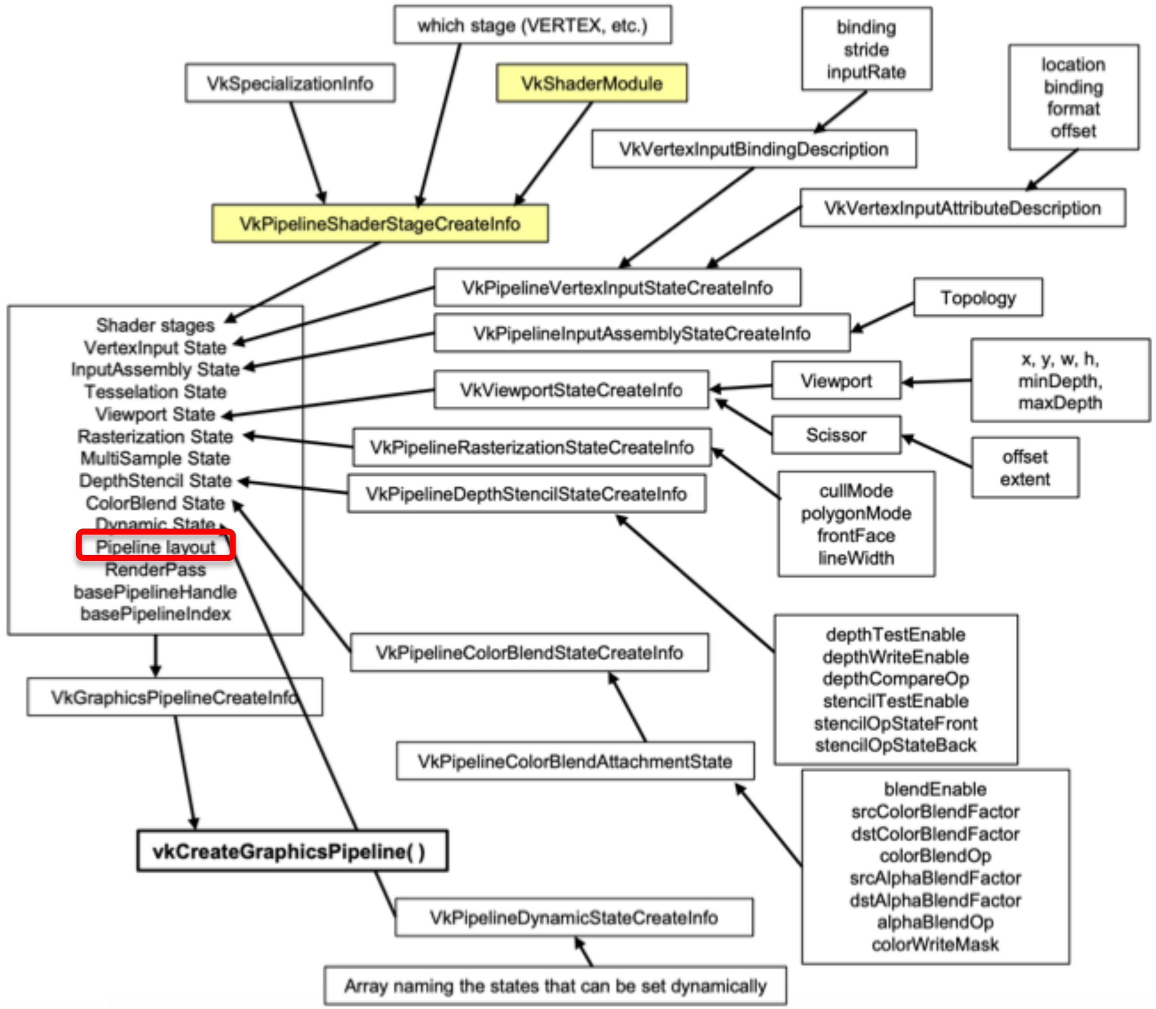
Uniforms and Buffers

Pipeline creation

When creating the pipeline, we did not consider how to create its layout in detail.

We will now see its purpose and how it can be defined.



From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Uniform buffers

As we have seen when introducing GLSL, application can send scene and mesh dependent data using *Uniform Blocks* global variables.

Shader-application communication

Communication between the Shaders and the application occurs using *Uniform Variables Blocks.*

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Marco Gribaudo, assoc.prof. DEIB Dept. POLITECNICO MILANO 1863

Uniform buffers

The same technique is used also to pass textures to Shaders.

We will start focusing on standard variables only, and we will see how to pass images in the following lesson.

Textures in Shaders

Textures are passed to shaders as particular uniform variables of "Combined Texture Sample" type.

```
layout(location = 0) in vec3 fragPos;
layout(location = 1) in vec3 fragNorm;
layout(location = 2) in vec2 fragUV;

layout(location = 0) out vec4 outColor;

layout(binding = 1) uniform sampler2D texSampler;


void main() {
    vec3 Diffuse = texture(texSampler, fragUV).rgb ;

    outColor = vec4(Diffuse, 1.0);

}
```

Uniforms: a typical rendering cycle

To better understand the reason for the specific organization of global variables chosen by Vulkan, let's see a typical rendering cycle of an application.

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: a typical rendering cycle

Some parameters used by the Shaders are scene dependent:

- Camera position (view matrix)
- Ambient light definition.
- Light types, positions, directions and colors.
- …

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: a typical rendering cycle

Each shader will require its own pipeline, plus specific parameters:

- Algorithm selection (i.e. GGX or other for Cook-Trorrance)
- Special debugging views (i.e. shows just the shading, or just the texture)
- …

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: a typical rendering cycle

Each material might have specific settings:

- Specular power
- Roughness
- Diffuse or specular colors
- Textures
- …

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: a typical rendering cycle

The same parameters might be used by several objects.

To reduce changes of information used by the GPU, the meshes with identical material properties are usually grouped together and drawn one after the other.

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: a typical rendering cycle

Finally, each mesh might have its own properties, which the Shaders will use for drawing all their triangles:

- World transform matrices
- UV animations
- …

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: Sets and Bindings

Vulkan groups uniform variables into *Sets*: each one represents one the “levels” of the frequency at which values are updated.

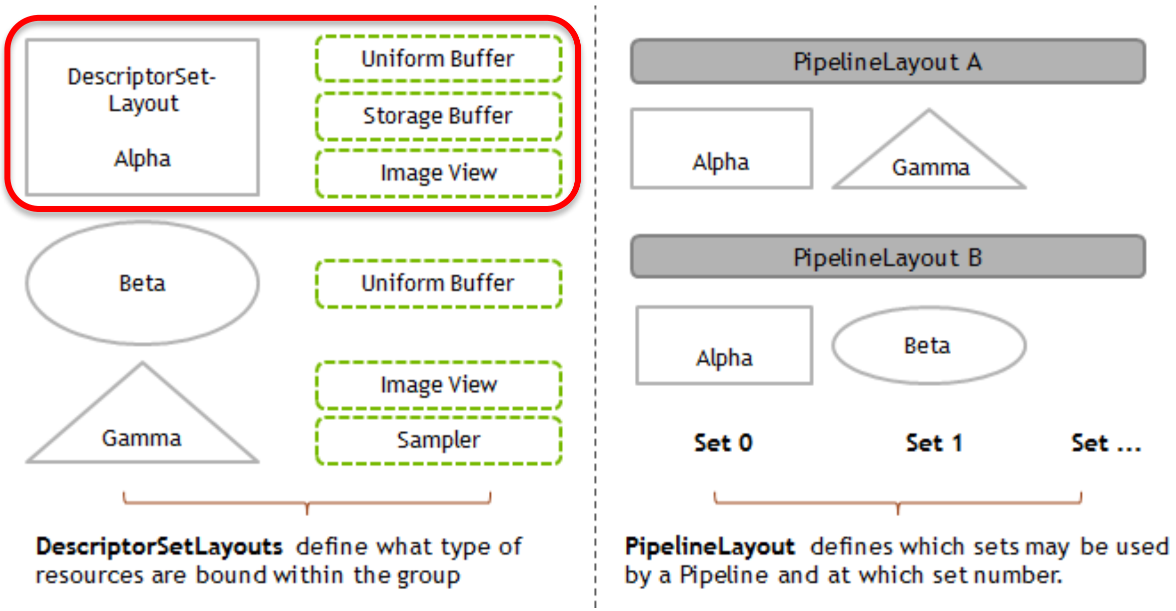
Each Set is characterized by an *ID* (starting from 0): sets with a smaller ID are assumed to change less often.

```
// example for typical loops in rendering
for each view {
  bind view resources          Set 0   // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      Set 1   // shader control values
    for each material {
      bind material resources  Set 2   // material parameters and textures
      for each object {
        bind object resources  Set 3   // object transforms
        draw object
      }
    }
  }
}
```

Uniforms: Sets and Bindings

Each set can contain a lot of resources:

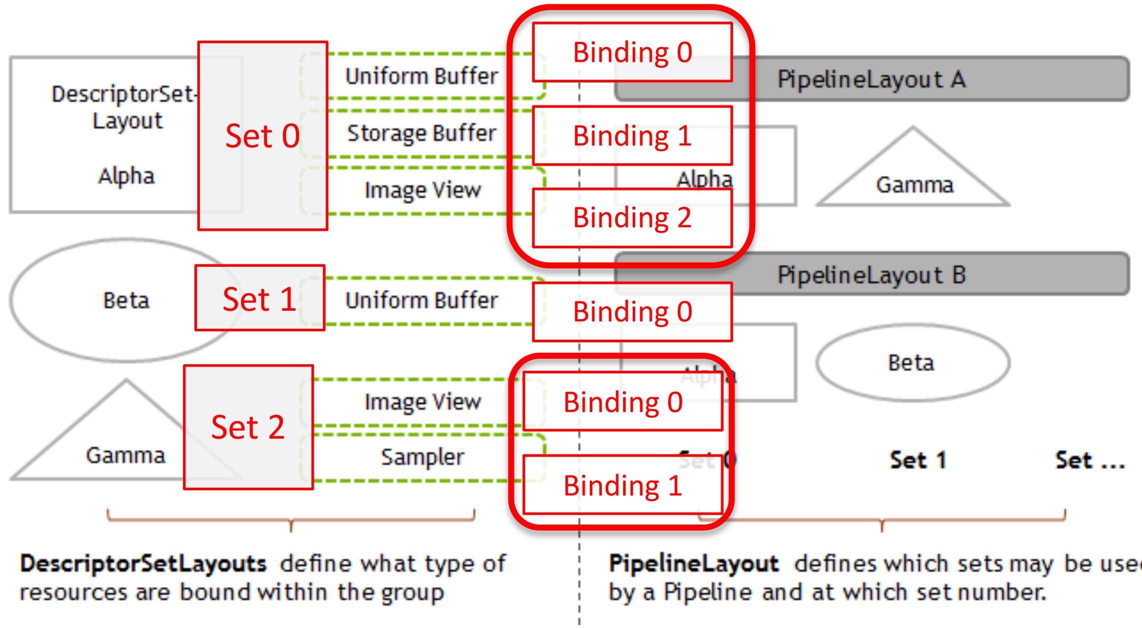
- Several uniform blocks with different purposes (i.e. light definitions, environment properties.)
- Textures
- Other data



From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: Sets and Bindings

Resources inside a set must be identified with a secondary index, again starting from zero, called the *Binding*.



From: https://developer.nvidia.com/vulkan-shader-resource-binding

Uniforms: Binding types

Several types of resources can be accessed as global *Uniform Variables*:

- An uniform block of variables
- A texture sampler
- An image
- A combined image + sampler
- A render pass attachment (in the same position)
- …

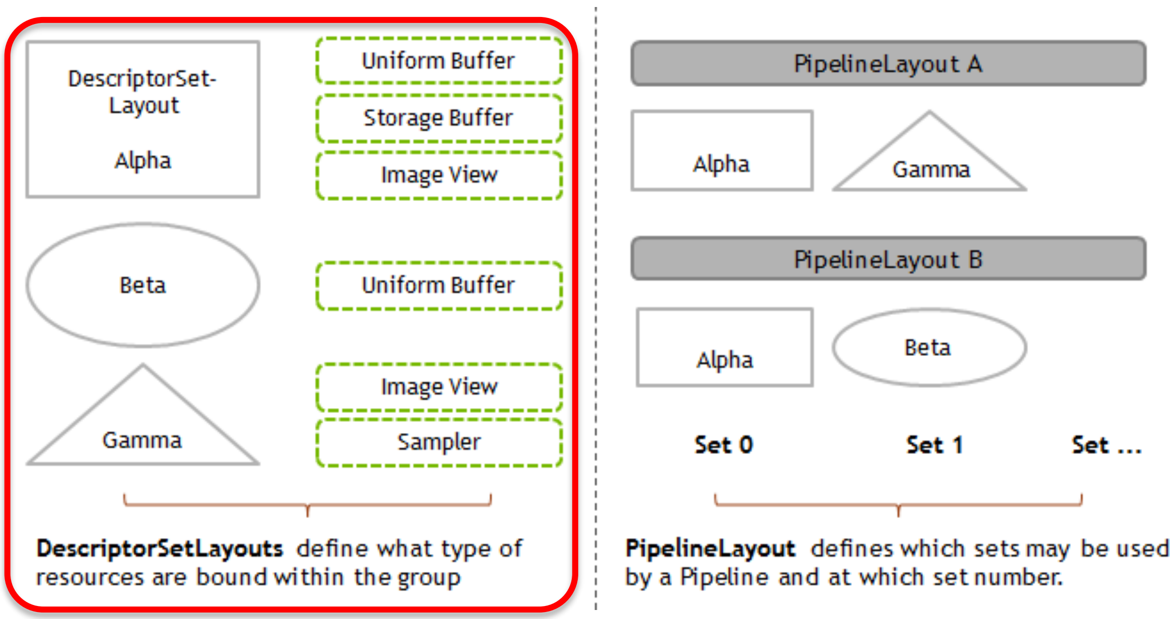
```
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorType {
    VK_DESCRIPTOR_TYPE_SAMPLER = 0,
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER = 1,
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE = 2,
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE = 3,
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER = 4,
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER = 5,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER = 6,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER = 7,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC = 8,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC = 9,
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT = 10,
  // Provided by VK_VERSION_1_3
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK = 10001
  // Provided by VK_KHR_acceleration_structure
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR =
  // Provided by VK_NV_ray_tracing
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV =
  // Provided by VK_VALVE_mutable_descriptor_type
    VK_DESCRIPTOR_TYPE_MUTABLE_VALVE = 1000351000,
  // Provided by VK_EXT_inline_uniform_block
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT = VI
} VkDescriptorType;
```

For a complete reference see:
https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkDescriptorType.html

Uniforms: Descriptor Sets, Descriptor Layouts and **Pipeline Layout**

In OOP notation, *Descriptor Layouts* represents the “class” of the uniform variables. They specify:

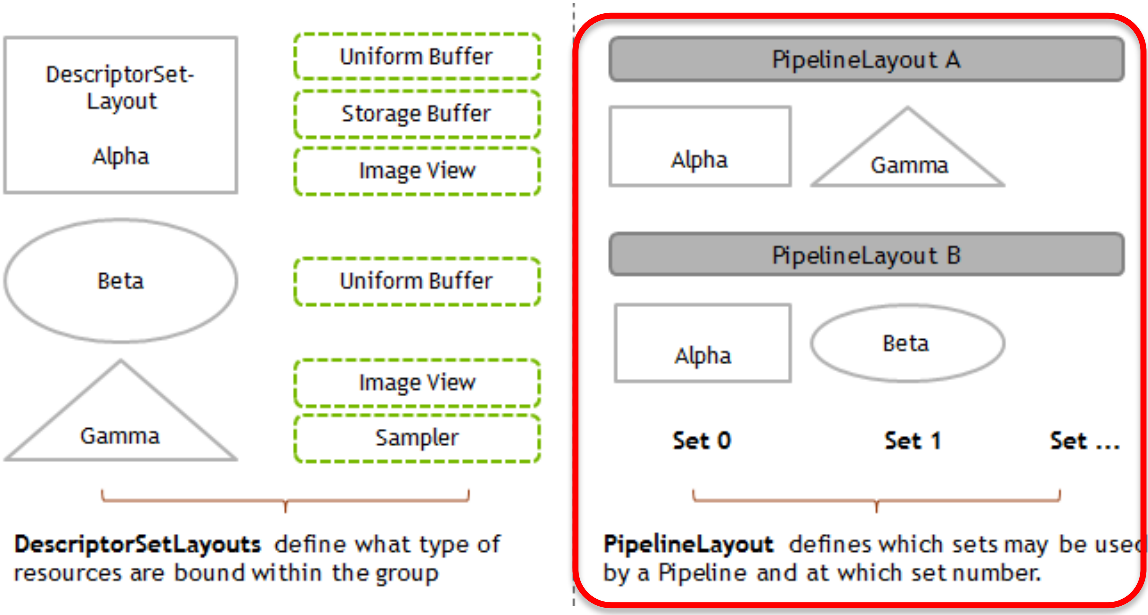
- The type of the descriptor (uniform, texture image, …)
- Its binding ID
- The stage in which this will be used (i.e. *Vertex* or *Fragment* shader).



Uniforms: Descriptor Sets, Descriptor Layouts and Pipeline Layout

The *Pipeline Layout*, selects which of the available *Descriptors Layouts* will be accessed in that specific pipeline.

It will also define at which *Set ID* such descriptors will be found in the Shader.



Uniforms: Descriptor Sets, Descriptor Layouts and Pipeline Layout

In OOP terms, *Descriptor Sets* are the *Instances* of the uniform data: they actually define the values that will be passed to the uniforms.

For example, different meshes with the same material, but requiring a different world matrix, will access a different *Descriptor Set* associated to the same *Descriptor Layout*.

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control values
    for each material {
      bind material resources  // material parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

Descriptor layout definition

Descriptor Layouts in the same set (but with different bindings), are defined inside an array of `VkDescriptorSetLayoutBinding`.

```
VkDescriptorSetLayoutBinding uboLayoutBinding{};
uboLayoutBinding.binding = 0;
uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboLayoutBinding.descriptorCount = 1;
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
uboLayoutBinding.pImmutableSamplers = nullptr;

VkDescriptorSetLayoutBinding samplerLayoutBinding{};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
samplerLayoutBinding.pImmutableSamplers = nullptr;

std::array<VkDescriptorSetLayoutBinding, 2> bindings =
                        {uboLayoutBinding, samplerLayoutBinding};
```

Descriptor layout definition

Each binding must specify its integer ID starting from zero, its type (*Uniform*, *Texture sampler*, etc), and which *Shaders* can use it.

```
VkDescriptorSetLayoutBinding uboLayoutBinding{};
uboLayoutBinding.binding = 0;
uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboLayoutBinding.descriptorCount = 1;
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
uboLayoutBinding.pImmutableSamplers = nullptr;

VkDescriptorSetLayoutBinding samplerLayoutBinding{};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
samplerLayoutBinding.pImmutableSamplers = nullptr;

std::array<VkDescriptorSetLayoutBinding, 2> bindings =
                         {uboLayoutBinding, samplerLayoutBinding};
```

Descriptor layout definition

Possible stage flags are:

- `VK_SHADER_STAGE_VERTEX_BIT`: *Vertex Shader*
- `VK_SHADER_STAGE_FRAGMENT_BIT`: *Fragment Shader*
- `VK_SHADER_STAGE_ALL_GRAPHICS`: all Shaders

Descriptor layout definition

Uniform blocks can be defined in arrays composed of several elements, and if a texture will not be varied in all the pipelines it appears in, some optimization might be triggered. These capabilities are however outside the scope of this course and will not be considered.

```
VkDescriptorSetLayoutBinding uboLayoutBinding{};
uboLayoutBinding.binding = 0;
uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboLayoutBinding.descriptorCount = 1;
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
uboLayoutBinding.pImmutableSamplers = nullptr;

VkDescriptorSetLayoutBinding samplerLayoutBinding{};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
samplerLayoutBinding.pImmutableSamplers = nullptr;

std::array<VkDescriptorSetLayoutBinding, 2> bindings =
                         {uboLayoutBinding, samplerLayoutBinding};
```

We will only use these values!

Descriptor layout creation

`VkDescriptorSetLayout` objects are then created with the `vkCraeteDescriptorSetLayout` function, receiving the required data inside a `VkDescriptorSetLayoutCreateInfo` structure, conatining a pointer to the binding array, and the number of elements.

```
…

VkDescriptorSetLayout DescriptorSetLayout;

VkDescriptorSetLayoutCreateInfo layoutInfo{};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
layoutInfo.pBindings = bindings.data();

VkResult result = vkCreateDescriptorSetLayout(device, &layoutInfo,
                                   nullptr, &DescriptorSetLayout);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create descriptor set layout!");
}
```

Descriptor layout and pipeline layout

Descriptor sets are then grouped inside an array, and passed in the `pSetLayouts` field of the `VkPipelineLayoutCreateInfo` structure, used to create the `VkPipelineLayout` in the `VkCreatePipelineLayout` command. The number of sets passed is defined in the `setLayoutCount` field.

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optional

VkResult result = vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr,
               &PipelineLayout);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create pipeline layout!");
}
```

Descriptor layout and pipeline layout

The position inside the array used to create the pipeline Layout corresponds to the *Set ID* used to define that particular *Descriptor Layout*.

```
// example for typical loops in rendering
for each view {
  bind view resources    Set 0   // camera, environment...
  for each shader {
    bind shader pipeline
    bind shader resources    Set 1   // shader control values
    for each material {
      bind material resources    Set 2   erial parameters and textures
      for each object {
        bind object resources    Set 3   ect transforms
        draw object
      }
    }
  }
}
```

Descriptor Pools

Descriptor sets must be allocated from *Descriptor Pools*, similarly to what we have seen for *Command Buffers*.

In this case, however, things are slightly more complex, since an accurate estimate of the number of sets is required.

Command Pools

Command Pools are created with the `vkCreateCommandPool()` function. The only parameter that needs to be defined in the creation structure is the *Queue* family on which its commands will be executed using the `queueFamilyIndex` field. On success, the handle to the command pool fills the `VkCommandPool` argument.

```
VkCommandPool commandPool;

VkCommandPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = aQueueWithGraphicsCapability.value();
poolInfo.flags = 0; // Optional

result = vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool);
if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
}
```

Currently, we are only interested in the queue for graphics creation.

Marco Gribaudo, assoc.prof. DEIB Dept.
POLITECNICO MILANO 1863

Descriptor Pools

The pool is defined as a set of `VkDescriptorPoolSize` objects, each one describing the `type` and the quantity of descriptors (`descriptorCount` field).

```
std::array<VkDescriptorPoolSize, 2> poolSizes{};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = NUniformBuffersInstances;
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = NTextures;
```

```
VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = max(NUniformBuffersInstances, NTextures);

VkDescriptorPool descriptorPool;
VkResult result = vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create descriptor pool!");
}
```

Descriptor Pools

This array of requests is used to fill a VkDescriptorPoolCreateInfo structure.

This also requires the specification of the maximum number of descriptor sets used by the application

```
std::array<VkDescriptorPoolSize, 2> poolSizes{};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = NUniformBuffersInstances;
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = NTextures;

VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = NDescriptorSets;

VkDescriptorPool descriptorPool;
VkResult result = vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create descriptor pool!");
}
```

Descriptor Pools

Determining the right number of descriptors and descriptor sets required by an application is quite challenging, and it deeply depends on how the rendering engine is structured.

If no special structure is used, they should be equal to the sum of the number of different Descriptor Sets and elements of a specific type used in the application.

* Even if in a production environment this must be avoided at all costs, in the code create for this course overprovisioning will be tolerated as a way to simplify this specific part!

Descriptor Pools

The descriptor pool can then be created using the `VkCreateDescriptorPool()` command.

```
std::array<VkDescriptorPoolSize, 2> poolSizes{};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = NUniformBuffersInstances;
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = NTextures;

VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = NDescriptorSets;

VkDescriptorPool descriptorPool;
VkResult result = vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create descriptor pool!");
}
```

Memory allocation for Descriptor Sets

Descriptor Pools are needed to allocate the *Descriptor Sets* using the `VkAllocateDescriptorSet()` command, and the information filled inside a `VkDescriptorSetAllocateInfo` structure.

```
std::vector<VkDescriptorSetLayout> layouts(NDescriptorSets, descriptorSetLayout);

VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = NDescriptorSets;
allocInfo.pSetLayouts = layouts.data();

std::vector<VkDescriptorSet> DescriptorSets;
DescriptorSets.resize(NDescriptorSets);

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, DescriptorSets.data());
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate descriptor sets!");
}
```

Memory allocation for Descriptor Sets

The creation structure contains a pointer to the Pool and the number of Descriptor Sets to create.

It also includes, for each element of the set, the pointer to the corresponding *Set Layout*.

```
std::vector<VkDescriptorSetLayout> layouts(NDescriptorSets, descriptorSetLayout);

VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = NDescriptorSets;
allocInfo.pSetLayouts = layouts.data();

std::vector<VkDescriptorSet> DescriptorSets;
DescriptorSets.resize(NDescriptorSets);

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, DescriptorSets.data());
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate descriptor sets!");
}
```

Memory allocation for Descriptor Sets

Since each *Descriptor Set* can have a potentially different *Set Layout*, this information needs to be passed as an array. Each pointer must repeated if several sets shares the same layout.

The `array` constructor having two parameters (the count and an instance of the template type), can be useful to assign the same value to all the elements.

```
std::vector<VkDescriptorSetLayout> layouts(NDescriptorSets, descriptorSetLayout);

VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = NDescriptorSets;
allocInfo.pSetLayouts = layouts.data();

std::vector<VkDescriptorSet> DescriptorSets;
DescriptorSets.resize(NDescriptorSets);

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, DescriptorSets.data());
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate descriptor sets!");
}
```

Memory allocation for Descriptor Sets

Pay attention to the return value of the allocation call: if all the Sets of the Pool have been used, this will be the point where the application will fail.

```
std::vector<VkDescriptorSetLayout> layouts(NDescriptorSets, descriptorSetLayout);

VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = NDescriptorSets;
allocInfo.pSetLayouts = layouts.data();

std::vector<VkDescriptorSet> DescriptorSets;
DescriptorSets.resize(NDescriptorSets);

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, DescriptorSets.data());
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate descriptor sets!");
}
```

Memory allocation for Descriptor Sets

Sets are then returned as an array of `VkDescriptorSet` elements. Each element of this array is (just) an handle to the corresponding *Descriptor Set*.

```
std::vector<VkDescriptorSetLayout> layouts(NDescriptorSets, descriptorSetLayout);

VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = NDescriptorSets;
allocInfo.pSetLayouts = layouts.data();

std::vector<VkDescriptorSet> DescriptorSets;
DescriptorSets.resize(NDescriptorSets);

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, DescriptorSets.data());
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate descriptor sets!");
}
```

Descriptor Sets

Descriptor Sets instances the Descriptor Layouts: unless advanced techniques are used, we need at least a Descriptor Set for each different value assigned to a Uniform.

For Uniforms that changes with the Scene, one per scene; for the ones that changes with the material, a Descriptor Set per material, and so on.

The way, in which the Descriptors Sets handles are linked to the corresponding objects, depends on their type: in this lesson we will focus on *Uniform Buffers*, and next time to Images and Textures.

Descriptor Buffers in RAM

First of all, a C++ data structure is created to store the variables that need to be sent to the shader.

This structure will occupy memory in the CPU space (i.e. in RAM).

```
struct UniformBufferObject {
     alignas(16) glm::mat4 mvpMat;
     alignas(16) glm::mat4 mMat;
     alignas(16) glm::mat4 nMat;
};
```

Descriptor Buffer: alignment requirements

For being accessible inside the shader, it must be transferred to GPU accessible memory (i.e. VRAM).

This type memory, might have different memory alignment requirements, which must be respected also inside the C++ version of the structure.

This can be obtained using the `alignas()` C++ command.

```
struct UniformBufferObject {
	alignas(16) glm::mat4 mvpMat;
	alignas(16) glm::mat4 mMat;
	alignas(16) glm::mat4 nMat;
};
```

Descriptor Buffer: alignment requirements

The alignment requirements for the most common data types are:

- `float : alignas(4)`
- `vec2  : alignas(8)`
- `vec3  : alignas(16)`
- `vec4  : alignas(16)`
- `mat3  : alignas(16)`
- `mat4  : alignas(16)`

Memory buffers

Memory buffers allows to store and retrieve information from the GPU accessible video memory.

They are characterized by two handles objects: a `VkBuffer` that identifies the buffer as a whole, and a `VkDeviceMemory` type that describes the corresponding allocated memory.

Since these types of buffers have lots of uses in Vulkan rendering, it is useful to define a specific procedure for creating them.

Memory buffers creation

To create a buffer, three elements are needed:

- The buffer size
- A set of flags determining what is the purpose of this memory area.
- A second set of flags defining which type of memory should be used.

The procedure returns both handles previously mentioned.

```
void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage,
                  VkMemoryPropertyFlags properties,
                  VkBuffer& buffer, VkDeviceMemory& bufferMemory) {
     …
}
```

Memory buffers creation

`size` and `usage` parameters are used to fill the corresponding fields of a `VkBufferCreateInfo`.

The `VkBuffer` object is then created with the `vkCreateBuffer()` command.

```
void createBuffer(…) {
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    VkResult result = vkCreateBuffer(device, &bufferInfo, nullptr, &buffer);
    if (result != VK_SUCCESS) {
        throw std::runtime_error("failed to create vertex buffer!");
    }
    …
}
```

Memory buffers creation

Possible usage flags are the following:

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` and `VK_BUFFER_USAGE_TRANSFER_DST_BIT` for memory transfer.
- `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` for Uniforms.
- `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` for Vertex buffers.
- `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` for Index buffers

For a complete list, see:
https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkBufferUsageFlagBits.html

Memory buffers creation

Each GPU, depending on the usage and on the size of the required buffer, might support such space only in specific types of its memory.

Moreover, for allowing such space, it might change the actual size and require specific alignments.

Vulkan allows Drivers to inform the Application about which type of memory can support the requested buffer with the `vkGetBufferMemoryRequirements()` command.

```
void createBuffer(…) {
    …
    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    …
}
```

Determine the correct memory type

As it was previously outlined, each Physical Device has different types of memory.

It is important to select a memory type that satisfies both user and GPU requirements.

This can be done creating a specific function.

GPU Memory

Memory types and heaps can be inspected with the `vkGetPhyiscalDeviceMemoryProperties()` command, which fills a `VkPhysicalDeviceMemoryPropreties` structure.

```
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &vpdmp);

std::cout << "\n\tMemory Types: " << vpdmp.memoryTypeCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryTypeCount; i++) {
     VkMemoryType vmt = vpdmp.memoryTypes[i];
     …
}

std::cout << "\n\tMemory Heaps: " << vpdmp.memoryHeapCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ) {
     VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
     …
}
```

Marco Gribaudo, assoc.prof. DEIB Dept. POLITECNICO MILANO 1863

Memory buffers creation

The memRequirment structure returned by `vkGetBufferMemoryRequirements` has one field called `memoryTypeBits`, which has one bit per memory type availble in the system.

If the bit at the same position as the index of the corresponding memory type is set, then that memory type can support the requested buffer.

```
void createBuffer(…) {
    …
    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    …
}
```

Determine the correct memory type

The procedure to find the correct storage space (as presented in the Vulkan tutorial), scans all the memory types that support the request, and considers only the ones with the required properties.

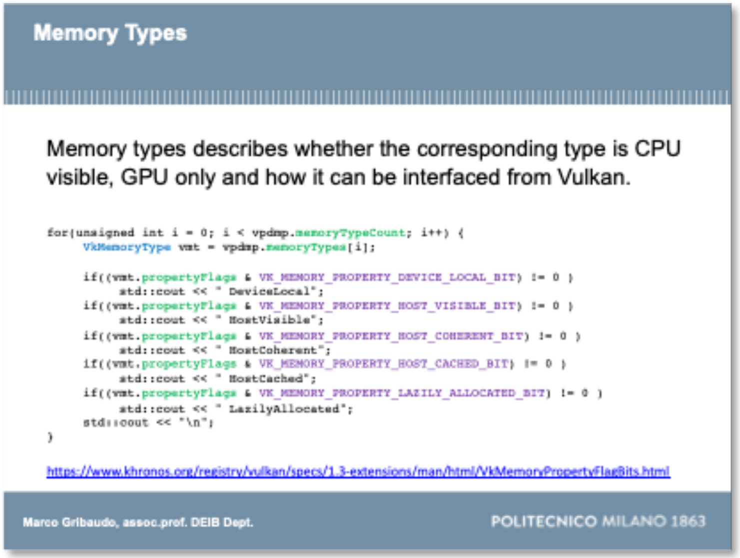
It returns the index of the first type supporting the buffer.

```
uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {
      VkPhysicalDeviceMemoryProperties memProperties;
      vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);

      for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
          if ((typeFilter & (1 << i)) &&
               (memProperties.memoryTypes[i].propertyFlags & properties) == properties) {
               return i;
          }
     }

     throw std::runtime_error("failed to find suitable memory type!");
}
```

Determine the correct memory type

The required memory properties which were briefly outlined when presenting device selection, includes:

- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`: the most efficient memory accessible by the GPU.
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`: the memory can be accessed by the CPU.
- `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`: the memory cache for this area is done automatically by the system.



Memory buffers allocation

The memory information can be used to allocate the actual memory filling a `VkMemoryAllocateInfo` and calling the `vkAllocateMemory()` command.

In particular, this structure requires the actual required memory size, and the index of the selected memory type.

```
void createBuffer(…) {
    …
    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    result = vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory);
    if (result != VK_SUCCESS) {
         throw std::runtime_error("failed to allocate vertex buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Memory buffers allocation

Finally the allocated memory can be associated with the buffer using the `vkBindBufferMemory()` command.

```
void createBuffer(…) {
     …
     VkMemoryAllocateInfo allocInfo{};
     allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
     allocInfo.allocationSize = memRequirements.size;
     allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

     result = vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory);
     if (result != VK_SUCCESS) {
          throw std::runtime_error("failed to allocate vertex buffer memory!");
     }

     vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Memory buffers allocation

Buffers for Uniform variables can then be created using type `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`, and specifying the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`.

```
VkDeviceSize bufferSize = sizeof(UniformBufferObject);

VkBuffer uniformBufferHandle;
VkDeviceMemory uniformBufferMemory;

createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
                         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                         uniformBufferHandle, uniformBuffersMemory);
```

Descriptor Buffer Info

Once the memory area on the graphic adapter, identified by a `VkBuffer` object, has been created, it must be linked with the corresponding *Descriptor Set* handle.

This is accomplished by filling a `VkDescriptorBufferInfo` object, where the VRAM buffer handle is stored into its `buffer` field.

```
VkDescriptorBufferInfo bufferInfo{};
bufferInfo.buffer = uniformBufferHandle;
bufferInfo.offset = 0;
bufferInfo.range = sizeof(UniformBufferObject);
```

Descriptor Writes

Descriptor sets are finally created by filling an array of `VkWriteDescriptorSet`, and calling the `vkUpdateDescriptorSets()` command.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = DescriptorSets[k];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount =  1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = DescriptorSets[k];
descriptorWrites[1].dstBinding = 1;
…
vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
               descriptorWrites.data(), 0, nullptr);
```

Descriptor Writes

Each element of the array connects a shader variable, specifying both its descriptor set, and its binding id.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = DescriptorSets[k];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount =  1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = DescriptorSets[k];
descriptorWrites[1].dstBinding = 1;
…
vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
               descriptorWrites.data(), 0, nullptr);
```

Descriptor Writes

It also specifies the type of descriptor, and the type dependent information.

For uniform buffer, the pointer to the `bufferInfo` data structure previously populated with the `VkBuffer` handle.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = DescriptorSets[k];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount =  1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = DescriptorSets[k];
descriptorWrites[1].dstBinding = 1;
…
vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
               descriptorWrites.data(), 0, nullptr);
```

Copy the Uniform Buffer in the GPU memory

Once the Descriptors have been setup, the application can update them in three steps:

1. Acquiring a pointer to a memory area where the CPU can write the data, using the `vkMapMemory()` command.
2. Filling that memory area with the new values – generally done with a standard `memcpy()` command.
3. Trigger the update of the video memory with the `vkUnmapMemory()` command.

```
void* data;

vkMapMemory(device, uniformBufferMemory, 0, sizeof(ubo), 0, &data);
memcpy(data, &ubo, sizeof(ubo));
vkUnmapMemory(device, uniformBuffersMemory[i]);
```

Copy the Uniform Buffer in the GPU memory

Once all the memory corresponding to Uniform Buffer has been set up, these are the only operations needed to update the values the Shaders will receive!

```
struct UniformBufferObject {
     alignas(16) glm::mat4 mvpMat;
     alignas(16) glm::mat4 mMat;
     alignas(16) glm::mat4 nMat;
} ubo;

Ubo.mMat = glm::mat4(1);

void* data;

vkMapMemory(device, uniformBufferMemory, 0, sizeof(ubo), 0, &data);
memcpy(data, &ubo, sizeof(ubo));
vkUnmapMemory(device, uniformBuffersMemory[i]);
```

Uniforms Binding in Shaders

The shaders must reflect the same data types, in the same order, as the corresponding CPU object.

Application

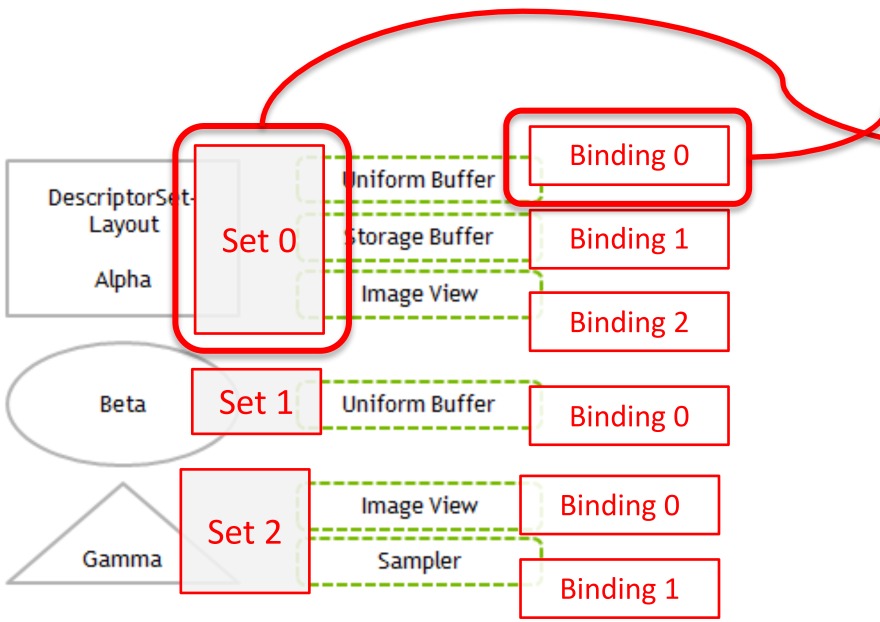
```
struct UniformBufferObject {
	alignas(16) glm::mat4 mvpMat;
	alignas(16) glm::mat4 mMat;
	alignas(16) glm::mat4 nMat;
};
```

Vertex shader

```
layout(set = 0, binding = 0) uniform
UniformBufferObject {
	mat4 mvpMat;
	mat4 mMat;
	mat4 nMat;
} ubo;
layout(location = 0) in vec3 inPosition;
…
// The main procedure
void main() {
	gl_Position = ubo.mvpMat *
				vec4(inPosition, 1.0);
}
```

Uniforms Binding in Shaders

Moreover, they should refer to the same *Set* and *Binding* IDs defined in the application.

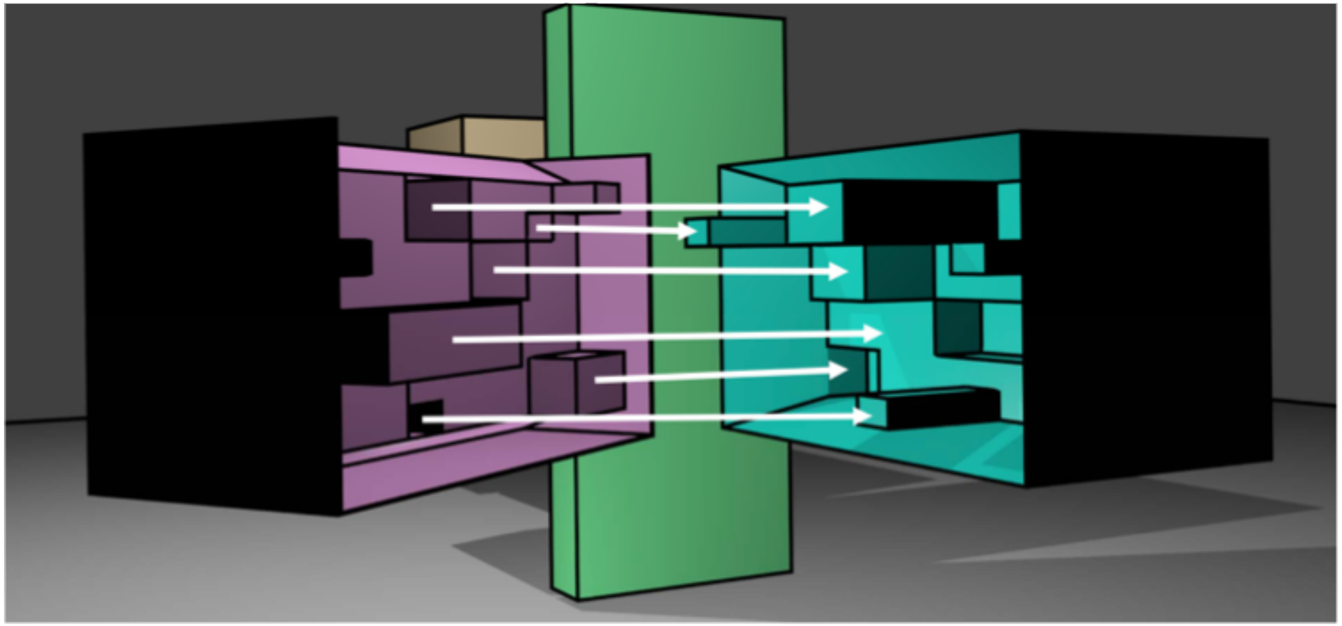


Vertex shader

```
layout(set = 0, binding = 0) uniform
UniformBufferObject {
      mat4 mvpMat;
      mat4 mMat;
      mat4 nMat;
} ubo;
layout(location = 0) in vec3 inPosition;
…
// The main procedure
void main() {
      gl_Position = ubo.mvpMat *
                  vec4(inPosition, 1.0);
}
```

Uniforms Binding in Shaders

In practice, every shader accessing the same data in the same set + layout position can be used as an alternative rendering algorithm.



From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Vertex Buffers

Similar memory allocation procedures should be used also for the *Vertex* and *Index Buffers* containing the mesh definition.

In particular, we can load the *Vertex Buffer* with the following procedure:

```
VkDeviceSize bufferSize = sizeof(Vertices[0]) * Vertices.size();

VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
                         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                         vertexBuffer, vertexBufferMemory);

void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
memcpy(data, Vertices.data(), (size_t) bufferSize);
vkUnmapMemory(device, vertexBufferMemory);
```

Vertex Buffers

In this case, the usage flag must specify that this memory area will be required to store a Vertex Buffer.

The `memcpy()` command will then copy all vertices data, according to the Input assembler specification previously presented.

```
VkDeviceSize bufferSize = sizeof(Vertices[0]) * Vertices.size();

VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
                         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                         vertexBuffer, vertexBufferMemory);

void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
memcpy(data, Vertices.data(), (size_t) bufferSize);
vkUnmapMemory(device, vertexBufferMemory);
```

Index Buffer

In a similar way, we can load the Index Buffer, where the more notable difference with the Index Buffer is the use of the VK_BUFFER_USAGE_INDEX_BUFFER BIT and use of the corresponding array in the `memcpy()` command.

```
VkDeviceSize bufferSize = sizeof(Indices[0]) * Indices.size();

VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;
createBuffer(bufferSize, VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
                         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                         indexBuffer, indexBufferMemory);

void* data;
vkMapMemory(device, indexBufferMemory, 0, bufferSize, 0, &data);
memcpy(data, Indices.data(), (size_t) bufferSize);
vkUnmapMemory(device, indexBufferMemory);
```

Resources release

This procedure allocates a lot of resources that needs to be released at the end of their use:

```
vkDestroyBuffer(device, uniformBufferHandle, nullptr);
vkFreeMemory(device, uniformBuffersMemory, nullptr);

vkDestroyDescriptorPool(device, descriptorPool, nullptr);

vkDestroyBuffer(device, indexBuffer, nullptr);
vkFreeMemory(device, indexBufferMemory, nullptr);

vkDestroyBuffer(device, vertexBuffer, nullptr);
vkFreeMemory(device, vertexBufferMemory, nullptr);
```

Staging buffer

The Vulkan tutorial, improves the previous procedure by using an extra memory area called *the Staging Buffer*.

The idea is to first write the Index and Vertex buffer in a CPU memory visible area, next use the GPU to transfer in one of its private local memory area, which could have a larger capacity and a faster access.

Please refer to the tutorial if you are interested in implementing it.