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:

1. Related uniform variables can be updated as a group, gaining efficiency.
2. 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.
3. 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.

In OpenGL, 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?

```cpp
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 vec3 uLightPos;
layout( std140, binding = 4 ) uniform vec4 uTime;
layout( std140, binding = 5 ) uniform int uMode;
layout( std140, binding = 6 ) 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 and then pull from that pool later.

### Step 2: Creating a Descriptor Pool

```cpp
vkCreateDescriptorPool();
for( each scene )
{    Bind Descriptor Set #0
    ( for each object )
    {        Bind Descriptor Set #1
        ( for each draw )
        {            Bind Descriptor Set #2
                Do the drawing
            }        }
    }
}
```
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.

Matrix Set DS Layout Binding: Texture Set DS Layout Binding: Misc Set DS Layout Binding: Tex Sampler Set DS Layout Binding:

<table>
<thead>
<tr>
<th>pipeline stage(s)</th>
<th>descriptorCount</th>
<th>descriptorType</th>
<th>binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix Set DS Layout CI: Texture Set DS Layout CI: Misc Set DS Layout CI: Tex Sampler Set DS Layout CI:

<table>
<thead>
<tr>
<th>pipeline stage(s)</th>
<th>descriptorCount</th>
<th>descriptorType</th>
<th>binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: Define the Descriptor Set Layouts

Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout
Step 4: Allocating the Memory for Descriptor Sets

```
void vkAllocateDescriptorSets( LogicalDevice, IN descriptorPool, OUT &DescriptorSets[4] );
```

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

```
// ds 0:
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pBufferInfo = IN
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.descriptorCount = 1;
vwds0.dstArrayElement = 0;
vwds0.dstBinding = 0;
vwds0.dstSet =
vwds0.pNext = nullptr;
vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
```

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

```
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &GraphicsPipeline, OUT &GraphicsPipelines[5] );
```

// end 

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

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

So, the Pipeline Layout contains the structure of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.

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

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.

Sidebar: The Entire Collection of Descriptor Set Paths

VkDescriptorPoolCreateInfo
vkCreateDescriptorPool( )

VkDescriptorSetLayoutCreateInfo
vkCreateDescriptorSetLayout( )

VkDescriptorSetAllocateInfo
vkAllocateDescriptorSets( )

VkWriteDescriptorSet
vkCmdBindDescriptorSets( )

VkBufferBarrier
vkCmdBufferBeginInfo

VkImageMemoryBarrier
vkCmdBufferEndInfo