API Without Secrets: Introduction to Vulkan* Part 7

Contents

Tutorial 7: Uniform Buffers—Using Buffers in Shaders

This is the time to summarize knowledge presented so far and create a more typical rendering scenario. Here we will see an example that is still very simple, yet it reflects the most common way to display 3D geometry on screen. We will extend code from the previous tutorial by adding a transformation matrix to the shader uniform data. This way we can see how to use multiple different descriptors in a single descriptor set.

Of course, knowledge presented here applies to many other use cases, as descriptor sets may contain multiple types of resources, both various or identical. Nothing stops us from creating a descriptor set with many storage buffers or sampled images. We can also mix them as shown in this tutorial—here we use both a texture (combined image sampler) and a uniform buffer. We will see how to create a layout for such a descriptor, how to create a descriptor set, and how to populate it with appropriate resources.

In a previous part of the tutorial we learned how to create images and use them as textures inside shaders. This knowledge is also used in this tutorial, but here we focus only on buffers, and learn how to use them as a source of uniform data. We also see how to prepare a projection matrix, how to copy it to a buffer, and how to access it inside shaders.

Creating a Uniform Buffer

In this example we want to use two types of uniform variables inside shaders: combined image sampler (sampler2D inside shader) and a uniform projection matrix (mat4). In Vulkan*, uniform variables other than opaque types like samplers cannot be declared in a global scope (as in OpenGL*); they must be accessed from within uniform buffers. We start by creating a buffer.

Buffers can be used for many different purposes—they can be a source of vertex data (vertex attributes); we can keep vertex indices in them so they are used as index buffers; they can contain shader uniform data, or we can store data inside buffers from within shaders and use them as storage buffers. We can even keep formatted data inside buffers, access it through buffer views, and treat them as texel buffers (similar to OpenGL's buffer textures). For all the above purposes we use the usual buffers, created always in the same way. But it is the usage provided during buffer creation that defines how we can use a given buffer during its lifetime.

We saw how to create a buffer in Introduction to Vulkan Part 4 – Vertex Attributes, so only source code is presented here without diving into specifics:

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

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

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

  if( vkBindBufferMemory( GetDevice(), buffer.Handle, buffer.Memory, 0 ) != VK_SUCCESS
) {
    std::cout << "Could not bind memory to a buffer!" << std::endl;
```

```
    return false;
  }

  return true;
```

1. Tutorial07.cpp, function CreateBuffer()

We first create a buffer by defining its parameters in a variable of type VkBufferCreateInfo. Here we define the buffer's most important parameters—its size and usage. Next, we create a buffer by calling the **vkCreateBuffer()** function. After that, we need to allocate a memory object (or use a part of another, existing memory object) to bind it to the buffer through the **vkBindBufferMemory()** function call. Only after that can we use the buffer the way we want to in our application. Allocating a dedicated memory object is performed as follows:

```
  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 & property) ) {

      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;
```

2. Tutorial07.cpp, function AllocateBufferMemory()

To create a buffer that can be used as a source of shader uniform data, we need to create a buffer with the VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT usage. But, depending on how we want to transfer data to it, we may also need other usages as well. Here we want to use a buffer with a device-local memory bound to it because such memory may have better performance. But, depending on the hardware's architecture, it may not be possible to map such memory and copy data to it directly from the CPU. That's why we want to use a staging buffer through which data will be copied from the CPU to our uniform buffer. And in order to do that, our uniform buffer must also be created with the VK_BUFFER_USAGE_TRANSFER_DST_BIT usage, as it will be a target of data copy operation. Below, we can see how our buffer is finally created:

```
  Vulkan.UniformBuffer.Size = 16 * sizeof(float);
  if( !CreateBuffer( VK_BUFFER_USAGE_TRANSFER_DST_BIT |
VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
Vulkan.UniformBuffer ) ) {
    std::cout << "Could not create uniform buffer!" << std::endl;
    return false;
  }

  if( !CopyUniformBufferData() ) {
    return false;
```

```
  }

  return true;
```

3. Tutorial07.cpp, function CreateUniformBuffer()

Copying data to buffers

The next thing is to upload appropriate data to our uniform buffer. In it we will store 16 elements of a 4 x 4 matrix. We are using an orthographic projection matrix but we can store any other type of data; we just need to remember that each uniform variable must be placed at an appropriate offset, counting from the beginning of a buffer's memory. Such an offset must be a multiple of a specific value. In other words, it must be aligned to a specific value or it must have a specific alignment. The alignment of each uniform variable depends on the variable's type, and the specification defines it as follows:

- A scalar variable whose type has N bytes must be aligned to an address that is a multiple of N.
- A vector with two elements of size N (whose type has N bytes) must be aligned to 2 N.
- A vector with three or four elements, each of size N, has an alignment of 4 N.
- An array's alignment is calculated as an alignment of its elements, rounded up to a multiple of 16.
- A structure's alignment is calculated as the largest alignment of any of its members, rounded up to a multiple of 16.
- A row-major matrix with C columns has an alignment equal to the alignment of a vector with C elements of the same type as the elements of the matrix.
- A column-major matrix has an alignment equal to the alignment of the matrix column type.

The above rules are similar to the rules defined for the standard GLSL 140 layout and we can apply it for Vulkan's uniform buffers as well. But we need to remember that if we place data at inappropriate offsets it will lead to incorrect values being fetched in shaders.

For the sake of simplicity, our example has only one uniform variable, so it can be placed at the very beginning of a buffer. To transfer data to it, we will use a staging buffer—it is created with VK_BUFFER_USAGE_TRANSFER_SRC_BIT usage and is backed by a memory supporting VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT property, so we can map it. Below, we can see how data is copied to the staging buffer:

```
  const std::array<float, 16> uniform_data = GetUniformBufferData();

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

  memcpy( staging_buffer_memory_pointer, uniform_data.data(), Vulkan.UniformBuffer.Size
);

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

  vkUnmapMemory( GetDevice(), Vulkan.StagingBuffer.Memory );
```

First, we prepare the projection matrix data. It is stored in a std::array, but we can keep it in any other type of variable. Next, we map memory bound to the staging buffer. We need to have access to memory that is at least as big as the size of data we want to copy, so we also need to remember to create a staging buffer that is big enough to hold it. Next, we copy data to the staging buffer using an ordinary memcpy() function. Now, we must tell the driver which parts of the buffer's memory were changed; this operation is called flushing. After that, we unmap the staging buffer's memory. Keep in mind that frequent mapping and unmapping may impact performance of our application. In Vulkan, resources can be mapped all the time and it doesn't impact our application in any way. So, if we want to frequently transfer data using staging resources, we should map them only once and keep the acquired pointer for future use. Here we are unmapping it to just show you how to do it.

Now we need to transfer data from the staging buffer to our target, the uniform buffer. In order to do that, we need to prepare a command buffer in which we will record appropriate operations, and which we will submit for these operations to occur.

We start by taking any unused command buffer. It must be allocated from a pool created for a queue that supports transfer operations. Vulkan specification requires that at least one general purpose queue must be available—a queue that supports graphics (rendering), compute, and transfer operations. In the case of Intel hardware, there is only one queue family with one general-purpose queue, so we don't have this problem. Other hardware vendors may support other types of queue families, maybe even a queue family that is dedicated for data transfer. In that case, we should choose a queue from such a family.

We start recording a command buffer by calling the **vkBeginCommandBuffer()** function. Next, we record the **vkCmdCopyBuffer()** command that performs the data transfer, where we tell it that we want to copy data from the very beginning of the staging buffer (0^{th} offset) to the very beginning of our uniform buffer (also 0^{th} offset). We also provide the size of data to be copied.

Next, we need to tell the driver that after the data transfer is performed, our whole uniform buffer will be used as, well, a uniform buffer. This is performed by placing a buffer memory barrier in which we tell it that, until now, we were transferring data to the buffer (VK_ACCESS_TRANSFER_WRITE_BIT), but from now on we will use (VK_ACCESS_UNIFORM_READ_BIT) as a source of data for uniform variables. The buffer memory barrier is placed using the **vkCmdPipelineBarrier()** function call. It occurs after the data transfer operation (VK_PIPELINE_STAGE_TRANSFER_BIT) but before the vertex shader execution, as we access our uniform variable inside the vertex shader (VK_PIPELINE_STAGE_VERTEX_SHADER_BIT).

Finally, we can end the command buffer and submit it to the queue. The whole process is presented in the code below:

```
    // Prepare command buffer to copy data from staging buffer to a uniform buffer
    VkCommandBuffer command_buffer = Vulkan.RenderingResources[0].CommandBuffer;

    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);

    VkBufferCopy buffer_copy_info = {
      0,                                // VkDeviceSize       srcOffset
      0,                                // VkDeviceSize       dstOffset
      Vulkan.UniformBuffer.Size         // VkDeviceSize       size
```

```
    };
    vkCmdCopyBuffer( command_buffer, Vulkan.StagingBuffer.Handle,
Vulkan.UniformBuffer.Handle, 1, &buffer_copy_info );

    VkBufferMemoryBarrier buffer_memory_barrier = {
      VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER, // VkStructureType    sType;
      nullptr,                          // const void        *pNext
      VK_ACCESS_TRANSFER_WRITE_BIT,     // VkAccessFlags      srcAccessMask
      VK_ACCESS_UNIFORM_READ_BIT,       // VkAccessFlags      dstAccessMask
      VK_QUEUE_FAMILY_IGNORED,          // uint32_t           srcQueueFamilyIndex
      VK_QUEUE_FAMILY_IGNORED,          // uint32_t           dstQueueFamilyIndex
      Vulkan.UniformBuffer.Handle,      // VkBuffer           buffer
      0,                                // VkDeviceSize       offset
      VK_WHOLE_SIZE                     // VkDeviceSize       size
    };
    vkCmdPipelineBarrier( command_buffer, VK_PIPELINE_STAGE_TRANSFER_BIT,
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, 0, 0, nullptr, 1, &buffer_memory_barrier, 0,
nullptr );

    vkEndCommandBuffer( command_buffer );

    // Submit command buffer and copy data from staging buffer to a vertex buffer
    VkSubmitInfo submit_info = {
      VK_STRUCTURE_TYPE_SUBMIT_INFO,    // VkStructureType    sType
      nullptr,                          // const void        *pNext
      0,                                // uint32_t           waitSemaphoreCount
      nullptr,                          // const VkSemaphore *pWaitSemaphores
      nullptr,                          // const VkPipelineStageFlags *pWaitDstStageMask;
      1,                                // uint32_t           commandBufferCount
      &command_buffer,                  // const VkCommandBuffer *pCommandBuffers
      0,                                // uint32_t           signalSemaphoreCount
      nullptr                           // const VkSemaphore *pSignalSemaphores
    };

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

    vkDeviceWaitIdle( GetDevice() );
    return true;
```

5. Tutorial07.cpp, function CopyUniformBufferData()

In the code above we call the **vkDeviceWaitIdle()** function to make sure the data transfer operation is finished before we proceed. But in real-life situations, we should perform more appropriate synchronizations by using semaphores and/or fences. Waiting for all the GPU operations to finish may (and probably will) kill performance of our application.

Preparing Descriptor Sets

Now we can prepare descriptor sets—the interface between our application and a pipeline through which we can provide resources used by shaders. We start by creating a descriptor set layout.

Creating descriptor set layout

The most typical way that 3D geometry is rendered is by multiplying vertices by model, view, and projection matrices inside a vertex shader. These matrices may be accumulated in a model-view-projection matrix. We need to provide such a matrix to the vertex shader in a uniform variable. Usually we want our geometry to be textured; the fragment shader needs access to a texture—a combined image sampler. We can also use a separate sampled image and a sampler; using combined image samplers may have better performance on some platforms.

When we issue drawing commands, we want the vertex shader to have access to a uniform variable, and the fragment shader to a combined image sampler. These resources must be provided in a descriptor set. In order to allocate such a set we need to create an appropriate layout, which defines what types of resources are stored inside the descriptor sets.

```
  std::vector<VkDescriptorSetLayoutBinding> layout_bindings = {
    {
      0,                                            // uint32_t           binding
      VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,    // VkDescriptorType   descriptorType
      1,                                            // uint32_t           descriptorCount
      VK_SHADER_STAGE_FRAGMENT_BIT,                 // VkShaderStageFlags stageFlags
      nullptr                                       // const VkSampler *pImmutableSamplers
    },
    {
      1,                                            // uint32_t           binding
      VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,            // VkDescriptorType   descriptorType
      1,                                            // uint32_t           descriptorCount
      VK_SHADER_STAGE_VERTEX_BIT,                   // VkShaderStageFlags stageFlags
      nullptr                                       // const VkSampler *pImmutableSamplers
    }
  };

  VkDescriptorSetLayoutCreateInfo descriptor_set_layout_create_info = {
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, // VkStructureType  sType
    nullptr,                                             // const void      *pNext
    0,                                                   //
VkDescriptorSetLayoutCreateFlags flags
    static_cast<uint32_t>(layout_bindings.size()),       // uint32_t
bindingCount
    layout_bindings.data()                               // const
VkDescriptorSetLayoutBinding *pBindings
  };

  if( vkCreateDescriptorSetLayout( GetDevice(), &descriptor_set_layout_create_info,
nullptr, &Vulkan.DescriptorSet.Layout ) != VK_SUCCESS ) {
    std::cout << "Could not create descriptor set layout!" << std::endl;
    return false;
  }

  return true;
```

6. Tutorial07.cpp, function CreateDescriptorSetLayout()

Descriptor set layouts are created by specifying bindings. Each binding defines a separate entry in a descriptor set and has its own, unique index within a descriptor set. In the above code we define that descriptor set (and its layout); it contains two bindings. The first binding, with index 0, is for one combined image sampler accessed by a fragment shader. The second binding, with index 1, is for a uniform buffer accessed by a vertex shader. Both are single resources; they are not arrays. But, we can also specify that each binding represents an array of resources by providing a value greater than 1 in the descriptorCount member of the VkDescriptorSetLayoutBinding structure.

Bindings are also used inside shaders. When we define uniform variables, we need to specify the same binding value as the one provided during layout creation:

```
layout( set=S, binding=B ) uniform <variable type> <variable name>;
```

Two things are worth mentioning. Bindings do not need to be consecutive. We can create a layout with three bindings occupying, for example, indices 2, 5, and 9. But unused slots may still use some memory, so we should keep bindings as close to 0 as possible.

We also specify which shader stages need access to which types of descriptors (which bindings). If we are not sure, we can provide more stages. For example, let's say we want to create several pipelines, all using descriptor sets with the same layout. In some of these pipelines, a uniform buffer will be accessed in a vertex shader, in others in a geometry shader, and in still others in both vertex and fragment shaders. For such a purpose we can create one layout in which we can specify that the uniform buffer will be accessed by vertex, geometry, and fragment shaders. But we should not provide unnecessary shader stages because, as usual, it may impact the performance (though this does not mean it will).

After specifying an array of bindings, we provide a pointer to it in a variable of type VkDescriptorSetLayoutCreateInfo. The pointer to this variable is provided in the **vkCreateDescriptorSetLayout()** function, which creates the actual layout. When we have it we can allocate a descriptor set. But first, we need a pool of memory from which the set can be allocated.

Creating a descriptor pool

When we want to create a descriptor pool we need to know what types of resources will be defined in descriptor sets allocated from the pool. We also need to specify not only the maximum number of resources of each type that can be stored in descriptor sets allocated from the pool, but also the maximum number of descriptor sets allocated from the pool. For example, we can prepare a storage for one combined image sampler and for one uniform buffer, but for two sets in total. This means that we can have two sets, one with a texture and one with a uniform buffer, or only one set, with both a uniform buffer and a texture (in such a situation the second pool is empty, as it cannot contain either of these two resources).

In our example we need only one descriptor set, and we can see below how to create a descriptor pool for it:

```
  std::vector<VkDescriptorPoolSize> pool_sizes = {
    {
      VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,   // VkDescriptorType  type
      1                                            // uint32_t          descriptorCount
    },
    {
      VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,           // VkDescriptorType  type
      1                                            // uint32_t          descriptorCount
    }
  };

  VkDescriptorPoolCreateInfo descriptor_pool_create_info = {
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO, // VkStructureType     sType
    nullptr,                                       // const void         *pNext
    0,                                             // VkDescriptorPoolCreateFlags flags
    1,                                             // uint32_t            maxSets
    static_cast<uint32_t>(pool_sizes.size()),      // uint32_t            poolSizeCount
    pool_sizes.data()                              // const VkDescriptorPoolSize
*pPoolSizes
  };

  if( vkCreateDescriptorPool( GetDevice(), &descriptor_pool_create_info, nullptr,
&Vulkan.DescriptorSet.Pool ) != VK_SUCCESS ) {
    std::cout << "Could not create descriptor pool!" << std::endl;
    return false;
  }

  return true;
```

7. Tutorial07.cpp, function CreateDescriptorPool()

Now, we are ready to allocate descriptor sets from the pool using the previously created layout.

Allocating descriptor sets

Descriptor set allocation is pretty straightforward. We just need a descriptor pool and a layout. We specify the number of descriptor sets to allocate and call the **vkAllocateDescriptorSets()** function like this:

```
  VkDescriptorSetAllocateInfo descriptor_set_allocate_info = {
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO, // VkStructureType    sType
    nullptr,                                        // const void       *pNext
    Vulkan.DescriptorSet.Pool,                      // VkDescriptorPool  descriptorPool
    1,                                              // uint32_t      descriptorSetCount
    &Vulkan.DescriptorSet.Layout                    // const VkDescriptorSetLayout
*pSetLayouts
  };

  if( vkAllocateDescriptorSets( GetDevice(), &descriptor_set_allocate_info,
&Vulkan.DescriptorSet.Handle ) != VK_SUCCESS ) {
    std::cout << "Could not allocate descriptor set!" << std::endl;
    return false;
  }

  return true;
```

8. Tutorial07.cpp, function AllocateDescriptorSet()

Updating descriptor sets

We have allocated a descriptor set. It is used to provide a texture and a uniform buffer to the pipeline so they can be used inside shaders. Now we must provide specific resources that will be used as descriptors. For the combined image sampler, we need two resources—an image, which can be sampled inside shaders (it must be created with the VK_IMAGE_USAGE_SAMPLED_BIT usage), and a sampler. These are two separate resources, but they are provided together to form a single, combined image sampler descriptor. For details about how to create these two resources, please refer to the Introduction to Vulkan Part 6 – Descriptor Sets. For the uniform buffer we will provide a buffer created earlier. To provide specific resources to a descriptor, we need to update a descriptor set. During updates we specify descriptor types, binding numbers, and counts in exactly the same way as we did during layout creation. These values must match. Apart from that, depending on the descriptor type, we also need to create variables of type:

- VkDescriptorImageInfo, for samplers, sampled images, combined image samplers, and input attachments
- VkDescriptorBufferInfo, for uniform and storage buffers and their dynamic variations
- VkBufferView, for uniform and storage texel buffers

Through them, we provide handles of specific Vulkan resources that should be used for corresponding descriptors. All this is provided to the **vkUpdateDescriptorSets()** function, as we can see below:

```
  VkDescriptorImageInfo image_info = {
    Vulkan.Image.Sampler,                    // VkSampler        sampler
    Vulkan.Image.View,                       // VkImageView      imageView
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL // VkImageLayout    imageLayout
  };

  VkDescriptorBufferInfo buffer_info = {
    Vulkan.UniformBuffer.Handle,             // VkBuffer         buffer
    0,                                       // VkDeviceSize     offset
    Vulkan.UniformBuffer.Size                // VkDeviceSize     range
  };

  std::vector<VkWriteDescriptorSet> descriptor_writes = {
    {
      VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET,    // VkStructureType     sType
      nullptr,                                   // const void         *pNext
      Vulkan.DescriptorSet.Handle,               // VkDescriptorSet     dstSet
      0,                                         // uint32_t            dstBinding
      0,                                         // uint32_t            dstArrayElement
      1,                                         // uint32_t            descriptorCount
      VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, // VkDescriptorType    descriptorType
```

```
      &image_info,                                  // const VkDescriptorImageInfo
*pImageInfo
      nullptr,                                      // const VkDescriptorBufferInfo
*pBufferInfo
      nullptr                                       // const VkBufferView
*pTexelBufferView
    },
    {
      VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET,     // VkStructureType     sType
      nullptr,                                      // const void         *pNext
      Vulkan.DescriptorSet.Handle,                  // VkDescriptorSet     dstSet
      1,                                            // uint32_t            dstBinding
      0,                                            // uint32_t            dstArrayElement
      1,                                            // uint32_t            descriptorCount
      VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,          // VkDescriptorType    descriptorType
      nullptr,                                      // const VkDescriptorImageInfo
*pImageInfo
      &buffer_info,                                 // const VkDescriptorBufferInfo
*pBufferInfo
      nullptr                                       // const VkBufferView
*pTexelBufferView
    }
  };

  vkUpdateDescriptorSets( GetDevice(), static_cast<uint32_t>(descriptor_writes.size()),
&descriptor_writes[0], 0, nullptr );
  return true;
```

9. Tutorial07.cpp, function UpdateDescriptorSet()

Now we have a valid descriptor set. We can bind it during command buffer recording. But, for that we need a pipeline object, which is created with an appropriate pipeline layout.

Preparing Drawing State

Created descriptor set layouts are required for two purposes:

- Allocating descriptor sets from pools
- Creating pipeline layout

The descriptor set layout specifies what types of resources the descriptor set contains. The pipeline layout specifies what types of resources can be accessed by a pipeline and its shaders. That's why before we can use a descriptor set during command buffer recording, we need to create a pipeline layout.

Creating a pipeline layout

The pipeline layout defines the resources that a given pipeline can access. These are divided into descriptors and push constants. To create a pipeline layout, we need to provide a list of descriptor set layouts, and a list of ranges of push constants.

Push constants provide a way to pass data to shaders easily and very, very quickly. Unfortunately, the amount of data is also very limited—the specification allows only 128 bytes to be available for push constants data provided to a pipeline at a given time. Hardware vendors may allow us to provide more data, but it is still a very small amount compared to usual descriptors, like uniform buffers.

In this example we don't use push constants ranges, so we only need to provide our descriptor set layout and call the **vkCreatePipelineLayout()** function. The code below does exactly that:

```
  VkPipelineLayoutCreateInfo layout_create_info = {
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, // VkStructureType
sType
```

```
    nullptr,                                                // const void
*pNext
    0,                                                      // VkPipelineLayoutCreateFlags
flags
    1,                                                      // uint32_t
setLayoutCount
    &Vulkan.DescriptorSet.Layout,                           // const VkDescriptorSetLayout
*pSetLayouts
    0,                                                      // uint32_t
pushConstantRangeCount
    nullptr                                                 // const VkPushConstantRange
*pPushConstantRanges
  };

  if( vkCreatePipelineLayout( GetDevice(), &layout_create_info, nullptr,
&Vulkan.PipelineLayout ) != VK_SUCCESS ) {
    std::cout << "Could not create pipeline layout!" << std::endl;
    return false;
  }
  return true;
```

10. Tutorial07.cpp, function CreatePipelineLayout()

Creating shader programs

Now we need a graphics pipeline. Pipeline creation is a very time-consuming process, from both the performance and code development perspective. I will skip the code and present only the GLSL source code of shaders.

The vertex shader used during drawing takes a vertex position and multiplies it by a projection matrix read from a uniform variable. This variable is stored inside a uniform buffer. The descriptor set, through which we provide our uniform buffer, is the first (and the only one in this case) in the list of descriptor sets specified during pipeline layout creation. So, when we record a command buffer, we can bind it to the 0^{th} index. This is because indices to which we bind descriptor sets must match indices corresponding to descriptor set layouts that are provided during pipeline layout creation. The same set index must be specified inside shaders. The uniform buffer is represented by the second binding within that set (it has an index equal to 1), and the same binding number must also be specified. This is the whole vertex shader source code:

```
  #version 450

  layout(set=0, binding=1) uniform u_UniformBuffer {
      mat4 u_ProjectionMatrix;
  };

  layout(location = 0) in vec4 i_Position;
  layout(location = 1) in vec2 i_Texcoord;

  out gl_PerVertex
  {
      vec4 gl_Position;
  };

  layout(location = 0) out vec2 v_Texcoord;

  void main() {
      gl_Position = u_ProjectionMatrix * i_Position;
      v_Texcoord = i_Texcoord;
  }
```

11. shader.vert, -

Inside the shader we also pass texture coordinates to a fragment shader. The fragment shader takes them and samples the combined image sampler. It is provided through the same descriptor set bound to index 0, but it is the first descriptor inside it, so in this case we specify 0 (zero) as the binding's value. Have a look at the full GLSL source code of the fragment shader:

```
#version 450

layout(set=0, binding=0) uniform sampler2D u_Texture;

layout(location = 0) in vec2 v_Texcoord;

layout(location = 0) out vec4 o_Color;

void main() {
  o_Color = texture( u_Texture, v_Texcoord );
}
```

12. shader.frag, -

The above two shaders need to be converted to a SPIR-V* assembly before we can use them in our application. The core Vulkan specification allows only for binary SPIR-V data to be used as a source of shader instructions. From them we can create two shader modules, one for each shader stage, and use them to create a graphics pipeline. The rest of the pipeline state remains unchanged.

Binding descriptor sets

Let's assume we have all the other resources created and ready to be used to draw our geometry. We start recording a command buffer. Drawing commands can only be called inside render passes. Before we can draw any geometry, we need to set all the required states—first and foremost, we need to bind a graphics pipeline. Apart from that, if we are using a vertex buffer, we need to bind the appropriate buffer for this purpose. If we want to issue indexed drawing commands, we need to bind a buffer with vertex indices too. And when we are using shader resources like uniform buffers or textures, we need to bind descriptor sets. We do this by calling the **vkCmdBindDescriptorSets()** function, in which we need to provide not only the handle of our descriptor set, but also the handle of the pipeline layout (so we need to keep it). Only after that can we record drawing commands. These operations are presented in the code below:

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

   vkCmdBindPipeline( command_buffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
Vulkan.GraphicsPipeline );

   // ...

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

   vkCmdBindDescriptorSets( command_buffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
Vulkan.PipelineLayout, 0, 1, &Vulkan.DescriptorSet.Handle, 0, nullptr );

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

   vkCmdEndRenderPass( command_buffer );
```

13. Tutorial07.cpp, function PrepareFrame()

Don't forget that a typical frame of animation requires us to acquire an image from a swapchain, record the command buffer (or more) as presented above, submit it to a queue, and present a previously acquired swapchain image so it gets displayed according to the present mode requested during swapchain creation.

Tutorial 7 Execution

Have a look at how the final image generated by the sample program should appear:

We still render a quad that has a texture applied to its surface. But the size of the quad should remain unchanged when we change the size of our application's window.

Cleaning Up

As usual, at the end of our application, we should perform a cleanup.

```
// ...

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

if( Vulkan.PipelineLayout != VK_NULL_HANDLE ) {
  vkDestroyPipelineLayout( GetDevice(), Vulkan.PipelineLayout, nullptr );
  Vulkan.PipelineLayout = VK_NULL_HANDLE;
}

// ...

if( Vulkan.DescriptorSet.Pool != VK_NULL_HANDLE ) {
  vkDestroyDescriptorPool( GetDevice(), Vulkan.DescriptorSet.Pool, nullptr );
  Vulkan.DescriptorSet.Pool = VK_NULL_HANDLE;
}

if( Vulkan.DescriptorSet.Layout != VK_NULL_HANDLE ) {
  vkDestroyDescriptorSetLayout( GetDevice(), Vulkan.DescriptorSet.Layout, nullptr );
  Vulkan.DescriptorSet.Layout = VK_NULL_HANDLE;
}

DestroyBuffer( Vulkan.UniformBuffer );
```

14. Tutorial07.cpp, function destructor

Most of the resources are destroyed, as usual. We do this in the order opposite to the order of their creation. Here, only the part of code relevant to our example is presented. The graphics pipeline is destroyed by calling the **vkDestroyPipeline()** function. To destroy its layout we need to call the **vkDestroyPipelineLayout()** function. We don't need to destroy all descriptor sets separately, because when we destroy a descriptor pool, all sets allocated from it also get destroyed. To destroy a descriptor pool we need to call the **vkDestroyDescriptorPool()** function. The descriptor set layout needs to be destroyed separately with the **vkDestroyDescriptorSetLayout()** function. After that, we can destroy the uniform buffer; it is done with the **vkDestroyBuffer()** function. But we can't forget to destroy the memory bound to it via the **vkFreeMemory()** function, as seen below:

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

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

15. Tutorial07.cpp, function DestroyBuffe()

Conclusion

In this part of the tutorial we extended the example from the previous part by adding the uniform buffer to a descriptor set. Uniform buffers are created as are all other buffers, but with the VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT usage specified during their creation. We also allocated dedicated memory and bound it to the buffer, and we uploaded projection matrix data to the buffer using a staging buffer.

Next, we prepared the descriptor set, starting with creating a descriptor set layout with one combined image sampler and one uniform buffer. Next, we created a descriptor pool big enough to contain these two types of descriptor resources, and we allocated a single descriptor set from it. After that, we updated the descriptor set with handles of a sampler, an image view of a sampled image, and the buffer created in this part of the tutorial.

The rest of the operations were similar to the ones we already know. The descriptor set layout was used during pipeline layout creation, which was then used when we bound the descriptor sets to a command buffer.

We have seen once again how to prepare shader code for both vertex and fragment shaders, and we learned how to access different types of descriptors provided through different bindings of the same descriptor set.

The next parts of the tutorial will be a bit different, as we will see and compare different approaches to managing multiple resources and handling various, more complicated, tasks.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.