



**POLITECNICO**  
MILANO 1863

DIPARTIMENTO DI ELETTRONICA  
INFORMAZIONE E BIOINGEGNERIA



**2024**

# Dipartimento di Elettronica, Informazione e Bioingegneria

## *Computer Graphics*

Milano, 2024

# Computer Graphics

- Pipeline creation and Commands execution



# Loading the Shaders

As we have seen, Vulkan can load the shaders from SPIR-V binary files.

We have seen how to compile them outside the application.

Here we see how to load them in the application code, and prepare them for being linked to a pipeline.

# Loading the Shaders

The SPIR-V can be loaded into a byte array using conventional C++ functions. In particular, we can create a `readFile()` function that receive as input the `filename` and returns a `char` array containing it.

```
static std::vector<char> readFile(const std::string& filename) {  
    std::ifstream file(filename, std::ios::ate | std::ios::binary);  
    if (!file.is_open()) {  
        throw std::runtime_error("failed to open file!");  
    }  
  
    size_t fileSize = (size_t) file.tellg();  
    std::vector<char> buffer(fileSize);  
  
    file.seekg(0);  
    file.read(buffer.data(), fileSize);  
  
    file.close();  
  
    return buffer;  
}
```

# Loading the Shaders

The `std::ios::ate` opening mode and the `tellg()` method can be used to determine the file size.

```
static std::vector<char> readFile(const std::string& filename) {  
    std::ifstream file(filename, std::ios::ate | std::ios::binary);  
    if (!file.is_open()) {  
        throw std::runtime_error("failed to open file!");  
    }  
  
    size_t fileSize = (size_t) file.tellg();  
    std::vector<char> buffer(fileSize);  
  
    file.seekg(0);  
    file.read(buffer.data(), fileSize);  
  
    file.close();  
  
    return buffer;  
}
```

# Loading the Shaders

In this way, a properly sized buffer can be created, and used to store the binary code of the shader read from the file.

```
static std::vector<char> readFile(const std::string& filename) {  
    std::ifstream file(filename, std::ios::ate | std::ios::binary);  
    if (!file.is_open()) {  
        throw std::runtime_error("failed to open file!");  
    }  
  
    size_t fileSize = (size_t) file.tellg();  
    std::vector<char> buffer(fileSize);  
  
    file.seekg(0);  
    file.read(buffer.data(), fileSize);  
  
    file.close();  
  
    return buffer;  
}
```

# Create the Shader Module

The binary code can be used to create a *Shader Module*, the data structure Vulkan uses to access the shaders. Since we need one module per Shader, it is handy to create a procedure to perform this task.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {  
    VkShaderModuleCreateInfo createInfo{};  
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;  
    createInfo.codeSize = code.size();  
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());  
  
    VkShaderModule shaderModule;  
  
    VkResult result = vkCreateShaderModule(device, &createInfo, nullptr,  
                                           &shaderModule);  
    if (result != VK_SUCCESS) {  
        throw std::runtime_error("failed to create shader module!");  
    }  
  
    return shaderModule;  
}
```

# Create the Shader Module

Shader modules handles are stored into `VkShaderModule` objects, created with the `vkCreateShaderModule()` function, after filling a `VkShaderModuleCreateInfo` structure.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {  
    VkShaderModuleCreateInfo createInfo{};  
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;  
    createInfo.codeSize = code.size();  
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());  
  
    VkShaderModule shaderModule;  
  
    VkResult result = vkCreateShaderModule(device, &createInfo, nullptr,  
                                           &shaderModule);  
    if (result != VK_SUCCESS) {  
        throw std::runtime_error("failed to create shader module!");  
    }  
  
    return shaderModule;  
}
```



# Create the Shader Module

Since SPIR-V files contain just an intermediate binary representation of the Shader programs, they are no longer necessary at the end of the pipeline creation process. For this reason they should be destroyed once the pipeline has been created using the `vkDestroyShaderModule()` function.

```
void createPipeline(...) {
    auto vertShaderCode = readFile((SHADER_PATH + VertexShaderName).c_str());
    auto fragShaderCode = readFile((SHADER_PATH + FragShaderName).c_str());

    VkShaderModule vertShaderModule =
        createShaderModule(vertShaderCode);
    VkShaderModule fragShaderModule =
        createShaderModule(fragShaderCode);

    ...

    vkDestroyShaderModule(device, fragShaderModule, nullptr);
    vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

# Configuring the Shader Stages

Shaders are then used in the pipeline creation process as an array of `VkPipelineShaderStageCreateInfo` objects.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};  
vertShaderStageInfo.sType =  
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;  
vertShaderStageInfo.module = vertShaderModule;  
vertShaderStageInfo.pName = "main";
```

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo{};  
fragShaderStageInfo.sType =  
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;  
fragShaderStageInfo.module = fragShaderModule;  
fragShaderStageInfo.pName = "main";
```

```
VkPipelineShaderStageCreateInfo shaderStages[] =  
    {vertShaderStageInfo, fragShaderStageInfo};
```

# Configuring the Shader Stages

The `stage` field of each element defines which type of shader is contained in the module: `VK_SHADER_STAGE_VERTEX_BIT` for the *Vertex*, and `VK_SHADER_STAGE_FRAGMENT_BIT` for the *Fragment*.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
    {vertShaderStageInfo, fragShaderStageInfo};
```

# Configuring the Shader Stages

The `module` field contain a pointer to the corresponding *Shader Module* previously created.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] =
    {vertShaderStageInfo, fragShaderStageInfo};
```

# Configuring the Shader Stages

Each shader might have several entry points. The `pName` field contains the name of the procedure that must be called to perform the corresponding function. Usually, this will be `"main"`.

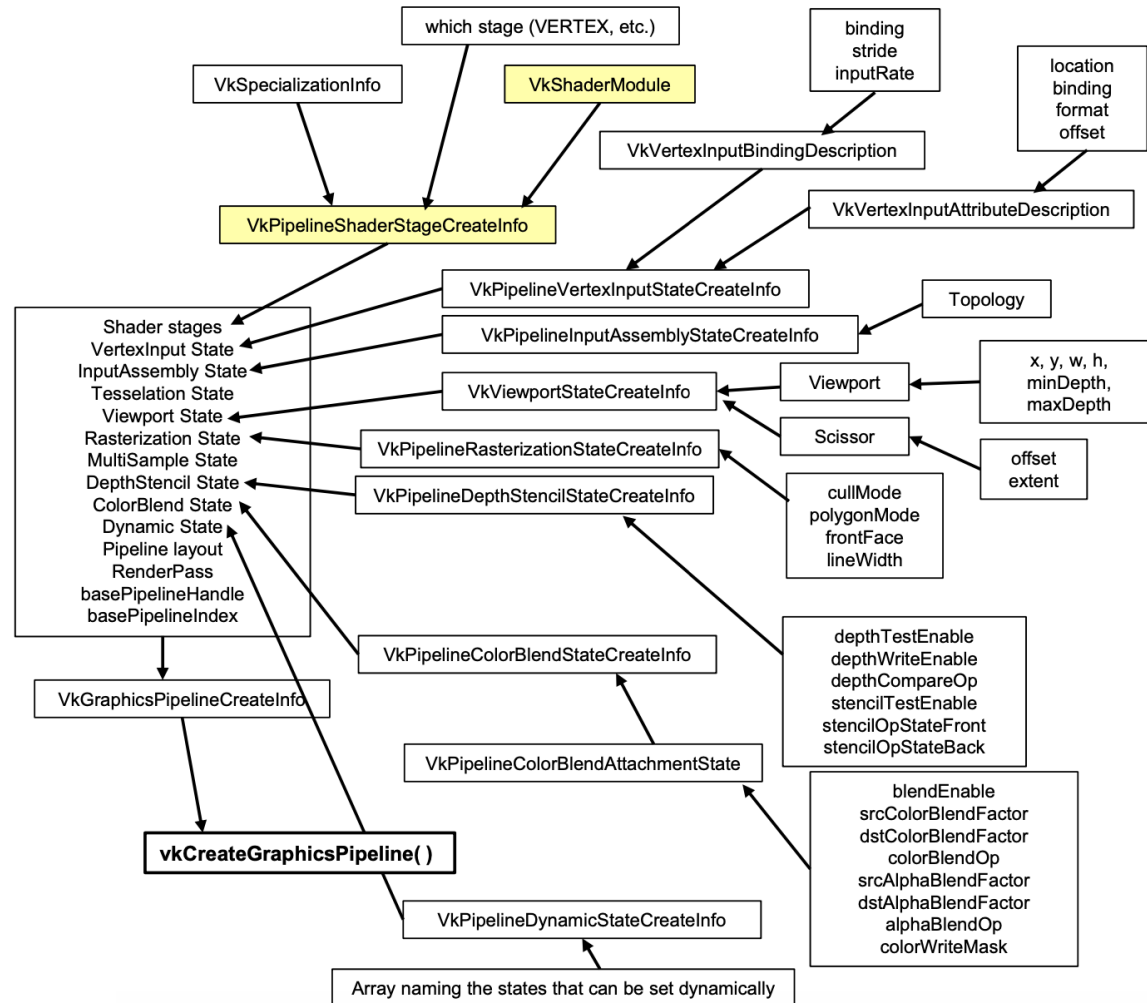
```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";
```

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";
```

```
VkPipelineShaderStageCreateInfo shaderStages[] =
    {vertShaderStageInfo, fragShaderStageInfo};
```

# Pipeline creation

The Vulkan pipeline is the most complex and important data structure of the entire 3D visualization process.

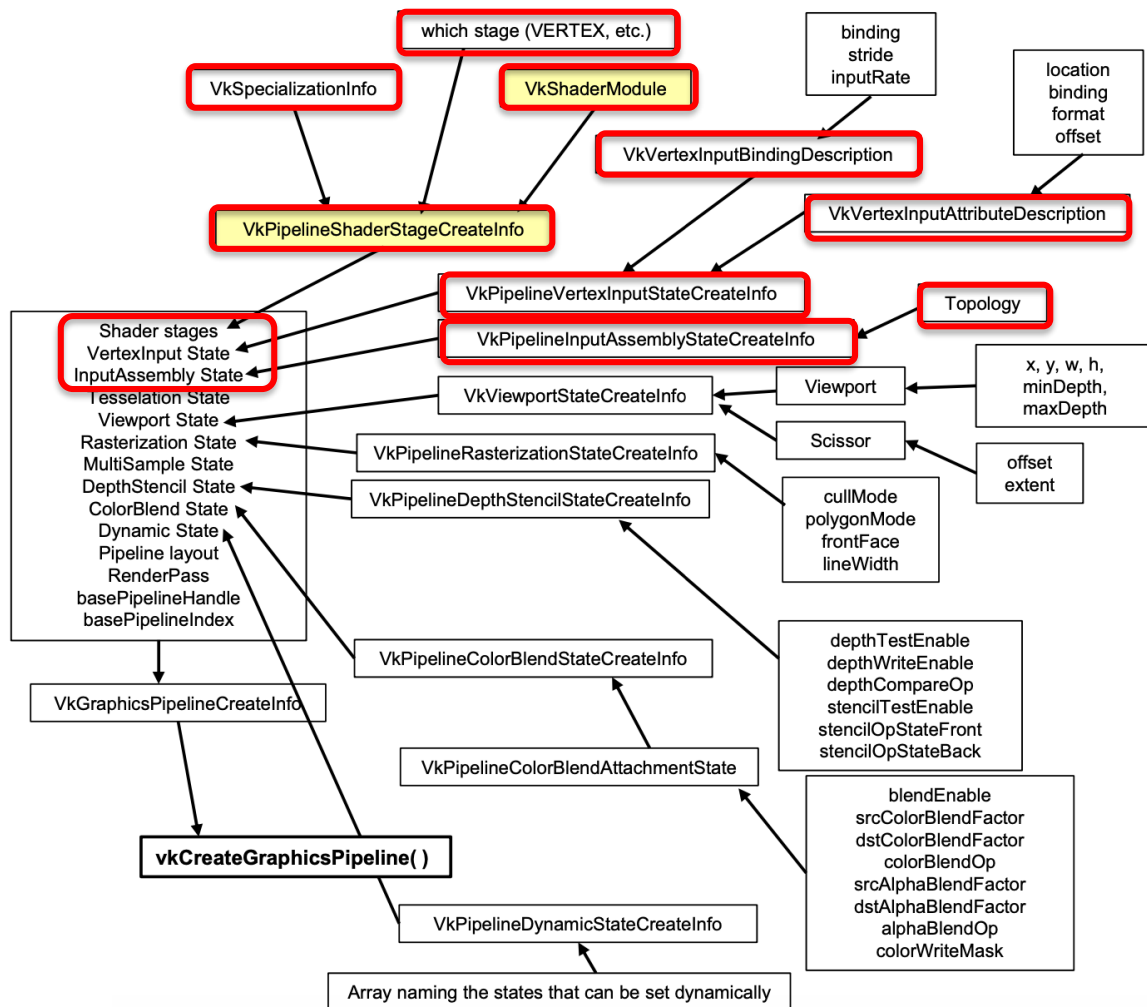


From: <https://web.engr.oregonstate.edu/~mjb/vulkan/>

# Pipeline creation

We have already seen how to configure the *Input Assembler*, the *Vertex Input*, and the *Shader stages*.

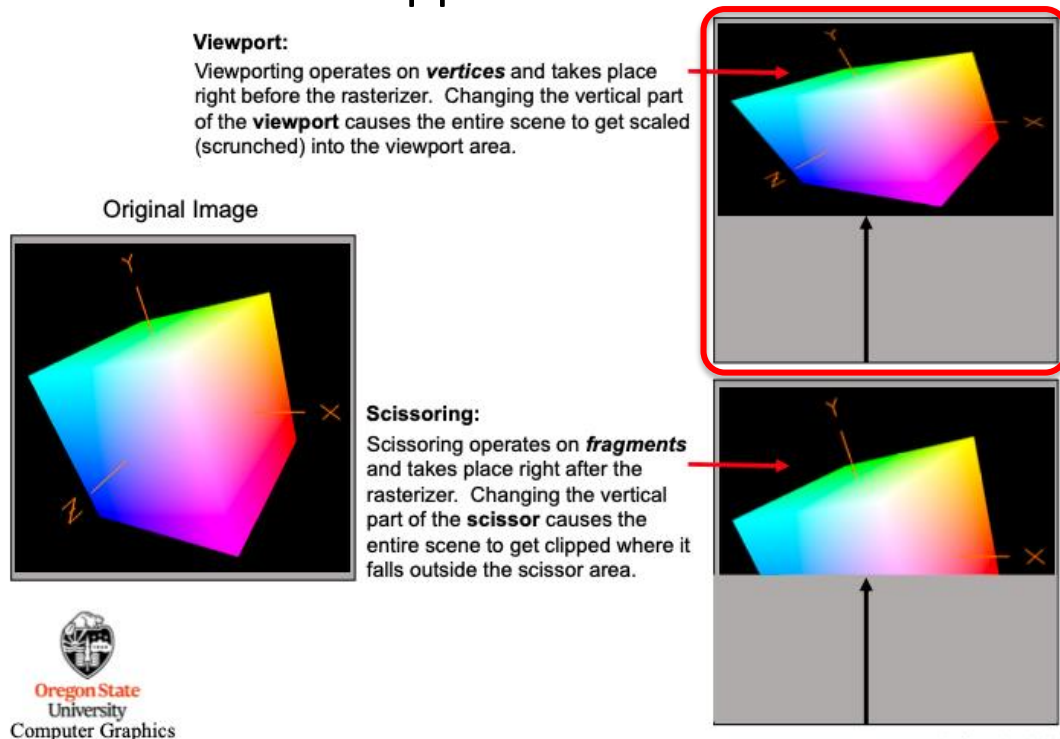
Now we will see how to complete the pipeline creation process.



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

# Viewport

A *Viewport* defines a rectangular area on the surface that will be used by the rendering system: i.e. where the  $(-1,-1)$  and  $(1,1)$  normalized screen coordinates will be mapped.





# Viewport

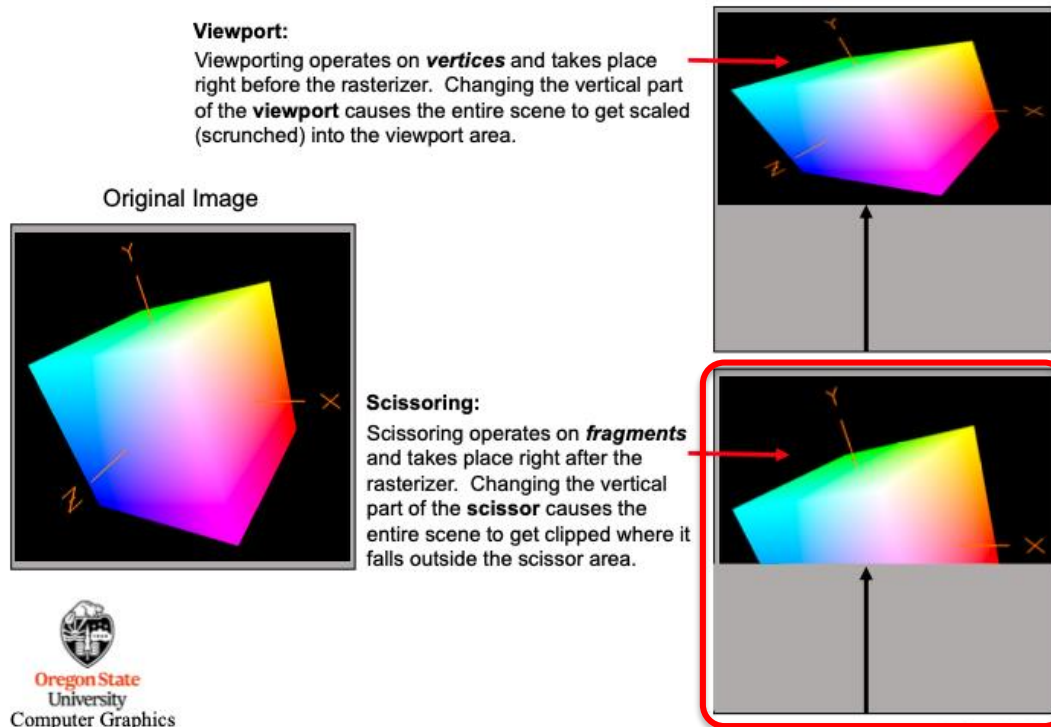
It is defined with a `VkViewport` object, where the `(x, y)` coordinates of the top-left corner and the `width` and `height` of the viewport, all specified in pixel are assigned to the corresponding fields. If the whole area needs to be used, we can read these value from the surface we created.

The range of the normalized z screen coordinates can also be specified in the `minDepth` and `maxDepth` field.

```
VkViewport viewport{};  
viewport.x = 0.0f;  
viewport.y = 0.0f;  
viewport.width = (float) extent.width;  
viewport.height = (float) extent.height;  
viewport.minDepth = 0.0f;  
viewport.maxDepth = 1.0f;
```

# Scissor test

If we need to leave some area of the screen unaffected by Vulkan (for example to use it to draw some GUI inside), we can “cut” rectangular area using *Scissor rectangles*.



# Scissor test

In this case we specify the position in pixel to cut in a `VkRect2D` structure.

If we want to use the full window, we can get these value from the surface as done before with the viewport.

```
VkRect2D scissor{};  
scissor.offset = {0, 0};  
scissor.extent = extent;
```

# Viewport State structure

Both viewport and scissor are combined into a `VkPipelineViewportStateCreateInfo` structure that will be used when creating the pipeline.

```
VkPipelineViewportStateCreateInfo viewportState{};  
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;  
  
viewportState.viewportCount = 1;  
viewportState.pViewports = &viewport;  
viewportState.scissorCount = 1;  
viewportState.pScissors = &scissor;
```

# Back-face culling

The rasterizer component of the pipeline, defined inside a `VkPipelineRasterizationStateCreateInfo` object, controls both back-face culling, and other parameters such as the possibility of drawing wireframe primitives, and the line width for segments (not supported on all hardware).

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

# Back-face culling

Field `polygonMode` controls if we want to draw filled triangles (`VK_POLYGON_MODE_FILL`), or wireframes (`VK_POLYGON_MODE_LINE`).

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

# Back-face culling

Backface culling is controlled by the `cullMode` field: on with `VK_CULL_MODE_BACK_BIT`, and off with `VK_CULL_MODE_NONE`. Accepted vertices order is contained into the `frontFace` field: either `VK_FRONT_FACE_CLOCKWISE` or `VK_FRONT_FACE_COUNTER_CLOCKWISE`.

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType =
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

# Z-buffer and stencil

Z-buffer and stencil are configured in a  
`VkPipelineDepthStencilStateCreateInfo` structure.

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.sType =
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```



# Z-buffer and stencil

In particular, depth testing is enabled setting field `depthTestEnable` and `depthWriteEnable` to `VK_TRUE`.

It is also possible to include or exclude the case in which two components have the same distance with the `depthCompareOp` field (“less than” is obtained with `VK_COMPARE_OP_LESS`).

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.sType =
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```

# Pipeline Layout and Uniforms Variables

The *pipeline layout* defines the uniform variables used by the shaders. A pipeline can also have no uniform variables: however, this section needs to be filled anyway with an empty layout element, and the corresponding `VkPipelineLayout` object created (and destroyed at the end).

```
// Pipeline Layout (Uniform variables) - currently empty
VkPipelineLayout pipelineLayout;

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

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

# Multisampling

Multisampling describes possible anti-aliasing techniques to reduce the artifacts and create smoother images. Also in this case, even if no anti-aliasing is used, the section needs to be filled with the following values specified inside a `VkPipelineMultisamplingStateCreateInfo` structure:

```
// Multisampling (in other things)
VkPipelineMultisampleStateCreateInfo multisampling{};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

multisampling.sampleShadingEnable = VK_FALSE;
multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optional
multisampling.pSampleMask = nullptr; // Optional
multisampling.alphaToCoverageEnable = VK_FALSE; // Optional
multisampling.alphaToOneEnable = VK_FALSE; // Optional
```

# Pipeline creation in Starter.hpp

Pipelines are created with `Pipeline` objects. They are initialized with the `init()` method, which beside the *Descriptor Set Layouts* we already discussed, receives the file names of the .SPV binary codes of the *Vertex* and *Fragment* Shaders.

```
// Pipelines [Shader couples]
Pipeline P1;
P1.init(this, &VD, "shaders/BlinnVert.spv", "shaders/BlinnFrag.spv", {&DSL0, &DSL1});
```

# Pipeline configuration in `Starter.hpp`

Pipelines are created with the most common default values:

- Entire screen (no viewport)
- Full normalized screen coordinates range (no scissor)
- Back-face culling enabled
- Filled triangles
- No transparency
- Use of indexed triangle lists
- ...

# Pipeline configuration in Starter.hpp

Other common settings can be changed, after having created the pipeline, with the `setAdvancedFeatures()` method.

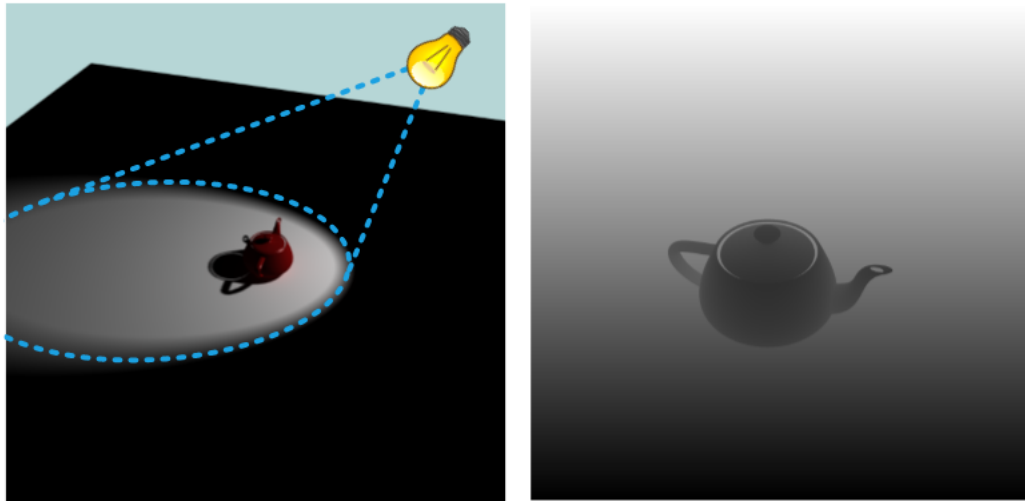
```
void Pipeline::setAdvancedFeatures(VkCompareOp _compareOp,  
                                   VkPolygonMode _polyModel,  
                                   VkCullModeFlagsBits _CM,  
                                   bool _transp) {  
    compareOp = _compareOp;  
    polyModel = _polyModel;  
    CM = _CM;  
    transp = _transp;  
}
```

**true** to enable transparency.

# Render Passes

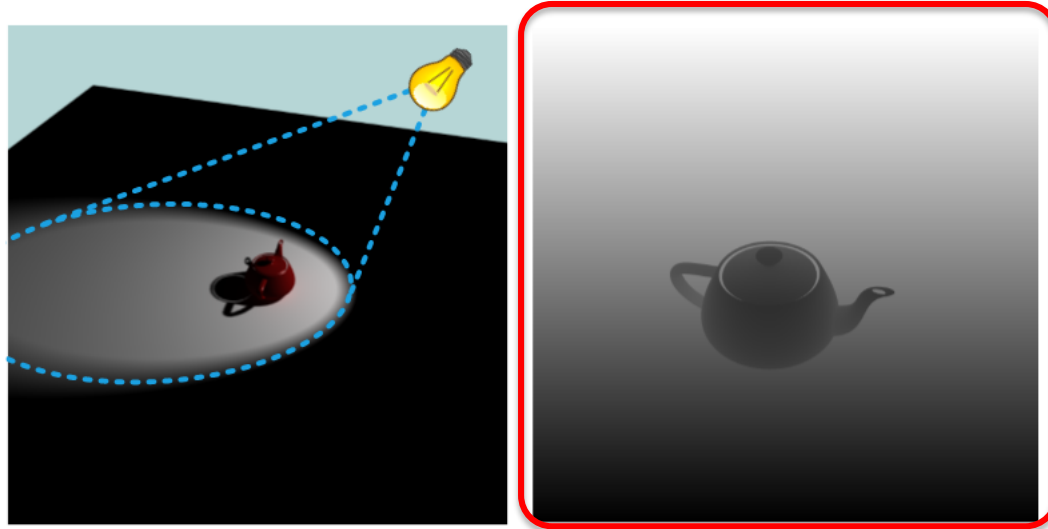
Rendering of current image might require several steps, and produce several images.

For example, to create shadows using the Shadow Map technique briefly introduced...



# Render Passes

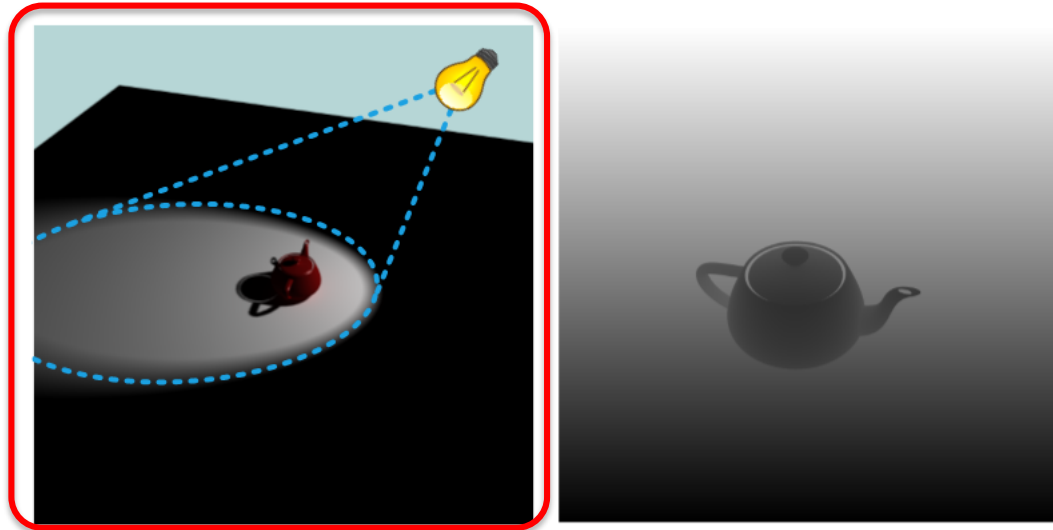
We need first to render the entire scene from the view of the light to produce a map that stores the distance of the points “seen” by the illumination source.





# Render Passes

Then we need to render a second time the scene, this time using the map produced at the previous step to determine whether a point is directly hit by the light and perform the Ray-casting approximation to the rendering equation.



# Render Passes

Render passes are defined inside a `VkRenderPass` object. Their purpose is to contain inside a single data structure, all the information required for performing a rendering producing the same output data on the considered geometry.

```
// Render Pass configuration  
VkRenderPass renderPass;
```

# Color attachments

A render pass can produce a lot of information for every pixel, which can be used for advanced rendering techniques such as *Deferred Rendering*. These are stored in memory buffers corresponding to rectangular areas that are called *Attachments*.

They are defined inside a `VkAttachmentDescription` object.

```
VkAttachmentDescription colorAttachment{};  
colorAttachment.format = surfaceFormat.format;  
colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;  
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;  
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;  
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;  
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;  
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;  
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

# Color attachments

Each attachment can have its own `format`, as described in the corresponding field. For conventional rendering techniques, this corresponds to the one defined for the drawing surface.

```
VkAttachmentDescription colorAttachment{};  
colorAttachment.format = surfaceFormat.format;  
colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;  
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;  
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;  
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;  
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;  
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;  
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

# Color attachments

Other parameters defines the operations that are allowed on the attachment, if stencils are used, and the possible conversions of the color formats done before or after its use.

A complete description of these fields is outside the scope of this course.

```
VkAttachmentDescription colorAttachment{};
    colorAttachment.format = surfaceFormat.format;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

# Depth attachment

To support Z-buffering, the depth buffer must be defined into a specific attachment.

```
VkAttachmentDescription depthAttachment{};
depthAttachment.format = findDepthFormat(physicalDevice);
depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depthAttachment.finalLayout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

# Attachment indices

Attachments are used through their id, which is wrapped inside `VkAttachmentReference` objects.

Their id is specified inside the `attachment` field, and their usage inside the `layout` property.

...

```
VkAttachmentReference colorAttachmentRef{};  
colorAttachmentRef.attachment = 0;  
colorAttachmentRef.layout =  
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

...

```
VkAttachmentReference depthAttachmentRef{};  
depthAttachmentRef.attachment = 1;  
depthAttachmentRef.layout =  
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

# Color Blending

For each color attachments (i.e. not the depth buffer), *color blending* must be defined. This controls whether new colors are combined with the ones previously present in the same position (and in this case how), or if the new value completely replaces the old.

```
VkPipelineColorBlendAttachmentState colorBlendAttachment{};
colorBlendAttachment.colorWriteMask =
    VK_COLOR_COMPONENT_R_BIT |
    VK_COLOR_COMPONENT_G_BIT |
    VK_COLOR_COMPONENT_B_BIT |
    VK_COLOR_COMPONENT_A_BIT;
colorBlendAttachment.blendEnable = VK_FALSE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```



# Color Blending

The information for all attachment is then collected inside a `VkPipelineColorBlendStateCreateInfo` structure, using its `pAttachment` field.

```
VkPipelineColorBlendStateCreateInfo colorBlending{};
colorBlending.sType =
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlending.logicOpEnable = VK_FALSE;
colorBlending.logicOp = VK_LOGIC_OP_COPY; // Optional
colorBlending.attachmentCount = 1;
colorBlending.pAttachments = &colorBlendAttachment;
colorBlending.blendConstants[0] = 0.0f; // Optional
colorBlending.blendConstants[1] = 0.0f; // Optional
colorBlending.blendConstants[2] = 0.0f; // Optional
colorBlending.blendConstants[3] = 0.0f; // Optional
```

# Render Sub-passes

In advanced rendering techniques, frames must be rendered several times, each one possibly using different output buffers (i.e. attachments).

The execution of a complete rendering cycle is called Sub-pass and must be described with a `VkSubpassDescription` structure.

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

# Render Sub-passes

Attachments references are inserted inside the p...Attachment fields:

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

# Render Sub-passes

Synchronization between render sub passes can be defined using the `VkSubpassDependency` objects.

A complete description of this feature is however outside the scope of this course.

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;
```

```
VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

# Render Sub-passes

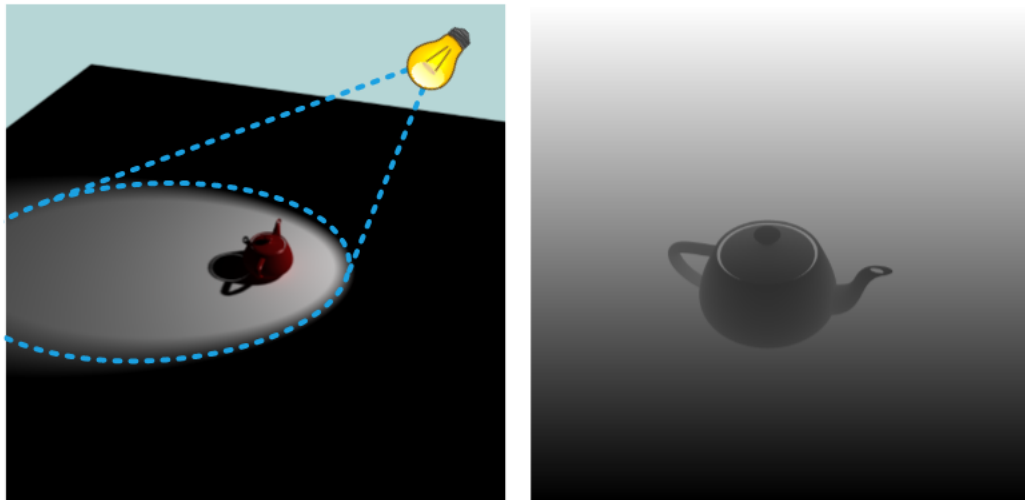
The difference between the passes and the sub-passes is the following: sub-passes can access the data, computed in previous sub-passes, corresponding to only the pixels in the exact same position.

This allows the hardware and the drivers to optimize the computation, but it can be useful only for a limited range of rendering techniques that do not need to sample images in different points.

However, important techniques, such as Deferred Rendering, belong to the class of procedures that can be optimized using sub-passes.

# Render Passes: summary

To summarize, the considered *Shadow Map* example will require two color attachments (the shadow map and the image itself), and two render passes: one for creating the shadow map, and the second one to render the final image.



# Render Pass creation

All the definitions about attachments and sub-passes are collected inside a `VkRenderPassCreateInfo`.

```
std::array<VkAttachmentDescription, 2> attachments =
    {colorAttachment, depthAttachment};

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;

result = vkCreateRenderPass(device, &renderPassInfo, nullptr,
    &renderPass);
if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to create render pass!");
}
```

# Render Pass creation

The corresponding `VkRenderPass` object is finally created with the `vkCreateRenderPass()` function.

```
std::array<VkAttachmentDescription, 2> attachments =
    {colorAttachment, depthAttachment};

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;

result = vkCreateRenderPass(device, &renderPassInfo, nullptr,
                           &renderPass);
if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to create render pass!");
}
```



# Vulkan Commands

We have seen that Vulkan stores all graphics related commands into *Commands Buffers*, that can later be submitted to a queue for execution.

Here we will see the main commands that can be issued in this way.

vkCmd{Set, Reset}Event	vkCmdExecuteCommands
vkCmdBeginQuery	vkCmdFillBuffer
vkCmdBeginRenderPass	vkCmdNextSubpass
vkCmdBindDescriptorSets	vkCmdPipelineBarrier
vkCmdBindIndexBuffer	vkCmdPushConstants
vkCmdBindPipeline	vkCmdResetQueryPool
vkCmdBindVertexBuffers	vkCmdResolveImage
vkCmdBlitImage	vkCmdSetBlendConstants
vkCmdClearAttachments	vkCmdSetDepthBias
vkCmdClearColorImage	vkCmdSetDepthBounds
vkCmdClearDepthStencilImage	vkCmdSetDeviceMask
vkCmdCopyBuffer[ToImage]	vkCmdSetLineWidth
vkCmdCopyImage[ToBuffer]	vkCmdSetScissor
vkCmdCopyQueryPoolResults	vkCmdSetStencil{Compare, Write}Mask
vkCmdDispatch*	vkCmdSetStencilReference
vkCmdDraw[Indirect]	vkCmdSetViewport
vkCmdDrawIndexed[Indirect]	vkCmdUpdateBuffer
vkCmdEndQuery	vkCmdWaitEvents
vkCmdEndRenderPass	vkCmdWriteTimestamp

# Beginning and ending command recording

First command recording must be started with `vkBeginCommandBuffer()`, and it must be stopped at the end with `vkEndCommandBuffer()`.

```
VkCommandBufferBeginInfo beginInfo{};  
beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;  
beginInfo.flags = 0; // Optional  
beginInfo.pInheritanceInfo = nullptr; // Optional
```

```
if (vkBeginCommandBuffer(commandBuffers[i], &beginInfo) != VK_SUCCESS) {  
    throw std::runtime_error("failed to begin recording command buffer!");  
}
```

...

```
if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {  
    throw std::runtime_error("failed to record command buffer!");  
}
```

# Starting a render pass

Not all the commands that can be issued are related with rendering. However, the ones that draws something must be called inside a render pass, that must be started with `vkCmdBeginRenderPass()`.

```
VkRenderPassBeginInfo renderPassInfo{};  
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;  
renderPassInfo.renderPass = renderPass;  
renderPassInfo.framebuffer = swapChainFramebuffers[i];  
renderPassInfo.renderArea.offset = {0, 0};  
renderPassInfo.renderArea.extent = swapChainExtent;
```

Render pass and frame buffer used need to be specified in the command parameters

```
std::array<VkClearColor, 2> clearValues{};  
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};  
clearValues[1].depthStencil = {1.0f, 0};
```

Here, both the screen and the depth buffer are cleared to an initial color value.

```
renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());  
renderPassInfo.pClearValues = clearValues.data();
```

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

# Starting a render pass

The most important information, is the specification of the Frame Buffer, that is of the area where the *out* produced by the Fragment Shader will be finally stored.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;

std::array<VkClearColorValue, 2> clearValues{};
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};
clearValues[1].depthStencil = {1.0f, 0};

renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
renderPassInfo.pClearValues = clearValues.data();

vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

# Starting a render pass

When using `Starter.hpp`, the render pass is handled automatically, and the initial color is specified inside a global variable.

```
// Here you set the window parameters
void setWindowParameters() {
    // window size, titile and initial background
    windowWidth = 800;
    windowHeight = 600;
    windowTitle = "Assignment_SM";
    windowResizable = GLEW_TRUE;
    initialBackgroundColor = {0.0f, 0.6f, 0.8f, 1.0f};

    // The initial aspect rati of the window. In this cc
    Ar = 4.0f / 3.0f;
```

```
VkRenderPassBeginInfo renderPassInfo{}; }
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;

std::array<VkClearColorValue, 2> clearValues{};
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};
clearValues[1].depthStencil = {1.0f, 0};

renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
renderPassInfo.pClearValues = clearValues.data();

vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

# Ending a render pass

Once all commands of one pass have been recorded, it must be concluded with `vkCmdEndRenderPass()`.

...

```
vkCmdEndRenderPass(commandBuffers[i]);
```

# Resources binding

The first step is binding (loading) all resources needed for drawing: pipeline, vertex and index buffer, descriptor sets.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
                  graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);

vkCmdBindDescriptorSets(commandBuffers[i],
                        VK_PIPELINE_BIND_POINT_GRAPHICS,
                        pipelineLayout, 0, 1, &descriptorSets[i],
                        0, nullptr);
```

# Resources binding

Bindings remain valid until changed by a new one. For example, to draw several objects using the same shaders (pipeline), only index and vertex buffer, and descriptor sets need to be updated.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, PBRPipeline);

for(int j = 0; j < Scene.size(); j++) {
    VkBuffer vertexBuffers[] = {Scene[j].MD.vertexBuffer};
    VkDeviceSize offsets[] = {0};
    vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
    vkCmdBindIndexBuffer(commandBuffers[i], Scene[j].MD.indexBuffer, 0, VK_INDEX_TYPE_UINT32);
    vkCmdBindDescriptorSets(commandBuffers[i],
                           VK_PIPELINE_BIND_POINT_GRAPHICS,
                           PBRPipelineLayout, 0, 1,
                           &PBRDescriptorSets[j][i],
                           0, nullptr);
    ...
}
```



# Resources binding

Vertex buffers can be split into several memory area. For this reason they must be passed into an array of `vertexBuffers` and `offset` inside such buffers. This use is however too advanced and outside the scope of this course.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
                  graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);

vkCmdBindDescriptorSets(commandBuffers[i],
                        VK_PIPELINE_BIND_POINT_GRAPHICS,
                        pipelineLayout, 0, 1, &descriptorSets[i],
                        0, nullptr);
```


# Resources binding

To improve performance, the same buffer can be used to include several primitives. For this reason, both the index and the vertex buffer binding commands can specify an offset inside them.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
                  graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);

vkCmdBindDescriptorSets(commandBuffers[i],
                        VK_PIPELINE_BIND_POINT_GRAPHICS,
                        pipelineLayout, 0, 1, &descriptorSets[i],
                        0, nullptr);
```



Index buffer can support  
16 or 32 bits indices.

# Resources binding

Descriptor Sets (which are the instances of Descriptor Sets Layout) requires both the specification of the pipeline type, and of its layout. Several consecutive Sets can be bound with a single command: for this reason they are passed as an array.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,  
                  graphicsPipeline);
```

```
VkBuffer vertexBuffers[] = {vertexBuffer};
```

```
VkDeviceSize offsets[] = {0};
```

```
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);
```

```
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
```

```
vkCmdBindDescriptorSets(commandBuffers[i],  
                        VK_PIPELINE_BIND_POINT_GRAPHICS,  
                        pipelineLayout, 0, 1, &descriptorSets[i],  
                        0, nullptr);
```

First set to be bound

Number of consecutive sets to bind

# Primitive Drawing

If indexed primitives are not used (i.e. simple triangle list or triangle strip), the

`vkCmdDraw()`

command can be used.

In this case, of course, binding the *Index Buffer* is not required.

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

The actual `vkCmdDraw` function is a bit anticlimactic, but it's so simple because of all the information we specified in advance. It has the following parameters, aside from the command buffer:

- **vertexCount:** Even though we don't have a vertex buffer, we technically still have 3 vertices to draw.
- **instanceCount:** Used for instanced rendering, use 1 if you're not doing that.
- **firstVertex:** Used as an offset into the vertex buffer, defines the lowest value of `gl_VertexIndex`.
- **firstInstance:** Used as an offset for instanced rendering, defines the lowest value of `gl_InstanceIndex`.

(from <https://vulkan-tutorial.com>)

# Indexed Primitive Drawing

Indexed primitives are drawn with the `vkCmdDrawIndexed()` command. The command requires the following parameters:

1. Command Buffer handle
2. Number of indices to consider
3. Number of instances to draw
4. Position (in the buffer) of the index from which start drawing
5. Offset added to the indices
6. First instance to be drawn

(can be useful to pack more meshes in the same index buffer)

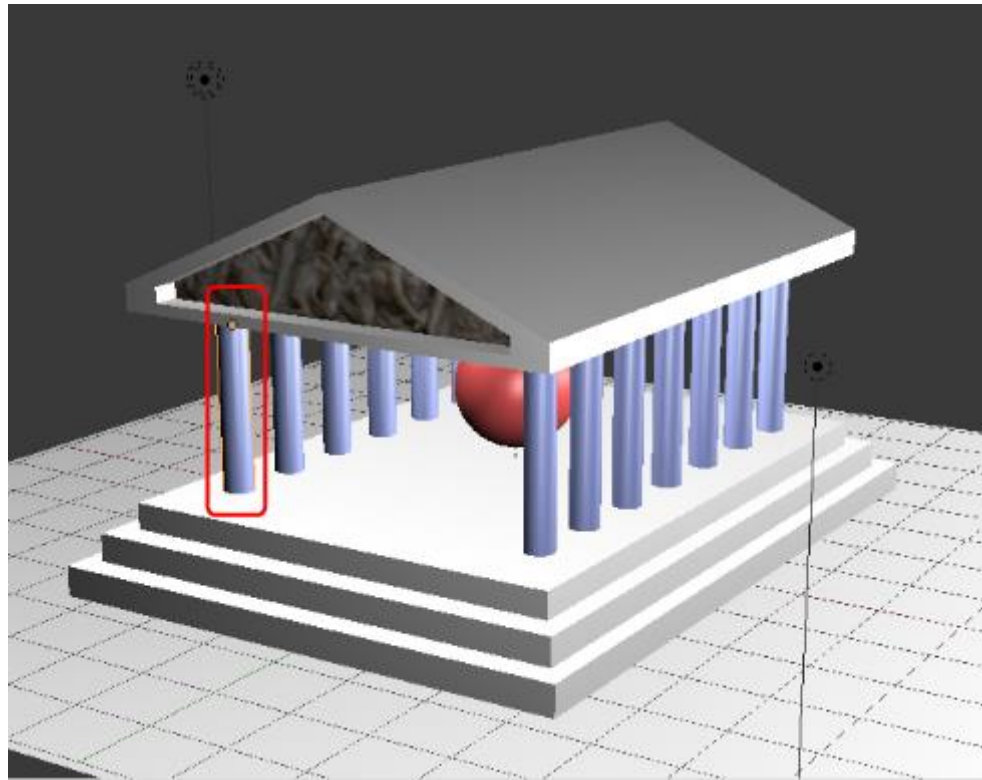
...

```
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

1 2 3 4 5 6

# Instancing

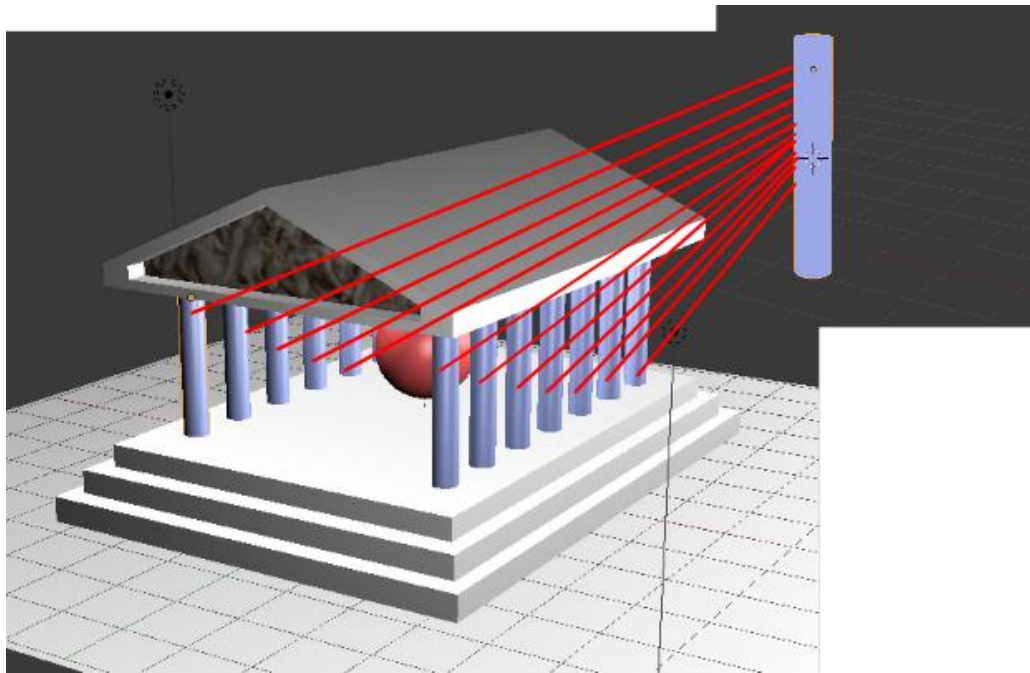
In many scenes, some objects are repeated several times. They all share the same geometry and Shaders, but they are characterized by *different World Matrices*.



# Instancing

In this case, the geometry can be stored only once, and it can be *instanced* (repeated) several times.

Each instance uses the same mesh, but shows the object in a different place with its own world matrix.



# Instancing

Vulkan natively supports instanced drawing:

- All drawing commands require the number of instances to be drawn.
- Each instance can find inside the built-in global variable `gl_InstanceIndex` the ID of the copy currently drawn, which can be used to find its specific parameters, such as its transform matrices.
- Moreover, it is possible to define a set of attributes, similar to the one used per vertices, which depends on the instance. A complete description of the subject is however outside the scopes of the course.

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

```
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```



# Non-rendering commands

Beside performing rendering using a graphic pipeline, Vulkan can use the GPU to perform other related and highly parallelizable tasks.

We will rapidly present the three required to support texture:

```
vkCmdCopyBufferToImage()
```

```
vkCmdPipelineBarrier()
```

```
vkCmdBlitImage()
```

# Copying into an image

Vulkan Images are special types of buffers that contain data which Vulkan can interpret as the pixels of a raster image.

Although Images requires extra information, they can be loaded from a memory area identified by a `VkBuffer` object, using the `vkCmdCopyBufferToImage()` command.

```
vkCmdCopyBufferToImage()
```

# Copying into an image

The command requires a set of parameters describing how the image is encoded into memory, which portion of it should be transferred.

Note that since Vulkan supports 3D textures, all dimensions need to be provided.

```
VkBufferImageCopy region{};
region.bufferOffset = 0;
region.bufferRowLength = 0;
region.bufferImageHeight = 0;
region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
region.imageSubresource.mipLevel = 0;
region.imageSubresource.baseArrayLayer = 0;
region.imageSubresource.layerCount = 1;
region.imageOffset = {0, 0, 0};
region.imageExtent = {width, height, 1};

vkCmdCopyBufferToImage(commandBuffer, buffer, image,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &region);
```

# Transforming the image format

The `vkCmdPipelineBarrier()` function handles synchronization between commands being executed in a pipeline. The barrier can also trigger an image format conversion and wait for it to conclude before continuing.

The latter operating mode is frequently used when handling textures.

```
vkCmdPipelineBarrier()
```

# Transforming the image format


The command requires the initial and final format, as well other specifications for images composed by multiple parts. These settings are however outside the scope of this course.

```
VkImageMemoryBarrier barrier{};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.oldLayout = oldLayout;
barrier.newLayout = newLayout;
barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.image = image;

barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
barrier.subresourceRange.baseMipLevel = 0;
barrier.subresourceRange.levelCount = mipLevels;
barrier.subresourceRange.baseArrayLayer = 0;
barrier.subresourceRange.layerCount = 1;

barrier.srcAccessMask = 0;
barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

vkCmdPipelineBarrier(commandBuffer,
                    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
                    VK_ACCESS_TRANSFER_WRITE_BIT, 0,
                    0, nullptr, 0, nullptr, 1, &barrier);
```



Initial and final image formats

# Image to image partial copy with scaling

The `vkCmdBlitImage()` command copies a portion of an image into another, possibly performing scaling and filtering of the result.

```
vkCmdBlitImage()
```

# Image to image partial copy with scaling

The command requires the description of the input and output region to transfer, as well as the formats and the filter.

```
VkImageBlit blit{};
```

```
blit.srcOffsets[0] = { 0, 0, 0 };  
blit.srcOffsets[1] = { mipWidth, mipHeight, 1 };  
blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;  
blit.srcSubresource.mipLevel = i - 1;  
blit.srcSubresource.baseArrayLayer = 0;  
blit.srcSubresource.layerCount = 1;
```

Source area

```
blit.dstOffsets[0] = { 0, 0, 0 };  
blit.dstOffsets[1] = { mipWidth / 2, mipHeight / 2, 1 };  
blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;  
blit.dstSubresource.mipLevel = i;  
blit.dstSubresource.baseArrayLayer = 0;  
blit.dstSubresource.layerCount = 1;
```

Destination area

```
vkCmdBlitImage(commandBuffer, image,  
               VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,  
               image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,  
               &blit, VK_FILTER_LINEAR);
```

Initial and final image formats

Filter

# Filling the command buffer with Starter.hpp

When using `Starter.hpp`, the command buffer is filled inside the pure virtual method `populateCommandBuffer()`.

The actual Vulkan draw calls are used to issue the drawing of a Mesh.

```
void populateCommandBuffer(VkCommandBuffer commandBuffer, int currentImage) {  
    // The pipeline with the shader for the object  
    Pl.bind(commandBuffer);  
  
    // The models (both index and vertex buffers)  
    Ml.bind(commandBuffer);  
  
    // The descriptor sets, for each descriptor set specified in the pipeline  
    DSG.bind(commandBuffer, Pl, 0, currentImage); // The Global Descriptor  
    DS1.bind(commandBuffer, Pl, 1, currentImage); // The Material and Posit  
  
    vkCmdDrawIndexed(commandBuffer,  
        static cast<uint32 t>(Ml.indices.size()), 1, 0, 0, 0);  
}
```



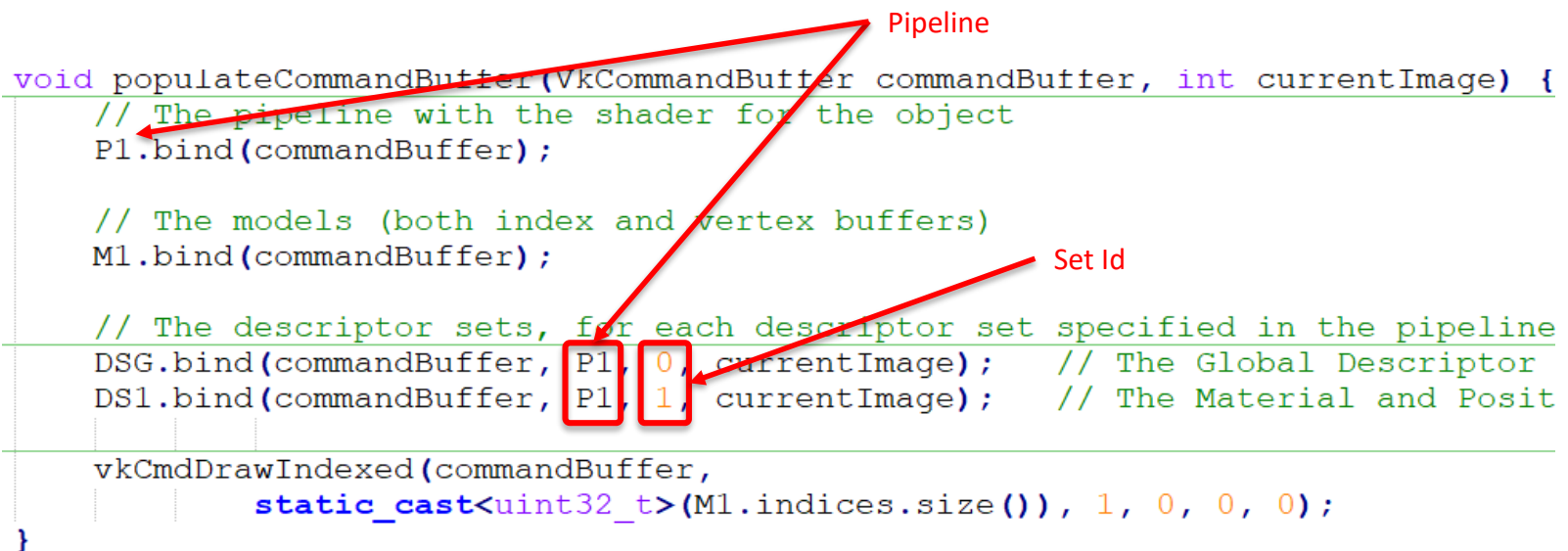
# Filling the command buffer with Starter.hpp

Pipelines, Models and Descriptor Sets are bound to the pipeline with their corresponding `bind()` method.

```
void populateCommandBuffer(VkCommandBuffer commandBuffer, int currentImage) {  
    // The pipeline with the shader for the object  
    Pl.bind(commandBuffer);  
  
    // The models (both index and vertex buffers)  
    Ml.bind(commandBuffer);  
  
    // The descriptor sets, for each descriptor set specified in the pipeline  
    DSG.bind(commandBuffer, Pl, 0, currentImage); // The Global Descriptor  
    DSL.bind(commandBuffer, Pl, 1, currentImage); // The Material and Posit  
  
    vkCmdDrawIndexed(commandBuffer,  
        static_cast<uint32_t>(Ml.indices.size()), 1, 0, 0, 0);  
}
```

# Filling the command buffer with Starter.hpp

Descriptor sets must be bound individually for each Set required by the corresponding Pipeline Layout.



```
void populateCommandBuffer(VkCommandBuffer commandBuffer, int currentImage) {  
    // The pipeline with the shader for the object  
    Pl.bind(commandBuffer);  
  
    // The models (both index and vertex buffers)  
    Ml.bind(commandBuffer);  
  
    // The descriptor sets, for each descriptor set specified in the pipeline  
    DSG.bind(commandBuffer, Pl, 0, currentImage); // The Global Descriptor  
    DS1.bind(commandBuffer, Pl, 1, currentImage); // The Material and Posit  
  
    vkCmdDrawIndexed(commandBuffer,  
        static_cast<uint32_t>(Ml.indices.size()), 1, 0, 0, 0);  
}
```



## Marco Gribaudo

*Associate Professor*

### CONTACTS

Tel. +39 02 2399 3568

[marco.gribaudo@polimi.it](mailto: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)