In Depth Overview of Descriptors and How to Organize Them

Trevor Hammock

Descriptors – Recap

• In OpenGL all of your uniform data is stored at different binding points

```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(std140, binding = 7) uniform sampler2D uSampler;
```

• In OpenGL each Shader program gets its own set of GPU memory for uniforms which can easily be updated with glUniform()

• This approach has some design and performance implications
  • First we cannot reuse similar uniform data, such as MVP matrices, between multiple Shader programs. Since their memory is independent we need to recall glUniform()
  • Second glUniform() updates our Shader's device memory with our CPU's host memory. Transferring data between the CPU and the GPU should be minimal
  • Third if we want to swap out or change our uniform data we must re-write the GPU memory. Even if this data is constant

Descriptors – Example 1

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

Descriptors – Example 2

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

Descriptors – Recap

• To solve most of these issues Vulkan introduced Descriptors
  • A Descriptor is basically a pointer to the GPU memory where your uniform data is stored. Vulkan uses these to map your GPU memory to your Shaders
  • With Descriptors we can re-use uniform data by having different Descriptors point to the same block of memory
  • With Descriptors we can minimize transferring uniform data between the CPU and the GPU
  • With Descriptors we can avoid re-writing GPU memory by simply changing the Descriptor instead

Descriptors – Recap
Overview

• This presentation has two goals
  • The first is to go into more detail about Descriptors and detail some of the ways that they can be set up
  • The second is to give advice about how to structure and organize them

How Vulkan Descriptors are handled

• **DescriptorSet**: a set or collection of individual descriptors
• **DescriptorSetLayout**: an array of zero or more descriptor bindings
• **DescriptorPool**: maintains a pool of descriptors from which **DescriptorSets** are allocated
• **PipelineLayout**: an array of zero or more **DescriptorSetLayouts** which describe the complete set of resources that can be accessed by an individual pipeline. Is used to make your pipelines
• Command buffers then link the uniform data to your Shaders by binding the **DescriptorSets** to the given pipeline

How Vulkan Descriptors are handled – ELI5

• **Descriptor**: a pointer to your uniform data
• **DescriptorSet**: holds pointers to your uniform data
• **DescriptorSetLayout**: describes how the uniform data is laid out. This is done through the binding specifications
• **DescriptorPool**: makes and manages your **DescriptorSet** memory
• **PipelineLayout**: tells your pipeline how the uniform data is laid out. Created using your **DescriptorSetLayouts**
• Use `vkCmdBindDescriptorSets()` to bind **DescriptorSets** to the pipeline. This links your uniform data.

Descriptor Model

Set vs Binding in the Layout Qualifier

• **Set** is the **DescriptorSet** the uniform is associated with
• **Binding** is a binding within that **DescriptorSet**

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

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

layout(std140, set = 1, binding = 0) uniform miscBuf
{ 
    float uTime;
    int uMode;
}
Misc;

layout(set = 1, binding = 1) uniform sampler2D uSampler;
```
Descriptor Layout

- Determines how the pointers (Descriptors) in a DescriptorSet are laid out
- Accomplishes this using an array of binding specifications
- The binding specifications are then used to create the DescriptorSetLayout

Descriptor Layout – Example

```glsl
/** GLSL code **/
layout(std140, set = 0, binding = 0) uniform matBuf
{
  mat4 uMVP_Matrix;
}
Matrices;

/** Vulkan code **/
VkDescritorSetLayoutBinding bindings[2];
//uniform matBuf
bindings[0].binding = 0;
bindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
bindings[0].descriptorCount = 1;
bindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
bindings[0].pImmutableSamplers = (VkSampler*)nullptr;
//uniform uSamplers
bindings[1].binding = 1;
bindings[1].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
bindings[1].descriptorCount = 3;
bindings[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
bindings[1].pImmutableSamplers = (VkSampler*)nullptr;

VkDescriptorSetLayoutCreateInfo descriptorLayout{};
descriptorLayout.type = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
descriptorLayout.bindingCount = sizeof(bindings) / sizeof(bindings[0]);
descriptorLayout.pBindings = bindings;
//Create the descriptor layout
VkResult result = vkCreateDescriptorSetLayout(device, &descriptorLayout, pAllocator, &DescriptorSetLayout1);
```

Descriptor Layout – Example Continued

```vulkan
/** Vulkan code **/
//uniform matBuf
vkGetUniformBufferOffsetBaseLayout(bindings[1]);
//uniform matBuf
bindings[0].binding = 0;
bindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
bindings[0].descriptorCount = 1;
bindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
bindings[0].pImmutableSamplers = (VkSampler*)nullptr;
//uniform uSamplers
bindings[1].binding = 1;
bindings[1].descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER;
bindings[1].descriptorCount = 3;
bindings[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
bindings[1].pImmutableSamplers = (VkSampler*)nullptr;

VkDescriptorSetLayoutCreateInfo descriptorLayout{};
descriptorLayout.type = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
descriptorLayout.bindingCount = sizeof(bindings) / sizeof(bindings[0]);
descriptorLayout.pBindings = bindings;
//Create the descriptor layout
VkResult result = vkCreateDescriptorSetLayout(device, &descriptorLayout, pAllocator, &DescriptorSetLayout1);
```

Descriptor Layout Tips

- VkDescriptorsetLayoutBinding
  - descriptorCount > 1 can be used to define uniform arrays
  - Setting descriptorCount to 0 can reserve that binding location
  - pImmutableSamplers can attach samplers to the DescriptorSetLayout (rather than a DescriptorSet). When the DescriptorSetLayout gets bound to the pipeline this will make them immutable
- VkDescriptorsetLayoutCreateInfo
  - Internally must create bindings between [0, maxBindNumber) from pBindings
  - To avoid wasted DescriptorSetLayout memory use a smaller binding number
  - Implicitly sets binding numbers that you did not specify but in range
  - Binding numbers that are not specified have descriptorCount and stageFlags set to 0

Descriptor Set

- Is a collection or set of Descriptors
- These Descriptors are pointers to where your uniform data is located
- Must adhere to the binding specifications of a DescriptorSetLayout
- Unlike a DescriptorSetLayout, its memory is managed by a DescriptorPool
```glsl
layout(std140, set=0, binding=0) uniform matBuf{mat4 uMVP_Matrix;}

layout(set=0, binding=1) uniform sampler uSamplers[3];
```

```vulkan
//Allocate memory for the descriptor set
VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = DescriptorPool;
allocInfo.descriptorSetCount = 1;
allocInfo.pSetLayouts = &DescriptorSetLayout1;

VkResult result = vkAllocateDescriptorSets(device, &allocInfo, &DescriptorSet1);
```

```vulkan
//Create the descriptors (pointers) for our Descriptor Set
VkDescriptorBufferInfo MatricesDescriptor{};
matBufDescriptor.buffer = MatricesUniformBuffer.buffer; //buffer handle
matBufDescriptor.offset = 0; //where to start within that buffer
matBufDescriptor.range = sizeof(Matrices); //how many bytes to grab at that offset

VkDescriptorImageInfo uSamplersDescriptors[3];
for(int i = 0; i < 3; i++)
{
    uSamplersDescriptors[i].sampler = Samplers[i];
    uSamplersDescriptors[i].imageView = VK_NULL_HANDLE;
    uSamplersDescriptors[i].imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
}
```

```vulkan
//Update the Descriptor Set with our descriptors (setting the pointers)
VkWriteDescriptorSet writeDescriptorSets[2][] = {
    {
        .sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET,
        .pNext = nullptr,
        .dsSet = DescriptorSet1,
        .dstBinding = 0,
        .dstArrayElement = 0,
        .descriptorCount = 1,
        .descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
        .pBufferInfo = &MatricesDescriptor,
        .pImageInfo = nullptr,
        .pTexelBufferView = nullptr,
    },
    {
        .sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET,
        .pNext = nullptr,
        .dsSet = DescriptorSet1,
        .dstBinding = 1,
        .dstArrayElement = 0,
        .descriptorCount = sizeof(uSamplersDescriptors)/sizeof(uSamplersDescriptors[0]),
        .descriptorType = VK_DESCRIPTOR_TYPE_SAMPLER,
        .pBufferInfo = nullptr,
        .pImageInfo = uSamplersDescriptors,
        .pTexelBufferView = nullptr,
    }
};

//not using any VkCopyDescriptorSets, set those parameters to 0 and nullptr
vkUpdateDescriptorSets(device, 2, &writeDescriptorSets, 0, nullptr);
```

### Descriptor Set Tips
- **VkDescriptorAllocateInfo**
  - Can make multiple DescriptorSets using the DescriptorSetLayouts within pSetLayouts:
  - Cannot easily make multiple Descriptor Sets using the same DescriptorSetLayout
- **VkWriteDescriptorSet**
  - Can use descriptorCount and dstArrayElement to grab N Descriptors from the Descriptors pointer starting at a given index
- **vkUpdateDescriptorSets**
  - Can also take VkCopyDescriptorSets which can copy an already updated DescriptorSet into another
Descriptor Pool

- Creates and manages your DescriptorSet memory
- It needs to be created before you make any of your DescriptorSets
- Internally it allocates memory for the DescriptorSets and the Descriptors that can be stored within them

Descriptor Pool – Example

```vulkan
/** *
 * Vulkan code
 */

// lets make a pool for the DescriptorSet that we created in the last example
// specify how many Descriptors will be used in this pool
// This makes the memory for the Descriptors in our DescriptorSets

VkDescriptorPoolSize poolSizes[2];
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = 1;
poolSizes[1].type = VK_DESCRIPTOR_TYPE_SAMPLER;
poolSizes[1].descriptorCount = 3;

// Create the DescriptorPool
VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.flags = 0;
poolInfo.maxSets = 1;
poolInfo.poolSizeCount = sizeof(poolSizes) / sizeof(poolSizes[0]);
poolInfo.pPoolSize = poolSizes;
VkResult result;
result = vkCreateDescriptorPool(device, &poolInfo, pAllocator, &DescriptorPool);
```

Descriptor Pool – Example Continued

```vulkan
/** *
 * Vulkan code
 */

result = vkResetDescriptorPool(device, &DescriptorPool);
```

Descriptor Pool Tips

- VkDescriptorCreateInfo
  - maxSets defines the maximum allowed DescriptorSets that this pool is allowed to make
- VkDescriptorPoolSize
  - This defines all of the Descriptors that can be used to make your DescriptorSets
  - This can also be described as the summation of the maximum allowed DescriptorSets times the DescriptorSets for its given layout
  - You somehow need to know this before you have defined the maximum number of DescriptorSets, have created all of your DescriptorSets, and defined the layout for all of them. Why?
  - If you don't specify enough Descriptors of a given type then you will get allocation failures. If there is a bug in your Descriptor code it is probably going to be here and should probably be checked first
- VkResetDescriptorPool
  - This is a function that you can call which can return all DescriptorSets and their Descriptors back to the pool

Multiple Descriptor Pools

- For the most part you can get away with one DescriptorPool
- There are 2 situations where you might want to do this
  - Organizing your Descriptors
  - When using dynamic DescriptorSets
**Multiple Descriptor Pools – Dynamic DescriptorSets**

- In VkDescriptorCreateInfo you can set the flag variable to VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT
- This will allow you to free individual Descriptors back to the pool using vkFreeDescriptorSets()
- You can use this to dynamically create and free DescriptorSets
- Dynamically creating/freeing DescriptorSets can fragment the memory in your DescriptorPool
- This can degrade your Descriptor performance and eventually the pool can become so fragmented it can no longer allocate memory
- By doing dynamic DescriptorSets in their own pool you can isolate them so they will not degrade the rest of your Descriptors

---

**How are the sets in your GLSL code determined?**

- This is determined by your PipelineLayout
- When you create this you give it an array of DescriptorSetLayouts
- The order of the array determines the layouts for a given set (also know as the DescriptorSet binding position)
- As long as a DescriptorSet adheres to the given layout at that set binding position, it can be bound there with vkCmdBindDescriptorSets()

---

**How Should You Design Your Descriptors?**

- Descriptors give you control over your uniform data that OpenGL could never provide
- However you have to use and deal with Descriptors even if your not necessarily taking advantage of them
- Each project has its own unique set of constraints but there is a general set of guidelines one could follow

---

**Vulkan Descriptor Code Structure**

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*The order the code should be defined in*

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>

*The order it currently is*

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descriptor Layout</td>
<td>Descriptor Set 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>

---

**Descriptor Model: Vulkan Style**

- Create as many set binding points as possible to minimize the amount of data that you have to bind

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Descriptor Pool</th>
<th>Descriptor Layout</th>
<th>Descriptor Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uniform Data</td>
</tr>
</tbody>
</table>
Descriptor Model: OpenGL Style

- Use fewer set binding points to make items simpler to organize

Hardware Limitations

- maxBoundDescriptorSets: the maximum number of DescriptorSets that can be used by a pipeline at a given time
- maxPerStageDescriptorUniformBuffers: the maximum number of uniform buffers you can have in a given stage
- maxDescriptorSetUniformBuffer: the maximum number of uniform buffers you can have in your entire pipeline
- maxUniformBufferRange: the maximum number of bytes your uniform buffer can point to

Hardware Limitations – Standard Minimums

- maxBoundDescriptorSets: 4
- maxPerStageDescriptorUniformBuffers: 12
- maxDescriptorSetUniformBuffers: 72
- maxUniformBufferRange: 16384 or 16KiB

- 6 stages * 12 buffers per stage = 72
- Arrays internally use as many as their size. Even though they only consume one binding point an array of 5 will consume 5 of your max allowed buffers
- There is no real good reason to make an array of uniform buffers

Hardware Limitations – maxBoundDescriptorSets

- ARM/Qualcomm: 4
- Intel/Nvidia: 4 for older drivers
- Intel/Nvidia: 8 with newer drivers
- AMD: 32

Source:
https://vulkan.gpuinfo.org/listreports.php?limit=maxBoundDescriptorSets

Hardware Limitations Overview

Portable Vulkan: 4 DescriptorSets
Mobile: 4 DescriptorSets
Older Graphics cards: 4 DescriptorSets
Newer Graphics Cards: 8 DescriptorSets

- Realistically at this point in time you will most likely have to design your Descriptor code with 4 DescriptorSets in mind

How DescriptorSets Could Be Organized

Least changing -> most changing
1. View resources (camera/environment)
2. Shader resources
3. Material resources (material properties/textures)
4. Object resources (object resources)
How Uniform Data Could be Organized

Minimal performance impact -> significant performance impact
1. Push Constants
2. Constantly Bound DescriptorSets
3. Dynamic Uniform Buffers
4. Non Constantly Bound DescriptorSets

Multiple binding points

• If you have multiple binding points within a DescriptorSet you need to be mindful of the data
• If one of the points needs to change then you have to rebind the entire DescriptorSet (and all of the data within it)
• Try to put data that changes at the same frequency within the same DescriptorSet. They have to be rebound anyway and it minimizes the number of DescriptorSet binding calls

Portable GLSL Code

• The set parameter in the layout qualifier is actually optional
• If it is not specified then it defaults to 0
• Since OpenGL does not use Descriptors this would make your GLSL code OpenGL compatible
• But you lose the control and performance optimizations that Descriptors allow for

Portable Layout – Example

```
layout(std140, binding = 0) uniform matBuf {
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat3 uNormalMatrix;
}
Matrices;
layout(std140, binding = 1) uniform lightBuf {
  vec4 uLightPos;
}
Light;
layout(std140, binding = 2) uniform miscBuf {
  float uTime;
  int uMode;
}
Misc;
layout(binding = 3) uniform sampler2D uSampler;
```

Reusing DescriptorSets

• When you bind a new pipeline to a command buffer, you do not have to rebind your DescriptorSets
• The previously bound DescriptorSets are still attached
• If you are consistent with your set binding points in your GLSL code you can minimize how much data could be bound