Descriptor Sets

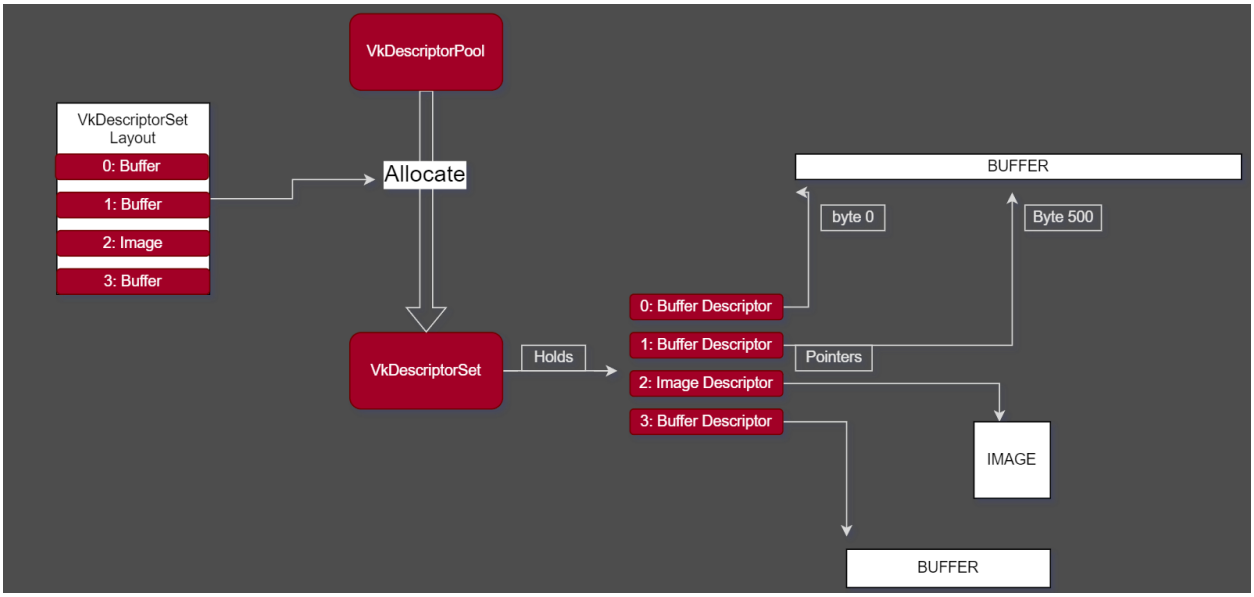
描述符是表示着色器资源的不透明数据结构。它们被组织成组或集合，其内容由描述符集布局指定。为了为着色器提供资源，我们将描述符集绑定到管线。可以一次绑定多个集合。要从着色器中访问资源，我们需要指定从哪个集合以及从集合中的哪个位置（称为绑定）获取给定资源。

1.创建Descriptor Set Layout

2.创建Descriptor Set Pool

3.分配Descriptor Set

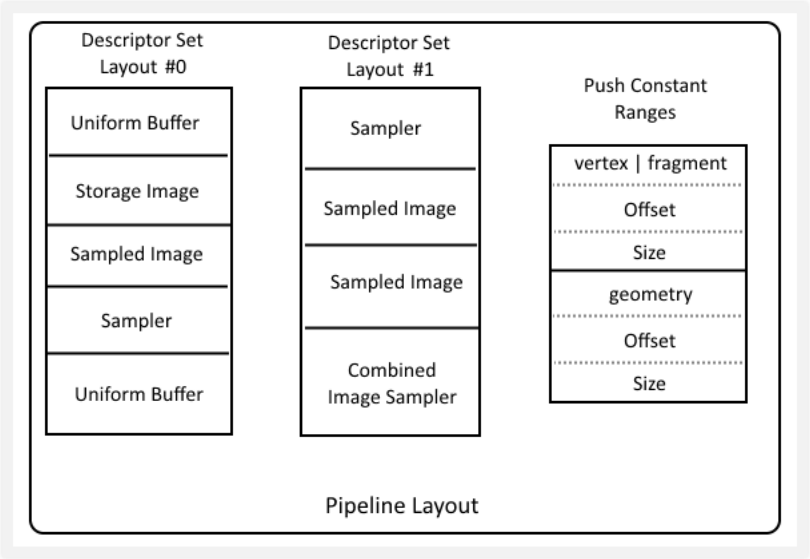
4.关联Uniform Buffer



一个 Descriptor 的种类有很多，常见的例如（这里使用 `VkDescriptorType` 中的枚举名称）：

- `VK_DESCRIPTOR_TYPE_SAMPLER`：采样器，指定图片被读取的方式，之后会提到。
- `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`：Shader 中能被采样的纹理对象。
- `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`：将采样器和能够采样的纹理对象打包起来，成为一个资源描述，之后会提到。
- `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`：Uniform Buffer Object (UBO)，见下章。
- `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`：存储缓冲 Shader Storage Buffer Object (SSBO)，允许我们在 Shader 中读写变量值。

Descriptor 大致上分为三类：**采样器、图像和通用数据缓冲（UBO, SSBO）**，而区别就在于用途和随之而来的性能优化



Shader中的输入数据—location

```
// Vertex attributes
layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inNormal;
layout (location = 2) in vec2 inUV;

// Instanced attributes
layout (location = 4) in vec3 instancePos;
layout (location = 5) in vec3 instanceRot;
layout (location = 6) in float instanceScale;
layout (location = 7) in int instanceTexIndex;
```

这些数据包括如下两部分：

一部分：是顶点的位置信息、顶点的Normal、贴图的UV，这部分信息是在渲染流水线中针对每个顶点的输入数据；

第二部分：就是针对每个实例的数据，在上面的例子中是，某个实例中的位置、旋转、缩放和texture的索引。

对shader 数据进行描述

```
// Per-vertex attributees
// These are advanced for each vertex fetched by the vertex shader
vkb::initializers::vertex_input_attribute_description(0, 0, VK_FORMAT_R32G32B32_SFLOAT, 0),                      // Location 0: Position
vkb::initializers::vertex_input_attribute_description(0, 1, VK_FORMAT_R32G32B32_SFLOAT, sizeof(float) * 3),      // Location 1: Normal
vkb::initializers::vertex_input_attribute_description(0, 2, VK_FORMAT_R32G32_SFLOAT, sizeof(float) * 6),         // Location 2: Texture c
// Per-Instance attributes
// These are fetched for each instance rendered
vkb::initializers::vertex_input_attribute_description(1, 4, VK_FORMAT_R32G32B32_SFLOAT, 0),                      // Location 4: Position
vkb::initializers::vertex_input_attribute_description(1, 5, VK_FORMAT_R32G32B32_SFLOAT, sizeof(float) * 3),      // Location 5: Rotation
vkb::initializers::vertex_input_attribute_description(1, 6, VK_FORMAT_R32_SFLOAT, sizeof(float) * 6),            // Location 6: Scale
vkb::initializers::vertex_input_attribute_description(1, 7, VK_FORMAT_R32_SINT, sizeof(float) * 7),              // Location 7: Texture a
```

最后将上面的信息绑定到要创建的pipeline上面 ：

```
// The instancing pipeline uses a vertex input state with two bindings
binding_descriptions = {
    // Binding point 0: Mesh vertex layout description at per-vertex rate
```

```
    vkb::initializers::vertex_input_binding_description(0, sizeof(Vertex), VK_VERTEX_INPUT_RATE_VERTEX),
    // Binding point 1: Instanced data at per-instance rate
    vkb::initializers::vertex_input_binding_description(1, sizeof(InstanceData), VK_VERTEX_INPUT_RATE_INSTANCE)};
}

attribute_descriptions = {
    // Per-vertex attributees
    // These are advanced for each vertex fetched by the vertex shader
    vkb::initializers::vertex_input_attribute_description(0, 0, VK_FORMAT_R32G32B32_SFLOAT, 0),
      ............
    // Location 6: Scale
    vkb::initializers::vertex_input_attribute_description(1, 7, VK_FORMAT_R32_SINT, sizeof(float) * 7),
};
input_state.pVertexBindingDescriptions   = binding_descriptions.data();
input_state.pVertexAttributeDescriptions = attribute_descriptions.data();

pipeline_create_info.pVertexInputState = &input_state;
```

1.Descriptor-描述符，最终告诉Pipeline，这块内存是个啥东西，是sampler啊，还是一个buffer，如果是buffer，应该告诉Pipeline地址啊，如果是sampler，也得告诉人家你寻址的图片的内存不是；

```
typedef struct VkDescriptorImageInfo {
    VkSampler        sampler;
    VkImageView      imageView;
    VkImageLayout    imageLayout;
} VkDescriptorImageInfo;

typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

2. DescriptorSet-描述符集，这个就比较好理解了，一组shader中，有可能使用到多个数据啊，总不可能只是用一个descriptor都搞定啊，嗯，那就需要一个集合。咱们这个例子中，本来就有两个啊；

3. DescriptorSetLayout-描述符集布局，在一个游戏中，一组shader不一定就使用一次对不对，可能针对不同的物体都使用一组shader来进行绘制，但是呢，里面的数据可能是通过不同的DescriptorSet来绑定的。

这个Layout布局，就相当于一个模具，里面有几个插孔，分别要求你是提供什么尺寸和材质的零件。所以，DescriptorLayout在定义的时候，往往是对一组Shader所定义的“元数据模型”的定义。

```
std::vector<VkDescriptorSetLayoutBinding> set_layout_bindings =
    {
        // Binding 0 : Vertex shader uniform buffer
        vkb::initializers::descriptor_set_layout_binding(
            VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
            VK_SHADER_STAGE_VERTEX_BIT,
            0),
        // Binding 1 : Fragment shader combined sampler
        vkb::initializers::descriptor_set_layout_binding(
            VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,
            VK_SHADER_STAGE_FRAGMENT_BIT,
            1),
    };

VkDescriptorSetLayoutCreateInfo descriptor_layout_create_info =
    vkb::initializers::descriptor_set_layout_create_info(
        set_layout_bindings.data(),
        vkb::to_u32(set_layout_bindings.size()));

V
```

4.PipelineLayout-渲染管线布局信息，这个也好理解了，这个布局中包含了上面的DescripotrSetLayout，以及push constant的访问范围。先看看代码呗：

```
typedef struct VkPipelineLayoutCreateInfo {
    VkPipelineLayoutCreateFlags     flags;                //保留供未来使用
```

```
    uint32_t                        setLayoutCount;         //DescriptorSetLayout的数量
    const VkDescriptorSetLayout*    pSetLayouts;            //提供数据
    uint32_t                        pushConstantRangeCount; //推送常量的范围数量
    const VkPushConstantRange*      pPushConstantRanges;    //推送常量的范围
} VkPipelineLayoutCreateInfo;
```

从上面的代码中，应该看得出来PipelineLayout是对DescriptorSetLayout和push Constant的一个包装。那有人就会问，什么情况下，会使用多个DescriptorSetLayout呢？因为在我们的shader中，我们支持下面的语法形式：

> layout (set=M, binding=N) uniform sampler2D variableNameArray[I];

M表示管线布局中的

```
pSetLayouts
```

N表示M的描述符集合布局中的

```
pBindings
```

I表示N的描述符集合中描述符的索引

```
K_CHECK(vkCreateDescriptorSetLayout(get_device().get_handle(), &descriptor_layout_create_info, nullptr, &descriptor_set_layout));
```

5.最后就是Pipeline了，就是整个渲染管线中的配置。里面不仅仅是针对可编程管线部分的定义和数据绑定，也包含了针对不可编程管线部分的配置信息的设定。

6.真正资源的绑定

其实啊，DescriptorSet真正绑定到流水线的时机是在真正开始构建绘制的command buffer的时候。上面这些工作都是为了在构建的时候，能够很容易地实现绑定。我们接下来，还是针对第一部分的两种情况，分别来看一下，它是如何绑定到渲染管线上的。

1.Shader的输入变量 ：

> // Binding point 0 : Mesh vertex buffer
> vkCmdBindVertexBuffers(draw_cmd_buffers[i], 0, 1, rock_vertex_buffer.get(), offsets);
> // Binding point 1 : Instance data buffer
> vkCmdBindVertexBuffers(draw_cmd_buffers[i], 1, 1, &instance_buffer.buffer, offsets);
> vkCmdBindIndexBuffer(draw_cmd_buffers[i], rock_index_buffer->get_handle(), 0,
> VK_INDEX_TYPE_UINT32);
> // Render instances
> vkCmdDrawIndexed(draw_cmd_buffers[i], models.rock->vertex_indices, INSTANCE_COUNT, 0, 0, 0);
> 绑定Vertex Buffer和index buffer，然后就绘制了。至于GPU如何去理解这个vertex buffer，就在我们Pipeline的VkPipelineVertexInputState这个指针中去对应就可以了

2.uniform变量

```
vkCmdBindDescriptorSets(draw_cmd_buffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline_layout, 0, 1, &descriptor_sets.instanced_rocks, 0,
NULL);
vkCmdBindPipeline(draw_cmd_buffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipelines.instanced 它的作用就是在开启绘制之前，将我们提前准备好的descri
ptorSet绑定到pipelineLayout中的第几个set中。
```