Getting started with the Vulkan Graphics API – The first Triangle

Contents

Prerequisites

All samples are tested with NDK and Android Studio versions as specified in the respective sample readme files. Please refer to those files for specific details. Any device that samples are to be deployed to must be supported by the Qualcomm® Adreno™ SDK for Vulkan™.

The Triangle Sample – your first triangle in Vulkan

The triangle sample is aimed at those who want to see the smallest number of Vulkan setup operations required to draw a single vertex-coloured triangle to the screen. There is no asset loading, and the required fragment and vertex shader binaries are embedded straight into a header file for simplicity.

There has been an effort made to avoid abstractions and helpers in this example as to show all the Vulkan API requirements very explicitly.

Loading the sample in Android Studio

The samples are already laid out as Android Studio projects, ready to be loaded into the IDE. Upon launching Android Studio, select “Open an existing Android Studio project” from the welcome dialog, or “Open” from the file menu.

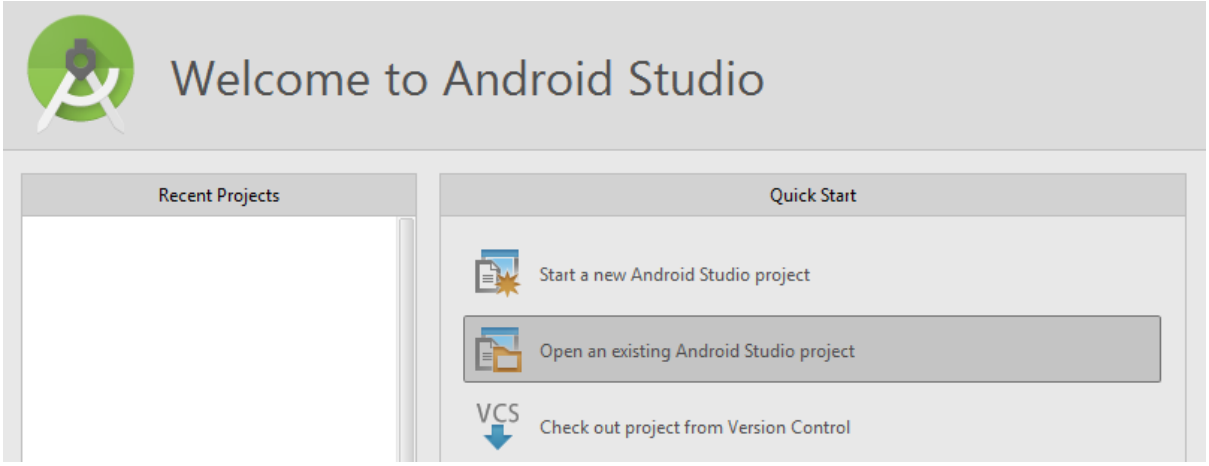


Figure 1 Welcome Dialog

Browse to the ‘triangle’ sample folder, and select it as the project to open.

Upon loading completing, you may be presented with an error about the location of the NDK.

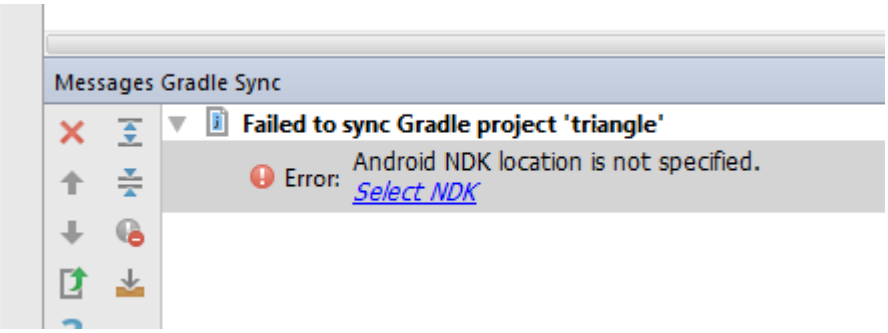


Figure 2 NDK location error

If so, click ‘Select NDK’ and point to the directory where it is installed in your system.

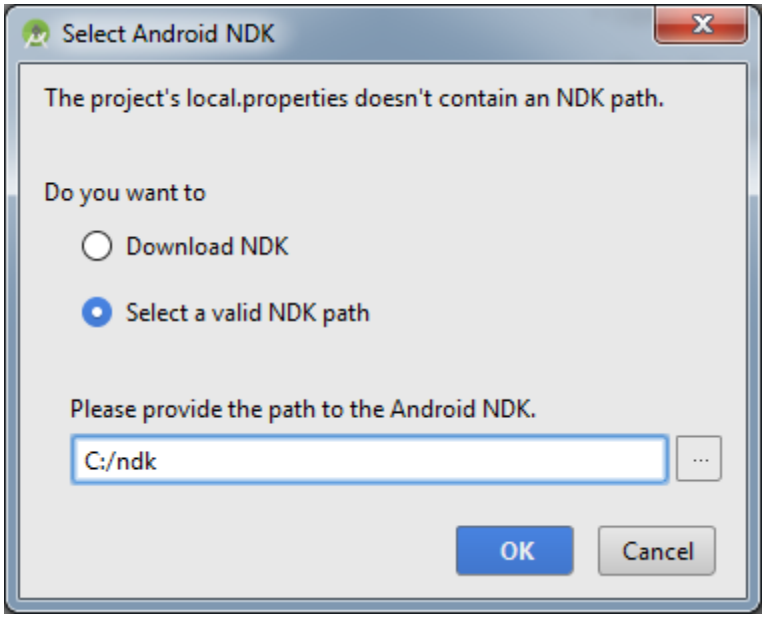


Figure 3 Specifying correct NDK path

At this point you can Build the sample, and then Run it on a supported device. It should display a triangle facing downwards, vertex coloured with red, green and blue corners, on a grey background.

There is no animation, it is a static sample.

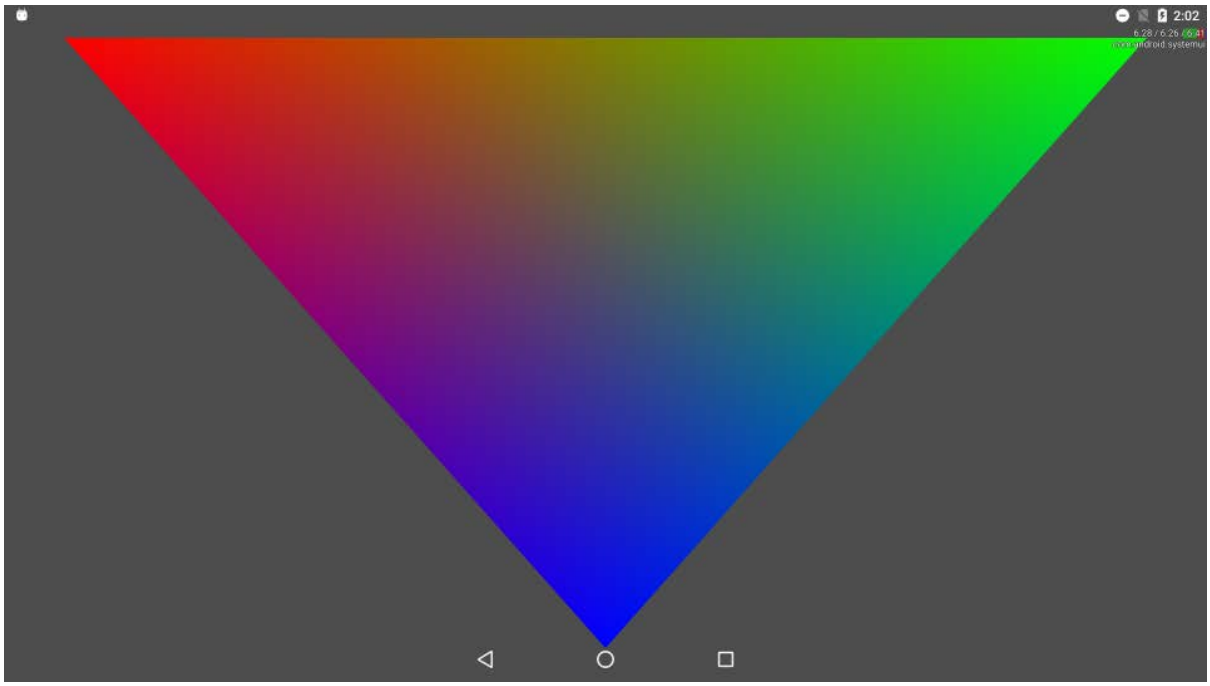


Figure 4 'triangle' sample running on tablet form factor

To view code of the project, open the Project pane. You can then expand `app` to view sources under the `jni` directory. Under `jni` there is another directory called `native_app_glue`. The files in that directory are from the base NDK 'native-activity' sample, and are un-modified. They allow applications and APKs be built easily without user-created Java boilerplate code.

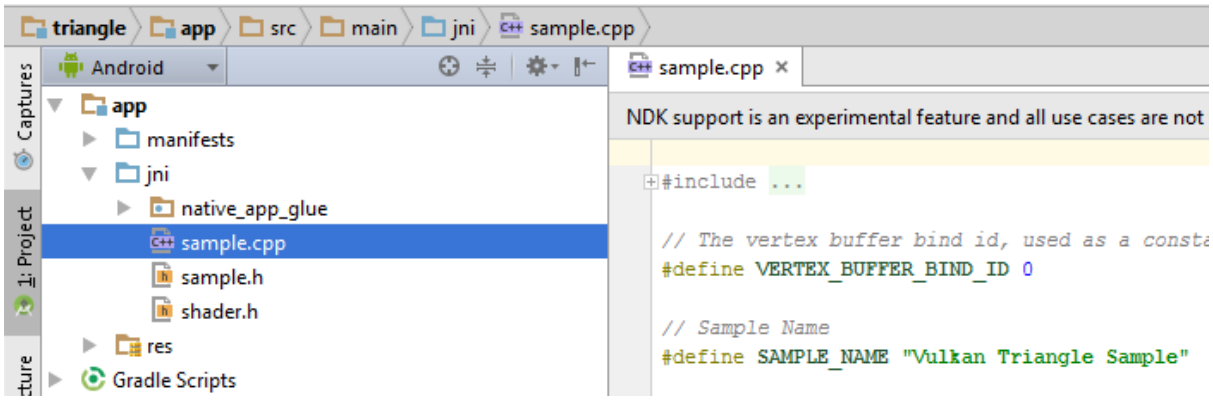


Figure 5 Android Studio Project view

The project is now loaded in Android Studio and can be viewed, edited and rebuilt to suit your own particular needs.

The sample contains three source files:

sample.cpp	VkSample implementation and native-activity Android entry point.
sample.h	VkSample class definition, helper structures and defines.
shader.h	Four global variables defining the code and size of code of our Vertex and Fragment shader binaries. It is compiled SPIR-V, created with glslang. It is embedded into the code instead of being loaded from an asset to focus on using the Vulkan API first.

What follows in this document is a step-by-step guide of what is required to implement the rendering of a static triangle using the Vulkan API.

Initialization

Overview

The `VkSample::Initialize` function contains function calls separated out for the various subsystems. The general process comes down to:

- Collecting required Android system hooks.
- Creating a Vulkan instance.
- Selecting a physical device.
- Creating the device.
- Initializing the presentation surface and swapchain.
- Creating command buffer structures.
- Creating vertex buffers and allocating device memory.
- Initializing descriptor layouts, and pipeline layouts.
- Creating the render pass state.
- Initializing the pipeline with all relevant state.
- Creating all frame buffer structures.
- Creating required synchronization primitives.
- Recording the scene into command buffers.

- Setting up the swapchain in preparation for rendering.

The Vulkan API defines functions which take additional parameters in the form of `*Info` structures. It is best practice to initialize these structures to zero before writing relevant data. This can be done by assigning to `{}` or using `memset()`, as demonstrated below. The sType should also always be set to the relevant enum value for the type.

```
VkPipelineCacheCreateInfo pipelineCache = {};
pipelineCache.sType = VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
```

```
VkPipelineCacheCreateInfo pipelineCache;
memset(&pipelineCache, 0, sizeof(pipelineCache));
pipelineCache.sType = VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
```

Required Android System Objects

The only object required for this sample is the `ANativeWindow*` structure, which is given to the native activity boilerplate code when the Android subsystem displays the sample to the screen. The initialize function gets passed this `window` and it is saved in a member variable for later use.

Creating the Vulkan Instance

The Vulkan instance is created in the `VkSample::CreateInstance()` function. As we defined VK_PROTOTYPES before including `<vulkan/vulkan.h>` we can directly call API functions, like `vkEnumerateInstanceExtensionProperties`.

We call that function initially to discover the number of extensions listed in the instance properties, as to query what is supported. When we create a Vulkan instance we pass it a list of the extensions we wish to enable, so querying whether extensions we require are supported first is always best practice.

The instance extensions we require are VK_KHR_surface and VK_KHR_android_surface. When we included the Vulkan header we also defined VK_USE_PLATFORM_ANDROID_KHR which allows us to use the functions specifically for that extension. We detect the extensions in the list returned from `vkEnumerateInstanceExtensionProperties` and build a list of extensions we require.

Before calling `vkCreateInstance` we must populate `Info` structures which contain all relevant data for the API to act on. It has fields for the application name, and the version of Vulkan that the sample was compiled with. This information will help the Vulkan driver provide the correct API version for the developer to utilize.

The `vkCreateInstance` function can specify a custom memory allocator, but for the purposes of this sample we leave that as undefined by passing `nullptr` in its place.

There are several instance layers that can be passed in to enable validation. Validation is an extremely useful optional feature that provides useful warning and error messages about Vulkan syntax and usage. In this sample validation can be toggled on by setting the `bUseValidation` flag. When this flag is set, the appropriate layers are detected and set when `vkCreateInstance` is called. A validation callback function is also connected to Vulkan. Warnings and error messages will be logged in the `DebugReportCallback` function.

Vulkan API functions can return `VkResult`, containing either VK_SUCCESS or an error code. After creating our Vulkan Instance and storing it in `mInstance`, we perform error checking. Then we return from the function to continue with further initialization.

Obtaining a Physical Device

Each instance will give access to a list of physical devices to operate with. The `VkSample::GetPhysicalDevices()` function simply obtains the list of physical devices available, selects the first one as our primary physical device, and obtains details such as memory properties which is used later in the sample when allocating buffers.

Creating the Device, Queue and Android Surface

Once we have an instance and our selected physical device, we can initialize the `VkDevice` structure, used throughout the remainder of the sample. `VkSample::InitDevice()`queries for the VK_KHR_`swapchain` extension in the same way as we dealt with instance extensions earlier. This extension is essential, and provides the mechanism for presenting images to the screen.

At this point we use the `ANativeWindow*` object provided by the Android system to initialize our display surface, using the VK_KHR_`android_surface` extension we initialized our vulkan instance with. This is performed in the `VkSample::InitSurface()` function. We create the surface now so that we can query supported display formats and ensure our device will work with those formats.

When we create our device object, we also provide information for creating the queues which accept our command buffers later in the sample. We are using a single queue for both graphical operations and presentation – and we query using `vkGetPhysicalDeviceQueueFamilyProperties` and `vkGetPhysicalDeviceSurfaceSupportKHR` to ensure we get a queue family index identifier which can be used for both types of operation.

Now we have enough details to finally create the device, and request the creation of our queue along with it. The `VkDeviceCreateInfo` structure points to a `VkDeviceQueueCreateInfo` structure which uses our queue family index and sets a priority of `1.0`, the highest. The presentation queues should always have a high priority, to avoid stuttering.

We also obtain the queue handle via a call to `vkGetDeviceQueue`.

Creating the Swapchain

The swapchain is very important if you want to present images to the screen in an efficient manner. You can think of it as a queue of backbuffers, with each frame rendering into a set buffer which is then queued for display at the relevant rate.

Before creating our swapchain in `VkSample::InitSwapchain()` we must ensure the surface formats we wish to use is supported. Generally, the basic 32-bit RGBA `VK_FORMAT_B8G8R8A8_UNORM` is supported. Once we have our format, we query the capabilities of the surface, which provides us with supported resolutions and the currently selected extent of the surface. In this example, we simply render at the full current extents of the surface, which can be significant (4K in many devices).

When creating the swapchain we provide a variety of parameters; we provide the minimum number of images the swapchain has internally via `surfaceCapabilities.minImageCount`, the dimension of the images and their format, and the present mode. There are a variety of present modes, but `VK_PRESENT_MODE_FIFO_KHR` is always supported and provides more than enough performance for the needs of a most applications.

After creating the swapchain with `vkCreateSwapchainKHR`, we obtain `VkImage` handles to the images which will become our backbuffers via `vkGetSwapchainImagesKHR`.

In our sample we group the resources needed for our swapchain buffers into a struct, `SwapchainBuffer`. Now that we have the `VkImage`, we need to create a view into that image. We state via `VkImageViewCreateInfo` structures the required context, and create VkimageView handles using `vkCreateImageView`.

At this point we can create an image to use as the depth buffer. Although we create one for each swap chain image in this example, another valid methodology is to share a single depth buffer.

Creating an image is simple, using `vkCreateImage` passing in an `Info` structure with format, size details and how it is used. For our depth buffers we state usage as `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

When you create a VkImage handle, it's important to remember that it's simply state and there is no storage associated with it. We now need to allocate memory though the Vulkan API to store the image data, and then bind that memory to the newly created image. When allocating memory, we need to be very specific, and as such Vulkan provides `vkGetImageMemoryRequirements` for obtaining the requirements on memory for any given `VkImage` handle. With the requirements obtained, we now need to find the correct memory type for the image. There are various different kinds of memory that can be allocated, and they are all handled via bits in a set which is provided by the physical device memory properties. The `GetMemoryTypeFromProperties` helper function will return a `memoryTypeIndex` for use in allocating memory given the `memoryTypeBits` field of the returned image memory requirements.

Calling `vkAllocateMemory` with our `VkMemoryAllocateInfo` will provide us with a `VkDeviceMemory` handle to the newly allocated memory. By calling `vkBindImageMemory` the creation of the image is now complete.

We create a view into that image as with the swapchain images above, and complete the creation of depth buffers, and essentially complete the initialization of the swapchain.

Creating the Command buffers

When creating command buffers in `VkSample::InitCommandbuffers()`, a command pool is specified. We create a pool to allocate from using `vkCreateCommandPool`, specifying the queue family the buffers will be submitted to.

When working with swapchains, ideally there should be one command buffer per swap-chain. We use the `vkAllocateCommandBuffers` function to allocate our command buffers, ensuring the level of our command buffers, specified in the `VkCommandBufferAllocateInfo` structure, is set to `VK_COMMAND_BUFFER_LEVEL_PRIMARY`. Only primary buffers can be submitted to device queues, and secondary queues are out with the scope of this example.

Allocating and initializing Vertex Buffers

To draw the triangle we will use a single vertex buffer, consisting of X,Y,Z single precision floating point position, and a float RGBA vertex color.

The vertex buffer data is simply a constant array defined within `VkSample::InitVertexBuffers()`, which will be copied into the correct device memory later.

We use the `vkCreateBuffer` function to create buffers with parameters specified in `VkBufferCreateInfo` structures. The structure provides the size of the buffer to create – in this example the size of our constant array, and also the usage. This is our vertex buffer, so we specify `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`.

As with our depth buffer images, we query the memory requirements for the created buffer handle, using `vkGetBufferMemoryRequirements` and `vkGetMemoryTypeFromProperties`. An important difference from the depth buffers is that we provide a requirement on the memory type with `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`. This is because we want to modify the data in this buffer from the CPU side, in our sample code.

After allocating the memory, we use `vkMapMemory` which provides us with a CPU-accessible pointer to the buffer data. We use this pointer to copy in our vertex data with `memcpy`, before calling `vkUnmapMemory`. We bind the memory to our buffer object before we can use it in graphics operations.

`vkMapMemory/vkUnmapMemory` are very powerful, allowing us to dynamically update resources such as uniforms and buffers during rendering, but the device cannot be using the memory at the same time. We do not need to synchronize the action in this instance, as we have not used the buffer yet.

After our vertex buffer is created, we need to create structures which describe it to any pipeline that wishes to use it as input. The `VkPipelineVertexInputStateCreateInfo` structure performs this, allowing definitions of vertex bindings, the input rates and stride per vertex.

Attributes are also defined. In this example we have two attributes: the vertex position, and the vertex color. We fill two `VkVertexInputAttributeDescription` structures, defining the data format and offset per-vertex. The locations in these descriptions associate with the location definitions in GLSL vertex shaders. Once those structures are complete, we have a vertex buffer ready for rendering.

Pipeline and Descriptor Layouts

`VkSample::InitLayouts()` creates our Descriptor Set Layout and our Pipeline Layout from it. In this simple example, the layouts are empty – but they are very powerful and define how the uniforms and other data is laid out and bound.

Creating the Render Pass

The renderpass defines the attachments to the framebuffer object that gets used in the pipeline. We have two attachments, the color buffer (our swapchain image), and the depth buffer.

The operations and layouts are set to defaults for this type of attachment, with the reference layout for the swapchain image set to `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, with the depth buffer set to `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`.

RenderPass handles are defined by attachments and subpasses. In `VkSample::InitRenderPass()`, we have simply one subpass, calling `vkCreateRenderPass` with the fully populated Info structures results in our `mRenderPass` handle.

Initializing pipeline state

The VkPipeline object contains all major state for rendering. There are Info structures for:

- Vertex Input state (and we populated this when creating our vertex buffer)
- Assembly state (defining topology as `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`).
- Rasterization state (cull modes, polygon mode, winding)
- Color Blending state (blend enables, color masks)
- Viewport and Scissor states
- DepthStencil state (depth testing definitions)
- MultiSample state
- Shader stage state

There is a lot of code in this function, but most of it can be traced to traditional graphical operations. The shader stage state for our example is a single vertex and single fragment shader. We create a `VkShaderModule` handle via `CreateShaderModule`, which calls `vkCreateShaderModule` with Info structures defining `pCode` and `codeSize` as specified to the SPIR-V shader binaries contained in shader.h.

Once a `VkPipeline` handle is created, `VkShaderModule` handles can be destroyed if they are no longer required. In our case, we destroy them via `vkDestroyShaderModule` now.

Framebuffer initialization

The framebuffer objects reference the renderpass and allow the references defined in that renderpass to now attach to image views. The views in this example are the color view, which is our swapchain image, and the depth buffer created manually earlier.

In `VkSample::InitFrameBuffers()`, we create framebuffer objects and store them in `mFrameBuffers`.

We see here that `Info` structures can be reused if we need to create multiple objects, with only few Info members changing each time.

At this point, we now have enough state initialized to draw to the surface.

Synchronization primitives

In `VkSample::InitSync()` we create two `VkSemaphore` objects. Synchronization in Vulkan can be done in many different ways, but for this sample we follow a simple chain.

The chain will be explained more in the 'per-frame actions' of this guide.

Building the command buffers

The command buffers contain the 'commands' for a device to execute. Command buffers allow developers to 'record' actions to them, and then queue them to the device for execution multiple times. In this example, we pre-record our drawing and presentation commands in `VkSample::BuildCmdBuffer()` . We have a separate command buffer for each swapchain image, but they will contain the same set of actions.

To set the command buffer into the recording state, we must call `vkBeginCommandBuffer` passing in relative `VkCommandBufferBeginInfo` details. There is not much required to be set in the Info structure.

When recording command buffers that are run multiple times, it's important to remember how buffers and state will be setup prior to the buffer being queued for execution. At this point, our back-buffer swapchain image will still be in the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, after being presented to the Android surface during a previous frame. To use it, we must change this to `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. We do this with an Image Memory Barrier, via the `vkCmdPipelineBarrier` call and `VkImageMemoryBarrier` structure. The pipeline stage where this occurs concerns color attachment, and `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` is specified. Note, that while the Vulkan specification requires these barriers, current Adreno drivers treat this operation as a no-op as the hardware does not require them.

We now perform several actions, putting commands into our command buffer:

1. Begin the render pass, passing the framebuffer for this swapchain image, and relevant clear values.
2. Bind our pipeline, at the graphics bind point.
3. Bind our vertex buffer
4. Issue a draw command of 3 vertices, making our triangle. The pipeline defines the topology in our vertex buffer as `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`.
5. End the render pass.

That is our triangle drawn, with the pipeline specifying our fragment and vertex shaders, viewport and depth state.

Now we need to do the opposite of previously: transition our swapchain image from VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL to VK_IMAGE_LAYOUT_PRESENT_SRC_KHR for presentation to the Android surface. This barrier specifies the destination pipeline stage as VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT at the end of the graphics pipeline.

We finish by calling vkEndCommandBuffer, taking the command buffer out of record mode. It can now be submitted to the device queue.

Set the starting back buffer of the swapchain

The last initialization step in the sample is to setup the swapchain so there is an image acquired and ready for rendering. This is explained in the 'per-frame actions' section of this guide, as it is called per-frame.

Initialization completed

The sample has now initialized all relevant Vulkan state and objects, ready for rendering on a per-frame basis.

Per-frame actions

DrawFrame()

The drawframe function assumes a swapchain image has been acquired, ready for use as back-buffer. This is why we call VkSample::SetNextBackBuffer() at the end of initialization.

All that the draw frame function does is queue our single command buffer. The VkSubmitInfo structure provides the pointer to the command buffer we want to queue – the handle corresponding to the current swapchain image index. The VkSubmitInfo structure has fields for waiting on a semaphore, and also signaling one on completion. We wait on the mBackBufferSemaphore, which is signaled when our swapchain image is ready for use. We will signal the mRenderCompleteSemaphore to indicate to PresentBackBuffer() that rendering has completed.

After submitting, we call PresentBackBuffer() to update the surface. There is also some basic FPS calculation.

PresentBackBuffer()

We call vkQueuePresentKHR with relevant VkPresentInfoKHR structure defining the swapchain image index we used, and also instructing it to wait on the mRenderCompleteSemaphore before presenting. SetNextBackBuffer is called before returning from the function, to begin the process of making the next swapchain image ready for use.

SetNextBackBuffer()

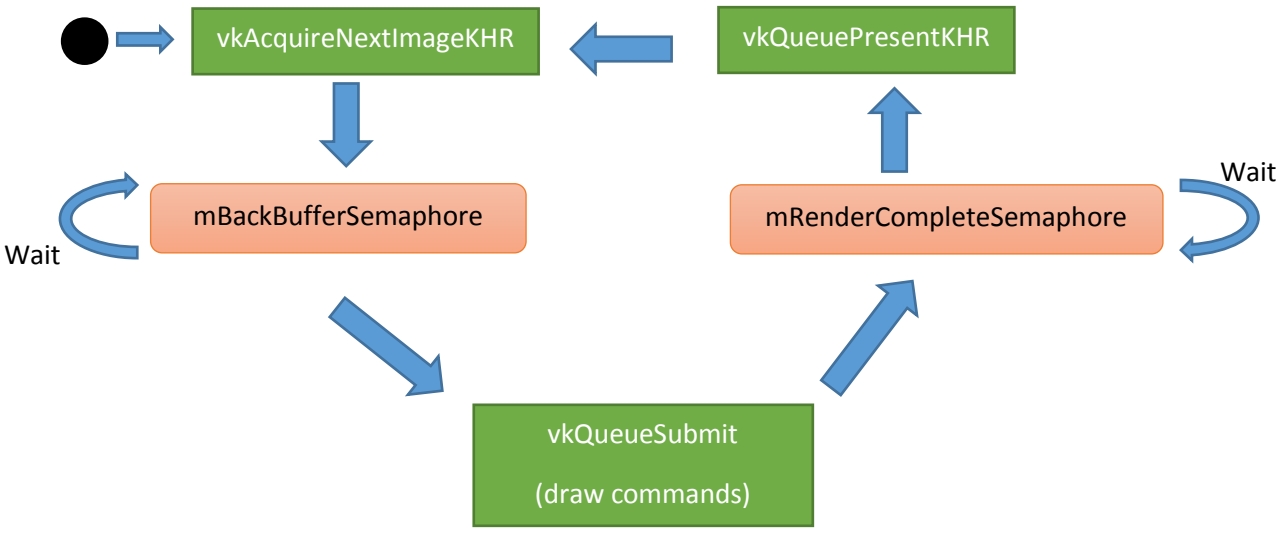
This function requests that the next image in the swapchain be acquired. This is done by calling `vkAcquireNextImageKHR`, which will immediately return with the next swapchain image index to use, and also provide a semaphore which will signal when that image is ready internally for use.

In this example, we then queue a wait on that semaphore, to ensure it is ready before returning from the function. In other examples, it may be wise to use this time for other operations, but in this sample it shows the synchronization points explicitly.

Synchronization

The flow of a frame follows three main functions;

1. Acquire the swapchain image index to use next
2. Submit draw command buffers, using the relevant swapchain image
3. Queue the swapchain image for presentation



The essential synchronization primitives are two semaphores: one for ensuring the back buffer swapchain image is ready, and the other for signaling rendering as complete.

The back buffer semaphore is passed in `pWaitSemaphores` on the `SubmitInfo` of the draw command buffer submit. The render completion semaphore is used on the `SubmitInfo` of the `vkQueueSubmit` draw commands as a `pSignalSemaphore`, meaning it will be signaled on completion of the command buffer execution. This semaphore is then passed as a wait semaphore to the `vkQueuePresentKHR` call. The CPU will progress once the GPU has consumed the command buffers.

Teardown

There is a single function to perform destruction of all Vulkan handles and associated memory, both device allocations and CPU allocation via standard new.

There is nothing significant, apart from ordering needs to be the mirror of initialization, as with most APIs.

Shader Compilation

The vertex and fragment shaders for a sample can be found within the `shaders/` directory of the sample. The shaders are compiled to SPIR-V using glslang.

To offline compile glslangValidator can be used, like follows:

```
> glslangValidator.exe -V –x shader.frag -o frag_shader.spv
```

In the triangle sample, `frag_shader.spv` is then embedded into `shader.h` and the byte size constant is updated to match the size of the shader. In other samples, the compiled SPIR-V files are loaded directly as asset buffers.

In further examples, the Android Studio project has build steps for automatically building SPIR-V binaries for shaders. For this to build successfully, glslangValidator needs to be on the system path.