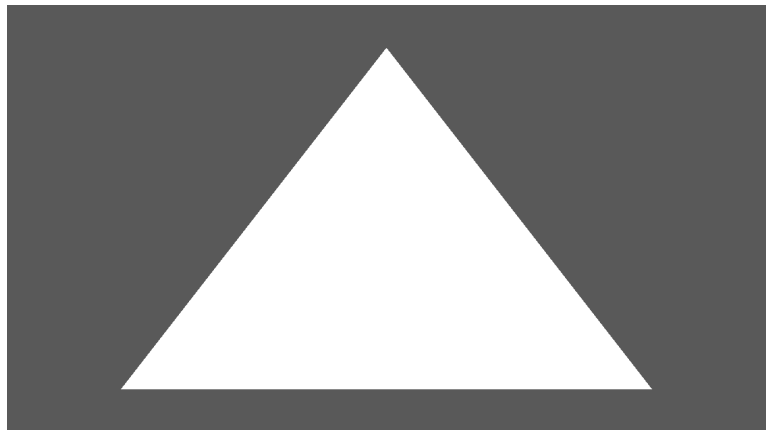


COMP5822M – Exercise 1.2

Vulkan Triangle

Contents

1	Project Updates	2
2	Labutils	2
2.1	Vulkan Context	2
2.2	Exception type: <code>Error</code>	3
2.3	Object wrappers	3
2.4	Vulkan utilities	4
3	Setup	4
3.1	Render Pass	4
3.2	Graphics Pipeline	7
3.3	Resources	13
3.4	Command objects	17
4	Command execution	19
4.1	Recording	19
4.2	Submission	21
5	Output	22
A	GLSL compiler: <code>glslc</code>	24



In this exercise, you will set up a minimal graphics pipeline and use it to render a very simple scene - a single triangle. The resulting program runs windowless and could thus easily be used on a machine without a display (e.g., headless servers). Unlike OpenGL, Vulkan explicitly supports this kind of operation.

Aside from setting up a simple graphics pipeline (`VkPipeline` & `VkPipelineLayout`), you will need set up a number of objects:

- a `VkRenderPass` with a single subpass that outputs into a single color attachment
- a `VkImage` and the associated `VkImageView` that serves as the color attachment into which we render
- a `VkFramebuffer` that references the color attachment image
- a `VkBuffer` for *downloading* the rendered image contents
- a `VkCommandPool` such that we can allocate a `VkCommandBuffer` to record Vulkan commands into
- a `VkFence` which is used to have the host code (our C++ program) wait until rendering has finished

To set up the graphics pipeline, you will need to provide the vertex and fragment shader for the pipeline. Exercise 2 therefore takes a brief look at shaders, compiling shaders to SPIR-V code, and at loading SPIR-V shader code at runtime.

With all objects ready, you will record the necessary commands into a command buffer and schedule it for execution. In order to be able to inspect the results, Exercise 2 will write the results to a PNG image on disk.

The exercise will introduce a few helper classes. These are intentionally kept quite simple, with the goal to remain as close to the “standard” Vulkan API as possible. Nevertheless, the helper classes reduce code complexity quite significantly, particularly in regard to cleanup and error handling.

We want to avoid to build generic abstractions around Vulkan anyway, as this tends to be less useful (but still a lot of work). Instead, when building abstractions, focus on your use case and your data. Build more advanced abstractions around these instead.



1 Project Updates

Download and unpack the exercise sources. Inspect the project. Exercise 2 will require you to work with the code in `exercise2` and with the utility functions in `labutils`. In addition to the C++ sources, the `exercise2` directory contains a `shaders` subdirectory. Shaders (extensions `.vert` and `.frag`) in that directory will be automatically compiled to SPIR-V code when the program is built (see the `exercise2-shaders` subproject defined in the top-level `premake5.lua`).

Compiled SPIR-V code is placed in the `assets/exercise2/shaders` directory, with an added `.spv` extension. For example, when the GLSL source `exercise2/shaders/triangle.vert` is compiled, it produces the SPIR-V output `assets/exercise2/shaders/triangle.vert.spv`.

Exercise 2 introduces a few new third party components:

shaderc The exercises include a copy of the `glslc` compiler for some platforms. The `glslc` compiler is shipped with the Vulkan SDK, so you can find it there for other platforms. See Appendix A for more information.

stb Copies of select single header libraries by [Sean Barret](#). In particular, Exercise 2 includes `stb_image.h` and `stb_image_write.h`. We use the latter to write the output image. The former is for loading images and will be used in later exercises.

You will also notice a new `util` containing a `glslc.lua` file. The Premake instructions to compile GLSL shaders with the included `glslc` compiler are listed in this file. *If you are on an unsupported, you will need to update this file with a few lines for your platform – see Appendix A. Further, the setup has only been tested with the `gmake2` and `vs20xx` Premake actions, and as such, there is no guarantee that they will work with other build systems!*

Try building the project. The provided shaders (`triangle.{vert, frag}`) do not do anything at the moment, but compiling them will still create the corresponding SPIR-V files (`triangle.{vert, frag}.spv`) in `assets/exercise2/shaders`. Verify that this is indeed the case.

2 Labutils

Before we dive into the Vulkan code, we will briefly inspect a few utilities defined in the `labutils` subproject. The last exercise concluded by indicating that we would want to improve (and automate) Vulkan object cleanup and error handling.

Inspect the `labutils` subproject/directory. You will find the familiar `to_string.{hpp, cpp}` pair of sources. But there are also a few new files:

- `error.{hpp, cpp}`
- `vkutil.{hpp, cpp}`
- `vkobject.{hpp, cpp, inl}`
- `vulkan_context.{hpp, cpp}`
- `context_helpers.{hxx, cpp}`

2.1 Vulkan Context

Study the files `vulkan_context.{hpp, cpp}` and `context_helpers.{hxx, cpp}`. The first header declares the `VulkanContext` class, which holds the Vulkan objects that were introduced in Exercise 1:

- a Vulkan instance (`VkInstance`)
- the selected physical device (`VkPhysicalDevice`)
- a logical device instance (`VkDevice`)
- a graphics queue (`VkQueue`) and the index of the queue family from which the queue was created
- a debug messenger handle (`VkDebugUtilsMessengerEXT`)

Review Exercise 1 if you need a refresher on these objects. A `VulkanContext` object *owns* the Vulkan objects and is therefore responsible for destroying these when the `VulkanContext` goes out of scope (see destructor of the `VulkanContext`).

The `VulkanContext` class is made non-copyable, but is *movable*. Hence, it is not possible to create copies of a `VulkanContext` object, but one can transfer ownership of its contents to a different instance. This is implemented by deleting the copy constructor and the copy assignment operator, and defining a [move constructor](#) and a [move assignment operator](#).

The reasoning behind making the `VulkanContext` move only is the fact that the object has ownership of the resources contained within it. When the object goes out of scope, it is responsible for destroying the resources it is responsible for. We cannot create copies of the Vulkan resources, hence it is not useful to allow the `VulkanContext` to be copied. (We could create a copy of the Vulkan handles that refer to the Vulkan objects, but then we would no longer know which `VulkanContext` instance owns the resources and thereby is responsible for destroying them.)



The C++ standard library defines a few *move only* types. See [std::unique_ptr](#) for an example and some additional information.

The function `make_vulkan_context()` performs the necessary setup as described in Exercise 1, and returns a `VulkanContext` instance with the created objects. Looking at the implementation in `vulkan_context.cpp`, you will find that much of the code matches the code from Exercise 1. The main difference is that errors are signalled by *throwing* an exception of type `labutils::Error` instead of a chain of return values.

2.2 Exception type: Error

The exception type `labutils::Error` is defined by `error.{hpp,cpp}`. It inherits from the standard exception type [std::exception](#).

With `labutils::Error` we can construct an error message with `std::printf()`-like formatting when throwing it. This is slightly less convenient with the standard C++ exception types; `labutils::Error` exists entirely to make producing useful error messages slightly more convenient.

Generating an exception thus looks like (for example)

```
throw lut::Error( "Unable to create Vulkan instance\n"
    "vkCreateInstance() returned %s", lut::to_string(res).c_str()
);
```

Exercise 2 catches exceptions with a [function try block](#) in `main()`. The catch statement simply prints the error message from the exception and causes the program to exit with an error code.

If an exception is thrown, it will propagate to `main()`, where it is caught. While propagating up the call stack, any objects with [automatic storage](#) will be destroyed (i.e., their destructors are called). By wrapping Vulkan handles into a class with a destructor, we thus ensure that the corresponding Vulkan objects are always destroyed appropriately.

2.3 Object wrappers

The `VulkanContext` object wraps the Vulkan objects that we have familiarized ourselves with in Exercise 1 in a C++ class, and thereby makes sure that these Vulkan objects are cleaned up properly. However, in Exercise 2 (and in the following exercises), additional Vulkan objects are introduced. We therefore need a strategy to deal with these.

We could try a similar approach as with the `VulkanContext`: identify reasonable groups of Vulkan objects and wrap these into a C++ class. However, we are not yet familiar with these objects, so we are not quite ready to do so (and in fact, identifying such groupings for general use is quite difficult).

Alternatively, we could create an individual C++ class for each Vulkan object. Such a class would just hold a single Vulkan object handle referring to the Vulkan object that the class owns. (Practically, we also need a handle to the parent Vulkan object to destroy the owned object.) This is the approach taken by some of the automatically generated C++ Vulkan API wrappers. However, for the exercises, we wish to stay with the default C API.

Defining a C++ class for each Vulkan object type that we encounter would quickly become somewhat tedious. Fortunately, C++ offers us a “simple” way out: C++ templates.

In `vkobject.hpp`, you will find the `labutils::UniqueHandle<>` template. If you are unfamiliar with templates, think of them as recipes to create ordinary C++ classes. The `labutils::UniqueHandle<>` template takes three parameters: the Vulkan handle type of the object we wish to wrap, the type of the parent Vulkan object, and a reference to the Vulkan function used to destroy the owned object (technically, a reference to a function pointer, since that is what the Volk library defines).

To create a class type that wraps a Vulkan render pass (`VkRenderPass`), we can use

```
using RenderPass = UniqueHandle< VkRenderPass, VkDevice, vkDestroyRenderPass >;
```

The class type `RenderPass` will hold a `VkRenderPass` object. Render pass objects belong to a logical device, and therefore we specify `VkDevice` as the parent object type. Finally, a `VkRenderPass` is destroyed with the `vkDestroyRenderPass` function, so we specify this as the final template argument. Looking at the definition of the destructor of `UniqueHandle`

```
template< typename tHandle, typename tParent, DestroyFn<tParent,tHandle>& tDestroyFn >
inline
UniqueHandle<tHandle,tParent,tDestroyFn>::~UniqueHandle()
{
    if( VK_NULL_HANDLE != handle )
    {
        assert( VK_NULL_HANDLE != mParent );
        tDestroyFn( mParent, handle, nullptr );
    }
}
```

we can manually do the substitution of the template parameters. We would end up with something equivalent to the following

```
RenderPass::~RenderPass()
{
    if( VK_NULL_HANDLE != handle )
    {
        assert( VK_NULL_HANDLE != mParent );
        vkDestroyRenderPass( mParent /* a VkDevice */, handle /* a VkRenderPass */, ▽
        ▷ nullptr );
    }
}
```

Similar to `VulkanContext`, the classes created from the `UniqueHandle<>` template are move only. In the header `vkobject.hpp`, you can additionally find the declaration for an equivalent `RenderPass` C++ class if we were to define it without the template.

Exercise 2 will use these light-weight C++ wrappers around most Vulkan objects.

2.4 Vulkan utilities

In Exercise 2 we will write a few functions that will be reused in future exercises. Such functions are to be declared in `vkutil.hpp` and defined in `vkutil.cpp`. However, initially, these files are relatively empty, so there is not much to see in them just yet.

3 Setup

The first step is to set up the Vulkan objects and resources necessary for rendering. The bulk of our work in terms of code will end up here. Setup is split into a few steps. We first create the render pass object. The render pass object is necessary to create the graphics pipeline, which also requires the pipeline layout to be defined. We then create the necessary resources: the image and its image view to which we render, and a buffer which we use to transfer the rendered image to the CPU. With the image view and the render pass objects, we can create the framebuffer. The final step is to prepare for command execute. This requires a command buffer, which is allocated from a command pool. We must wait for rendering to finish before accessing the resulting image data. This is done with a fence, which provides the facilities necessary to have the C++ host program wait for GPU processing to finish.

3.1 Render Pass

([Render Pass Attachments](#) — [Subpass Definition](#) — [Subpass Dependencies](#) — [Render Pass Creation](#))

All rendering in Vulkan takes place in a render pass. The render pass declares the output from the rendering in terms of individual attachments and their formats. A render pass is split into one or more subpasses, each potentially writing data to a subset of the attachments associated with the render pass. The render pass further defines a load and a store operation for each attachment. The load operation specifies via the [VkAttachmentLoadOp](#) enumeration how the contents of an attachment are treated when the render pass

starts. The store operation specifies via the [VkAttachmentStoreOp](#) how contents of an attachment are treated at the end of the render pass. With multiple subpasses, some amount of data can be passed between the subpasses via input attachments, and subsequently, it is necessary to declare dependencies between subpasses upfront. The same mechanism, i.e., subpass dependencies, may be used to synchronize rendering with external operations that happen before and after the render pass.

Even with [dynamic rendering](#) (introduced into core Vulkan with version 1.3), rendering still takes place in a “render pass”. The dynamic rendering functionality simply enables bypassing the creation of the `VkRenderPass` and `VkFramebuffer` objects when the full flexibility of these is not required (e.g., single subpass only). Instead, the corresponding information is specified when the “render pass” is started – this is done with [vkCmdBeginRendering](#) instead of the traditional `vkCmdBeginRenderPass`. Many of the same parameters still need to be specified, though.



In Exercise 2, the render pass is fairly simple. We have a single attachment: the color image to which we render our results. All rendering takes place in a single subpass, which further simplifies the setup.

Since this is the only rendering that will take place in Exercise 2, we must ensure that the target image is initialized properly. In this case, we want to clear the image to a specific background color. We can do so by specifying the `VK_ATTACHMENT_LOAD_OP_CLEAR` load operation. Furthermore, we do want to store the rendered results in the target image, and therefore specify the `VK_ATTACHMENT_STORE_OP_STORE` store operation – this ensures that the rendered color values are stored into the target color attachment at the end of the render pass.

Vulkan image resources have a [layout](#). Inside of a render pass, the image layout can change multiple times, as the image is potentially used in different ways. Therefore the render pass requires declaring multiple image layouts for each attachment that the render pass uses:

- an *initial layout*, i.e., the layout in which the image of a certain attachment is expected to be in when the render pass starts
- a layout for each subpass. Vulkan will automatically transition the image to the specified layout when the corresponding subpass starts.
- a *final layout*. Vulkan will transition the image to the specified layout when the render pass ends.

When we first create an image, it is in the `VK_IMAGE_LAYOUT_UNDEFINED` layout. We do not do anything with the image before rendering, so the image will be in this layout when the render pass starts. Hence we specify it as the initial layout for the render pass. When rendering to a color attachment, the image is required to be in the `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` layout. We specify this as the attachment’s layout in our only subpass. In order to transfer (copy) data out of the image to a host visible buffer, the image must be in the `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` layout. We specify this as the final layout of the render pass, such that our color image is automatically transitioned to this layout when the render pass ends.

Vulkan has very little implicit synchronization. In fact, we need to instruct Vulkan that the copy operation transferring the rendered image’s contents to the host visible buffer cannot commence until after rendering in the render pass has finished. This is done with a subpass dependency that declares the destination subpass to be external (`VK_SUBPASS_EXTERNAL`). The constant `VK_SUBPASS_EXTERNAL` refers to operations that happen outside to the render pass.

Render Pass Attachments The render pass object ([VkRenderPass](#)) is created in the `create_render_pass` function in `main.cpp`. First we must declare the properties of the attachments that the render pass uses. These are described with the [VkAttachmentDescription](#) structure. Each attachment used by the render pass requires a separate instance of this structure; however, in this case we require only one attachment, and therefore only a single copy of the structure:

```
// Note: the stencilLoadOp & stencilStoreOp members are left initialized to 0 (=DONT_CARE). The image
// format (R8G8B8A8_SRGB) of the color attachment does not have a stencil component, so these are ignored
// either way.
VkAttachmentDescription attachments[1]{};
attachments[0].format      = cfg::kImageFormat; // VK_FORMAT_R8G8B8A8_SRGB
attachments[0].samples     = VK_SAMPLE_COUNT_1_BIT; // no multisampling
attachments[0].loadOp      = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[0].storeOp     = VK_ATTACHMENT_STORE_OP_STORE;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[0].finalLayout  = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
```

Subpass Definition Next, we declare our single subpass using the [VkSubpassDescription](#) structure. This subpass uses the attachment above as a color attachment. This is expressed by referencing the attachment via a [VkAttachmentReference](#) structure; here we also specify that the attachment image should be transitioned to the `COLOR_ATTACHMENT_OPTIMAL` layout for this subpass.

```

VkAttachmentReference subpassAttachments[1]{};
subpassAttachments[0].attachment = 0; // the zero refers to attachments[0] declared earlier.
subpassAttachments[0].layout      = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkSubpassDescription subpasses[1]{};
subpasses[0].pipelineBindPoint    = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpasses[0].colorAttachmentCount = 1;
subpasses[0].pColorAttachments   = subpassAttachments;

// This subpass only uses a single color attachment, and does not use any other attachment types. We can
// therefore leave many of the members at zero/nullptr. If this subpass used a depth attachment (=depth buffer),
// we would specify this via the pDepthStencilAttachment member.
//
// See the documentation for VkSubpassDescription for other attachment types and the use/meaning of those.

```

Subpass Dependencies Finally, we use a [VkSubpassDependency](#) to introduce a dependency/barrier between the rendering in Subpass 0 and the image copy that takes place afterwards (`=VK_SUBPASS_EXTERNAL`). This ensures that all rendering completes before the copy takes place.

```

VkSubpassDependency deps[1]{};
deps[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
deps[0].srcSubpass      = 0; // == subpasses[0] declared above
deps[0].srcAccessMask   = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
deps[0].srcStageMask    = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
deps[0].dstSubpass      = VK_SUBPASS_EXTERNAL;
deps[0].dstAccessMask   = VK_ACCESS_TRANSFER_READ_BIT;
deps[0].dstStageMask    = VK_PIPELINE_STAGE_TRANSFER_BIT;

```

Normally, Vulkan does not introduce any implicit functionality without the programmer specifically asking for it. Subpass dependencies are one of the few exceptions to this. Vulkan will generate up to two implicit subpass dependencies. One from `VK_SUBPASS_EXTERNAL` (=commands before the renderpass) to the first subpass, and one from the last subpass to `VK_SUBPASS_EXTERNAL` (=commands after the renderpass). It will generate those *only* if no such subpass dependencies are explicitly defined and if an initial/final image layout transition is defined. The exact rules are documented in [Chapter 8.1 \(Render Pass Creation\)](#).

We have not defined a subpass dependency from `VK_SUBPASS_EXTERNAL` to Subpass 0 (the only and therefore first subpass). Here, Vulkan will insert an implicit subpass dependency. It is equivalent to the following:

```

// From
// https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#renderpass-implicit-dependencies
VkSubpassDependency implicitBefore{};
implicitBefore.srcSubpass      = VK_SUBPASS_EXTERNAL;
implicitBefore.dstSubpass      = 0; // The first and only subpass in this example
implicitBefore.srcStageMask    = VK_PIPELINE_STAGE_NONE;
implicitBefore.dstStageMask    = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
implicitBefore.srcAccessMask   = 0;
implicitBefore.dstAccessMask   = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT | ▽
    ▷ VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | ▽
    ▷ VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | ▽
    ▷ VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
implicitBefore.dependencyFlags = 0;

```

Our custom subpass dependency defined above is from the Subpass 0 (the only and therefore last subpass) to `VK_SUBPASS_EXTERNAL`. Hence, Vulkan will not generate a second implicit subpass dependency. If we had not defined the subpass dependency, the implicit dependency had looked like the following:

```

VkSubpassDependency implicitAfter{};
implicitAfter.srcSubpass      = 0; // The only and therefore last subpass
implicitAfter.dstSubpass      = VK_SUBPASS_EXTERNAL;
implicitAfter.srcStageMask    = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
implicitAfter.dstStageMask    = VK_PIPELINE_STAGE_NONE;

```



```
implicitAfter.srcAccessMask    = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | ▽
    ▷ VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
implicitAfter.dstAccessMask    = 0;
implicitAfter.dependencyFlags  = 0;
```

Note that the `.dstAccessMask` is set to zero; this subpass dependency would lack the necessary synchronization with the following image copy.

Render Pass Creation With the declarations in place, we can create our render pass. The create info structure [VkRenderPassCreateInfo](#) essentially only references the structures just defined:

```
VkRenderPassCreateInfo passInfo{};
passInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
passInfo.attachmentCount = 1;
passInfo.pAttachments = attachments;
passInfo.subpassCount = 1;
passInfo.pSubpasses = subpasses;
passInfo.dependencyCount = 1;
passInfo.pDependencies = deps;

VkRenderPass rpass = VK_NULL_HANDLE;
if( auto const res = vkCreateRenderPass( aContext.device, &passInfo, nullptr, &rpass▽
    ▷ ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create render pass\n"
        "vkCreateRenderPass() returned %s", lut::to_string(res).c_str()
    );
}

return lut::RenderPass( aContext.device, rpass );
```

We return the Vulkan `VkRenderPass` object wrapped into a `lut::RenderPass` object. This is one of the lightweight `UniqueHandle<>` wrappers discussed in Section 2.3. (If you have not done so already, also remove the line that unconditionally throws a `lut::Error` from the `create_render_pass` function.)

Vulkan has a policy to not create new words. If you look at the Vulkan documentation, it will always spell render pass with two words. The Vulkan API reflects this, and uses `RenderPass` (note the capital 'P' to indicate that it is two words). This is somewhat in contrast to e.g. "framebuffer", which is spelled as a single word (both in the specification and in code, as `Framebuffer`). The decision was that the term "framebuffer" was common enough already. You can observe the same pattern in other object names.



3.2 Graphics Pipeline

([Shader Code](#) — [Shader Loading](#) — [Pipeline Layout](#) — [Pipeline Creation](#))

The graphics pipeline describes how a set of primitives is to be drawn. It defines the inputs that the drawing consumes, the outputs that it produces, and a large number of options that affect the rendering process. Exercise 2 simplifies the graphics pipeline significantly by "hard coding" the geometry that it draws in the shaders. As such, the pipeline does not require any input (neither per-vertex nor uniform). We nevertheless have to create an (empty) pipeline layout object to declare the absence of any uniform input data.

Shaders Additionally, we need to load the SPIR-V shader code. The SPIR-V code is produced from the GLSL shader sources in `exercise2/shaders`, as described in Section 1. Right now, the shaders are not doing anything, and the first step is to implement these.

For now, we aim to draw a single triangle. The triangle is defined by three vertices. That is a sufficiently small amount of data such that we can easily "hard code" it in the vertex shader (`triangle.vert`):

```
const vec2 kVertexPositions[3] = vec2[3](
    vec2( 0.0f, -0.8f ),
    vec2( -0.7f, 0.8f ),
    vec2( +0.7f, 0.8f )
);
```

(add this in the global scope, just before `main()`).

Inside of `main()`, we will use the built-in value `gl_VertexIndex` to determine which of the three vertices the shader invocation is processing:

```
const vec2 xy = kVertexPositions[gl_VertexIndex];  
gl_Position = vec4( xy, 0.5f, 1.f );
```

Here, we just write position to the built-in `gl_Position`. This is a homogeneous clip-space coordinate. The Vulkan clip space extends -1 to $+1$ in the x and y directions, but only from 0 to $+1$ in z (after homogenization). (This is unlike OpenGL, where the clip space is the unit cube centered at $(0, 0, 0)$.)

The fragment shader (`triangle.frag`) is quite simple. It outputs a single color to the first color attachment. We must first declare this output (global scope, before `main()`):

```
layout( location = 0 ) out vec4 oColor;
```

The `layout(location = 0)` clause indices that this is the first (index 0) color attachment of the subpass. Specifically, the location index identifies a color attachment in the `pColorAttachments` array of the corresponding subpass' `VkSubpassDescription` structure (Section 3.1).

For now, we render a solid white triangle by outputting white for all fragments from `main()`:

```
oColor = vec4( 1.f, 1.f, 1.f, 1.f );
```

Compile the shaders. As we have previously observed, this generates matching SPIR-V code files in the `assets/exercise2/shaders` directory.

Shader loading In Exercise 2, we will load the SPIR-V files at runtime using the standard C input-output routines. The SPIR-V content is used to create a `VkShaderModule`, which is later used to define the vertex and fragment shader programs of the graphics pipeline.

Loading shader modules in this way will not change much in the future, so we will implement the function in the `labutils` project, where it can be reused by later exercises.

In `labutils/vkutil.hpp`, you can find the declaration for `load_shader_module`. The function needs to be implemented in `labutils/vkutil.cpp`. There are two parts to this: first, load the data from the file into memory, next create the `VkShaderModule` with

- `vkCreateShaderModule` and
- `VkShaderModuleCreateInfo`.

```
assert( aSpirvPath );  
  
if( std::FILE* fin = std::fopen( aSpirvPath, "rb" ) )  
{  
    std::fseek( fin, 0, SEEK_END );  
    auto const bytes = std::size_t( std::ftell( fin ) );  
    std::fseek( fin, 0, SEEK_SET );  
  
    // SPIR-V consists of a number of 32-bit = 4 byte words  
    assert( 0 == bytes % 4 );  
    auto const words = bytes / 4;  
  
    std::vector<std::uint32_t> code( words );  
  
    std::size_t offset = 0;  
    while( offset != words )  
    {  
        auto const read = std::fread( code.data()+offset, sizeof( std::uint32_t ), words -  
▷ -offset, fin );  
  
        if( 0 == read )  
        {  
            std::fclose( fin );  
  
            throw Error( "Error reading '%s': ferror = %d, feof = %d", aSpirvPath, std::  
▷ ::ferror(fin), std::feof(fin) );  
        }  
    }  
}
```



```

        offset += read;
    }

    std::fclose( fin );

    VkShaderModuleCreateInfo moduleInfo{};
    moduleInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    moduleInfo.codeSize = bytes;
    moduleInfo.pCode = code.data();

    VkShaderModule smod = VK_NULL_HANDLE;
    if( auto const res = vkCreateShaderModule( aContext.device, &moduleInfo, nullptr, &smod ); VK_SUCCESS != res )
    {
        throw Error( "Unable to create shader module from %s\n"
                    "vkCreateShaderModule() returned %s", aSpirvPath, to_string(res).c_str() );
    }

    return ShaderModule( aContext.device, smod );
}

throw Error( "Cannont open '%s' for reading", aSpirvPath );

```

The use of the C-style IO is a personal preference. If you feel more comfortable using the C++ iostream interface, you are welcome to use that instead. Either way, do not forget to handle errors from the IO routines!



Pipeline Layout As mentioned, our shaders currently do not have any uniform inputs. While we still need to create a pipeline layout object ([VkPipelineLayout](#)), it is fairly simple.

The lack of uniform inputs also means that we do not have to deal with descriptors, descriptor sets and descriptor set layouts just yet. They will appear in later exercises.



The pipeline layout is however specific to this exercise, so we return to `exercise2/main.cpp`. (Exercise 3 will use the same pipeline layout, but that is due to the fact that it performs the same rendering.)

Locate the `create_triangle_pipeline_layout` function. Creating a pipeline layout uses

- [vkCreatePipelineLayout](#) and
- [VkPipelineLayoutCreateInfo](#).

Take a look at the documentation – safe in the knowledge that you can ignore most of it (for now!).

```

VkPipelineLayoutCreateInfo layoutInfo{};
layoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
layoutInfo.setLayoutCount = 0;
layoutInfo.pSetLayouts = nullptr;
layoutInfo.pushConstantRangeCount = 0;
layoutInfo.pPushConstantRanges = nullptr;

VkPipelineLayout layout = VK_NULL_HANDLE;
if( auto const res = vkCreatePipelineLayout( aContext.device, &layoutInfo, nullptr, &layout ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create pipeline layout\n"
                    "vkCreatePipelineLayout() returned %s", lut::to_string(res).c_str() );
}

return lut::PipelineLayout( aContext.device, layout );

```

Graphics Pipeline With this, the components required to create a graphics pipeline are in place. We can therefore now focus on creating the graphics [VkPipeline](#) object.

Locate the function `create_triangle_pipeline` in `main.cpp`. The first step is to load the necessary shader modules. (If those are unavailable, we cannot create the pipeline anyway.)

```
// Load shader modules
// For this example, we only use the vertex and fragment shaders. Other shader stages (geometry, tessellation)
// aren't used here, and as such we omit them.
lut::ShaderModule vert = lut::load_shader_module( aContext, cfg::kVertShaderPath );
lut::ShaderModule frag = lut::load_shader_module( aContext, cfg::kFragShaderPath );
```

Thanks to using C++ exceptions, we were able to omit any explicit error handling code. If `load_shader_module` fails, it will throw an exception. We are not explicitly catching the exception, so the exception will propagate out of the `create_triangle_pipeline` function. It will continue to propagate through the call stack until a matching handler is found. In our case, this is first after leaving `main()`.

This will not leak any resources. For example, if the first call to `load_shader_module` succeeds, but the second one fails, the `lut::ShaderModule` returned by the first call will be destroyed appropriately as we leave the scope of `create_triangle_pipeline` by the way of exception. The destructor will execute, destroying the underlying Vulkan object appropriately.

To create a graphics `VkPipeline` object, we use

- [vkCreateGraphicsPipelines](#) and
- [VkGraphicsPipelineCreateInfo](#).

Note the plural: `vkCreateGraphicsPipelines` can indeed create multiple pipelines at once. To do so, we would pass multiple `VkGraphicsPipelineCreateInfo` structures to it.

Aside from communicating that the Vulkan creators potentially expect us to create many graphics pipeline objects, there are some specific reasons for this. Vulkan has the concept of [derived pipelines](#), i.e. graphics pipelines that have much in common. The members `basePipelineHandle` and `basePipelineIndex` of `VkGraphicsPipelineCreateInfo` relate to this. Creating a derivative pipeline may be cheaper than creating an independent pipeline (hypothetically, some resources such as the compiled shaders could be reused).

Exercise 2 uses a single graphics pipeline, so a filling in a single `VkGraphicsPipelineCreateInfo` is sufficient. Take a look at the documentation. As you can see, there are quite a few members that are structures in their own right that we need to fill in. The following covers the nested structure members for Exercise 2 in the order they appear in the documentation/`VkGraphicsPipelineCreateInfo` declaration.

Shader Stages The `pStages` member points to an array of `stageCount` [VkPipelineShaderStageCreateInfo](#) structures. In Exercise 2, there will be two, one for the vertex shader, and one for the fragment shader. The `VkPipelineShaderStageCreateInfo` structure is relatively uncomplicated. We specify the Vulkan handle to a `VkShaderModule` and specify the shader stage it corresponds to (specifically `VK_SHADER_STAGE_VERTEX_BIT` for the vertex shader and `VK_SHADER_STAGE_FRAGMENT_BIT` for the fragment shader). We also specify the name of each shader's entry point. In Exercise 2 this will be `main`, referring to the `main()` function in each shader.

In theory, you can use a different name. This used to be fairly buggy in the past (and originally neither of the `glslc` and `glslValidator` compilers supported it), but this might have been fixed in the meantime. If you want to use this feature, you will need to change command line used for `glslc`, where you define your custom entry point via the `--source-entrypoint` command line parameter.

For Exercise 2, we can leave the other parameters at zero. The `pSpecializationInfo` member is a bit interesting, as it allows you to specialize your shaders via [specialization constants](#). This can be useful if you want to have shaders optimized for e.g. specific numbers of resources without compiling multiple copies of the shader upfront.

The code ends up as follows:

```
// Define shader stages in the pipeline
VkPipelineShaderStageCreateInfo stages[2]{};
stages[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
stages[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
stages[0].module = vert.handle;
stages[0].pName = "main";

stages[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
stages[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
```

```

    stages[1].module = frag.handle; 10
    stages[1].pName   = "main";      11

```

Vertex Input State The `pVertexInputState` member describes the input vertex format. In it, we specify what buffers vertices are sourced from, and what vertex attributes in our shaders these correspond to. We will return to these in Exercise 4; Exercise 2 does not have any input attributes, making the code simple:

```

// For now, we don't have any vertex input attributes - the geometry is generated/defined in the vertex shader. 1
VkPipelineVertexInputStateCreateInfo inputInfo{}; 2
inputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO; 3

```

Input Assembly State With the `pInputAssemblyState` member, we define what type of primitive we are drawing. The supported primitives are enumerated by the [VkPrimitiveTopology](#) enum. The most important ones are `POINT_LIST`, `LINE_LIST` and `TRIANGLE_LIST`, for points, lines and triangles, respectively (the “list” suffix indicates that we are expected to draw multiple primitives at once). Additionally, for lines and triangles, there are the `_STRIP` versions, which can be used to draw connected piecewise linear curves (`LINE_STRIP`) and [triangle strips](#). Both are more efficient than just a plain line/triangle soup (though, indexed meshes have similar benefits and are more general).

Exercise 2 draws a single triangle to begin with, so we chose the simplest triangle-based primitive topology (for a single triangle, we would not see any benefits from strips/fans):

```

// Define which primitive (point, line, triangle, ...) the input is assembled into for rasterization. 1
VkPipelineInputAssemblyStateCreateInfo assemblyInfo{}; 2
assemblyInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO; 3
assemblyInfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST; 4
assemblyInfo.primitiveRestartEnable = VK_FALSE; 5

```

Tessellation State Exercise 2 does not use the tessellation stage, so the `pTessellationState` member can be left at `nullptr`.

Viewport State The viewport state (`pViewportState`) determines which parts of the framebuffer we draw into (and potentially allows us restrict the depth range that we use). The viewport determines how the normalized device coordinates are mapped to pixel coordinates. The scissor area can be used to restrict drawing to a part of the framebuffer without changing the coordinate systems. (So, the former will “rescale” the drawing, whereas the latter cuts off parts.) In Exercise 2, we just draw to the whole framebuffer:

```

// Define viewport and scissor regions 1
VkViewport viewport{}; 2
viewport.x = 0.f; 3
viewport.y = 0.f; 4
viewport.width = float(cfg::kImageWidth); 5
viewport.height = float(cfg::kImageHeight); 6
viewport.minDepth = 0.f; 7
viewport.maxDepth = 1.f; 8
9
VkRect2D scissor{}; 10
scissor.offset = VkOffset2D{ 0, 0 }; 11
scissor.extent = VkExtent2D{ cfg::kImageWidth, cfg::kImageHeight }; 12
13
VkPipelineViewportStateCreateInfo viewportInfo{}; 14
viewportInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO; 15
viewportInfo.viewportCount = 1; 16
viewportInfo.pViewports = &viewport; 17
viewportInfo.scissorCount = 1; 18
viewportInfo.pScissors = &scissor; 19

```

Rasterization State The `pRasterizationState` member controls aspects related to rasterization (and are mostly relevant when drawing triangles). Most of the options exist in OpenGL, and may therefore already be familiar:

- `polygonMode` ([VkPolygonMode](#)) determines if the selected primitive should be filled, the vertices should be connected by lines, or the vertices should be drawn as points. This could be used to produce wire-frame drawings (e.g. for debugging).
- `cullMode` ([VkCullModeFlags](#)) controls face culling. Like OpenGL, we generally want to cull back-facing triangles for efficiency.
- `frontFace` ([VkFrontFace](#)) determines which side of the triangle is considered to be front facing. We stick to OpenGL’s default of considering counter-clockwise triangle winding to be front facing.

- `depthClampEnable` and `depthBiasEnable` relate to depth buffer operation, and the latter is important when rendering shadow maps. Having no depth buffer, we can ignore these and just set them to `VK_FALSE`.
- `rasterizerDiscardEnable` instructs Vulkan to discard primitives before the rasterization stage. This effectively means that only the vertex processing stage (i.e., the vertex shader) will be used. We do want our triangle to be rasterized, so we set this to `VK_FALSE`. (Rasterizer discard is less useful these days, where compute shaders are available. In OpenGL, rasterizer discard could be used together with [transform feedback](#) to just transform vertices using a vertex shader/geometry shader, but without actually drawing anything.)
- `lineWidth` determines the line width when drawing lines or when using `VK_POLYGON_MODE_LINE`.

For Exercise 2, we select the following:

```
// Define rasterization options
VkPipelineRasterizationStateCreateInfo rasterInfo{};
rasterInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterInfo.depthClampEnable = VK_FALSE;
rasterInfo.rasterizerDiscardEnable = VK_FALSE;
rasterInfo.polygonMode = VK_POLYGON_MODE_FILL;
rasterInfo.cullMode = VK_CULL_MODE_BACK_BIT;
rasterInfo.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterInfo.depthBiasEnable = VK_FALSE;
rasterInfo.lineWidth = 1.f; // required.
```

Multisample State The `pMultisampleState` member controls hardware multisampling (MSAA, CSAA, etc). Multisampling allows us to increase the number of samples per pixel, frequently without increasing the number of fragment shader invocations. Enabling multisampling here also requires us to enable it in various related resources (image and renderpass). Finally, to get a “normal” image, we have to resolve the multiple samples per pixel to a single pixel color. For now, do not enable multisampling and just use a single sample per pixel:

```
// Define multisampling state
VkPipelineMultisampleStateCreateInfo samplingInfo{};
samplingInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
samplingInfo.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
```

Depth/Stencil State Exercise 2 does not use depth or stencil buffers/tests, so the `pDepthStencilState` member is left at `nullptr`. Exercise 4 uses a depth buffer, and will examine it in detail there.

Color Blend State The `pColorBlendState` controls blending (alpha blending) and logical operations. Each color attachment has a separate blend state ([VkPipelineColorBlendAttachmentState](#)), so with multiple attachments, we would need to specify multiple blend states. In Exercise 2, blending is not used, so we disable it for our single color attachment (Exercise 4 will use blending).

The `colorWriteMask` member determines which [color channels will be written](#). To write all color channels (which is likely the most common option), we toggle the four bits for the R, G, B and A channels in the mask.

```
// Define blend state
// We define one blend state per color attachment - this example uses a single color attachment, so we only
// need one. Right now, we don't do any blending, so we can ignore most of the members.
VkPipelineColorBlendAttachmentState blendStates[1]{};
blendStates[0].blendEnable = VK_FALSE;
blendStates[0].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
    VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;

VkPipelineColorBlendStateCreateInfo blendInfo{};
blendInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
blendInfo.logicOpEnable = VK_FALSE;
blendInfo.attachmentCount = 1;
blendInfo.pAttachments = blendStates;
```

[Logical operations](#) apply a logical operation to the fragment’s color values and the framebuffer’s color values. Read the Vulkan documentation for details. [OpenGL](#) supports them as well. They are (no longer) used very frequently.



Dynamic States Exercise 2 does not use any dynamic state, so the `pDynamicState` member is left at `nullptr`. Exercise 3 will enable some dynamic states to allow us to deal more easily with changes in the framebuffer size (e.g. when the window gets resized). Exercise 2 draws to a fixed-size image/framebuffer.

The remaining members are relatively straight-forward. We specify the

- `VkPipelineLayout` (see [Pipeline Layout](#)) with the `layout` member.
- `VkRenderPass` (see [Section 3.1](#)) with the `renderPass` member. We additionally need to specify the render pass's subpass index in which the pipeline is used with the `subpass` member.

A few additional options can be selected with the `flags` member. Exercise 2 does not need any of these, but you might want to take a look at the available options defined in the [VkPipelineCreateFlags](#) enumeration.

Assembling the `VkGraphicsPipelineCreateInfo` structure thereby becomes:

```

// Create pipeline
VkGraphicsPipelineCreateInfo pipeInfo{};
pipeInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

pipeInfo.stageCount = 2; // vertex + fragment stages
pipeInfo.pStages = stages;

pipeInfo.pVertexInputState = &inputInfo;
pipeInfo.pInputAssemblyState = &assemblyInfo;
pipeInfo.pTessellationState = nullptr; // no tessellation
pipeInfo.pViewportState = &viewportInfo;
pipeInfo.pRasterizationState = &rasterInfo;
pipeInfo.pMultisampleState = &samplingInfo;
pipeInfo.pDepthStencilState = nullptr; // no depth or stencil buffers
pipeInfo.pColorBlendState = &blendInfo;
pipeInfo.pDynamicState = nullptr; // no dynamic states

pipeInfo.layout = aPipelineLayout;
pipeInfo.renderPass = aRenderPass;
pipeInfo.subpass = 0; // first subpass of aRenderPass

VkPipeline pipe = VK_NULL_HANDLE;
if( auto const res = vkCreateGraphicsPipelines( aContext.device, VK_NULL_HANDLE, 1, &pipeInfo, nullptr, &pipe ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create graphics pipeline\n"
        "vkCreateGraphicsPipelines() returned %s", lut::to_string(res).c_str()
    );
}

return lut::Pipeline( aContext.device, pipe );

```

As already mentioned, creating pipelines can be quite costly (especially if many pipelines are being created). Vulkan provides another mechanism to keep the costs down: [pipeline caches](#). With pipeline caches ([VkPipelineCache](#)), it is possible to cache pipeline data on disk. The data can be loaded in a later run of the application and reused, potentially reducing the time necessary to create pipeline objects.

Exercise 2 (and other exercises) do not use the pipeline cache (and therefore set the third argument of `vkCreateGraphicsPipelines` to `VK_NULL_HANDLE`). The low number of pipelines that we create is not sufficient to introduce noticeable loading times. Furthermore, a pipeline cache does not help as much if the shaders change frequently (e.g. during development).

3.3 Resources

([Image & Image View](#) — [Framebuffer](#) — [Buffer](#))

It is now time to create the resources necessary for our application: we need to create the image to which we render. With the image, we can create the associated framebuffer object. Finally, to store the rendered image (which may reside in VRAM only accessible from the device and has a unknown memory layout as per `VK_IMAGE_TILING_OPTIMAL`), we need to transfer the image into a buffer backed with memory from a memory type that has the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` property set. The copy operation from the image to the buffer will also normalize the memory layout.

Both images and buffers require at least two Vulkan objects: the image or buffer itself and the associated memory allocation. To deal with this, Exercise 2 uses separate classes that hold both of these objects. The classes are defined locally for Exercise 2. While we use them in Exercise 3 as well, Exercise 4 and forward will introduce slightly different implementations.

The classes are defined in `image.{hpp,cpp}` and `buffer.{hpp,cpp}`, respectively. Take note of the destructors, where not only the corresponding resource is destroyed, but the associated memory is deallocated as well with [vkFreeMemory](#).

The classes are defined “locally” for Exercise 2 only. While Exercise 3 will use them as well, they will change in Exercise 4 to use suballocations from a larger `VkDeviceMemory` allocation. In Exercise 4, the final version of the classes will move into `labutils`.



Image and Image View Aside from the `VkImage` image object and the `VkDeviceMemory` memory allocation, we also require an image view (`VkImageView`). Find the `create_framebuffer_image` function. It returns all three in a `std::tuple`: the first two are packed into the `Image` class, and the image view is returned separately.

When implementing the `create_framebuffer_image` function, we must be careful. Multiple objects are created, meaning that there are multiple places in which we can exit the function with an error. It is important that we ensure that the necessary resources are all cleaned up, regardless of the code path taken. To do so, create a `Image` object at the top of the function:

```
Image image( aContext.device );
```

1

As we construct the Vulkan objects, we assign them to the relevant members in the class. Regardless of how the function is left, the destructor of the `Image` instance will run, potentially cleaning up the parts that we constructed successfully.

Image object Next, we create the `VkImage` itself. By now, many of the members of the [VkImageCreateInfo](#) structure should make sense. We are recommended to use the optimal tiling (`VK_IMAGE_TILING_OPTIMAL`) whenever possible. The image will be used as a color attachment primarily, but we also copy data out of it, so we must declare this in the `usage` member by setting the corresponding options:

- `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`: image used as a color attachment for rendering
- `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`: image is used as a source for a copy/transfer operation

(See [VkImageUsageFlagBits](#) for a complete list of possible usage options.)

The image is used with a single queue only, so we can trivially select the `VK_SHARING_MODE_EXCLUSIVE` option. The initial layout must be either `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREDEFINED`. The latter is special and allows us to use memory that was initialized appropriately somewhere else. This is not the case here, so we select `UNDEFINED`:

```
VkImageCreateInfo imageInfo{};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType = VK_IMAGE_TYPE_2D;
imageInfo.format = cfg::kImageFormat;
imageInfo.extent = VkExtent3D{ cfg::kImageWidth, cfg::kImageHeight, 1 };
imageInfo.mipLevels = 1;
imageInfo.arrayLayers = 1;
imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
imageInfo.usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | ▽
    ▸ VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

if( auto const res = vkCreateImage( aContext.device, &imageInfo, nullptr, &image.▽
    ▸ image ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create image\n"
        "vkCreateImage() returned %s", lut::to_string(res).c_str()
    );
}
```


Image memory We need to allocate memory to hold the image data in the `VkImage`. This involves a few steps. We first query the newly created `VkImage` for its memory requirements ([↗VkMemoryRequirements](#)) with the [↗vkGetImageMemoryRequirements](#) functions. The memory requirements tell us the (minimum) amount of memory required by the image (size member) and it gives us list of compatible memory types (see Exercise 1) in the `memoryTypeBits` bitmask.

```
VkMemoryRequirements memoryRequirements{};           1
vkGetImageMemoryRequirements( aContext.device, image.image, &memoryRequirements );           2
```

To allocate memory, we need to decide which memory type we want to use (assuming the `memoryTypeBits` gives us a choice). In this case, we will want to put the image into the most efficient type of memory possible, i.e., into a memory type that has the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` property. We use an utility function, `find_memory_type`, to identify a compatible type of memory.

Locate the definition of the `find_memory_type` function (towards the bottom of `main.cpp`). To find the correct memory type index, we need to know what types of memory the device supports. For simplicity, we will simply query the information from the physical device (a better approach might be to cache this data somewhere). We then iterate over the different memory types, checking that the memory type is allowed by the `memoryTypeBits` and whether or not it has the desired memory properties set:

```
// Based on findMemoryType() from           1
// https://vulkan-tutorial.com/Vertex_buffers/Vertex_buffer_creation           2
VkPhysicalDeviceMemoryProperties props;           3
vkGetPhysicalDeviceMemoryProperties( aContext.physicalDevice, &props );           4

for( std::uint32_t i = 0; i < props.memoryTypeCount; ++i )           5
{           6
    auto const& type = props.memoryTypes[i];           7
           8
    if( aProps == (aProps & type.propertyFlags) && (aMemoryTypeBits & (1u<<i)) )           9
    {           10
        return i;           11
    }           12
}           13
           14
throw lut::Error( "Unable to find suitable memory type (allowed memory types = 0x%x,           15
    > required properties = %s)", aMemoryTypeBits, lut::memory_property_flags(aProps)           16
    > .c_str() );           17
```

With this in place, we can return to `create_framebuffer_image`, and allocate the memory for the image:

```
VkMemoryAllocateInfo allocInfo{};           1
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;           2
allocInfo.allocationSize = memoryRequirements.size;           3
allocInfo.memoryTypeIndex = find_memory_type( aContext, memoryRequirements.           4
    > memoryTypeBits, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT );           5

if( auto const res = vkAllocateMemory( aContext.device, &allocInfo, nullptr, &image.           6
    > memory ); VK_SUCCESS != res )           7
{           8
    throw lut::Error( "Unable to allocate memory for image\n"           9
        "vkAllocateMemory() returned %s", lut::to_string(res).c_str()           10
    );           11
}           12
```

At the moment, we are using a separate Vulkan memory allocation for each image that is created this way. The number of memory allocations may be quite limited, so it is recommended to perform a few large memory allocations upfront, and then distribute memory from those allocations to the individual resources. Exercise 4 will do so – Exercise 2 only requires two memory allocations, so we are very unlikely to be hitting any of the limits.

Once the memory is allocated, we need to tell the `VkImage` to use this memory. The [↗vkBindImageMemory](#) function effects this:

```
vkBindImageMemory( aContext.device, image.image, image.memory, 0 );           1
```

Image View Finally, we want to create an image view (`VkImageView`) for the image that we just created. Since the image is quite simple (i.e., no mipmap levels, no layers, ...), the image view simply refers to the whole image:

```

VkComponentMapping mapping;
mapping.r = VK_COMPONENT_SWIZZLE_IDENTITY; // VK_COMPONENT_SWIZZLE_R
mapping.g = VK_COMPONENT_SWIZZLE_IDENTITY; // VK_COMPONENT_SWIZZLE_G
mapping.b = VK_COMPONENT_SWIZZLE_IDENTITY; // VK_COMPONENT_SWIZZLE_B
mapping.a = VK_COMPONENT_SWIZZLE_IDENTITY; // VK_COMPONENT_SWIZZLE_A

VkImageSubresourceRange range;
range.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
range.baseMipLevel = 0;
range.levelCount = 1;
range.baseArrayLayer = 0;
range.layerCount = 1;

VkImageViewCreateInfo viewInfo{};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = image.image;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = cfg::kImageFormat;
viewInfo.components = mapping;
viewInfo.subresourceRange = range;

VkImageView view = VK_NULL_HANDLE;
if( auto const res = vkCreateImageView( aContext.device, &viewInfo, nullptr, &view ) )
    ▷ ; VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create image view\n"
        "vkCreateImageView() returned %s", lut::to_string(res).c_str()
    );
}

```

Finally, we return the tuple with all the created resource:

```

return { std::move(image), lut::ImageView( aContext.device, view ) };

```

(An explicit `std::move` is required here, as we are moving ownership of the `Image` resources into a temporary `Image` instance in the `std::tuple`, before returning the tuple.)

Framebuffer The framebuffer for Exercise 2 is relatively simple. It has a single attachment, identified by the image view created earlier. Additionally, creation of the framebuffer requires the render pass object that we created in Section 3.1. This guarantees that the framebuffer is compatible with the render pass (and if validation is enabled, the validation layers will ensure that the attachments we use to create the framebuffer are correct).

Locate the `create_framebuffer` method in `main.cpp` and implement it as follows:

```

VkImageView attachments[1] = {
    aTargetImageView
};

VkFramebufferCreateInfo fbInfo{};
fbInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
fbInfo.flags = 0; // normal framebuffer
fbInfo.renderPass = aRenderPass;
fbInfo.attachmentCount = 1;
fbInfo.pAttachments = attachments;
fbInfo.width = cfg::kImageWidth;
fbInfo.height = cfg::kImageHeight;
fbInfo.layers = 1;

VkFramebuffer fb = VK_NULL_HANDLE;
if( auto const res = vkCreateFramebuffer( aContext.device, &fbInfo, nullptr, &fb ) )
    ▷ VK_SUCCESS != res )
{

```

```

        throw lut::Error( "Unable to create framebuffer\n"
            "vkCreateFramebuffer() returned %s", lut::to_string(res).c_str()
        );
    }

    return lut::Framebuffer( aContext.device, fb );

```

Buffer Buffer creation is similar to image creation: we first create the `VkBuffer` object, then query it for compatible memory types, identify a memory type with the right properties, allocate the memory, and finally bind the memory to the buffer object.

As with the image, we must declare how the buffer will be used. In this case, the buffer is only used as a destination for a copy operation. Therefore we only set the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

Locate the `create_download_buffer` method in `main.cpp`. First create the buffer object:

```

Buffer buffer( aContext.device );

VkBufferCreateInfo bufferInfo{};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = cfg::kImageSize;
bufferInfo.usage = VK_BUFFER_USAGE_TRANSFER_DST_BIT;
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

if( auto const res = vkCreateBuffer( aContext.device, &bufferInfo, nullptr, &buffer.
    ▷ buffer ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create buffer\n"
        "vkCreateBuffer() returned %s", lut::to_string(res).c_str()
    );
}

```

The rest of the method is almost identical to the image creation. Implement it. The relevant functions are:

- [vkGetBufferMemoryRequirements](#)
- [vkBindBufferMemory](#)

To identify the memory type index, you can reuse the `find_memory_type` function that was defined earlier. Make sure that you select a memory type that has the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` property set – otherwise we will not be able to access the buffer's data from our C++ program.

3.4 Command objects

([Command Pool](#) — [Command Buffer](#) — [Fence](#))

The final few objects that we need to prepare are related to command execution. A [VkCommandPool](#) is required to be able to allocate a [VkCommandBuffer](#). Finally, we need a [VkFence](#) in order to be able to wait until all commands have finished executing – only then will the rendered results be guaranteed to be available.

Since there is not much specific to Exercise 2 relating to these objects, all of the code for creating them goes into the `vkutil.{hpp, cpp}` files in the `labutils` project.

Command Pool The command pool is created with

- [vkCreateCommandPool](#) and
- [VkCommandPoolCreateInfo](#).

There are a few flags that control command buffer behaviour. We do not need these just now, and will return to them in Exercise 3. We need to set the `queueFamilyIndex` of the `VkCommandPoolCreateInfo` structure to the correct value.

Locate the `create_command_pool` function in `vkutil.cpp` and implement it as follows:

```

VkCommandPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = aContext.graphicsFamilyIndex;
poolInfo.flags = 0;

VkCommandPool cpool = VK_NULL_HANDLE;

```

```

if( auto const res = vkCreateCommandPool( aContext.device, &poolInfo, nullptr,
    ▷ &cpool ); VK_SUCCESS != res )
{
    throw Error( "Unable to create command pool\n"
        "vkCreateCommandPool() returned %s", to_string(res).c_str()
    );
}

return CommandPool( aContext.device, cpool );

```

Command Buffer Next, locate the `alloc_command_buffer` function. We use [vkAllocateCommandBuffers](#) to allocate a command buffer from the provided command pool. Note the plural – we can use the function to allocate multiple command buffers at once. However, Exercise 2 uses just one command buffer:

```

VkCommandBufferAllocateInfo cbufInfo{};
cbufInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
cbufInfo.commandPool = aCmdPool;
cbufInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
cbufInfo.commandBufferCount = 1;

VkCommandBuffer cbuff = VK_NULL_HANDLE;
if( auto const res = vkAllocateCommandBuffers( aContext.device, &cbufInfo, &cbuff );
    ▷ VK_SUCCESS != res )
{
    throw Error( "Unable to allocate command buffer\n"
        "vkAllocateCommandBuffers() returned %s", to_string(res).c_str()
    );
}

return cbuff;

```

The `VkCommandBuffer` is returned as-is, and is not wrapped into one of the `labutils` wrappers. We are not required to free command buffers individually, as they will be freed automatically when the parent command pool is destroyed (see the *Description* section in the documentation for [vkDestroyCommandPool](#)).

Fence The final object we require is a `VkFence`. It is created with

- [vkCreateFence](#) and
- [VkFenceCreateInfo](#).

The `flags` member of `VkFenceCreateInfo` accepts a single flag: `VK_FENCE_CREATE_SIGNALED_BIT`. Normally, fences are created unsignaled; this flag instead instructs Vulkan that the fence should be created signaled. For Exercise 2, we want the fence to be unsignaled, so we leave the `flags` member a zero.

Implement the `create_fence` method in `vkutil.cpp`:

```

VkFenceCreateInfo fenceInfo{};
fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceInfo.flags = 0;

VkFence fence = VK_NULL_HANDLE;
if( auto const res = vkCreateFence( aContext.device, &fenceInfo, nullptr, &fence );
    ▷ VK_SUCCESS != res )
{
    throw Error( "Unable to create fence\n"
        "vkCreateFence() returned %s", to_string(res).c_str()
    );
}

return Fence( aContext.device, fence );

```

(We will revisit `create_fence` in Exercise 3, where we do want to create fences in the signaled state.)

4 Command execution

With the necessary objects in place, we turn towards executing commands on our Vulkan device. While Exercise 2 only renders a single frame, in an interactive application, this part of the code would potentially run many times. We therefore avoid creating any new resources in this “hot code” portion of the program. Vulkan helps us do so, in that it does not require any new objects to be created.

The process is split into two steps. First, commands are recorded into a command buffer. Second, the command buffer is submitted for execution.

From previous experience, a somewhat common error has been to just record the commands, but not submit the resulting command buffer for execution. If you ever encounter a situation where seemingly nothing happens, double check that you not only record the necessary rendering commands, but subsequently also submit them for execution.

The error is quite understandable, especially when frequently switching between the OpenGL and Vulkan APIs. In OpenGL, many of the very similarly named functions do automatically submit the corresponding commands for execution.

Somewhat amusingly, OpenGL originally had a concept very similar to command buffers: [display lists](#). However, they were deprecated with OpenGL 3.1, likely in favour of the then-modern approaches involving vertex buffer objects, vertex array objects and similar objects.



4.1 Recording

Command recording begins by calling `vkBeginCommandBuffer` on a `VkCommandBuffer` object. The command buffer object is put into the *recording* state. Commands are recorded into the buffer by calling the corresponding `vkCmd*` functions with the command buffer as the first argument. Once all necessary commands are recorded into the buffer, recording ends when `vkEndCommandBuffer` is called on the command buffer object. The command buffer then goes into the *executable* state, and may be submitted for execution. The command buffer life cycle with all the possible states and state transitions is described in the Vulkan specification in [Chapter 6](#).

Multiple command buffers can be recorded at “the same time”. Each is identified by its `VkCommandBuffer` handle, which specifies the command buffer into which a command is recorded.

Vulkan makes a distinction between primary and secondary command buffers. We will be using primary command buffers exclusively for the moment, and not discuss secondary command buffers in detail. Refer to the Vulkan documentation for more information.

There are nevertheless a few interesting options that we can specify when starting command buffer recording. Options are passed to [vkBeginCommandBuffer](#) through the [VkCommandBufferBeginInfo](#) structure. For now, the `flags` member is the most interesting one. Through it, one or more of the options listed in [VkCommandBufferUsageFlagBits](#) can be selected:

VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT The flag indicates that the command buffer will only be submitted once for execution. As an implication, the command buffer will move to an *invalid* state after executing, rather than returning to the *executable* state. This is useful if the command buffer is re-recorded each time it is submitted, and may improve performance when doing so.

VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT This flag enables the command buffer to be submitted multiple times concurrently. When submitting a command buffer, it moves from the *executable* state to the *pending* state. Normally, a *pending* command buffer may not be submitted again. With this flag, (re-)submitting the command buffer while it is in the *pending* state is permitted.

Since we are only using the command buffer once, we will specify the `ONE_TIME_SUBMIT` flag.

The commands that need to be recorded are (in sequence):

1. [vkCmdBeginRenderPass](#) begins the render pass (and implicitly, the first subpass of it). Here, we specify the `VkFramebuffer` object that we render to; this framebuffer object has to be compatible with the render pass in question. Furthermore, we define the *clear values* that are used for attachments with `VK_ATTACHMENT_LOAD_OP_CLEAR`. After starting the render pass, we can begin recording rendering-related commands.
2. [vkCmdBindPipeline](#) specifies which pipeline the following rendering commands use. With it, we select the graphics pipeline that we created earlier in the exercise.
3. [vkCmdDraw](#) submits vertices for drawing. They will be drawn according to the graphics pipeline that was previously bound. We specify the number of vertices that we wish to draw, the number of times we wish to draw them (=number of instances), and can optionally specify what index the first drawn

vertex/instance should correspond to. We want to draw a single triangle, i.e., three vertices in one instance. *We are required to bind a graphics pipeline before submitting any vertices for drawing. Unlike OpenGL, there is no “default” pipeline!*

4. [vkCmdEndRenderPass](#) ends the current render pass.
5. [vkCmdCopyImageToBuffer](#) issues a copy from the rendered image to the “download” buffer that we previously created. Once the copy completes, we can access the rendered image through the buffer.

Most of the `vkCmd*` functions do not return a `VkResult`. Refer to the Vulkan documentation for additional information on various arguments that the functions accept.

After all the necessary commands are recorded, `vkEndCommandBuffer` ends the command recording.

In `main.cpp`, find the function `record_commands`. Implement the command buffer recording in it:

```
// Begin recording commands 1
VkCommandBufferBeginInfo begInfo{}; 2
begInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO; 3
begInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT; 4
begInfo.pInheritanceInfo = nullptr; 5
6
if( auto const res = vkBeginCommandBuffer( aCmdBuff, &begInfo ); VK_SUCCESS != res ) 7
{ 8
    throw lut::Error( "Unable to begin recording command buffer\n" 9
        "vkBeginCommandBuffer() returned %s", lut::to_string(res).c_str() 10
    ); 11
} 12
13
// Begin render pass 14
VkClearValue clearValues[1]{}; 15
clearValues[0].color.float32[0] = 0.1f; // Clear to a dark gray background. 16
clearValues[0].color.float32[1] = 0.1f; // If we were debugging, this would potentially 17
clearValues[0].color.float32[2] = 0.1f; // help us see whether the render pass took 18
clearValues[0].color.float32[3] = 1.f; // place, even if nothing else was drawn. 19
20
VkRenderPassBeginInfo passInfo{}; 21
passInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO; 22
passInfo.renderPass = aRenderPass; 23
passInfo.framebuffer = aFramebuffer; 24
passInfo.renderArea.offset = VkOffset2D{ 0, 0 }; 25
passInfo.renderArea.extent = VkExtent2D{ cfg::kImageWidth, cfg::kImageHeight }; 26
passInfo.clearValueCount = 1; 27
passInfo.pClearValues = clearValues; 28
29
vkCmdBeginRenderPass( aCmdBuff, &passInfo, VK_SUBPASS_CONTENTS_INLINE ); 30
31
// Begin drawing with our graphics pipeline 32
vkCmdBindPipeline( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aGraphicsPipe ); 33
34
// Draw a triangle = three vertices 35
vkCmdDraw( aCmdBuff, 3, 1, 0, 0 ); 36
37
// End the render pass 38
vkCmdEndRenderPass( aCmdBuff ); 39
40
// Copy image to our download buffer 41
VkImageSubresourceLayers layers{}; 42
layers.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT; 43
layers.mipLevel = 0; 44
layers.baseArrayLayer = 0; 45
layers.layerCount = 1; 46
47
VkBufferImageCopy copy{}; 48
copy.bufferOffset = 0; // Note: these are initialized to zero already; 49
copy.bufferRowLength = 0; // they're listed here explicitly for purposes 50
copy.bufferImageHeight = 0; // of showing them. 51
copy.imageSubresource = layers; 52
```



```

copy.imageOffset      = VkOffset3D{ 0, 0, 0 }; // See comment above.           53
copy.imageExtent      = VkExtent3D{ cfg::kImageWidth, cfg::kImageHeight, 1 }; 54
                                                                55
vkCmdCopyImageToBuffer( aCmdBuff, aFbImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, ▽ 56
    ▷ aDownloadBuffer, 1, &copy );
                                                                57
// End command recording                                           58
if( auto const res = vkEndCommandBuffer( aCmdBuff ); VK_SUCCESS != res )      59
{
    throw lut::Error( "Unable to end recording command buffer\n"           60
        "vkEndCommandBuffer() returned %s", lut::to_string(res).c_str()    61
    );
                                                                62
}
                                                                63
                                                                64

```

4.2 Submission

The function `record_commands` leaves the command buffer in the *executable* state, meaning that it is ready for submission. Command buffers are submitted for execution with [vkQueueSubmit](#). The function accepts several arguments:

- The `VkQueue` object identifying the queue that the commands should be submitted to. Note that this queue has to stem from the queue family that was specified when we created the `VkCommandPool` that the to-be-submitted command buffers were allocated from.
- A number of [VkSubmitInfo](#) structures, specifying one or more command buffers to be submitted. The structure is inspected in detail below.
- A [VkFence](#) that will be signaled once *all* submitted command buffers have completed execution. The host program can wait for the fence to become signaled, for example. The fence is optional, and if no device-to-host signaling is required, the argument can be set to `VK_NULL_HANDLE`.

We specify the command buffers to be submitted with one or more `VkSubmitInfo` structures. Through the `VkSubmitInfo` structure we can specify several options:

- A set of “wait” [VkSemaphores](#) that must become signaled before the commands in the command buffers may execute.
- A set of command buffers containing commands to be executed. These will share the other options specified in the in this `VkSubmitInfo` structure. For example, if a command buffer should have different wait conditions, a separate `VkSubmitInfo` structure must be used.
- A set of “signal” `VkSemaphores`. These will be signaled when the commands have finished execution (technically, when the specified *stages* of the commands have finished executing).

Unlike the `VkFence`, which provides device-to-host synchronization, the `VkSemaphore` provides synchronization between different device commands. Exercise 2 only uses a fence; Exercise 3 and other following exercises will cover semaphores in more detail.

The semaphore behaviour described above refers to the original binary semaphores that were shipped with Vulkan 1.0. Vulkan 1.2 introduces a second kind of semaphore, the [timeline semaphore](#). The timeline semaphore uses a 64-bit value as its state. The timeline semaphore is a superset of both the original semaphore and fence, enabling both device-to-device and device-to-host synchronization. Additional, it introduces mechanisms to signal semaphores from the host, which was not possible previously.

One drawback of timeline semaphores is that do not yet interact well with Vulkan’s window presentation system, which is one of the early places where semaphores are used.



In Exercise 2, we only have a single command buffer and do not need to use the semaphore mechanism. This simplifies queue submission. Locate the function `submit_commands` in `main.cpp`, and implement it:

```

VkSubmitInfo subInfo{};
subInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
subInfo.commandBufferCount = 1;
subInfo.pCommandBuffers = &aCmdBuff;

if( auto const res = vkQueueSubmit( aContext.graphicsQueue, 1, &subInfo, aFence ); ▽ 6
    ▷ VK_SUCCESS != res )
{
    throw lut::Error( "Unable to submit command buffer to queue\n"
        "vkQueueSubmit() returned %s", lut::to_string(res).c_str()
    );
}

```

```
);
}
```

10
11

5 Output

(Wait/Fence — Map Memory — Write Image)

At this point, our program is successfully (hopefully) rendering a triangle. However, we would like be able to see the results. To do so, we will store the resulting image to disk.

This essentially involves three steps. First, we need to wait for rendering to finish. Although we have submitted the rendering commands for execution in the previous steps, they might not have executed at this point in our program.

While rendering a single triangle does not take much time, the commands that we have submitted must be transferred to the GPU, and the GPU must then schedule these for execution. In contrast, in the C++ program's execution, we are likely just a handful of instructions from the point of having called `VkQueueSubmit`, so very little time has passed in the host program's execution.



Once we have verified that the rendering commands have executed, we will want to access the image data that we have transferred into the buffer object that we allocated. We do this by *mapping* the memory of the buffer with `vkMapMemory`. Mapping memory requires that the memory was allocated from a memory type with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` property and gives us a normal C/C++ pointer to the memory's contents. With the pointer, we can access the image data in the buffer like in any other C/C++ memory object. Mappings created by `vkMapMemory` must be destroyed with `vkUnmapMemory` once they are no longer required.

Finally, we use `stbi_write_png()` from the `stb_image_write.h` library to store the image data on disk.

The `stbi_write_png` function just accepts a pointer to the image data. We might be tempted to pass the pointer that we received from `vkMapMemory` to it directly. This is valid, but a potential performance problem.

Normal memory, such as the memory that we receive through `::operator new` or `malloc` is *cached*. Repeated access to the same piece of memory are very fast. Many algorithms rely on this property, and this includes the image compression that takes place in the `stbi_write_png` function.

The memory in which the buffer object was allocated might not be cached, depending on the memory type that was used for the buffer. Memory types would indicate that they reside in "normal" cached memory through the `VK_MEMORY_PROPERTY_HOST_CACHED_BIT`. We did not require this bit to be set when selecting a memory type – not all GPUs/drivers expose memory with it set. It is therefore prudent to assume that the memory is uncached.



To avoid the potential performance hazard that uncached memory would introduce, we create an additional conventional C/C++ memory buffer, and copy data from the mapped Vulkan buffer into it. We then pass the conventional buffer to `stbi_write_png`. Copying out the uncached memory is not a problem, since each byte of memory is only touched once. (In fact, good implementations of `memcpy` will use the non-temporal/"streaming" memory copy operations that are available on e.g. x86_64 that bypass caches/use a dedicated caching mechanism, when possible.)

Wait (Fence) We implement these steps directly within `main()`. First, we call `vkWaitForFences` to wait for the fence to be signaled, indicating that the rendering commands have finished:

```
// Commands are executed asynchronously. Before we can access the result, we need to wait for the processing 1
// to finish. The fence that we passed to VkQueueSubmit() will become signalled when the command buffer has 2
// finished processing – we will wait for that to occur with vkWaitForFences(). 3
4
// Wait for commands to execute 5
constexpr std::uint64_t kMaxWait = std::numeric_limits<std::uint64_t>::max(); 6
7
if( auto const res = vkWaitForFences( context.device, 1, &fence.handle, VK_TRUE, ∇ 8
    ▷ kMaxWait ); VK_SUCCESS != res ) 9
{ 10
    throw lut::Error( "Waiting for fence\n" 11
        "vkWaitForFences() returned %s", lut::to_string(res).c_str() 12
    ); 13
}
```

As indicated by the use of plural in the function, `vkWaitForFences` can be used to wait on multiple fences. With the `waitAll` argument, we can indicate if the method should return when any fence is signaled, or only if all the fences become signaled. With a single fence, we can set this to either `VK_TRUE` or `VK_FALSE`, it does not matter much.

We can additionally specify a maximum timeout, after which the function will return regardless of the state of the fence(s). The value is specified in nanoseconds. We just set this to the maximum value, essentially indicating that the function should potentially wait “forever” (the timeout is specified in an unsigned 64-bit value, meaning that “forever” corresponds to just around 585 years ... but our rendering should have finished by then).

Map memory After waiting, we use [vkMapMemory](#) to map the buffer’s underlying memory into the C/C++ address space. This gives us a `void*` pointer. We copy the data from this pointer into a normal C/C++ buffer (created in a `std::vector` for convenience) with `std::memcpy`. With that done, we no longer require the Vulkan mapping and can destroy it with `vkUnmapMemory`:

```
// Access image and write it to disk.
// We copied the image to the buffer dlBuffer - we allocated the memory backing this buffer, dlMemory, from a
// memory type with the property VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT. This means that we can
// “map” the contents of this buffer such that it becomes accessible through a normal C/C++ pointer.
void* dataPtr = nullptr;
if( auto const res = vkMapMemory( context.device, dlBuffer.memory, 0, cfg::▽
    ▷ kImageSize, 0, &dataPtr ); VK_SUCCESS != res )
{
    throw lut::Error( "Mapping memory\n"
        "vkMapMemory() returned %s", lut::to_string(res).c_str()
    );
}

assert( dataPtr );

std::vector<std::byte> buffer( cfg::kImageSize );
std::memcpy( buffer.data(), dataPtr, cfg::kImageSize );

// Why the extra copy? dataPtr points into a special memory region. This memory region may be created
// uncached (e.g., reads bypass CPU caches). Streaming out of such memory is OK - so memcpy will touch
// each byte exactly once. Reading multiple times from the memory, which the compression method likely does,
// is significantly more expensive.
//
// In one test, passing dataPtr directly to stbi_write_png() takes about 4.5s, whereas using the extra buffer
// reduces this time to 0.5s.
//
// To avoid the extra copy, we could request memory with the VK_MEMORY_PROPERTY_HOST_CACHED
// property in addition. However, not all devices support this, and it may have other overheads (i.e., the
// device/driver likely needs to snoop on the CPU caches, similar to HOST_COHERENT).

vkUnmapMemory( context.device, dlBuffer.memory );
```

Write output image The last part simply calls `stbi_write_png` with the correct parameters. You can find the declaration (and definition) of the function, and some documentation, in the [stb_image_write.h header](#). To store a four-channel (RGBA) PNG image, you can use the following:

```
if( !stbi_write_png( cfg::kImageOutput, cfg::kImageWidth, cfg::kImageHeight, 4, ▽
    ▷ buffer.data(), cfg::kImageWidth*4 ) )
{
    throw lut::Error( "Unable to write image: stbi_write_png() returned error" );
}
```

Experiment with passing the pointer returned by `vkMapMemory` directly to `stbi_write_png` and compare the performance. If it is in uncached memory, the performance difference should be pretty noticeable, even without precise measurements. Not all systems may experience this, though. On a system where the Vulkan buffer ends up in host-cached memory, the extra copy into the `std::vector`-allocated buffer is unnecessary (and may be avoided for increased efficiency).



Wrapping up

Compile and execute the program. Make sure it produces the `output.png` image file. The image should look like the one shown on the first page of these instructions.

Note that we did not need to perform any manual object cleanup. The light-weight C++ classes introduced in this exercise automated the cleanup for us. The cleanup is guaranteed to take place, both when the program runs successfully and when it exits via an error.

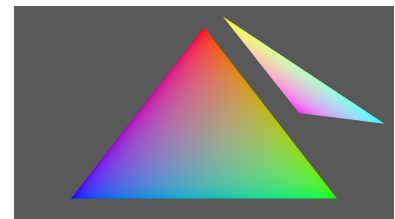
“Hard coding” the triangle may seem a bit artificial and useless. Defining geometry this way certainly does not scale to larger models. However, it does demonstrate an important technique: we *can* define the geometry fully from the vertex processing stage, with no per-vertex input attributes. Instead of hard-coding, we could use an algorithm to determine the output vertices – e.g., to procedurally generate geometry. That said, using the vertex shader to generate a single triangle is in fact useful. In post-processing, we often perform a *full screen pass*, where the goal is to invoke the fragment shader once for each pixel on the screen. As we shall see, using a single (procedurally defined) triangle is most convenient for this purpose.

The next exercise, Exercise 3, will render into a window surface. We will reuse much of the code from Exercise 2, which will allow us to focus on the aspects related to the window system integration and the surrounding windowing and event handling logic.

Optional experiments

There are quite a few optional experiments that you can conduct at this point. The following lists a few suggestions.

Add a second triangle We render a single triangle up until this point. Add a second one. Try to place it to the top-right of the existing one. Can you figure out the correct clip-space coordinates on the first attempt?



Colorize the triangle(s) In the vertex shader, define a per-vertex color in addition to the position. Pass this color from the vertex shader to the fragment shader, by defining an `out` “varying” in the vertex shader and a corresponding `in` “varying” in the fragment shader:






```
// Vertex shader:
// Set this in the vertex shader's main()
layout( location = 0 ) out vec3 v2fColor;

// Fragment shader:
// Read from this in the fragment shader's main()
layout( location = 0 ) in vec3 v2fColor;
```

Viewport, scissor and render area In the graphics pipeline, we define a view port and a scissor rectangle. When starting the render pass, we define a render area. Right now these are all set to the full image size. Play around with these (especially the former two) and determine how they affect the output.

[Advanced] Multisampling See if you can enable multisampled rendering. There are a few places where you have to enable multisampling (look for `SAMPLE_COUNT_1_BIT`). You will need an additional (non-multisampled) image, into which the MSAA surface is resolved, to produce the final “normal” image. Resolving the MSAA should be done as part of the renderpass/subpass. Look up “resolve attachments” in the subpass definition.

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 GLSL compiler: `glslc`

The `glslc` compiler is one of the options for compiling GLSL shader code. The compiler is part of Google’s [shaderc](https://github.com/google/shaderc) project. Aside from the compiler, the tool offers `libshaderc`, which can be used to compile GLSL shader code “online” (e.g., at runtime, similar to how GLSL shader code is consumed in OpenGL).

The exercises use the “offline” standalone `glslc` compiler tool to compile GLSL shader code to SPIR-V code at build-time. This setup makes it easy to utilize shared code through `#include` statements. While Exercise 2 does not utilize this yet, we will see this in action in later exercises.

The exercise distributes `glslc` binaries for x86_64 Windows, Linux and MacOS. You can find the compiler binaries in:

- Linux: `third_party/shaderc/linux-x86_64/glslc`
- Windows: `third_party/shaderc/win-x86_64/glslc.exe`
- MacOS: `third_party/shaderc/macos-x86_64/glslc`

Recent versions of the Vulkan SDK ship the compiler binary, so if you are on a different system or architecture, you can find a working binary there.

The Premake configuration includes setup to use the included `glslc` binaries to compile shader code. The main definitions are in `util/glslc.lua`.

If you are on a different platform, you will need to minimally update the platform-specific selection logic in the top of the `glslc.lua` file:

```
local host = os.host();
if "windows" == host then
    binname = "win-x86_64/glslc.exe";
elseif "linux" == host then
    binname = "linux-x86_64/glslc";
elseif "macosx" == host then
    binname = "macos-x86_64/glslc";
else
    error( "No glslc binary for this platform ( " .. host .. " )" );
end
```



Add a case for your OS, and point it at the correct `glslc` binary.

This currently only accounts for different OSes, and ignores the host's architecture (it assumes it will be x86_64). If you plan to use the same OS on multiple different architectures, you would need to add the corresponding logic.

Exercise 2 uses the following `glslc` command line to compile the shaders:

```
path-to-glslc/glslc -O -o path-to-output.spv path-to-input-shader
```

This expands to (for example):

```
../../../../third_party/shaderc/linux-x86_64/glslc -O -o ../../assets/exercise2/shaders/▽
▷ triangle.vert.spv triangle.vert
```