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?

```
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;
```

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.

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

Our example will assume the following shader uniform variables:

```
// non-opaque must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( std140, set = 2, binding = 0 ) uniform miscBuf
{
    float uTime;
    int uMode;
} Misc;

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```
Descriptor Sets

CPU:
- Uniform data created in a C++ data structure
- CPU:
- Knows the data's size
- Doesn't know where the data starts
- Knows the GPU data structure
- Knows where the data starts
- Doesn't know where each piece of data starts

GPU:
- Uniform data in a "blob"*
- gateway
- Knows the shader data structure
- Doesn't know where each piece of data starts
- Has access to the CPU data structure through

```cpp
struct matBuf
{
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
    glm::mat3 uNormalMatrix;
};

struct lightBuf
{
    glm::vec4 uLightPos;
};

struct miscBuf
{
    float uTime;
    int uMode;
};
```

```
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
}; Matrices;

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
}; Light;

layout( std140, set = 2, binding = 0 ) uniform miscBuf
{
    float uTime;
    int uMode;
}; Misc;

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 and then pull from that pool later.

```cpp
VkResult 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[0];
    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.

Matrix Set DS Layout
- Binding: 0
- descriptorCount: 4
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER

Light Set DS Layout
- Binding: 1
- descriptorCount: 1
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER

Misc Set DS Layout
- Binding: 2
- descriptorCount: 1
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER

Texture Sampler Set DS Layout
- Binding: 3
- descriptorCount: 1
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
Step 2: Define the Descriptor Set Layouts

Matrix Set DS Layout Binding: Light Set DS Layout Binding: Misc Set DS Layout Binding:

// DS #0:
MatrixSet[0].binding = 0; MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; MatrixSet[0].descriptorCount = 1; MatrixSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

// DS #1:
LightSet[0].binding = 0; LightSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; LightSet[0].descriptorCount = 1; LightSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;

// DS #2:
MiscSet[0].binding = 0; MiscSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER; MiscSet[0].descriptorCount = 1; MiscSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;

// DS #3:
TexSamplerSet[0].binding = 0; TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER; TexSamplerSet[0].descriptorCount = 1; TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

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

VkResult Init14GraphicsPipelineLayout() {
    VkResult result;
    VkDescriptorSetLayoutCreateInfo vdslc0 = { .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, .pNext = nullptr, .flags = 0, .bindingCount = 1, .pBindings = &MatrixSet[0] };
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc0, PALLOCATOR, OUT &DescriptorSetLayouts[0]);
    return result;
}

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

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

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

```cpp
vkWriteDescriptorSet
```

This struct identifies what buffer it owns and how big it is

```cpp
// ds 0:vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds0.pNext = nullptr;
vwds0.dstSet = DescriptorSets[0];
vwds0.dstBinding = 0;
vwds0.dstArrayElement = 0;vwds0.descriptorCount = 1;
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.pBufferInfo = IN &vdbi0;
```

This struct links a Descriptor Set to the buffer it is pointing to

```cpp
// ds 1:
```

This struct identifies what texture sampler and image view it owns

```cpp
// ds 1:
```

This struct identifies what buffer it owns and how big it is

```cpp
// ds 2:
```

This struct identifies what buffer it owns and how big it is

```cpp
// ds 3:
```

This struct identifies what texture sampler and image view it owns
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 = &vdbi2;
vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds2.pTexelBufferView = (VkBufferView *)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;
#endif
vgpci.stageCount = 2;                           // number of stages in this pipeline
vgpci.pStages = vpsci;
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 = GraphicsPipelineLayout;         // this could have been done with one call and an array of VkWriteDescriptorSets:
vgpci.renderPass = &vpri;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, &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 );
```

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.