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
    debugInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | ▽ 13
    ▽ VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT | ▽
    ▽ VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
    debugInfo.pfnUserCallback = &debug_util_callback; 14
    debugInfo.pUserData = nullptr; 15

    16
    debugInfo.pNext = instanceInfo.pNext; 17
    instanceInfo.pNext = &debugInfo; 18
} 19
```

This does not replace the previous debug messenger setup. We still need to create the debug messenger object if we wish to receive validation messages from other parts of the program.



5 Logical Device (VkDevice)

Now that we have the validation set up, we will finally create the logical device instance.

When creating a device, we need to minimally request one queue. (Without a queue, we could not perform any processing the device, so this makes sense.) Since we are mainly interesting in using the device for graphics operations in later exercises, we will request a queue with `VK_QUEUE_GRAPHICS_BIT` set.

Creating a logical device therefore involves two steps at this point:

- Identify the index of the queue family that has `VK_QUEUE_GRAPHICS_BIT` set.
- Create a device with [vkCreateDevice](#) and [VkDeviceCreateInfo](#).

We will also need the `VkPhysicalDevice` handle to the physical device that we selected earlier.

The key elements of `VkDeviceCreateInfo` are:

- We request queues through `.queueCreateInfoCount` and `.pQueueCreateInfos`.
- `.enabledExtensionCount` and `.ppEnabledExtensionNames` allow us to enable device extensions on the logical device.
- `.pEnabledFeature` allows us to enable optional features on the logical device.

For now, we are not enabling any extra features or extensions and therefore leave the corresponding fields at zero/`nullptr`.

We request queues by filling in one or more [VkDeviceQueueCreateInfo](#) structures and pointing the relevant fields in `VkDeviceCreateInfo` at them.

Practically, we will implement this with two functions. Declare them at the top of `main.cpp` as usual:

```
std::optional<std::uint32_t> find_graphics_queue_family( VkPhysicalDevice ); 1
2
VkDevice create_device( VkPhysicalDevice, std::uint32_t aQueueFamily ); 3
```

The first one will enumerate the physical device's queue families and find the index of the appropriate family. It is possible that no such family exists, so we return the index through an `std::optional` object. This allows us to indicate if no index was found. The second function will create the logical device.

Define the functions somewhere below `main()` again. We already inspected devices' queue families when enumerating physical devices and printing their properties. The following should therefore look familiar:

```
std::optional<std::uint32_t> find_graphics_queue_family( VkPhysicalDevice ▽ 1
    ▷ aPhysicalDev )
{ 2
    assert( VK_NULL_HANDLE != aPhysicalDev ); 3

    4
    std::uint32_t numQueues = 0; 5
    vkGetPhysicalDeviceQueueFamilyProperties( aPhysicalDev, &numQueues, nullptr ); 6
    7
    std::vector<VkQueueFamilyProperties> families( numQueues ); 8
    vkGetPhysicalDeviceQueueFamilyProperties( aPhysicalDev, &numQueues, families.data() ▽9
    ▷ () );
    10
    for( std::uint32_t i = 0; i < numQueues; ++i ) 11
    { 12
        auto const& family = families[i]; 13
        14
        if( VK_QUEUE_GRAPHICS_BIT & family.queueFlags ) 15
            return i; 16
        17
    } 18
    19
    return {}; 20
}
```

Creating the logical device with `vkCreateDevice` follows the same pattern that we have seen before:

```
VkDevice create_device( VkPhysicalDevice aPhysicalDev, std::uint32_t aQueueFamily ) 1
{ 2
    assert( VK_NULL_HANDLE != aPhysicalDev ); 3

    4
    float queuePriorities[1] = { 1.f }; 5
    6
    VkDeviceQueueCreateInfo queueInfo{}; 7
    queueInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO; 8
    queueInfo.queueFamilyIndex = aQueueFamily; 9
    queueInfo.queueCount = 1; 10
    queueInfo.pQueuePriorities = queuePriorities; 11
    12
    VkPhysicalDeviceFeatures deviceFeatures{}; 13
    // No extra features for now. 14
    15
    VkDeviceCreateInfo deviceInfo{}; 16
    deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO; 17
    18
    deviceInfo.queueCreateInfoCount = 1; 19
    deviceInfo.pQueueCreateInfos = &queueInfo; 20
    21
    deviceInfo.pEnabledFeatures = &deviceFeatures; 22
    23
    VkDevice device = VK_NULL_HANDLE; 24
    if( auto const res = vkCreateDevice( aPhysicalDev, &deviceInfo, nullptr, &device ▽25
    ▷ ); VK_SUCCESS != res )
    { 26
        std::fprintf( stderr, "Error: can't create logical device\n" ); 27
        std::fprintf( stderr, "vkCreateDevice() returned %s\n", lut::to_string(res). ▽28
        ▷ c_str() );
        return VK_NULL_HANDLE; 29
    } 30
    31
    return device; 32
}
```

Finally, in `main()` we put the pieces together (after selecting a physical device):

```
// Create a logical device
```

1

```

auto const graphicsFamilyIndex = find_graphics_queue_family( physicalDevice );
if( !graphicsFamilyIndex )
{
    vkDestroyInstance( instance, nullptr );
    std::fprintf( stderr, "Error: no graphics queue found\n" );
    return 1;
}

VkDevice device = create_device( physicalDevice, *graphicsFamilyIndex );
if( VK_NULL_HANDLE == device )
{
    vkDestroyInstance( instance, nullptr );
    return 1;
}

```

Now that we have a valid logical device, we can optionally instruct Volk to load function pointers directly to the device's implementation, potentially bypassing the indirection from the Vulkan loader:

```

// Optional: Specialize Vulkan functions for this device
volkLoadDevice( device );

```

Additionally, we should retrieve the handle to the queue that we requested with our device. We do so with [vkGetDeviceQueue](#):

```

// Retrieve VkQueue
VkQueue graphicsQueue = VK_NULL_HANDLE;
vkGetDeviceQueue( device, *graphicsFamilyIndex, 0, &graphicsQueue );

assert( VK_NULL_HANDLE != graphicsQueue );

```

The `VkQueue` object is somewhat special in that we do not need to (and in fact, cannot) destroy it explicitly. However, we do need to destroy the device at the end of our program, so make sure to call `vkDestroyDevice` in the *Cleanup* section at the bottom of the `main()` method (make sure the Vulkan objects are destroyed in the reversed order in which they were created).

Build and run the program in both debug and release mode. The program should execute successfully in both cases. Further, in debug mode, there should be no validation messages at this point.

Next up...





This exercise provided a fairly detailed guide on the Vulkan initialization. You should now have a program that initializes Vulkan all the way to creating a logical device. We also enable the validation, which will be extremely helpful during development.

The validation layer reminded us of the importance of properly cleaning up after ourselves. Right now, we are doing this manually, which is quite error prone. (In fact, the program still forgets to completely clean up under some conditions. Can you identify when this occurs?)

In the next exercise, we will ultimately set up a graphics pipeline and render a small image. This process involves creating many more Vulkan objects, most of which we need to remember to cleanup when appropriate. Therefore, the next exercise also introduces some code to automate the cleanup process, ensuring that we do not forget to do so.

This will enable a second improvement. Right now we are manually returning error codes and checking whether our functions in fact succeeded. When the cleanup process is automated (using C++'s destructor mechanism), we can instead switch to exceptions for reporting errors. This eliminates some manual work and reduces the amount of code somewhat as well. (You will probably, at this point, agree that we will have enough code either way. :-))

Acknowledgements

The document uses icons from <https://icons8.com>: , , , . The "free" license requires attribution in documents that use the icons. Note that each icon is a link to the original source thereof.

A Building the exercises

This section covers brief instructions on building the exercises.

Debug vs. Release build configurations

It's a common practice to have multiple *build configurations*, or *builds* for short. Different builds typically use different compiler options, but may also subtly change the code that is compiled. In some cases, different builds will also link against different libraries.

The most common setup is to have a *debug build* and a *release build*. The debug build typically disables (or limits) compiler optimizations and asks the compiler to generate debug information. This makes it easier to debug the code using standard debugging tools. The release build, in contrast, enables optimizations and does not generate debug information. This makes it more difficult to debug a program, but can significantly boost runtime performance. It also reduces the size of the resulting binaries. Compilation of debug builds is frequently a bit faster, compared to release builds (aside from optimizations taking time, tools like Visual Studio may utilize incremental builds to cut down build times further).

The base code defines both a debug and a release build. Instructions for selecting build types is included below. You typically want to do most of your work with debug builds (such that you can easily use debuggers and similar tools). You should nevertheless test the release build occasionally. Coursework code *must* be tested in both build variants.

In addition to the differences listed above, the handed-out code changed behaviour slightly between debug and release builds. First, the code uses the standard `assert()` macro to perform additional run-time checks in debug builds. The macro is ignored (compiled out) in release build. Secondly, in debug builds, exercises will automatically enable the Vulkan validation layers.

Building on Linux (command line)

If you are in the 24h teaching lab (2.15) in Bragg, you should start by loading the gcc module:

```
> module load dependent-modules
> module load gcc
```

Note: don't type the '`>`' symbol! This is used to distinguish input lines (commands that you type) from output generated by the command.

(2023/24) This may print a warning about `gcc-runtime/8.5.0-np7rz` not being loaded. It seems to work nevertheless, so you can ignore it for the time being.



The machines in the teaching lab are occasionally in a ... suboptimal state. In particular, some machines may not have modules available (this seems to be caused by network filesystems not being mounted successfully). You can check for this with the command

```
> module avail
```



If the list is very short (e.g., just two modules), there are some problems. You can try using the built-in GCC (version 8.5), but this is somewhat untested.

If the GCC module loads successfully, you will now have a more recent GCC compiler version (13.2.0). Verify that this is the case by checking the version of the C++ compiler, `g++`:

```
> g++ --version
g++ (Spack GCC) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
(...)
```

The output should contain a line similar to something like `g++ (Spack GCC) 13.2.0`. Verify that the version number at the end is correct. You will need to do this in every terminal/shell.

If you are using a non-standard machine (e.g., own laptop), you probably don't need to load modules as described above. However, you should still verify that you have a recent GCC available on your machine.

The next step is generating Makefiles with Premake. The handed-out code includes a Premake binary for this purpose. To generate Makefiles, specify the `gmake2` action:

```
./premake5 gmake2
```

Appendix B contains further information about Premake.

Whenever you add new source files to the project (or delete existing ones), you must rerun Premake as above. This will update the Makefiles to include those changes. You do not have to rerun Premake if you just change existing files.



Under no circumstances should you edit the generated Makefiles.

Now, you can simply use `make` to build the project:

```
> make -j6
==== Building x-glad (debug_x64) ====
==== Building x-glfw (debug_x64) ====
(...)
Linking exercisel
```

The `-j6` flag indicates to make that `make` should run up to six build processes in parallel. You can adjust this number, somewhat depending on the number of CPU cores that you have in your machine (check `/proc/cpuinfo`).

The default is to perform a *debug* build. This is reflected in the name of the final binary, which you can find in the `bin` directory. Run it:

```
> ./bin/exercisel-debug-x64-gcc.exe
```

(The generated binary gets a `.exe` extension even on Linux. This is non-standard, but has proven helpful in the past.)

Launching the application should open a empty (black) window. You can close it by pressing the `Escape` key (or through any standard means).

To perform a *release* build, use:

```
> make -j6 config=release_x64
==== Building x-glad (release_x64) ====
==== Building x-glfw (release_x64) ====
(...)
Linking exercisel
```

The `x64`-suffix refers to the platform. In this case, the build targets a 64-bit x86 architecture.

Building on Windows (Visual Studio)

If you own a Windows machine, you can opt to use Visual Studio for your work. Visual Studio has in recent years gained a very competent and up-to-date C++ compiler. It also includes one of the best and easy-to-use debuggers. The Visual Studio community edition is available for free for individuals and for academic use. For details, see the separate document that describes how to get and install Visual Studio on your own machine. The following assumes that you have set up Visual Studio 2022 for C++ development on your machine.

Visual Studio, referred to above, is not the same as Visual Studio Code. Although the names are similar, Code is a completely different product. The instructions herein will likely not work for Code. Instead, use the proper Visual Studio!



You will first need to generate Visual Studio 2022 project files. To do so, you will need to run Premake on the command line (`cmd.exe`) with the `vs2022` action:

```
.\premake5.exe vs2022
```

This creates the main Visual Studio solution file (`COMP5822M-exercisel.sln`) and the associated project files. The solution file will be located in the root directory of the project. Open it to launch Visual Studio.

Whenever you add new source files to the project (or delete existing ones), you must rerun Premake as above. This will update the Visual Studio project files to include those changes. You do not have to rerun Premake if you just change existing files.



To build the project(s) in Visual Studio, you can either explicitly issue a build command via the *Build* menu (Figure 1), or by launching a program from the *Debug* menu (Figure 2). The latter will try to build (or ask if it should try to build) the program before running it.

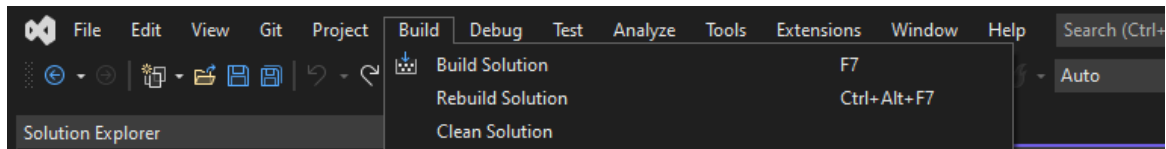


Figure 1: Visual Studio's Build menu. There are further options in the menu that might be relevant to you – check them out.

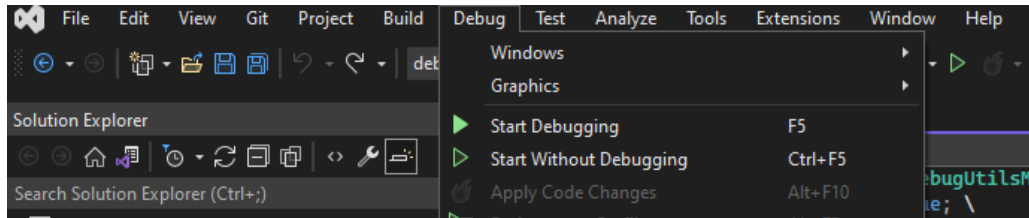


Figure 2: Visual Studio's Debug menu. From here, you can launch the currently selected project ("startup project") either in the Visual Studio debugger or without it.

In the debug menu, you can launch the program either in the debugger (*Start Debugging*, F5) or "normally" (*Start Without Debugging*, Ctrl+F5). The latter option is useful for quick tests, as it launches the program without switching over the Visual Studio interface to the debugger. Either option will try to launch the project that is currently designed as the *Startup Project*. To change the project, right-click on any project in the *Solution Explorer* and designate it as *Startup Project*. (Note that trying to start a library project, rather than an application project will not do much.)

The options *Rebuild Solution*/*Clean solution* in the *Build* menu can be useful if you run into mysterious errors that cause linking (and sometimes even compilation) to fail. This doesn't happen very often, but it's a relatively cheap option to try for smaller projects. Note that cleaning a solution will *not* get rid of all of Visual Studio's temporary files or build artifacts. You will still have to pay attention what files you include in e.g. coursework submissions.

To switch between debug and release build (Visual Studio refers to these as "configurations"), use the small pull-down menu that is by default located below the *Build* and *Debug* menus.

Building on MacOS (command line)

MacOS users should be able to follow the Linux command line instructions. It might be possible to use e.g. XCode. This is untested (and no additional instructions exist).

B Using Premake

The exercises use *Premake* (<https://premake.github.io/>) to define the various projects and sub-projects. Premake can generate project files for a number of different build tools and IDEs. For convenience, both generated Visual Studio 2022 project files and Linux Makefiles are already included with the exercise code.

You can re-generate project files using the included Premake executables (no installation is required). For example, if you needed VS 2019 files, you would run

```
.\premake5.exe vs2019
```

from the Windows command line (assuming you are in the directory where you unpacked the exercise code). For Linux, you'd run

```
./premake5 gmake2
```

which would regenerate the included Makefiles. See [Using Premake](#) for a full list of supported project file types (or run Premake with the `--help` flag). If you need Premake for a different platform, you can download pre-built binaries at <https://premake.github.io/download/>.

For Apple, use the same command line as for Linux to generate Makefiles. Instead of the standard `premake5` executable, use the `premake5.apple` executable. This is a x86_64 executable, so M1 and M2 Macs might require a different one. Check the [premake homepage](#) to see if there are official M1/M2 releases.



Premake uses the Visual Studio terminology. If you open up the `premake5.lua` in the project root directory, it'll start with `workspace`. This corresponds to a Visual Studio solution. In each workspace/solution, we can define multiple projects. Scroll down to the end of the file, and find `project "exercise1"`. This defines the main project for this exercise – as you see, it includes the source files from the `exercise1/` directory.

The third party dependencies are separate projects defined in `third_party/premake5.lua`. The file is loaded via the `include "third_party"` statement in the top-level `premake5.lua`.

The full Premake documentation is available at <https://premake.github.io/docs/>.