POLITECNICO MILANO 1863

DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

DEIB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

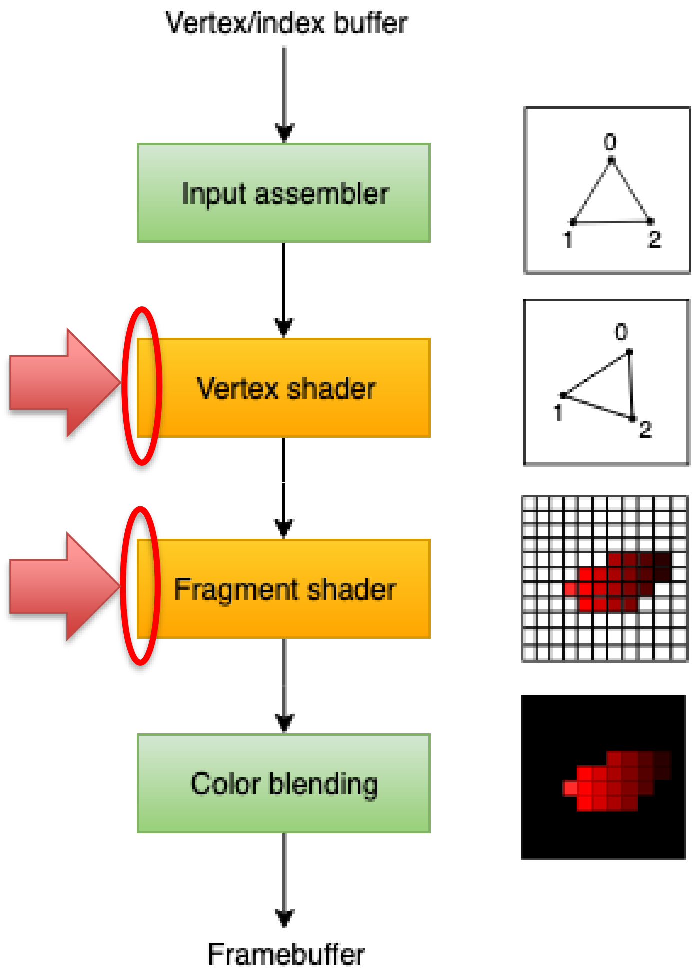
Computer Graphics

- Layouts – Part II

Marco Gribaudo - DEIB

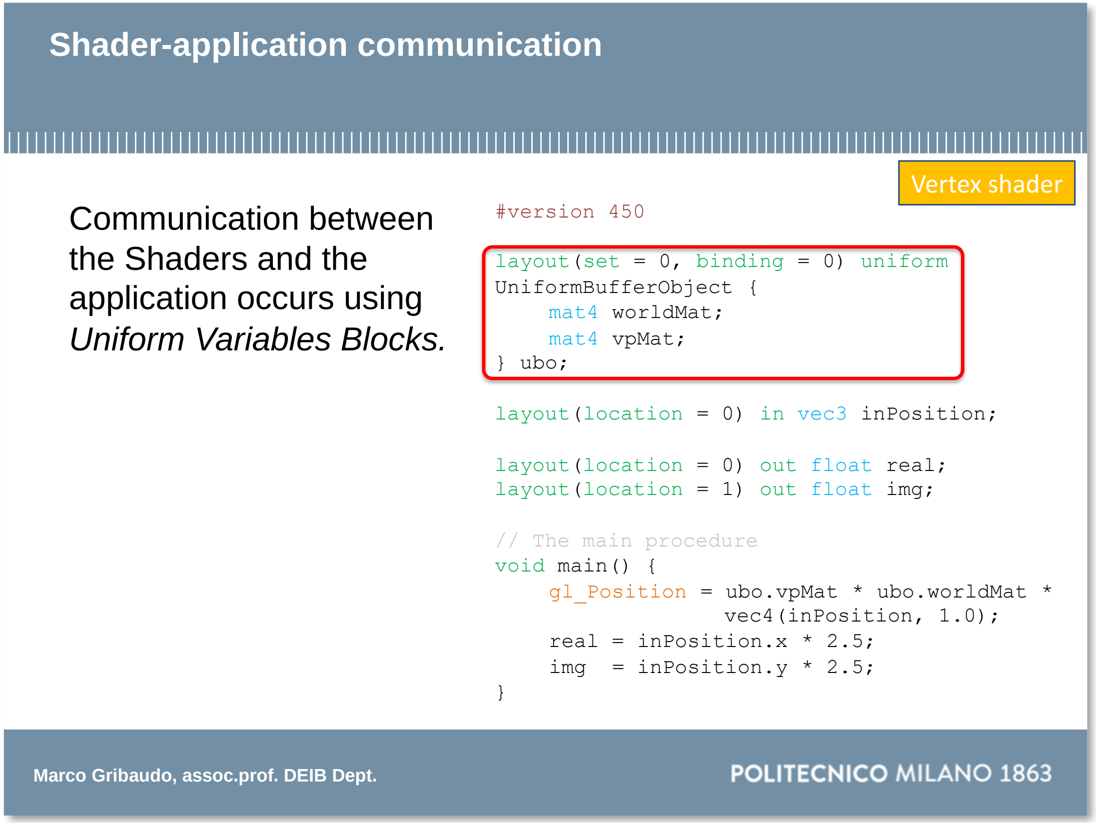
POLITECNICO MILANO 1863

Communication between main application and the Shaders



Uniform buffers

As we have seen when introducing GLSL, application can send scene- and mesh-dependent data to the shaders, 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;
}
```

Uniform buffers

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

Uniform blocks are addressed with two levels of indices.

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 this 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:

- values to configure the BRDF (i.e. GGX or Blinn for Cook-Trorrance)
- values to select debugging views (i.e. 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

In some scenario, the parameters that configures a BRDF, are called *materials*.

Each material, depending on the shader, requires 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 settings 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 has its own properties, which the Shaders use for drawing 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 the “levels” of the frequency at which values are updated.

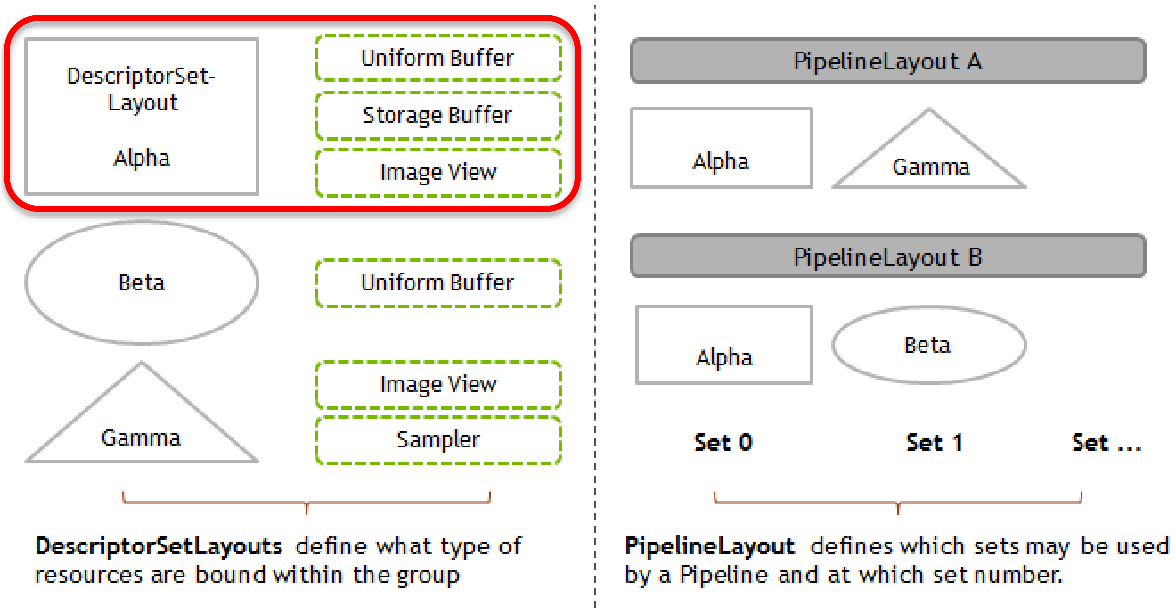
Each Set is characterized by an *ID* (starting from 0): sets with a smaller ID are assumed to change less often.

```
// 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:

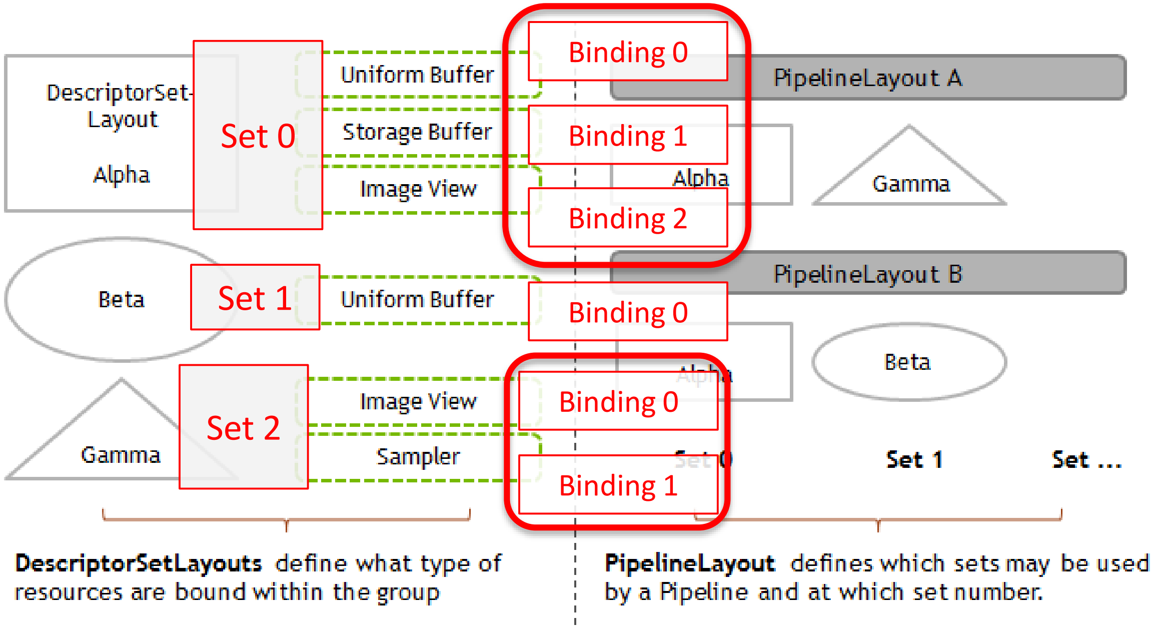
- Uniform blocks with different purposes (i.e. light definitions, environment properties.)
- Textures
- Other data



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

Uniforms: Sets and Bindings

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



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 (we will return on this in a future lesson)
- ...

```
// 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 = V
} VkDescriptorType;
```

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

Sets and layouts

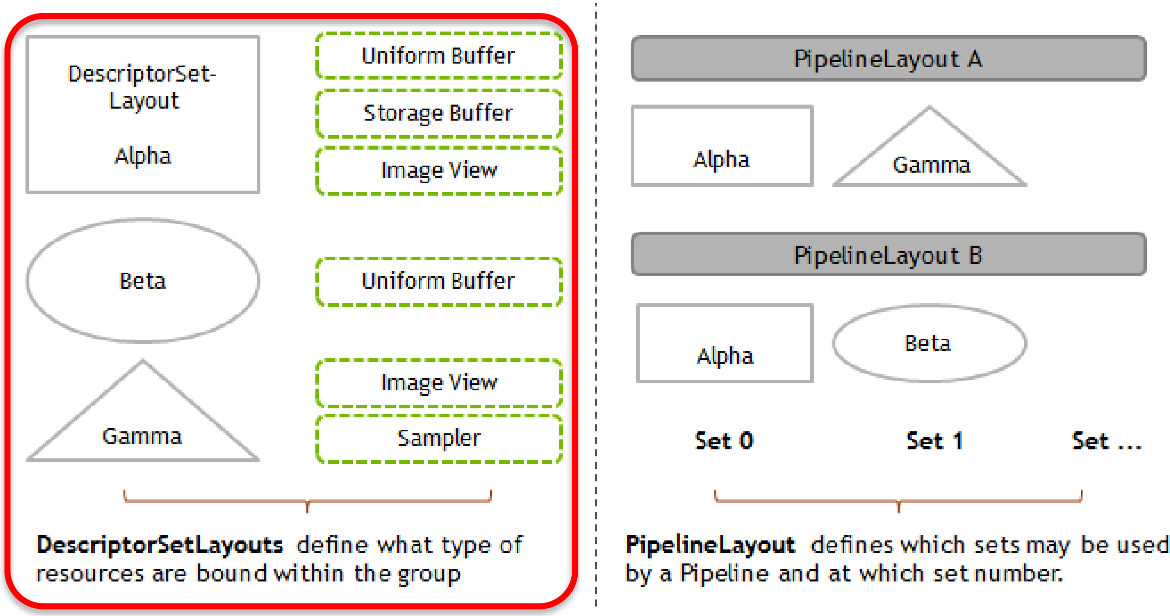
In this context, three relevant definitions are important:

- The *Descriptor Set Layouts*
- The *Descriptor Sets*
- The *Pipeline Layout*

Uniforms: Descriptor Sets Layouts

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

- The type of the descriptors (uniform, texture image, …)
- Their binding ID
- The stage in which they will be used (i.e. *Vertex Shader, Fragment Shader*, or both).



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 specifies its integer ID starting from zero, its type (*Uniform*, *Texture sampler*, etc), and which *Shader Stage* 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 stages 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 Sets Layouts in `Starter.hpp`

In `Starter.hpp` Descriptor Sets Layouts are defined using objects belonging to the `DescriptorSetLayout` class.

```
DescriptorSetLayout DSLBlinn, DSLToon, DSLWard;

    DSLWard.init(this, {
                {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_VERTEX_BIT,
                    sizeof(UniformBufferObject), 1},
                {1, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    0, 1},
                {2, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(GlobalUniformBufferObject), 1},
                {3, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    1, 1}
            });
```

Descriptor Sets Layouts in `Starter.hpp`

Layouts are configured with the `init()` method that receives an array of definitions, with one element per binding.

```
DescriptorSetLayout DSLBlinn, DSLToon, DSLWard;

    DSLWard.init(this, {
                {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_VERTEX_BIT,
                    sizeof(UniformBufferObject), 1},
                {1, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    0, 1},
                {2, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(GlobalUniformBufferObject), 1},
                {3, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    1, 1}
            });
```

Descriptor Sets Layouts in `Starter.hpp`

Each binding is defined by its index (starting from zero), and the Vulkan constants previously introduced, that define its type and the stages where it is used.

```
struct DescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType type;
    VkShaderStageFlags flags;
    int linkSize;
    int count;
};
```

```
{2, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
    VK_SHADER_STAGE_ALL_GRAPHICS,
    sizeof(GlobalUniformBufferObject), 1},
{3, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
    VK_SHADER_STAGE_FRAGMENT_BIT,
    1, 1}
});
```

Descriptor Sets Layouts in `Starter.hpp`

The `linkSize` attribute has a special meaning used to assign values to the field, which will be described later. `count` is used to create arrays of descriptors.

```
struct DescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType type;
    VkShaderStageFlags flags;
    int linkSize;
    int count;
};
```

```
    {2, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
        VK_SHADER_STAGE_ALL_GRAPHICS,
        sizeof(GlobalUniformBufferObject), 1},
    {3, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
        VK_SHADER_STAGE_FRAGMENT_BIT,
        1, 1}
});
```

Uniforms: Descriptor Sets

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 Sets in `Starter.hpp`

In `Starter.hpp` Descriptor Sets are defined with objects of the `DescrptorSet` class.

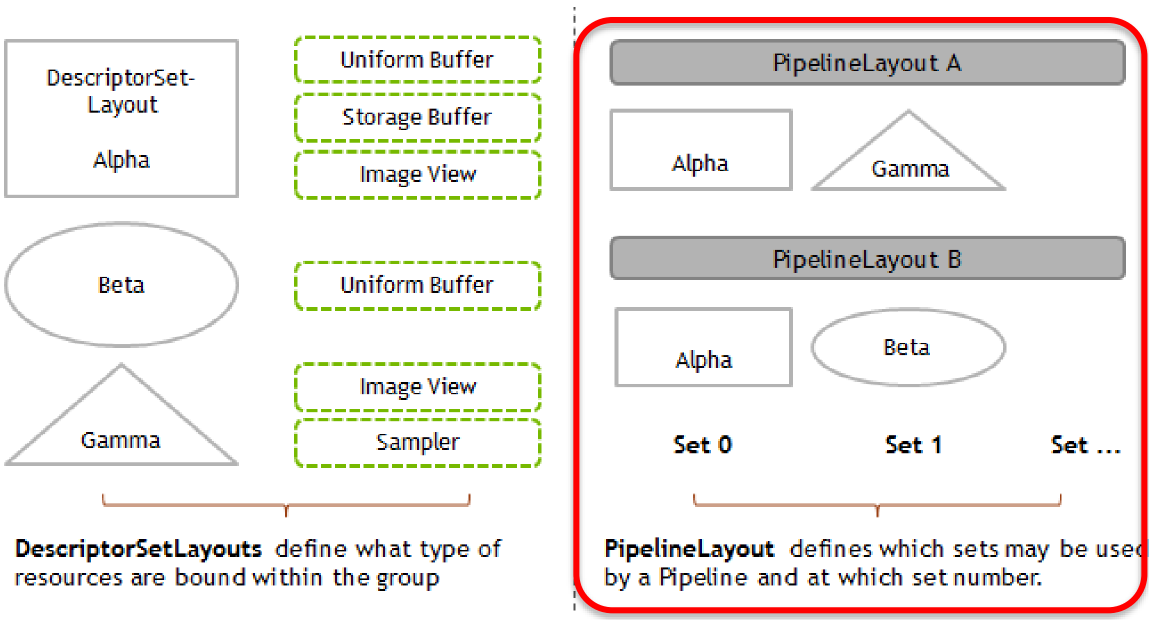
They are configured with the `init()` method, which receives as parameters the pointer to the `DescriptorSetLayout` to which the object belongs, and a texture array on which we will return later, and which can be empty if no binding in the set requires any textures.

```
DescriptorSet DS1, DS2;

    Texture *aT1[] = {&T1[0],  &T1[1],  &T1[2],  &T1[3],  &T1[4],  &T1[5],
                      &T1[6],  &T1[7],  &T1[8],  &T1[9],  &T1[10], &T1[11],
                      &T1[12], &T1[13], &T1[14], &T1[15], &T1[16], &T1[17],
                      &T1[18], &T1[19], &T1[20], &T1[21], &T1[22], &TC};
    DS1.init(this, &DSL1, aT1);
    Texture *aT2[] = {&T2[0],  &T2[1],  &T2[2],  &T2[3],  &T2[4],  &T2[5],
                      &T2[6],  &T2[7],  &T2[8],  &T2[9],  &T2[10], &T2[11],
                      &T2[12], &T2[13], &T2[14], &T2[15], &T2[16], &T2[17],
                      &T2[18], &T2[19], &T2[20], &T2[21], &T2[22], &TC};
    DS2.init(this, &DSL1, aT2);
```

Uniforms: Pipeline Layout

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

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



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 in the pipeline definition, corresponds to the *Set ID* that the code in the Shaders will use to access the corresponding *Descriptor Set*.

```
// 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
      }
    }
  }
}
```

Pipeline Layouts in `Starter.hpp`

In `Starter.hpp` Pipelines Layouts, which will be described in depth in the following lesson, receives during their initialization an array of pointer to the *Descriptor Set Layouts* used by the *Shaders*.

```
// Pipelines [Shader couples]
// The last array, is a vector of pointer to the layouts of the sets that will
// be used in this pipeline. The first element will be set 0, and so on..
P1.init(this, &VD, "shaders/BlinnVert.spv", "shaders/BlinnFrag.spv", {&DSL1});
```

Descriptor layout and pipeline layout

Note that different pipelines can access the same Descriptor Sets at different Set IDs, depending on the order in which they were defined.

Shader FX

```
// example for typical loops in rendering
for each view {
  bind view resources   Set 0   // camera, environment...
  for each shader {
    bind shader pipeline

    for each material {
      bind material resources   Set 1
      for each object {
        bind object resources   Set 2
        draw object
      }
    }
  }
}
```

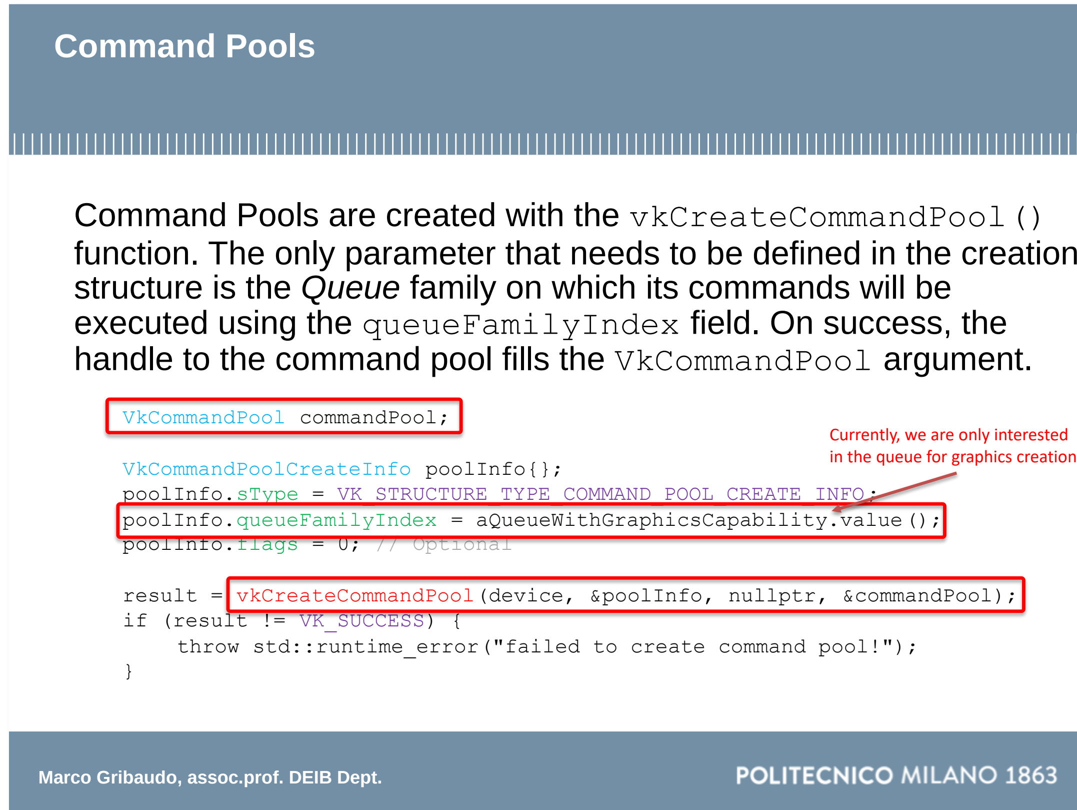
Shader Cook Torrance

```
// 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 {
                                    Set 2   erial parameters and textures
      for each object {
                                    Set 3   ect transforms
        draw object
      }
    }
  }
}
```

Descriptor Pools

Descriptor sets must be allocated from a *Descriptor Pool*, 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.



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 = 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

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

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!");
}
```

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.

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 Pool sizes in `Starter.hpp`

In `Starter.hpp`, the sizes to be used in the creation of the Descriptor Pool, are defined inside the fields of a global object called `DPSZs` (Descriptor Pool SiZes).

```
// Descriptor pool sizes
// WARNING!!!!!!!!
// Must be set before initializing the text
DPSZs.uniformBlocksInPool = 7;
DPSZs.texturesInPool = 52;
DPSZs.setsInPool = 4;
```

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

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: normally, 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: the process however has several similarities with what we have seen for defining the layout of vertices.

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.

Instances of this structure 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.

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]);
```

Updating Descriptor Sets in `Starter.hpp`

In `Starter.hpp`, the size of Uniform Blocks inside Descriptor Sets, is specified in the linkSize attribute of the corresponding line in the definition of the matching Descriptor Set Layout.

```cpp
struct DescriptorSetLayoutBinding {
	uint32_t binding;
	VkDescriptorType type;
	VkShaderStageFlags flags;
	int linkSize;
	int count;
};
```

```cpp
				{2, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
				    VK_SHADER_STAGE_ALL_GRAPHICS,
				    sizeof(GlobalUniformBufferObject), 1},
				{3, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
				    VK_SHADER_STAGE_FRAGMENT_BIT,
				    1, 1}
			});
```

Updating Descriptor Sets in `Starter.hpp`

The application code includes an instance of the object with the CPP version of the Uniform Block data structure.

This instance is filled with the values that needs to be passed to the shader.

```
struct UniformBufferObject {
    alignas(16) glm::mat4 mvpMat;
    alignas(16) glm::mat4 mMat;
    alignas(16) glm::mat4 nMat;
};
    DescriptorSetLayout DSL1;
        DSL1.init(this, {
                    {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                        VK_SHADER_STAGE_ALL_GRAPHICS,
                        sizeof(UniformBufferObject),1},
                    {1, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                        VK_SHADER_STAGE_ALL_GRAPHICS,
                        sizeof(GlobalUniformBufferObject),1},
                    {2, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                        VK_SHADER_STAGE_FRAGMENT_BIT,
                        0,24}
                });
        Texture *aT1[] = {&T1[0],  &T1[1],  &T1[2],  &T1[3],  &T1[4],
                          &T1[6],  &T1[7],  &T1[8],  &T1[9],  &T1[10],
                          &T1[12], &T1[13], &T1[14], &T1[15], &T1[16],
                          &T1[18], &T1[19], &T1[20], &T1[21], &T1[22],
        DS1.init(this, &DSL1, aT1);
        UniformBufferObject ubo{};
        // Here is where you actually update your uniforms

        // updates global uniforms
        ubo.mMat = glm::mat4(1);
        ubo.mvpMat = ViewPrj;
        ubo.nMat = glm::inverse(glm::transpose(ubo.mMat));


        DS1.map(currentImage, &ubo, 0);
    }
```

Updating Descriptor Sets in `Starter.hpp`

The values of the object are then passed to the Shader with the map() method of the corresponding Descriptor Set.

The second parameter, is the pointer to the CPP object that needs to be transferred, and the last value is the corresponding slot.

```
struct UniformBufferObject {
    alignas(16) glm::mat4 mvpMat;
    alignas(16) glm::mat4 mMat;
    alignas(16) glm::mat4 nMat;
};
    DescriptorSetLayout DSL1;
        DSL1.init(this, {
                    {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                        VK_SHADER_STAGE_ALL_GRAPHICS,
                        sizeof(UniformBufferObject),1},
                    {1, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                        VK_SHADER_STAGE_ALL_GRAPHICS,
                        sizeof(GlobalUniformBufferObject),1},
                    {2, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                        VK_SHADER_STAGE_FRAGMENT_BIT,
                        0,24}
                });
        Texture *aT1[] = {&T1[0],  &T1[1],  &T1[2],  &T1[3],  &T1[4],
                          &T1[6],  &T1[7],  &T1[8],  &T1[9],  &T1[10],
                          &T1[12], &T1[13], &T1[14], &T1[15], &T1[16],
                          &T1[18], &T1[19], &T1[20], &T1[21], &T1[22],
        DS1.init(this, &DSL1, aT1);
        UniformBufferObject ubo{};
        // Here is where you actually update your uniforms

        // updates global uniforms
        ubo.mMat = glm::mat4(1);
        ubo.mvpMat = ViewPrj;
        ubo.nMat = glm::inverse(glm::transpose(ubo.mMat));

        DS1.map(currentImage, &ubo, 0);
    }
```

Uniforms Binding in Shaders

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

Application

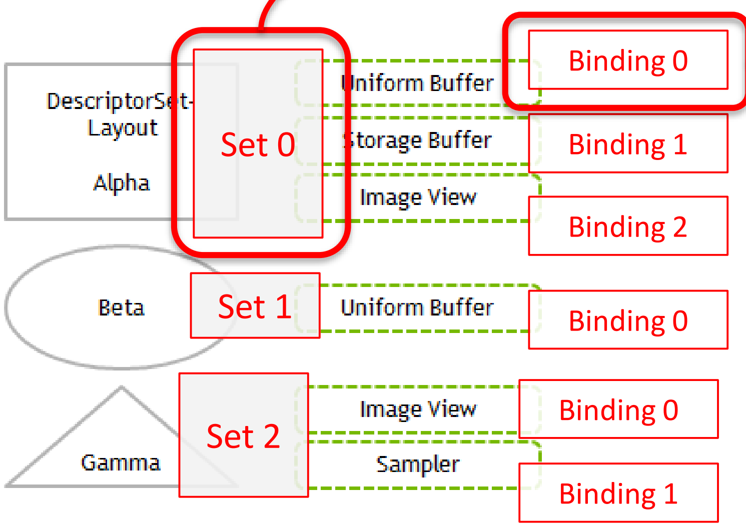
```
struct UniformBufferObject {
	alignas(16) glm::mat4 mvpMat;
	alignas(16) glm::mat4 mMat;
	alignas(16) glm::mat4 nMat;
};
```

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);
}
```

Uniform Layout and Sets definition for textures

Texture are passed as special *Layout Bindings* into *Sets Layouts*.

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};
```

Uniform Layout and Sets definition for textures

As introduced, when creating the *Descriptor Pool*, a special request for the *Combined Image Sampler* must be included to support textures.

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!");
}
```

Uniform Layout and Sets definition for textures

The actual texture pointer is specified when creating the *Descriptor Set* object corresponding to the specified *Descriptor Set Layout*.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
…
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
…
VkDescriptorImageInfo imageInfo{};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType =
                         VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device,
                    static_cast<uint32_t>(descriptorWrites.size()),
                    descriptorWrites.data(), 0, nullptr);
```

Uniform Layout and Sets definition for textures

In this case the pointers to the Sampler and to the Image View of the texture must be provided inside a `VkDescriptorImageInfo` object.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
…
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
…
VkDescriptorImageInfo imageInfo{};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType =
                         VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device,
                    static_cast<uint32_t>(descriptorWrites.size()),
                    descriptorWrites.data(), 0, nullptr);
```

Uniform Layout and Sets definition for textures

The `descriptorWrite` object, beside the correct binding, specifies that this descriptor is a *Combined Image Sampler*.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
…
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
…
VkDescriptorImageInfo imageInfo{};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType =
                         VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device,
                    static_cast<uint32_t>(descriptorWrites.size()),
                    descriptorWrites.data(), 0, nullptr);
```

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);

}
```

Textures in Shaders

A lot of different samplers exists. The most important ones are:

- `sampler1D`
- `sampler2D`
- `sampler3D`
- `samplerCube`

Textures in Shaders

The shader, can obtain the color at a given position with the `texture()` command.

```
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);

}
```

Textures in Shaders

The first parameter determines the images, and the second defines te coordinates of the texel (with a format dependent on the sampler type).

The function returns a `vec4` color, where the last component is the alpha channel (transparency).

```
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);

}
```

Textures in Shaders

For Mip-mapped images, we can use the `textureLod()` command to read from an image at a specific minification level.

```
textureLod(sampler, pos, lod)
```

The `sample` and `pos` parameters are identical to the conventional version of the command.

The `lod` parameter addresses the image with the highest detail with *0*, the first reduction with *1*, the second reduction with *2*, and so on up to the number of available levels.

Separate Texture and Sampler

Vulkan also allows to specify the texture and its sampler in two different uniforms.

We consider this opportunity outside the scope of this course, and we will not investigate it.

19 lines (17 sloc) | 721 Bytes

```
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout (set = 0, binding = 1) uniform texture2D tex;
layout (set = 0, binding = 2) uniform sampler samp;
layout (location = 0) in vec2 inTexCoords;
layout (location = 0) out vec4 outColor;
void main() {

     // Combine the selected texture with sampler as a parameter
     vec4 resColor = texture(sampler2D(tex, samp), inTexCoords);

     // Create a border to see the cube more easily
    if (inTexCoords.x < 0.01 || inTexCoords.x > 0.99)
        resColor *= vec4(0.1, 0.1, 0.1, 1.0);
    if (inTexCoords.y < 0.01 || inTexCoords.y > 0.99)
        resColor *= vec4(0.1, 0.1, 0.1, 1.0);
    outColor = resColor;
}
```

Texturing

Each *Graphics Adapter* has its own internal format for images:

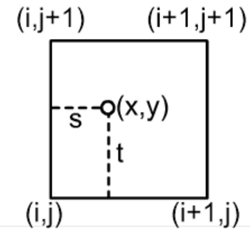
- Resolution per color component (8, 10, 12, 16 bit or floating point)
- Texture sizes restrictions (i.e. only power of 2 or multiple of a given size)
- In memory data organization (i.e. per row, per column, per block …)

Texturing

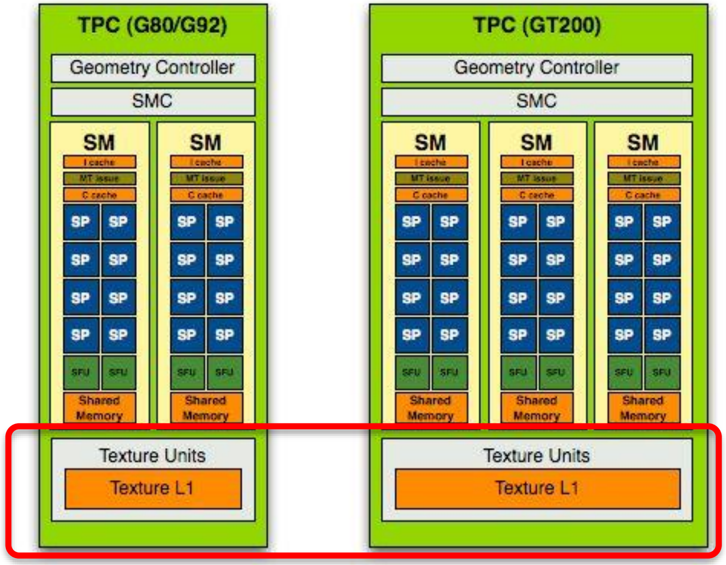
Texture sampling, as we have seen, requires a lot of interpolations and other floating point operations.

Special hardware blocks on the GPU, called *Texture Units*, can help handling texture sampling.

The source image given by the application, however, almost never corresponds to the format internally used by the GPU.



$$p' = (1-s)\,(1-t)\,p_{i,j} + s\,(1-t)\,p_{i+1,j} + \\ +(1-s)\,t\,p_{i,j+1} + s\,t\,p_{i+1,j+1}$$



Texturing

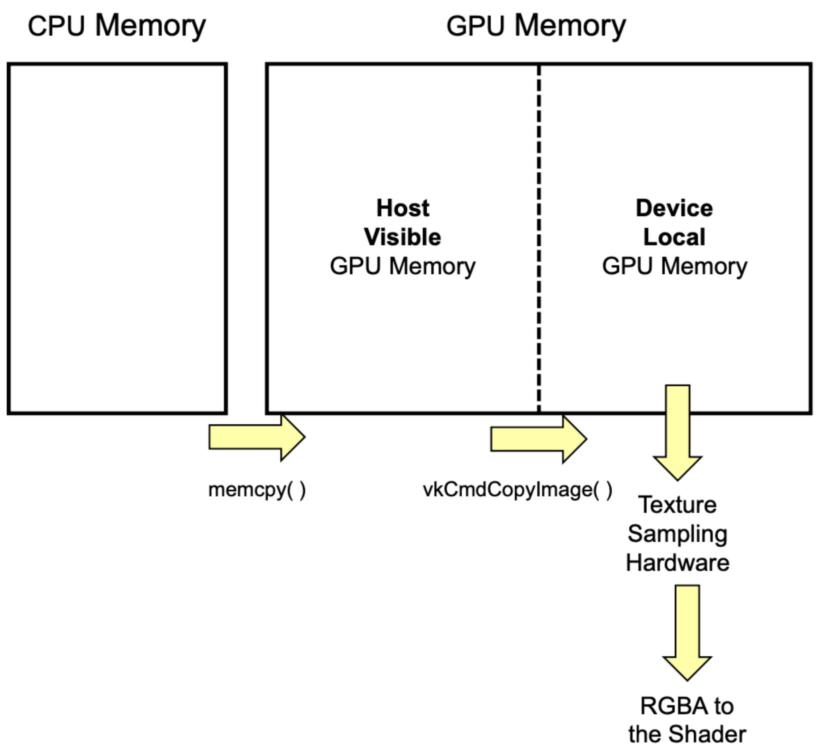
The input texture data must then be converted in the format that the GPU and its Texture Units can handle more efficiently.

In OpenGL and other graphic sub-system this was done automatically by the drivers and it does not require any special support from the developer.

Vulkan requires programmers to explicitly implement this process.

Texturing

In particular, the following steps must be accomplished:



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

Texturing

Unfortunately, Vulkan Developer have chosen a very arguable terminology which is somehow confusing:

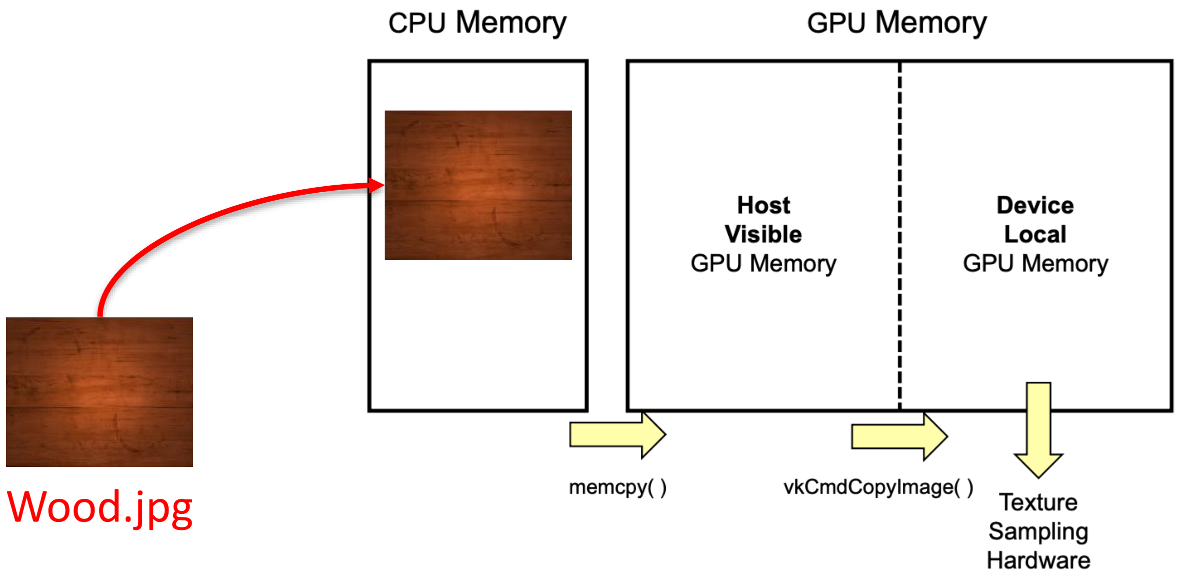
- Organization of texels in memory and their color specification formats are again called *Layout*.
- The verb “to *Transition*” has been used to define the process of converting a texture format from one *Layout* to another. Although grammatically correct, it was not the best choice for non-native speaker, who associate the word “transition” more naturally to a noum rather than a verb.

Texturing

Especially, for the Italians, where the verb “transizionare” has a totally different meaning…

Texture preparation: loading

The first step for defining a texture, is loading the corresponding image into a memory area accesible by the CPU.



Texture preparation: loading

This can be done using special libraries, such as `stb_image`:

https://github.com/nothings/stb/blob/master/stb_image.h

In particular, the `stbi_load()` command, returns an array of RGBA pixels and the size information of the texture.

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

…

int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight,
                         &texChannels, STBI_rgb_alpha);
if (!pixels) {
     throw std::runtime_error("failed to load texture image!");
}

VkDeviceSize imageSize = texWidth * texHeight * 4;
mipLevels = static_cast<uint32_t>(std::floor(
                    std::log2(std::max(texWidth, texHeight)))) + 1;
```

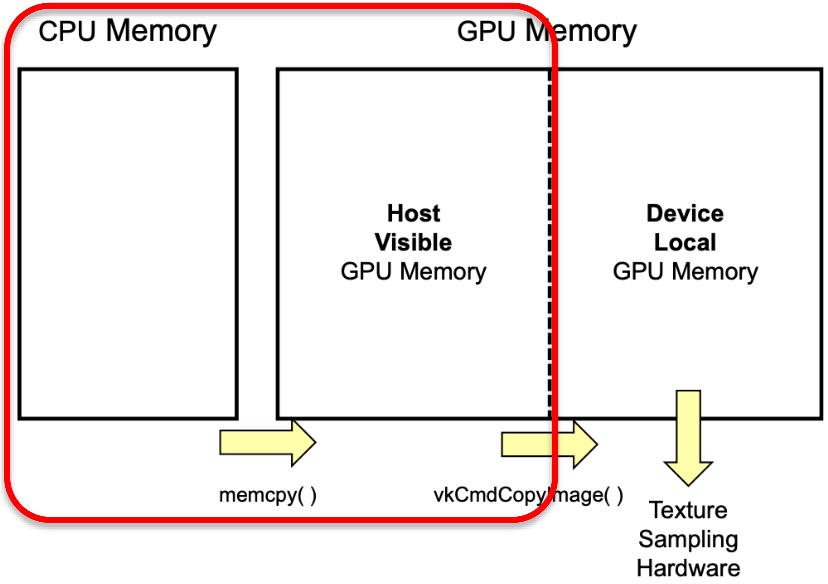
Texture preparation: loading

Size measurements can be used to determine other useful information such as the memory requirements and the number of Mip-Map levels.

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>

…

int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight,
                            &texChannels, STBI_rgb_alpha);
if (!pixels) {
     throw std::runtime_error("failed to load texture image!");
}

VkDeviceSize imageSize = texWidth * texHeight * 4;
mipLevels = static_cast<uint32_t>(std::floor(
                    std::log2(std::max(texWidth, texHeight)))) + 1;
```

Texture preparation: staging buffer

Next, the texture needs to be moved inside a memory area accessible by the GPU. This area is known as the *Staging Buffer*.



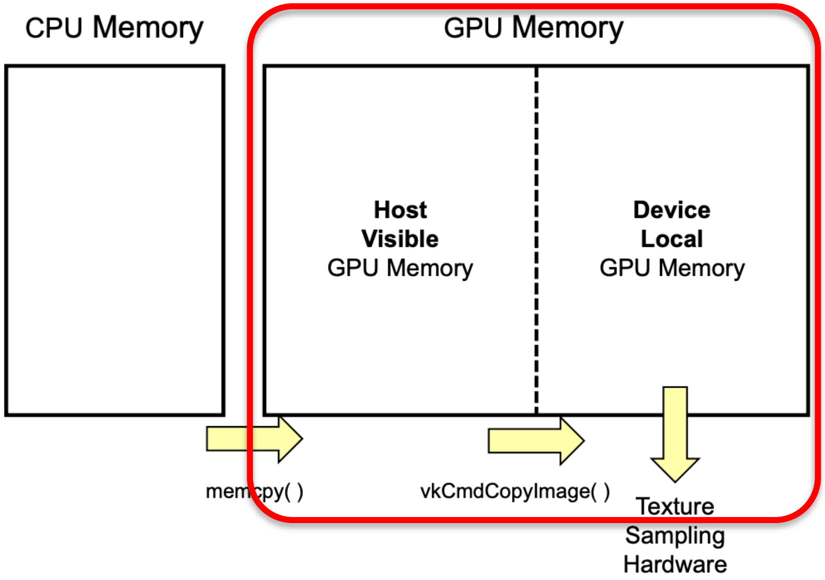
Texture preparation: staging buffer

Note that after the image has been put inside the staging buffer, its source memory area can be released.

```
VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;

createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
                             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                             VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                             stagingBuffer, stagingBufferMemory);
void* data;
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
memcpy(data, pixels, static_cast<size_t>(imageSize));
vkUnmapMemory(device, stagingBufferMemory);

stbi_image_free(pixels);
```

Texture preparation: staging buffer

Finally, the texture must be moved into its specific memory area, and its format converted into one known by the GPU.



Mip-Mapping

In older engines such as OpenGL, generation of Mip-Maps was left to the Driver, with the opportunity for the developer to pass the Mip-Map layer directly.

In Vulkan, everything must be done by the user.

Mip-Mapping

The Vulkan tutorial suggestes a procedure for generating the Mip-Maps.

```
void generateMipmaps(VkImage image, VkFormat imageFormat,
                          int32_t texWidth, int32_t texHeight,
                          uint32_t mipLevels) {
     …
     VkCommandBuffer commandBuffer = beginSingleTimeCommands();
     …
     for (uint32_t i = 1; i < mipLevels; i++) {
          …
          VkImageBlit blit{};
          …
          vkCmdBlitImage(commandBuffer, image,
               VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, image,
               VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
               &blit, VK_FILTER_LINEAR);
          …

     }
     endSingleTimeCommands(commandBuffer);
}
```

Sampler creation

Next, the Sampler for the texture must be defined. It will be contained inside a `VkSampler` object, created using the `VkCreateSampler()` function.

```
VkSamplerCreateInfo samplerInfo{};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
samplerInfo.unnormalizedCoordinates = VK_FALSE;
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = static_cast<float>(mipLevels);

VkResult result = vkCreateSampler(device, &samplerInfo,
                                  nullptr, &textureSampler);
if (result != VK_SUCCESS) {
     PrintVkError(result);
}
```

Sampler creation

Minification and Magnification filters can be specified.

Use of Mip-Maps or Anisotropic Filter are specified in separate fields.

Possible values for the filters type are:

- `VK_FILTER_NEAREST`
- `VK_FILTER_LINEAR`

```
VkSamplerCreateInfo samplerInfo{};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
samplerInfo.unnormalizedCoordinates = VK_FALSE;
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = static_cast<float>(mipLevels);

VkResult result = vkCreateSampler(device, &samplerInfo,
                                   nullptr, &textureSampler);
if (result != VK_SUCCESS) {
     PrintVkError(result);
}
```

Sampler creation

Since texture can be 3D, three behaviors for specifying what to do for values outside the [0,1] range have to be specified.

```
VkSamplerCreateInfo samplerInfo{};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
samplerInfo.unnormalizedCoordinates = VK_FALSE;
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = static_cast<float>(mipLevels);

VkResult result = vkCreateSampler(device, &samplerInfo,
                                  nullptr, &textureSampler);
if (result != VK_SUCCESS) {
     PrintVkError(result);
}
```

Sampler creation

Valid modes are the following:

- VK_SAMPLER_ADDRESS_MODE_REPEAT
- VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

They support respectively repeat, mirror and clamp behaviors.

Textures in Starter.hpp

Textures are created in `Starter.hpp` as instances of the `Texture` class. The `init()` method specifies the name of the file with the image to load.

```
DescriptorSetLayout DSL1;
DescriptorSet DS1, DS2;
Texture T1[23], T2[23], TC;
    DSL1.init(this, {
                {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(UniformBufferObject),1},
                {1, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(GlobalUniformBufferObject),1},
                {2, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    0,24}
            });
    for(int i = 0; i < 23; i++) {
        T2[i].init(this, TexNames[i]);
    }
    TC.init(this, "textures/IntGradient.png");
    Texture *aT2[] = {&T2[0],  &T2[1],  &T2[2],  &T2[3],  &T2[4],  &T2[5],
                      &T2[6],  &T2[7],  &T2[8],  &T2[9],  &T2[10], &T2[11],
                      &T2[12], &T2[13], &T2[14], &T2[15], &T2[16], &T2[17],
                      &T2[18], &T2[19], &T2[20], &T2[21], &T2[22], &TC};
    DS2.init(this, &DSL1, aT2);
```

Textures in Starter.hpp

Each time a Descriptor Set is initialized, and array of texture is passed: this array contains a pointer to all the textures used in the corresponding Shader.

```
DescriptorSetLayout DSL1;
DescriptorSet DS1, DS2;
Texture T1[23], T2[23], TC;
    DSL1.init(this, {
                {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(UniformBufferObject),1},
                {1, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(GlobalUniformBufferObject),1},
                {2, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    0,24}
            });
    for(int i = 0; i < 23; i++) {
        T2[i].init(this, TexNames[i]);
    }
    TC.init(this, "textures/IntGradient.png");
    Texture *aT2[] = {&T2[0],  &T2[1],  &T2[2],  &T2[3],  &T2[4],  &T2[5],
                      &T2[6],  &T2[7],  &T2[8],  &T2[9],  &T2[10], &T2[11],
                      &T2[12], &T2[13], &T2[14], &T2[15], &T2[16], &T2[17],
                      &T2[18], &T2[19], &T2[20], &T2[21], &T2[22], &TC};
    DS2.init(this, &DSL1, aT2);
```

Textures in Starter.hpp

When the Descriptor Set Layout is created, the `linkSize` field for a texture binding describes the position in the array passed to the Descriptor Set initialization, where the corresponding image can be found.

```
DescriptorSetLayout DSL1;
DescriptorSet DS1, DS2;
Texture T1[23], T2[23], TC;
    DSL1.init(this, {
                {0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(UniformBufferObject),1},
                {1, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
                    VK_SHADER_STAGE_ALL_GRAPHICS,
                    sizeof(GlobalUniformBufferObject),1},
                {2, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
                    VK_SHADER_STAGE_FRAGMENT_BIT,
                    0, 24}
            });
    for(int i = 0; i < 23; i++) {
        T2[i].init(this, TexNames[i]);
    }
    TC.init(this, "textures/IntGradient.png");
    Texture *aT2[] = {&T2[0],  &T2[1],  &T2[2],  &T2[3],  &T2[4],  &T2[5],
                      &T2[6],  &T2[7],  &T2[8],  &T2[9],  &T2[10], &T2[11],
                      &T2[12], &T2[13], &T2[14], &T2[15], &T2[16], &T2[17],
                      &T2[18], &T2[19], &T2[20], &T2[21], &T2[22], &TC};
    DS2.init(this, &DSL1, aT2);
```

Texture Samplesrs in Starter.hpp

Texture samplers can be modified, by passing a fourth value equal to **`false`** during texture initialization.

```
T1[i].init(this, TexNames[i], VK_FORMAT_R8G8B8A8_SRGB, false);
T1[i].createTextureSampler(
                          VK_FILTER_NEAREST,
                          VK_FILTER_NEAREST,
                          VK_SAMPLER_ADDRESS_MODE_REPEAT,
                          VK_SAMPLER_ADDRESS_MODE_REPEAT,
                          VK_SAMPLER_MIPMAP_MODE_NEAREST,
                          VK_FALSE, 1, 0
);
```

Texture Samplesrs in Starter.hpp

The texture sampler can then be configured with the `createTextureSampler()` method.

```
void Texture::createTextureSampler(
        VkFilter magFilter = VK_FILTER_LINEAR,
        VkFilter minFilter = VK_FILTER_LINEAR,
        VkSamplerAddressMode addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT,
        VkSamplerAddressMode addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT,
        VkSamplerMipmapMode mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR,
        VkBool32 anisotropyEnable = VK_TRUE,
        float maxAnisotropy = 16,
        float maxLod = -1
    ) {
```

```
T1[i].init(this, TexNames[i], VK_FORMAT_R8G8B8A8_SRGB, false);
T1[i].createTextureSampler(
                    VK_FILTER_NEAREST,
                    VK_FILTER_NEAREST,
                    VK_SAMPLER_ADDRESS_MODE_REPEAT,
                    VK_SAMPLER_ADDRESS_MODE_REPEAT,
                    VK_SAMPLER_MIPMAP_MODE_NEAREST,
                    VK_FALSE, 1, 0
);
```

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568
marco.gribaudo@polimi.it
https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)