Ongoing Notes and Code

The notes and code presented here are constantly being updated. Go to:

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

for all the latest versions.

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 Vulkan class at OSU – Winter Quarter, 2018. Thanks for your courage and patience!

Second, thanks to NVIDIA! Their 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!
What Prompted the Community’s 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.
Vulkan

- Largely 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
  - 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.
VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = << buffer size in bytes ... = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = nullptr;

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

    VkMemoryRequirements vmr;
    result = vkGetBufferMemoryRequirements ( LogicalDevice, Buffer, OUT &vmr );  // fills vmr

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

    result = vkAllocateMemory ( LogicalDevice, IN &vmai, PALLOCATOR, &MatrixBufferMemoryHandle );
    result = vkBindBufferMemory ( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );

Vulkan Code has a Distinct “Style”

Vulkan Quick Reference Card
**Vulkan Highlights: a More Typical Block Diagram**

- **Application**
- **Instance**
- **Physical Device**
- **Logical Device**

**Vulkan Synchronization**

- Synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan does not ever lock – that's the application's job
- Threads can concurrently read from the same object
- Threads can concurrently write to different objects

**Vulkan Shaders**

- GLSL is the same as before ... almost
- In places it’s still an implied `#define VULKAN 100` is automatically supplied by the compiler
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)
- SPIR-V gets turned into fully-compiled code at runtime
- The SPIR-V spec has been public for months – new shader languages are surely being developed
- OpenCL and OpenGL will be moving to SPIR-V as well

**The Sample2017.zip File Contains This**

1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends
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:

```c
typedef VkBuffer VkDataBuffer;
```

### Terminology Issues

- **A 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:

```c
typedef VkBuffer VkDataBuffer;
```
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

```c
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

```c
int FindMemoryThatIsHostVisible( uint32_t memoryTypeBits )
{
    VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
    for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
    {
        VkMemoryType vmt = vpdmp.memoryTypes[i];
        VkMemoryPropertyFlags vmpf = vmt.propertyFlags;
        if( ( memoryTypeBits & (1<<i) ) != 0 )
        {
            if( ( vmpf & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )
            {
                return i;
            }
        }
    }
    return -1;
}
```

11 Memory Types:

- Memory 0:
- Memory 1:
- Memory 2:
- Memory 3:
- Memory 4:
- Memory 5:
- Memory 6:
- Memory 7: DeviceLocal
- Memory 8: DeviceLocal
- Memory 9: HostVisible HostCoherent
- Memory 10: HostVisible HostCoherent HostCached

2 Memory Heaps:
- Heap 0: size = 0xb7c00000 DeviceLocal
- Heap 1: size = 0xfac00000

Finding the Right Type of Memory

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

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

```c
Something I've Found Useful
```

```
It's the usual object-oriented benefit – you can pass around just one data item and everyone can access whatever information they need.
```
Initializing a Data Buffer

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

```c
VkResult Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer)
{
    vbci.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
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
}
```

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));
```

The Parade of Data

There is one more step in here – Descriptor Sets. Here’s a quick preview…
Creating and Filling the Data Buffer – the Details

Filling the Data Buffer

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

```
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;
    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.

Creating and Filling the Data Buffer – the Details

Vertex Buffers
Vertex Buffers are how you draw things in Vulkan. They are very much like Vertex Buffer Objects in OpenGL, but more detail is exposed to you (a lot more…).

But, the good news is that Vertex Buffers are really just ordinary Data Buffers, so some of the functions will look familiar to you.

First, a quick review of computer graphics geometry . . .

**What is a Vertex Buffer?**

**Vertex Orientation Issues**

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system. Internally, however, the Vulkan pipeline uses a left-handed system:

The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the projection matrix, like this:

ProjectionMatrix[1][1] *= -1;

This is like saying \(Y' = -Y\).

**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**

**A Colored Cube Example**

```cpp
static GLuint CubeTriangleIndices[3][3] = {
    {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}
};
```
Triangles in an Array of Structures

This object was modeled such that triangles that face the viewer will look like their vertices are oriented CCW (this is detected by looking at vertex orientation at the start of the rasterization).

Because this 3D object is closed, Vulkan can save rendering time by not even bothering with triangles whose vertices look like they are oriented CW. This is called backface culling.

Vulkan’s change in coordinate systems can mess up the backface culling. So I recommend, at least at first, that you do no culling.

Vertex Orientation Issues

MyBuffer MyVertexDataBuffer;
InitMyVertexDataBuffer(sizeof(VertexData), &MyVertexDataBuffer);
FillMyDataBuffer( MyVertexDataBuffer, (void *) VertexData );

VkResult InitMyVertexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = InitDataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
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++
struct vertex {
  glm::vec3 position;
  glm::vec3 normal;
  glm::vec3 color;
  glm::vec2 texCoord;
};

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

VkVertexInputAttributeDescription vviad[4];      // array per vertex input attribute
// 4 = vertex, normal, color, texture coord
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++
VkPipelineInputAssemblyStateCreateInfo vpiasci;
// used to describe the input vertex attributes
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
#nullable disable
vpiasci.flags = 0;
#nullable enable
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

VkPipelineInputAssemblyStateCreateInfo vpiasci;
// used to describe the input vertex attributes
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
#nullable disable
vpiasci.flags = 0;
#nullable enable
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

Vulkan vertex shader code

```c++
void renderPass(VkPipeline &pipeline)
{
  VkCommandBuffer commandBuffer = BeginNongraphicCommandBuffer();

  VkRenderPassBeginInfo renderPassInfo;
  renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  renderPassInfo.renderPass = renderPass;
  renderPassInfo.framebuffer = framebuffer;
  renderPassInfo.renderArea = VkRect2D{VK_OFFSET_2D, VK_OFFSET_2D, VkExtent2D{width, height}};
  vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

  VkPipelineInputAssemblyStateCreateInfo vpiasci;
  vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
  vpiasci.pNext = nullptr;
  vpiasci.flags = 0;
  vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

  VkPipelineInputLayoutCreateInfo vpiaci;
  vpiaci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_LAYOUT_CREATE_INFO;
  vpiaci.pNext = nullptr;
  vpiaci.setLayoutCount = layoutCount;
  vpiaci.pSetLayouts = layoutHandles;

  VkPipelineInputAssemblyStateCreateInfo vpiasci;
  vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
  vpiasci.pNext = nullptr;
  vpiasci.flags = 0;
  vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

  VkPipelineInputLayoutCreateInfo vpiaci;
  vpiaci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_LAYOUT_CREATE_INFO;
  vpiaci.pNext = nullptr;
  vpiaci.setLayoutCount = layoutCount;
  vpiaci.pSetLayouts = layoutHandles;

  VkPipelineVertexInputStateCreateInfo vpvisci;
  vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
  vpvisci.pNext = nullptr;
  vpvisci.flags = 0;
  vpvisci.vertexBindingDescriptionCount = 1;
  vpvisci.pVertexBindingDescriptions = &vvibd[0];
  vpvisci.vertexAttributeDescriptionCount = 4;
  vpvisci.pVertexAttributeDescriptions = &vviad[0];

  VkGraphicsPipelineCreateInfo vgpci;
  vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
  vgpci.pNext = nullptr;
  vgpci.flags = 0;
  vgpci.stageCount = 2;
  vgpci.pStages = &vpssci[0];
  vgpci.pVertexInputState = &vpvisci;
  vgpci.pInputAssemblyState = &vpiasci;
  vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
  vgpci.pViewportState = &vpvsci;
  vgpci.pRasterizationState = &vprsci;
  vgpci.pMultisampleState = &vpmsci;
  vgpci.pColorBlendState = &vpbcs;
  vgpci.pDynamicState = &vpds;
  vgpci.layout = IN GRAPHICSPipelineLayout;
  vgpci.renderPass = IN RenderPass;
  vgpci.basePipelineHandle = VK_NULL_HANDLE;
  result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vgpci,
                                     ALLOCATOR, 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;
vkCmdBindVertexBuffer(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);
```

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

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

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

Triangles Represented as an Array of Structures

From the file SampleVertexData.cpp:

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

```c
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. }
    },
};
```

Non-indexed Buffer Drawing

From the file SampleVertexData.cpp:

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

```c
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. }
    },
};
```
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;
}

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

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.

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

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

C/C++:

GLSL Shader:

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 );

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

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
Drawing with an Indexed Buffer

vkCmdBindVertexBuffer( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );

vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );

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

Drawing with an Indexed Buffer

VkResult
Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    return result;
}

Init05MyVertexDataBuffer( sizeof(JustVertexData), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );

Init05MyIndexDataBuffer( sizeof(JustIndexData), &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );

Drawing with an Indexed Buffer

VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer }; VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer }; vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets ); // 0, 1 = firstBinding, bindingCount vkCmdBindIndexBuffer( 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

vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );

Sometimes the Same Point Needs Multiple Attributes

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.
Sometimes the Same Point Needs Multiple Attributes

Where values match at the corners (color)

Where values do not match at the corners (texture coordinates)

Shaders and SPIR-V

Vulkan Shader Stages

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,
} 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

Vertex and Instance indices:
- gl_VertexIndex
- gl_InstanceIndex
- Both are 0-based

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

Shader combinations of separate texture data and samplers:
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );

Descriptor Sets:
layout( set=0, binding=0 )

Specialization Constants:
layout( constant_id = 3 ) const int N = 5;
- 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:
layout( local_size_x_id = 8, local_size_y_id = 16 );
- gl_WorkGroupSize.z is still as it was

Vulkan: Shaders’ use of Layouts for Uniform Variables

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

All opaque (non-sampler) uniform variables must be in block buffers

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:
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

```
```

Shaderfile 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 10 from a Bash Shell

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

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
```
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 ...
```

Reading a SPIR-V File into a Vulkan Shader Module

```
VkResult
Init12spirVShader( 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 );

    VkShaderModuleCreateInfo vsmci;
    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, &vsmci, PALLOCATOR, pShaderModule );
    fprintf( FpDebug, "Shader Module '%s' successfully loaded
                filename.c_str() );
    delete [][] code;
    return result;
}
```

Vulkan Sample Code
Caveats on the Sample Code

- I've written everything out in appalling longhand.
- Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.
- 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.
- I’ve setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
- At times, I’ve setup things that didn’t need to be setup just to show you what could go there.
- There are good uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
- I’ve typedef’ed a couple things to make the Vulkan phraseology more consistent.
- Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
- At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
- I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Sample Program Output

// Keyboard commands:
//      'i', 'I': Toggle using a vertex buffer only vs. a vertex/index buffer
//      'l', 'L': Toggle lighting off and on
//      'm', 'M': Toggle display mode (textures vs. colors)
//      'p', 'P': Pause the animation
//      'q', 'Q': Esc: exit the program
//      'r', 'R': Toggle rotation-animation and using the mouse
//      '1', '4', '9': Number of instances

Sample Code Keyboard Commands
Main Program

```c
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 );
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
    Reset();
    InitGraphics();
    // loop until the user closes the window:
    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;
}
```

InitGraphics() I

```c
void InitGraphics() {
    HERE_I_AM "InitGraphics";
    VkResult result = VK_SUCCESS;
    InitGLFW();
    InitCreateDebugCallbacks();
    InitPhysicalDeviceAndGetQueueFamilyProperties();
    InitLogicalDeviceAndQueue();
    InitUniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    FillDataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    InitUniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    FillDataBuffer( MyLightUniformBuffer, (void *) &Light );
    InitMyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    FillDataBuffer( MyVertexDataBuffer, (void *) VertexData );
    InitCommandPool( );
    InitCommandBuffers( );
    InitTextureSampler( &MyPuppyTexture.texSampler );
    InitTextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    InitSwapchain();
    InitDepthStencilImage();
    InitRenderPasses();
    InitFramebuffers();
    InitPrivShader( "sample-vert.spv", ShaderModuleVertex );
    InitPrivShader( "sample-frag.spv", ShaderModuleFragment );
    InitDescriptorSetPool();
    InitDescriptorSetLayouts();
    InitDescriptorSets();
    InitGraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
                                      VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

InitGraphics() II

```c
#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.,  0.,  0. },
    {  1., 1. }
};
```

The Vertex Data is in a Separate File
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 typedef'ed structs pass all the necessary information into a function. For example, where we might normally say in C++:

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

   we would actually say in C:

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

Vulkan Conventions

My Conventions

- **Vk** | Typedef, probably a struct
- **vk** | Function call
- **VK_XXX** | Constant

"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 function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, they have no significance.

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

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

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

Your Sample2017.zip File Contains This

- Double-click here to launch Visual Studio 2017 with this solution
- Windows shader compiler
- Linux shader compiler

Double-click here to launch Visual Studio 2017 with this solution
Setting Up GLFW

```c
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);
    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\"n", key, key);
                fflush(FpDebug);
        }
    }
}
```

Looping and Closing GLFW

```c
while (glfwWindowShouldClose(MainWindow) == 0)
{
    glfwPollEvents();
    Time = glfwGetTime();          // elapsed time, in double-precision seconds
    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/

You invoke GLM like this:

```cpp
#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. Later on, these notes will show you how to use it from there.

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:

- `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:

```cpp
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) );
```

Exactly the same concept, but a different expression of it. Read on for details …

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

```cpp
// constructor:
glm::mat4( );
glm::vec4( ); // identity matrix
glm::vec3( );

// multiplications:
glm::mat4 * glm::mat4
glm::vec4 * glm::vec4( glm::vec3 ) // 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 );
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::ortho( float left, float right, float bottom, float top );
glm::mat4 glm::frustum( float left, float right, float bottom, float top, float near, float far );
glm::mat4 glm::perspective( float fovy, float aspect, float near, float far );

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

Installing GLM into your own space

I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.

Telling Visual Studio about where the GLM folder is

1. A period, indicating that the project folder should also be searched when a #include <xxx> is encountered. If you put it somewhere else, enter that full or relative path instead.

2.
GLM in the Vulkan sample.cpp Program

```cpp
if (UseMouse)
{
    if (Scale < MINSCALE)
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4(); // identity
    Matrices.uModelMatrix = glm::scale(Matrices.uModelMatrix, glm::vec3(Scale, Scale, Scale));
    Matrices.uModelMatrix = glm::rotate(Matrices.uModelMatrix, Yrot, glm::vec3(0., 1., 0.));
    Matrices.uModelMatrix = glm::rotate(Matrices.uModelMatrix, Xrot, glm::vec3(1., 0., 0.));
    // done this way, the Yrot is applied first, then the Xrot, then the Scale
}
else
{
    if (!Paused)
    {
        const glm::vec3 axis = glm::vec3(0., 1., 0.);
        Matrices.uModelMatrix = glm::rotate(glm::mat4(), (float)glm::radians(360.f * Time / SECONDS_PER_CYCLE), axis);
    }
}
Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
Matrices.uProjectionMatrix[1][1] *= -1.;
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
Fill05DataBuffer(MyMatrixUniformBuffer, (void *) &Matrices);
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer(MyMiscUniformBuffer, (void *) &Misc);
}
```

Your Sample2017.zip File Contains GLM Already

Instancing

- 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

Instancing – What and why?

- `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:

```glsl
int NUMINSTANCES = 16;
float DELTA = 3.0;
float xdelta = DELTA * float( gl_InstanceIndex % 4 );
float ydelta = DELTA * float( gl_InstanceIndex / 4 );
vColor = vec3( 1., float( (1.+gl_InstanceIndex) ) / float( NUMINSTANCES ), 0. );
xdelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;
ydelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;
vec4 vertex = vec4( aVertex.xyz + vec3( xdelta, ydelta, 0. ), 1. );
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:

```glsl
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;
```
In OpenGL

OpenGL puts all uniform data in the same "set", but with different binding numbers, so you can get at each one.

Each uniform variable gets updated one-at-a-time. Wouldn't it be nice if we could update a bunch of related uniform variables all at once?

```glsl
layout( std140, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, binding = 1 ) uniform mat4 uViewMatrix;
layout( std140, binding = 2 ) uniform mat4 uProjectionMatrix;
layout( std140, binding = 3 ) uniform mat3 uNormalMatrix;
layout( std140, binding = 4 ) uniform vec4 uLightPos;
layout( std140, binding = 5 ) uniform float uTime;
layout( std140, binding = 6 ) uniform int uMode;
layout( binding = 7 ) uniform sampler2D uSampler;
```

Descriptor Sets are an intermediate data structure that tells shaders how to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders. There are three advantages in doing things this way:

- Related uniform variables can be updated as a group, gaining efficiency.
- Descriptor Sets are activated when the Command Buffer is filled. Different values for the uniform buffer variables can be toggled by just swapping out the Descriptor Set that points to GPU memory, rather than re-writing the GPU memory.
- Values for the shaders' uniform buffer variables can be compartmentalized into what quantities change often and what change seldom (scene-level, model-level, draw-level), so that uniform variables need to be re-written no more often than is necessary.

```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
    }
  }
}
```

What are Descriptor Sets?

Descriptor Sets help connect uniform variables to GPU memory. They make it easier to manage related data:

- **CPU**: Uniform data created in a C++ data structure
  - Know the CPU data structure
  - Know where the data starts
  - Know the data's size

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

- **GPU**: Uniform data used in the shader
  - Knows where the data starts
  - Knows the data's size

```glsl
struct matBuf
{
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat3 uNormalMatrix;
} Matrices;

struct lightBuf
{
  vec4 uLightPos;
} Light;

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

struct samplerBuf
{
  sampler2D uSampler;
} Samplers;
```

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

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

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

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```
Step 1: Descriptor Set Pools

You don’t allocate Descriptor Sets on the fly — that is too slow. Instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

```
mjb – August 1, 2018
```

```
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.

```
mjb – August 1, 2018
```

```
VkResult
Init13DescriptorSetPool()
{
    VkResult result;
    VkDescriptorPoolSize vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;
    #ifdef CHOICES
    VK_DESCRIPTOR_TYPE_SAMPLER
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
    #endif
    VkDescriptorPoolCreateInfo
    vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;
    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = &vdps[0];
    result = vkCreateDescriptorPool
    ( LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool );
    return result;
}
```

```
VkResult
Init13DescriptorSetLayouts()
{
    VkResult result;
    // DS #0:
    VkDescriptorSetLayoutBinding
    MatrixSet[1];
    MatrixSet[0].binding            = 0;
    MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MatrixSet[0].descriptorCount = 1;
    MatrixSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    MatrixSet[0].pImmutableSamplers = (VkSampler *)nullptr;
    // DS #1:
    VkDescriptorSetLayoutBinding
    LightSet[1];
    LightSet[0].binding            = 0;
    LightSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    LightSet[0].descriptorCount = 1;
    LightSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    LightSet[0].pImmutableSamplers = (VkSampler *)nullptr;
    // DS #2:
    VkDescriptorSetLayoutBinding
    MiscSet[1];
    MiscSet[0].binding            = 0;
    MiscSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MiscSet[0].descriptorCount = 1;
    MiscSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    MiscSet[0].pImmutableSamplers = (VkSampler *)nullptr;
    // DS #3:
    VkDescriptorSetLayoutBinding
    TexSamplerSet[1];
    TexSamplerSet[0].binding            = 0;
    TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    TexSamplerSet[0].descriptorCount = 1;
    TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
    TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;
```

```
uniform sampler2D uSampler;
vec4 rgba = texture( uSampler, vST );
```
Step 2: Define the Descriptor Set Layouts

MatrixSet DS Layout Binding: 
<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LightSet DS Layout Binding: 
<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MiscSet DS Layout Binding: 
<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TexSamplerSet DS Layout Binding: 
<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Layout

MatrixSet CI: 
<table>
<thead>
<tr>
<th>binding</th>
<th>type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LightSet CI: 
<table>
<thead>
<tr>
<th>binding</th>
<th>type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MiscSet CI: 
<table>
<thead>
<tr>
<th>binding</th>
<th>type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TexSamplerSet CI: 
<table>
<thead>
<tr>
<th>binding</th>
<th>type</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Step 4: Allocating the Memory for Descriptor Sets

vkAllocateDescriptorSets() {
    VkDescriptorSetAllocateInfo descriptorSetCount;
    descriptorSetCount.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    descriptorSetCount.pNext = nullptr;
    descriptorSetCount.descriptorCount = 4;
    descriptorSetCount.pPool = DescriptorSetPool;
    descriptorSetCount.pSetLayouts = &GraphicsPipelineLayout;
    result = vkAllocateDescriptorSets(LogicalDevice, IN descriptorSetCount, PALLOCATOR, OUT &DescriptorSet);
    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, &vdsai, DescriptorSets[0]);
}
```

Step 5: Allocating the Memory for Descriptor Sets

```c
VkWriteDescriptorSet
vwds0;
vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds0.pNext = nullptr;
vwds0.dstSet = DescriptorSets[0];
vwds0.dstBinding = 0;
vwds0.dstArrayElement = 0;
vwds0.descriptorCount = 1;
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.pBufferInfo = &vdbi0;
vwds0.pImageInfo = nullptr;
vwds0.pTexelBufferView = nullptr;
```

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

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

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

```c
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 = &vdbi1;
vwds2.pImageInfo = nullptr;
vwds2.pTexelBufferView = nullptr;
```

Step 5: Tell the Descriptor Sets where their data is

```c
Step 5: Tell the Descriptor Sets where their data is

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

```c
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.pBufferInfo = nullptr;
vwds3.pImageInfo = &vdii0;
vwds3.pTexelBufferView = nullptr;
```

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

```
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#ifdef CHOICESVK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
#endif
vgpci.stageCount = 2;                           // number of stages in this pipeline
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 = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines ( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT &GraphicsPipeline );
```

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

```
vkCmdBindDescriptorSets ( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipelineLayout, 0, 4, descriptorSets, 0, (uint32_t *)nullptr );
```

The Graphics Pipeline

What is the Vulkan Graphics Pipeline?

```
Don’t worry if this is too small to read – a larger version is coming up.
There is also a Vulkan Compute Pipeline.
```

Here’s what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”.
2. There’s a lot that goes into it.
3. 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.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.
Graphics Pipeline Stages and what goes into them

- The GPU and Driver specify the Pipeline Stages – the Vulkan Graphics Pipeline declares what goes in them

- **Input Assembly**
  - Topology
  - Viewport

- **Viewport**
  - Depth Clipping
  - Scissoring

- **Rasterization**
  - CullMode
  - FrontFace
  - LineWidth

- Dynamic State
  - Which states are dynamic
  - DepthTestEnable
  - DepthWriteEnable
  - DepthCompareOp
  - StencilTestEnable
  - stencilOpStateFront
  - stencilOpStateBack
  - blendEnable
  - srcColorBlendFactor
  - dstColorBlendFactor
  - colorBlendOp
  - srcAlphaBlendFactor
  - dstAlphaBlendFactor
  - alphaBlendOp
  - colorWriteMask

- **Color Blending Stage**
  - Color Blending parameters

- **Tesselation State**
  - Fragment Shader module
  - Specialization info

- **Vertex Input Stage**
  - Vertex Shader module
  - Specialization info
  - Vertex Input binding

- **Vertex Input attributes**

Vulkan: A Pipeline Records the Following Items:

- **Graphics Pipeline Stages and what goes into them**
  - 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
  - Topology
  - Viewport
  - Scissoring
  - CullMode
  - FrontFace
  - LineWidth
  - Depth Clipping
  - Depth Test Enable
  - Depth Write Enable
  - Depth Compare Op
  - Stencil Test Enable
  - Stencil Op State Front
  - Stencil Op State Back
  - Blend Enable
  - Color Blending parameters

**Bold/Static** indicates that this state item can also be set with Dynamic Variables

Creating a Graphics Pipeline from a lot of Pieces

- **Graphics Pipeline Stages and what goes into them**
  - 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
  - Topology
  - Viewport
  - Scissoring
  - CullMode
  - FrontFace
  - LineWidth
  - Depth Clipping
  - Depth Test Enable
  - Depth Write Enable
  - Depth Compare Op
  - Stencil Test Enable
  - Stencil Op State Front
  - Stencil Op State Back
  - Blend Enable
  - Color Blending parameters

- **Dynamic State** which states can be set dynamically (bound to the command buffer, outside the Pipeline)
Creating a Typical Graphics Pipeline

Link in the Shaders

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 Per-Vertex Attributes

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.
vpiasci.primitiveRestartEnable = VK_FALSE;

"Restart Enable" is used with:
• Indexed drawing.
• Triangle Fan and "Strip topologies

If vpiasci.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, it is 0xffffffff.

One Really Good use of Restart Enable is in Drawing Terrain
Surfaces with Triangle Strips

Triangle Strip #0:
Triangle Strip #1:
Triangle Strip #2:

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 (squished) 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.
Setting the Rasterizer State

- `VkPipelineRasterizationStateCreateInfo`:
  - `sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;`
  - `pNext = nullptr;`
  - `flags = 0;`
  - `depthClampEnable = VK_FALSE;`
  - `rasterizerDiscardEnable = VK_FALSE;`
  - `polygonMode = VK_POLYGON_MODE_FILL;`
  - `cullMode = VK_CULL_MODE_NONE;`
  - `frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;`
  - `depthBiasEnable = VK_FALSE;`
  - `depthBiasConstantFactor = 0.f;`
  - `depthBiasClamp = 0.f;`
  - `depthBiasSlopeFactor = 0.f;`
  - `lineWidth = 1.f;`

Which Pipeline Variables can be Set Dynamically

- `VkDynamicState`:
  - `vds = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };`

- `VkPipelineDynamicStateCreateInfo`:
  - `vpdsci = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;`
  - `pNext = nullptr;`
  - `flags = 0;`
  - `dynamicStateCount = 0;`
  - `pDynamicStates = vds;`

Stencil Operations for Front and Back Faces

- `VkStencilOpState`:
  - `vsosf = vk::StencilOpState();`  // 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;`  // what to do if stencil operation succeeds
  - `vsosf.compareMask = ~0;`
  - `vsosf.writeMask = ~0;`
  - `vsosf.reference = 0;`

  - `vsosb = vk::StencilOpState();`  // back
  - `vsosb.depthFailOp = VK_STENCIL_OP_KEEP;`
  - `vsosb.failOp = VK_STENCIL_OP_KEEP;`
  - `vsosb.passOp = VK_STENCIL_OP_KEEP;`
  - `vsosb.compareOp = VK_COMPARE_OP_NEVER;`
  - `vsosb.compareMask = ~0;`
  - `vsosb.writeMask = ~0;`
  - `vsosb.reference = 0;`

Uses for Stencil Operations

- Magic Lenses
- Polygon edges without Z-fighting
Group all of the individual state information and create the pipeline

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

Queues and Command Buffers

Vulkan: a More Typical (and Simplified) Block Diagram
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.

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;
}
```

Querying what Queue Families are Available

```c
uint32_t count;
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++) {
    fprintf(FpDebug, "Queue Family Count = %2d  ;   
    if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
        fprintf(FpDebug, "Graphics\n");            
    if((vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
        fprintf(FpDebug, "Compute \n");          
    if((vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
        fprintf(FpDebug, "Transfer\n");          
    fprintf(FpDebug, \n";
}  
```

Creating a Logical Device Queue Needs to Know Queue Family Information

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

Creating the Command Buffers

```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:
    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;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR(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);
    vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
```
These are the Commands that could be entered into the Command Buffer, I

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);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdBlitImage(commandBuffer, filter);
vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdPushDescriptorSetKHR(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);
vkCmdPushDescriptorSetWithTemplateKHR(commandBuffer, descriptorUpdateTemplate, layout, set, pData);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyImage(commandBuffer, pRegions);
vkCmdCopyImageToBuffer(commandBuffer, pRegions);
vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdResetEvent(commandBuffer, event, stageMask);
vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdResolveImage(commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdSetDeviceMaskKHR(commandBuffer, deviceMask);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdDrawIndirect(commandBuffer, stride);
vkCmdDrawIndirectCountAMD(commandBuffer, stride);
vkCmdEndQuery(commandBuffer, query);
vkCmdEndRenderPass(commandBuffer);
vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);
vkCmdSetStencilReference(commandBuffer, faceMask, reference);
vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewportWScalings);
vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);
vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query);

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

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

VkResult result;
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);

VkClearDepthStencilValue vcdsv;
vcdsv.depth = 1.f;

result = vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN VK_NULL_HANDLE, OUT &nextImageIndex);

VkExtent2D e2d = { Width, Height };

VkRect2D r2d = { o2d, e2d };

result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);

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(commandBuffer, IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
mjb – August 1, 2018

161

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

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);

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);
How We Think of OpenGL Framebuffers

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

Vulkan Thinks of it This Way

- Swap Chain
- Depth-Buffer
- Update

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. This 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”)

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?**

*Image of a swap chain diagram showing being drawn into and being presented.*

---

**We Need to Find Out What our Display Capabilities Are**

- **VulkanDebug.txt output:**
  - `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`:
    - `minImageCount = 2`
    - `maxImageCount = 8`
    - `currentExtent = 1024 x 1024`
    - `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` (`VK_FORMAT_B8G8R8A8_UNORM`, `VK_COLORSPACE_SRGB_NONLINEAR_KHR`)
    - 1: `50` (`VK_FORMAT_B8G8R8A8_SRGB`, `VK_COLORSPACE_SRGB_NONLINEAR_KHR`)
  - Found 3 Present Modes:
    - 0: `2` (`VK_PRESENT_MODE_FIFO_KHR`)
    - 1: `3` (`VK_PRESENT_MODE_FIFO_RELAXED_KHR`)
    - 2: `1` (`VK_PRESENT_MODE_MAILBOX_KHR`)

---

**Creating a Swap Chain**

- **Creating a Swap Chain**
  - `VkSurfaceCapabilitiesKHR vsc;`  
  - `vkGetPhysicalDeviceSurfaceCapabilitiesKHR(PhysicalDevice, Surface, OUT &vsc);`
  - `VkExtent2D surfaceRes = vsc.currentExtent;`
  - `VkSwapchainCreateInfoKHR vscci;`  
  - `vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;`  
  - `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;`  
  - `vkSwapchainCreateInfoKHR = NULL;`  
  - `vkCreateSwapchainKHR(LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain);`

---

**We Need to Find Out What our Display Capabilities Are**

- **VulkanDebug.txt output:**
  - `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`:
    - `minImageCount = 2`
    - `maxImageCount = 8`
    - `currentExtent = 1024 x 1024`
    - `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` (`VK_FORMAT_B8G8R8A8_UNORM`, `VK_COLORSPACE_SRGB_NONLINEAR_KHR`)
    - 1: `50` (`VK_FORMAT_B8G8R8A8_SRGB`, `VK_COLORSPACE_SRGB_NONLINEAR_KHR`)
  - Found 3 Present Modes:
    - 0: `2` (`VK_PRESENT_MODE_FIFO_KHR`)
    - 1: `3` (`VK_PRESENT_MODE_FIFO_RELAXED_KHR`)
    - 2: `1` (`VK_PRESENT_MODE_MAILBOX_KHR`)

---

**Creating a Swap Chain**

- **Creating a Swap Chain**
  - `VkSurfaceCapabilitiesKHR vsc;`  
  - `vkGetPhysicalDeviceSurfaceCapabilitiesKHR(PhysicalDevice, Surface, OUT &vsc);`
  - `VkExtent2D surfaceRes = vsc.currentExtent;`
  - `VkSwapchainCreateInfoKHR vscci;`  
  - `vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;`  
  - `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;`  
  - `vkSwapchainCreateInfoKHR = NULL;`  
  - `vkCreateSwapchainKHR(LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain);`
Creating the Swap Chain Images and Image Views

```c
uint32_t imageCount; // # of display buffers = 2? 3?
result = vkGetSwapchainImagesKHR(LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr);

PresentImages = new VkImage[imageCount];
result = vkGetSwapchainImagesKHR(LogicalDevice, SwapChain, OUT &imageCount, PresentImages);

// present views for the double-buffering:
PresentImageViews = new VkImageView[imageCount];
for( unsigned int i = 0; i < imageCount; i++ )
{
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_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.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vivci.subresourceRange.baseMipLevel = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;
    vivci.image = PresentImages[i];
    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
}
```

Rendering into the Swap Chain, I

```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);

uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
result = vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
... vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
... vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
```

Rendering into the Swap Chain, II

```c
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
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 = ~0u;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = &SemaphoreRenderFinished;
result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence); // 1 = submitCount
```
1. vkBeginRenderPass()
2. vkCmdBindPipeline(CommandBuffer, …)
3. vkCmdSetxxx(CommandBuffer, yyy)  dynamic states
4. vkCmdBindDescriptorSets(CommandBuffer, …), which also includes Push Constants
5. vkCmdBindVertexBuffers(CommandBuffer, …)
6. vkCmdDraw(CommandBuffer, vertexCount, instanceCount, firstVertex, firstInstance)
7. vkEndRenderPass()

vkCreateRenderPass()
VkRenderPassCreateInfo
VkAttachmentDescription
VkSubpassDescription
VkPipelineBindPoint
Depth/Stencil Attributes
Color Attributes
vkCreateFrameBuffer()
VkFrameBufferCreateInfo
vkRenderPassBeginInfo
Clear Values
Depth/Stencil
Color
RenderArea
width
height
layers
bindingCount
BuffersOffsets
VkSubpassDescription
vkCreateGraphicsPipeline()

Vulkan: Rendering
Vulkan: Beginning a Command Buffer
Vulkan: Submitting to a Queue
VkResult RenderScene() {
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, &imageReadySemaphore);
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR(LogicalDevice, SwapChain, UINT64_MAX, imageReadySemaphore, VK_NULL_HANDLE, &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], &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.f;
    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; // used for VK_ATTACHMENT_LOAD_OP_CLEAR
    vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], &vrpbi, VK_SUBPASS_CONTENTS_INLINE);
    vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
    VkViewport viewport = { 0., 0., (float)Width, (float)Height, 0., 1. };
    vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, &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);
    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]);
    vkCmdEndCommandBuffer(CommandBuffers[nextImageIndex]);
    VkFenceCreateInfo vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = 0;
    VkFence renderFence;
    vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, &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 );

vkPresentInfoKHR vpi;
  vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  vpi.pNext = nullptr;
  vpi.waitSemaphoreCount = 0;
  vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
  vpi.swapchainCount = 1;
  vpi.pSwapchains = &SwapChain;
  vpi.pImageIndices = &nextImageIndex;
  vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR( presentQueue, IN &vpi );
vkDestroySemaphore( LogicalDevice, imageReadySemaphore, PALLOCATOR );
```

```
Textures

```

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

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

```

```
VkDescriptorSetLayoutBinding TexSamplerSet[1];
  TexSamplerSet[0].binding            = 0;
  TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
  // uniform sampler2D uSampler
  // vec4 rgba = texture( uSampler, vST );
  TexSamplerSet[0].descriptorCount = 1;
  TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
  TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;
  . . .

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

```
Textures

```

```
```

```
Texture RGBA Data Values

```

```
Texture Sampler

```

```
Combined Image Sampler

```

```
```

NVIDIA Discrete Graphics:
11 Memory Types:
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal
- Memory 2: HostVisible HostCoherent
- Memory 3: HostVisible HostCoherent HostCached

Intel Integrated Graphics:
3 Memory Types:
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal HostVisible HostCoherent
- Memory 2: DeviceLocal HostVisible HostCoherent HostCached

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 been included in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets included somehow in all 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 that the T:P ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin: multōm in parvo, “many things in a small place”
// *******************************************************************************
// this second {...} is to create the actual texture image:
// *******************************************************************************
{
    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;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

    if (vsl.rowPitch == 4 * texWidth) {
        result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled
        memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
    } else {
        unsigned char *gpuBytes = (unsigned char *)gpuMemory;
        for (unsigned int y = 0; y < texHeight; y++) {
            vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
            memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth));
        }
        if (Verbose) {
            fprintf(FpDebug, "Texture vmr.size = %lld
", vmr.size);
            fprintf(FpDebug, "Texture vmr.alignment = %lld
", vmr.alignment);
            fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
            fflush(FpDebug);
        }
    }

    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal();  // because we want to sample from it

    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
    result = vkBindImageMemory(LogicalDevice, IN textureImage, IN vdm, 0);        // 0 = offset
}

// copy pixels from the staging image to the texture:
// *********************************************************** *******************
{
    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;

    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;

    result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

    // ************************************************************ *******************
    // transition the staging buffer layout:// *******************************************************************************
    {
        vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        vimb.image = textureImage;
        vimb.srcAccessMask = 0;
        vimb.dstAccessMask = 0;
        vimb.subresourceRange = visr;

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

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

    // now do the final image transfer:
    // *********************************************************** *******************
    {
        voice.x = 0; voice.y = 0; voice.z = 0;
        voice = 0, (VkMemoryBarrier *)nullptr, 1, IN &vimb );
    }

    // *********************************************************** *******************
    // copy pixels from the staging image to the texture:
    // *********************************************************** *******************
    {
        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;

        visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        visr.baseMipLevel = 0;
        visr.levelCount = 1;
        visr.baseArrayLayer = 0;
        visr.layerCount = 1;

        result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

        vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        vimb.pNext = nullptr;
        vimb.oldLayout = ... = stagingImage;
        vimb.srcAccessMask = 0;
        vimb.dstAccessMask = 0;
        vimb.subresourceRange = visr;

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

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

    // now do the final image transfer:
    // *********************************************************** *******************
    {
        Voice.x = 0; Voice.y = 0; Voice.z = 0;
        voice = 0, (VkMemoryBarrier *)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);

VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT; visr.baseMipLevel = 0; visr.levelCount = 1; visr.baseArrayLayer = 0; visr.layerCount = 1;
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER; vimb.pNext = nullptr; vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_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 = 0; 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);

result = vkEndCommandBuffer(TextureCommandBuffer);
VkSubmitInfo vsi;
vsii.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsii.pNext = nullptr;
vsii.commandBufferCount = 1; vsii.pCommandBuffers = &TextureCommandBuffer; vsii.waitSemaphoreCount = 0; vsii.pWaitSemaphores = nullptr; vsii.pSignalSemaphores = nullptr; vsii.pWaitDstStageMask = nullptr;

result = vkQueueSubmit(Queue, 1, IN &vsi, VK_NULL_HANDLE);
result = vkQueueWaitIdle(Queue);

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.

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;
Vulkan: Overall Block Diagram

Instance

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Vulkan: Identifying the Physical Devices

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

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

Querying the Number of Physical Devices

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

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

Vulkan: Which Physical Device to Use, I

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties( IN physicalDevices[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 =", vpdp.deviceType) ;
    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
    }
Which Physical Device to Use, II

// need some logic 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;

int 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\n" );
    return VK_SHOULD_EXIT;

Here’s What the NVIDIA 1080ti Produced

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

Here’s What the Intel HD Graphics 520 Produced

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
Asking About the Physical Device’s Different Memories

VkPhysicalDeviceMemoryProperties

vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
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, "%d Memory Heaps:
", vpdmp.memoryHeapCount);
for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ )
{
fprintf(FpDebug, "Heap %d:
", i);
VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
fprintf( FpDebug, " size = 0x%08lx
", (unsigned long int)vmh.size);
if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT  ) != 0 )     fprintf( FpDebug, "DeviceLocal" );
// only one in use
}
Vulkan

Logical Devices

Vulkan: Overall Block Diagram

Instance → Physical Device

Logical Device → Logical Device

Queue

Command Buffer

Vulkan: Specifying a Logical Device Queue

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

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

result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
Vulkan: Creating the Logical Device’s Queue

// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue ); // 0, 0 = queueFamilyIndex, queueIndex

Layers and Extensions

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_monitor",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_NV_optimus";
};

const char * instanceExtensions[ ] = {
    "VK_KHR_surface",
    #ifdef _WIN32
    "VK_KHR_win32_surface",
    #endif
    "VK_EXT_debug_report",
};

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

vkEnumerateInstanceLayerProperties( &numLayersAvailable, (VkLayerProperties *)nullptr );

InstanceLayers = new VkLayerProperties[ numLayersAvailable ];
result = vkEnumerateInstanceLayerProperties( &numLayersAvailable, InstanceLayers );

vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, (VkExtensionProperties *)nullptr );

InstanceExtensions = new VkExtensionProperties[ numExtensionsAvailable ];
result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, InstanceExtensions );

vkEnumerateInstanceLayerProperties:
13 instance layers enumerated:
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'
0x00400038   1  'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x0040000d   1  'VK_LAYER_NV_nsight' 'NVIDIA Nsight interception layer'
0x00400000  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_KHR_get_physical_device_properties2’
0x00000001  ‘VK_KHR_get_surface_capabilities2’
0x00000019  ‘VK_KHR_surface’
0x00000006  ‘VK_KHR_win32_surface’
0x00000001  ‘VK_KHR_device_group_creation’
0x00000001  ‘VK_KHR_external_fence_capabilities’
0x00000001  ‘VK_KHR_external_fence_capabilities’
0x00000001  ‘VK_NV_external_memory_capabilities’

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’

Will now ask for only 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report

// look for extensions both on the wanted list and the available list:
std::vector<char *> extensionsWantedAndAvailable;
extensionsWantedAndAvailable.clear();
for( uint32_t wanted = 0; wanted < numExtensionsWanted; wanted++ )
{
    for( uint32_t available = 0; available < numExtensionsAvailable; available++ )
    {
        if(strcmp( instanceExtensions[wanted], InstanceExtensions[available].extensionName ) == 0)
        {
            extensionsWantedAndAvailable.push_back( InstanceExtensions[available].extensionName );
            break;
        }
    }
}

// create the instance, asking for the layers and extensions:
VkInstanceCreateInfo vici;
vici.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
vici.pNext = nullptr;
vici.flags = 0;
vici.pApplicationInfo = &vai;
vici.enabledLayerCount = sizeof(instanceLayers) / sizeof(char *);
vici.ppEnabledLayerNames = instanceLayers;
vici.enabledExtensionCount = extensionsWantedAndAvailable.size();
vici.ppEnabledExtensionNames = extensionsWantedAndAvailable.data();;

result = vkCreateInstance( IN &vici, PALLOCATOR, OUT &Instance );
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[PhysicalDeviceCount];
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 );
    // 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;
}

int 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;

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. };
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_LUNARG vrouwen",
};

const char * myDeviceExtensions[ ] = {
    "VK_KHR_swapchain",
};

uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
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);
    delete[] deviceExtensions;
}

delete[] myDeviceLayers;
delete[] myDeviceExtensions;
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':

Synchronization

Vulkan Highlights: Overall Block Diagram

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

Your program continues

Try to Use the Something

Semaphore
Creating a Semaphore

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

result = vkCreateSemaphore(LogicalDevice, &vsci, &imageReadySemaphore);
```

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

Semaphores Example during the Render Loop

```
// returns right away:
result = vkGetFenceStatus(LogicalDevice, fence);
// result = VK_SUCCESS means it has signaled
// result = VK_NOT_READY means it has not signaled
// blocks:
result = vkWaitForFences(LogicalDevice, 1, &fence, waitForAll, timeout);
// 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
```

Fences

```
#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

result = vkCreateFence(LogicalDevice, &vfci, &fence);
result = vkGetFenceStatus(LogicalDevice, fence);
// result = VK_SUCCESS means it has signaled
// result = VK_NOT_READY means it has not signaled
// blocks:
result = vkWaitForFences(LogicalDevice, 1, &fence, waitForAll, timeout);
// 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

```c
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);
```

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

```c
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
```

Controlling Events from the Device

```c
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);
```

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

Note: the device cannot test for an event, but it can block
From the Command Buffer Notes:
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)`
- `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)`
- `vkCmdPushDescriptorSetKHR(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites)`
- `vkCmdPushDescriptorSetWithTemplateKHR(commandBuffer, descriptorUpdateTemplate, layout, set, pData)`
- `vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo)`
- `vkCmdResetEvent(commandBuffer, event, stageMask)`
- `vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount)`
- `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)`
- `vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewportWScalings)`
- `vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData)`
- `vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers)`
- `vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query)`

From the Command Buffer Notes:
These are the Commands that could 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)`
- `vkCmdPushDescriptorSetKHR(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites)`
- `vkCmdPushDescriptorSetWithTemplateKHR(commandBuffer, descriptorUpdateTemplate, layout, set, pData)`
- `vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo)`
- `vkCmdResetEvent(commandBuffer, event, stageMask)`
- `vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount)`
- `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)`
- `vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewportWScalings)`
- `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
vkCmdPipelineBarrier( ) Function Call

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.

```c
vkCmdPipelineBarrier(commandBuffer,
                     srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT,
                     memoryBarrierCount, pMemoryBarriers,
                     bufferMemoryBarrierCount, pBufferMemoryBarriers,
                     imageMemoryBarrierCount, pImageMemoryBarriers);
```

Defines what data we will be blocking/un-blocking on.

Pipeline Stage Flags – Where in the Pipeline is this Memory being Accessed?

- 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

Note: the concept of an in-order pipeline is accurate, but really the src and dst triggering action only depends on the name of the street where you are right now.

The Scenario

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 its way through the 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).
### Access Masks – What are you Interested in Generating or Consuming this Memory for?

<table>
<thead>
<tr>
<th>Access Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
</tr>
</tbody>
</table>

### Pipeline Stages and what Access Operations can Happen There

<table>
<thead>
<tr>
<th>Stage</th>
<th>Access Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_INTERCEPT=<a href="https://www.example.com">https://www.example.com</a></td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>dst</td>
</tr>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>src</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>src</td>
</tr>
</tbody>
</table>

### Access Operations and what Pipeline Stages they can be used In

<table>
<thead>
<tr>
<th>Access Types</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>dst</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>src</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>dst</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
</tr>
<tr>
<td>src</td>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
</tr>
<tr>
<td>dst</td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
</tr>
<tr>
<td>src</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>dst</td>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER</td>
</tr>
<tr>
<td>src</td>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
</tr>
<tr>
<td>dst</td>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
</tr>
</tbody>
</table>

### Example: Be sure we are done writing an output image before using it for something else

Example: Be sure we are done writing an output image before using it for something else.
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:

```
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:

```
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, PALLOCATOR, OUT &GraphicsPipelineLayout);
```

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
};
```

struct armArm1;
struct armArm2;
struct armArm3;

In the Reset Function

```c++
struct arm
{
    Arm1;
    Arm2;
    Arm3;
    ...
};
```

Arm1.armMatrix = glm::mat4(1);
Arm1.armColor = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale = 6.f;

Arm2.armMatrix = glm::mat4(1);
Arm2.armColor = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale = 4.f;

Arm3.armMatrix = glm::mat4(1);
Arm3.armColor = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale = 2.f;

The constructor glm::mat4() produces an identity matrix. The actual transformation matrices will be set in UpdateScene().
Setup the Push Constant for the Pipeline Structure

```cpp
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);

struct VkPipelineLayoutCreateInfo
{
    VkStructureType sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    const void* pNext = nullptr;
    uint32_t flags = 0;
    uint32_t setLayoutCount = 4;
    const DescriptorSetLayout* pSetLayouts = DescriptorSetLayouts;
    uint32_t pushConstantRangeCount = 1;
    const VkPushConstantRange* pPushConstantRanges = &vpcr;
}

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

In the UpdateScene Function

```cpp
float rot1 = (float)Time;
float rot2 = 2.f * rot1;
float rot3 = 2.f * rot2;

glm::vec3 zaxis = glm::vec3(0., 0., 1.);
glm::mat4 m1g = glm::mat4();
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);
m1g = glm::translate(m1g, glm::vec3(0., 0., 2.));

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

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

Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m1g * m21; // m2g
Arm3.armMatrix = m1g * m21 * m32; // m3g
```

In the RenderScene Function

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

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

In the Vertex Shader

```cpp
layout( push_constant ) uniform arm
{
    mat4 armMatrix;
    vec3 armColor; // scale factor in x
}

RobotArm;

layout( location = 0 ) in vec3 aVertex;

vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.]
vec3 Scale = (RobotArm.armMatrix * vec4(bVertex, 1.));

vec3 bVertex = Scale * (RobotArm.armScale);
```

v1 = (1.0 * v1 - 1.0) * (1.0 * v1 - 1.0);
```
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 to this:

1. **Supersampling**: Pick some number of unique points within a pixel, render the image into each of these sub-pixels (including depth and stencil tests), then average them together.

2. **Multisampling**: Pick some number of unique points within each pixel and perform a depth and stencil render there. Then, perform a single color render for that pixel. Assign that RGBA to all the sub-pixels that made it through the depth and stencil tests.
**Supersampling**

\[
\text{Final Pixel Color} = \frac{\sum_{i=1}^{8} \text{Color sample from subpixel}_i}{8}
\]

# Fragment Shader calls = 8

**Multisampling**

\[
\text{Final Pixel Color} = \frac{3 \times \text{One color sample from A} + 5 \times \text{One color sample from B}}{8}
\]

# Fragment Shader calls = 2

---

**Setting up the Image**

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

VkGraphicsPipelineCreateInfo vgpci;
vpmci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vpmci.pNext = nullptr;
...;
vgpci.pMultisampleState = &vpmsci;

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );
```

---

**Setting up the Image**

- VK_TRUE means to allow some sort of multisampling to take place
- 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
Setting up the Image

```c
VkAttachmentDescription vad[2];
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
vad[0].flags = 0;
vad[1].flags = 0;

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;
```

Resolving the Image:

```c
VlOffset3D 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;
vir.dstSubresource = visl;
vir.dstOffset = vo3;

vkCmdResolveImage(cmdBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, 1, &vir);
```

Multipass Rendering

```c
VkSubpassDescription vsd;
vsd.flags = 0;
vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd.inputAttachmentCount = 0;
vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
vsd.colorAttachmentCount = 1;
vsd.pColorAttachments = &colorReference;
vsd.pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd.pDepthStencilAttachment = &visl;
vsd.preserveAttachmentCount = 0;
vsd.pPreserveAttachments = (uint32_t *)nullptr;

VkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.attachmentCount = 2; // color and depth/stencil
vrpci.pAttachments = vad;
vrpci.subpassCount = 1;
vrpci.pSubpasses = &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = &vsd;
result = vkCreateRenderPass(LogicalDevice, IN &vrpci, 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:

- Subpass 1: Attachment
- Subpass 2: Attachment
- Framebuffer

Multipass 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.

Multipass Algorithm to Render and then Image Process

Back in Our Single-pass Days

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

VkAttachmentDescription

#include [ 3 ];
#include [0].flags = 0;
#include [0].format = VK_FORMAT_B8G8R8A8_SRGB;
#include [0].samples = VK_SAMPLE_COUNT_1_BIT;
#include [0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
#include [0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
#include [0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
#include [0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
#include [0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
#include [0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

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

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

Multipass, II

VkAttachmentReference colorReference;

#include [0].attachment = 0;colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference depthReference;

#include [1].attachment = 1;depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference outputReference;

#include [2].attachment = 2;outputReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

Multipass, III

VkSubpassDescription

#include [2];

VkSubpassDescription

pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;

inputAttachmentCount = 0;

pInputAttachments = (VkAttachmentReference *)nullptr;

colorAttachmentCount = 1;

pColorAttachments = colorReference;

colorAttachmentCount = 1;

pColorAttachments = &outputReference;

colorAttachmentCount = 1;

pColorAttachments = &depthReference;

colorAttachmentCount = 1;

pColorAttachments = &colorReference;

colorAttachmentCount = 1;

pColorAttachments = &colorReference;

colorAttachmentCount = 1;

pColorAttachments = &colorReference;

colorAttachmentCount = 1;

pColorAttachments = &colorReference;

Multipass, IV

VkSubpassDependency

#include [1];

srcSubpass = 0; // 3D rendering

dstSubpass = 1; // image processing

srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;

dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;

dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

VkRenderPassCreateInfo

#include [1];

flags = 0;

attachmentCount = 3;

pAttachments = vad;

subpassCount = 2;

pSubpasses = vsd;

dependencyCount = 1;

pDependencies = vsdp;

result = vkCreateRenderPass

LogicalDevice, IN &vpci, PALLOCATOR, OUT &RenderPass );
Placing a Pipeline Barrier so an Image is not used before it is Ready

```c
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;
vCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, nullptr, 0, nullptr, 1, &vimb);
```

Multipass, V

```c
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);
vkCmdBindVertexBuffer(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
... vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// second subpass is started here – doesn’t need any new drawing vkCmd’s ...
vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
```

A Wrap-up: Here are some good Vulkan References


The notes and code presented here are constantly being updated. Go to:

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

for all the latest versions.

Mike Bailey
Oregon State University
mjb@cs.oregonstate.edu

Please Fill out the Course Survey!

SIGGRAPH 2018 Mobile App

Complete the Survey

It’s as easy as:
- Navigating to this session in the app
- Scrolling to the bottom of the screen and...
- Answering less than 5 questions