API without Secrets: Introduction to Vulkan* Part 4

Table of Contents

Tutorial 4: Vertex Attributes – Buffers, Images, and Fences

In previous tutorials we learned the basics. The tutorials themselves were long and (I hope) detailed enough. This is because the learning curve of a Vulkan API is quite steep. And, as you can see, a considerable amount of knowledge is necessary to prepare even the simplest application.

But now we can build on these foundations. So the tutorials will be shorter and focus on smaller topics related to a Vulkan API. In this part I present the recommended way of drawing arbitrary geometry by providing vertex attributes through vertex buffers. As the code of this lesson is similar to the code from the "03 – First Triangle" tutorial, I focus on and describe only the parts that are different.

I also show a different way of organizing the rendering code. Previously we recorded command buffers before the main rendering loop. But in real-life situations, every frame of animation is different, so we can't prerecord all the rendering commands. We should record and submit the command buffer as late as possible to minimize input lag and acquire as recent input data as possible. We will record the command buffer just before it is submitted to the queue. But a single command buffer isn't enough. We should not record the same command buffer until the graphics card finishes processing it after it was submitted. This moment is signaled through a fence. But waiting on a fence every frame is a waste of time, so we need more command buffers used interchangeably. With more command buffers, more fences are also needed and the situation gets more complicated. This tutorial shows how to organize the code so it is easily maintained, flexible, and as fast as possible.

Specifying Render Pass Dependencies

We start by creating a render pass, in the same way as the previous tutorial. But this time we will provide additional information. Render pass describes the internal organization of rendering resources (images/attachments), how they are used, and how they change during the rendering process. Images' layout changes can be performed explicitly by creating image memory barriers. But they can also be performed implicitly, when proper render pass description is specified (initial, subpass, and final image layouts). Implicit transition is preferred, as drivers can perform such transitions more optimally.

In this part of tutorial, identically as in the previous part, we specify "transfer src" for initial and final image layouts, and "color attachment optimal" subpass layout for our render pass. But previous tutorials lacked important, additional information, specifically how the image was used (that is, what types of operations occurred in connection with an image), and when it was used (which parts of a rendering pipeline were using an image). This information can be specified both in the image memory barrier and the render pass description. When we create an image memory barrier, we specify the types of operations which concern the given image (memory access types before and after barrier), and we also specify when this barrier should be placed (pipeline stages in which image was used before and after the barrier).

When we create a render pass and provide a description for it, the same information is specified through subpass dependencies. This additional data is crucial for a driver to optimally prepare an implicit barrier. Below is the source code that creates a render pass and prepares subpass dependencies.

```
    std::vector<VkSubpassDependency> dependencies = {
      {
        VK_SUBPASS_EXTERNAL,                            // uint32_t
srcSubpass
        0,                                              // uint32_t
dstSubpass
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,           // VkPipelineStageFlags
srcStageMask
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,  // VkPipelineStageFlags
dstStageMask
        VK_ACCESS_MEMORY_READ_BIT,                      // VkAccessFlags
srcAccessMask
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           // VkAccessFlags
dstAccessMask
        VK_DEPENDENCY_BY_REGION_BIT                     // VkDependencyFlags
dependencyFlags
```

```
    },
    {
      0,                                                  // uint32_t
srcSubpass
      VK_SUBPASS_EXTERNAL,                                // uint32_t
dstSubpass
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,  // VkPipelineStageFlags
srcStageMask
      VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,           // VkPipelineStageFlags
dstStageMask
      VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,           // VkAccessFlags
srcAccessMask
      VK_ACCESS_MEMORY_READ_BIT,                          // VkAccessFlags
dstAccessMask
      VK_DEPENDENCY_BY_REGION_BIT                         // VkDependencyFlags
dependencyFlags
    }
  };

  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
    static_cast<uint32_t>(dependencies.size()),       // uint32_t
dependencyCount
    &dependencies[0]                                    // const VkSubpassDependency
*pDependencies
  };

  if( vkCreateRenderPass( GetDevice(), &render_pass_create_info, nullptr,
&Vulkan.RenderPass ) != VK_SUCCESS ) {
    std::cout << "Could not create render pass!" << std::endl;
    return false;
  }
```

1. Tutorial04.cpp, function CreateRenderPass()

Subpass dependencies describe dependencies between different subpasses. When an attachment is used in one specific way in a given subpass (for example, rendering into it), but in another way in another subpass (sampling from it), we can create a memory barrier or we can provide a subpass dependency that describes the intended usage of an attachment in these two subpasses. Of course, the latter option is recommended, as the driver can (usually) prepare the barriers in a more optimal way. And the code itself is improved—everything required to understand the code is gathered in one place, one object.

In our simple example, we have only one subpass, but we specify two dependencies. This is because we can (and should) specify dependencies between render passes (by providing the number of a given subpass) and operations outside of them (by providing a VK_SUBPASS_EXTERNAL value). Here we provide one dependency for color attachment between operations occurring before a render pass and its only subpass. The second dependency is defined for operations occurring inside a subpass and after the render pass.

What operations are we talking about? We are using only one attachment, which is an image acquired from a presentation engine (swapchain). The presentation engine uses an image as a source of a presentable data. It only displays an image. So the only operation that involves this image is "memory read" on the image with "present src" layout. This operation doesn't occur in any normal pipeline stage, but it can be represented in the "bottom of pipeline" stage.

Inside our render pass, in its only subpass (with index 0), we are rendering into an image used as a color attachment. So the operation that occurs on this image is "color attachment write", which is performed in the "color attachment output" pipeline stage (after a fragment shader). After that the image is presented and returned to a presentation engine, which again uses this image as a source of data. So, in our example, the operation after the render pass is the same as before it: "memory read".

We specify this data through an array of VkSubpassDependency members. And when we create a render pass and a VkRenderPassCreateInfo structure, we specify the number of elements in the dependencies array (through dependencyCount member), and provide an address of its first element (through pDependencies). In a previous part of the tutorial we have provided 0 and nullptr for these two fields. VkSubpassDependency structure contains the following fields:

- srcSubpass – Index of a first (previous) subpass or VK_SUBPASS_EXTERNAL if we want to indicate dependency between subpass and operations outside of a render pass.
- dstSubpass – Index of a second (later) subpass (or VK_SUBPASS_EXTERNAL).
- srcStageMask – Pipeline stage during which a given attachment was used before (in a src subpass).
- dstStageMask – Pipeline stage during which a given attachment will be used later (in a dst subpass).
- srcAccessMask – Types of memory operations that occurred in a src subpass or before a render pass.
- dstAccessMask – Types of memory operations that occurred in a dst subpass or after a render pass.
- dependencyFlags – Flag describing the type (region) of dependency.

Graphics Pipeline Creation

Now we will create a graphics pipeline object. (We should create framebuffers for our swapchain images, but we will do that during command buffer recording). We don't want to render a geometry that is hardcoded into a shader. We want to draw any number of vertices, and we also want to provide additional attributes, not only vertex positions. What should we do first?

Writing Shaders

First have a look at the vertex shader written in GLSL code:

```
#version 450

layout(location = 0) in vec4 i_Position;
layout(location = 1) in vec4 i_Color;

out gl_PerVertex
{
  vec4 gl_Position;
};

layout(location = 0) out vec4 v_Color;

void main() {
    gl_Position = i_Position;
    v_Color = i_Color;
}
```

2. shader.vert

This shader is quite simple, though more complicated than the one from Tutorial 03.

We specify two input attributes (named i_Position and i_Color). In Vulkan, all attributes must have a location layout qualifier. When we specify a description of the vertex attributes in Vulkan API, the names of these attributes don't matter, only their indices/locations. In OpenGL* we could ask for a location of an attribute with a given name. In Vulkan we can't do this. Location layout qualifiers are the only way to go.

Next, we redeclare the gl_PerVertex block in the shader. Vulkan uses shader I/O blocks, and we should redeclare a gl_PerVertex block to specify exactly what members of this block to use. When we don't, the default definition is used. But we must remember that the default definition contains gl_ClipDistance[], which requires us to enable a feature named shaderClipDistance (and in Vulkan we can't use features that are not enabled during device creation or our application may not work correctly). Here we are using only a gl_Position member so the feature is not required.

We then specify an additional output varying variable called v_Color in which we store vertices' colors. Inside a main function we copy values provided by an application to proper output variables: position to gl_Position and color to v_Color.

Now look at a fragment shader to see how attributes are consumed.

```
#version 450

layout(location = 0) in vec4 v_Color;

layout(location = 0) out vec4 o_Color;

void main() {
  o_Color = v_Color;
}
```

3. shader.frag

In a fragment shader, the input varying variable v_Color is copied to the only output variable called o_Color. Both variables have location layout specifiers. The v_Color variable has the same location as the output variable in the vertex shader, so it will contain color values interpolated between vertices.

These shaders can be converted to a SPIR-V assembly the same way as previously. The following commands do this:

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

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

So now, when we know what attributes we want to use in our shaders, we can create the appropriate graphics pipeline.

Vertex Attributes Specification

The most important improvement in this tutorial is added to the vertex input state creation, for which we specify a variable of type VkPipelineVertexInputStateCreateInfo. In this variable we provide pointers to structures, which define the type of vertex input data and number and layout of our attributes.

We want to use two attributes: vertex positions, which are composed of four float components, and vertex colors, which are also composed of four float values. We will lay all of our vertex data in one buffer using the interleaved attributes layout. This means that position for the first vertex will be placed, next color for the same vertex, next the position of second vertex, after that the color of the second vertex, then position and color of third vertex and so on. All this specification is performed with the following code:

```
    std::vector<VkVertexInputBindingDescription> vertex_binding_descriptions = {
      {
        0,                                                          // uint32_t
binding
        sizeof(VertexData),                                         // uint32_t
stride
```

```
      VK_VERTEX_INPUT_RATE_VERTEX                                    // VkVertexInputRate
inputRate
    }
  };

  std::vector<VkVertexInputAttributeDescription> vertex_attribute_descriptions = {
    {
      0,                                                             // uint32_t
location
      vertex_binding_descriptions[0].binding,                        // uint32_t
binding
      VK_FORMAT_R32G32B32A32_SFLOAT,                                 // VkFormat
format
      offsetof(struct VertexData, x)                                 // uint32_t
offset
    },
    {
      1,                                                             // uint32_t
location
      vertex_binding_descriptions[0].binding,                        // uint32_t
binding
      VK_FORMAT_R32G32B32A32_SFLOAT,                                 // VkFormat
format
      offsetof( struct VertexData, r )                               // uint32_t
offset
    }
  };

  VkPipelineVertexInputStateCreateInfo vertex_input_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,       // VkStructureType
sType
    nullptr,                                                         // const void
*pNext
    0,                                                               //
VkPipelineVertexInputStateCreateFlags          flags;
    static_cast<uint32_t>(vertex_binding_descriptions.size()),       // uint32_t
vertexBindingDescriptionCount
    &vertex_binding_descriptions[0],                                 // const
VkVertexInputBindingDescription         *pVertexBindingDescriptions
    static_cast<uint32_t>(vertex_attribute_descriptions.size()),     // uint32_t
vertexAttributeDescriptionCount
    &vertex_attribute_descriptions[0]                                // const
VkVertexInputAttributeDescription       *pVertexAttributeDescriptions
  };
```

4. Tutorial04.cpp, function CreatePipeline()

First specify the binding (general memory information) of vertex data through VkVertexInputBindingDescription. It contains the following fields:

- binding – Index of a binding with which vertex data will be associated.
- stride – The distance in bytes between two consecutive elements (the same attribute for two neighbor vertices).
- inputRate – Defines how data should be consumed, per vertex or per instance.

The stride and inputRate fields are quite self-explanatory. Additional information may be required for a binding member. When we create a vertex buffer, we bind it to a chosen slot before rendering operations. The slot number (an index) is this binding and here we describe how data in this slot is aligned in memory and how it should be consumed (per vertex or per instance). Different vertex buffers can be bound to different bindings. And each binding may be differently positioned in memory.

Next step is to define all vertex attributes. We must specify a location (index) for each attribute (the same as in a shader source code, in location layout qualifier), source of data (binding from which data will be read), format (data type and number of components), and offset at which data for this specific attribute can be found (offset from the beginning of a data for a given vertex, not from the beginning of all vertex data). The situation here is exactly the same as in OpenGL where we created Vertex Buffer Objects (VBO, which can be thought of as an equivalent of "binding") and defined attributes using glVertexAttribPointer() function through which we specified an index of an attribute (location), size and type (number of components and format), stride and offset. This information is provided through the VkVertexInputAttributeDescription structure. It contains these fields:

- location – Index of an attribute, the same as defined by the location layout specifier in a shader source code.
- binding – The number of the slot from which data should be read (source of data like VBO in OpenGL), the same binding as in a VkVertexInputBindingDescription structure and **vkCmdBindVertexBuffers()** function (described later).
- format – Data type and number of components per attribute.
- offset – Beginning of data for a given attribute.

When we are ready, we can prepare vertex input state description by filling a variable of type VkPipelineVertexInputStateCreateInfo which consist of the following fields:

- sType – Type of structure, here it should be equal to VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO.
- pNext – Pointer reserved for extensions. Right now set this value to null.
- flags – Parameter reserved for future use.
- vertexBindingDescriptionCount – Number of elements in the pVertexBindingDescriptions array.
- pVertexBindingDescriptions – Array describing all bindings defined for a given pipeline (buffers from which values of all attributes are read).
- vertexAttributeDescriptionCount – Number of elements in the pVertexAttributeDescriptions array.
- pVertexAttributeDescriptions – Array with elements specifying all vertex attributes.

This concludes vertex attributes specification at pipeline creation. But to use them, we must create a vertex buffer and bind it to command buffer before we issue a rendering command.

Input Assembly State Specification

Previously we have drawn a single triangle using a triangle list topology. Now we will draw a quad, which is more convenient to draw by defining just four vertices, not two triangles and six vertices. To do this, we must use triangle strip topology. We define it through VkPipelineInputAssemblyStateCreateInfo structure that has the following members:

- sType – Structure type, here equal to VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO.
- pNext – Pointer reserved for extensions.
- flags – Parameter reserved for future use.
- topology – Topology used for drawing vertices (like triangle fan, strip, list).
- primitiveRestartEnable – Parameter defining whether we want to restart assembling a primitive by using a special value of vertex index.

Here is the code sample used to define triangle strip topology:

```
   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_STRIP,                           //
VkPrimitiveTopology                             topology
    VK_FALSE                                                        // VkBool32
primitiveRestartEnable
  };
```

5. Tutorial04.cpp, function CreatePipeline()

Viewport State Specification

In this tutorial we introduce another change. Previously, for the sake of simplicity, we have hardcoded the viewport and scissor test parameters, which unfortunately caused our image to be always the same size, no matter how big the application window was. This time, we won't specify these values through the VkPipelineViewportStateCreateInfo structure. We will use a dynamic state for that. Here is a code responsible for defining static viewport state parameters:

```
  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
    nullptr,                                                        // const VkViewport
*pViewports
    1,                                                              // uint32 t
scissorCount
    nullptr                                                         // const VkRect2D
*pScissors
  };
```

6. Tutorial04.cpp, function CreatePipeline()

The structure that defines static viewport parameters has the following members:

- sType – Type of the structure, VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO here.
- pNext – Pointer reserved for extension-specific parameters.
- flags – Parameter reserved for future use.
- viewportCount – Number of viewports.
- pViewports – Pointer to a structure defining static viewport parameters.
- scissorCount – Number of scissor rectangles (must have the same value as viewportCount parameter).
- pScissors – Pointer to an array of 2D rectangles defining static scissor test parameters for each viewport.

When we want to define viewport and scissor parameters through a dynamic state, we don't have to fill pViewports and pScissors members. That's why they are set to null in the example above. But, we always have to define the number of viewports and scissor test rectangles. These values are always specified through the VkPipelineViewportStateCreateInfo structure, no matter if we want to use dynamic or static viewport and scissor state.

Dynamic State Specification

When we create a pipeline, we can specify which parts of it are always static, defined through structures at a pipeline creation, and which are dynamic, specified by proper function calls during command buffer recording. This allows us to lower the number of pipeline objects that differ only with small details like line widths, blend constants, or stencil parameters, or mentioned viewport size. Here is the code used to define parts of pipeline that should be dynamic:

```
  std::vector<VkDynamicState> dynamic_states = {
    VK_DYNAMIC_STATE_VIEWPORT,
```

```
    VK_DYNAMIC_STATE_SCISSOR,
  };

  VkPipelineDynamicStateCreateInfo dynamic_state_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO,         // VkStructureType
sType
    nullptr,                                                      // const void
*pNext
    0,                                                            //
VkPipelineDynamicStateCreateFlags             flags
    static_cast<uint32_t>(dynamic_states.size()),                 // uint32_t
dynamicStateCount
    &dynamic_states[0]                                            // const
VkDynamicState                          *pDynamicStates
  };
```

7. Tutorial04.cpp, function CreatePipeline()

It is done by using a structure of type VkPipelineDynamicStateCreateInfo, which contains the following fields:

- sType – Parameter defining the type of a given structure, here equal to VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO.
- pNext – Parameter reserved for extensions.
- flags – Parameter reserved for future use.
- dynamicStateCount – Number of elements in pDynamicStates array.
- pDynamicStates – Array containing enums, specifying which parts of a pipeline should be marked as dynamic. Each element of this array is of type VkDynamicState.

Pipeline Object Creation

We now have defined all the necessary parameters of a graphics pipeline, so we can create a pipeline object. Here is the code that does it:

```
  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
```

```
    &dynamic_state_create_info,                                    // 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 ) {
    std::cout << "Could not create graphics pipeline!" << std::endl;
    return false;
  }
  return true;
```

8. Tutorial04.cpp, function CreatePipeline()

The most important variable, which contains references to all pipeline parameters, is of type VkGraphicsPipelineCreateInfo. The only change from the previous tutorial is an addition of the pDynamicState parameter, which points to a structure of VkPipelineDynamicStateCreateInfo type, described above. Every pipeline state, which is specified as dynamic, must be set through a proper function call during command buffer recording.

The pipeline object itself is created by calling the **vkCreateGraphicsPipelines()** function.

Vertex Buffer Creation

To use vertex attributes, apart from specifying them during pipeline creation, we need to prepare a buffer that will contain all the data for these attributes. From this buffer, the values for attributes will be read and provided to the vertex shader.

In Vulkan, buffer and image creation consists of at least two stages. First, we create the object itself. Next, we need to create a memory object, which will then be bound to the buffer (or image). From this memory object, the buffer will take its storage space. This approach allows us to specify additional parameters for the memory and control it with more details.

To create a (general) buffer object we call **vkCreateBuffer()**. It accepts, among other parameters, a pointer to a variable of type VkBufferCreateInfo, which defines parameters of created buffer. Here is the code responsible for creating a buffer used as a source of data for vertex attributes:

```
  VertexData vertex_data[] = {
    {
      -0.7f, -0.7f, 0.0f, 1.0f,
      1.0f, 0.0f, 0.0f, 0.0f
    },
    {
      -0.7f, 0.7f, 0.0f, 1.0f,
      0.0f, 1.0f, 0.0f, 0.0f
    },
    {
      0.7f, -0.7f, 0.0f, 1.0f,
      0.0f, 0.0f, 1.0f, 0.0f
    },
    {
      0.7f, 0.7f, 0.0f, 1.0f,
```

```
      0.3f, 0.3f, 0.3f, 0.0f
    }
  };

  Vulkan.VertexBuffer.Size = sizeof(vertex_data);

  VkBufferCreateInfo buffer_create_info = {
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO,              // VkStructureType        sType
    nullptr,                                           // const void            *pNext
    0,                                                 // VkBufferCreateFlags    flags
    Vulkan.VertexBuffer.Size,                          // VkDeviceSize           size
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,                 // VkBufferUsageFlags     usage
    VK_SHARING_MODE_EXCLUSIVE,                         // VkSharingMode
sharingMode
    0,                                                 // uint32_t
queueFamilyIndexCount
    nullptr                                            // const uint32_t
*pQueueFamilyIndices
  };

  if( vkCreateBuffer( GetDevice(), &buffer_create_info, nullptr,
&Vulkan.VertexBuffer.Handle ) != VK_SUCCESS ) {
    std::cout << "Could not create a vertex buffer!" << std::endl;
    return false;
  }
```

9. Tutorial04.cpp, function CreateVertexBuffer()

At the beginning of the CreateVertexBuffer() function we define a set of values for position and color attributes. First, four position components are defined for first vertex, next four color components for the same vertex, after that four components of a position attribute for second vertex are specified, next a color values for the same vertex, after that position and color for third and fourth vertices. The size of this array is used to define the size of a buffer. Remember though that internally graphics driver may require more storage for a buffer than the size requested by an application.

Next we define a variable of VkBufferCreateInfo type. It is a structure with the following fields:

- sType – Type of the structure, which should be set to VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO value.
- pNext – Parameter reserved for extensions.
- flags – Parameter defining additional creation parameters. Right now it allows creation of a buffer backed by a sparse memory (something similar to a mega texture). As we don't want to use sparse memory, we can set this parameter to zero.
- size – Size, in bytes, of a buffer.
- usage – This parameter defines how we intend to use this buffer in future. We can specify that we want to use buffer as a uniform buffer, index buffer, source of data for transfer (copy) operations, and so on. Here we intend to use this buffer as a vertex buffer. Remember that we can't use a buffer for a purpose that is not defined during buffer creation.
- sharingMode – Sharing mode, similarly to swapchain images, defines whether a given buffer can be accessed by multiple queues at the same time (concurrent sharing mode) or by just a single queue (exclusive sharing mode). If a concurrent sharing mode is specified, we must provide indices of all queues that will have access to a buffer. If we want to define an exclusive sharing mode, we can still reference this buffer in different queues, but only in one at a time. If we want to use a buffer in a different queue (submit commands that reference this buffer to another queue), we need to specify buffer memory barrier that transitions buffer's ownership from one queue to another.
- queueFamilyIndexCount – Number of queue indices in pQueueFamilyIndices array (only when concurrent sharing mode is specified).

- pQueueFamilyIndices – Array with indices of all queues that will reference buffer (only when concurrent sharing mode is specified).

To create a buffer we must call **vkCreateBuffer()** function.

Buffer Memory Allocation

We next create a memory object that will back the buffer's storage.

```
  VkMemoryRequirements buffer_memory_requirements;
  vkGetBufferMemoryRequirements( GetDevice(), buffer, &buffer_memory_requirements );

  VkPhysicalDeviceMemoryProperties memory_properties;
  vkGetPhysicalDeviceMemoryProperties( GetPhysicalDevice(), &memory_properties );

  for( uint32_t i = 0; i < memory_properties.memoryTypeCount; ++i ) {
    if( (buffer_memory_requirements.memoryTypeBits & (1 << i)) &&
      (memory_properties.memoryTypes[i].propertyFlags &
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT) ) {

      VkMemoryAllocateInfo memory_allocate_info = {
        VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,      // VkStructureType
sType
        nullptr,                                     // const void
*pNext
        buffer_memory_requirements.size,             // VkDeviceSize
allocationSize
        i                                            // uint32_t
memoryTypeIndex
      };

      if( vkAllocateMemory( GetDevice(), &memory_allocate_info, nullptr, memory ) ==
VK_SUCCESS ) {
        return true;
      }
    }
  }
  return false;
```

10. Tutorial04.cpp, function AllocateBufferMemory()

First we must check what the memory requirements for a created buffer are. We do this by calling the **vkGetBufferMemoryRequirements()** function. It stores parameters for memory creation in a variable that we provided the address of in the last parameter. This variable must be of type VkMemoryRequirements and it contains information about required size, memory alignment, and supported memory types. What are memory types?

Each device may have and expose different memory types—heaps of various sizes that have different properties. One memory type may be a device's local memory located on the GDDR chips (thus very, very fast). Another may be a shared memory that is visible both for a graphics card and a CPU. Both the graphics card and application may have access to this memory, but such memory type is slower than the device local-only memory (which is accessible only to a graphics card).

To check what memory heaps and types are available, we need to call the **vkGetPhysicalDeviceMemoryProperties()** function, which stores information about memory in a variable of type VkPhysicalDeviceMemoryProperties. It contains the following information:

- memoryHeapCount – Number of memory heaps exposed by a given device.
- memoryHeaps – An array of memory heaps. Each heap represents a memory of different size and properties.
- memoryTypeCount – Number of different memory types exposed by a given device.

- memoryTypes – An array of memory types. Each element describes specific memory properties and contains an index of a heap that has these particular properties.

Before we can allocate a memory for a given buffer, we need to check which memory type fulfills a buffer's memory requirements. If we have additional, specific needs, we can also check them. For all of this, we iterate over all available memory types. Buffer memory requirements have a field called memoryTypeBits and if a bit on a given index is set in this field, it means that for a given buffer we can allocate a memory of the type represented by that index. But we must remember that while there must always be a memory type that fulfills buffer's memory requirements, it may not support some other, specific needs. In this case we need to look for another memory type or change our additional requirements.

Here, our additional requirement is that memory needs to be host visible. This means that application can map this memory and get access to it—read it or write data to it. Such memory is usually slower than the device local-only memory, but this way we can easily upload data for our vertex attributes. The next tutorial will show how to use device local-only memory for better performance.

Fortunately, the host visible requirement is popular, and it should be easy to find a memory type that supports both the buffer's memory requirements and the host visible property. We then prepare a variable of type VkMemoryAllocateInfo and fill all its fields:

- sType – Type of the structure, here set to VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO.
- pNext – Pointer reserved for extensions.
- allocationSize – Minimum required memory size that should be allocated.
- memoryTypeIndex – Index of a memory type we want to use for a created memory object. It is the index of one of bits that are set (has value of one) in buffer's memory requirement.

After we fill such a structure we call **vkAllocateMemory()** and check whether the memory object allocation succeeded.

Binding a Buffer's Memory

When we are done creating a memory object, we must bind it to our buffer. Without it, there will be no storage space in a buffer and we won't be able to store any data in it.

```
    if( !AllocateBufferMemory( Vulkan.VertexBuffer.Handle, &Vulkan.VertexBuffer.Memory )
) {
      std::cout << "Could not allocate memory for a vertex buffer!" << std::endl;
      return false;
    }

    if( vkBindBufferMemory( GetDevice(), Vulkan.VertexBuffer.Handle,
Vulkan.VertexBuffer.Memory, 0 ) != VK_SUCCESS ) {
      std::cout << "Could not bind memory for a vertex buffer!" << std::endl;
      return false;
    }
```

11. Tutorial04.cpp, function CreateVertexBuffer()

AllocateBufferMemory() is a function that allocates a memory object. It was presented earlier. When a memory object is created, we bind it to the buffer by calling the **vkBindBufferMemory()** function. During the call we must specify a handle to a buffer, handle to a memory object, and an offset. Offset is very important and requires some additional explanation.

When we queried for buffer memory requirement, we acquired information about required size, memory type, and alignment. Different buffer usages may require different memory alignment. The beginning of a memory object (offset of 0) satisfies all alignments. This means that all memory objects are created at addresses that fulfill the requirements of all different usages. So when we specify a zero offset, we don't have to worry about anything.

But we can create larger memory object and use it as a storage space for multiple buffers (or images). This, in fact, is the recommended behavior. Creating larger memory objects means we are creating fewer memory objects. This allows

driver to track fewer objects in general. Memory objects must be tracked by a driver because of OS requirements and security measures. Larger memory objects don't cause big problems with memory fragmentation. Finally, we should allocate larger memory amounts and keep similar objects in them to increase cache hits and thus improve performance of our application.

But when we allocate larger memory objects and bind them to multiple buffers (or images), not all of them can be bound at offset zero. Only one can be bound at this offset, others must be bound further away, after a space used by the first buffer (or image). So the offset for the second, and all other buffers bound to the same memory object, must meet alignment requirements reported by the query. And we must remember it. That's why alignment member is important.

When our buffer is created and memory for it is allocated and bound, we can fill the buffer with data for vertex attributes.

Uploading Vertex Data

We have created a buffer and we have bound a memory that is host visible. This means we can map this memory, acquire a pointer to this memory, and use this pointer to copy data from our application to the buffer itself (similar to the OpenGL's glBufferData() function):

```
   void *vertex_buffer_memory_pointer;
   if( vkMapMemory( GetDevice(), Vulkan.VertexBuffer.Memory, 0,
Vulkan.VertexBuffer.Size, 0, &vertex_buffer_memory_pointer ) != VK_SUCCESS ) {
     std::cout << "Could not map memory and upload data to a vertex buffer!" <<
std::endl;
     return false;
   }

   memcpy( vertex_buffer_memory_pointer, vertex_data, Vulkan.VertexBuffer.Size );

   VkMappedMemoryRange flush_range = {
     VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE,           // VkStructureType        sType
     nullptr,                                         // const void            *pNext
     Vulkan.VertexBuffer.Memory,                      // VkDeviceMemory         memory
     0,                                               // VkDeviceSize           offset
     VK_WHOLE_SIZE                                    // VkDeviceSize           size
   };
   vkFlushMappedMemoryRanges( GetDevice(), 1, &flush_range );

   vkUnmapMemory( GetDevice(), Vulkan.VertexBuffer.Memory );

   return true;
```

12. Tutorial04.cpp, function CreateVertexBuffer()

To map memory, we call the **vkMapMemory()** function. In the call we must specify which memory object we want to map and a region to access. Region is defined by an offset from the beginning of a memory object's storage and size. After the successful call we acquire a pointer. We can use it to copy data from our application to the provided memory address. Here we copy vertex data from an array with vertex positions and colors.

After a memory copy operation and before we unmap a memory (we don't need to unmap it, we can keep a pointer and this shouldn't impact performance), we need to tell the driver which parts of the memory was modified by our operations. This operation is called flushing. Through it we specify all memory ranges that our application copied data to. Ranges don't have to be continuous. Ranges are defined by an array of VkMappedMemoryRange elements which contain these fields:

- sType – Structure type, here equal to VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE.
- pNext – Pointer reserved for extensions.
- memory – Handle of a mapped and modified memory object.

- offset – Offset (from the beginning of a given memory object's storage) at which a given range starts.
- size – Size, in bytes, of an affected region. If the whole memory, from an offset to the end, was modified, we can use the special value of VK_WHOLE_SIZE.

When we define all memory ranges that should be flashed, we can call the **vkFlushMappedMemoryRanges()** function. After that, the driver will know which parts were modified and will reload them (that is, refresh cache). Reloading usually occurs on barriers. After modifying a buffer, we should set a buffer memory barrier, which will tell the driver that some operations influenced a buffer and it should be refreshed. But, fortunately, in this case such a barrier is placed implicitly by the driver on a submission of a command buffer that references the given buffer and no additional operations are required. Now we can use this buffer during rendering commands recording.

Rendering Resources Creation

We now must prepare resources required for a command buffer recording. In previous tutorials we have recorded one static command buffer for each swapchain image. Here we will reorganize the rendering code. We will still display a simple, static scene, but the approach presented here is useful in real-life scenarios, where displayed scenes are dynamic.

To record command buffers and submit them to queue in an efficient way, we need four types of resources: command buffers, semaphores, fences and framebuffers. Semaphores, as we already discussed, are used for internal queue synchronization. Fences, on the other hand, allow the application to check if some specific situation occurred, e.g. if command buffer's execution after it was submitted to queue, has finished. If necessary, application can wait on a fence, until it is signaled. In general, semaphores are used to synchronize queues (GPU) and fences are used to synchronize application (CPU).

To render a single frame of animation we need (at least) one command buffer, two semaphores—one for a swapchain image acquisition (image available semaphore) and the other to signal that presentation may occur (rendering a finished semaphore)—a fence, and a framebuffer. The fence is used later to check whether we can rerecord a given command buffer. We will keep several numbers of such rendering resources, which we can call a *virtual frame*. The number of these *virtual frames* (consisting of a command buffer, two semaphores, a fence, and a framebuffer) should be independent of a number of swapchain images.

The rendering algorithm progresses like this: We record rendering commands to the first *virtual frame* and then submit it to a queue. Next we record another frame (command buffer) and submit it to queue. We do this until we are out of all *virtual frames*. At this point we will start reusing frames by taking the oldest (least recently submitted) command buffer and rerecording it again. Then we will use another command buffer, and so on.

This is where the fences come in. We are not allowed to record a command buffer that has been submitted to a queue until its execution in the queue is finished. During command buffer recording, we can use the "simultaneous use" flag, which allows us to record or resubmit a command buffer that has already been submitted. This may impact performance though. A better way is to use fences and check whether a command buffer is not used any more. If a graphics card is still processing a command buffer, we can wait on a fence associated with a given command buffer, or use this additional time for other purposes, like improved AI calculations, and after some time check again to see whether a fence is signaled.

How many *virtual frames* should we have? One is not enough. When we record and submit a single command buffer, we immediately wait until we can rerecord it. It is a waste of time of both the CPU and the GPU. The GPU is usually faster, so waiting on a CPU causes more waiting on a GPU. We should keep the GPU as busy as possible. That is why thin APIs like Vulkan were created. Using two *virtual frames* gives huge performance gain, as there is much less waiting both on the CPU and the GPU. Adding a third *virtual frame* gives additional performance gain, but the increase isn't as big. Using four or more groups of rendering resource doesn't make sense, as the performance gain is negligible (of course this may depend on the complexity of the rendered scene and calculations performed by the CPU-like physics or AI). When we increase the number of *virtual frames* we also increase the input lag, as we present a frame that's one to three frames behind the CPU. So two or three *virtual frames* seems to be the most reasonable compromise between performance, memory usage, and input lag.

You may wonder why the number of *virtual frames* shouldn't be connected with the number of swapchain images. This approach may influence the behavior of our application. When we create a swapchain, we ask for the minimal required number of images, but the driver is allowed to create more. So different hardware vendors may implement drivers that offer different numbers of swapchain images, even for the same requirements (present mode and minimal number of images). When we connect the number of *virtual frames* with a number of swapchain images, our application will use only two *virtual frames* on one graphics card, but four *virtual frames* on another graphics card. This may influence both performance and mentioned input lag. It's not a desired behavior. By keeping the number of *virtual frames* fixed, we can control our rendering algorithm and fine-tune it to our needs, that is, balance the time spent on rendering and AI or physics calculations.

Command Pool Creation

Before we can allocate a command buffer, we first need to create a command pool.

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

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

13. Tutorial04.cpp, function CreateCommandPool()

The command pool is created by calling **vkCreateCommandPool()**, which requires us to provide a pointer to a variable of type VkCommandPoolCreateInfo. The code remains mostly unchanged, compared to previous tutorials. But this time, two additional flags are added for command pool creation:

- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT – Indicates that command buffers, allocated from this pool, may be reset individually. Normally, without this flag, we can't rerecord the same command buffer multiple times. It must be reset first. And, what's more, command buffers created from one pool may be reset only all at once. Specifying this flag allows us to reset command buffers individually, and (even better) it is done implicitly by calling the **vkBeginCommandBuffer()** function.
- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT – This flag tells the driver that command buffers allocated from this pool will be living for a short amount of time, they will be often recorded and reset (re-recorded). This information helps optimize command buffer allocation and perform it more optimally.

Command Buffer Allocation

Allocating command buffers remains the same as previously.

```
  for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if( !AllocateCommandBuffers( Vulkan.CommandPool, 1,
&Vulkan.RenderingResources[i].CommandBuffer ) ) {
      std::cout << "Could not allocate command buffer!" << std::endl;
      return false;
    }
  }
  return true;
```

14. Tutorial04.cpp, function CreateCommandBuffers()

The only change is that command buffers are gathered into a vector of rendering resources. Each rendering resource structure contains a command buffer, image available semaphore, rendering finished semaphore, a fence and a framebuffer. Command buffers are allocated in a loop. The number of elements in a rendering resources vector is chosen arbitrarily. For this tutorial it is equal to three.

Semaphore Creation

The code responsible for creating a semaphore is simple and the same as previously shown:

```
  VkSemaphoreCreateInfo semaphore_create_info = {
    VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,      // VkStructureType          sType
    nullptr,                                      // const void*              pNext
    0                                             // VkSemaphoreCreateFlags   flags
  };

  for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if( (vkCreateSemaphore( GetDevice(), &semaphore_create_info, nullptr,
&Vulkan.RenderingResources[i].ImageAvailableSemaphore ) != VK_SUCCESS) ||
      (vkCreateSemaphore( GetDevice(), &semaphore_create_info, nullptr,
&Vulkan.RenderingResources[i].FinishedRenderingSemaphore ) != VK_SUCCESS) ) {
      std::cout << "Could not create semaphores!" << std::endl;
      return false;
    }
  }
  return true;
```

15. Tutorial04.cpp, function CreateSemaphores()

Fence Creation

Here is the code responsible for creating fence objects:

```
  VkFenceCreateInfo fence_create_info = {
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO,             // VkStructureType
sType
    nullptr,                                         // const void
*pNext
    VK_FENCE_CREATE_SIGNALED_BIT                     // VkFenceCreateFlags
flags
  };

  for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
    if( vkCreateFence( GetDevice(), &fence_create_info, nullptr,
&Vulkan.RenderingResources[i].Fence ) != VK_SUCCESS ) {
      std::cout << "Could not create a fence!" << std::endl;
      return false;
    }
  }
  return true;
```

16. Tutorial04.cpp, function CreateFences()

To create a fence object we call the **vkCreateFence()** function. It accepts, among other parameters, a pointer to a variable of type VkFenceCreateInfo, which has the following members:

- sType – Type of the structure. Here it should be set to VK_STRUCTURE_TYPE_FENCE_CREATE_INFO.
- pNext – Pointer reserved for extensions.
- flags – Right now this parameter allows for creating a fence that is already signaled.

A fence may have two states: signaled and unsignaled. The application checks whether a given fence is in a signaled state, or it may wait on a fence until the fence gets signaled. Signaling is done by the GPU after all operations submitted to the queue are processed. When we submit command buffers, we can provide a fence that will be signaled when a queue has finished executing all commands that were issued in this one submit operation. After the fence is signaled, it is the application's responsibility to reset it to an unsignaled state.

Why create a fence that is already signaled? Our rendering algorithm will record commands to the first command buffer, then to the second command buffer, after that to the third, and then once again to the first (after its execution in a queue has ended). We use fences to check whether we can record a given command buffer once again. But what about the first recording? We don't want to keep separate code paths for the first command buffer recording and for the following recording operations. So when we issue a command buffer recording for the first time, we also check whether a fence is already signaled. But because we didn't submit a given command buffer, the fence associated with it can't become signaled as a result of the finished execution. So the fence needs to be created in an already signaled state. This way, for the first time, we won't have to wait for it to become signaled (as it is already signaled), but after the check we will reset it and immediately go to the recording code. After that we submit a command buffer and provide the same fence, which will get signaled by the queue when operations are done. The next time, when we want to rerecord rendering commands to the same command buffer, we can do the same operations: wait on the fence, reset it, and then start command buffer recording.

Drawing

Now we are nearly ready to record rendering operations. We are recording each command buffer just before it is submitted to the queue. We record one command buffer and submit it, then the next command buffer and submit it, then yet another one. After that we take the first command buffer, check whether we can use it, and we record it and submit it to the queue.

```
   static size_t           resource_index = 0;
   RenderingResourcesData &current_rendering_resource =
Vulkan.RenderingResources[resource_index];
   VkSwapchainKHR          swap_chain = GetSwapChain().Handle;
   uint32_t                image_index;

   resource_index = (resource_index + 1) % VulkanTutorial04Parameters::ResourcesCount;

   if( vkWaitForFences( GetDevice(), 1, &current_rendering_resource.Fence, VK_FALSE,
1000000000 ) != VK_SUCCESS ) {
     std::cout << "Waiting for fence takes too long!" << std::endl;
     return false;
   }
   vkResetFences( GetDevice(), 1, &current_rendering_resource.Fence );

   VkResult result = vkAcquireNextImageKHR( GetDevice(), swap_chain, UINT64_MAX,
current_rendering_resource.ImageAvailableSemaphore, 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:
       std::cout << "Problem occurred during swap chain image acquisition!" <<
std::endl;
       return false;
   }

   if( !PrepareFrame( current_rendering_resource.CommandBuffer,
GetSwapChain().Images[image_index], current_rendering_resource.Framebuffer ) ) {
     return false;
   }
```

```
   VkPipelineStageFlags wait_dst_stage_mask =
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
   VkSubmitInfo submit_info = {
     VK_STRUCTURE_TYPE_SUBMIT_INFO,                           // VkStructureType
sType
     nullptr,                                                 // const void
*pNext
     1,                                                       // uint32_t
waitSemaphoreCount
     &current_rendering_resource.ImageAvailableSemaphore,     // const VkSemaphore
*pWaitSemaphores
     &wait_dst_stage_mask,                                    // const
VkPipelineStageFlags  *pWaitDstStageMask;
     1,                                                       // uint32_t
commandBufferCount
     &current_rendering_resource.CommandBuffer,               // const VkCommandBuffer
*pCommandBuffers
     1,                                                       // uint32_t
signalSemaphoreCount
     &current_rendering_resource.FinishedRenderingSemaphore  // const VkSemaphore
*pSignalSemaphores
   };

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

   VkPresentInfoKHR present_info = {
     VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,                      // VkStructureType
sType
     nullptr,                                                 // const void
*pNext
     1,                                                       // uint32_t
waitSemaphoreCount
     &current_rendering_resource.FinishedRenderingSemaphore, // 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:
       std::cout << "Problem occurred during image presentation!" << std::endl;
       return false;
   }

   return true;
```

17. Tutorial04.cpp, function Draw()

So first we take the least recently used rendering resource. Then we wait until the fence associated with this group is signaled. If it is, this means that we can safely take a command buffer and record it. But this also means that we can take semaphores used to acquire and present an image that was referenced in a given command buffer. We shouldn't use the same semaphore for different purposes or in two different submit operations, until the previous submission is finished. The fences prevent us from altering both command buffers and semaphores. And as you will soon see, framebuffers too.

When a fence is finished, we reset the fence and perform normal drawing-related operations: we acquire an image, record operations rendering into an acquired image, submit the command buffer, and present an image.

After that we take another set of rendering resources and perform these same operations. Thanks to keeping three groups of rendering resources, three *virtual frames*, we lower the time wasted on waiting for a fence to be signaled.

Recording a Command Buffer

A function responsible for recording a command buffer is quite long. This time it is even longer, because we use a vertex buffer and a dynamic viewport and scissor test. And we also create temporary framebuffers!

Framebuffer creation is simple and fast. Keeping framebuffer objects along with a swapchain means that we need to recreate them when the swapchain needs to be recreated. If our rendering algorithm is complicated, we have multiple images and framebuffers associated with them. If those images need to have the same size as swapchain images, we need to recreate all of them (to include potential size change). So it is better and more convenient to create framebuffers on demand. This way, they always have the desired size. Framebuffers operate on image views, which are created for a given, specific image. When a swapchain is recreated, old images are invalid, not existent. So we must recreate image views and also framebuffers.

In the "03 – First Triangle" tutorial, we had framebuffers of a fixed size and they had to be recreated along with a swapchain. Now we have a framebuffer object in each of our *virtual frame* group of resources. Before we record a command buffer, we create a framebuffer for an image to which we will be rendering, and of the same size as that image. This way, when swapchain is recreated, the size of the next frame will be immediately adjusted and a handle of the new swapchain's image and its image view will be used to create a framebuffer.

When we record a command buffer that uses a render pass and framebuffer objects, the framebuffer must remain valid for the whole time the command buffer is processed by the queue. When we create a new framebuffer, we can't destroy it until commands submitted to a queue are finished. But as we are using fences, and we have already waited on a fence associated with a given command buffer, we are sure that the framebuffer can be safely destroyed. We then create a new framebuffer to include potential size and image handle changes.

```
    if( framebuffer != VK_NULL_HANDLE ) {
      vkDestroyFramebuffer( GetDevice(), framebuffer, nullptr );
    }

    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
      &image_view,                                    // const VkImageView
*pAttachments
      GetSwapChain().Extent.width,                    // uint32_t
width
```

```
    GetSwapChain().Extent.height,                   // uint32_t
height
    1                                               // uint32_t
layers
  };

  if( vkCreateFramebuffer( GetDevice(), &framebuffer_create_info, nullptr, &framebuffer
) != VK_SUCCESS ) {
    std::cout << "Could not create a framebuffer!" << std::endl;
    return false;
  }

  return true;
```

18. Tutorial04.cpp, function CreateFramebuffer()

When we create a framebuffer, we take current swapchain extents and image view for an acquired swapchain image.

Next we start recording a command buffer:

```
  if( !CreateFramebuffer( framebuffer, image_parameters.View ) ) {
    return false;
  }

  VkCommandBufferBeginInfo command_buffer_begin_info = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,        // VkStructureType
sType
    nullptr,                                            // const void
*pNext
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,        // VkCommandBufferUsageFlags
flags
    nullptr                                             // const
VkCommandBufferInheritanceInfo  *pInheritanceInfo
  };

  vkBeginCommandBuffer( command_buffer, &command_buffer_begin_info );

  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
  };

  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_MEMORY_READ_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
      image_parameters.Handle,                          // VkImage
image
      image_subresource_range                           // VkImageSubresourceRange
subresourceRange
    };
    vkCmdPipelineBarrier( command_buffer,
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 );
  }

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

  VkRenderPassBeginInfo render_pass_begin_info = {
    VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,           // VkStructureType
sType
    nullptr,                                            // const void
*pNext
    Vulkan.RenderPass,                                  // VkRenderPass
renderPass
    framebuffer,                                        // VkFramebuffer
framebuffer
    {                                                   // VkRect2D
renderArea
      {                                                 // VkOffset2D
offset
        0,                                                // int32_t
x
        0                                                 // int32_t
y
      },
      GetSwapChain().Extent,                            // VkExtent2D
extent;
    },
    1,                                                  // uint32_t
clearValueCount
    &clear_value                                        // const VkClearValue
*pClearValues
  };

  vkCmdBeginRenderPass( command_buffer, &render_pass_begin_info,
VK_SUBPASS_CONTENTS_INLINE );
```

19. Tutorial04.cpp, function PrepareFrame()

First we define a variable of type VkCommandBufferBeginInfo and specify that a command buffer will be submitted only once. When we specify a VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT flag, we can't submit a given command buffer more times. After each submission it must be reset. But the recording operation resets it due to the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag used during command pool creation.

Next we define subresource ranges for image memory barriers. The layout transitions of the swapchain images are performed implicitly inside a render pass, but if the graphics and presentation queue are different, the queue transition must be manually performed.

After that we begin a render pass with the temporary framebuffer object.

```
   vkCmdBindPipeline( command_buffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
Vulkan.GraphicsPipeline );

   VkViewport viewport = {
     0.0f,                                                 // float
x
     0.0f,                                                 // float
y
     static_cast<float>(GetSwapChain().Extent.width),      // float
width
     static_cast<float>(GetSwapChain().Extent.height),     // float
height
     0.0f,                                                 // float
minDepth
     1.0f                                                  // float
maxDepth
   };

   VkRect2D scissor = {
     {                                                     // VkOffset2D
offset
       0,                                                    // int32_t
x
       0                                                     // int32_t
y
     },
     {                                                     // VkExtent2D
extent
       GetSwapChain().Extent.width,                          // uint32_t
width
       GetSwapChain().Extent.height                          // uint32_t
height
     }
   };

   vkCmdSetViewport( command_buffer, 0, 1, &viewport );
   vkCmdSetScissor( command_buffer, 0, 1, &scissor );

   VkDeviceSize offset = 0;
   vkCmdBindVertexBuffers( command_buffer, 0, 1, &Vulkan.VertexBuffer.Handle, &offset );

   vkCmdDraw( command_buffer, 4, 1, 0, 0 );

   vkCmdEndRenderPass( command_buffer );

   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_MEMORY_READ_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
        image_parameters.Handle,                            // VkImage
image
        image_subresource_range                             // VkImageSubresourceRange
subresourceRange
      };
      vkCmdPipelineBarrier( command_buffer,
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( command_buffer ) != VK_SUCCESS ) {
      std::cout << "Could not record command buffer!" << std::endl;
      return false;
    }
    return true;
```

20. Tutorial04.cpp, function PrepareFrame()

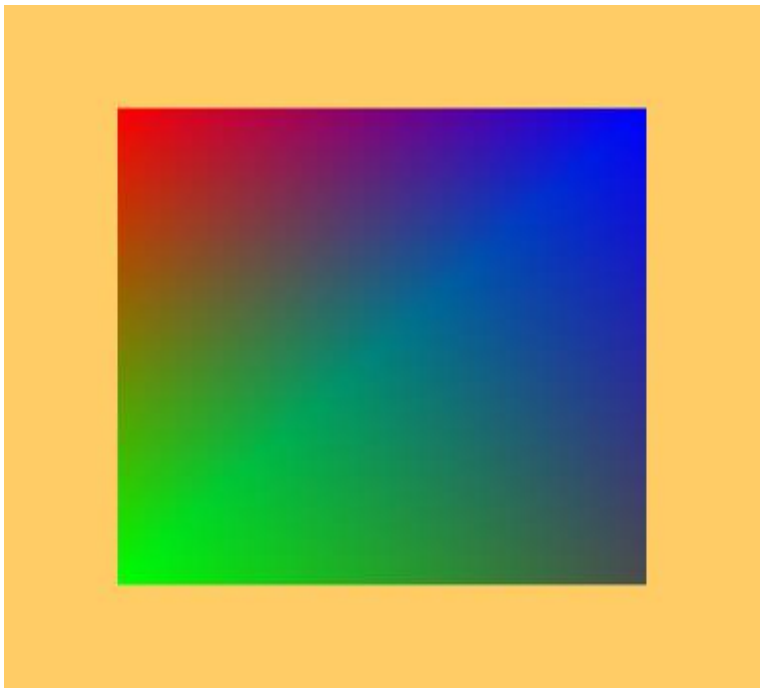
Next we bind a graphics pipeline. It has two states marked as dynamic: viewport and scissor test. So we prepare structures that define viewport and scissor test parameters. The dynamic viewport state is set by calling the **vkCmdSetViewport()** function. The dynamic scissor test is set by calling the **vkCmdSetScissor()** function. This way, our graphics pipeline can be used for rendering into images of different sizes.

One last thing before we can draw anything is to bind appropriate vertex buffer, providing buffer data for vertex attributes. We do this through the **vkCmdBindVertexBuffers()** function call. We specify a binding number (which set of vertex attributes should take data from this buffer), a pointer to a buffer handle (or more handles if we want to bind buffers for multiple bindings) and an offset. The offset specifies that data for vertex attributes should be taken from further parts of the buffer. But we can't specify offset larger than the size of a corresponding buffer (buffer, not memory object bound to this buffer).

Now we have specified all the required elements: framebuffer, viewport and scissor test, and a vertex buffer. We can draw the geometry, finish the render pass, and end the command buffer.

Tutorial04 Execution

Here is the result of rendering operations:



We are rendering a quad that has different colors in each corner. Try resizing the window; previously, the triangle was always the same size, only the black frame on the right and bottom sides of an application window grew larger or smaller. Now, thanks to the dynamic viewport state, the quad is growing or shrinking along with the window.

Cleaning Up

After rendering and before closing the application, we should destroy all resources. Here is a code responsible for this operation:

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

      for( size_t i = 0; i < Vulkan.RenderingResources.size(); ++i ) {
        if( Vulkan.RenderingResources[i].Framebuffer != VK_NULL_HANDLE ) {
          vkDestroyFramebuffer( GetDevice(), Vulkan.RenderingResources[i].Framebuffer,
nullptr );
        }
        if( Vulkan.RenderingResources[i].CommandBuffer != VK_NULL_HANDLE ) {
          vkFreeCommandBuffers( GetDevice(), Vulkan.CommandPool, 1,
&Vulkan.RenderingResources[i].CommandBuffer );
        }
        if( Vulkan.RenderingResources[i].ImageAvailableSemaphore != VK_NULL_HANDLE ) {
          vkDestroySemaphore( GetDevice(),
Vulkan.RenderingResources[i].ImageAvailableSemaphore, nullptr );
        }
        if( Vulkan.RenderingResources[i].FinishedRenderingSemaphore != VK_NULL_HANDLE ) {
          vkDestroySemaphore( GetDevice(),
Vulkan.RenderingResources[i].FinishedRenderingSemaphore, nullptr );
        }
        if( Vulkan.RenderingResources[i].Fence != VK_NULL_HANDLE ) {
          vkDestroyFence( GetDevice(), Vulkan.RenderingResources[i].Fence, nullptr );
        }
      }

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

```
    if( Vulkan.VertexBuffer.Handle != VK_NULL_HANDLE ) {
      vkDestroyBuffer( GetDevice(), Vulkan.VertexBuffer.Handle, nullptr );
      Vulkan.VertexBuffer.Handle = VK_NULL_HANDLE;
    }

    if( Vulkan.VertexBuffer.Memory != VK_NULL_HANDLE ) {
      vkFreeMemory( GetDevice(), Vulkan.VertexBuffer.Memory, nullptr );
      Vulkan.VertexBuffer.Memory = 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;
    }
  }
```

21. Tutorial04.cpp, function ChildClear()

We destroy all resources after the device completes processing all commands submitted to all its queues. We destroy resources in a reverse order. First we destroy all rendering resources: framebuffers, command buffers, semaphores and fences. Fences are destroyed by calling the **vkDestroyFence()** function. Then the command pool is destroyed. After that we destroy buffer by calling the **vkDestroyBuffer()** function, and free memory object by calling the **vkFreeMemory()** function. Finally the pipeline object and a render pass are destroyed.

Conclusion

This tutorial is based on the"03 – First Triangle" tutorial. We improved rendering by using vertex attributes in a graphics pipeline and vertex buffers bound during command buffer recording. We described the number and layout of vertex attributes. We introduced dynamic pipeline states for the viewport and scissors test. We learned how to create buffers and memory objects and how to bind one to another. We also mapped memory and upload data from the CPU to the GPU.

We have created a set of rendering resources that allow us to efficiently record and issue rendering commands. These resources consisted of command buffers, semaphores, fences, and framebuffers. We learned how to use fences, how to set up values of dynamic pipeline states, and how to bind vertex buffers (source of vertex attribute data) during command buffer recording.

The next tutorial will present staging resources. These are intermediate buffers used to copy data between the CPU and GPU. This way, buffers (or images) used for rendering don't have to be mapped by an application and can be bound to a device's local (very fast) memory.