Descriptor Sets

In OpenGL

OpenGL puts all uniform data in the same “set”, but with different binding numbers, so you can get at each one.

Each uniform variable gets updated one-at-a-time.

Wouldn’t it be nice if we could update a collection of related uniform variables all at once, without having to update the uniform variables that are not related to this collection?

```glsl
layout( std140, binding = 0 ) uniform mat4         uModelMatrix;
layout( std140, binding = 1 ) uniform mat4          uViewMatrix;
layout( std140, binding = 2 ) uniform mat4          uProjectionMatrix;
layout( std140, binding = 3 ) uniform mat3          uNormalMatrix;
layout( std140, binding = 4 ) uniform vec4           uLightPos;
layout( std140, binding = 5 ) uniform float           uTime;
layout( std140, binding = 6 ) uniform int           uMode;
layout( binding = 7 ) uniform sampler2D uSampler;
```

std140 has to do with the alignment of the different data types. It is the simplest, and so we use it in class to give everyone the highest probability that their system will be compatible with the alignment.

What are Descriptor Sets?

Descriptor Sets are an intermediate data structure that tells shaders how to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders. There are three advantages in doing things this way:

- Related uniform variables can be updated as a group, gaining efficiency.
- Descriptor Sets are activated when the Command Buffer is filled. Different values for the uniform buffer variables can be toggled by just swapping out the Descriptor Set that points to GPU memory, rather than re-writing the GPU memory.
- Values for the shaders’ uniform buffer variables can be compartmentalized into what quantities change often and what change seldom (scene-level, model-level, draw-level), so that uniform variables need to be re-written no more often than is necessary.

```glsl
for( sporadically )
{
    Bind Descriptor Set #0
    for( the entire scene )
    {
        Bind Descriptor Set #1
        for( each object in the scene )
        {
            Bind Descriptor Set #2
            Do the drawing
        }
    }
}
```

Descriptor Sets

Our example will assume the following shader uniform variables:

```glsl
// non-opaque must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform sporadicBuf
{
    int uMode;
    int uUseLighting;
    int uNumInstances;
} Sporadic;

layout( std140, set = 1, binding = 0 ) uniform sceneBuf
{
    mat4 uProjection;
    mat4 uView;
    mat4 uSceneOrient;
    vec4 uLightPos;
    vec4 uLightColor;
    vec4 uLightKaKdKs;
    float uTime;
} Scene;

layout( std140, set = 2, binding = 0 ) uniform objectBuf
{
    mat4 uModel;
    mat4 uNormal;
    vec4 uColor;
    float uShininess;
} Object;

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```
Step 1: Descriptor Set Pools

You don’t allocate Descriptor Sets on the fly – that is too slow. Instead, you allocate a “pool” of Descriptor Sets during initialization and then pull from that pool later.

### Code Snippet

```cpp
void Init13DescriptorSetPool()
{
    VkResult result;
    VkDescriptorPoolSize vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;
    #ifdef CHOICES
        VK_DESCRIPTOR_TYPE_SAMPLER
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
        VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
        VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
        VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
        VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
        VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
        VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
    #endif
    VkDescriptorPoolCreateInfo vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;
    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = vdps;
    result = vkCreateDescriptorPool( LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
    return result;
}
```

Step 2: Define the Descriptor Set Layouts

I think of Descriptor Set Layouts as a kind of “Rosetta Stone” that allows the Graphics Pipeline data structure to allocate room for the uniform variables and to access them.
Step 2: Define the Descriptor Set Layouts

- TexSamplerSet DS Layout Binding:
  - bindingCount = 1
  - binding = 0
  - descriptorCount = 1
  - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - set = 0

- ObjectSet DS Layout Binding:
  - bindingCount = 1
  - binding = 0
  - descriptorCount = 1
  - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - set = 1

- SceneSet DS Layout Binding:
  - bindingCount = 1
  - binding = 0
  - descriptorCount = 1
  - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - set = 2

- SporadicSet DS Layout Binding:
  - bindingCount = 1
  - binding = 0
  - descriptorCount = 1
  - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - set = 3

Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```cpp
VkResult
Init14GraphicsPipelineLayout() 
{
    VkResult result = VK_OK;
    VkPipelineLayoutCreateInfo vplci = {VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO};
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.setLayoutCount = 4;
    vplci.flags = 0;
    vplci.pNext = nullptr;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
    return result;
}
```
Step 4: Allocating the Memory for Descriptor Sets

```c
vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );
```

Step 5: Tell the Descriptor Sets where their CPU Data is

- **VkDescriptorBufferInfo**
  ```
  vdbi0:
  vdbi0.buffer = MySporadicUniformBuffer.buffer;
  vdbi0.offset = 0;
  vdbi0.range = sizeof(Sporadic);
  ```

- **VkDescriptorImageInfo**
  ```
  vdii0:
  vdii0.sampler = MyPuppyTexture.texSampler;
  vdii0.imageView = MyPuppyTexture.texImageView;
  vdii0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
  ```
Step 5: Tell the Descriptor Sets where their data is

```c
VkWriteDescriptorSet vwds2;
// ds 2:
vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds2.pNext = nullptr;
vwds2.dstSet = DescriptorSets[2];
vwds2.dstBinding = 0;
vwds2.dstArrayElement = 0;
vwds2.descriptorCount = 1;
vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds2.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
vwds2.pImageInfo = nullptr;
vwds2.pTexelBufferView = nullptr;
```

```c
// ds 3:
VkWriteDescriptorSet vwds3;
vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds3.pNext = nullptr;
vwds3.dstSet = DescriptorSets[3];
vwds3.dstBinding = 0;
vwds3.dstArrayElement = 0;
vwds3.descriptorCount = 1;
vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
vwds3.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
vwds3.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds3.pTexelBufferView = nullptr;
```

Remind
```
uint32_t copyCount = 0;
```

```c
// this could have been done with one call and an array of VkWriteDescriptorSets:
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);
```

Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

```c
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif
vgpci.stageCount = 2;                           // number of stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT &GraphicsPipeline );
```

Step 7: Bind Descriptor Sets into the Command Buffer when Drawing

```c
vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
```

Sidebar: The Entire Descriptor Set Journey

- `vkCreateDescriptorPool( )`: Create the pool of Descriptor Sets for future use
- `vkCreateDescriptorSetLayout( )`: Describe a particular Descriptor Set layout and use it in a specific Pipeline layout
- `vkAllocateDescriptorSets( )`: Allocate memory for particular Descriptor Sets
- `vkCreateDescriptorSet( )`: Tell a particular Descriptor Set where its CPU data is
- `vkUpdateDescriptorSets( )`: Re-write CPU data into a particular Descriptor Set
- `vkCmdBindDescriptorSets( )`: Make a particular Descriptor Set "current" for rendering
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

The pieces of the Pipeline Data Structure are fixed in size – with the exception of the Descriptor Sets and the Push Constants. Each of these two can be any size, depending on what you allocate for them. So, the Pipeline Data Structure needs to know how these two are configured before it can set its own total layout.

Think of the DS layout as being a particular-sized hole in the Pipeline Data Structure. Any data you have that matches this hole’s shape and size can be plugged in there.

The Pipeline Data Structure

Fixed Pipeline Elements

Specific Descriptor Set Layout

Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

Any set of data that matches the Descriptor Set Layout can be plugged in there.