API without Secrets: Introduction to Vulkan* Part 3

Table of Contents

Tutorial 3: First Triangle – Graphics Pipeline and Drawing

In this tutorial we will finally draw something on the screen. One single triangle should be just fine for our first Vulkan-generated "image."

The graphics pipeline and drawing in general require lots of preparations in Vulkan (in the form of filling many structures with even more different fields). There are potentially many places where we can make mistakes, and in Vulkan, even simple mistakes may lead to the application not working as expected, displaying just a blank screen, and leaving us wondering what went wrong. In such situations validation layers can help us a lot. But I didn't want to dive into too many different aspects and the specifics of the Vulkan API. So I prepared the code to be as small and as simple as possible.

This led me to create an application that is working properly and displays a simple triangle the way I expected, but it also uses mechanics that are not recommended, not flexible, and also probably not too efficient (though correct). I don't want to teach solutions that aren't recommended, but here it simplifies the tutorial quite considerably and allows us to focus only on the minimal required set of API usage. I will point out the "disputable" functionality as soon as we get to it. And in the next tutorial, I will show the recommended way of drawing triangles.

To draw our first simple triangle, we need to create a render pass, a framebuffer, and a graphics pipeline. Command buffers are of course also needed, but we already know something about them. We will create simple GLSL shaders and compile them into Khronos's SPIR*-V language—the only (at this time) form of shaders that Vulkan (officially) understands.

If nothing displays on your computer's screen, try to simplify the code as much as possible or even go back to the second tutorial. Check whether command buffer that just clears image behaves as expected, and that the color the image was cleared to is properly displayed on the screen. If yes, modify the code and add the parts from this tutorial. Check every return value if it is not VK_SUCCESS. If these ideas don't help, wait for the tutorial about validation layers.

About the Source Code Example

For this and succeeding tutorials, I've changed the sample project. Vulkan preparation phases that were described in the previous tutorials were placed in a "VulkanCommon" class found in separate files (header and source). The class for a given tutorial that is responsible for presenting topics described in a given tutorial, inherits from the "VulkanCommon" class and has access to some (required) Vulkan variables like device or swap chain. This way I can reuse Vulkan creation code and prepare smaller classes focusing only on the presented topics. The code from the earlier chapters works properly so it should also be easier to find potential mistakes.

I've also added a separate set of files for some utility functions. Here we will be reading SPIR-V shaders from binary files, so I've added a function for checking loading contents of a binary file. It can be found in Tools.cpp and Tools.h files.

Creating a Render Pass

To draw anything on the screen, we need a graphics pipeline. But creating it now will require pointers to other structures, which will probably also need pointers to yet other structures. So we'll start with a render pass.

What is a render pass? A general picture can give us a "logical" render pass that may be found in many known rendering techniques like deferred shading. This technique consists of many subpasses. The first subpass draws the geometry with shaders that fill the G-Buffer: store diffuse color in one texture, normal vectors in another, shininess in another, depth (position) in yet another. Next for each light source, drawing is performed that reads some of the data (normal vectors, shininess, depth/position), calculates lighting and stores it in another texture. Final pass aggregates lighting data with diffuse color. This is a (very rough) explanation of deferred shading but describes the render pass—a set of data required to perform some drawing operations: storing data in textures and reading data from other textures.

In Vulkan, a render pass represents (or describes) a set of framebuffer attachments (images) required for drawing operations and a collection of subpasses that drawing operations will be ordered into. It is a construct that collects all color, depth and stencil attachments and operations modifying them in such a way that driver does not have to deduce this information by itself what may give substantial optimization opportunities on some GPUs. A subpass consists of drawing operations that use (more or less) the same attachments. Each of these drawing operations may read from some input attachments and render data into some other (color, depth, stencil) attachments. A render pass also describes the dependencies between these attachments: in one subpass we perform rendering into the texture, but in another this texture will be used as a source of data (that is, it will be sampled from). All this data help the graphics hardware optimize drawing operations.

To create a render pass in Vulkan, we call the **vkCreateRenderPass()** function, which requires a pointer to a structure describing all the attachments involved in rendering and all the subpasses forming the render pass. As usual, the more attachments and subpasses we use, the more array elements containing properly filed structures we need. In our simple example, we will be drawing only into a single texture (color attachment) with just a single subpass.

Render Pass Attachment Description

```
   VkAttachmentDescription attachment_descriptions[] = {
     {
       0,                                          // VkAttachmentDescriptionFlags
flags
       GetSwapChain().Format,                      // VkFormat
format
       VK_SAMPLE_COUNT_1_BIT,                      // VkSampleCountFlagBits
samples
       VK_ATTACHMENT_LOAD_OP_CLEAR,                // VkAttachmentLoadOp
loadOp
       VK_ATTACHMENT_STORE_OP_STORE,               // VkAttachmentStoreOp
storeOp
       VK_ATTACHMENT_LOAD_OP_DONT_CARE,            // VkAttachmentLoadOp
stencilLoadOp
       VK_ATTACHMENT_STORE_OP_DONT_CARE,           // VkAttachmentStoreOp
stencilStoreOp
       VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout
initialLayout;
       VK_IMAGE_LAYOUT_PRESENT_SRC_KHR             // VkImageLayout
finalLayout
     }
   };
```

1. Tutorial03.cpp, function CreateRenderPass()

To create a render pass, first we prepare an array with elements describing each attachment, regardless of the type of attachment and how it will be used inside a render pass. Each array element is of type VkAttachmentDescription, which contains the following fields:

- flags – Describes additional properties of an attachment. Currently, only an aliasing flag is available, which informs the driver that the attachment shares the same physical memory with another attachment; it is not the case here so we set this parameter to zero.
- format – Format of an image used for the attachment; here we are rendering directly into a swap chain so we need to take its format.
- samples – Number of samples of the image; we are not using any multisampling here so we just use one sample.
- loadOp – Specifies what to do with the image's contents at the beginning of a render pass, whether we want them to be cleared, preserved, or we don't care about them (as we will overwrite them all). Here we want to clear the image to the specified value. This parameter also refers to depth part of depth/stencil images.
- storeOp – Informs the driver what to do with the image's contents after the render pass (after a subpass in which the image was used for the last time). Here we want the contents of the image to be preserved after the render pass as we intend to display them on screen. This parameter also refers to the depth part of depth/stencil images.

- stencilLoadOp – The same as loadOp but for the stencil part of depth/stencil images; for color attachments it is ignored.
- stencilStoreOp – The same as storeOp but for the stencil part of depth/stencil images; for color attachments this parameter is ignored.
- initialLayout – The layout the given attachment will have when the render pass starts (what the layout image is provided with by the application).
- finalLayout – The layout the driver will automatically transition the given image into at the end of a render pass.

Some additional information is required for load and store operations and initial and final layouts.

Load op refers to the attachment's contents at the beginning of a render pass. This operation describes what the graphics hardware should do with the attachment: clear it, operate on its existing contents (leave its contents untouched), or it shouldn't matter about the contents because the application intends to overwrite them. This gives the hardware an opportunity to optimize memory operations. For example, if we intend to overwrite all of the contents, the hardware won't bother with them and, if it is faster, may allocate totally new memory for the attachment.

Store op, as the name suggests, is used at the end of a render pass and informs the hardware whether we want to use the contents of the attachment after the render pass or if we don't care about it and the contents may be discarded. In some scenarios (when contents are discarded) this creates the ability for the hardware to create an image in temporary, fast memory as the image will "live" only during the render pass and the implementations may save some memory bandwidth avoiding writing back data that is not needed anymore.

When an attachment has a depth format (and potentially also a stencil component) load and store ops refer only to the depth component. If a stencil is present, stencil values are treated the way stencil load and store ops describe. For color attachments, stencil ops are not relevant.

Layout, as I described in the swap chain tutorial, is an internal memory arrangement of an image. Image data may be organized in such a way that neighboring "image pixels" are also neighbors in memory, which can increase cache hits (faster memory reading) when image is used as a source of data (that is, during texture sampling). But caching is not necessary when the image is used as a target for drawing operations, and the memory for that image may be organized in a totally different way. Image may have linear layout (which gives the CPU ability to read or populate image's memory contents) or optimal layout (which is optimized for performance but is also hardware/vendor dependent). So some hardware may have special memory organization for some types of operations; other hardware may be operations-agnostic. Some of the memory layouts may be better suited for some intended image "usages." Or from the other side, some usages may require specific memory layouts. There is also a general layout that is compatible with all types of operations. But from the performance point of view, it is always best to set the layout appropriate for an intended image usage and it is application's responsibility to inform the driver about transitions.

Image layouts may be changed using image memory barriers. We did this in the swap chain tutorial when we first changed the layout from the presentation source (image was used by the presentation engine) to transfer destination (we wanted to clear the image with a given color). But layouts, apart from image memory barriers, may also be changed automatically by the hardware inside a render pass. If we specify a different initial layout, subpass layouts (described later), and final layout, the hardware does the transition automatically at the appropriate time.

Initial layout informs the hardware about the layout the application "provides" (or "leaves") the given attachment with. This is the layout the image starts with at the beginning of a render pass (in our example we acquire the image from the presentation engine so the image has a "presentation source" layout set). Each subpass of a render pass may use a different layout, and the transition will be done automatically by the hardware between subpasses. The final layout is the layout the given attachment will be transitioned into (automatically) at the end of a render pass (after a render pass is finished).

This information must be prepared for each attachment that will be used in a render pass. When graphics hardware receives this information a priori, it may optimize operations and memory during the render pass to achieve the best possible performance.

Subpass Description

```
   VkAttachmentReference color_attachment_references[] = {
     {
       0,                                          // uint32_t
attachment
       VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL    // VkImageLayout
layout
     }
   };

   VkSubpassDescription subpass_descriptions[] = {
     {
       0,                                          // VkSubpassDescriptionFlags
flags
       VK_PIPELINE_BIND_POINT_GRAPHICS,            // VkPipelineBindPoint
pipelineBindPoint
       0,                                          // uint32_t
inputAttachmentCount
       nullptr,                                    // const VkAttachmentReference
*pInputAttachments
       1,                                          // uint32_t
colorAttachmentCount
       color_attachment_references,                // const VkAttachmentReference
*pColorAttachments
       nullptr,                                    // const VkAttachmentReference
*pResolveAttachments
       nullptr,                                    // const VkAttachmentReference
*pDepthStencilAttachment
       0,                                          // uint32_t
preserveAttachmentCount
       nullptr                                     // const uint32_t*
pPreserveAttachments
     }
   };
```

2. Tutorial03.cpp, function CreateRenderPass()

Next we specify the description of each subpass our render pass will include. This is done using VkSubpassDescription structure, which contains the following fields:

- flags – Parameter reserved for future use.
- pipelineBindPoint – Type of pipeline in which this subpass will be used (graphics or compute). Our example, of course, uses a graphics pipeline.
- inputAttachmentCount – Number of elements in the pInputAttachments array.
- pInputAttachments – Array with elements describing which attachments are used as an input and can be read from inside shaders. We are not using any input attachments here so we set this value to 0.
- colorAttachmentCount – Number of elements in pColorAttachments and pResolveAttachments arrays.
- pColorAttachments – Array describing (pointing to) attachments which will be used as color render targets (that image will be rendered into).
- pResolveAttachments – Array closely connected with color attachments. Each element from this array corresponds to an element from a color attachments array; any such color attachment will be resolved to a given resolve attachment (if a resolve attachment at the same index is not null or if the whole pointer is not null). This is optional and can be set to null.

- pDepthStencilAttachment – Description of an attachment that will be used for depth (and/or stencil) data. We don't use depth information here so we can set it to null.
- preserveAttachmentCount – Number of elements in pPreserveAttachments array.
- pPreserveAttachments – Array describing attachments that should be preserved. When we have multiple subpasses not all of them will use all attachments. If a subpass doesn't use some of the attachments but we need their contents in the later subpasses, we must specify these attachments here.

The pInputAttachments, pColorAttachments, pResolveAttachments, pPreserveAttachments, and pDepthStencilAttachment parameters are all of type VkAttachmentReference. This structure contains only these two fields:

- attachment – Index into an attachment_descriptions array of VkRenderPassCreateInfo.
- layout – Requested (required) layout the attachment will use during a given subpass. The hardware will perform an automatic transition into a provided layout just before a given subpass.

This structure contains references (indices) into the attachment_descriptions array of VkRenderPassCreateInfo. When we create a render pass we must provide a description of all attachments used during a render pass. We've prepared this description earlier in "Render pass attachment description" when we created the attachment_descriptions array. Right now it contains only one element, but in more advanced scenarios there will be multiple attachments. So this "general" collection of all render pass attachments is used as a reference point. In the subpass description, when we fill pColorAttachments or pDepthStencilAttachment members, we provide indices into this very "general" collection, like this: take the first attachment from all render pass attachments and use it as a color attachment. The second attachment from that array will be used for depth data.

There is a separation between a whole render pass and its subpasses because each subpass may use multiple attachments in a different way, that is, in one subpass we are rendering into one color attachment but in the next subpass we are reading from this attachment. In this way, we can prepare a list of all attachments used in the whole render pass, and at the same time we can specify how each attachment will be used in each subpass. And as each subpass may use a given attachment in its own way, we must also specify each image's layout for each subpass.

So before we can specify a description of all subpasses (an array with elements of type VkSubpassDescription) we must create references for each attachment used in each subpass. And this is what the color_attachment_references variable was created for. When I write a tutorial for rendering into a texture, this usage will be more apparent.

Render Pass Creation

We now have all the data we need to create a render pass.

```
    VkRenderPassCreateInfo render_pass_create_info = {
      VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO,    // VkStructureType
sType
      nullptr,                                      // const void
*pNext
      0,                                            // VkRenderPassCreateFlags
flags
      1,                                            // uint32_t
attachmentCount
      attachment_descriptions,                      // const VkAttachmentDescription
*pAttachments
      1,                                            // uint32_t
subpassCount
      subpass_descriptions,                         // const VkSubpassDescription
*pSubpasses
      0,                                            // uint32_t
dependencyCount
      nullptr                                       // const VkSubpassDependency
*pDependencies
```

```
  };

  if( vkCreateRenderPass( GetDevice(), &render_pass_create_info, nullptr,
&Vulkan.RenderPass ) != VK_SUCCESS ) {
    printf( "Could not create render pass!\n" );
    return false;
  }

  return true;
```

3. Tutorial03.cpp, function CreateRenderPass()

We start by filling the VkRenderPassCreateInfo structure, which contains the following fields:

- sType – Type of structure (VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO here).
- pNext – Parameter not currently used.
- flags – Parameter reserved for future use.
- attachmentCount – Number of all different attachments (elements in pAttachments array) used during whole render pass (here just one).
- pAttachments – Array specifying all attachments used in a render pass.
- subpassCount – Number of subpasses a render pass consists of (and number of elements in pSubpasses array – just one in our simple example).
- pSubpasses – Array with descriptions of all subpasses.
- dependencyCount – Number of elements in pDependencies array (zero here).
- pDependencies – Array describing dependencies between pairs of subpasses. We don't have many subpasses so we don't have dependencies here (set it to null here).

Dependencies describe what parts of the graphics pipeline use memory resource in what way. Each subpass may use resources in a different way. Layouts of each resource may not solely define how they use resources. Some subpasses may render into images or store data through shader images. Other may not use images at all or may read from them at different pipeline stages (that is, vertex or fragment).

This information helps the driver optimize automatic layout transitions and, more generally, optimize barriers between subpasses. When we are writing into images only in a vertex shader there is no point waiting until the fragment shader executes (of course in terms of used images). After all the vertex operations are done, images may immediately change their layouts and memory access type, and even some parts of graphics hardware may start executing the next operations (that are referencing or reading the given images) without the need to wait for the rest of the commands from the given subpass to finish. For now, just remember that dependencies are important from a performance point of view.

So now that we have prepared all the information required to create a render pass, we can safely call the **vkCreateRenderPass()** function.

Creating a Framebuffer

We have created a render pass. It describes all attachments and all subpasses used during the render pass. But this description is quite abstract. We have specified formats of all attachments (just one image in this example) and described how attachments will be used by each subpass (also just one here). But we didn't specify WHAT attachments we will be using or, in other words, what images will be used as these attachments. This is done through a framebuffer.

A framebuffer describes specific images that the render pass operates on. In OpenGL*, a framebuffer is a set of textures (attachments) we are rendering into. In Vulkan, this term is much broader. It describes all the textures (attachments) used during the render pass, not only the images we are rendering into (color and depth/stencil attachments) but also images used as a source of data (input attachments).

This separation of render pass and framebuffer gives us some additional flexibility. We can use the given render pass with different framebuffers and a given framebuffer with different render passes, if they are compatible, meaning that they operate in a similar fashion on images of similar types and usages.

Before we can create a framebuffer, we must create image views for each image used as a framebuffer and render pass attachment. In Vulkan, not only in the case of framebuffers, but in general, we don't operate on images themselves. Images are not accessed directly. For this purpose, image views are used. Image views represent images, they "wrap" images and provide additional (meta)data for them.

Creating Image Views

In this simple application, we want to render directly into swap chain images. We have created a swap chain with multiple images, so we must create an image view for each of them.

```
    const std::vector<VkImage> &swap_chain_images = GetSwapChain().Images;
    Vulkan.FramebufferObjects.resize( swap_chain_images.size() );

    for( size_t i = 0; i < swap_chain_images.size(); ++i ) {
      VkImageViewCreateInfo image_view_create_info = {
        VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO,   // VkStructureType
sType
        nullptr,                                    // const void
*pNext
        0,                                          // VkImageViewCreateFlags
flags
        swap_chain_images[i],                       // VkImage
image
        VK_IMAGE_VIEW_TYPE_2D,                      // VkImageViewType
viewType
        GetSwapChain().Format,                      // VkFormat
format
        {                                           // VkComponentMapping
components
          VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle           r
          VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle           g
          VK_COMPONENT_SWIZZLE_IDENTITY,              // VkComponentSwizzle           b
          VK_COMPONENT_SWIZZLE_IDENTITY               // VkComponentSwizzle           a
        },
        {                                           // VkImageSubresourceRange
subresourceRange
          VK_IMAGE_ASPECT_COLOR_BIT,                  // VkImageAspectFlags
aspectMask
          0,                                          // uint32_t
baseMipLevel
          1,                                          // uint32_t
levelCount
          0,                                          // uint32_t
baseArrayLayer
          1                                           // uint32_t
layerCount
        }
      };

      if( vkCreateImageView( GetDevice(), &image_view_create_info, nullptr,
&Vulkan.FramebufferObjects[i].ImageView ) != VK_SUCCESS ) {
        printf( "Could not create image view for framebuffer!\n" );
        return false;
      }
```

4. Tutorial03.cpp, function CreateFramebuffers()

To create an image view, we must first create a variable of type VkImageViewCreateInfo. It contains the following fields:

- sType – Structure type, in this case it should be set to VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO.
- pNext – Parameter typically set to null.
- flags – Parameter reserved for future use.
- image – Handle to an image for which view will be created.
- viewType – Type of view we want to create. View type must be compatible with an image it is created for. (that is, we can create a 2D view for an image that has multiple array layers or we can create a CUBE view for a 2D image with six layers).
- format – Format of an image view; it must be compatible with the image's format but may not be the same format (that is, it may be a different format but with the same number of bits per pixel).
- components – Mapping of an image components into a vector returned in the shader by texturing operations. This applies only to read operations (sampling), but since we are using an image as a color attachment (we are rendering into an image) we must set the so-called identity mapping (R component into R, G -> G, and so on) or just use "identity" value (VK_COMPONENT_SWIZZLE_IDENTITY).
- subresourceRange – Describes the set of mipmap levels and array layers that will be accessible to a view. If our image is mipmapped, we may specify the specific mipmap level we want to render to (and in case of render targets we must specify exactly one mipmap level of one array layer).

As you can see here, we acquire handles to all swap chain images, and we are referencing them inside a loop. This way we fill the structure required for image view creation, which we pass to a **vkCreateImageView()** function. We do this for each image that was created along with a swap chain.

Specifying Framebuffer Parameters

Now we can create a framebuffer. To do this we call the **vkCreateFramebuffer()** function.

```
     VkFramebufferCreateInfo framebuffer_create_info = {
       VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO,  // VkStructureType
sType
       nullptr,                                    // const void
*pNext
       0,                                          // VkFramebufferCreateFlags
flags
       Vulkan.RenderPass,                          // VkRenderPass
renderPass
       1,                                          // uint32_t
attachmentCount
       &Vulkan.FramebufferObjects[i].ImageView,    // const VkImageView
*pAttachments
       300,                                        // uint32_t
width
       300,                                        // uint32_t
height
       1                                           // uint32_t
layers
     };

     if( vkCreateFramebuffer( GetDevice(), &framebuffer_create_info, nullptr,
&Vulkan.FramebufferObjects[i].Handle ) != VK_SUCCESS ) {
       printf( "Could not create a framebuffer!\n" );
       return false;
     }
   }
   return true;
```

5. Tutorial03.cpp, function CreateFramebuffers()

vkCreateFramebuffer() function requires us to provide a pointer to a variable of type VkFramebufferCreateInfo so we must first prepare it. It contains the following fields:

- sType – Structure type set to VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO in this situation.
- pNext – Parameter most of the time set to null.
- flags – Parameter reserved for future use.
- renderPass – Render pass this framebuffer will be compatible with.
- attachmentCount – Number of attachments in a framebuffer (elements in pAttachments array).
- pAttachments – Array of image views representing all attachments used in a framebuffer and render pass. Each element in this array (each image view) corresponds to each attachment in a render pass.
- width – Width of a framebuffer.
- height – Height of a framebuffer.
- layers – Number of layers in a framebuffer (OpenGL's layered rendering with geometry shaders, which could specify the layer into which fragments rasterized from a given polygon will be rendered).

The framebuffer specifies what images are used as attachments on which the render pass operates. We can say that it translates image (image view) into a given attachment. The number of images specified for a framebuffer must be the same as the number of attachments in a render pass for which we are creating a framebuffer. Also, each pAttachments array's element corresponds directly to an attachment in a render pass description structure. Render pass and framebuffer are closely connected, and that's why we also must specify a render pass during framebuffer creation. But we may use a framebuffer not only with the specified render pass but also with all render passes that are compatible with the one specified. Compatible render passes, in general, must have the same number of attachments and corresponding attachments must have the same format and number of samples. But image layouts (initial, final, and for each subpass) may differ and doesn't involve render pass compatibility.

After we have finished creating and filling the VkFramebufferCreateInfo structure, we call the **vkCreateFramebuffer()** function.

The above code executes in a loop. A framebuffer references image views. Here the image view is created for each swap chain image. So for each swap chain image and its view, we are creating a framebuffer. We are doing this in order to simplify the code called in a rendering loop. In a normal, real-life scenario we wouldn't (probably) create a framebuffer for each swap chain image. I assume that a better solution would be to render into a single image (texture) and after that use command buffers that would copy rendering results from that image into a given swap chain image. This way we will have only three simple command buffers that are connected with a swap chain. All other rendering commands would be independent of a swap chain, making it easier to maintain.

Creating a Graphics Pipeline

Now we are ready to create a graphics pipeline. A pipeline is a collection of stages that process data one stage after another. In Vulkan there is currently a compute pipeline and a graphics pipeline. The compute pipeline allows us to perform some computational work, such as performing physics calculations for objects in games. The graphics pipeline is used for drawing operations.

In OpenGL there are multiple programmable stages (vertex, tessellation, fragment shaders, and so on) and some fixed function stages (rasterizer, depth test, blending, and so on). In Vulkan, the situation is similar. There are similar (if not identical) stages. But the whole pipeline's state is gathered in one monolithic object. OpenGL allows us to change the state that influences rendering operations anytime we want, we can change parameters for each stage (mostly) independently. We can set up shader programs, depths test, blending, and whatever state we want, and then we can render some objects. Next we can change just some small part of the state and render another object. In Vulkan, such operations can't be done (we say that pipelines are "immutable"). We must prepare the whole state and set up parameters for pipeline stages and group them in a pipeline object. At the beginning this was one of the most startling pieces information for me. I'm not able to change shader program anytime I want? Why?

The easiest and more valid explanation is because of the performance implications of such state changes. Changing just one single state of the whole pipeline may cause graphics hardware to perform many background operations like state and error checking. Different hardware vendors may implement (and usually are implementing) such functionality differently. This may cause applications to perform differently (meaning unpredictably, performance-wise) when executed on different graphics hardware. So the ability to change anything at any time is convenient for developers. But, unfortunately, it is not so convenient for the hardware.

That's why in Vulkan the state of the whole pipeline is to gather in one, single object. All the relevant state and error checking is performed when the pipeline object is created. When there are problems (like different parts of pipeline are set up in an incompatible way) pipeline object creation fails. But we know that upfront. The driver doesn't have to worry for us and do whatever it can to properly use such a broken pipeline. It can immediately tell us about the problem. But during real usage, in performance-critical parts of the application, everything is already set up correctly and can be used as is.

The downside of this methodology is that we have to create multiple pipeline objects, multiple variations of pipeline objects when we are drawing many objects in a different way (some opaque, some semi-transparent, some with depth test enabled, others without). Unfortunately, even different shaders make us create different pipeline objects. If we want to draw objects using different shaders, we also have to create multiple pipeline objects, one for each combination of shader programs. Shaders are also connected with the whole pipeline state. They use different resources (like textures and buffers), render into different color attachments, and read from different attachments (possibly that were rendered into before). These connections must also be initialized, prepared, and set up correctly. We know what we want to do, the driver does not. So it is better and far more logical that we do it, not the driver. In general this approach makes sense.

To begin the pipeline creation process, let's start with shaders.

Creating a Shader Module

Creating a graphics pipeline requires us to prepare lots of data in the form of structures or even arrays of structures. The first such data is a collection of all shader stages and shader programs that will be used during rendering with a given graphics pipeline bound.

In OpenGL, we write shaders in GLSL. They are compiled and then linked into shader programs directly in our application. We can use or stop using a shader program anytime we want in our application.

Vulkan on the other hand accepts only a binary representation of shaders, an intermediate language called SPIR-V. We can't provide GLSL code like we did in OpenGL. But there is an official, separate compiler that can transform shaders written in GLSL into a binary SPIR-V language. To use it, we have to do it offline. After we prepare the SPIR-V assembly we can create a shader module from it. Such modules are then composed into an array of VkPipelineShaderStageCreateInfo structures, which are used, among other parameters, to create graphics pipeline.

Here's the code that creates a shader module from a specified file that contains a binary SPIR-V.

```
    const std::vector<char> code = Tools::GetBinaryFileContents( filename );
    if( code.size() == 0 ) {
      return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>();
    }

    VkShaderModuleCreateInfo shader_module_create_info = {
      VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO,    // VkStructureType
sType
      nullptr,                                        // const void
*pNext
      0,                                              // VkShaderModuleCreateFlags
flags
      code.size(),                                    // size_t
codeSize
```

```
      reinterpret_cast<const uint32_t*>(&code[0])      // const uint32_t
*pCode
    };

    VkShaderModule shader_module;
    if( vkCreateShaderModule( GetDevice(), &shader_module_create_info, nullptr,
&shader_module ) != VK_SUCCESS ) {
      printf( "Could not create shader module from a %s file!\n", filename );
      return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>();
    }

    return Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule>( shader_module,
vkDestroyShaderModule, GetDevice() );
```

6. Tutorial03.cpp, function CreateShaderModule()

First we prepare a VkShaderModuleCreateInfo structure that contains the following fields:

- sType – Type of structure, in this example set to VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO.
- pNext – Pointer not yet used.
- flags – Parameter reserved for future use.
- codeSize – Size in bytes of the code passed in pCode parameter.
- pCode – Pointer to an array with source code (binary SPIR-V assembly).

To acquire the contents of the file, I have prepared a simple utility function **GetBinaryFileContents()** that reads the entire contents of a specified file. It returns the content in a vector of chars.

After we prepare a structure, we can call the **vkCreateShaderModule()** function and check whether everything went fine.

The AutoDeleter<> class from Tools namespace is a helper class that wraps a given Vulkan object handle and takes a function that is called to delete that object. This class is similar to smart pointers, which delete the allocated memory when the object (the smart pointer) goes out of scope. AutoDeleter<> class takes the handle of a given object and deletes it with a provided function when the object of this class's type goes out of scope.

```
    template<class T, class F>
    class AutoDeleter {
    public:
      AutoDeleter() :
        Object( VK_NULL_HANDLE ),
        Deleter( nullptr ),
        Device( VK_NULL_HANDLE ) {
      }

      AutoDeleter( T object, F deleter, VkDevice device ) :
        Object( object ),
        Deleter( deleter ),
        Device( device ) {
      }

      AutoDeleter( AutoDeleter&& other ) {
        *this = std::move( other );
      }

      ~AutoDeleter() {
        if( (Object != VK_NULL_HANDLE) && (Deleter != nullptr) && (Device !=
VK_NULL_HANDLE) ) {
          Deleter( Device, Object, nullptr );
        }
      }
      AutoDeleter& operator=( AutoDeleter&& other ) {
        if( this != &other ) {
          Object = other.Object;
          Deleter = other.Deleter;
          Device = other.Device;
          other.Object = VK_NULL_HANDLE;
        }
        return *this;
      }

      T Get() {
        return Object;
      }

      bool operator !() const {
        return Object == VK_NULL_HANDLE;
      }

    private:
      AutoDeleter( const AutoDeleter& );
      AutoDeleter& operator=( const AutoDeleter& );
      T         Object;
      F         Deleter;
      VkDevice  Device;
    };
```

7. Tools.h, -

Why so much effort for one simple object? Shader modules are one of the objects required to create the graphics pipeline. But after the pipeline is created, we don't need these shader modules anymore. Sometimes it is convenient to keep them as we may need to create additional, similar pipelines. But in this example they may be safely destroyed after we create a graphics pipeline. Shader modules are destroyed by calling the **vkDestroyShaderModule()** function. But in the example, we would need to call this function in many places: inside multiple "ifs" and at the end of the whole function. Because I don't want to remember where I need to call this function and, at the same time, I don't want any memory leaks to occur, I have prepared this simple class just for convenience. Now, I don't have to remember to delete the created shader module because it will be deleted automatically.

Preparing a Description of the Shader Stages

Now that we know how to create and destroy shader modules, we can create data for shader stages compositing our graphics pipeline. As I have written, the data that describes what shader stages should be active when a given graphics pipeline is bound has a form of an array with elements of type VkPipelineShaderStageCreateInfo. Here is the code that creates shader modules and prepares such an array:

```
   Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule> vertex_shader_module =
CreateShaderModule( "Data03/vert.spv" );
   Tools::AutoDeleter<VkShaderModule, PFN_vkDestroyShaderModule> fragment_shader_module
= CreateShaderModule( "Data03/frag.spv" );

   if( !vertex_shader_module || !fragment_shader_module ) {
     return false;
   }

   std::vector<VkPipelineShaderStageCreateInfo> shader_stage_create_infos = {
     // Vertex shader
     {
       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,        // VkStructureType
sType
```

```
       nullptr,                                                      // const void
*pNext
       0,                                                            //
VkPipelineShaderStageCreateFlags                flags
       VK_SHADER_STAGE_VERTEX_BIT,                                   //
VkShaderStageFlagBits                           stage
       vertex_shader_module.Get(),                                   // VkShaderModule
module
       "main",                                                       // const char
*pName
       nullptr                                                       // const
VkSpecializationInfo                    *pSpecializationInfo
     },
     // Fragment shader
     {
       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,         // VkStructureType
sType
       nullptr,                                                      // const void
*pNext
       0,                                                            //
VkPipelineShaderStageCreateFlags                flags
       VK_SHADER_STAGE_FRAGMENT_BIT,                                 //
VkShaderStageFlagBits                           stage
       fragment_shader_module.Get(),                                 // VkShaderModule
module
       "main",                                                       // const char
*pName
       nullptr                                                       // const
VkSpecializationInfo                    *pSpecializationInfo
     }
   };
```

8. Tutorial03.cpp, function CreatePipeline()

At the beginning we are creating two shader modules for vertex and fragment stages. They are created with the function presented earlier. When any error occurs and we return from the **CreatePipeline()** function, any created module is deleted automatically by a wrapper class with a provided deleter function.

The code for the shader modules is read from files that contain the binary SPIR-V assembly. These files are generated with an application called “glslangValidator”. This is a tool distributed officially with the Vulkan SDK and is designed to validate GLSL shaders. But “glslangValidator” also has the capability to compile or rather transform GLSL shaders into SPIR-V binary files. A full explanation of the command line for its usage can be found at the official SDK site. I’ve used the following commands to generate SPIR-V shaders for this tutorial:

```
glslangValidator.exe -V -H shader.vert > vert.spv.txt

glslangValidator.exe -V -H shader.frag > frag.spv.txt
```

“glslangValidator” takes a specified file and generates SPIR-V file from it. The type of shader stage is automatically detected by the input file’s extension (“.vert” for vertex shaders, “.geom” for geometry shaders, and so on). The name of the generated file can be specified, but by default it takes a form “<stage>.spv”. So in our example “vert.spv” and “frag.spv” files will be generated.

SPIR-V files have a binary format so it may be hard to read and analyze them—but not impossible. When the “-H” option is used, “glslangValidator” outputs SPIR-V in a form that can be more easily read. This form is printed on standard output and that’s why I’m using the “> *.spv.txt” redirection operator.

Here are the contents of a “shader.vert” file from which SPIR-V assembly was generated for the vertex stage:

```
 #version 400
```

```
void main() {
    vec2 pos[3] = vec2[3]( vec2(-0.7, 0.7), vec2(0.7, 0.7), vec2(0.0, -0.7) );
    gl_Position = vec4( pos[gl_VertexIndex], 0.0, 1.0 );
}
```

9. shader.vert, -

As you can see I have hardcoded the positions of all vertices used to render the triangle. They are indexed using the Vulkan-specific "gl_VertexIndex" built-in variable. In the simplest scenario, when using non-indexed drawing commands (which takes place here) this value starts from the value of the "firstVertex" parameter of a drawing command (zero in the provided example).

This is the disputable part I wrote about earlier—this approach is acceptable and valid but not quite convenient to maintain and also allows us to skip some of the "structure filling" needed to create the graphics pipeline. I've chosen it in order to shorten and simplify this tutorial as much as possible. In the next tutorial, I will present a more typical way of drawing any number of vertices, similar to using vertex arrays and indices in OpenGL.

Below is the source code of a fragment shader from the "shader.frag" file that was used to generate the SPIRV-V assembly for the fragment stage:

```
#version 400

layout(location = 0) out vec4 out_Color;

void main() {
  out_Color = vec4( 0.0, 0.4, 1.0, 1.0 );
}
```

10. shader.frag, -

In Vulkan's shaders (when transforming from GLSL to SPIR-V) layout qualifiers are required. Here we specify to what output (color) attachment we want to store the color values generated by the fragment shader. Because we are using only one attachment, we must specify the first available location (zero).

Now that you know how to prepare shaders for applications using Vulkan, we can move on to the next step. After we have created two shader modules, we check whether these operations succeeded. If they did we can start preparing a description of all shader stages that will constitute our graphics pipeline.

For each enabled shader stage we need to prepare an instance of VkPipelineShaderStageCreateInfo structure. Arrays of these structures along with the number of its elements are together used in a graphics pipeline create info structure (provided to the function that creates the graphics pipeline). VkPipelineShaderStageCreateInfo structure has the following fields:

- sType – Type of structure that we are preparing, which in this case must be equal to VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO.
- pNext – Pointer reserved for extensions.
- flags – Parameter reserved for future use.
- stage – Type of shader stage we are describing (like vertex, tessellation control, and so on).
- module – Handle to a shader module that contains the shader for a given stage.
- pName – Name of the entry point of the provided shader.
- pSpecializationInfo – Pointer to a VkSpecializationInfo structure, which we will leave for now and set to null.

When we are creating a graphics pipeline we don't create too many (Vulkan) objects. Most of the data is presented in a form of just such structures.

Preparing Description of a Vertex Input

Now we must provide a description of the input data used for drawing. This is similar to OpenGL's vertex data: attributes, number of components, buffers from which to take data, data's stride, or step rate. In Vulkan this data is of course prepared in a different way, but in general the meaning is the same. Fortunately, because of the fact that vertex data is hardcoded into a vertex shader in this tutorial, we can almost entirely skip this step and fill the VkPipelineVertexInputStateCreateInfo with almost nulls and zeros:

```
   VkPipelineVertexInputStateCreateInfo vertex_input_state_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,    // VkStructureType
sType
     nullptr,                                                      // const void
*pNext
     0,                                                            //
VkPipelineVertexInputStateCreateFlags          flags;
     0,                                                            // uint32_t
vertexBindingDescriptionCount
     nullptr,                                                      // const
VkVertexInputBindingDescription         *pVertexBindingDescriptions
     0,                                                            // uint32_t
vertexAttributeDescriptionCount
     nullptr                                                       // const
VkVertexInputAttributeDescription       *pVertexAttributeDescriptions
   };
```

11. Tutorial03.cpp, function CreatePipeline()

But for clarity here is a description of the members of the VkPipelineVertexInputStateCreateInfo structure:

- sType – Type of structure, VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO here.
- pNext – Pointer to an extension-specific structure.
- flags – Parameter reserved for future use.
- vertexBindingDescriptionCount – Number of elements in the pVertexBindingDescriptions array.
- pVertexBindingDescriptions – Array with elements describing input vertex data (stride and stepping rate).
- vertexAttributeDescriptionCount – Number of elements in the pVertexAttributeDescriptions array.
- pVertexAttributeDescriptions – Array with elements describing vertex attributes (location, format, offset).

Preparing the Description of an Input Assembly

The next step requires us to describe how vertices should be assembled into primitives. As with OpenGL, we must specify what topology we want to use: points, lines, triangles, triangle fan, and so on.

```
   VkPipelineInputAssemblyStateCreateInfo input_assembly_state_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,  // VkStructureType
sType
     nullptr,                                                      // const void
*pNext
     0,                                                            //
VkPipelineInputAssemblyStateCreateFlags        flags
     VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,                          //
VkPrimitiveTopology                            topology
     VK_FALSE                                                      // VkBool32
primitiveRestartEnable
   };
```

12. Tutorial03.cpp, function CreatePipeline()

We do that through the VkPipelineInputAssemblyStateCreateInfo structure, which contains the following members:

- sType – Structure type set here to VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO.

- pNext – Pointer not yet used.
- flags – Parameter reserved for future use.
- topology – Parameter describing how vertices will be organized to form a primitive.
- primitiveRestartEnable – Parameter that tells whether a special index value (when indexed drawing is performed) restarts assembly of a given primitive.

Preparing the Viewport's Description

We have finished dealing with input data. Now we must specify the form of output data, all the part of the graphics pipeline that are connected with fragments, like rasterization, window (viewport), depth tests, and so on. The first set of data we must prepare here is the state of the viewport, which specifies to what part of the image (or texture, or window) we want do draw.

```
   VkViewport viewport = {
     0.0f,                                                      // float
x
     0.0f,                                                      // float
y
     300.0f,                                                    // float
width
     300.0f,                                                    // float
height
     0.0f,                                                      // float
minDepth
     1.0f                                                       // float
maxDepth
   };

   VkRect2D scissor = {
     {                                                          // VkOffset2D
offset
       0,                                                         // int32_t
x
       0                                                          // int32_t
y
     },
     {                                                          // VkExtent2D
extent
       300,                                                       // int32_t
width
       300                                                        // int32_t
height
     }
   };

   VkPipelineViewportStateCreateInfo viewport_state_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO,     // VkStructureType
sType
     nullptr,                                                   // const void
*pNext
     0,                                                         //
VkPipelineViewportStateCreateFlags             flags
     1,                                                         // uint32_t
viewportCount
     &viewport,                                                 // const VkViewport
*pViewports
     1,                                                         // uint32_t
scissorCount
     &scissor                                                   // const VkRect2D
*pScissors
   };
```

In this example, the usage is simple: we just set the viewport coordinates to some predefined values. I don't check the size of the swap chain image we are rendering into. But remember that in real-life production applications this has to be done because the specification states that dimensions of the viewport cannot exceed the dimensions of the attachments that we are rendering into.

To specify the viewport's parameters, we fill the VkViewport structure that contains these fields:

- x – Left side of the viewport.
- y – Upper side of the viewport.
- width – Width of the viewport.
- height – Height of the viewport.
- minDepth – Minimal depth value used for depth calculations.
- maxDepth – Maximal depth value used for depth calculations.

When specifying viewport coordinates, remember that the origin is different than in OpenGL. Here we specify the upper-left corner of the viewport (not the lower left).

Also worth noting is that the minDepth and maxDepth values must be between 0.0 and 1.0 (inclusive) but maxDepth can be lower than minDepth. This will cause the depth to be calculated in "reverse."

Next we must specify the parameters for the scissor test. The scissor test, similarly to OpenGL, restricts generation of fragments only to the specified rectangular area. But in Vulkan, the scissor test is always enabled and can't be turned off. We can just provide the values identical to the ones provided for viewport. Try changing these values and see how it influences the generated image.

The scissor test doesn't have a dedicated structure. To provide data for it we fill the VkRect2D structure which contains two similar structure members. First is VkOffset2D with the following members:

- x – Left side of the rectangular area used for scissor test
- y – Upper side of the scissor area

The second member is of type VkExtent2D, which contains the following fields:

- width – Width of the scissor rectangular area
- height – Height of the scissor area

In general, the meaning of the data we provide for the scissor test through the VkRect2D structure is similar to the data prepared for viewport.

After we have finished preparing data for viewport and the scissor test, we can finally fill the structure that is used during pipeline creation. The structure is called VkPipelineViewportStateCreateInfo, and it contains the following fields:

- sType – Type of the structure, VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO here.
- pNext – Pointer reserved for extensions.
- flags – Parameter reserved for future use.
- viewportCount – Number of elements in the pViewports array.
- pViewports – Array with elements describing parameters of viewports used when the given pipeline is bound.
- scissorCount – Number of elements in the pScissors array.
- pScissors – Array with elements describing parameters of the scissor test for each viewport.

Remember that the viewportCount and scissorCount parameters must be equal. We are also allowed to specify more viewports, but then the multiViewport feature must be also enabled.

Preparing the Rasterization State’s Description

The next part of the graphics pipeline creation applies to the rasterization state. We must specify how polygons are going to be rasterized (changed into fragments), which means whether we want fragments to be generated for whole polygons or just their edges (polygon mode) or whether we want to see the front or back side or maybe both sides of the polygon (face culling). We can also provide depth bias parameters or indicate whether we want to enable depth clamp. This whole state is encapsulated into VkPipelineRasterizationStateCreateInfo. It contains the following members:

- sType – Structure type, VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO in this example.
- pNext – Pointer reserved for extensions.
- flags – Parameter reserved for future use.
- depthClampEnable – Parameter describing whether we want to clamp depth values of the rasterized primitive to the frustum (when true) or if we want normal clipping to occur (false).
- rasterizerDiscardEnable – Deactivates fragment generation (discards primitive before rasterization turning off fragment shader).
- polygonMode – Controls how the fragments are generated for a given primitive (triangle mode): whether they are generated for the whole triangle, only its edges, or just its vertices.
- cullMode – Chooses the triangle’s face used for culling (if enabled).
- frontFace – Chooses which side of a triangle should be considered the front (depending on the winding order).
- depthBiasEnable – Enabled or disables biasing of fragments’ depth values.
- depthBiasConstantFactor – Constant factor added to each fragment’s depth value when biasing is enabled.
- depthBiasClamp – Maximum (or minimum) value of bias that can be applied to fragment’s depth.
- depthBiasSlopeFactor – Factor applied for fragment’s slope during depth calculations when biasing is enabled.
- lineWidth – Width of rasterized lines.

Here is the source code responsible for setting rasterization state in our example:

```
    VkPipelineRasterizationStateCreateInfo rasterization_state_create_info = {
      VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO,    // VkStructureType
sType
      nullptr,                                                       // const void
*pNext
      0,                                                             //
VkPipelineRasterizationStateCreateFlags        flags
      VK_FALSE,                                                      // VkBool32
depthClampEnable
      VK_FALSE,                                                      // VkBool32
rasterizerDiscardEnable
      VK_POLYGON_MODE_FILL,                                          // VkPolygonMode
polygonMode
      VK_CULL_MODE_BACK_BIT,                                         // VkCullModeFlags
cullMode
      VK_FRONT_FACE_COUNTER_CLOCKWISE,                               // VkFrontFace
frontFace
      VK_FALSE,                                                      // VkBool32
depthBiasEnable
      0.0f,                                                          // float
depthBiasConstantFactor
      0.0f,                                                          // float
depthBiasClamp
      0.0f,                                                          // float
depthBiasSlopeFactor
      1.0f                                                           // float
lineWidth
    };
```

14. Tutorial03.cpp, function CreatePipeline()

In the tutorial we are disabling as many parameters as possible to simplify the process, the code itself, and the rendering operations. The parameters that matter here set up (typical) fill mode for polygon rasterization, back face culling, and similar to OpenGL's counterclockwise front faces. Depth biasing and clamping are also disabled (to enable depth clamping, we first need to enable a dedicated feature during logical device creation; similarly we must do the same for polygon modes other than "fill").

Setting the Multisampling State's Description

In Vulkan, when we are creating a graphics pipeline, we must also specify the state relevant to multisampling. This is done using the VkPipelineMultisampleStateCreateInfo structure. Here are its members:

- sType – Type of structure, VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO here.
- pNext – Pointer reserved for extensions.
- flags – Parameter reserved for future use.
- rasterizationSamples – Number of per pixel samples used in rasterization.
- sampleShadingEnable – Parameter specifying that shading should occur per sample (when enabled) instead of per fragment (when disabled).
- minSampleShading – Specifies the minimum number of unique sample locations that should be used during the given fragment's shading.
- pSampleMask – Pointer to an array of static coverage sample masks; this can be null.
- alphaToCoverageEnable – Controls whether the fragment's alpha value should be used for coverage calculations.
- alphaToOneEnable – Controls whether the fragment's alpha value should be replaced with one.

In this example, I wanted to minimize possible problems so I've set parameters to values that generally disable multisampling—just one sample per given pixel with the other parameters turned off. Remember that if we want to enable sample shading or alpha to one, we also need to enable two respective features. Here is a source code that prepares the VkPipelineMultisampleStateCreateInfo structure:

```
   VkPipelineMultisampleStateCreateInfo multisample_state_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO,     // VkStructureType
sType
     nullptr,                                                      // const void
*pNext
     0,                                                            //
VkPipelineMultisampleStateCreateFlags          flags
     VK_SAMPLE_COUNT_1_BIT,                                        //
VkSampleCountFlagBits                          rasterizationSamples
     VK_FALSE,                                                     // VkBool32
sampleShadingEnable
     1.0f,                                                         // float
minSampleShading
     nullptr,                                                      // const VkSampleMask
*pSampleMask
     VK_FALSE,                                                     // VkBool32
alphaToCoverageEnable
     VK_FALSE                                                      // VkBool32
alphaToOneEnable
   };
```

15. Tutorial03.cpp, function CreatePipeline()

Setting the Blending State's Description

Another thing we need to prepare when creating a graphics pipeline is a blending state (which also includes logical operations).

```
   VkPipelineColorBlendAttachmentState color_blend_attachment_state = {
     VK_FALSE,                                                   // VkBool32
blendEnable
     VK_BLEND_FACTOR_ONE,                                        // VkBlendFactor
srcColorBlendFactor
     VK_BLEND_FACTOR_ZERO,                                       // VkBlendFactor
dstColorBlendFactor
     VK_BLEND_OP_ADD,                                            // VkBlendOp
colorBlendOp
     VK_BLEND_FACTOR_ONE,                                        // VkBlendFactor
srcAlphaBlendFactor
     VK_BLEND_FACTOR_ZERO,                                       // VkBlendFactor
dstAlphaBlendFactor
     VK_BLEND_OP_ADD,                                            // VkBlendOp
alphaBlendOp
     VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |       //
VkColorComponentFlags                          colorWriteMask
     VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT
   };

   VkPipelineColorBlendStateCreateInfo color_blend_state_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO,     // VkStructureType
sType
     nullptr,                                                    // const void
*pNext
     0,                                                          //
VkPipelineColorBlendStateCreateFlags           flags
     VK_FALSE,                                                   // VkBool32
logicOpEnable
     VK_LOGIC_OP_COPY,                                           // VkLogicOp
logicOp
     1,                                                          // uint32_t
attachmentCount
     &color_blend_attachment_state,                              // const
VkPipelineColorBlendAttachmentState     *pAttachments
     { 0.0f, 0.0f, 0.0f, 0.0f }                                  // float
blendConstants[4]
   };
```

16. Tutorial03.cpp, function CreatePipeline()

Final color operations are set up through the VkPipelineColorBlendStateCreateInfo structure. It contains the following fields:

- sType – Type of the structure, set to VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO in this example.
- pNext – Pointer reserved for future, extension-specific use.
- flags – Parameter also reserved for future use.
- logicOpEnable – Indicates whether we want to enable logical operations on pixels.
- logicOp – Type of the logical operation we want to perform (like copy, clear, and so on)
- attachmentCount – Number of elements in the pAttachments array.
- pAttachments – Array containing state parameters for each color attachment used in a subpass for which the given graphics pipeline is bound.
- blendConstants – Four-element array with color value used in blending operation (when a dedicated blend factor is used).

More information is needed for the attachmentCount and pAttachments parameters. When we want to perform drawing operations we set up parameters, the most important of which are graphics pipeline, render pass, and framebuffer. The graphics card needs to know how to draw (graphics pipeline which describes rendering state, shaders, test, and so on) and where to draw (the render pass gives general setup; the framebuffer specifies exactly what images are used). As I have already mentioned, the render pass specifies how operations are ordered, what the dependencies are, when we are rendering into a given attachment, and when we are reading from the same attachment. These stages take the form of subpasses. And for each drawing operation we can (but don't have to) enable/use a different pipeline. But when we are drawing, we must remember that we are drawing into a set of attachments. This set is defined in a render pass, which describes all color, input, depth attachments (the framebuffer just specifies what images are used for each of them). For the blending state, we can specify whether we want to enable blending at all. This is done through the pAttachments array. Each of its elements must correspond to each color attachment defined in a render pass. So the value of attachmentCount elements in the pAttachments array must equal the number of color attachments defined in a render pass.

There is one more restriction. By default all elements in pAttachments array must contain the same values, must be specified in the same way, and must be identical. By default, blending (and color masks) is done in the same way for all attachments. So why it is an array? Why can't we just specify one value? Because there is a feature that allows us to perform independent, distinct blending for each active color attachment. When we enable the independent blending feature during device creation we can provide different values for each color attachment.

Each pAttachments array's element is of type VkPipelineColorBlendAttachmentState. It is a structure with the following members:

- blendEnable – Indicates whether we want to enable blending at all.
- srcColorBlendFactor – Blending factor for color of the source (incoming) fragment.
- dstColorBlendFactor – Blending factor for the destination color (stored already in the framebuffer at the same location as the incoming fragment).
- colorBlendOp – Type of operation to perform (multiplication, addition, and so on).
- srcAlphaBlendFactor – Blending factor for the alpha value of the source (incoming) fragment.
- dstAlphaBlendFactor – Blending factor for the destination alpha value (already stored in the framebuffer).
- alphaBlendOp – Type of operation to perform for alpha blending.
- colorWriteMask – Bitmask selecting which of the RGBA components are selected (enabled) for writing.

In this example, we disable blending, which causes all other parameters to be irrelevant. Except for colorWriteMask, we select all components for writing but you can freely check what will happen when this parameter is changed to some other R, G, B, A combinations.

Creating a Pipeline Layout

The final thing we must do before pipeline creation is create a proper pipeline layout. A pipeline layout describes all the resources that can be accessed by the pipeline. In this example we must specify how many textures can be used by shaders and which shader stages will have access to them. There are of course other resources involved. Apart from shader stages, we must also describe the types of resources (textures, buffers), their total numbers, and layout. This layout can be compared to OpenGL's active textures and shader uniforms. In OpenGL we bind textures to the desired texture image units and for shader uniforms we don't provide texture handles but IDs of the texture image units to which actual textures are bound (we provide the number of the unit which the given texture was associated with).

With Vulkan, the situation is similar. We create some form of a memory layout: first there are two buffers, next we have three textures and an image. This memory "structure" is called a set and a collection of these sets is provided for the pipeline. In shaders, we access specified resources using specific memory "locations" from within these sets (layouts). This is done through a layout (set = X, binding = Y) specifier, which can be translated to: take the resource from the Y memory location from the X set.

And pipeline layout can be thought of as an interface between shader stages and shader resources as it takes these groups of resources, describes how they are gathered, and provides them to the pipeline.

This process is complex and I plan to devote a tutorial to it. Here we are not using any additional resources so I present an example for creating an "empty" pipeline layout:

```
   VkPipelineLayoutCreateInfo layout_create_info = {
     VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,  // VkStructureType
sType
     nullptr,                                        // const void
*pNext
     0,                                              // VkPipelineLayoutCreateFlags
flags
     0,                                              // uint32_t
setLayoutCount
     nullptr,                                        // const VkDescriptorSetLayout
*pSetLayouts
     0,                                              // uint32_t
pushConstantRangeCount
     nullptr                                         // const VkPushConstantRange
*pPushConstantRanges
   };

   VkPipelineLayout pipeline_layout;
   if( vkCreatePipelineLayout( GetDevice(), &layout_create_info, nullptr,
&pipeline_layout ) != VK_SUCCESS ) {
     printf( "Could not create pipeline layout!\n" );
     return Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout>();
   }

   return Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout>(
pipeline_layout, vkDestroyPipelineLayout, GetDevice() );
```

17. Tutorial03.cpp, function CreatePipelineLayout()

To create a pipeline layout we must first prepare a variable of type VkPipelineLayoutCreateInfo. It contains the following fields:

- sType – Type of structure, VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO in this example.
- pNext – Parameter reserved for extensions.
- flags – Parameter reserved for future use.
- setLayoutCount – Number of descriptor sets included in this layout.
- pSetLayouts – Pointer to an array containing descriptions of descriptor layouts.
- pushConstantRangeCount – Number of push constant ranges (I will describe it in a later tutorial).
- pPushConstantRanges – Array describing all push constant ranges used inside shaders (in a given pipeline).

In this example we create "empty" layout so almost all the fields are set to null or zero.

We are not using push constants here, but they deserve some explanation. Push constants in Vulkan allow us to modify the data of constant variables used in shaders. There is a special, small amount of memory reserved for push constants. We update their values through Vulkan commands, not through memory updates, and it is expected that updates of push constants' values are faster than normal memory writes.

As shown in the above example, I'm also wrapping pipeline layout in an "AutoDeleter" object. Pipeline layouts are required during pipeline creation, descriptor sets binding (enabling/activating this interface between shaders and shader resources) and push constants setting. None of these operations, except for pipeline creation, take place in this tutorial. So here, after we create a pipeline, we don't need the layout anymore. To avoid memory leaks, I have used this helper class to destroy the layout as soon as we leave the function in which graphics pipeline is created.

Creating a Graphics Pipeline

Now we have all the resources required to properly create graphics pipeline. Here is the code that does that:

```
   Tools::AutoDeleter<VkPipelineLayout, PFN_vkDestroyPipelineLayout> pipeline_layout =
CreatePipelineLayout();
   if( !pipeline_layout ) {
     return false;
   }

   VkGraphicsPipelineCreateInfo pipeline_create_info = {
     VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,              // VkStructureType
sType
     nullptr,                                                      // const void
*pNext
     0,                                                            //
VkPipelineCreateFlags                                    flags
     static_cast<uint32_t>(shader_stage_create_infos.size()),      // uint32_t
stageCount
     &shader_stage_create_infos[0],                                // const
VkPipelineShaderStageCreateInfo         *pStages
     &vertex_input_state_create_info,                              // const
VkPipelineVertexInputStateCreateInfo    *pVertexInputState;
     &input_assembly_state_create_info,                            // const
VkPipelineInputAssemblyStateCreateInfo  *pInputAssemblyState
     nullptr,                                                      // const
VkPipelineTessellationStateCreateInfo   *pTessellationState
     &viewport_state_create_info,                                  // const
VkPipelineViewportStateCreateInfo       *pViewportState
     &rasterization_state_create_info,                             // const
VkPipelineRasterizationStateCreateInfo  *pRasterizationState
     &multisample_state_create_info,                               // const
VkPipelineMultisampleStateCreateInfo    *pMultisampleState
     nullptr,                                                      // const
VkPipelineDepthStencilStateCreateInfo   *pDepthStencilState
     &color_blend_state_create_info,                               // const
VkPipelineColorBlendStateCreateInfo     *pColorBlendState
     nullptr,                                                      // const
VkPipelineDynamicStateCreateInfo        *pDynamicState
     pipeline_layout.Get(),                                        // VkPipelineLayout
layout
     Vulkan.RenderPass,                                            // VkRenderPass
renderPass
     0,                                                            // uint32_t
subpass
     VK_NULL_HANDLE,                                               // VkPipeline
basePipelineHandle
     -1                                                            // int32_t
basePipelineIndex
   };

   if( vkCreateGraphicsPipelines( GetDevice(), VK_NULL_HANDLE, 1, &pipeline_create_info,
nullptr, &Vulkan.GraphicsPipeline ) != VK_SUCCESS ) {
     printf( "Could not create graphics pipeline!\n" );
     return false;
   }
   return true;
```

18. Tutorial03.cpp, function CreatePipeline()

First we create a pipeline layout wrapped in an object of type “AutoDeleter”. Next we fill the structure of type VkGraphicsPipelineCreateInfo. It contains many fields. Here is a brief description of them:

- sType – Type of structure, VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO here.
- pNext – Parameter reserved for future, extension-related use.
- flags – This time this parameter is not reserved for future use but controls how the pipeline should be created: if we are creating a derivative pipeline (if we are inheriting from another pipeline) or if we allow creating derivative pipelines from this one. We can also disable optimizations, which should shorten the time needed to create a pipeline.
- stageCount – Number of stages described in the pStages parameter; must be greater than zero.
- pStages – Array with descriptions of active shader stages (the ones created using shader modules); each stage must be unique (we can't specify a given stage more than once). There also must be a vertex stage present.
- pVertexInputState – Pointer to a variable contain the description of the vertex input's state.
- pInputAssemblyState – Pointer to a variable with input assembly description.
- pTessellationState – Pointer to a description of the tessellation stages; can be null if tessellation is disabled.
- pViewportState – Pointer to a variable specifying viewport parameters; can be null if rasterization is disabled.
- pRasterizationState – Pointer to a variable specifying rasterization behavior.
- pMultisampleState – Pointer to a variable defining multisampling; can be null if rasterization is disabled.
- pDepthStencilState – Pointer to a description of depth/stencil parameters; this can be null in two situations: when rasterization is disabled or we're not using depth/stencil attachments in a render pass.
- pColorBlendState – Pointer to a variable with color blending/write masks state; can be null also in two situations: when rasterization is disabled or when we're not using any color attachments inside the render pass.
- pDynamicState – Pointer to a variable specifying which parts of the graphics pipeline can be set dynamically; can be null if the whole state is considered static (defined only through this create info structure).
- layout – Handle to a pipeline layout object that describes resources accessed inside shaders.
- renderPass – Handle to a render pass object; pipeline can be used with any render pass compatible with the provided one.
- subpass – Number (index) of a subpass in which the pipeline will be used.
- basePipelineHandle – Handle to a pipeline this one should derive from.
- basePipelineIndex – Index of a pipeline this one should derive from.

When we are creating a new pipeline, we can inherit some of the parameters from another one. This means that both pipelines should have much in common. A good example is shader code. We don't specify what fields are the same, but the general message that the pipeline inherits from another one may substantially accelerate pipeline creation. But why are there two fields to indicate a "parent" pipeline? We can't use them both—only one of them at a time. When we are using a handle, this means that the "parent" pipeline is already created and we are deriving from the one we have provided the handle of. But the pipeline creation function allows us to create many pipelines at once. Using the second parameter, "parent" pipeline index, we can create both "parent" and "child" pipelines in the same call. We just specify an array of graphics pipeline creation info structures and this array is provided to pipeline creation function. So the "basePipelineIndex" is the index of pipeline creation info in this very array. We just have to remember that the "parent" pipeline must be earlier (must have a smaller index) in this array and it must be created with the "allow derivatives" flag set.

In this example we are creating a pipeline with the state being entirely static (null for the "pDynamicState" parameter). But what is a dynamic state? To allow for some flexibility and to lower the number of created pipeline objects, the dynamic state was introduced. We can define through the "pDynamicState" parameter what parts of the graphics pipeline can be set dynamically through additional Vulkan commands and what parts are being static, set once during pipeline creation. The dynamic state includes parameters such as viewports, line widths, blend constants, or some stencil parameters. If we specify that a given state is dynamic, parameters in a pipeline creation info structure that are related to that state are ignored. We must set the given state using the proper Vulkan commands during rendering because initial values of such state may be undefined.

So after these quite overwhelming preparations we can create a graphics pipeline. This is done by calling the **vkCreateGraphicsPipelines()** function which, among others, takes an array of pointers to the pipeline create info structures. When everything goes well, VK_SUCCESS should be returned by this function and a handle of a graphics pipeline should be stored in a variable we've provided the address of. Now we are ready to start drawing.

Preparing Drawing Commands

I introduced you to the concept of command buffers in the previous tutorial. Here I will briefly explain what are they and how to use them.

Command buffers are containers for GPU commands. If we want to execute some job on a device, we do it through command buffers. This means that we must prepare a set of commands that process data (that is, draw something on the screen) and record these commands in command buffers. Then we can submit whole buffers to device's queues. This submit operation tells the device: here is a bunch of things I want you to do for me and do them now.

To record commands, we must first allocate command buffers. These are allocated from command pools, which can be thought of as memory chunks. If a command buffer needs to be larger (as we record many complicated commands in it) it can grow and use additional memory from a pool it was allocated with. So first we must create a command pool.

Creating a Command Pool

Command pool creation is simple and looks like this:

```
   VkCommandPoolCreateInfo cmd_pool_create_info = {
     VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,      // VkStructureType
sType
     nullptr,                                         // const void
*pNext
     0,                                               // VkCommandPoolCreateFlags
flags
     queue_family_index                               // uint32_t
queueFamilyIndex
   };

   if( vkCreateCommandPool( GetDevice(), &cmd_pool_create_info, nullptr, pool ) !=
VK_SUCCESS ) {
     return false;
   }
   return true;
```

19. Tutorial03.cpp, function CreateCommandPool()

First we prepare a variable of type VkCommandPoolCreateInfo. It contains the following fields:

- sType – Standard type of structure, set to VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO here.
- pNext – Pointer reserved for extensions.
- flags – Indicates usage scenarios for command pool and command buffers allocated from it; that is, we can tell the driver that command buffers allocated from this pool will live for a short time; for no specific usage we can set it to zero.
- queueFamilyIndex – Index of a queue family for which we are creating a command pool.

Remember that command buffers allocated from a given pool can only be submitted to a queue from a queue family specified during pool creation.

To create a command pool, we just call the **vkCreateCommandPool()** function.

Allocating Command Buffers

Now that we have the command pool ready, we can allocate command buffers from it.

```
   VkCommandBufferAllocateInfo command_buffer_allocate_info = {
     VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO, // VkStructureType
sType
     nullptr,                                        // const void
*pNext
     pool,                                           // VkCommandPool
commandPool
     VK_COMMAND_BUFFER_LEVEL_PRIMARY,                // VkCommandBufferLevel
level
     count                                           // uint32_t
bufferCount
   };

   if( vkAllocateCommandBuffers( GetDevice(), &command_buffer_allocate_info,
command_buffers ) != VK_SUCCESS ) {
     return false;
   }
   return true;
```

20. Tutorial03.cpp, function AllocateCommandBuffers()

To allocate command buffers we specify a variable of structure type. This time its type is VkCommandBufferAllocateInfo, which contains these members:

- sType – Type of the structure; VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO for this purpose.
- pNext – Pointer reserved for extensions.
- commandPool – Pool from which we want our command buffers to take their memory.
- level – Command buffer level; there are two levels: primary and secondary; right now we are only interested in primary command buffers.
- bufferCount – Number of command buffers we want to allocate.

To allocate command buffers, call the **vkAllocateCommandBuffers()** function and check whether it succeeded. We can allocate many buffers at once with one function call.

I've prepared a simple buffer allocating function to show you how some Vulkan functions can be wrapped for easier use. Here is a usage of two such wrapper functions that create command pools and allocate command buffers from them.

```
   if( !CreateCommandPool( GetGraphicsQueue().FamilyIndex, &Vulkan.GraphicsCommandPool )
) {
     printf( "Could not create command pool!\n" );
     return false;
   }

   uint32_t image_count = static_cast<uint32_t>(GetSwapChain().Images.size());
   Vulkan.GraphicsCommandBuffers.resize( image_count, VK_NULL_HANDLE );

   if( !AllocateCommandBuffers( Vulkan.GraphicsCommandPool, image_count,
&Vulkan.GraphicsCommandBuffers[0] ) ) {
     printf( "Could not allocate command buffers!\n" );
     return false;
   }
   return true;
```

21. Tutorial03.cpp, function CreateCommandBuffers()

As you can see, we are creating a command pool for a graphics queue family index. All image state transitions and drawing operations will be performed on a graphics queue. Presentation is done on another queue (if the presentation queue is different from the graphics queue) but we don't need a command buffer for this operation.

And we are also allocating command buffers for each swap chain image. Here we take number of images and provide it to this simple "wrapper" function for command buffer allocation.

Recording Command Buffers

Now that we have command buffers allocated from the command pool we can finally record operations that will draw something on the screen. First we must prepare a set of data needed for the recording operation. Some of this data is identical for all command buffers, but some is referencing a specific swap chain image. Here is a code that is independent of swap chain images:

```
   VkCommandBufferBeginInfo graphics_commandd_buffer_begin_info = {
     VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,    // VkStructureType
sType
     nullptr,                                        // const void
*pNext
     VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT,   // VkCommandBufferUsageFlags
flags
     nullptr                                         // const
VkCommandBufferInheritanceInfo  *pInheritanceInfo
   };

   VkImageSubresourceRange image_subresource_range = {
     VK_IMAGE_ASPECT_COLOR_BIT,                      // VkImageAspectFlags
aspectMask
     0,                                              // uint32_t
baseMipLevel
     1,                                              // uint32_t
levelCount
     0,                                              // uint32_t
baseArrayLayer
     1                                               // uint32_t
layerCount
   };

   VkClearValue clear_value = {
     { 1.0f, 0.8f, 0.4f, 0.0f },                     // VkClearColorValue
color
   };

   const std::vector<VkImage>& swap_chain_images = GetSwapChain().Images;
```

22. Tutorial03.cpp, function RecordCommandBuffers()

Performing command buffer recording is similar to OpenGL's drawing lists where we start recording a list by calling the glNewList() function. Next we prepare a set of drawing commands and then we close the list or stop recording it (glEndList()). So the first thing we need to do is to prepare a variable of type VkCommandBufferBeginInfo. It is used when we start recording a command buffer and it tells the driver about the type, contents, and desired usage of a command buffer. Variables of this type contain the following members:

- sType – Standard structure type, here set to VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO.
- pNext – Pointer reserved for extensions.
- flags – Parameters describing the desired usage (that is, if we want to submit this command buffer only once and destroy/reset it or if it is possible that the buffer will submitted again before the processing of its previous submission has finished).
- pInheritanceInfo – Parameter used only when we want to record a secondary command buffer.

Next we describe the areas or parts of our images that we will set up image memory barriers for. Here we set up barriers to specify that queues from different families will reference a given image. This is done through a variable of type VkImageSubresourceRange with the following members:

- aspectMask – Describes a “type” of image, whether it is for color, depth, or stencil data.
- baseMipLevel – Number of a first mipmap level our operations will be performed on.
- levelCount – Number of mipmap levels (including base level) we will be operating on.
- baseArrayLayer – Number of an first array layer of an image that will take part in operations.
- layerCount – Number of layers (including base layer) that will be modified.

Next we set up a clear value for our images. Before drawing we need to clear images. In previous tutorials, we performed this operation explicitly by ourselves. Here images are cleared as a part of a render pass attachment load operation. We set to “clear” so now we must specify the color to which an image must be cleared. This is done using a variable of type VkClearValue in which we provide R, G, B, A values.

Variables we have created thus far are independent of an image itself, and that’s why we have specified them before a loop. Now we can start recording command buffers:

```
    for( size_t i = 0; i < Vulkan.GraphicsCommandBuffers.size(); ++i ) {
      vkBeginCommandBuffer( Vulkan.GraphicsCommandBuffers[i],
&graphics_commandd_buffer_begin_info );

      if( GetPresentQueue().Handle != GetGraphicsQueue().Handle ) {
        VkImageMemoryBarrier barrier_from_present_to_draw = {
          VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,     // VkStructureType
sType
          nullptr,                                    // const void
*pNext
          VK_ACCESS_MEMORY_READ_BIT,                  // VkAccessFlags
srcAccessMask
          VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,       // VkAccessFlags
dstAccessMask
          VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout
oldLayout
          VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,            // VkImageLayout
newLayout
          GetPresentQueue().FamilyIndex,              // uint32_t
srcQueueFamilyIndex
          GetGraphicsQueue().FamilyIndex,             // uint32_t
dstQueueFamilyIndex
          swap_chain_images[i],                       // VkImage
image
          image_subresource_range                     // VkImageSubresourceRange
subresourceRange
        };
        vkCmdPipelineBarrier( Vulkan.GraphicsCommandBuffers[i],
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, 0, 0, nullptr, 0, nullptr, 1,
&barrier_from_present_to_draw );
      }

      VkRenderPassBeginInfo render_pass_begin_info = {
        VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,     // VkStructureType
sType
        nullptr,                                      // const void
*pNext
        Vulkan.RenderPass,                            // VkRenderPass
renderPass
        Vulkan.FramebufferObjects[i].Handle,          // VkFramebuffer
framebuffer
        {                                             // VkRect2D
renderArea
          {                                           // VkOffset2D
offset
```

```
          0,                                          // int32_t
x
          0                                           // int32_t
y
        },
        {                                           // VkExtent2D
extent
          300,                                        // int32_t
width
          300,                                        // int32_t
height
        }
      },
        1,                                            // uint32_t
clearValueCount
        &clear_value                                  // const VkClearValue
*pClearValues
     };

     vkCmdBeginRenderPass( Vulkan.GraphicsCommandBuffers[i], &render_pass_begin_info,
VK_SUBPASS_CONTENTS_INLINE );

     vkCmdBindPipeline( Vulkan.GraphicsCommandBuffers[i],
VK_PIPELINE_BIND_POINT_GRAPHICS, Vulkan.GraphicsPipeline );

     vkCmdDraw( Vulkan.GraphicsCommandBuffers[i], 3, 1, 0, 0 );

     vkCmdEndRenderPass( Vulkan.GraphicsCommandBuffers[i] );

     if( GetGraphicsQueue().Handle != GetPresentQueue().Handle ) {
       VkImageMemoryBarrier barrier_from_draw_to_present = {
         VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,       // VkStructureType
sType
         nullptr,                                      // const void
*pNext
         VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,         // VkAccessFlags
srcAccessMask
         VK_ACCESS_MEMORY_READ_BIT,                    // VkAccessFlags
dstAccessMask
         VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,              // VkImageLayout
oldLayout
         VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,              // VkImageLayout
newLayout
         GetGraphicsQueue().FamilyIndex,               // uint32_t
srcQueueFamilyIndex
         GetPresentQueue( ).FamilyIndex,               // uint32_t
dstQueueFamilyIndex
         swap_chain_images[i],                         // VkImage
image
         image_subresource_range                       // VkImageSubresourceRange
subresourceRange
       };
       vkCmdPipelineBarrier( Vulkan.GraphicsCommandBuffers[i],
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0,
0, nullptr, 0, nullptr, 1, &barrier_from_draw_to_present );
     }
     if( vkEndCommandBuffer( Vulkan.GraphicsCommandBuffers[i] ) != VK_SUCCESS ) {
       printf( "Could not record command buffer!\n" );
       return false;
     }
   }
   return true;
```

23. Tutorial03.cpp, function RecordCommandBuffers()

Recording a command buffer is started by calling the **vkBeginCommandBuffer()** function. At the beginning we set up a barrier that tells the driver that previously queues from one family referenced a given image but now queues from a different family will be referencing it (we need to do this because during swap chain creation we specified exclusive sharing mode). The barrier is set only when the graphics queue is different than the present queue. This is done by calling the **vkCmdPipelineBarrier()** function. We must specify when in the pipeline the barrier should be placed (VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT) and how the barrier should be set up. Barrier parameters are prepared through the VkImageMemoryBarrier structure:

- sType – Type of the structure, here set to VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER.
- pNext – Pointer reserved for extensions.
- srcAccessMask – Type of memory operations that took place in regard to a given image before the barrier.
- dstAccessMask – Type of memory operations connected with a given image that will take place after the barrier.
- oldLayout – Current image memory layout.
- newLayout – Memory layout image you should have after the barrier.
- srcQueueFamilyIndex – Index of a family of queues which were referencing image before the barrier.
- dstQueueFamilyIndex – Index of a queue family queues from which will be referencing image after the barrier.
- image – Handle to the image itself.
- subresourceRange – Parts of an image for which we want the transition to occur.

In this example we don't change the layout of an image, for two reasons: (1) The barrier may not be set at all (if the graphics and present queues are the same), and (2) the layout transition will be performed automatically as a render pass operation (at the beginning of the first—and only—subpass).

Next we start a render pass. We call the **vkCmdBeginRenderPass()** function for which we must provide a pointer to a variable of VkRenderPassBeginInfo type. It contains the following members:

- sType – Standard type of structure. In this case we must set it to a value of VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO.
- pNext – Pointer reserved for future use.
- renderPass – Handle of a render pass we want to start.
- framebuffer – Handle of a framebuffer, which specifies images used as attachments for this render pass.
- renderArea – Area of all images that will be affected by the operations that takes place in this render pass. It specifies the upper-left corner (through x and y parameters of an offset member) and width and height (through extent member) of a render area.
- clearValueCount – Number of elements in pClearValues array.
- pClearValues – Array with clear values for each attachment.

When we specify a render area for the render pass, we must make sure that the rendering operations won't modify pixels outside this area. This is just a hint for a driver so it could optimize its behavior. If we won't confine operations to the provided area by using a proper scissor test, pixels outside this area may become undefined (we can't rely on their contents). We also can't specify a render area that is greater than a framebuffer's dimensions (falls outside the framebuffer).

And with a pClearValues array, it must contain the elements for each render pass attachment. Each of its members specifies the color to which the given attachment must be cleared when its loadOp is set to clear. For attachments where loadOp is not clear, the values provided for them are ignored. But we can't provide an array with a smaller amount of elements.

We have begun a command buffer, set a barrier (if necessary), and started a render pass. When we start a render pass we are also starting its first subpass. We can switch to the next subpass by calling the **vkCmdNextSubpass()** function. During these operations, layout transitions and clear operations may occur. Clears are done in a subpass in which the image is first used (referenced). Layout transitions occur each time a subpass layout is different than the layout in a previous subpass or (in the case of a first subpass or when the image is first referenced) different than the initial layout (layout before the render pass). So in our example when we start a render pass, the swap chain image's layout is changed automatically from "presentation source" to a "color attachment optimal" layout.

Now we bind a graphics pipeline. This is done by calling the **vkCmdBindPipeline()** function. This "activates" all shader programs (similar to the **glUseProgram()** function) and sets desired tests, blending operations, and so on.

After the pipeline is bound, we can finally draw something by calling the **vkCmdDraw()** function. In this function we specify the number of vertices we want to draw (three), number of instances that should be drawn (just one), and a numbers or indices of a first vertex and first instance (both zero).

Next the **vkCmdEndRenderPass()** function is called which, as the name suggests, ends the given render pass. Here all final layout transitions occur if the final layout specified for a render pass is different from the layout used in the last subpass the given image was referenced in.

After that, the barrier may be set in which we tell the driver that the graphics queue finished using a given image and from now on the present queue will be using it. This is done, once again, only when the graphics and present queues are different. And after the barrier, we stop recording a command buffer for a given image. All these operations are repeated for each swap chain image.

Drawing

The drawing function is the same as the **Draw()** function presented in Tutorial 2. We acquire the image's index, submit a proper command buffer, and present an image. We are using semaphores the same way they were used previously: one semaphore is used for acquiring an image and it tells the graphics queue to wait when the image is not yet available for use. The second command buffer is used to indicate whether drawing on a graphics queue is finished. The present queue waits on this semaphore before it can present an image. Here is the source code of a **Draw()** function:

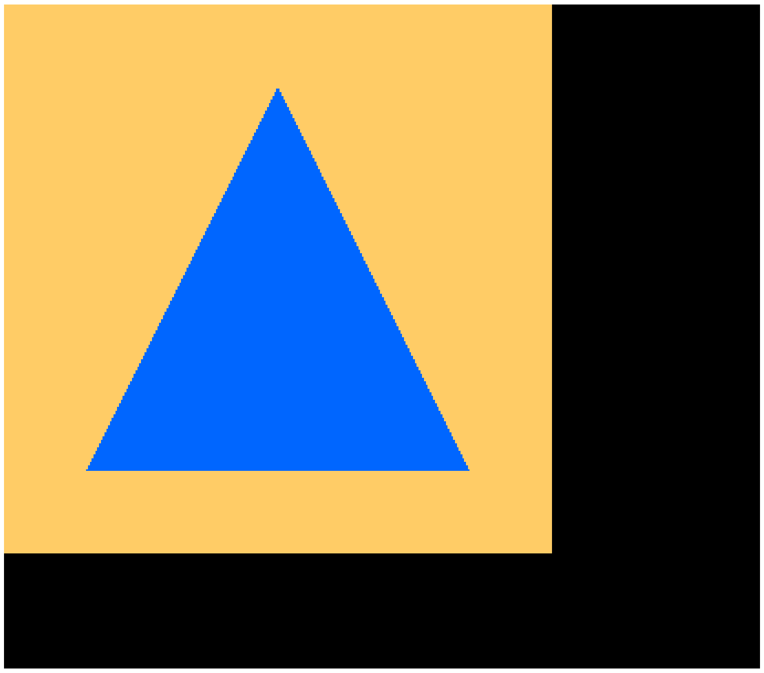
```
    VkSemaphore image_available_semaphore = GetImageAvailableSemaphore();
    VkSemaphore rendering_finished_semaphore = GetRenderingFinishedSemaphore();
    VkSwapchainKHR swap_chain = GetSwapChain().Handle;
    uint32_t image_index;

    VkResult result = vkAcquireNextImageKHR( GetDevice(), swap_chain, UINT64_MAX,
 image_available_semaphore, VK_NULL_HANDLE, &image_index );
    switch( result ) {
      case VK_SUCCESS:
      case VK_SUBOPTIMAL_KHR:
        break;
      case VK_ERROR_OUT_OF_DATE_KHR:
        return OnWindowSizeChanged();
      default:
        printf( "Problem occurred during swap chain image acquisition!\n" );
        return false;
    }

    VkPipelineStageFlags wait_dst_stage_mask = VK_PIPELINE_STAGE_TRANSFER_BIT;
    VkSubmitInfo submit_info = {
      VK_STRUCTURE_TYPE_SUBMIT_INFO,                // VkStructureType              sType
      nullptr,                                      // const void                  *pNext
      1,                                            // uint32_t
 waitSemaphoreCount
      &image_available_semaphore,                   // const VkSemaphore
 *pWaitSemaphores
```

```
      &wait_dst_stage_mask,                           // const VkPipelineStageFlags
*pWaitDstStageMask;
      1,                                              // uint32_t
commandBufferCount
      &Vulkan.GraphicsCommandBuffers[image_index],   // const VkCommandBuffer
*pCommandBuffers
      1,                                              // uint32_t
signalSemaphoreCount
      &rendering_finished_semaphore                  // const VkSemaphore
*pSignalSemaphores
    };

    if( vkQueueSubmit( GetGraphicsQueue().Handle, 1, &submit_info, VK_NULL_HANDLE ) !=
VK_SUCCESS ) {
      return false;
    }

    VkPresentInfoKHR present_info = {
      VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,            // VkStructureType              sType
      nullptr,                                        // const void                  *pNext
      1,                                              // uint32_t
waitSemaphoreCount
      &rendering_finished_semaphore,                 // const VkSemaphore
*pWaitSemaphores
      1,                                              // uint32_t
swapchainCount
      &swap_chain,                                    // const VkSwapchainKHR
*pSwapchains
      &image_index,                                   // const uint32_t
*pImageIndices
      nullptr                                         // VkResult
*pResults
    };
    result = vkQueuePresentKHR( GetPresentQueue().Handle, &present_info );

    switch( result ) {
      case VK_SUCCESS:
        break;
      case VK_ERROR_OUT_OF_DATE_KHR:
      case VK_SUBOPTIMAL_KHR:
        return OnWindowSizeChanged();
      default:
        printf( "Problem occurred during image presentation!\n" );
        return false;
    }

    return true;
```

24. Tutorial03.cpp, function Draw()

Tutorial 3 Execution

In this tutorial we performed "real" drawing operations. A simple triangle may not sound too convincing, but it is a good starting point for a first Vulkan-created image. Here is what the triangle should look like:



If you're wondering why there are black parts in the image, here is an explanation: To simplify the whole code, we created a framebuffer with a fixed size (width and height of 300 pixels). But the window's size (and the size of the swap chain images) may be greater than these 300 x 300 pixels. The parts of an image that lay outside of the framebuffer's dimensions are uncleared and unmodified by our application. They may even contain some "artifacts," because the memory from which the driver allocates the swap chain images may have been previously used for other purposes and could contain some data. The correct behavior is to create a framebuffer with the same size as the swap chain images and to recreate it when the window's size changes. But as long as the blue triangle is rendered on an orange/gold background, it means that the code works correctly.

Cleaning Up

One last thing to learn before this tutorial ends is how to release resources created during this lesson. I won't repeat the code needed to release resources created in the previous chapter. Just look at the VulkanCommon.cpp file. Here is the code needed to destroy resources specific to this chapter:

```
    if( GetDevice() != VK_NULL_HANDLE ) {
      vkDeviceWaitIdle( GetDevice() );

      if( (Vulkan.GraphicsCommandBuffers.size() > 0) && (Vulkan.GraphicsCommandBuffers[0]
!= VK_NULL_HANDLE) ) {
        vkFreeCommandBuffers( GetDevice(), Vulkan.GraphicsCommandPool,
static_cast<uint32_t>(Vulkan.GraphicsCommandBuffers.size()),
&Vulkan.GraphicsCommandBuffers[0] );
        Vulkan.GraphicsCommandBuffers.clear();
      }

      if( Vulkan.GraphicsCommandPool != VK_NULL_HANDLE ) {
        vkDestroyCommandPool( GetDevice(), Vulkan.GraphicsCommandPool, nullptr );
        Vulkan.GraphicsCommandPool = VK_NULL_HANDLE;
      }

      if( Vulkan.GraphicsPipeline != VK_NULL_HANDLE ) {
        vkDestroyPipeline( GetDevice(), Vulkan.GraphicsPipeline, nullptr );
        Vulkan.GraphicsPipeline = VK_NULL_HANDLE;
      }
      if( Vulkan.RenderPass != VK_NULL_HANDLE ) {
        vkDestroyRenderPass( GetDevice(), Vulkan.RenderPass, nullptr );
        Vulkan.RenderPass = VK_NULL_HANDLE;
      }

      for( size_t i = 0; i < Vulkan.FramebufferObjects.size(); ++i ) {
        if( Vulkan.FramebufferObjects[i].Handle != VK_NULL_HANDLE ) {
          vkDestroyFramebuffer( GetDevice(), Vulkan.FramebufferObjects[i].Handle, nullptr
);
          Vulkan.FramebufferObjects[i].Handle = VK_NULL_HANDLE;
        }

        if( Vulkan.FramebufferObjects[i].ImageView != VK_NULL_HANDLE ) {
          vkDestroyImageView( GetDevice(), Vulkan.FramebufferObjects[i].ImageView,
nullptr );
          Vulkan.FramebufferObjects[i].ImageView = VK_NULL_HANDLE;
        }
      }
      Vulkan.FramebufferObjects.clear();
    }
```

25. Tutorial03.cpp, function ChildClear()

As usual we first check whether there is any device. If we don't have a device, we don't have a resource. Next we wait until the device is free and we delete all the created resources. We start from deleting command buffers by calling a **vkFreeCommandBuffers()** function. Next we destroy a command pool through a **vkDestroyCommandPool()** function and after that the graphics pipeline is destroyed. This is achieved through a **vkDestroyPipeline()** function call. Next we call a **vkDestroyRenderPass()** function, which releases the handle to a render pass. Finally, all framebuffers and image views associated with each swap chain image are deleted.

Each object destruction is preceded by a check whether a given resource was properly created. If not we skip the process of destruction of such resource.

Conclusion

In this tutorial, we created a render pass with one subpass. Next we created image views and framebuffers for each swap chain image. One of the most difficult parts was to create a graphics pipeline, because it required us to prepare lots of data. We had to create shader modules and describe all the shader stages that should be active when a given graphics pipeline is bound. We had to prepare information about input vertices, their layout, and assembling them into polygons. Viewport, rasterization, multisampling, and color blending information was also necessary. Then we created a simple pipeline layout and after that we could create the pipeline itself. Next we created a command pool and allocated command buffers for each swap chain image. Operations recorded in each command buffer involved setting up an image memory barrier, beginning a render pass, binding a graphics pipeline, and drawing. Next we ended a render pass and set up another image memory barrier. The drawing itself was performed the same way as in the previous tutorial (2).

In the next tutorial, we will learn about the vertex attributes, images and buffers.