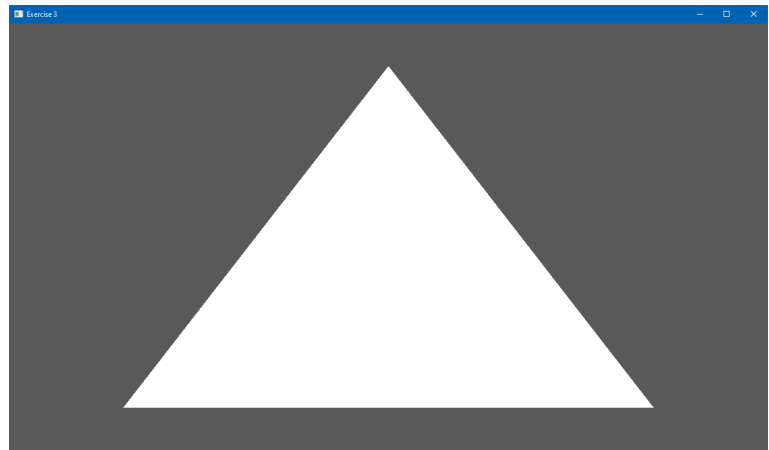


# COMP5822M – Exercise 1.3

## Vulkan Window

### Contents

1	Project Updates	1
2	GLFW Setup	2
3	Swapchain Setup	6
4	Handling User Input	13
5	Main Loop	13



Exercise 3 takes the renderer from Exercise 2 and updates it to render into a window. For this purpose, the exercise utilizes [GLFW](#), a cross-platform library that abstracts platform-specific window-system interaction into a platform independent interface.

GLFW implements the platform specific window creation. It uses the Vulkan Window System Integration (WSI) extensions to create a [VkSurfaceKHR](#). Using the `VkSurfaceKHR`, our application must identify a device that can render to the surface/window, create a swapchain with images, and set up the graphics pipeline to render to the swapchain images.

Much of the setup from Exercise 2 remains the same. One of the major changes relates to the target images: there are now have multiple different images that will be used in turn. Each frame, we are provided with one specific target image and expected to render to this. The image format is determined by the swapchain.

To provide a general and extensible solution, Exercise 3 will record rendering commands each frame (even though the commands do not change frame-to-frame). To avoid asynchronous execution hazards, this requires using multiple command buffers and some extra synchronization.

## 1 Project Updates

Grab the exercise code and inspect it. You will be working with the code in the `exercise3` directory and with some of the code in `labutils` project. Exercise 3 builds on Exercise 2. You will want to transfer some of your solutions from Exercise 2 to Exercise 3:

- Transfer your implementations from `labutils/vkutils.cpp` from Exercise 2. Do not overwrite the `vkutils.hpp`, as it contains a few new prototypes. Note that some of the function declarations have changed, so if you copy over your old `vkutils.cpp`, make sure to update the function definitions with the additional parameters. The additional parameters will be unused for the moment; the exercise has instructions on how they are used later.
- Transfer your shaders (`exercise2/shaders/triangle.{vert,frag}`) to Exercise 3.
- Look at the local functions defined in `exercise3/main.cpp`. Do not transfer any code yet – the exercise will instruct you when to do this.

Exercise 3 will change some of the functions defined in `vkutils.{hpp,cpp}`. The updates are such that they remain *source compatible* with Exercise 2. You can copy your solution for Exercise 2 into the `exercise2` subdirectory of the Exercise 3 project. It will be built alongside Exercise 3's main project in `exercise3`.



**Third Party Software** As indicated in the introductory text, Exercise 3 uses [GLFW](#). You can find it in the `third_party` directory, along with the third party software from previous exercises. Exercise 3 builds GLFW from source (and should be compatible with the three major platforms).

**Labutils** You will find a few new files in `labutils`:

**`vulkan_window.{hpp,cpp}`** Defines the `VulkanWindow` class that extends the `VulkanContext` with additions necessary to render to a GLFW window. Section 2 will introduce the `VulkanWindow` in detail.

The functions defined in `context_helpers.{hxx,cpp}` are intended to be used only to help implement the functionality in `vulkan_{context,window}.cpp` in `labutils`. As such, I consider these to be internal to the `labutils` project, and do not intend them to be used outside of it. I indicate this in two ways: all the functions defined in the internal headers are defined in a nested `detail` namespace, and I give headers that only define internal functions a `.hxx` extension instead of the more common `.hpp` one. The former (`detail` namespace) is fairly common practice, whereas the latter (`.hxx` extension) is more of a personal convention (although I have seen it used in other projects as well).



## 2 GLFW Setup

([GLFW Initialization](#) — [Vulkan Instance](#) — [GLFW Window](#) — [Device Selection](#) — [Vulkan Device](#))

Study the `vulkan_window.hpp` header. It declares the `VulkanWindow` class, which inherits from Exercise 2's `VulkanContext` class. `VulkanWindow` extends the `VulkanContext` with a few additional members:

- [GLFWwindow\\*](#): GLFW window handle.
- [VkSurfaceKHR](#): surface handle, that is a platform-independent abstraction from native platform window or surface objects. The term *surface* is frequently used in this context to refer to a block of pixels that we can draw to (such as the block of pixels that define a window's contents).
- A second queue - the *presentQueue* and its associated queue family index (`presentFamilyIndex`).
- [VkSwapchainKHR](#) and the associated `VkImages` and their image views (`VkImageView`).
- Information about the swapchain, specifically its format (`VkFormat`) and size in pixels (`VkExtent2D`)

The header further declares two functions. The first one, `make_vulkan_window`, is analogous to Exercise 2's `make_vulkan_context` function, and is used to create a default window and the associated Vulkan resources. The second one, `recreate_swapchain`, is used to re-create the swapchain. A swapchain may become outdated and no longer usable, e.g., when the underlying window changes size or shape. In this case, Vulkan requires the swapchain to be re-created, taking the changed properties into account.

Swapchain creation and re-creation is dealt with in Sections 3 and 5. First, we will focus on setting up a window with GLFW, and implement the necessary changes to instance and device creation.

**GLFW Initialization** The GLFW library requires special [initialization](#) to take place, performed with the [glfwInit](#) function. GLFW allows some options to be set ahead of initializations (via `glfwInitHint`), but Exercise 3 does not require this.

As we intend to use Vulkan with GLFW, we should check that GLFW supports Vulkan. GLFW tries to use the Vulkan loader independently from the Vulkan library (which GLFW does not know about), so the check is necessary to ensure that GLFW has been able to locate and use the system's Vulkan loader.

In `make_vulkan_window` (`labutils/vulkan_window.cpp`) locate the `//TODO: initialize GLFW` line and add the following code:

```
// Initialize GLFW and make sure this GLFW supports Vulkan.
// Note: this assumes that we will not create multiple windows that exist concurrently. If multiple windows are
// to be used, the glfwInit() and the glfwTerminate() (see destructor) calls should be moved elsewhere.
if( GLFW_TRUE != glfwInit() )
{
    char const* errMsg = nullptr;
}
```

```

    glfwGetError( &errMsg );
    throw lut::Error( "GLFW initialization failed: %s", errMsg );
}

if( !glfwVulkanSupported() )
{
    throw lut::Error( "GLFW: Vulkan not supported." );
}

```

This can take place before or after the initialization of Volk (via `volkInitialize`), whichever you prefer.

After calling `glfwInit`, we should call [glfwTerminate](#) when we are done with GLFW. Right now, the destructor of `VulkanWindow` does this when it destroys the GLFW window. Technically, this is not entirely correct, as there are some conditions in which GLFW does not get unloaded properly, and it also prevents us from using more than one `VulkanWindow` concurrently. None of the exercises will do so, and this keeps the code a bit simpler.



**Vulkan Instance Creation** Rendering to a window surface requires platform-specific instance extensions. These extensions link the platform-specific window handle types to a platform-independent `VkSurfaceKHR` handle. While GLFW deals with the exact setup that this requires, we need to make sure that the necessary extensions are enabled in the Vulkan instance. To facilitate this, GLFW provides a list of extensions it will require, via the [glfwGetRequiredInstanceExtensions](#) function.

The following code will check that the required extensions are available and request them to be enabled during instance creation.

```

// GLFW may require a number of instance extensions
// GLFW returns a bunch of pointers-to-strings; however, GLFW manages these internally, so we must not
// free them ourselves. GLFW guarantees that the strings remain valid until GLFW terminates.
std::uint32_t reqExtCount = 0;
char const** requiredExt = glfwGetRequiredInstanceExtensions( &reqExtCount );

for( std::uint32_t i = 0; i < reqExtCount; ++i )
{
    if( !supportedExtensions.count( requiredExt[i] ) )
    {
        throw lut::Error( "GLFW/Vulkan: required instance extension %s not supported",
        requiredExt[i] );
    }

    enabledExensions.emplace_back( requiredExt[i] );
}

```

You would expect to see one or more of the following extensions to be required, depending on the platform that you are running on:

- [VK\\_KHR\\_surface](#)
- [VK\\_KHR\\_xcb\\_surface](#) (for Linux/X11 with the XCB X11 interface)
- [VK\\_KHR\\_win32\\_surface](#) (for Windows with the standard Windows API)
- [VK\\_EXT\\_metal\\_surface](#) (for MacOS, for surfaces created for Apple's Metal framework)
- [VK\\_MVK\\_macos\\_surface](#) (older extension, now replaced by the above extension)



Aside from requesting the additional instance extensions, the Vulkan instance object, along with the optional debug messenger object, is created as before. With the instance, we can proceed with the window creation.

**GLFW Window Creation** GLFW windows are created with the [glfwCreateWindow](#) function. GLFW was originally designed with OpenGL in mind, so by default GLFW will perform all the necessary OpenGL setup, including the creation of an OpenGL rendering context. For Vulkan, no OpenGL context is desired. We indicate this to GLFW via the [GLFW\\_CLIENT\\_API](#) [window hint](#).

Technically, the window creation can take place before creating the Vulkan instance. However, I prefer to delay the window creation as long as possible. Creating a window is noticeable to a user, especially if the window is created in a visible state (which is the default for GLFW). Delaying the window creation reduces the number of code paths where we create a window and then immediately tear it down due to an error.



Once we have created the window, we need to get the `VkSurfaceKHR` handle corresponding to the window. GLFW handles this for us with the [glfwCreateWindowSurface](#) function. If you are interested in the underlying platform-specific mechanism, check the documentation of the platform extensions above. These mainly provide Vulkan functions to take the platform's native window handle (`HWND`, `xcb_window_t`, ...) and create a generic `VkSurfaceKHR` object from it.

Putting the above into code:

```
// Create GLFW Window and the Vulkan surface
glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );

ret.window = glfwCreateWindow( 1280, 720, "Exercise 3", nullptr, nullptr );
if( !ret.window )
{
    char const* errMsg = nullptr;
    glfwGetError( &errMsg );

    throw lut::Error( "Unable to create GLFW window\n"
        "Last error = %s", errMsg );
}

if( auto const res = glfwCreateWindowSurface( ret.instance, ret.window, nullptr,
    ▷ &ret.surface ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to create VkSurfaceKHR\n"
        "glfwCreateWindowSurface() returned %s", lut::to_string(res).c_str() );
}
```

**Vulkan Device Selection** The underlying logic for selecting devices remains unchanged from Exercise 1. We will however need to take a few additional factors into account. The device must support the necessary device extensions, specifically `VK_KHR_swapchain`. Further, it must support rendering to the provided `VkSurfaceKHR` with at least one queue family. At this point, we can also check that there is at least one queue family that supports graphics commands.

It is not guaranteed that any graphics queue family can present with the given surface – presenting might be handled by a different queue family. There does not seem to be any device, especially when looking at desktop computers, where the graphics *cannot* also do the presentation. Nevertheless, this is not guaranteed to be the case by the Vulkan specification, so if we are aiming to write a portable and correct Vulkan program, we should not assume such.



To facilitate the latter two checks, Exercise 3 introduces the `find_queue_family` function, which generalizes the old `find_graphics_queue_family` function. It takes two additional parameters. The first one, of type `VkQueueFlags`, lets the caller specify which queue flags must be set. The second argument, of type `VkSurfaceKHR`, indicates that the queue family must also support presenting to the specified surface. (However, if the second argument is set to `VK_NULL_HANDLE`, we ignore it.).

The second check requires the [vkGetPhysicalDeviceSurfaceSupportKHR](#) function that is also part of the `VK_KHR_surface` extension.

Implement `find_queue_famaily`:

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

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

```

for( std::uint32_t i = 0; i < numQueues; ++i )
{
    auto const& family = families[i];

    if( aQueueFlags == (aQueueFlags & family.queueFlags) )
    {
        if( VK_NULL_HANDLE == aSurface )
            return i;

        VkBool32 supported = VK_FALSE;
        auto const res = vkGetPhysicalDeviceSurfaceSupportKHR( aPhysicalDev, i, ▽
▷ aSurface, &supported );

        if( VK_SUCCESS == res && supported )
            return i;
    }
}

```

We can now implement the `score_device` function (which has already been updated to additionally take a `VkSurfaceKHR` parameter):

```

// Check that the device supports the VK_KHR_swapchain extension
auto const exts = lut::detail::get_device_extensions( aPhysicalDev );

if( !exts.count( VK_KHR_SWAPCHAIN_EXTENSION_NAME ) )
{
    std::fprintf( stderr, "Info: Discarding device '%s': extension %s missing\n", ▽
▷ props.deviceName, VK_KHR_SWAPCHAIN_EXTENSION_NAME );
    return -1.f;
}

// Ensure there is a queue family that can present to the given surface
if( !find_queue_family( aPhysicalDev, 0, aSurface ) )
{
    std::fprintf( stderr, "Info: Discarding device '%s': can't present to surface\n", ▽13
▷ props.deviceName );
    return -1.f;
}

// Also ensure there is a queue family that supports graphics commands
if( !find_queue_family( aPhysicalDev, VK_QUEUE_GRAPHICS_BIT ) )
{
    std::fprintf( stderr, "Info: Discarding device '%s': no graphics queue family\n", ▽20
▷ props.deviceName );
    return -1.f;
}

```

The `score_device` function has additionally been updated to print the reason for discarding devices that do not fulfill certain conditions. This is useful if, for some reason, the provided logic discards all possible Vulkan devices.



**Vulkan Device Creation** With an appropriate physical device selected, we can create the logical device instance. Unlike earlier exercises, we need enable some device extensions – specifically, the `VK_KHR_swapchain` extension mentioned earlier. Additionally, we might need to create two different queues: one for graphics commands and one for presentation. But, if possible, we prefer a single queue that supports both operations.

The `create_device` function has already been updated to take two additional arguments: first, a list of queue families from which to instantiate a queue; second, a list of device extensions to enable. Updating the logic of the function with this should be fairly straight forward at this point, so Exercise 3 provides the complete `create_device` definition for you. Inspect it briefly.

In `make_vulkan_window`, request the `VK_KHR_swapchain` extension to be enabled:

```

enabledDevExensions.emplace_back( VK_KHR_SWAPCHAIN_EXTENSION_NAME );

```

Below, implement the logic to select the queues that we wish to instantiate:

```

if( auto const index = find_queue_family( ret.physicalDevice, VK_QUEUE_GRAPHICS_BIT, &
    > ret.surface ) )
{
    ret.graphicsFamilyIndex = *index;

    queueFamilyIndices.emplace_back( *index );
}
else
{
    auto graphics = find_queue_family( ret.physicalDevice, VK_QUEUE_GRAPHICS_BIT );
    auto present = find_queue_family( ret.physicalDevice, 0, ret.surface );

    assert( graphics && present );

    ret.graphicsFamilyIndex = *graphics;
    ret.presentFamilyIndex = *present;

    queueFamilyIndices.emplace_back( *graphics );
    queueFamilyIndices.emplace_back( *present );
}

```

As before, we need to retrieve the `VkQueue` handles from the created device with `vkGetDeviceQueue`. If two separate queues were required, the second queue handle must be retrieved as well. The code for this is already in place in `make_vulkan_window`.

The final step in `make_vulkan_window` relates to swapchain creation. It is described in the following section.

### 3 Swapchain Setup

When creating the swapchain, we are faced with a number of options. One of the more important options is the image format. We cannot choose the image format freely (i.e., not even from a subset of formats like in Exercise 2), but must rather inquire about supported formats and choose from the provided list. The image format affects creation of several of the objects down the line, such as the render pass (which is an argument to both the framebuffer and the pipeline).

Looking at the final few lines of `make_vulkan_window`, you will find three function calls before the `return` statement. The first one is to `create_swapchain`, with which we will start.

**Swapchain Creation** To create the swapchain object, we need to decide on a few parameters. The image format was already mentioned. A similar approach is required for the presentation mode, as defined by [VkPresentModeKHR](#). Both the format and the presentation mode require us to enumerate possible values, and then select one from the returned options.

The `vulkan_window.cpp` source declares two local helper functions for this: `get_surface_formats` and `get_present_modes`. These return a list/set of the formats ([VkSurfaceFormatKHR](#)) and presentation modes (`VkPresentModeKHR`), respectively. The functions to enumerate the available choices are:

- [vkGetPhysicalDeviceSurfaceFormatsKHR](#) and
- [vkGetPhysicalDeviceSurfacePresentModesKHR](#).

Enumerating these properties follows the pattern already seen in Exercise 1: we call each function twice. The first call (last argument set to `nullptr`) fills in the number of formats/presentation modes into the provided `std::vector<uint32_t>`. We use the count to allocate a buffer to hold the returned options. The buffer is then passed to the second call, which fills the results into the buffer.

Implement the two functions `get_surface_formats` and `get_present_modes`. Note that the second function returns a `std::unordered_set`, so the entries of the filled-in buffer have to be inserted into a set before returning it. If necessary, refer back to (for example) the implementation of `get_instance_layers` in Exercise 1 for additional guidance.

Looking at `create_swapchain`, you will see the two above methods called at the beginning of the function. With the information about the available formats and modes in hand, we now have to pick appropriate ones from the possible options. In both cases, the appropriate choice will depend on the application. The method presented here is by no means the only correct one.



The `VkSurfaceFormatKHR` structure has two members: the `VkFormat` that we are already familiar with from earlier, and an additional [VkColorSpaceKHR](#) member that defines the color space of the display (“presentation engine”). The strategy for Exercise 3 (and other exercises) is to strongly prefer sRGB RGBA color formats with 8 bits per color channel and using the sRGB color space on the display. However, if no such format is available, we will just take whatever the first format returned to us is, and hope for the best. In practice this might look like the following:

```

// Pick the surface format
// If there is an 8-bit RGB(A) SRGB format available, pick that. There are two main variations possible here
// RGBA and BGRA. If neither is available, pick the first one that the driver gives us.
//
// See http://vulkan.gpuinfo.org/listsurfaceformats.php for a list of formats and statistics about where they're
// supported.
assert( !formats.empty() );

VkSurfaceFormatKHR format = formats[0];
for( auto const fmt : formats )
{
    if( VK_FORMAT_R8G8B8A8_SRGB == fmt.format && VK_COLOR_SPACE_SRGB_NONLINEAR_KHR ==
    > fmt.colorSpace )
    {
        format = fmt;
        break;
    }

    if( VK_FORMAT_B8G8R8A8_SRGB == fmt.format && VK_COLOR_SPACE_SRGB_NONLINEAR_KHR ==
    > fmt.colorSpace )
    {
        format = fmt;
        break;
    }
}

```

For the presentation mode ([VkPresentModeKHR](#)), we follow a similar idea. The presentation mode relates to when new images are shown on the display. This is often lumped into “v-sync”, but the options available to use a bit more subtle than this. Vulkan defines the following presentation modes by default (additional ones might be available from extensions, but they are not listed here):

**VK\_PRESENT\_MODE\_IMMEDIATE\_KHR** Do not wait for “v-sync” (the vertical blanking period) when switching to a new image; rather show it as soon as possible. This may result in tearing (i.e., where parts of different frames are visible on the display). This may reduce latency if we are able to render at a rate higher than the display’s refresh rate.

**VK\_PRESENT\_MODE\_MAILBOX\_KHR** In mailbox mode, we deposit a single image that is to be shown in the following vertical blanking period. This image is only displayed at the vertical blanking period, meaning that no tearing can occur. If we deposit a new image (e.g. the next frame) *before* the vertical blanking period, the previously deposited image is replaced with the new image. As such, the user will always see the most recent frame that was available when the display refreshes. The mailbox presentation mode is likely useful for applications where low latency is desirable, and where we can render the scene faster than the display refresh rate. Note that mailbox mode is somewhat wasteful in this case, as we may end up rendering many images that are never shown.

**VK\_PRESENT\_MODE\_FIFO\_KHR** The FIFO (first-in, first-out) displays rendered frames in the order they are submitted for presentation. Like the mailbox mode, a new image is only shown at the vertical blanking period, meaning that there is no tearing. Unlike mailbox mode, all submitted images are shown, which may cause additional latency. FIFO mode is the only presentation mode that is guaranteed to be available.

**VK\_PRESENT\_MODE\_FIFO\_RELAXED\_KHR** The “relaxed” FIFO mode combines the normal FIFO mode with some aspects of the immediate mode. It functions like FIFO mode if we submit an image for presentation before each vertical blanking period. If we miss the vertical blanking period, the next frame will instead be shown immediately. If we render new images at a fast enough rate, no tearing will occur. If we miss the deadline for a new frame, tearing might occur (but latency is slightly lower, as we do not have to wait for the next vertical blanking period).

Check the Vulkan documentation for `VkPresentModeKHR` for additional information. Each of the presentation modes is explained in-depth there.

All of the modes assume double-buffering (or more). There is no equivalent to single-buffered mode, which (legacy) OpenGL libraries occasionally advertise.



For Exercise 3, we will use `FIFO_RELAXED` if it is available, and otherwise fall back to the plain `FIFO` mode. The latter is guaranteed to be available:

```
// Pick a presentation mode
VkPresentModeKHR presentMode = VK_PRESENT_MODE_FIFO_KHR;

// Prefer FIFO_RELAXED if it's available.
if( modes.count( VK_PRESENT_MODE_FIFO_RELAXED_KHR ) )
    presentMode = VK_PRESENT_MODE_FIFO_RELAXED_KHR;
```

Next, we need to specify the number of images that the swapchain should use. Using two images would correspond to double-buffering, three images to triple-buffering. The Vulkan device specifies a range of valid choices (i.e., minimum number of images required, and maximal number of images permissible), but we are free to pick any number in this range. To figure out the range of choices, we have to query the physical device for the information. This is done with `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, which fills in a `VkSurfaceCapabilitiesKHR` structure with information specific to the selected device. The `minImageCount` and `maxImageCount` members of the structure define the range of valid choices (caveat: the `maxImageCount` may be set to zero, which indicates that there is no limit to the number of images we can choose, assuming there is memory to allocate those images).

The choice will depend slightly on the presentation mode that we intend to pick. Most presentation modes will function with just two images. Mailbox mode will need three images to function properly (one image is currently displayed, one image is deposited to be shown next, and one image that we render to at the same time). The [Vulkan tutorial](#) recommends using `minImageCount+1` images, and we will follow this recommendation:

```
// Pick an image count
VkSurfaceCapabilitiesKHR caps;
if( auto const res = vkGetPhysicalDeviceSurfaceCapabilitiesKHR( aPhysicalDev, ▽
    ▷ aSurface, &caps ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to get surface capabilities\n"
        "vkGetPhysicalDeviceSurfaceCapabilitiesKHR() returned %s", lut::to_string(res) ▽
    ▷ .c_str()
    );
}

std::uint32_t imageCount = 2;

if( imageCount < caps.minImageCount+1 )
    imageCount = caps.minImageCount+1;

if( caps.maxImageCount > 0 && imageCount > caps.maxImageCount )
    imageCount = caps.maxImageCount;
```

On my machine, running X11 with an NVIDIA RTX 2070, the minimum image count is two, and the maximum image count is eight. The above strategy would end up picking three images (triple-buffering) in this case.



We need to define the size/extent of the swap chain in pixels. Normally, this corresponds to the size of the renderable area of the window (the size of the window, minus any space taken by standard window decorations). If the Vulkan driver has this information, it exposes it in the `currentExtent` member of the swapchain capabilities queried earlier. If the driver does not have the information, it sets the extent to `(0xffffffff, 0xffffffff)` (the maximum value `std::uint32_t` can hold). In this case, the information needs to be queried directly from the operating system/window system. GLFW does for us with the `glfwGetFramebufferSize` function. The capabilities structure specifies a minimum and maximum extent (`minImageExtent` and `maxImageExtent`, respectively), so we must make sure our selected extent is within these limits:

```
// Figure out the swap extent
VkExtent2D extent = caps.currentExtent;
if( std::numeric_limits<std::uint32_t>::max() == extent.width )
{
    int width, height;
    glfwGetFramebufferSize( aWindow, &width, &height );
```



```

// Note: we must ensure that the extent is within the range defined by [ minImageExtent, maxImageExtent ]
auto const& min = caps.minImageExtent;
auto const& max = caps.maxImageExtent;

extent.width = std::clamp( std::uint32_t(width), min.width, max.width );
extent.height = std::clamp( std::uint32_t(height), min.height, max.height );
}

```

The [VkSwapchainCreateInfoKHR](#) structure exposes a few more options that we need to pin down. However, these are much simpler (but, as always, check the documentation for additional detail):

**imageArrayLayers** We use a single layer, so this should be set to one.

**imageUsage** Here, we specify how the images from the swapchain will be used. We already dealt with image usage flags in Exercise 2, when creating the framebuffer image there. Unlike Exercise 2, we only require `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, as we do not intend to copy the image data out of the image (and there is no separate image usage flag for presentation). Allowed image usage flags are listed in the `supportedUsageFlags` in the surface capabilities structure queried earlier.

**imageSharingMode** Like ordinary image setup, we specify an image sharing mode. By default, we choose `VK_SHARING_MODE_EXCLUSIVE`, but use `VK_SHARING_MODE_CONCURRENT` if the presentation queue is separate from the graphics queue. (This simplifies the rest of the code somewhat, as we do not need to introduce extra barriers to transfer ownership of the image between queues.)

**preTransform** Should be set to `currentTransform` from the capabilities queried early. Otherwise allows applying transformations to the image (see [VkSurfaceTransformFlagBitsKHR](#)). Allowed transforms are listed in the `supportedTransforms` field of the capabilities structure.

**compositeAlpha** Select alpha compositing mode when compositing rendered contents with other windows. Possible compositing modes are listed in the capabilities structure (`supportedCompositeAlpha`). If none are listed there, some modes might be available through platform-specific functionality outside of Vulkan.

**clipped** Determines whether areas in the swap chain images that are obscured by e.g. other windows needs to be preserved. Setting this to `VK_TRUE` is recommended by default (obscured content is *not* perserved). If we wish to read back content (e.g., for screen shots) from the swapchain images, we must however set it to `VK_FALSE` instead.

**oldSwapchain** If we are recreating the swapchain, we can pass the old swapchain object here. This might allow some resources associated with the swapchain to be reused. We will use this functionality later, when implementing swapchain re-creation. In preparation, the `create_swapchain` method accepts the handle to the old swapchain already (`aOldSwapchain`), so we set it to this handle. When creating the swapchain initially, this handle is set to `VK_NULL_HANDLE`, indicating that a new swapchain is indeed being created from scratch.

If we wanted to be able to save screenshots directly from our application, we would need to specify the `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` in addition to the `COLOR_ATTACHMENT_BIT`. Without it, we would not be allowed to copy image data out of the swap chain images. Additionally, we should set `clipped` to `VK_FALSE` to ensure that all pixels in the image are rendered/stored.



We should end up with something like the following:

```

// Finally create the swap chain
VkSwapchainCreateInfoKHR chainInfo{};
chainInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
chainInfo.surface = aSurface;
chainInfo.minImageCount = imageCount;
chainInfo.imageFormat = format.format;
chainInfo.imageColorSpace = format.colorSpace;
chainInfo.imageExtent = extent;
chainInfo.imageArrayLayers = 1;
chainInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
chainInfo.preTransform = caps.currentTransform;
chainInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
chainInfo.presentMode = presentMode;
chainInfo.clipped = VK_TRUE;

```

```

chainInfo.oldSwapchain      = aOldSwapchain;
15
16
if( aQueueFamilyIndices.size() <= 1 )
17
{
18
    chainInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
19
}
20
else
21
{
22
    // Multiple queues may access this resource. There are two options. SHARING_MODE_CONCURRENT
23
    // allows access from multiple queues without transferring ownership. EXCLUSIVE would require explicit
24
    // ownership transfers, which we're avoiding for now. EXCLUSIVE may result in better performance than
25
    // CONCURRENT.
26
    chainInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
27
    chainInfo.queueFamilyIndexCount = std::uint32_t(aQueueFamilyIndices.size());
28
    chainInfo.pQueueFamilyIndices = aQueueFamilyIndices.data();
29
}
30
31
VkSwapchainKHR chain = VK_NULL_HANDLE;
32
if( auto const res = vkCreateSwapchainKHR( aDevice, &chainInfo, nullptr, &chain ); ▽ 33
    > VK_SUCCESS != res )
34
{
35
    throw lut::Error( "Unable to create swap chain\n"
36
        "vkCreateSwapchainKHR() returned %s", lut::to_string(res).c_str()
37
    );
38
}
39
return { chain, format.format, extent };
40

```

**Swapchain Images and Views** Creating the swapchain automatically creates a number of associated swapchain images, greater or equal to the number of images that we requested during swapchain creation. Swapchain images are referenced by a `VkImage` handle just like ordinary images. We first need to retrieve these image handles. Next, we need to create image views (`VkImageView`) for each of the images, such that we can later include them in the framebuffer(s).

The `VkImage` handles for a `VkSwapchainKHR` are retrieved with [vkGetSwapchainImagesKHR](#). The function uses the same two-call pattern that we seen before: first call retrieves the number of images, and the second call writes the `VkImage` handles into a buffer that we provide. In this case, we want to write them into the `aImages` argument of type `std::vector<VkImage>&` that is passed to `get_swapchain_images`.

Implement the `get_swapchain_images` function.

Unlike “ordinary” images, the swapchain images are owned by the `VkSwapchainKHR` object. We are therefore not required (or allowed) to destroy these images ourselves. They will be destroyed automatically when the swapchain object is destroyed.



Next, we need to create a `VkImageView` for each of the images. The `create_swapchain_image_views` function is responsible for this. Although, we have already done so in Exercise 2, the following code demonstrates this process again:

```

for( std::size_t i = 0; i < aImages.size(); ++i )
1
{
2
    VkImageViewCreateInfo viewInfo{};
3
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
4
    viewInfo.image = aImages[i];
5
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
6
    viewInfo.format = aSwapchainFormat;
7
    viewInfo.components = VkComponentMapping{
8
        VK_COMPONENT_SWIZZLE_IDENTITY,
9
        VK_COMPONENT_SWIZZLE_IDENTITY,
10
        VK_COMPONENT_SWIZZLE_IDENTITY,
11
        VK_COMPONENT_SWIZZLE_IDENTITY
12
    };
13
    viewInfo.subresourceRange = VkImageSubresourceRange{
14
        VK_IMAGE_ASPECT_COLOR_BIT,
15
        0, 1,
16
        0, 1
17
    };
}

```

```

};
18
19
VkImageView view = VK_NULL_HANDLE;
20
if( auto const res = vkCreateImageView( aDevice, &viewInfo, nullptr, &view ); ▽
21
▷ VK_SUCCESS != res )
{
22
    throw lut::Error( "Unable to create image view for swap chain image %zu\n"
23
        "vkCreateImageView() returned %s", i, lut::to_string(res).c_str()
24
    );
25
}
26
27
aViews.emplace_back( view );
28
}
29

```

Note that we use the image format that we selected for the with the swapchain's images for the image views. (The code is slightly more compact than before, avoiding defining separate variables for the component mapping and subresource range.)

**Render Pass and Pipeline** We now turn our attention towards creating the render pass and the graphics pipeline. The set up that we perform for these is project specific (here, we render a single triangle defined with hardcoded clip space coordinates). Because of this, the definitions of the `create_render_pass`, `create_triangle_pipeline_layout` and `create_triangle_pipeline` functions remain in `main.cpp`.

Inspect the declaration of the three functions. You will see that two of these take the new `VulkanWindow` class as their argument, while the `create_triangle_pipeline_layout` keeps the old `VulkanContext`. The code for `create_triangle_pipeline_layout` is in fact unchanged – we can pass our `VulkanWindow` instance to it, since -thanks to public inheritance- the `VulkanWindow` [is-a](#) `VulkanContext`.

Transfer your code for `create_triangle_pipeline_layout` from Exercise 2 to Exercise 3. The code will remain unchanged.

For the render pass, some minor changes are needed. We need to account for the following:

- The single attachment's format is now the format that we selected during ssswapchain setup. (It is stored in `VulkanWindow` in the `swapchainFormat` member.)
- The final layout out the attachment is no longer `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` (which was previously required to be able to copy the image data out of the `VkImage`). Rather, it is now `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. This is a special layout defined specifically with respect to presentation (and it is, as such, part of the `VK_KHR_swapchain` extension).
- Exercise 2 used an explicit subpass dependency to provide synchronization (a barrier) between the rendering and the subsequent copy of the results. Exercise 3, which renders frames repeatedly, will require two subpass dependencies. One to synchronize with the previous frame (previous presentation) and one with the commands after (next presentation). The first one, from `SUBPASS_EXTERNAL` to our rendering subpass (0), we need to define ourselves. For the second one, notionally from our rendering subpass (0) to `SUBPASS_EXTERNAL`, the implicit subpass dependency generated by Vulkan suffices. You can find more information about the first dependency in the [Synchronization Examples page](#) in the Vulkan Wiki.

The render pass creation therefore becomes (`create_render_pass`):

```

VkAttachmentDescription attachments[1]{};
1
attachments[0].format      = aWindow.swapchainFormat; //changed!
2
attachments[0].samples     = VK_SAMPLE_COUNT_1_BIT;
3
attachments[0].loadOp      = VK_ATTACHMENT_LOAD_OP_CLEAR;
4
attachments[0].storeOp     = VK_ATTACHMENT_STORE_OP_STORE;
5
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
6
attachments[0].finalLayout  = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR; //changed!
7
8
VkAttachmentReference subpassAttachments[1]{};
9
subpassAttachments[0].attachment = 0; // this refers to attachments[0]
10
subpassAttachments[0].layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
11
12
VkSubpassDescription subpasses[1]{};
13
subpasses[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
14
subpasses[0].colorAttachmentCount = 1;
15
subpasses[0].pColorAttachments   = subpassAttachments;
16
17

```

```

// Requires a subpass dependency to ensure that the first transition happens after the presentation engine is
// done with it.
// https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples-(Legacy-synchronization-
// -APIs)#combined-graphicspresent-queue
//
// WARNING: the following has changed! Make sure to update it!
VkSubpassDependency deps[1]{};
deps[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
deps[0].srcSubpass      = VK_SUBPASS_EXTERNAL;
deps[0].srcAccessMask   = 0;
deps[0].srcStageMask    = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
deps[0].dstSubpass      = 0;
deps[0].dstAccessMask   = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
deps[0].dstStageMask    = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;

// https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VkRenderPassCreateInfo.html
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; // different dependency, same code
passInfo.pDependencies = deps; // different dependency, same code

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

return lut::RenderPass( aWindow.device, rpass );

```

The changes to the pipeline setup are minor as well: the `VkViewport` and the `VkRect2D` need to use the size of the swapchain images (`swapchainExtent` in `VulkanWindow`). Transfer your pipeline creation code from Exercise 3 and update it accordingly.

**Framebuffer** The swapchain defines multiple images. Each frame, we will select an image to draw to. In that frame, the image be used as the single color attachment/render target that we have declared in our render pass. The `VkFramebuffer` needs to match this use. For each image, we will define a separate `VkFramebuffer` that refers to the image in question. When we wish to render to a certain image, we select the corresponding framebuffer.

This process has now been simplified twice in Vulkan. First, in Vulkan 1.2, the *imageless framebuffers* enabled creation of a single framebuffer object, to which a corresponding image would be bound at render-time. Vulkan 1.3 introduces the *dynamic rendering* that obsoletes framebuffer objects entirely for the use cases that it covers.

Why the separation between the framebuffer object and the images in the first place? In Exercise 3, we need to deal with multiple `VkImage` objects that we draw to, cycling through them. This does not mean that all the framebuffer resources need to be duplicated (triplicated, or more). In Exercise 4, we will introduce a depth buffer for hidden surface removal. While we need multiple images, as we might be rendering to one while another one is presented, this is unnecessary for the depth buffer. A single depth buffer image will be sufficient.



The framebuffers are created in the `create_swapchain_framebuffer` function. It fills in the provided `std::vector` with `VkFramebuffer` objects wrapped into the `lut::Framebuffer` object introduced in Exercise 2. Aside from creating multiple framebuffers, the changes are similar to those in previous steps. The width and height match those of the swapchain, and we refer to the swapchain image's views:

```

for( std::size_t i = 0; i < aWindow.swapViews.size(); ++i )
{
    VkImageView attachments[1] = {

```

```

        aWindow.swapViews[i]
    };

    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 = aWindow.swapchainExtent.width;
    fbInfo.height = aWindow.swapchainExtent.height;
    fbInfo.layers = 1;

    VkFramebuffer fb = VK_NULL_HANDLE;
    if( auto const res = vkCreateFramebuffer( aWindow.device, &fbInfo, nullptr, &fb )
    ▷ ); VK_SUCCESS != res )
    {
        throw lut::Error( "Unable to create framebuffer for swap chain image %zu\n"
            "vkCreateFramebuffer() returned %s", i, lut::to_string(res).c_str()
        );
    }

    aFramebuffers.emplace_back( lut::Framebuffer( aWindow.device, fb ) );
}

```

## 4 Handling User Input

Before diving into the render loop, we need to discuss user interaction. For the typical interactive application, we need a way to receive events that correspond to e.g., user input (mouse, keyboard, ...). GLFW exposes this through a callback mechanism. In essence, we define a function that handles a certain type of input, and then register this function with GLFW. GLFW will then call this function when the corresponding event occurs (e.g., the user presses a key).

Unlike other frameworks (Qt, GLUT), GLFW does not take over the main loop of the application. Rather, we remain in control (and write the “main” loop ourselves – see `while` loop in `main()`). However, we need to periodically tell GLFW to check for and process events.

GLFW provides two functions for this purpose: [glfwPollEvents](#) and [glfwWaitEvents](#). These checks for events, and if there are any, they will process them appropriately. If there are none, `glfwPollEvents` returns immediately. This is crucial if we wish to maintain a steady frame rate, independent of user inputs.

In contrast, `glfwWaitEvents` will check for events, and if there are none, wait for an event to occur. This is useful if our application only needs to render in response to external events, such as user input, if the window is resized or uncovered, and so on. This is more efficient in terms of compute/energy usage when it is applicable.

Polling for events is more common in real-time rendering applications such as games. The exercises therefore use `glfwPollEvents`. You can change this to `glfwWaitEvents` if you wish, though.

There is a difference between polling for events via `glfwPollEvents` and (for example) polling for individual key states (which GLFW also supports). I would recommend *against* the latter – it is easy to miss short key presses, especially at suboptimal frame rates. It also adds an unnecessary overhead: with `glfwPollEvents`, we poll for events once per frame and the process all events that occurred. When polling specific key state, it is likely that one will end up making multiple queries and checks each frame, and even in frames where nothing has happened.



GLFW’s input handling is described in detail in its [input guide](#). For now, the exercises only listen for a single input event: a press of the escape key. This terminates the application. Inspect the corresponding code. The callback function is named `glfw_callback_key_press` (defined in `main.cpp`), and is registered with GLFW through the `glfwSetKeyCallback` function.

## 5 Main Loop

(Setup — Rendering — Swapchain Re-creation)

The *main* loop is where the application spends most of its time. In each iteration of the loop, we will process events (if any), render and display a frame. A more complex application may additionally choose to update internal application state (such as animations, simulation, camera views, ...). The current application does not have any such state, so we will omit this for now.

Event process is mostly handled via GLFW's mechanism: i.e., the application calls `glfwPollEvents` and leverages GLFW's event handling system to receive and react to relevant events (Section 4). There is one exception to this, namely Vulkan may notify us that the swapchain has become out of date or suboptimal and needs to be re-created. The most common cause of this is that the window has been resized.

For rendering, we need to perform a number of steps. First, we need to decide which swapchain image we can render to. Next, we will record the commands necessary for rendering the relevant frame. We prepare several Vulkan command buffer objects ahead of time, and re-use these. When re-using command buffers, it is our responsibility to ensure that the command buffer is available for re-use. This means that if it was used to submit commands for execution in an earlier frame (putting the command buffer into the *pending* state), these commands must have finished executing (command buffer is no longer in the *pending* state). Similar to Exercise 2, we will use fences to provide this synchronization.

While the commands are unchanged between frames in this particular application, which would theoretically allow us to re-use command buffers between frames, Exercise 3 will not do so. Instead, commands will be recorded anew each frame, which is much closer to how many "real-world" applications need to function.



After recording the rendering commands, they are submitted for execution. The rendering commands may only execute when the chosen swapchain image is no longer being used elsewhere (e.g. presentation). Vulkan provides two options here. We can be notified via a fence (meaning that we will wait on the CPU) or via a semaphore (meaning that the wait will take place on the device). Exercise 3 uses the latter.

Subsequently, we will *present* the resulting image. This will show the rendered image according to the presentation mode that we chose earlier in this exercise. Like Exercise 2, this requires some synchronization – we must ensure that the rendering commands have finished before we can do anything with the rendered results. In Exercise 3, we will rely on semaphores to provide this synchronization, as both rendering commands and presentation take place on the GPU.

**Setup** There are a few outstanding objects that we need to create before we enter the main loop:

- Command buffers and associated fences
- A command pool to allocate the command buffers from
- Two semaphore objects: one to delay rendering commands from executing until the swapchain image is ready, and one to delay presentation of the swapchain image until the rendering commands have finished

For simplicity, we choose to allocate command buffers from a single command pool. In order for this to work, we need to be able to reset the individual command buffers independently from each other. The command pool is therefore created with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` flag. Locate the call to `create_command_pool` in `main.cpp`, and you will see that this flag is passed to the function, along with the `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` flag. The `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` indicates that the command buffers allocated from the pool are short-lived, and are reset after a short time. Make sure to update your implementation of `create_command_pool` in `labutils` to pass the additional `aFlags` argument to the `VkCommandPoolCreateInfo` via its `flags` member.

We then allocate a number of command buffers from the pool and create a fence for each command buffer. The number of command buffers and fences is equal to the number of swapchain images. This is a particular choice that we make, so these two numbers do not have to match. However, having them match simplifies some aspects of the code.

Here, we simply re-use the `alloc_command_buffer` and `create_fence` functions from Exercise 2. Note that `create_fence` is passed an additional parameter, `VK_FENCE_CREATE_SIGNALED_BIT`. Update your implementation of `create_fence` similar to `create_command_pool`, i.e., make sure that the new `aFlags` argument to `create_fence` is copied to `VkFenceCreateInfo`'s `flags` member.

The final two objects that we need are the semaphores `imageAvailable` and `renderFinished`. Implement the corresponding `create_semaphore` function in `labutils`:

```
VkSemaphoreCreateInfo semaphoreInfo{};
semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
```

1

2

3



```

VkSemaphore semaphore = VK_NULL_HANDLE;
if( auto const res = vkCreateSemaphore( aContext.device, &semaphoreInfo, nullptr, & 5
    ▷ semaphore ); VK_SUCCESS != res )
{
    throw Error( "Unable to create semaphore\n"
        "vkCreateSemaphore() returned %s", to_string(res).c_str()
    );
}

return Semaphore( aContext.device, semaphore );

```

**Rendering** Look at the structure of the main render loop (the `while` loop) in `main`. You will see that it starts with a call to `glfwPollEvents` (Section 2). Next, there is some logic to re-create the swapchain and associated objects when necessary. This is treated in the next part, [swapchain re-creation](#).

The first step is to acquire a swapchain image to render to. Vulkan provides, via the `VK_KHR_swapchain` extension, the [`vkAcquireNextImageKHR`](#) function for this. Study the documentation for the various options that can be passed to it. In essence, `vkAcquireNextImageKHR` returns the index of the swapchain image that we should use for rendering via the `pImageIndex` out-parameter. As previously mentioned, Exercise 3 uses the `VkSemaphore`-based synchronization mechanism. We pass the handle to a semaphore object to `vkAcquireNextImageKHR`. When the returned swapchain image is ready to be rendered to, the semaphore will be signalled.

Like many other Vulkan functions, `vkAcquireNextImageKHR` returns a return value of type `VkResult`. However, in the case of `vkAcquireNextImageKHR`, there are two special return conditions that we wish to handle specifically:

- `VK_SUBOPTIMAL_KHR`
- `VK_ERROR_OUT_OF_DATE_KHR`

Both indicate that the swapchain needs to be recreated. The difference between the two status codes is that the former indicates that while some of the surface properties have changed, the swapchain can still be used for rendering and presentation (i.e., this is technically not an error, but a qualified success). The latter indicates that the surface has changed in a way that makes it impossible to use the current swapchain, and the swapchain must be re-created before proceeding. The simplest option is to unconditionally re-create the swapchain in both cases:

```

// Acquire next swap chain image
std::uint32_t imageIndex = 0;
auto const acquireRes = vkAcquireNextImageKHR(
    window.device,
    window.swapchain,
    std::numeric_limits<std::uint64_t>::max(),
    imageAvailable.handle,
    VK_NULL_HANDLE,
    &imageIndex
);

if( VK_SUBOPTIMAL_KHR == acquireRes || VK_ERROR_OUT_OF_DATE_KHR == acquireRes )
{
    // This occurs e.g., when the window has been resized. In this case we need to recreate the swap chain to
    // match the new dimensions. Any resources that directly depend on the swap chain need to be recreated
    // as well. While rare, re-creating the swap chain may give us a different image format, which we should
    // handle.
    //
    // In both cases, we set the flag that the swap chain has to be re-created and jump to the top of the loop.
    // Technically, with the VK_SUBOPTIMAL_KHR return code, we could continue rendering with the
    // current swap chain (unlike VK_ERROR_OUT_OF_DATE_KHR, which does require us to recreate the
    // swap chain).
    recreateSwapchain = true;
    continue;
}

if( VK_SUCCESS != acquireRes )
{
    throw lut::Error( "Unable to acquire next swapchain image\n"

```

```

        "vkAcquireNextImageKHR() returned %s", lut::to_string(acquireRes).c_str()
    );
}

```

Before recording rendering commands, we need to ensure that the command buffer is not in use. We do so with `vkWaitForFences`. In addition, we reset the fence to an unsignalled state, such that it can be reused immediately:

```

// Make sure that the command buffer is no longer in use
assert( std::size_t(imageIndex) < cbfences.size() );

if( auto const res = vkWaitForFences( window.device, 1, &cbfences[imageIndex].handle,
    VK_TRUE, std::numeric_limits<std::uint64_t>::max() ); VK_SUCCESS != res )
{
    throw lut::Error( "Unable to wait for command buffer fence %u\n"
        "vkWaitForFences() returned %s", imageIndex, lut::to_string(res).c_str()
    );
}

if( auto const res = vkResetFences( window.device, 1, &cbfences[imageIndex].handle );
    VK_SUCCESS != res )
{
    throw lut::Error( "Unable to reset command buffer fence %u\n"
        "vkResetFences() returned %s", imageIndex, lut::to_string(res).c_str()
    );
}

```

By creating the fences with `VK_FENCE_CREATE_SIGNALED_BIT`, this code is quite a bit simpler than the alternative. Each fence is initially signalled, meaning that the first time we wait on it, the wait will return immediately. If the fences were created unsignalled, we would wait indefinitely, as there is nothing signalling the fence (yet).



Similar to Exercise 2, we use the `record_commands` and `submit_commands` helper functions to record and submit commands, respectively:

```

// Record and submit commands for this frame
assert( std::size_t(imageIndex) < cbuffers.size() );
assert( std::size_t(imageIndex) < framebuffers.size() );

record_commands(
    cbuffers[imageIndex],
    renderPass.handle,
    framebuffers[imageIndex].handle,
    pipe.handle,
    window.swapchainExtent
);

submit_commands(
    window,
    cbuffers[imageIndex],
    cbfences[imageIndex].handle,
    imageAvailable.handle,
    renderFinished.handle
);

```

Implement the `record_commands` function. It is simpler than in Exercise 2, as there is no need for the additional copy commands that transferred the rendered image to a `VkBuffer` (the last recorded command should be `vkCmdEndRenderPass`). Make sure to set the `renderArea` of the `VkRenderPassBeginInfo` structure correctly (`aImageExtent` argument).

The `submit_commands` function needs to be updated to both wait and signal the two semaphores. We must wait for the `imageAvailable` semaphore to become signalled, indicating that the swapchain image is ready, before we draw to the image. Specifically, the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage of the rendering commands may only start executing once the image is ready. Other pipeline stages (vertex shaders etc) may in fact start earlier, as they do not access the image.

Secondly, we want to signal the `renderFinished` semaphore when commands have finished, to indicate that the rendered image is ready for presentation.

In practice, this looks like the following:

```

VkPipelineStageFlags waitPipelineStages = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;

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

submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &aWaitSemaphore;
submitInfo.pWaitDstStageMask = &waitPipelineStages;

submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &aSignalSemaphore;

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

```

Finally, we can return to implement last part of the render loop, namely, the presentation. We request a swapchain image to be presented via the `vkQueuePresentKHR` function. Study the documentation. We pass the index of the image that we want to present via the `pImageIndices` member of the `VkPresentInfoKHR` structure. This is the index that we previously acquired with `vkAcquireNextImageKHR`. Additionally, we pass the `renderFinished` semaphore to `pWaitSemaphores`, to indicate that presentation should only occur once the semaphore is signalled (which happens when our rendering commands have finished executing):

```

// Present the results
VkPresentInfoKHR presentInfo{};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &renderFinished.handle;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &window.swapchain;
presentInfo.pImageIndices = &imageIndex;
presentInfo.pResults = nullptr;

auto const presentRes = vkQueuePresentKHR( window.presentQueue, &presentInfo );

if( VK_SUBOPTIMAL_KHR == presentRes || VK_ERROR_OUT_OF_DATE_KHR == presentRes )
{
    recreateSwapchain = true;
}
else if( VK_SUCCESS != presentRes )
{
    throw lut::Error( "Unable present swapchain image %u\n"
        "vkQueuePresentKHR() returned %s", imageIndex, lut::to_string(presentRes).c_str()
    );
}

```

Similar to `vkAcquireNextImageKHR`, `vkQueuePresentKHR` may return a `VkResult` indicating that the swapchain needs to be recreated. As with `vkAcquireNextImageKHR`, we set the `recreateSwapchain` flag to true, to signal that the swapchain must be re-created before the next frame can be rendered.

**Swapchain Re-creation** When required, we must re-create the swapchain and any resources that depend on it (e.g., the any resources that depend on the swapchain's images, image format or extent). In Exercise 3, this means that the following resources:

- The swapchain object itself, and the associated image views
- The framebuffers, since the image views have changed
- The render pass object (if the swapchain's image format changed)
- The graphics pipeline (if the image size changed)

Additionally, we need to ensure that any old objects are destroyed appropriately. The `labutils`-wrappers will aid with this. When destroying objects, we must be careful to ensure that the objects are no longer in use. We do so with `vkDeviceWaitIdle`, which causes the host program to wait for all commands on all queues for a certain logical device to finish executing.

The first part of the swapchain re-creating takes place in `labutils`, in `vulkan_window.{hpp, cpp}`. Locate the `recreate_swapchain` function definition. We first destroy the old image views. We then create a new swapchain, passing the old swapchain object along, to give Vulkan the opportunity to reuse internal resources. We then destroy the old swapchain object. Finally, we create the new image views. Much of the code already exists, in form of the various helper functions that we previously defined:

```
// Remember old format & extents
// These are two of the properties that may change. Typically only the extent changes (e.g., window resized),
// but the format may in theory also change. If the format changes, we need to recreate additional resources.
auto const oldFormat = aWindow.swapchainFormat;
auto const oldExtent = aWindow.swapchainExtent;

// Destroy old objects (except for the old swap chain)
// We keep the old swap chain object around, such that we can pass it to vkCreateSwapchainKHR() via the
// oldSwapchain member of VkSwapchainCreateInfoKHR.
VkSwapchainKHR oldSwapchain = aWindow.swapchain;

for( auto view : aWindow.swapViews )
    vkDestroyImageView( aWindow.device, view, nullptr );

aWindow.swapViews.clear();
aWindow.swapImages.clear();

// Create swap chain
std::vector<std::uint32_t> queueFamilyIndices;
if( aWindow.presentFamilyIndex != aWindow.graphicsFamilyIndex )
{
    queueFamilyIndices.emplace_back( aWindow.graphicsFamilyIndex );
    queueFamilyIndices.emplace_back( aWindow.presentFamilyIndex );
}

try
{
    std::tie(aWindow.swapchain, aWindow.swapchainFormat, aWindow.swapchainExtent) = ▽
    ▷ create_swapchain( aWindow.physicalDevice, aWindow.surface, aWindow.device, ▽
    ▷ aWindow.window, queueFamilyIndices, oldSwapchain );
}
catch( ... )
{
    // Put back the old swap chain handle into the VulkanWindow; this ensures that the old swap chain is
    // destroyed when this error branch occurs.
    aWindow.swapchain = oldSwapchain;
    throw;
}

// Destroy old swap chain
vkDestroySwapchainKHR( aWindow.device, oldSwapchain, nullptr );

// Get new swap chain images & create associated image views
get_swapchain_images( aWindow.device, aWindow.swapchain, aWindow.swapImages );
create_swapchain_image_views( aWindow.device, aWindow.swapchainFormat, aWindow.▽
    ▷ swapImages, aWindow.swapViews );

// Determine which swap chain properties have changed and return the information indicating this
SwapChanges ret{};
```

```

if( oldExtent.width != aWindow.swapchainExtent.width || oldExtent.height != aWindow.
    > swapchainExtent.height )
    ret.changedSize = true;
if( oldFormat != aWindow.swapchainFormat )
    ret.changedFormat = true;

return ret;

```

After re-creating the swapchain object and the image views, we simply re-create the other objects as necessary with the various helper functions that we have defined. In the `recreateSwapchain` branch, implement this as follows:

```

// We need to destroy several objects, which may still be in use by the GPU. Therefore, first wait for the GPU
// to finish processing.
vkDeviceWaitIdle( window.device );

// Recreate them
auto const changes = recreate_swapchain( window );

if( changes.changedFormat )
    renderPass = create_render_pass( window );

framebuffers.clear();
create_swapchain_framebuffers( window, renderPass.handle, framebuffers );

if( changes.changedSize )
    pipe = create_triangle_pipeline( window, renderPass.handle, pipeLayout.handle );

```

Make sure that you do not remove the `recreateSwapchain = false` statement at the end of the branch. Doing so would cause the swapchain to be re-created each frame (if it is ever recreated). Re-creating the swapchain and related resources is quite expensive, so this would definitively impact frame rates negatively.



## Wrapping up

As always, compile and execute the program. You should see a window that renders the triangle as shown in the “teaser” on the first page of the exercise. (Or two colorful triangles if you implemented some of the optional parts of Exercise 2).

Try resizing the window. Make sure the rendering adapts to the new window size (i.e., the triangle should remain centered and should not be cut off). Check the validation messages. Ensure that there are no untoward validation errors.

I occasionally get an validation message with ID 0x7cd0911d

```

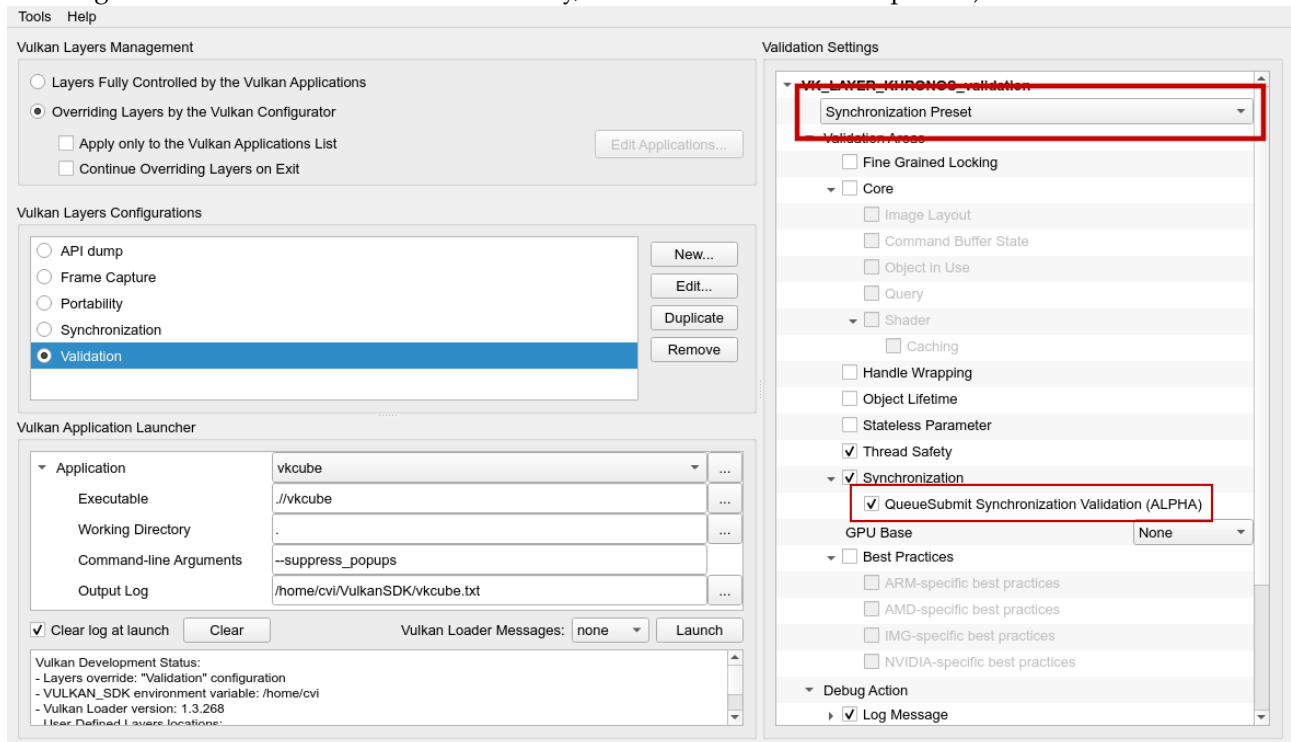
SEVERITY_ERROR (VALIDATION): VUID-VkSwapchainCreateInfoKHR-imageExtent-01274
(2094043421)
Validation Error: [ VUID-VkSwapchainCreateInfoKHR-imageExtent-01274 ] Object 0:
handle = 0x5567507a0338, type = VK_OBJECT_TYPE_DEVICE; | MessageID = 0x7cd0911d
| vkCreateSwapchainKHR() called with imageExtent = (1067,1027), which is outside
the bounds returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR():
currentExtent = (1067,1025), minImageExtent = (1067,1025), maxImageExtent =
(1067,1025). The Vulkan spec states: imageExtent must be between minImageExtent
and maxImageExtent, inclusive, where minImageExtent and maxImageExtent are
members of the VkSurfaceCapabilitiesKHR structure returned by
vkGetPhysicalDeviceSurfaceCapabilitiesKHR for the surface (https://www.khronos.
org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#VUID-
VkSwapchainCreateInfoKHR-imageExtent-01274)

```



during the process of resizing the window (i.e., when actively dragging the window to change size). The message is not printed always, and stops once I stop actively resizing the window. I suspect that this is a parallel processing hazard, where the window is further resized in-between the swapchain re-creation and the validation layer’s check of the correct window size. If you get the *occasional* validation message of this type *while* actively resizing the window, you can disregard it. Make sure the message does not repeat, and especially that the message stops when you no longer resize the window.

Configure the validation layers to check for synchronization errors. You can do so with the `vkconfig` / *Vulkan Configurator Tool* utility from the Vulkan SDK. Switch from *Standard Preset* to the *Synchronization Preset* in the right-hand panel. This should enable the *Synchronization* validation area in the list below the presets (avoid enabling different validation areas unnecessarily, as this seems to affect the presets):



If you have a sufficiently recent Vulkan SDK, you should experiment with the *QueueSubmit Synchronization Validation (ALPHA)* option in the *Synchronization* category.

Leave the configurator tool running while you run the Exercise 3 application (closing the configurator will reset the validation to the default state). Check that there are no synchronization problems.

## Optional experiments





There are a few optional experiments that you can conduct. Suggestions:

**Explicit subpass dependencies** Exercise 3's render pass setup relied on one implicit subpass dependencies that Vulkan automatically inserts when we do not specify it ourselves. Define the second explicit subpass dependencies, with `srcSubpass = 0` and `dstSubpass = SUBPASS_EXTERNAL`, to provide the necessary barriers/synchronization. Make sure that you double check for synchronization-related issues with the synchronization validation described above (preferably with the experimental QueueSubmit synchronization enabled!).

**Dynamic pipeline state** We currently re-create the graphics pipeline when the swapchain's image extent changes. This is required as we specify the image extent in the `VkPipelineViewportStateCreateInfo` structure. We can avoid this. We do so by declaring the viewport and scissor properties to be *dynamic states* using the `VkPipelineDynamicStateCreateInfo` structure (which we have omitted thus far). Specifically, we would enable the `VK_DYNAMIC_STATE_VIEWPORT` and `VK_DYNAMIC_STATE_SCISSOR` dynamic states. In this case, we must set the viewport and scissor regions via the `vkCmdSetViewport` and `vkCmdSetScissor` commands when recording rendering commands.

Doing so, you will no longer need to re-create the graphics pipeline when only the swapchain's image extent changes (the format is much less likely to change).

## 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.