An Introduction to the Vulkan Computer Graphics API

Mike Bailey
Oregon State University
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

Acknowledgements

First of all, thanks to the inaugural class of 19 students who braved new, unrefined, and just-in-time course materials to take the first Vulkan class at Oregon State University – Winter Quarter, 2018. Thanks for your courage and patience!

Second, thanks to NVIDIA! The GeForce 1080ti cards are what made this course possible.

Third, thanks to Kathleen Mattson and the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding onto them.)

What Prompted the Move to Vulkan?

1. Performance
2. Performance
3. Performance

Vulkan is better at keeping the GPU busy than OpenGL is. OpenGL drivers need to do a lot of CPU work before handing work off to the GPU. Vulkan lets you get more power from the GPU card you already have.

This is especially important if you can hide the complexity of Vulkan from your customer base and just let them see the improved performance. Thus, Vulkan has had a lot of support and interest from game engine developers, 3rd party software vendors, etc.

As an aside, the Vulkan development effort was originally called glNext, which created the false impression that this was a replacement for OpenGL. It’s not.
Why is it so important to keep the GPU Busy?

The Khronos Group, Inc. is a non-profit member-funded industry consortium, focused on the creation of open standard, royalty-free application programming interfaces (APIs) for authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. Khronos members may contribute to the development of Khronos API specifications, vote at various stages before public deployment, and accelerate delivery of their platforms and applications through early access to specification drafts and conformance tests.

Playing “Where’s Waldo” with Khronos Membership
Who's Been Specifically Working on Vulkan?

Vulkan

- Somewhat derived from AMD's Mantle API
- Also heavily influenced by Apple's Metal API and Microsoft's DirectX 12
- Goal: much less driver complexity and overhead than OpenGL has
- Goal: much less user hand-holding — Vulkan can crash
- Goal: higher single-threaded performance than OpenGL can deliver
- Goal: able to do multithreaded graphics
- Goal: able to handle tiled rendering

Vulkan Differences from OpenGL

- More low-level information must be provided (by you!) in the application, rather than the driver
- Screen coordinate system is Y-down
- No “current state”, at least not one maintained by the driver
- All of the things that we have talked about being deprecated in OpenGL are really deprecated in Vulkan: built-in pipeline transformations, begin-end, fixed-function, etc.
- You must manage your own transformations.
- All transformation, color, texture functionality must be done in shaders.
- Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the pipeline-building process.

Moving part of the driver into the application

Complex drivers lead to driver overhead and vendor unpredictability
Error management is always active
Driver processes full shading language source
Separate APIs for desktop and mobile markets

Simpler drivers for low-overhead efficiency and cross vendor portability
Layered architecture so validation and debug layers can be unloaded when not needed
Run-time only has to ingest SPIR-V intermediate language
Unified API for mobile, desktop, console and embedded platforms
Your Sample2017.zip File Contains This

- **Drawing.pptx**

Drawing

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

**Vulkan Topologies**

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST**
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST**
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**

```c
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;
```
**A Colored Cube Example**

static GLuint CubeTriangleIndices[] = {
    { 0, 2, 3 },
    { 0, 3, 1 },
    { 4, 5, 7 },
    { 4, 7, 6 },
    { 1, 3, 7 },
    { 1, 7, 5 },
    { 0, 4, 6 },
    { 0, 6, 2 },
    { 2, 6, 7 },
    { 2, 7, 3 },
    { 0, 1, 5 },
    { 0, 5, 4 }
};

Non-indexed Buffer Drawing

From the file `SampleVertexData.cpp`:

Vertex 7
Vertex 5
Vertex 4
Vertex 1
Vertex 3
Vertex 0
Vertex 3
Vertex 2
Vertex 0

Triangles

**Triangles Represented as an Array of Structures**

From the file `SampleVertexData.cpp`:

struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};

struct vertex VertexData[] = {
    // triangle 0-2-3:
    // vertex #0:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. },
    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. },
    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
};

Filling the Vertex Buffer

```cpp
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
```

`VkResult
Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result;
    result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}`
What Init05DataBuffer Does

```c
VkResult Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer)
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size;  // size
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer(LogicalDevice, &vbci, PALLOCATOR, &pMyBuffer->buffer);
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements(LogicalDevice, pMyBuffer->buffer, &vmr);  // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, &vmai, PALLOCATOR, &vdm);
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory(LogicalDevice, pMyBuffer->buffer, vdm, 0);  // 0 is the offset
    return result;
}
```

Telling the Pipeline about its Input

We will come to the Pipeline later, but for now, know that a Vulkan pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its input.

### C/C++:

```c
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

```c
layout( location = 0 ) in vec3 aVertex;layout( location = 1 ) in vec3 aNormal;layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
```

### GLSL Shader:

```glsl
struct vertex {
    vec3 position;
    vec3 normal;
    vec3 color;
    vec2 texCoord;
};
```

```glsl
layout( location = 0 ) in vec3 aVertex;layout( location = 1 ) in vec3 aNormal;layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
```

```glsl
VkVertexInputBindingDescription vvibd[1];      // one of these per buffer data buffer
vvibd[0].binding = 0;          // which binding # this isvvibd[0].stride = sizeof( struct vertex );            // bytes between successive structs
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

```glsl
VkVertexInputAttributeDescription vviad[4];      // array per ve rtex input attribute
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position );  // 0
vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;       // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal );  // 12
vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;       // r, g, b
vviad[2].offset = offsetof( struct vertex, color );  // 24
vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;       // s, t
vviad[3].offset = offsetof( struct vertex, texCoord );  // 36
```
Telling the Pipeline about its Input

We will come to the Pipeline later, but for now, know that a Vulkan Pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its input.

```c
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
v pci.pNext = nullptr;
v pci.flags = 0;
v pci.stageCount = 2; // number of shader stages in this pipeline
v pci.pStages = vpssci;
v pci.pVertexInputState = &vpvisci;
v pci.pInputAssemblyState = &vpiasci;
v pci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr; // &vptsci
v pci.pViewportState = &vpvsci;
v pci.pRasterizationState = &vprscivgpci.pMultisampleState = ... GraphicsPipelineLayout;
v pci.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 pGraphicsPipeline );
```

Telling the Command Buffer what Vertices to Draw

We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

```c
VkBuffer buffers[1] = [MyVertexDataBuffer.buffer];
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );
const uint32_t vertexCount = sizeof( VertexData ) / sizeof( VertexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

### Drawing with an Indexed Buffer

Transmission Order

<table>
<thead>
<tr>
<th>Index</th>
<th>Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Triangle 0-2-3</td>
</tr>
<tr>
<td>1</td>
<td>Triangle 0-3-1</td>
</tr>
<tr>
<td>2</td>
<td>Triangle 4-5-7</td>
</tr>
</tbody>
</table>

**Actual Index Data**

<table>
<thead>
<tr>
<th>Index</th>
<th>Transmissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Triangles</td>
</tr>
<tr>
<td>1</td>
<td>Draw</td>
</tr>
</tbody>
</table>

### Drawing with an Indexed Buffer

```c
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
    VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
} VkIndexType;
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, offsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```
VkResult Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    return result;
}

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
// 0, 1 = firstBinding, bindingCount
vkCmdBindIndexedBuffer(CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32);
const uint32_t vertexCount = sizeof(JustVertexData) / sizeof(JustVertexData[0]);
const uint32_t indexCount = sizeof(JustIndexData) / sizeof(JustIndexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
const uint32_t vertexOffset = 0;

#ifdef VERTEX_BUFFER
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
#endif

#ifdef INDEX_BUFFER
vkCmdDrawIndexed(CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);
#endif

Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

A color-interpolated cube like this actually has both. Point #7 above has the same color, regardless of what face it is in. However, Point #7 has 3 different normal vectors, depending on which face you are defining. Same with its texture coordinates.

Thus, when using index-ed buffer drawing, you need to create a new vertex struct if any of (position, normal, color, texCoords) changes from what was previously-stored at those coordinates.
A **Data Buffer** is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a "Binary Large Object", or "BLOB").

It is up to you to be sure that the writer and the reader of the Data Buffer are interpreting the bytes in the same way!

Vulkan calls these things "Buffers". But, Vulkan calls other things "Buffers", too, such as Texture Buffers and Command Buffers. So, I have taken to calling these things "Data Buffers" and have even gone to far as to override some of Vulkan’s own terminology:

```cpp
typedef VkBuffer VkDataBuffer;
```

### Terminology Issues

**Vulkan: Buffers**

- `vkCreateBuffer()`: creates a `VkBuffer` data structure
- `vkGetBufferMemoryRequirements()`: returns `VkMemoryRequirements` for a `VkBuffer`
- `vkAllocateMemory()`: allocates memory
- `vkBindBufferMemory()`: binds a `VkBuffer` to a `VkMemory`
- `vkMapMemory()`: maps GPU memory

**Vulkan: Creating a Data Buffer**

```cpp
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = buffer size in bytes; // hint
vbci.usage = (or'ed bits of):
  VK_USAGE_TRANSFER_SRC_BIT
  VK_USAGE_TRANSFER_DST_BIT
  VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
  VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
  VK_USAGE_UNIFORM_BUFFER_BIT
  VK_USAGE_STORAGE_BUFFER_BIT
  VK_USAGE_INDEX_BUFFER_BIT
  VK_USAGE_VERTEX_BUFFER_BIT
  VK_USAGE_INDIRECT_BUFFER_BIT
vbci.sharingMode = one of:
  VK_SHARING_MODE_EXCLUSIVE
  VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t *) nullptr;

VkBuffer Buffer; result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```

Doesn’t actually allocate memory – just creates a `VkBuffer` data structure.
Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

```cpp
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();

VkDeviceMemory vdm;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );
<< do the memory copy >>
result = vkUnmapMemory( LogicalDevice, IN vdm );
```

Finding the Right Type of Memory

```cpp
int FindMemoryThatIsHostVisible()
{
    VkPhysicalDeviceMemoryProperties vpdm;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdm );
    for( unsigned int i = 0; i < vpdm.memoryTypeCount; ++i )
    {
        VkMemoryType vmt = vpdm.memoryTypes[ i ];
        if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )
            return i;
    }
    return -1;
}
```

```cpp
int FindMemoryThatIsDeviceLocal()
{
    VkPhysicalDeviceMemoryProperties vpdm;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdm );
    for( unsigned int i = 0; i < vpdm.memoryTypeCount; ++i )
    {
        VkMemoryType vmt = vpdm.memoryTypes[ i ];
        if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT ) != 0 )
            return i;
    }
    return -1;
}
```

Finding the Right Type of Memory

```
VkPhysicalDeviceMemoryProperties vpdm;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdm );
```

11 Memory Types:
- Memory 0:
- Memory 1:
- Memory 2:
- Memory 3:
- Memory 4:
- Memory 5:
- Memory 6:
- DeviceLocal
- DeviceLocal
- HostVisible HostCoherent
- HostVisible HostCoherent HostCached

2 Memory Heaps:
- Heap 0: size = 0x87c00000 DeviceLocal
- Heap 1: size = 0x60000000 DeviceLocal
Something I’ve Found Useful

I find it handy to encapsulate buffer information in a struct:

```c
typedef struct MyBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;

MyBuffer MyMatrixUniformBuffer;
```

It’s the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need from it.

Initializing a Data Buffer

```c
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    ... vbc.size = pMyBuffer->size = size;
    ... result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    ... pMyBuffer->vdm = vdm;
    ... }
```

Here’s the C struct to hold some uniform variables

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

Here’s the shader code to access those uniform variables

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

Filling those Uniform Variables

```c
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```
This C struct is holding the actual data. It is writeable by the application.

The MyBuffer does not hold any actual data itself. It just represents a container of data buffer information that will be used by Vulkan.

```
MyBuffer MyMatrixUniformBuffer;
```

The Data Buffer in GPU memory is holding the actual data. It is readable by the shaders.

```
uniform matBuf Matrices;
```

There is one more step in here—Descriptor Sets. Here's a quick preview…

CPU:

```
struct matBuf Matrices;
```

GPU:

```
uint matBuf Matrices;
```

### The Parade of Data

```cpp
MyBuffer MyMatrixUniformBuffer;
```

```cpp
uniform matBuf Matrices;
```

### The Descriptor Set for the Buffer

We will come to Descriptor Sets later, but for now think of them as the link between the BLOB of uniform variables in GPU memory and the block of variable names in your shader programs.

```cpp
VkDescriptorBufferInfo vdbi0;
vdbi0.buffer = MyMatrixUniformBuffer.buffer;
vdbi0.offset = 0; // bytes
vdbi0.range = sizeof(Matrices);
```

```cpp
VkWriteDescriptorSet vwds0;
// ds 0:vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds0.pNext = nullptr;
vwds0.dstSet = ... = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.pBufferInfo = &vdbi0;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
```

```cpp
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN 0, (VkCopyDescriptorSet *)nullptr );
```

We will come to Descriptor Sets later, but for now think of them as the link between the BLOB of uniform variables in GPU memory and the block of variable names in your shader programs.

```
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr ); // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 ); // 0 is the offset
    return result;
}
```

### Creating and Filling the Data Buffer – the Details

```cpp
Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
```

```cpp
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr ); // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 ); // 0 is the offset
    return result;
}
```
VkResult
Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to Init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}

Remember – to Vulkan and GPU memory, these are just bits. It is up to you to handle their meaning correctly.

The Shaders’ View of the Basic Computer Graphics Pipeline

- In general, you want to have a vertex and fragment shader as a minimum.
- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.
- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shaders

Vulkan Shader Stages

typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
Vulkan: GLSL Differences from OpenGL

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:
- In the compiler, there is an automatic `#define VULKAN 100`

Vertex and Instance indices:

- Both are 0-based

- `gl_VertexIndex`
- `gl_InstanceIndex`

In OpenGL, the Vulkan names make more sense.

- `gl_FragColor`:
  - In OpenGL, it broadcasts to all color attachments
  - In Vulkan, it just broadcasts to color attachment location #0
  - Best idea: don’t use it – explicitly declare out variables to have specific location numbers

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:
- In the compiler, there is an automatic `#define VULKAN 100`

These are called:
- `gl_VertexID`
- `gl_InstanceID`

In OpenGL, it just broadcasts to color attachment location #0

Specialization Constants:
- You can only use basic operators, declarations, and constructors
- Only for scalars, but a vector can be constructed from specialization constants

Specialization Constants for Compute Shaders:
- `gl_WorkGroupSize.x` is still as it was

Push Constants:
- Only for basic operators, declarations, and constructors
- Only for scalars, but a vector can be constructed from specialization constants

Shader combinations of separate texture data and samplers:

- `uniform sampler2D t;`
- `vec4 RGBA = texture( sampler2D( t, vST ), ) ;`

Descriptor Sets:
- `layout( set=0, binding=0 ) . . .  ;`

Specialization Constants:
- `layout( constant_id = 3 ) const int N = 5;`

Vulkan Shader Compiling
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V
- SPIR-V gets turned into fully-compiled code at runtime
- SPIR-V spec has been public for a couple of years –new shader languages are surely being developed
- OpenGL and OpenCL will be moving to SPIR-V as well

Advantages:
- You do:
  - External GLSL Compiler
- Driver does:
  - Compiler in driver
  - Vendor-specific code

GLSL Source

1. Software vendors don’t need to ship their shader source
2. Syntax errors appear during the SPIR-V step, not during runtime
3. Software can launch faster because half of the compilation has already taken place
4. This guarantees a common front-end syntax
5. This allows for other language front-ends
 SPIR-V:
Standard Portable Intermediate Representation for Vulkan

```
```

Shader file extensions:
- .vert Vertex
- .tesc Tessellation Control
- .tese Tessellation Evaluation
- .geom Geometry
- .frag Fragment
- .comp Compute

(Can be overridden by the –S option)

- V Compile for Vulkan
- G Compile for OpenGL
- I Directory(ies) to look in for #includes
- S Specify stage rather than get it from shaderfile extension
- c Print out the maximum sizes of various properties

Windows: glslangValidator.exe
Linux: setenv LD_LIBRARY_PATH /usr/local/common/gcc-6.3.0/lib64/

You Can Run the SPIR-V Compiler on Windows from a Bash Shell

1. Click on the Microsoft Start icon
2. Type word bash

You Can Run the SPIR-V Compiler on Windows from a Bash Shell

Pick one:
- Can get to your personal folders
- Does not have make
- Cannot get to your personal folders
- Does have make

Running glslangValidator.exe

```
glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
```
```
glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```
You can also run SPIR-V from a Linux Shell

```
$ glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
$ glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```

Compile for Vulkan ("-G" is compile for OpenGL)

The input file. The compiler determines the shader type by the file extension:
- `.vert` Vertex shader
- `.tcs` Tessellation Control Shader
- `.tecs` Tessellation Evaluation Shader
- `.geom` Geometry shader
- `.frag` Fragment shader
- `.comp` Compute shader

Specify the output file

Reading a SPIR-V File into a Vulkan Shader Module

```
int l2SpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
    FILE * fp;
    (void) fopen_s( &fp, filename.c_str(), "rb" );
    if( fp == NULL )
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str( ) );
        return VK_SHOULD_EXIT;
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
        fprintf( FpDebug, "Magic number for spir-v file '%s is 0x%08x -- should be 0x%08x
", filename.c_str(), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char * code = new unsigned char [ size ];
    fread( code, size, 1, fp );
    fclose( fp );
    VkResult
```

How do you know if SPIR-V compiled successfully?

Same as C/C++: the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of 0x07230203

So, if you do an `od -x` on the .spv file, the magic number looks like this:

```
0203 0723 . . .
```
VkShaderModuleCreateInfo
vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
vsmci.pNext = nullptr;
vsmci.flags = 0;
vsmci.codeSize = size;
vsmci.pCode = (uint32_t *)code;

VkResult result = vkCreateShaderModule( LogicalDevice, IN &vsmci, PALLOCATOR, pShaderModule );

fprintf( FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );

delete [] code;
return result;

You can also take a look at SPIR-V Assembly

glslangValidator.exe   -V   -H    sample-vert.vert -o sample-vert.spv

This prints out the SPIR-V "assembly" to standard output.
Other than nerd interest, there is no graphics-programming reason to look at this. 😊
SPIR-V: More Information

SPIR-V Tools:
http://github.com/KhronosGroup/SPIRV-Tools

Installing bash on Windows

1. Open Settings.
2. Click on Update & security.
3. Click on For Developers.
4. Under "Use developer features", select the Developer mode option to setup the environment to install Bash.
5. On the message box, click Yes to turn on developer mode.
6. After the necessary components install, you'll need to restart your computer.
7. Once your computer reboots, open Control Panel.
8. Click on Programs.
9. Click on Turn Windows features on or off.
10. Check the Windows Subsystem for Linux (beta) option.
11. Click OK.
12. Once the components installed on your computer, click the Restart now button to complete the task.

After your computer restarts, you will notice that Bash will not appear in the "Recently added" list of apps, this is because Bash isn't actually installed yet. Now that you have setup the necessary components, use the following steps to complete the installation of Bash.

1. Open Start, do a search for bash.exe, and press Enter.
2. On the command prompt, type y and press Enter to download and install Bash from the Windows Store.
3. Then you'll need to create a default UNIX user account. This account doesn't have to be the same as your Windows account. Enter the username in the required field and press Enter (you can't use the username "admin").
4. Close the "bash.exe" command prompt.

Now that you completed the installation and setup, you can open the Bash tool from the Start menu like you would with any other app.

Vulkan Sample Code

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

Sample Program Output
Caveats on the Sample Code

1. I’ve written everything out in appalling longhand.
2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.
3. At times, I could have hidden complexity, but I didn’t. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that’s a secret to you.
4. I’ve setup Vulkan structs every time they are used, even though, in most cases, they could have been partially or completely setup once and then re-used.
5. At times, I’ve setup things that didn’t need to be setup just to show you what could go there.
6. There are good uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
7. I’ve typedef’ed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you how certain options could be set.
10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

void InitGraphics()
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance();
    Init02CreateDebugCallbacks();
    Init03PhysicalDeviceAndGetQueueFamilyProperties();
    Init04LogicalDeviceAndQueue();
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool();
    Init06CommandBuffers();
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init08Swapchain();
    Init09DepthStencilImage();
    Init10RenderPasses();
    Init11Framebuffers();
    Init12Shader( "sample.vert.spv", &ShaderModuleVertex );
    Init12Shader( "sample.frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool();
    Init13DescriptorSetLayouts();
    Init13DescriptorSets();
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}

int main( int argc, char * argv[] )
{
    Width = 800;
    Height = 600;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE );
        PyDebug = stderr;
        fprintf(PyDebug, "PyDebug: Width = %d ; Height = %d
", Width, Height);
        return 1;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
    Reset();
    InitGraphics();
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents();
        Time = glfwGetTime();          // elapsed time, in double-precision seconds
        UpdateScene();
        RenderScene();
    }
    fprintf(FpDebug, "Closing the GLFW window
");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan();
    glfwDestroyWindow( MainWindow );
    glfwTerminate();
    return 0;
A Colored Cube

struct vertex
{
  glm::vec3 position;
  glm::vec3 normal;
  glm::vec3 color;
  glm::vec2 texCoord;
};

struct vertex VertexData[ ] =
{
  // triangle 0-2-3:
  // vertex #0:
  {
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. }
  },
  // vertex #2:
  {
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. }
  },
  // vertex #3:
  {
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
  }
};

What if you don’t need all of this information?

For example, what if you are not doing texturing in this application? Should you re-do this struct and leave the texCoord element out?

As best as I can tell, the only penalty for leaving in vertex attributes you aren’t going to use is memory space, but not performance. So, I recommend keeping this struct intact, and, if you don’t need texturing, simply don’t use the texCoord values in your vertex shader.

The Vertex Data is in a Separate File

#include "SampleVertexData.cpp"

struct vertex
{
  glm::vec3 position;
  glm::vec3 normal;
  glm::vec3 color;
  glm::vec2 texCoord;
};

struct vertex VertexData[ ] =
{
  // triangle 0-2-3:
  // vertex #0:
  {
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. }
  },
  // vertex #2:
  {
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. }
  },
  // vertex #3:
  {
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
  }
};

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums
2. Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary information into a function. For example, where we might normally say in C++:

   ```cpp
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex, OUT &Queue );
   ```

   we would actually say in C:

   ```c
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```
Vulkan Conventions

VkXxx is a typedef, probably a struct
vkXxx( ) is a function call
VK XXX is a constant

My Conventions

“Init” in a function call name means that something is being setup that only needs to be setup once.

The number after “Init” gives you the ordering.

In the source code, after main( ) comes InitGraphics( ), then all of the InitXXYYY( ) functions in numerical order. After that comes the helper functions.

“Find” in a function call name means that something is being looked for.

“Fill” in a function call name means that some data is being supplied to Vulkan.

“IN” and “OUT” ahead of pointer (address) arguments are just there to let you know how a pointer is used by the function. Otherwise, they have no significance.

#define IN
#define OUT

Querying the Number of Something and Allocating Structures to Hold Them All

This way of querying information is a recurring Vulkan (and OpenCL) pattern (get used to it):

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *) nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );

# Reporting Error Results, I

struct errorcode
{
VkResult resultCode;
std::string meaning;
}

ErrorCodes[] =
{
VK_NOT_READY, "Not Ready" },
VK_TIMEOUT, "Timeout" ),
VK_EVENT_SET, "Event Set" ),
VK_EVENT_RESET, "Event Reset" ),
VK_INCOMPLETE, "Incomplete" ),
VK_ERROR_OUT_OF_HOST_MEMORY, "Out of Host Memory" ),
VK_ERROR_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" ),
VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" ),
VK_ERROR_DEVICE_LOST, "Device Lost" ),
VK_ERROR_MEMORY_MAP_FAILED, "Memory Map Failed" ),
VK_ERROR_LAYER_NOT_PRESENT, "Layer Not Present" ),
VK_ERROR_EXTENSION_NOT_PRESENT, "Extension Not Present" ),
VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" ),
VK_ERROR_INCOMPATIBLE_DRIVER, "Incompatible Driver" ),
VK_ERROR_TOO_MANY_OBJECTS, "Too Many Object" ),
VK_ERROR_FORMAT_NOT_SUPPORTED, "Format Not Supported" ),
VK_ERROR_FRAGMENTED_POOL, "Fragmented Pool" ),
VK_ERROR_SURFACE_LOST, "Surface Lost" ),
VK_ERROR_NATIVE_WINDOW_IN_USE, "Native Window in Use" ),
VK_SUBOPTIMAL_KHR, "Suboptimal" ),
VK_ERROR_OUT_OF_DATE_KHR, "Out of Date" ),
VK_ERROR_INVALID_EXCHANGE_FORMAT_KHR, "Invalid Exchange Format" ),
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" ),
VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" ),
VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" ),
VK_ERROR_OUT_OF_POOL_MEMORY, "Out of Pool Memory" ),
VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR, "Invalid External Handle" ),
}. 
### Reporting Error Results, II

```cpp
void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s
", prefix.c_str(), "Successful");
        fflush(FpDebug);
        return;
    }
    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = "";
    for( int i = 0; i < numErrorCodes; i++ )
    {
        if( result == ErrorCodes[i].resultCode )
        {
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }
    fprintf( FpDebug, "%s: %s
", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```

### Extras in the Code

```cpp
#define REPORT(s)               PrintVkError( result, s );  fflush(FpDebug);
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****
", s );  fflush(FpDebug); }
bool Paused;
bool Verbose;
#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
```

### Setting Up GLFW

```cpp
void InitGLFW( )
{
    glfwInit( );
    glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );
    glfwWindowHint( GLFW_RESIZABLE, GLFW_FALSE );
    MainWindow = glfwCreateWindow( Width, Height, "Vulkan Sample", NULL, NULL);
    VkResult result = glfwCreateWindowSurface( Instance, MainWindow, NULL, &Surface );
    glfwSetErrorCallback( GLFWErrorCallback );
    glfwSetKeyCallback( MainWindow, GLFWKeyboard );
    glfwSetMouseButtonCallback( MainWindow, GLFWMouseButton );
    glfwSetCursorPosCallback( MainWindow, GLFWMouseMotion );
    glfwSetMouseButtonCallback( MainWindow, GLFWMouseButton );
}
```
GLFW Keyboard Callback

```c
void GLFWKeyboard(GLFWwindow * window, int key, int scancode, int action, int mods )
{
    if( action == GLFW_PRESS )
    {
        switch( key )
        {
            //case GLFW_KEY_M:
            case 'm':
            case 'M':
            Mode++;
            if( Mode >= 2 )
            Mode = 0;
            break;
            default:
                fprintf( FpDebug, "Unknow key hit: 0x%04x = '%c'
", key, key );
                fflush(FpDebug);
        }
    }
}
```

GLFW Mouse Button Callback

```c
void GLFWMouseButton( GLFWwindow *window, int button, int action, int mods )
{
    int b = 0;              // LEFT, MIDDLE, or RIGHT
    // get the proper button bit mask:
    switch( button ){
    case GLFW_MOUSE_BUTTON_LEFT:
        b = LEFT;               break;
    case GLFW_MOUSE_BUTTON_MIDDLE:
        b = MIDDLE;             break;
    case GLFW_MOUSE_BUTTON_RIGHT:
        b = RIGHT;              break;
    default:
        b = 0;
        fprintf( FpDebug, "Unknown mouse button "button", button ");
        break;
    }
    // button down sets the bit, up clears the bit:
    if( action == GLFW_PRESS )
    {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;Ymouse = (int)ypos;
        ActiveButton |= b;              // set the proper bit
    }else{
        ActiveButton &= ~b;             // clear the proper bit
    }
}
```

GLFW Mouse Motion Callback

```c
void GLFWMouseMotion( GLFWwindow *window, double xpos, double ypos )
{
    int dx = (int)xpos - Xmouse;            // change in mouse coords
    int dy = (int)ypos - Ymouse;
    if( ( ActiveButton & LEFT ) != 0 )
    {
        Xrot += ( ANGFACT*dy );
        Yrot += ( ANGFACT*dx );
    }
    if( ( ActiveButton & MIDDLE ) != 0 )
    {
        Scale += SCLFACT * (float) ( dx - dy );
        // keep object from turning inside-out or disappearing:
        if( Scale < MINSCALE )
            Scale = MINSCALE;
    }
    Xmouse = (int)xpos;                     // new current position
    Ymouse = (int)ypos;
}
```

Looping and Closing GLFW

```c
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );
    if( glfwGetTime() - Time )
    {
        Time = glfwGetTime();
        UpdateScene( );
        RenderScene( );
    }
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
}
```
What is GLM?

GLM is a set of C++ classes and functions to fill in the programming gaps in writing the basic vector and matrix mathematics for OpenGL applications. However, even though it was written for OpenGL, it works fine with Vulkan (with one small exception which can be worked around).

Even though GLM looks like a library, it actually isn’t—it is all specified in *.hpp header files so that it gets compiled in with your source code.

You can find it at: [http://glm.g-truc.net/0.9.8.5/](http://glm.g-truc.net/0.9.8.5/)

You invoke GLM like this:

```c++
#define    GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include  <glm/gtc/matrix_inverse.hpp>
```

If GLM is not installed in a system place, put it somewhere you can get access to.

Why are we even talking about this?

All of the things that we have talked about being deprecated in OpenGL are really deprecated in Vulkan—built-in pipeline transformations, begin-end, fixed-function, etc. So, where you might have said in OpenGL:

```c
    gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0.);
    glRotatef( (GLfloat)Yrot, 0., 1., 0. );
    glRotatef( (GLfloat)Xrot, 1., 0., 0. );
    glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

You would now have to say:

```c
    glm::mat4 modelview;
    glm::vec3 eye(0.,0.,3.);
    glm::vec3 look(0.,0.,0.);
    glm::vec3 up(0.,1.,0.);
    modelview = glm::lookAt( eye, look, up );
    modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) );
    modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) );
    modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) );
```

The Most Useful GLM Variables, Operations, and Functions

GLM recommends that you use the “glm::” syntax and avoid “using namespace” syntax because they have not made any effort to create unique function names.

```c
// constructor:
glm::mat4( );  // identity matrix
glm::vec4( );
```

```
// multiplications:
glm::mat4 * glm::mat4
glm::vec4 * glm::vec4( glm::vec3, 1. )  // promote vec3 to a vec4 via a constructor
```

```
// emulating OpenGL transformations with concatenation:
glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
```

```c
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```
// viewing volume (assign, not concatenate):
glm::mat4 glm::ortho( float left, float right, float bottom, float top, float near, float far );
glm::mat4 glm::orthographic( float left, float right, float bottom, float top, float near, float far );

// viewing (assign, not concatenate):
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );

Your Sample2017.zip File Contains GLM Already

Matrix Multiplication is not Commutative
Matrix Multiplication is Associative

\[
\begin{pmatrix}
  x' \\
  y' \\
  z'
\end{pmatrix}
= \begin{pmatrix}
  T_{+A,+B} & [R_y] & T_{-A,-B}
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]

One matrix — the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

\[
\begin{pmatrix}
  T_{+A,+B} & [R_y] & T_{-A,-B}
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
= \begin{pmatrix}
  [T_{+A,+B}][R_y][T_{-A,-B}]
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]

Why Isn't The Normal Matrix just the same as the Model Matrix?

It is, if the Model Matrix is all rotations and uniform scalings, but if it has non-uniform scalings, then it is not.

Wrong!

Right!

 glm::mat3 NormalMatrix = glm::mat3(Model);

Original object and normal

 glm::mat3 NormalMatrix = glm::inverseTranspose( glm::mat3(Model) );

Instancing

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan
Instancing – What and why?

- Instancing is the ability to draw the same object multiple times
- It uses all the same vertices and graphics pipeline each time
- It avoids the overhead of the program asking to have the object drawn again, letting the GPU/driver handle all of that

```
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

But, this will only get us multiple instances of identical objects drawn on top of each other. How can we make each instance look differently?

Making each Instance look differently – Approach #1

Use the built-in vertex shader variable `gl_InstanceIndex` to define a unique display property, such as position or color.

`gl_InstanceIndex` starts at 0

In the vertex shader:

```
int NUMINSTANCES = 16;
float DELTA = 3.0;
float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vec3 vColor = vec3(1.0, float((1.+gl_InstanceIndex) / float(NUMINSTANCES)), 0.0);
xdelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.00;
ydelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.00;
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.0), 1.0);
gl_Position = PVM * vertex;
```

Making each Instance look differently – Approach #2

Put the unique characteristics in a uniform buffer and reference them

Still uses `gl_InstanceIndex`

In the vertex shader:

```
layout(std140, set = 3, binding = 0) uniform colorBuf
{
  vec3 uColors[1024];
} Colors;
out vec3 vColor;
...
int index = gl_InstanceIndex % 1024; // 0 - 1023
vColor = Colors.uColors[index];
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```
Put a series of unique characteristics in a data buffer, one element per instance.

Read a new characteristic for each instance.

Internally uses `gl_InstanceIndex`, but you don’t.

**How We Constructed the Graphics Pipeline Structure Before**

```c
VkVertexInputBindingDescription vvbds[1];  // an array containing one of these per buffer being used
  vvbds[0].binding = 0;  // which binding # this is
  vvbds[0].stride = sizeof(struct vertex);  // bytes between successive
  vvbds[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

This definition says that we should advance through the input buffer by this much every time we hit a new vertex.
```

```c
VkVertexInputAttributeDescription vviads[4];  // an array containing one of these per vertex attribute in all bindings
  vviads[0].location = 0;  // location in the layout decoration
  vviads[0].binding = 0;  // which binding description this is part of
  vviads[0].format = VK_FORMAT_VEC3;  // x, y, z
  vviads[0].offset = offsetof(struct vertex, position);  // 0
  vviads[1].location = 1;
  vviads[1].binding = 0;
  vviads[1].format = VK_FORMAT_VEC3;  // nx, ny, nz
  vviads[1].offset = offsetof(struct vertex, normal);  // 12
  vviads[2].location = 2;
  vviads[2].binding = 0;
  vviads[2].format = VK_FORMAT_VEC3;  // r, g, b
  vviads[2].offset = offsetof(struct vertex, color);  // 24
  vviads[3].location = 3;
  vviads[3].binding = 0;
  vviads[3].format = VK_FORMAT_VEC2;  // s, t
  vviads[3].offset = offsetof(struct vertex, texCoord);  // 36
```
How We Constructed the Graphics Pipeline Structure Before

VkPipelineVertexInputStateCreateInfo vpvisci;
    // used to describe the input vertex attributes
    vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
    vpvisci.pNext = nullptr;
    vpvisci.flags = 0;
    vpvisci.vertexBindingDescriptionCount = 1;
    vpvisci.pVertexBindingDescriptions = vvibd;
    vpvisci.vertexAttributeDescriptionCount = 4;
    vpvisci.pVertexAttributeDescriptions = vviad;

VkGraphicsPipelineCreateInfo vgpci;
    vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    vgpci.pNext = nullptr;
    vgpci.flags = 0;
    . . .
    vgpci.pVertexInputState = &vpvisci;
    . . .
    result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
                                           PALLOCATOR, OUT pGraphicsPipeline );

How We Construct the Graphics Pipeline Structure Now

Let’s assign a different color per instance. Create a data buffer with one glm::vec3 (to hold r, g, b) for each Instance.

This definition says that we should advance through the input buffer by this much every time we hit a new instance.

How We Construct the Graphics Pipeline Structure Now

Let’s assign a different color per instance. Create a data buffer with one glm::vec3 (to hold r, g, b) for each Instance.

Note: same names as before, but different sizes.
How We Write the Vertex Shader Now

```glsl
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
layout( location = 4 ) in vec3 aInstanceColor;

layout ( location = 0 ) out vec3 vNormal;
layout ( location = 1 ) out vec3 vColor;
layout ( location = 2 ) out vec2 vTexCoord;

void main( )
{
  mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
  vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );
  //vColor = aColor;
  vColor = aInstanceColor;
  vTexCoord = aTexCoord;
  gl_Position = PVM * vec4( aVertex, 1. );
}
```

How We Write the Vertex Shader Now

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;
```

In OpenGL, these are all in one set. They all get bound, whether you need them here or not.

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.

```glsl
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:

```c++
struct matBuf {
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
    glm::mat3 uNormalMatrix;
}; Matrices;

struct lightBuf {
    float uKa, uKd, uKs, uShininess;
    glm::vec4 uLightPos; // Light position
    glm::vec4 uLightSpecularColor;
    glm::vec4 uEyePos; // Camera position
}; Light;

struct miscBuf {
    float uTime;
    int uMode;
    int uLighting;
}; Misc;

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 {
    float uKa, uKd, uKs, uShininess;
    vec4 uLightPos; // Light position
    vec4 uLightSpecularColor;
    vec4 uEyePos; // Camera position
} Light;

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

---

**CPU:**

Uniform data created in a C++ data structure

**GPU:**

Uniform data used in the shader

---

**Descriptor Sets**

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

- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size

**Uniform data in a “blob”**

- Knows where each piece of data starts
- Doesn’t know the CPU or GPU data structure

**Uniform data used in the shader**

- Knows the shader data structure
- Doesn’t know where each piece of data starts

---

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

```c++
struct matBuf {
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
    glm::mat3 uNormalMatrix;
};
```
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.

```
// DS #0:
MatrixSet[0].binding = 0;
mat4 uModelMatrix;mat4 uViewMatrix;mat4 uProjectionMatrix;mat3 uNormalMatrix;
MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
MatrixSet[0].descriptorCount = 1;
layout( std140, set = 1, binding = 0 ) uniform lightBuf
MatrixSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
MatrixSet[0].pImmutableSamplers = (VkSampler *)nullptr;
float  uKa, uKd, uKs, uShininess;
vec4 uLightPos;vec4 uLightSpecularColor;vec4 uEyePos;

// DS #1:
Light;
LightSet[0].binding = 0;
LightSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
LightSet[0].descriptorCount = 1;
LightSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
LightSet[0].pImmutableSamplers = (VkSampler *)nullptr;
layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```

Computer Graphics

mjb – November 24, 2018
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```c
VkResult Init14GraphicsPipelineLayout() {
    VkResult result;
    VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}
```

Step 4: Allocating the Memory for Descriptor Sets

```c
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]);
    return result;
}
```

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

```c
// 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;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pTexelBufferView = (VkBufferView *)nullptr;
// ds 1:
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds1.pNext = nullptr;
vwds1.dstSet = DescriptorSets[1];
vwds1.dstBinding = 0;
vwds1.dstArrayElement = 0;
vwds1.descriptorCount = 1;
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.pBufferInfo = IN &vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
```

```
This struct identifies what buffer it owns and how big it is
This struct identifies what buffer it owns and how big it is
This struct identifies what buffer it owns and how big it is
This struct identifies what texture sampler and image view it owns
Good to use sizeof
```
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;
```

```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.pImageInfo = &vdii0;
vwds3.pTexelBufferView = (VkBufferView *)nullptr;
```

```
uint32_t copyCount = 0;
```

```
// this could have been done with one call and an array of VkWriteDescriptorSets:

vkUpdateDescriptorSets(LogicalDevice, descriptorSetCount, (VkWriteDescriptorSet *)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;

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 = GraphicsPipelineLayout;
vgpci.renderPass = RenderPass;
vgpci.subpass = 0;

vgpci.basePipelineHandle = VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;

result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, (VkGraphicsPipelineCreateInfo *)nullptr, 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], 0, (uint32_t *)nullptr);
```
What is the Vulkan Graphics Pipeline?

The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context".

There's a lot that goes into it.

For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.

The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

The First Step: Create the Graphics Pipeline Layout

The Graphics Pipeline Layout is fairly static. Only the layout of the Descriptor Sets and information on the Push Constants need to be supplied.

Vulkan: A Pipeline Records the Following Items:

- Pipeline Layout: DescriptorSets, PushConstants
- Which Shaders are going to be used
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissoring: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, blendColorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

*Bold/Italics* indicates that this state item can also be set with Dynamic Variables.
Creating a Graphics Pipeline from a lot of Pieces

Creating a Typical Graphics Pipeline

Layouts which stage (VERTEX, etc.) binding

VkResult Init14GraphicsVertexFragmentPipeline

vertexShader

VkShaderModule

vertexShader

VkShaderModule

vertexShader

VkPipelineLayoutCreateInfo

vertexShader

VkPipelineShaderStageCreateInfo

vertexShader

VkVertexInputStateCreateInfo

vertexShader

VkPipelineInputAssemblyStateCreateInfo

vertexShader

VkPipelineRasterizationStateCreateInfo

vertexShader

VkPipelineColorBlendStateCreateInfo

vertexShader

VkPipelineDynamicStateCreateInfo

vertexShader

vkCreatePipelineLayout( )

VkVertexInputBindingDescription

VkVertexInputAttributeDescription

vkCreateGraphicsPipeline( )

These settings seem pretty typical to me. Let’s write a simplified Pipeline-creator that accepts Vertex and Fragment shader modules and the topology, and always uses the settings in red above.

Link in the Shaders

Use one vvpsci array member per shader module you are using

Use one vvibd array member per vertex shader input array-of-structures you are using

Link in the Per-Vertex Attributes

Use one vvadv array member per element in the struct for the array-of-structures element you are using as vertex input

These are defined at the top of the sample code so that you don’t need to use confusing image-looking formats for positions, normals, and tex coords
Declare the binding descriptions and attribute descriptions

Declare the vertex topology

Tessellation Shader info

Geometry Shader info

What is “Primitive Restart Enable”?  

vpvsci.primitiveRestartEnable = VK_FALSE;

“Restart Enable” is used with:

• Indexed drawing.
• Triangle Fan and “Strip topologies

If vpvsci.primitiveRestartEnable is VK_TRUE, then a special “index” indicates that the
primitive should start over. This is more efficient than explicitly ending the current
primitive and explicitly starting a new primitive of the same type.

typedef enum VkIndexType
{
    VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
    VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
} VkIndexType;

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 0xffff
If your VkIndexType is VK_INDEX_TYPE_UINT32, then the special index is 0xffffffff

When using the primitive restart code, the easy way to do it is like this:

short int restartIndex = ~0;

or,

int restartIndex = ~0;

One Really Good use of Restart Enable is in Drawing Terrain
Surfaces with Triangle Strips
What is the Difference Between Changing the Viewport and Changing the Scissoring?

Viewporting operates on vertices and takes place right before the rasterizer. Changing the vertical part of the viewport causes the entire scene to get scaled (scrunch) into the viewport area.

Scissoring operates on fragments and takes place right after the rasterizer. Changing the vertical part of the scissor causes the entire scene to get clipped where it falls outside the scissor area.

What is “Depth Clamp Enable”? 

Depth Clamp Enable causes the fragments that would normally have been discarded because they are closer to the viewer than the near clipping plane to instead get projected to the near clipping plane and displayed.

A good use for this is Polygon Capping:

The front of the polygon is clipped, revealing to the viewer that this is really a shell, not a solid.

The gray area shows what would happen without depthClampEnable (except it would have been red).
What is “Depth Bias Enable”?

```cpp
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasClamp = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;
```

Depth Bias Enable allows scaling and translation of the Z-depth values as they come through the rasterizer to avoid Z-fighting.

**Z-fighting**

---

MultiSampling State

```cpp
VkPipelineMultisampleStateCreateInfo vpmsci = {};
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
vpmsci.sampleShadingEnable = VK_FALSE;
vpmsci.minSampleShading = 0;
vpmsci.pSampleMask = (VkSampleMask *)nullptr;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;
```

Declare information about how the multisampling will take place.

---

Color Blending State for each Color Attachment

Create an array with one of these for each color buffer attachment. Each color buffer attachment can use different blending operations.

```cpp
VkPipelineColorBlendAttachmentState vpcbas;
```

```cpp
vpcbas.blendEnable = VK_FALSE;
vpcbas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vpcbas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
vpcbas.colorBlendOp = VK_BLEND_OP_ADD;
vpcbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpcbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpcbas.alphaBlendOp = VK_BLEND_OP_ADD;
vpcbas.colorWriteMask =
    VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
```

This controls blending between the output of each color attachment and its image memory.

---

Color Blending State for each Color Attachment

```cpp
VkPipelineColorBlendStateCreateInfo vpcbsci = {};
```

```cpp
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
```

```cpp
vpcbsci.pNext = nullptr;
```

```cpp
vpcbsci.flags = 0;
```

```cpp
vpcbsci.logicOpEnable = VK_FALSE;
```

```cpp
vpcbsci.logicOp = VK_LOGIC_OP_COPY;
```

```cpp
#ifdef CHOICES
    VK_LOGIC_OP_CLEAR
    VK_LOGIC_OP_AND
    VK_LOGIC_OP_AND_REVERSE
    VK_LOGIC_OP_COPY
    VK_LOGIC_OP_AND_INVERTED
    VK_LOGIC_OP_NO_OP
    VK_LOGIC_OP_XOR
    VK_LOGIC_OP_OR
    VK_LOGIC_OP_NOR
    VK_LOGIC_OP_EQUIVALENT
    VK_LOGIC_OP_INVERT
    VK_LOGIC_OP_OR_REVERSE
    VK_LOGIC_OP_COPY_INVERTED
    VK_LOGIC_OP_NAND
    VK_LOGIC_OP_SET
#endif
```

```cpp
vpcbsci.attachmentCount = 1;
```

```cpp
vpcbsci.pAttachments = &vpcbas;
```

```cpp
vpcbsci.blendConstants[0] = 0;
vpcbsci.blendConstants[1] = 0;
vpcbsci.blendConstants[2] = 0;
vpcbsci.blendConstants[3] = 0;
```

This controls blending between the output of the fragment shader and the input to the color attachments.
## Which Pipeline Variables can be Set Dynamically?

```c
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };  
```

- `VK_DYNAMIC_STATE_VIEWPORT` — `vkCmdSetViewport( )`
- `VK_DYNAMIC_STATE_SCISSOR` — `vkCmdSetScissor( )`
- `VK_DYNAMIC_STATE_LINE_WIDTH` — `vkCmdSetLineWidth( )`
- `VK_DYNAMIC_STATE_DEPTH_BIAS `— `vkCmdSetDepthBias( )`
- `VK_DYNAMIC_STATE_BLEND_CONSTANTS `— `vkCmdSetBlendConstants( )`
- `VK_DYNAMIC_STATE_DEPTH_BOUNDS` — `vkCmdSetDepthZBounds( )`
- `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` — `vkCmdSetStencilCompareMask( )`
- `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK` — `vkCmdSetStencilWriteMask( )`
- `VK_DYNAMIC_STATE_STENCIL_REFERENCE` — `vkCmdSetStencilReference( )`

## Stencil Operations for Front and Back Faces

- **Stencil Operations**
- **Stencil Op States**
  - `VkStencilOpState` for front and back faces

```c
VkStencilOpState vsosf;  // front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds
vsosf.compareOp = VK_COMPARE_OP_NEVER; // compare operation
vsosf.compareMask = ~0;  // mask for compare operation
vsosf.writeMask = ~0;  // mask for writing to stencil
vsosf.reference = 0;  // initial reference value
```

**Magic Lenses**

- **Polygon edges without Z-fighting**

## Uses for Stencil Operations

- **Polygon edges without Z-fighting**

## Operations for Depth Values

- **Depth Values**

```c
VkPipelineDepthStencilStateCreateInfo vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthComparisionOp = VK_COMPARE_OP_LESS;
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.stencilTestEnable = VK_FALSE;
```

**Depth Values**

- **Depth Values**
- **Depth Values**

- **Depth Values**

```c
if (!front) {  
    vfront = vold;  
    vback = vold;  
    voldDepthBounds = 0;  
    voldDepthBounds = 1;  
    voldStencilTestEnable = VK_FALSE;  
```
Putting it all Together! (finally…)

```cpp
VKGraphicsPipelineCreateInfo
{vendor:
  vkpcpiType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,
  vkpcpiNext = nullptr,
  vkpcpiFlags = 0;

rgba CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
rgba

vkpcpi.stageCount = 2; // number of stages in this pipeline
vkpcpi.pStages = vpssci;
vkpcpi.pVertexInputState = &vpvisci;
vkpcpi.pInputAssemblyState = &vpiasci;
vkpcpi.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vkpcpi.pViewportState = &vpvsci;
vkpcpi.pRasterizationState = &vprsci;
vkpcpi.pMultisampleState = &vpmsci;
vkpcpi.pDepthStencilState = &vpdssci;
vkpcpi.pColorBlendState = &vpcbsci;
vkpcpi.pDynamicState = &vpdsci;

vkpcpi.layout = IN GraphicsPipelineLayout;
vkpcpi.renderPass = IN RenderPass;
vkpcpi.subpass = 0; // subpass number
vkpcpi.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vkpcpi.basePipelineIndex = 0;

vkCreateGraphicsPipelines
{LogicalDevice, VK_NULL_HANDLE, 1, IN &
vkpcpi
PALLOCATOR, OUT pGraphicsPipeline

return result;
```

Later on, we will Bind the Graphics Pipeline to the Command Buffer when Drawing

```cpp
vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```

Vulkan: a More Typical (and Simplified) Block Diagram

![Vulkan Block Diagram](http://cs.oregonstate.edu/~mjb/vulkan)
Vulkan Queues and Command Buffers

- Graphics commands are recorded in command buffers, e.g., `vkCmdDoSomething(cmdBuffer, ...)`;
- You can have as many simultaneous Command Buffers as you want;
- Each command buffer can be filled from a different thread;
- Command Buffers record our commands, but no work takes place until a Command Buffer is submitted to a Queue;
- We don’t create Queues – the Logical Device has them already;
- Each Queue belongs to a Queue Family;
- We don’t create Queue Families – the Physical Device already has them.

**CPU Thread**  
buffer  
queue  
buffer  
queue  
buffer  
queue

Querying what Queue Families are Available

```c
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceFamilyProperties(IN PhysicalDevice, &count, OUT &vqfp);
for(unsigned int i = 0; i < count; i++) {
    fprintf(FpDebug, "	%d: Queue Family Count = %2d  ;   ", i, vqfp[i].queueCount);
    if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)       fprintf(FpDebug, " Graphics ");
    if((vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)       fprintf(FpDebug, " Compute ");
    if((vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)       fprintf(FpDebug, " Transfer ");
    fprintf(FpDebug, "\n");
}
```

```c
Similarly, we Can Write a Function that Finds the Proper Queue Family
```

```c
int FindQueueFamilyThatDoesGraphics() {
    uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT vqfp);
    for(unsigned int i = 0; i < count; i++) {
        if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) return i;
    }
    return -1;
}
```

Creating a Logical Device Queue Needs to Know Queue Family Information

```c
float queuePriorities[] = {
    1. // one entry per queueCount
};
VkDeviceQueueCreateInfo vdqci[1];
vdqci.sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
vdqci.pNext = nullptr;
vdqci.flags = 0;
vdqci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci.queueCount = 1;
vdqci.queuePriorities = (float *) queuePriorities;
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1; // # of device queues wanted
vdci.pQueueCreateInfos = IN &vdqci[0]; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.ppEnabledLayerNames = myDeviceLayers;
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures; // already created
result = vkCreateLogicalDevice(IN PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice);
```

```c
VkQueue Queue;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
uint32_t queueIndex = 0;
result = vkGetDeviceQueue(IN LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue);
```
Creating the Command Pool as part of the Logical Device

```c
VkResult Init06CommandPool() {
    VkResult result;
    VkCommandPoolCreateInfo vcpci;
    vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcpci.pNext = nullptr;
    vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT |
                  VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
    result = vkCreateCommandPool(LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
    return result;
}
```

Creating the CommandBuffers

```c
VkResult Init06CommandBuffers() {
    VkResult result;
    // allocate 2 command buffers for the double-buffered rendering:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 2;           // 2, because of double-buffering
        result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
    }
    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 1;
        result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer);
    }
    return result;
}
```

Beginning a Command Buffer

```c
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    uint32_t nextImageIndex;
    result = vkAcquireNextImage(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);
    VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
    . . .
    result = vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
}
```
These are the Commands that could be entered into the Command Buffer, I

```
vkCmdBeginQuery( commandBuffer, flags );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexType );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdBlitImage( commandBuffer, filter );
vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );
vkCmdBindDescriptorSetWithTemplate( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdCopyBuffer( commandBuffer, pRegions );
vkCmdCopyBufferToImage( commandBuffer, pRegions );
vkCmdCopyImage( commandBuffer, pRegions );
vkCmdCopyImageToBuffer( commandBuffer, pRegions );
vkCmdCopyQueryPoolResults( commandBuffer, flags );
vkCmdDebugMarkerBeginEXT( commandBuffer, pMarkerInfo );
vkCmdDebugMarkerEndEXT( commandBuffer );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );
vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );
vkCmdDispatchIndirect( commandBuffer, offset );
vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdDrawIndexedIndirect( commandBuffer, stride );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdDrawIndexedIndirectCountAMD( commandBuffer, stride );
vkCmdDrawIndirect( commandBuffer, stride );
vkCmdDrawIndirectCountAMD( commandBuffer, stride );
vkCmdEndQuery( commandBuffer, query );
vkCmdEndRenderPass( commandBuffer );
vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask);
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
```

These are the Commands that could be entered into the Command Buffer, II

```
{ RenderScene( ) }
VkResult
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;vsci.flags = 0;
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
VkClearValue vcv[2];
vcv[0].color = vccv;
result = vkAcquireNextImage( imageReadySemaphore, IN &imageReadySemaphore, OUT &image );
VkOffset2D o2d = { 0, 0 };
VkExtent2D e2d = { Width, Height };
VkRect2D r2d = { o2d, e2d };
VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
VkRenderPassBeginInfo vrpbi;
vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;vrpbi.pNext = nullptr;
vrpbi.renderPass = RenderPass;
vrpbi.clearValueCount = 2;
vrpbi.pClearValues = vcv;               // used for VK_ATTACHMENT_LOAD_OP_CLEAR
vkCmdBeginRenderPass( commandBuffer, query, queryCount );
```
VkViewport viewport =
{
  0.0, // x0,
  0.0, // y
  Width, // Width,
  Height, // Height,
  0.0, // minDepth
  1.0. // maxDepth
};

vkCmdSetViewport
(CommandBuffers[nextImageIndex], 0, 1, IN &viewport);

VkRect2D scissor =
{
  0, 0, // x0, y
  Width, Height // Width, Height
};

vkCmdSetScissor
(CommandBuffers[nextImageIndex], 0, 1, IN &scissor);

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

vkCmdBindPushConstants
(CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values);

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };

VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers
(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);

const uint32_t instanceCount = 1;

const uint32_t firstVertex = 0;

const uint32_t firstInstance = 0;

vkCmdDraw
(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdEndRenderPass
(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer
(CommandBuffers[nextImageIndex]);

VkSubmitInfo vsi;

vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

vsi.pNext = nullptr;

vsi.commandBufferCount = 1;

vsi.pCommandBuffers = &CommandBuffer;

vsi.waitSemaphoreCount = 1;

vsi.pWaitSemaphores = imageReadySemaphore;

vsi.signalSemaphoreCount = 0;

vsi.pSignalSemaphores = (VkSemaphore *)nullptr;

vkQueueSubmit
(presentQueue, 1, IN &vsi, IN renderFence);

result = vkWaitForFences
(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);

vkDestroyFence
(LogicalDevice, renderFence, PALLOCATOR);

VkPresentInfo vpi;

vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO;

vpi.pNext = nullptr;

vpi.swapchainCount = 1;

vsi.pSwapchains = &SwapChain;

vsi.pImageIndices = &nextImageIndex;

vsi.pResults = (VkResult *)nullptr;

vkQueuePresent
(presentQueue, IN &vpi);
How We Think of OpenGL Framebuffers

- **Video Driver**
- **Update**
- **Refresh**
- **Depth-Buffer**
- **Double-buffered Color Framebuffers**

Vulkan Thinks of it as a Ring Buffer

- **Swap Chain**

What is a Swap Chain?

Vulkan does not use the idea of a “back buffer”. So, we need a place to render into before moving an image into place for viewing. The is called the **Swap Chain**.

In essence, the Swap Chain manages one or more image objects that form a sequence of images that can be drawn into and then given to the Surface to be presented to the user for viewing.

- **Swap Chains are arranged as a ring buffer**
- **Swap Chains are tightly coupled to the window system.**

After creating the Swap Chain in the first place, the process for using the Swap Chain is:

1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to “front buffer”)

What is a Swap Chain?

Because it has the word “chain” in it, let’s try to visualize the Swap Chain as a physical chain.

A bicycle chain isn’t far off. A bicycle chain goes around and around, each section of the chain taking its turn on the gear teeth, off the gear teeth, on, off, on, off, etc.

Because the Swap Chain is actually a ring buffer, the images in a Swap Chain go around and around too, each image taking its turn being drawn into, being presented, drawn into, being presented etc.

In the same way that bicycle chain links are “re-used”, Swap Chain images get re-used too.
What is a Swap Chain?

This is a pretty good analogy, except that there can be many more images in the ring buffer than are being shown here.

We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

- minImageCount = 2, maxImageCount = 8
- minImageExtent = 1024 x 1024
- maxImageExtent = 1024 x 1024
- maxImageArrayLayers = 1
- supportedTransforms = 0x0001
- currentTransform = 0x0001
- supportedCompositeAlpha = 0x0001
- supportedUsageFlags = 0x009f
- ** This Surface is supported by the Graphics Queue **

Found 2 Surface Formats:
- 0: 44 0 (VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR )
- 1: 50 0 (VK_FORMAT_B8G8R8A8_SRGB, VK_COLOR_SPACE_SRGB_NONLINEAR )

Found 3 Present Modes:
- 0: 2 (VK_PRESENT_MODE_FIFO )
- 1: 3 (VK_PRESENT_MODE_FIFO_RELAXED )
- 2: 1 (VK_PRESENT_MODE_MAILBOX )

Creating a Swap Chain
Creating a Swap Chain

```cpp
VkSurfaceCapabilities vsc;
vkGetPhysicalDeviceSurfaceCapabilities(PhysicalDevice, Surface, OUT &vsc);
VkSwapchainCreateInfo vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO;
vscci.pNext = nullptr;
vscci.flags = 0;
vscci.surface = Surface;
vscci.minImageCount = 2;  // double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.imageArrayLayers = 1;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.queueFamilyIndexCount = 0;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
result = vkCreateSwapchain(LogicalDevice, &vscci, PALLOCATOR, OUT &SwapChain);
```

Creating the Swap Chain Images and Image Views

```cpp
uint32_t imageCount;  // # of display buffers – 2?  3?
result = vkGetSwapchainImages(LogicalDevice, SwapChain, OUT &imageCount, PresentImages);
PresentImages = new VkImage[imageCount];
for(unsigned int i = 0; i < imageCount; i++) {
    VkImageCreateInfo ivci;
    ivci.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    ivci.pNext = nullptr;
    ivci.flags = 0;
    ivci.imageType = VK_IMAGE_TYPE_2D;
    ivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    ivci.extent.width = surfaceRes.width;
    ivci.extent.height = surfaceRes.height;
    ivci.mipLevels = 1;
    ivci.arrayLayers = 1;
    ivci.samples = VK_SAMPLE_COUNT_1_BIT;
    ivci.usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
    ivci.tiling = VK_IMAGE_TILING_OPTIMAL;
    ivci.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    ivci.queueFamilyIndexCount = 0;
    ivci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    VkPresentModeKHR presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
    ivci.oldSwapchain = VK_NULL_HANDLE;
    ivci.clipped = VK_TRUE;
    result = vkCreateImage(LogicalDevice, &ivci, PALLOCATOR, OUT &PresentImages[i]);
}
```

Rendering into the Swap Chain, I

```cpp
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, OUT &imageReadySemaphore);
uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
vkAcquireNextImage(LogicalDevice, SwapChain, timeout, imageReadySemaphore, VK_NULL_HANDLE, OUT &nextImageIndex);
result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], &vcbbi);
```

Rendering into the Swap Chain, II

```cpp
VkFenceCreateInfo vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = 0;
VkFence renderFence;
vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);
VkQueue presentQueue;
vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue);
VkSubmitInfo vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.waitSemaphoreCount = 1;
    vsi.pWaitSemaphores = &imageReadySemaphore;
    vsi.pWaitDstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = &SemaphoreRenderFinished;
result = vkQueueSubmit(presentQueue, 1, &vsi, renderFence);
result = vkQueueSubmit(presentQueue, 1, &vsi, renderFence);
```
result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);

VkPresentInfo
  vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO;
  vpi.pNext = nullptr;
  vpi.swapchainCount = 1;
  vpi.pSwapchains = &SwapChain;
  vpi.pImageIndices = &nextImageIndex;
  vpi.pResults = (VkResult *) nullptr;

result = vkQueuePresent(presentQueue, IN &vpi);

Rendering Scene()
{
  VkSemaphore imageReadySemaphore;
  result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
  uint32_t nextImageIndex;
  vkAcquireNextImage(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

  VkCommandBufferBeginInfo vcbbi;
  vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
  vcbbi.pNext = nullptr;
  vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
  vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
  result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);

  VkClearColorValue vccv;
  vccv.float32[0] = 0.0;
  vccv.float32[1] = 0.0;
  vccv.float32[2] = 0.0;
  vccv.float32[3] = 1.0;

  VkClearDepthStencilValue vcdsv;
  vcdsv.depth = 1.0;
  vcdsv.stencil = 0;

  VkClearValue vcv[2];
  vcv[0].color = vccv;
  vcv[1].depthStencil = vcdsv;

  VkOffset2D o2d = {0, 0};
  VkExtent2D e2d = {Width, Height};
  VkRect2D r2d = {o2d, e2d};

  VkRenderPassBeginInfo vrpbi;
  vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  vrpbi.pNext = nullptr;
  vrpbi.renderPass = RenderPass;
  vrpbi.framebuffer = Framebuffers[nextImageIndex];
  vrpbi.renderArea = r2d;
  vrpbi.clearValueCount = 2;
  vrpbi.pClearValues = vcv;

  vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);

  vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
VkViewport viewport =
{
    0., // x
    0., // y
    (float)Width, // width
    (float)Height, // height
    0., // minDepth
    1. // maxDepth
};

vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &viewport); // 0=firstViewport, 1=viewportCount

VkRect2D scissor =
{
    0,
    0,
    Width,
    Height
};

vkCmdSetScissor(CommandBuffers[nextImageIndex], 0, 1, &scissor);

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

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets); // 0, 1 = firstBinding, bindingCount

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);

VkFenceCreateInfo vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

VkFence renderFence;

vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkQueue presentQueue;

vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue); // 0 = queueIndex

VkSubmitInfo vsi;

vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

vsi.pNext = nullptr;

vsi.waitSemaphoreCount = 1;

vsi.pWaitSemaphores = &imageReadySemaphore;

vsi.pWaitDstStageMask = &waitAtBottom;

vsi.commandBufferCount = 1;

vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];

vsi.signalSemaphoreCount = 0;

vsi.pSignalSemaphores = &SemaphoreRenderFinished;

result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence); // 1 = submitCount

result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX); // waitAll, timeout

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

VkPresentInfo vpi;

vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO;

vpi.pNext = nullptr;

vpi.waitSemaphoreCount = 0;

vpi.pWaitSemaphores = NULL;

vpi.swapchainCount = 1;

vpi.pSwapchains = &SwapChain;

vpi.pImageIndices = &nextImageIndex;

vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresent(presentQueue, IN &vpi);

vkDestroySemaphore(LogicalDevice, imageReadySemaphore, PALLOCATOR);

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

vkDestroyCommandBuffer(LogicalDevice, CommandBuffers[nextImageIndex], PALLOCATOR);

vkDestroyCommandBuffer(LogicalDevice, CommandBuffers[nextImageIndex], PALLOCATOR);

vkDestroyCommandBuffer(LogicalDevice, CommandBuffers[nextImageIndex], PALLOCATOR);

vkDestroyCommandBuffer(LogicalDevice, CommandBuffers[nextImageIndex], PALLOCATOR);
In OpenGL terms: assigning an (s,t) to each vertex

Enable texture mapping:
```
glEnable( GL_TEXTURE_2D );
```

Draw your polygons, specifying s and t at each vertex:
```
gBegin( GL_POLYGON );
gTexCoord2f( s0, t0 );
gNormal3f( nx0, ny0, nz0 );
gVertex3f( x0, y0, z0 );
gTexCoord2f( s1, t1 );
gNormal3f( nx1, ny1, nz1 );
gVertex3f( x1, y1, z1 );
```

... 
```
gEnd();
```

Disable texture mapping:
```
gDisable( GL_TEXTURE_2D );
```

Using a Texture: How do you know what (s,t) to assign to each vertex?

The easiest way to figure out what s and t are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane,

\[
s = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \\
t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}}
\]

Using a Texture: How do you know what (s,t) to assign to each vertex?

Or, for a sphere,

\[
s = \frac{\theta - (-\pi)}{2\pi} \\
t = \frac{\phi - \left(\frac{-\pi}{2}\right)}{\pi}
\]

From the Sphere code:

\[
s = \left(\text{lng} + \text{M_PI}\right) / (2.*\text{M_PI}); \\
t = \left(\text{lat} + \text{M_PI}/2.\right) / \text{M_PI};
\]
Using a Texture: How do you know what (s,t) to assign to each vertex?

Uh-oh. Now what? Here's where it gets tougher…

\[
\begin{align*}
    s &= ? \\
    t &= ?
\end{align*}
\]

You really are at the mercy of whoever did the modeling...

---

**Memory Types**

**CPU Memory**
- You create your texture here

**GPU Memory**
- Host Visible GPU Memory (the "Staging Buffer")
- Device Local GPU Memory

- Texture Sampling Hardware
- RGBA to the Shader
**Memory Types**

**NVIDIA Discrete Graphics:**
- 11 Memory Types:
  - Memory 0: DeviceLocal
  - Memory 1: DeviceLocal
  - Memory 2: DeviceLocal HostVisible HostCoherent HostCached

**Intel Integrated Graphics:**
- 3 Memory Types:
  - Memory 0: DeviceLocal
  - Memory 1: DeviceLocal HostVisible HostCoherent HostCached

---

**Texture Sampling Parameters**

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

**VK SamplerCreateInfo**

```c
VkSamplerCreateInfo vsci;
vsci.magFilter = VK_FILTER_LINEAR;
vsci.minFilter = VK_FILTER_LINEAR;
vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

```c
result = vkCreateSampler(LogicalDevice, &vsci, PALLOCATOR, pTextureSampler);
```

---

**Textures’ Undersampling Artifacts**

As an object gets farther away and covers a smaller and smaller part of the screen, the texels : pixels ratio used in the coverage becomes larger and larger. This means that there are pieces of the texture leftover in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all white, with the red texel having never influenced the colors in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets filtered into in resolution-level textures.

---

**Texture Mip*-mapping**

- Total texture storage is ~ 2x what it was without mip-mapping
- Graphics hardware determines which level to use based on the texels : pixels ratio.
- In addition to just picking one mip-map level, the rendering system can sample from two of them, one less than the T:P ratio and one more, and then blend the two RGBAs returned. This is known as `VK_SAMPLER_MIPMAP_MODE_LINEAR`.

* Latin: multum in parvo, “many things in a small place”
VkResult InitTextureSampler(MyTexture * pMyTexture) {
    return InitTextureBuffer(INOUT MyTexture * pMyTexture);
}

uint32_t texWidth = pMyTexture->width;
uint32_t texHeight = pMyTexture->height;
unsigned char *texture = pMyTexture->pixels;
VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each

VkSamplerCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
#ifdef CHOICES
    // ************************************************************ *******************
    VK_SAMPLER_ADDRESS_MODE_REPEAT
    // this first {...} is to create the staging image:
    // ************************************************************ *******************
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE#endif

    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.anisotropyEnable = VK_FALSE;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_LINEAR;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled
    vkGetImageMemoryRequirements(LogicalDevice, IN stagingImage, OUT &vmr);
    void * gpuMemory;
    vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
    if (vsl.rowPitch == 4 * texWidth)
        fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
    unsigned char *gpuBytes = (unsigned char *)gpuMemory;
    for (unsigned int y = 0; y < texHeight; y++)
        VkMemoryAllocateInfo vmai;
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        vmai.pNext = nullptr;
        memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
        vmai.allocationSize = vmr.size;
        vmai.memoryTypeIndex = FindMemoryThatIsHostVisible(); // because we want to mmap it
        result = vkAllocateMemory(LogicalDevice, OUT &pMyTexture->vdm, vdm, vmai);
        vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
        // we have now created the staging image -- fill it with the pixel data:
        VkSubresourceLayout vsl;
        vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);
        if (Verbose)
            fprintf(FpDebug, "Subresource Layout:\n");
            fprintf(FpDebug, "	\toffset = %lld\n", vsl.offset);
            fprintf(FpDebug, "	\tsize = %lld\n", vsl.size);
            fprintf(FpDebug, "	\trowPitch = %lld\n", vsl.rowPitch);
            fprintf(FpDebug, "	\tarrayPitch = %lld\n", vsl.arrayPitch);
            fflush(FpDebug);

    VkImage stagingImage;
    VkImageLayout stagingLayout;
    return result;
}

void *gpuMemory
#endif CHOICES
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_PREINITIALIZED
copy pixels from the staging image to the texture:

// *******************************************************************************
// copy pixels from the staging image to the texture:
// create the actual texture image:
// *******************************************************************************

VkCommandBufferBeginInfo vcbbi;
{ vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO; vcbbi.pNext = nullptr; vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT; vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr; }

VkImageCreateInfo vici;
{ vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO; vici.pNext = nullptr; vici.flags = 0; vici.imageType = VK_IMAGE_TYPE_2D; vici.format = VK_FORMAT_R8G8B8A8_UNORM; vici.extent.width = texWidth; vici.extent.height = texHeight; vici.extent.depth = 1; vici.mipLevels = 1; vici.arrayLayers = 1; vici.samples = VK_SAMPLE_COUNT_1_BIT; vici.tiling = VK_IMAGE_TILING_OPTIMAL; vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT; }

result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled

VkImageMemoryBarrier vimb;
{ vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER; vimb.pNext = nullptr; vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED; vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL; vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED; vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED; vimb.image = textureImage; vimb.srcAccessMask = 0; vimb.dstAccessMask = 0; vimb.subresourceRange = visr; }

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

VkImageCopy vic;
{ vic.srcSubresource = visl; vic.srcOffset = vo3; vic.dstSubresource = visl; vic.dstOffset = vo3; vic.extent = ve3; }

vkCmdCopyImage(TextureCommandBuffer, stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);

// now do the final image transfer:

VkImageCopy vic;
{ vic.srcSubresource = visl; vic.srcOffset = vo3; vic.dstSubresource = visl; vic.dstOffset = vo3; vic.extent = ve3; }

vkCmdCopyImage(TextureCommandBuffer, stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, textureImage, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, IN &vic);
Reading in a Texture from a BMP File

```cpp
// create an image view for the texture image:
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

VkImageViewCreateInfo vivci;
vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
vivci.pNext = nullptr;
vivci.flags = 0;
vivci.image = textureImage;
vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
vivci.components.r = VK_COMPONENT_SWIZZLE_R;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.subresourceRange = visr;
result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
return result;
```

This function can be found in the `sample.cpp` file. The BMP file needs to be created by something that writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick’s `convert`, Adobe Photoshop, or GNU’s GIMP.
**Vulkan: More Typical (and Simplified) Block Diagram**

```
Application
    ↓
Instance
    ↓
Physical Device
    ↓
Logical Device
    ↓
Command Buffer
```

**Vulkan: Identifying the Physical Devices**

```c
VkResult result = VK_SUCCESS;
result = vkEnumeratePhysicalDevices( Instance, OUT &physicalDeviceCount, nullptr);  
if( result != VK_SUCCESS )  
{  
    fprintf( FpDebug, "Could not count the physical devices
    return VK_SHOULD_EXIT;
    }
    fprintf(FpDebug, "

Device %2d:
    fprintf( FpDebug, "API version: %d
    fprintf( FpDebug, "Driver version: %d
    fprintf( FpDebug, "Vendor ID: 0x%04x
    fprintf( FpDebug, "Device ID: 0x%04x
    fprintf( FpDebug, "Physical Device Type: %d (";  
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )  
        fprintf( FpDebug, "Discrete GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )  
        fprintf( FpDebug, "Integrated GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )  
        fprintf( FpDebug, "Virtual GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )  
        fprintf( FpDebug, "CPU"
    
    fprintf( FpDebug, "Device Name: %s"
    fprintf( FpDebug, "Pipeline Cache Size: %d
    return VK_SHOULD_EXIT;

Vulkan: Querying the Number of Physical Devices**

```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );
```

This way of querying information is a recurring OpenCL and Vulkan pattern (get used to it):

```
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
```

**Which Physical Device to Use, I**

```c
int discreteSelect = -1;  
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )  
{  
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties( IN physicalDevice[i], OUT &vpdp );
    if( result != VK_SUCCESS )  
    {  
        fprintf( FpDebug, "Could not get the physical device properties of device %d
        return VK_SHOULD_EXIT;
    }
    fprintf( FpDebug, "

Device %2d:
    fprintf( FpDebug, "API version: %d
    fprintf( FpDebug, "Driver version: %d
    fprintf( FpDebug, "Vendor ID: 0x%04x
    fprintf( FpDebug, "Device ID: 0x%04x
    fprintf( FpDebug, "Physical Device Type: %d (";
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )  
        fprintf( FpDebug, "Discrete GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )  
        fprintf( FpDebug, "Integrated GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )  
        fprintf( FpDebug, "Virtual GPU"
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )  
        fprintf( FpDebug, "CPU"
    
    fprintf( FpDebug, "Device Name: %s"
    fprintf( FpDebug, "Pipeline Cache Size: %d
    return VK_SHOULD_EXIT;
```
Asking About the Physical Device’s Features

```c
VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
printf( FpDebug, "Physical Device Features:
" );
printf( FpDebug, "geometryShader = %2d
", PhysicalDeviceFeatures.geometryShader );
printf( FpDebug, "tessellationShader = %2d
", PhysicalDeviceFeatures.tessellationShader );
printf( FpDebug, "multiDrawIndirect = %2d
", PhysicalDeviceFeatures.multiDrawIndirect );
printf( FpDebug, "wideLines = %2d
", PhysicalDeviceFeatures.wideLines );
printf( FpDebug, "largePoints = %2d
", PhysicalDeviceFeatures.largePoints );
printf( FpDebug, "multiViewport = %2d
", PhysicalDeviceFeatures.multiViewport );
printf( FpDebug, "occlusionQueryPrecise = %2d
", PhysicalDeviceFeatures.occlusionQueryPrecise );
printf( FpDebug, "pipelineStatisticsQuery = %2d
", PhysicalDeviceFeatures.pipelineStatisticsQuery );
printf( FpDebug, "shaderFloat64 = %2d
", PhysicalDeviceFeatures.shaderFloat64 );
printf( FpDebug, "shaderInt64 = %2d
", PhysicalDeviceFeatures.shaderInt64 );
printf( FpDebug, "shaderInt16 = %2d
", PhysicalDeviceFeatures.shaderInt16 );
```

Here’s What the NVIDIA 1080Ti Produced

```c
vkEnumeratePhysicalDevices:
Device 0:  
API version: 4194360  
Driver version: 4194360  
Vendor ID: 0x10de  
Device ID: 0x1b06  
Physical Device Type: 2 = (Discrete GPU)  
Device Name: GeForce GTX 1080 Ti  
Pipeline Cache Size: 13  
Device #0 selected (GeForce GTX 1080 Ti)  
Physical Device Features:
geometryShader = 1  
tessellationShader = 1  
multiDrawIndirect = 1  
wideLines = 1  
largePoints = 1  
multiViewport = 1  
occlusionQueryPrecise = 1  
pipelineStatisticsQuery = 1  
shaderFloat64 = 1  
shaderInt64 = 1  
shaderInt16 = 0
```

Which Physical Device to Use, II

```c
vkEnumeratePhysicalDevices:
Device 0:
API version: 4194360
Driver version: 4194360
Vendor ID: 0x8086
Device ID: 0x1916
Physical Device Type: 1 = (Integrated GPU)
Device Name: Intel(R) HD Graphics 520
Pipeline Cache Size: 213
Device #0 selected (Intel(R) HD Graphics 520)
Physical Device Features:
geometryShader = 1
tessellationShader = 1
multiDrawIndirect = 1
wideLines = 1
largePoints = 1
multiViewport = 1
occlusionQueryPrecise = 1
pipelineStatisticsQuery = 1
shaderFloat64 = 1
shaderInt64 = 1
shaderInt16 = 1
```

// need some logical here to decide which physical device to select:
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
  discreteSelect = i;
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
  integratedSelect = i;
else
  which = -1;
  if( discreteSelect >= 0 )
    which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
  else if( integratedSelect >= 0 )
    which = integratedSelect;
    PhysicalDevice = physicalDevices[which];
  else
    fprintf( FpDebug, "Could not select a Physical Device
" );
    return VK_SHOULD_EXIT;
```
Asking About the Physical Device's Different Memories

VkPhysicalDeviceMemoryProperties vpdm;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdm );
fprintf( FpDebug, "%d Memory Types:
", vpdmp.memoryTypeCount );
for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
{
  VkMemoryType vmt = vpdmp.memoryTypes[i];
  fprintf( FpDebug, "Memory %2d: ", i );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT       ) != 0 )    fprintf( FpDebug, " DeviceLocal" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT       ) != 0 )    fprintf( FpDebug, " HostVisible" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT      ) != 0 )    fprintf( FpDebug, " HostCoherent" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT        ) != 0 )    fprintf( FpDebug, " HostCached" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT   ) != 0 )    fprintf( FpDebug, " LazilyAllocated" );
  fprintf(FpDebug, "
" );
}

Asking About the Physical Device's Queue Families

uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
fprintf( FpDebug, "Found %d Queue Families:
", count );
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceQueueFamilyProperties( PhysicalDevice, &count, OUT vqfp );
for( unsigned int i = 0; i < count; i++ )
{
  fprintf( FpDebug, "%d: queueCount = %2d  ;   ", i, vqfp[i].queueCount );
  if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 ) fprintf( FpDebug, " Graphics" );
  if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT  ) != 0 ) fprintf( FpDebug, " Compute " );
  if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 ) fprintf( FpDebug, " Transfer" );
  fprintf(FpDebug, "
" );
}

Here's What I Got

11 Memory Types:
Memory 0: Memory 1: Memory 2: Memory 3: Memory 4: Memory 5: Memory 6: DeviceLocal Memory 8: DeviceLocal Memory 9: HostVisible HostCoherent HostCached

2 Memory Heaps:
Heap 0: size = 0xb7c00000 DeviceLocal Heap 1: size = 0xfac00000 HostVisible HostCoherent HostCached

Here's What I Got

Found 3 Queue Families:
0: queueCount = 16  ;    Graphics Compute Transfer
1: queueCount =  1  ;    Transfer
2: queueCount =  8  ;    Compute
Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

Vulkan: a More Typical (and Simplified) Block Diagram

Looking to See What Device Layers are Available

```c
const char * myDeviceLayers[] =
{ 
  "VK_LAYER_LUNARG_api_dump", 
  "VK_LAYER_LUNARG_core_validation", 
  "VK_LAYER_LUNARG_image", 
  "VK_LAYER_LUNARG_object_tracker", 
  "VK_LAYER_LUNARG_parameter_validation", 
  "VK_LAYER_NV_optimus", 
};
```

Looking to See What Device Extensions are Available

```c
// see what device extensions are available:
uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
```
What Device Layers and Extensions are Available

3 physical device layers enumerated:

- 0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
- 0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
- 0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'

Vulkan: Specifying a Logical Device Queue

```c
float queuePriorities[1] = { 1.0f; };

VkDeviceQueueCreateInfo vdqci;
vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci.pNext = nullptr;
vdqci.flags = 0;
vdqci.queueFamilyIndex = 0;
vdqci.queueCount = 1;
vdqci.pQueueProperties = queuePriorities;
```

Vulkan: Creating a Logical Device

```c
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1; // # of device queues
vdci.pQueueCreateInfos = &vdqci; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.enabledLayerCount = 0;
vdci.ppEnabledLayerNames = myDeviceLayers;
vk.device.enabledExtensionCount = 0;
vk.device.enabledExtensionNames = nullptr; // no extensions
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vk.device.enabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = &PhysicalDeviceFeatures;

result = vkCreateLogicalDevice( PhysicalDevice, &vdci, PALLOCATOR, OUT &LogicalDevice );
```

Vulkan: Creating the Logical Device’s Queue

```c
// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0, &Queue ); // 0, 0 = queueFamilyIndex, queueIndex
```
Layers and Extensions

Mike Bailey
mj@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

Vulkan Layers

Application
Layer A
Layer B
Layer C
Vulkan

Layers are code that can be installed between the Application and Vulkan. Normally, Vulkan is meant to run "flat out". Layers can take the extra time to perform useful functions like printing debugging messages, printing function calls, etc.

They are not always necessary, but when you need them, you will be really glad they are there!

Looking to See What Instance Layers and Instance Extensions are Available

```
const char * instanceLayers[] =
{
  "VK_LAYER_LUNARG_api_dump",  // turn this on if want to see each function call and its arguments (very slow!)
  "VK_LAYER_LUNARG_core_validation",
  "VK_LAYER_LUNARG_object_tracker",
  "VK_LAYER_LUNARG_parameter_validation",
  "VK_LAYER_LUNARG_screenshot",
  "VK_LAYER_LUNARG_standard_validation",
  "VK_LAYER_GOOGLE_threading",
  "VK_LAYER_GOOGLE_unique_objects",
  "VK_LAYER_LUNARG_vktrace",
  "VK_LAYER_NV_optimus",
};

const char * instanceExtensions[] =
{
  "VK_surface",
#ifdef _WIN32
  "VK_win32_surface",
#endif
  "VK_EXT_debug_report",
};

uint32_t numExtensionsWanted = sizeof(instanceExtensions) / sizeof(char *);

result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, InstanceExtensions );
```

Looking to See What InstanceLayers and InstanceExtensions are Available

13 instance layers available:

- 0x00400033   2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
- 0x00400033   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
- 0x00400033   1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
- 0x00400033   1  'VK_LAYER_GOOGLE_threading'  'Google Validation Layer'
- 0x00400033   1  'VK_LAYER_GOOGLE_unique_objects'  'Google Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_vktrace'  'Vktrace tracing library'
- 0x00400033   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
- 0x00400033   1  'VK_LAYER_NV_nsid'  'NVIDIA Nsight interception layer'
- 0x00400033  34  'VK_LAYER_RENDERDOC_Capture'  'Debugging capture layer for RenderDoc'

Looking to See What InstanceLayers and InstanceExtensions are Available

13 instance layers available:

- 0x00400033   2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
- 0x00400033   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
- 0x00400033   1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
- 0x00400033   1  'VK_LAYER_GOOGLE_threading'  'Google Validation Layer'
- 0x00400033   1  'VK_LAYER_GOOGLE_unique_objects'  'Google Validation Layer'
- 0x00400033   1  'VK_LAYER_LUNARG_vktrace'  'Vktrace tracing library'
- 0x00400033   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
- 0x00400033   1  'VK_LAYER_NV_nsid'  'NVIDIA Nsight interception layer'
- 0x00400033  34  'VK_LAYER_RENDERDOC_Capture'  'Debugging capture layer for RenderDoc'
vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_EXT_display_surface_counter'
0x00000001  'VK_get_physical_device_properties2'
0x00000019  'VK_surface'
0x00000006  'VK_win32_surface'
0x00000001  'VK_KHR_device_group_creation'
0x00000001  'VK_external_fence_capabilities'
0x00000001  'VK_external_memory_capabilities'
0x00000001  'VK_NV_external_memory_capabilities'

vkEnumeratePhysicalDevices:
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );
int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ ){
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        discreteSelect = i;
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        integratedSelect = i;
}
int which = -1;
if( discreteSelect >= 0 )
    which = discreteSelect;
else if( integratedSelect >= 0 )
    which = integratedSelect;
else
    fprintf( FpDebug, "Could not select a Physical Device
" );
return VK_SHOULD_EXIT;
delete[] physicalDevices;
vkGetPhysicalDeviceProperties(PhysicalDevice, OUT &PhysicalDeviceProperties);
vkGetPhysicalDeviceFeatures(IN PhysicalDevice, OUT &PhysicalDeviceFeatures);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp);
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(PhysicalDevice, OUT &vpdmp);
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT vqfp);
delete[] vqfp;

VkResult result;
float queuePriorities[NUM_QUEUES_WANTED] = {
    1.0f,
};
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];
vdqci[0].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1;                // how many queues to create
vdqci[0].pQueuePriorities = queuePriorities;    // array of queue priorities [0...1]

const char *myDeviceLayers[] = {
    ("VK_LAYER_LUNARG_api_dump"),
    ("VK_LAYER_LUNARG_core_validation"),
    ("VK_LAYER_LUNARG_image"),
    ("VK_LAYER_LUNARG_object_tracker"),
    ("VK_LAYER_LUNARG_parameter_validation"),
    ("VK_LAYER_NV_optimus"),
};

const char *myDeviceExtensions[] = {
    ("VK_swapchain"),
};

uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties *deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);
for (unsigned int i = 0; i < layerCount; i++)
{
    // see what device extensions are available:
    uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
        (VkExtensionProperties *)nullptr);
    VkExtensionProperties *deviceExtensions = new VkExtensionProperties[extensionCount];
    result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
        deviceExtensions);
    for (uint32_t j = 0; j < extensionCount; j++)
    {
        // do something with device extensions
    }
    delete[] deviceExtensions;
    delete[] deviceLayers;
}

4 physical device layers enumerated:
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033 1 'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
vkEnumerateDeviceLayerProperties:
3 physical device layers enumerated:
0x00400038  1  ‘VK_LAYER_NV_optimus’  ‘NVIDIA Optimus layer’
0 device extensions enumerated for ‘VK_LAYER_NV_optimus’;
0x00400033  1  ‘VK_LAYER_LUNARG_object_tracker’  ‘LunarG Validation Layer’
0 device extensions enumerated for ‘VK_LAYER_LUNARG_object_tracker’;
0x00400033  1  ‘VK_LAYER_LUNARG_parameter_validation’  ‘LunarG Validation Layer’
0 device extensions enumerated for ‘VK_LAYER_LUNARG_parameter_validation’;

Vulkan Highlights: Overall Block Diagram

Application
Instance
Physical Device
Logical Device
Logical Device
Fence
Host
Semaphore
Command Buffer
Event Buffer

Semaphores
• Used to control readiness of resources within one queue or across different queues belonging to the same logical device
• You create them, and give them to a Vulkan function which sets them. Later on, you tell a Vulkan function to wait on this particular semaphore
• You don’t end up setting, resetting, or checking the semaphore yourself
• Semaphores must be initialized (“created”) before they can be used

Ask for Something
Try to Use the Something
Semaphore

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan
Creating a Semaphore

```
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

VkSemaphore semaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &semaphore );
```

Fences

- Used to synchronize the application with commands submitted to a queue
- Announces that queue-submitted work is finished
- Much finer control than semaphores
- You can un-signal, signal, test or block-while-waiting

```
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

VkFenceCreateInfo vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT;  // = 0

// VK_FENCE_CREATE_SIGNALED_BIT is only other option

VkFence fence;
result = vkCreateFence( LogicalDevice, IN &vfci, PALLOCATOR, OUT &fence );

// returns right away:
result = vkGetFenceStatus( LogicalDevice, IN fence );
    // result = VK_SUCCESS means it has signaled
    // result = VK_NOT_READY means it has not signaled

// blocks:
result = vkWaitForFences( LogicalDevice, IN fence, waitForAll, timeout );
    // waitForAll = VK_TRUE:   wait for all fences in the list
    // waitForAll = VK_FALSE: wait for any one fence in the list
    // timeout is a uint64_t timeout in nanoseconds  (could be 0, which means to return immediately)
    // timeout can be up to UINT64_MAX = 0xffffffffffffffff (= 580+ years)
    // result = VK_SUCCESS means it returned because a fence (or all fences) signaled
    // result = VK_TIMEOUT means it returned because the timeout was exceeded
```
Fence Example

```
VkFence renderFence;
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue);

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore) nullptr;

result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence );
...
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );
...
result = vkQueuePresent( presentQueue, IN &vpi );
```

Events

- Events provide even finer-grained synchronization
- Events are a primitive that can be signaled by the host or the device
- Can even signal at one place in the pipeline and wait for it at another place in the pipeline
- Signaling in the pipeline means "signal as the last piece of this draw command passes that point in the pipeline."
- You can signal, un-signal, or test from a vk function or from a vkCmd function
- Can wait from a vkCmd function

Controlling Events from the Host

```
VkEventCreateInfo veci;
veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;
veci.pNext = nullptr;
veci.flags = 0;

VkEvent event;
result = vkCreateEvent( LogicalDevice, IN &veci, PALLOCATOR, OUT &event );
result = vkSetEvent( LogicalDevice, IN event );
result = vkResetEvent( LogicalDevice, IN event );
result = vkGetEventStatus( LogicalDevice, IN event );
  // result = VK_EVENT_SET: signaled
  // result = VK_EVENT_RESET: not signaled
```

Note: the CPU cannot block waiting for an event, but it can test for one

Controlling Events from the Device

```
result = vkCmdSetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdResetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdWaitEvents( CommandBuffer, 1, &event, srcPipelineStageBits, dstPipelineStageBits, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
```

Could be an array of events

Where signaled, where wait for the signal

Memory barriers get executed after events have been signaled

Note: the GPU cannot test for an event, but it can block waiting for one
**Vulkan**

Pipeline Barriers: A case of Gate-ing and Wait-ing

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

---

From the Command Buffer Notes:

**These are the Commands that can be entered into the Command Buffer, I**

- `vkCmdBeginQuery(commandBuffer, flags);`
- `vkCmdBeginRenderPass(commandBuffer, const contents);`
- `vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);`
- `vkCmdBindIndexBuffer(commandBuffer, indexType);`
- `vkCmdBindPipeline(commandBuffer, pipeline);`
- `vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);`
- `vkCmdBlitImage(commandBuffer, filter);`
- `vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);`
- `vkCmdClearColorImage(commandBuffer, pRanges);`
- `vkCmdClearDepthStencilImage(commandBuffer, pRanges);`
- `vkCmdCopyBuffer(commandBuffer, pRegions);`
- `vkCmdCopyBufferToImage(commandBuffer, pRegions);`
- `vkCmdCopyImage(commandBuffer, pRegions);`
- `vkCmdCopyImageToBuffer(commandBuffer, pRegions);`
- `vkCmdCopyQueryPoolResults(commandBuffer, flags);`
- `vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDebugMarkerEndEXT(commandBuffer);`
- `vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);`
- `vkCmdDispatchIndirect(commandBuffer, offset);`
- `vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);`
- `vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);`
- `vkCmdDrawIndexedIndirect(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);`
- `vkCmdDrawIndirect(commandBuffer, stride);`
- `vkCmdDrawIndirectCountAMD(commandBuffer, stride);`
- `vkCmdEndQuery(commandBuffer, query);`
- `vkCmdEndRenderPass(commandBuffer);`
- `vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);`
- `vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);`
- `vkCmdNextSubpass(commandBuffer, contents);`
- `vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);`
- `vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);`
- `vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, size, pValues);`
- `vkCmdPushDescriptorSet(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);`
- `vkCmdPushDescriptorSetWithTemplate(commandBuffer, descriptorUpdateTemplate, layout, set, pData);`
- `vkCmdResolveImage(commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions);`
- `vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);`
- `vkCmdSetDeviceMaskKHX(commandBuffer, deviceMask);`
- `vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);`
- `vkCmdSetEvent(commandBuffer, event, stageMask);`
- `vkCmdSetLineWidth(commandBuffer, lineWidth);`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference);`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);`
- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);`
- `vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);`
- `vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);`
- `vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query);`

**Potential Memory Race Conditions that Pipeline Barriers can Prevent**

1. Write-then-Read (WtR) – the memory write in one operation starts overwriting the memory that another operation’s read needs to use
2. Read-then-Write (RtW) – the memory read in one operation hasn’t yet finished before another operation starts overwriting that memory
3. Write-then-Write (WtW) – two operations start overwriting the same memory and the end result is non-deterministic

Note: there is no problem with Read-then-Read (RtR) as no data has been changed

---

We don’t any one of these commands to have to wait on a previous command unless you say so. In general, we want all of these commands to be able to run “flat-out”.

But, if we do that, surely there will be nasty race conditions!

---

From the Command Buffer Notes:

**These are the Commands that can be entered into the Command Buffer, II**

- `vkCmdBeginQuery(commandBuffer, flags);`
- `vkCmdBeginRenderPass(commandBuffer, const contents);`
- `vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);`
- `vkCmdBindIndexBuffer(commandBuffer, indexType);`
- `vkCmdBindPipeline(commandBuffer, pipeline);`
- `vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);`
- `vkCmdBlitImage(commandBuffer, filter);`
- `vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);`
- `vkCmdClearColorImage(commandBuffer, pRanges);`
- `vkCmdClearDepthStencilImage(commandBuffer, pRanges);`
- `vkCmdCopyBuffer(commandBuffer, pRegions);`
- `vkCmdCopyBufferToImage(commandBuffer, pRegions);`
- `vkCmdCopyImage(commandBuffer, pRegions);`
- `vkCmdCopyImageToBuffer(commandBuffer, pRegions);`
- `vkCmdCopyQueryPoolResults(commandBuffer, flags);`
- `vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDebugMarkerEndEXT(commandBuffer);`
- `vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);`
- `vkCmdDispatchIndirect(commandBuffer, offset);`
- `vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);`
- `vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);`
- `vkCmdDrawIndexedIndirect(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);`
- `vkCmdDrawIndirect(commandBuffer, stride);`
- `vkCmdDrawIndirectCountAMD(commandBuffer, stride);`
- `vkCmdEndQuery(commandBuffer, query);`
- `vkCmdEndRenderPass(commandBuffer);`
- `vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);`
- `vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);`
- `vkCmdNextSubpass(commandBuffer, contents);`
- `vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);`
- `vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);`
- `vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, size, pValues);`
- `vkCmdPushDescriptorSet(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);`
- `vkCmdPushDescriptorSetWithTemplate(commandBuffer, descriptorUpdateTemplate, layout, set, pData);`
- `vkCmdResolveImage(commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions);`
- `vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);`
- `vkCmdSetDeviceMaskKHX(commandBuffer, deviceMask);`
- `vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);`
- `vkCmdSetEvent(commandBuffer, event, stageMask);`
- `vkCmdSetLineWidth(commandBuffer, lineWidth);`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference);`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);`
- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);`
- `vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);`
- `vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);`
- `vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query);`

We don’t any one of these commands to have to wait on a previous command unless you say so. In general, we want all of these commands to be able to run “flat-out”.

But, if we do that, surely there will be nasty race conditions!
vkCmdPipelineBarrier( ) Function Call

vkCmdPipelineBarrier( commandBuffer, 
srcStageMask, Guarantee that this pipeline stage has completely generated one set of data before

memoryBarrierCount, pMemoryBarriers,  
bufferMemoryBarrierCount, pBufferMemoryBarriers,  
imageMemoryBarrierCount, pImageMemoryBarriers

dstStageMask, allowing this pipeline stage to consume it

VK_DEPENDENCY_BY_REGION_BIT,  
);

A Pipeline Barrier is a way to establish a memory dependency between commands that were submitted before the barrier and commands that are submitted after the barrier.

The Scenario

The Scenario Rules

1. The cross-streets are named after pipeline stages
2. All traffic lights start out green (“we want all of these commands to be able to run flat-out”)
3. There are special sensors at all intersections that will know when the first car in the src group enters that intersection
4. There are connections from those sensors to the traffic lights so that when the first car in the src group enters its intersection, the dst traffic light will be turned red
5. When the last car in the src group completely makes it through its intersection, the dst traffic light can be turned back to green
6. The Vulkan command pipeline ordering is this: (1) the src cars get released, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released (which end up being stopped by a red light somewhere), (4) the src cars clear their intersection, (5) the dst cars get released

Pipeline Stage Masks –
Where in the Pipeline is this Memory Data being Generated or Consumed?

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
Pipeline Stages and what Access Operations can Happen There

Access Operations and what Pipeline Stages they can be used In

Access Masks – What are you Interested in Generating or Consuming this Memory for?

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_MEMORY_WRITE_BIT
Example: Be sure we are done writing an output image before using it for something else

Stages
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Example: Don’t read a buffer back to the host until a shader is done writing it

Stages
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Access types
- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_MEMORY_WRITE_BIT

The Scenario
src cars are generating the image
dst cars are doing something with that image
 VkImageLayout – How an Image gets Laid Out in Memory depends on how it will be Used

VkImageMemoryBarrier

- srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
- dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
- image = ??;
- subresourceRange = visr;
- srcAccessMask = ??;
- dstAccessMask = ??;
- oldLayout = ??;
- newLayout = ??;

Push Constants

In an effort to expand flexibility and retain efficiency, Vulkan provides something called Push Constants. Like the name implies, these let you “push” constant values out to the shaders. These are typically used for small, frequently-updated data values. This is good, since Vulkan, at times, makes it cumbersome to send changes to the graphics.

By “small”, Vulkan specifies that these must be at least 128 bytes in size, although they can be larger. For example, the maximum size is 256 bytes on the NVIDIA 1080ti. (You can query this limit by looking at the maxPushConstantSize parameter in the VkPhysicalDeviceLimits structure.) Unlike uniform buffers and vertex buffers, these are not backed by memory. They are actually part of the Vulkan pipeline.

On the shader side, if, for example, you are sending a 4x4 matrix, the use of push constants in the shader looks like this:

```c
layout( push_constant ) uniform matrix{
  mat4 modelMatrix;
} Matrix;
```

On the application side, push constants are pushed at the shaders by binding them to the Vulkan Command Buffer:

```c
vkCmdPushConstants( CommandBuffer, PipelineLayout, stageFlags, offset, size, pValues );
```

where:

- `stageFlags` are or’ed bits of VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, etc.
- `size` is in bytes
- `pValues` is a void * pointer to the data, which in this 4x4 matrix example, would be of type glm::mat4.
Setting up the Push Constants for the Pipeline Structure

Prior to that, however, the pipeline layout needs to be told about the Push Constants:

```c
VkPushConstantRange
vpcr[1];

vpcr[0].stageFlags =
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

vpcr[0].size = sizeof(glm::mat4);

VkPipelineLayoutCreateInfo
vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

result = vkCreatePipelineLayout(

LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
```

Creating a Pipeline

An Robotic Example using Push Constants

A robotic animation (i.e., a hierarchical transformation system)

Where each arm is represented by:

```c
struct arm
{
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale; // scale factor in x
};
```

Forward Kinematics:

You Start with Separate Pieces, all Defined in their Own Local Coordinate System
Forward Kinematics:
Hook the Pieces Together, Change Parameters, and Things Move
(All Young Children Understand This)

1. Rotate by $\Theta_1$
2. Translate by $T_{1/G}$

$$\begin{bmatrix} M_{1/G} \end{bmatrix} = \begin{bmatrix} T_{1/G} \end{bmatrix} \times \begin{bmatrix} R_{\Theta_1} \end{bmatrix}$$

Say it

Positioning Part #1 With Respect to Ground

Forward Kinematics:
Given the Lengths and Angles, Where do the Pieces Move To?

1. Rotate by $\Theta_2$
2. Translate the length of part 1
3. Rotate by $\Theta_1$
4. Translate by $T_{1/G}$

$$\begin{bmatrix} M_{2/G} \end{bmatrix} = \begin{bmatrix} T_{1/G} \end{bmatrix} \times \begin{bmatrix} R_{\Theta_1} \end{bmatrix} \times \begin{bmatrix} T_{2/1} \end{bmatrix} \times \begin{bmatrix} R_{\Theta_2} \end{bmatrix}$$

$$\begin{bmatrix} M_{2/G} \end{bmatrix} = \begin{bmatrix} M_{1/G} \end{bmatrix} \times \begin{bmatrix} M_{2/1} \end{bmatrix}$$

Say it

Positioning Part #2 With Respect to Ground
Positioning Part #3 With Respect to Ground

1. Rotate by $\Theta_3$
2. Translate the length of part 2
3. Rotate by $\Theta_2$
4. Translate the length of part 1
5. Rotate by $\Theta_1$
6. Translate by $T_{1/G}$

\[
[M_{3/G}] = [T_{1/G}] * [R_{\Theta_1}] * [T_{2/1}] * [R_{\Theta_2}] * [T_{3/2}] * [R_{\Theta_3}]
\]

In the Reset Function

```
struct arm
struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
... 
Arm1.armMatrix = glm::mat4(1.0);
Arm1.armColor  = glm::vec3(0.0, 0.0, 1.0);
Arm1.armScale  = 6.0;
Arm2.armMatrix = glm::mat4(1.0);
Arm2.armColor  = glm::vec3(0.0, 0.0, 1.0);
Arm2.armScale  = 4.0;
Arm3.armMatrix = glm::mat4(1.0);
Arm3.armColor  = glm::vec3(0.0, 0.0, 1.0);
Arm3.armScale  = 2.0;
```

The constructor `glm::mat4(1.0)` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`.

Setup the Push Constant for the Pipeline Structure

```
VkPushConstantRange vpcr[1];
vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vpcr[0].offset = 0;
vpcr[0].size = sizeof(struct arm);

VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = vpcr;

result = vkCreatePipelineLayout(LogicalDevice, &vplci, PALLOCATOR, &GraphicsPipelineLayout);
```

In the UpdateScene Function

```
float rot1 = (float)Time;
float rot2 = 2.0f * rot1;
float rot3 = 2.0f * rot2;

glm::vec3 zaxis = glm::vec3(0., 0., 1.);
glm::mat4 m1g = glm::mat4(1.0);
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);

glm::mat4 m21 = glm::mat4(1.0);
m21 = glm::translate(m21, glm::vec3(2.0*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));

glm::mat4 m32 = glm::mat4(1.0);
m32 = glm::translate(m32, glm::vec3(2.0*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));

Arm1.armMatrix = m1g;  // m1g
Arm2.armMatrix = m1g * m21;  // m2g
Arm3.armMatrix = m1g * m21 * m32;  // m3g
```
In the `RenderScene` Function Without Pipeline Barriers

```
VkBuffer buffers[1] = { MyVertexBuffer.buffer };  
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );
```

```
vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1 );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

```
vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2 );
kCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

```
vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3 );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

But, the problem is that
1. The `vkCmdDraws` must not start until the `vkCmdPushConstants` are done, and
2. The `vkCmdPushConstants` must not start until the `vkCmdDraws` are done

This is the type of problem that Pipeline Barriers were meant to solve.

Setting Up Global Memory Pipeline Barriers

```
VkMemoryBarrier vm;
vm.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
vm.pNext = nullptr;
vm.srcAccessMask = vm.dstAccessMask =
```

```
vCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN &vm, 
0, nullptr, 0, nullptr );
```

Setting Up Buffer Memory Pipeline Barriers

```
VkBufferMemoryBarrier vb;
vb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vb.pNext = nullptr;
vb.srcAccessMask = vb.dstAccessMask =
vb.srcQueueFamilyIndex = vb.dstQueueFamilyIndex =
vb.buffer = vb.offset = 
vb.size =
```

```
vCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, 0, nullptr );
```

Setting Up Image Memory Pipeline Barriers

```
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.srcAccessMask = vimb.dstAccessMask =
vimb.srcQueueFamilyIndex = vimb.dstQueueFamilyIndex =
vimb.image = vimb.subResourceRange =
```

```
vCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN &vimb, 
0, nullptr );
```

```
vCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 
0, NULL, 
0, nullptr );
```
In the **RenderScene** Function

```c
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer }; 
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

evCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

evCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1 );

vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

evCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2 );

vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

evCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3 );

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

### In the Vertex Shader

```c
layout( push_constant ) uniform arm
{
  mat4 armMatrix;
  vec3 armColor;
  float armScale;  // scale factor in x
} RobotArm;

layout( location = 0 ) in vec3 aVertex;
... 

vec3 bVertex = aVertex;  // arm coordinate system is [-1., 1.] in X
bVertex.x += 1.;  // now is [0., 2.]
bVertex.x *= RobotArm.armScale;  // now is [0., RobotArm.armScale]

bVertex = vec3( RobotArm.armMatrix * vec4( bVertex, 1. ) );
...

gl_Position = PVM * vec4( bVertex, 1. );  // Projection * Viewing * Modeling matrices
```

---

**Antialiasing and Multisampling**

Mike Bailey
mjba@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan

---

*Oregon State University Computer Graphics*
Anti-aliasing by Multisampling

MultiSampling

Multisampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there. There are two approaches:

1. **Supersampling**: Pick some number of unique sub-pixels within a pixel, render the image at each of these individual sub-pixels (including depth and stencil tests), then average them together. This results in lots of renders.

2. **Multisampling**: Perform a single color render for the one entire pixel. Then, pick some number of unique sub-pixels within that pixel and perform depth and stencil tests there. Assign the single color to all the sub-pixels that made it through the depth and stencil tests.

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.
Consider Two Triangles Whose Edges Pass Through the Same Pixel

\[
\text{Final Pixel Color} = \frac{\sum \text{Color sample from subpixel}}{8}
\]

# Fragment Shader calls = 8

Multisampling

\[
\text{Final Pixel Color} = 3 \times \text{One color sample from A} + 5 \times \text{One color sample from B}
\]

# Fragment Shader calls = 2
Setting up the Image

```c
VkPipelineMultisampleStateCreateInfo vpmsci;
vpmci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmci.pNext = nullptr;
vpmci.flags = 0;
vpmci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vpmci.sampleShadingEnable = VK_TRUE;
vpmci.minSampleShading = 0.5;
vpmci.pSampleMask = (VkSampleMask *)nullptr;
vpmci.alphaToCoverageEnable = VK_FALSE;
vpmci.alphaToOneEnable = VK_FALSE;

vkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
...
vgpci.pMultisampleState = &vpmsci;
```

Result = `vkCreateGraphicsPipelines` (LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline).

**Setting up the Image**

**At least** this fraction of samples will get their own fragment shader calls (as long as they pass the depth and stencil tests).

- 0 produces simple multisampling
- (0,1) produces partial supersampling
- 1 produces complete supersampling

```c
VkAttachmentDescription vad;
va[0].format = VK_FORMAT_B8G8R8A8_SRGB;
va[0].samples = VK_SAMPLE_COUNT_8_BIT;
va[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
va[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
va[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
va[0].alphaToCoverageEnable = VK_FALSE;
va[0].alphaToOneEnable = VK_FALSE;
va[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
va[0].flags = 0;
...
va[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
va[1].samples = VK_SAMPLE_COUNT_8_BIT;
va[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
va[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
va[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
va[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
va[1].flags = 0;
```

```c
VkAttachmentReference colorReference;
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
VkAttachmentReference depthReference;
depthReference.attachment = 1;
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Result = `vkCreateRenderPass` (LogicalDevice, IN &vrsd, PALLOCATOR, OUT &RenderPass);
Multipass Rendering uses Attachments -- What is a Vulkan Attachment Anyway?

"[An attachment is] an image associated with a renderpass that can be used as the input or output of one or more of its subpasses."

-- Vulkan Programming Guide

An attachment can be written to, read from, or both.

For example:

```
  Attachment
  Attachment
  Subpass
  Subpass
  Subpass
  Framebuffer
```

Back in Our Single-pass Days

So far, we've only performed single-pass rendering, within a single Vulkan RenderPass.

Here comes a quick reminder of how we did that. Afterwards, we will extend that.

```
  Render
  Framebuffer
```

Resolving the Image:
Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

```c
VOffset3D vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;

VkExtent3D ve3;
ve3.width = Width;
ve3.height = Height;
ve3.depth = 1;

VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.mipLevel = 0;
visl.baseArrayLayer = 0;
visl.layerCount = 1;

VkImageResolve vir;
vir.srcSubresource = visl;
vir.srcOffset = vo3;

vkCmdResolveImage( cmdBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, 1, &vir );
```
Back in Our Single-pass Days, I

Multisubpass Rendering

So far, we've only performed single-pass rendering, but within a single Vulkan RenderPass, we can also have several subpasses, each of which is feeding information to the next subpass or subpasses.

In this case, we will look at following up a 3D rendering with some image processing on the outcome.

Notice how close this resembles a Directed Acyclic Graph (DAG) data structure: nodes connected by arrows that point in one direction.

Back in Our Single-pass Days, II

Multisubpass Algorithm to Render and then Image Process

Original Sharpened Edge Detected

No Noise Noise
VkAttachmentDescription vad[3];

vad[0].flags = 0;
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

vad[1].flags = 0;
vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

vad[2].flags = 0;
vad[2].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[2].samples = VK_SAMPLE_COUNT_1_BIT;
vad[2].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[2].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC;

VkAttachmentReference colorReference;
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference depthReference;
depthReference.attachment = 1;
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference outputReference;
outputReference.attachment = 2;
outputReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkSubpassDescription vsd[2];

vsd[0].flags = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[0].inputAttachmentCount = 0;
vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
vsd[0].colorAttachmentCount = 1;
vsd[0].pColorAttachments = colorReference;
vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pDepthStencilAttachment = &depthReference;
vsd[0].preserveAttachmentCount = 0;
vsd[0].pPreserveAttachments = (uint32_t *) nullptr;

vsd[1].flags = 0;
vsd[1].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[1].inputAttachmentCount = 1;
vsd[1].pInputAttachments = colorReference;
vsd[1].colorAttachmentCount = 1;
vsd[1].pColorAttachments = &outputReference;
vsd[1].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[1].pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
vsd[1].preserveAttachmentCount = 0;
vsd[1].pPreserveAttachments = (uint32_t *) nullptr;

VkSubpassDependency vsdp[1];

vsdp[0].srcSubpass = 0; // 3D rendering
vsdp[0].dstSubpass = 1; // image processing
vsdp[0].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[0].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vsdp[0].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[0].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

VkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.attachmentCount = 3; // color, depth/stencil, output
vrpci.pAttachments = vad;
vrpci.subpassCount = 2;
vrpci.pSubpasses = vsd;
vrpci.dependencyCount = 1;
vrpci.pDependencies = vsdp;

result = vkCreateRenderPass( LogicalDevice, &vrpci, pAllocator, &RenderPass );
Placing a Pipeline Barrier so an Image is not used before it is Ready

```cpp
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_OUTPUT_BIT;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vimb.subresourceRange = visr;

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN & vimb);
```

Multipass, V

```cpp
vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// first subpass is automatically started here
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *) nullptr);
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

```cpp
vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// second subpass is started here – doesn’t need any new drawing vkCmd’s
```

```cpp
vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
```

Creating a Pipeline with Dynamically Changeable State Variables

The graphics pipeline is full of state information, and, as previously-discussed, is immutable, that is, the information contained inside it is fixed, and can only be changed by creating a new graphics pipeline with new information.

That isn’t quite true. To a certain extent, you can declare parts of the pipeline state changeable. This allows you to change pipeline information on the fly.

This is useful for managing state information that needs to change frequently. This also creates possible optimization opportunities for the Vulkan driver.

Dynamic State Variables

Mike Bailey
mjb@ics.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan
Which Pipeline State Variables can be Changed Dynamically

The possible uses for dynamic variables are shown in the `VkDynamicState` enum:

- `VK_DYNAMIC_STATE_VIEWPORT`
- `VK_DYNAMIC_STATE_SCISSOR`
- `VK_DYNAMIC_STATE_LINE_WIDTH`
- `VK_DYNAMIC_STATE_DEPTH_BIAS`
- `VK_DYNAMIC_STATE_BLEND_CONSTANTS`
- `VK_DYNAMIC_STATE_DEPTH_BOUNDS`
- `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK`
- `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK`
- `VK_DYNAMIC_STATE_STENCIL_REFERENCE`

Creating a Pipeline

```cpp
VkDynamicState vds[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState);
vpdsci.pDynamicStates = &vds;

VkGraphicsPipelineCreateInfo vgpci;

vkCreateGraphicsPipelines( LogicalDevice, pipelineCache, 1, &vgpci, PALLOCATOR, &GraphicsPipeline );
```

If you declare certain state variables to be dynamic like this, then you must fill them in the command buffer! Otherwise, they are undefined and bad things are likely to happen.

Filling State Variables in the Command Buffer

The command buffer-bound function calls to set these dynamic states are:

```cpp
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetLineWidth( commandBuffer, linewidth );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
```
There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp.

Vulkan requires you to first setup "Query Pools", some for each specific type.

This indicates that Vulkan thinks that Queries are time-consuming (relatively) to setup, and thus better to set them up in program-setup than in program-runtime.

### Setting up Query Pools

- **Occlusion:**
  - Vulkan requires you to set up "Query Pools" for each specific type.
  - There are 3 types of queries: Occlusion, Pipeline Statistics, and Timestamp.
  - Vulkan's Query Pools indicate that Queries are time-consuming (relatively) to setup, thus better to set them up in program-setup rather than in program-runtime.

### Code Snippet

```cpp
// VkQueryPoolCreateInfo vqpci;
// vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
// vqpci.pNext = nullptr;
// vqpci.flags = 0;
// vqpci.queryType = (one of: VK_QUERY_TYPE_OCCLUSION, VK_QUERY_TYPE_PIPELINE_STATISTICS, VK_QUERY_TYPE_TIMESTAMP);
// vqpci.queryCount = 3;
// vqpci.pipelineStatistics = 0; // bitmask of what stats you are querying for if you are doing a pipeline statistics query

// VkQueryPool occlusionQueryPool;
// result = vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &occlusionQueryPool );

// VkQueryPool statisticsQueryPool;
// result = vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &statisticsQueryPool );

// VkQueryPool timestampQueryPool;
// result = vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &timestampQueryPool );
```

- **Resetting, Filling, and Examining a Query Pool**
  - **Resetting:**
  - Use `vkCmdResetQueryPool` to reset a query pool.
  - **Filling:**
  - Use `vkCmdBeginQuery` to begin filling the query pool.
  - **Examining:**
  - Use `vkGetQueryPoolResults` to get the results of the query pool.

```cpp
vkCmdResetQueryPool( CommandBuffer, occlusionQueryPool, 0, 3 );
kCmdBeginQuery( CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT );

do { ...

vkCmdEndQuery( CommandBuffer, occlusionQueryPool, 0 );

result = vkGetQueryPoolResults( LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE, data, stride, flags );
// VK_QUERY_RESULT_64_BIT
// VK_QUERY_RESULT_WAIT_BIT
// VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
// VK_QUERY_RESULT_PARTIAL_BIT
// stride is # of bytes in between each result

vkCmdCopyQueryPoolResults( CommandBuffer, occlusionQueryPool, 0, 1, buffer, 0, stride, flags );
// VK_QUERY_RESULT_64_BIT
// VK_QUERY_RESULT_WAIT_BIT
// VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
// VK_QUERY_RESULT_PARTIAL_BIT
// stride is # of bytes in between each result
```
Occlusion Queries count the number of fragments drawn between the vkCmdBeginQuery and the vkCmdEndQuery that pass both the Depth and Stencil tests. This is commonly used to see what level-of-detail should be used when drawing a complicated object.

Some hints:

• Don’t draw the whole scene – just draw the object you are interested in.
• Don’t draw the whole object – just draw a simple bounding volume at least as big as the object.
• Don’t draw the whole bounding volume – cull away the back faces (two reasons: time and correctness).
• Don’t draw the colors – just draw the depths (especially if the fragment shader is time-consuming).

```c
uint32_t fragmentCount;
result = vkGetQueryPoolResults( LogicalDevice, occlusionQueryPool, 0, 1, sizeof(uint32_t), &fragmentCount, 0, VK_QUERY_RESULT_WAIT_BIT);
```

```c
vkCmdCopyQueryPoolResults( CommandBuffer, occlusionQueryPool, 0, 1, buffer, 0, 0, VK_QUERY_RESULT_WAIT_BIT );
```

Pipeline Statistics Queries count how many of various things get done between the vkCmdBeginQuery and the vkCmdEndQuery.

```c
uint32_t counts[NUM_STATS];
result = vkGetQueryPoolResults( LogicalDevice, statisticsQueryPool, 0, 1, NUM_STATS*sizeof(uint32_t), counts, 0, VK_QUERY_RESULT_WAIT_BIT );
```

```c
vkCmdCopyQueryPoolResults( CommandBuffer, occlusionQueryPool, 0, 1, buffer, 0, 0, VK_QUERY_RESULT_WAIT_BIT );
```

```c
VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT
VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT
VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT
VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT
VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT
VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT
VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT
VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT
VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT
VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT
VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT
```

Timestamp Queries count how many nanoseconds of time elapsed between the vkCmdBeginQuery and the vkCmdEndQuery.

```c
uint64_t nanosecondsCount;
result = vkGetQueryPoolResults( LogicalDevice, timestampQueryPool, 0, 1, sizeof(uint64_t), &nanosecondsCount, 0, VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT );
```

```c
vkCmdCopyQueryPoolResults( CommandBuffer, timestampQueryPool, 0, 1, buffer, 0, 0, VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT );
```

The vkCmdWriteTimeStamp( ) function produces the time between when this function is called and when the first thing reaches the specified pipeline stage. Even though the stages are “bits”, you are supposed to only specify one of them.

```c
vkCmdWriteTimeStamp( CommandBuffer, pipelineStages, timestampQueryPool, 0 );
```

```c
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
```
An Introduction to the Vulkan Computer Graphics API

Mike Bailey
Oregon State University
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.