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 bunch of related uniform variables all at once?

```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;
```

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 mat4Buf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 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

We're using Descriptors Sets to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders. Here are the 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.
- 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
        }
    }
```

For each object

```
    Bind Descriptor Set #3
    Do the drawing
```

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.
- 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.
**Uniform data created in a C++ data structure**

**CPU:**
- Knows the CPU data structure
- Knows where the data starts
- Knows the data size
- Doesn't know the GPU data structure

**GPU:**
- Knows the shader data structure
- Doesn't know where each piece of data starts
- Knows where the data starts
- Knows the data's size
- Doesn't know the CPU or GPU data structure

**GPU:**
- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size

* "binary large object"

```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] = {
        { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1 },
        { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1 },
        { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1 },
        { VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1 },
    };
    VkDescriptorPoolCreateInfo vdpci = {
        .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO,
        .pNext = nullptr,
        .flags = 0,
        .maxSets = 4,
        .poolSizeCount = 4,
        .pPoolSizes = vdps,
    };
    result = vkCreateDescriptorPool( LogicalDevice, &vdpci, PALLOCATOR, &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.

```
 DescriptorSetLayout MatrixSetLayout Binding:
 descriptorType = array( descriptorCount )
 pipeline stage(s) = vertex

 descriptorType = array( descriptorCount )
 pipeline stage(s) = fragment

 DescriptorSetLayout LightSetLayout Binding:
 descriptorType = array( descriptorCount )
 pipeline stage(s) = vertex

 DescriptorSetLayout MiscSetLayout Binding:
 descriptorType = array( descriptorCount )
 pipeline stage(s) = fragment

 DescriptorSetLayout TexSamplerSetLayout Binding:
 descriptorType = array( descriptorCount )
 pipeline stage(s) = fragment
```
Step 1: Define the Descriptor Set Layouts

VkResult

vkCreateDescriptorSetLayouts()

{ // DS #0:
  VkResult result = 0;

  VkDescriptorSetLayoutCreateInfo MatrixSet;
  MatrixSet.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
  MatrixSet.pNext = nullptr;
  MatrixSet.flags = 0;
  MatrixSet.bindingCount = 1;
  MatrixSet.pBindings = &MatrixSet[0];

  result = vkCreateDescriptorSetLayout(LogicalDevice, &MatrixSet, PALLOCATOR, OUT DescriptorSetLayouts[0]);
  return result;
}

// DS #1:

// DS #2:

// DS #3:

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

VkResult

vkCreatePipelineLayout()

{ VkResult result = 0;

  VkPipelineLayoutCreateInfo vplci;
  vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  vplci.pNext = nullptr;vplci.flags = 0;
  vplci.setLayoutCount = 4;
  vplci.pSetLayouts = &DescriptorSetLayouts[0];

  result = vkCreatePipelineLayout(LogicalDevice, &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
  return result;
}
Step 4: Allocating the Memory for Descriptor Sets

```cpp
VkResult Init13DescriptorSets()
{
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;

    result = vkAllocateDescriptorSets(LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
}
```

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

```cpp
VkWriteDescriptorSet vwds0;
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 = &vdbi0;
vwds0.pImageInfo = nullptr;
vwds0.pTexelBufferView = nullptr;
```

This struct identifies what buffer it owns and how big it is.
Step 5: Tell the Descriptor Sets where their data is

```c
// ds 2:
VkWriteDescriptorSet vwds2;
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;

// 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.pImageInfo = &vdii0;
vwds3.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;

VkPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.stageCount = 2; // number of stages in this pipeline
vgpci.pVertexInputState = &vpvisci; vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = nullptr; vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci; vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpsisci; vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = GraphicsPipelineLayout;
vgpci.renderPass = RenderPass; vgpci.subpass = 0; // subpass number
vgpci.basePipelineHandle = VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vgpci, PALLOCATOR, &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 );
```