What Prompted the Move to Vulkan?

1. Performance
2. Performance
3. Performance

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

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

As an aside, the Vulkan development effort was originally called glNext, which created the false impression that this was a replacement for OpenGL. It’s not.

Why is it so important to keep the GPU Busy?

From Wikipedia:

“Vulcan is the god of fire including the fire of volcanoes, metalworking, and the forge in ancient Roman religion and myth. Vulcan is often depicted with a blacksmith’s hammer. The Vulcanalia was the annual festival held August 23 in his honor. His Greek counterpart is Hephaestus, the god of fire and smithy. In Etruscan religion, he is identified with Sethians. Vulcan belongs to the most ancient stage of Roman religion. Varro, the ancient Roman scholar and writer, citing the Annales Maximi, records that king Titus Tatius dedicated altars to a series of deities among which Vulcan is mentioned.”

Who was the original Vulcan?
Why Name it after the God of the Forge?

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.

Who is the Khronos Group?

Who’s Been Specifically Working on Vulkan?

Playing “Where’s Waldo” with Khronos Membership

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 – Vulkan can crash
• Goal: higher single-threaded performance than OpenGL can deliver
• Goal: able to do multithreaded graphics
• Goal: able to handle tiled rendering

Vulkan Differences from OpenGL

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

The Basic Computer Graphics Pipeline, Vulkan-style

A Complete API Redesign

Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- Think OpenCL...
- E.g., vkCmdDraw(…, cmdBuffer, …);
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed when the application wants them flushed
- Each command buffer can be filled from a different thread (i.e., filling is thread-safe)
Vulkan Highlights: Pipelines

- In OpenGL, your "pipeline state" is whatever your current graphics attributes are: color, transformations, textures, shaders, etc.
- Changing the state on-the-fly one item at-a-time is very expensive
- Vulkan forces you to set all your state at once into a "pipeline state object" (PSO) and then invoke the entire PSO whenever you want to use that state combination
- Think of the pipeline state as being immutable.
- Potentially, you could have thousands of these pre-prepared states
- This is a good time to talk about how game companies view Vulkan…

Vulkan Code has a Distinct "Style"

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = <buffer size in bytes>;
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;

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

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

Querying the Number of Things

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

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

Vulkan Quick Reference Card

1. Vulkan Highlights: Pipelines
2. Vulkan Code has a Distinct "Style"
3. Querying the Number of Things
4. Vulkan Quick Reference Card
Vulkan Highlights: a More Typical Block Diagram

1. Application
2. Instance
3. Physical Device
4. Logical Device

Steps in Creating Graphics using Vulkan

1. Create the Instance
2. Setup the Debug Callbacks
3. Create the Surface
4. List the Physical Devices
5. Pick the right Physical Device
6. Create the Logical Device
7. Create the Uniform Variable Buffers
8. Create the Vertex Data Buffers
9. Create the texture sampler
10. Create the texture images
11. Create the Swap Chain
12. Create the Depth and Stencil Images
13. Create the RenderPass
14. Create the Framebuffers
15. Create the Descriptor Set Pool
16. Create the Command Buffer Pool
17. Create the Command Buffer(s)
18. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render- ...

Vulkan: Creating a Pipeline

- VkGraphicsPipelineCreateInfo
- Shader stages
- VertexInput State
- InputAssembly State
- Tesselation State
- Viewport State
- Rasterization State
- MultiSample State
- DepthStencil State
- ColorBlend State
- Dynamic State
- Pipeline layout
- RenderPass
- basePipelineHandle
- basePipelineIndex

- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkVertexInputBindingDescription
- VkViewportStateCreateInfo
- VkPipelineRasterizationStateCreateInfo
- VkPipelineInputAssemblyStateCreateInfo
- VkVertexInputAttributeDescription
- VkPipelineDepthStencilStateCreateInfo
- VkPipelineColorBlendStateCreateInfo
- VkPipelineDynamicStateCreateInfo
- VkPipelineLayout

- VkShaderModule
- VkSpecializationInfo

- VkPipeline::createGraphicsPipeline

Vulkan GPU Memory

- Your application allocates GPU memory for the objects it needs
- You map GPU memory to the CPU address space for access
- Your application is responsible for making sure what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.

From the OpenGL Shader Storage Buffer notes:

- glGenBuffers( 1, &posSSbo );
- glBindBuffer( GL_SHADER_STORAGE_BUFFER, posSSbo );
- glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct pos), NULL, GL_STATIC_DRAW );
- GLint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT; // the invalidate makes a big difference when re-writing
- struct pos *points = (struct pos *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct pos), bufMask );

Vulkan Render Passes

- Drawing is done inside a render pass
- Each render pass contains what framebuffer attachments to use
- Each render pass is told what to do when it begins and ends
- Multiple render passes can be merged

Vulkan Compute Shaders

- Compute pipelines are allowed, but they are treated as something special (just like OpenGL does)
- Compute passes are launched through dispatches
- Compute command buffers can be run asynchronously
Vulkan Synchronization

- Vulkan tries to run "flat out"
- Therefore, 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
- For places it’s not, an implied `#define VULKAN 100`
- 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

GLSL Source → SPIR-V

Advantages:
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

Your Sample2017.zip File Contains This

Vertex Buffers.pptx

VerteBuffers 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...
typedef enum VkPrimitiveTopology
{
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;

OpenGL Topologies – Polygon Requirements

- Convex
- Planar

Vulkan Topologies – Requirements and Orientation

- Convex and Planar
- CCW when viewed from outside the solid object

What does "Convex Polygon" Mean?

We can go all mathematical here, but let's go visual instead. In a convex polygon, a line between any two points inside the polygon never leaves the inside of the polygon.
Why is there a Requirement for Polygons to be Convex?

Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon’s edges.

Convex Not Convex

Why is there a Requirement for Polygons to be Planar?

Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time.

OK OK Not OK

What if you need to display Polygons that are not Convex?

There is an open source library to break a non-convex polygon into convex polygons. It is called Polypartition, and is found here:

https://github.com/ivanfratric/polypartition

If you ever need to do this, contact me. I have working code …

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”.

A Colored Cube Example

Triangles in an Array of Structures

Modelled in right-handed coordinates
Vertex Orientation Issues

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

Because the 2D 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.

The Vulkan Pipeline

| Vertex Input Stage | Input Assembly | Transformation | Vertex Iteration | Rasterization | Dynamic State | Color Blend
|--------------------|----------------|----------------|------------------|--------------|---------------|-----------|

Filling the Vertex Buffer

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::vec2 texCoord;
};
```

CUDA

```c
layout( location = 0 ) in vec3 aPosition;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec2 aTexCoord;
layout( location = 3 ) in vec2 aTexCoord2;
```

GLSL

```glsl
void main() {
    gl_Position = vec4(position, 1.0);
    normal = vec3(position - vPosition);
    texCoord = texCoord2;  // use texture coordinates
}
```
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
VkPipelineVertexInputStateCreateInfo vpvisci;          // used t o describe the input vertex attributes
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.vertexBindingDescriptionCount = 1;
vpvisci.pVertexBindingDescriptions = &vvibd;
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = &vviad;

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;;

VkGraphicsPipelineCreateInfo vgpci;

vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

vgpci.pNext = nullptr;

vgpci.flags = 0;

vgpci.stageCount = 2;                // number of shader stages in this pipeline
vgpci.pStages = &vpssci;

vgpci.pVertexInputState = &vpvisci;

vgpci.pInputAssemblyState = &vpiasci;

vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci

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.subpass = 0;                              // subpass number

vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;

vgpci.basePipelineIndex = 0;

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

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.

Telling the Command Buffer what Vertices to Draw

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

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

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, 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 );
```

Better to do this than to hard-code a number

Vulkan Topologies

```
typedef enum VkPrimitiveTopology
{
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;
```
Non-indexed Buffer Drawing

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

Filling the Vertex Buffer

```
result = Init05DataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer);
```

A Reminder of What Init05DataBuffer Does

```
body[4] = pMyBuffer->buffer + pMyBuffer->size * i1;  // pMyBuffer is a pointer to MyBuffer
```

GLSL Shader:

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

We will come to the Pipeline later, but for now, know that a Vulkan pipeline is essentially a very large state machine, including how to parse its input.
### 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
VkPipeline vertexPipeline;  // used to describe the input vertex attributes
vkGetPipelineInfoKHR(vertexPipeline, &vertexInfo);  // get vertex info
vkGetPipelineInfoKHR(vertexPipeline, &vertexInfo);  // get vertex info
```

### 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
VulkanBuffer vertexBuffer = ...;  // vertex buffer
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);  // draw vertices
```

### Drawing with an Indexed Buffer

Remember that integer-indexed buffers are just BLOBs too.

```c
vkCmdBindIndexBuffer(commandBuffer, indexDataBuffer, indexOffset, indexType);  // bind index buffer
```
Drawing with an Indexed Buffer

```c
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 );
#else
vkCmdDrawIndirect( CommandBuffers[nextImageIndex], buffer, offset, drawCount, stride );
#endif

typedef struct
{
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;

Compare this with:

vkCmdDrawIndexedIndirect( commandBuffer, buffer, offset, drawCount, stride );

typedef struct
{
    uint32_t    indexCount;
    uint32_t    instanceCount;
    uint32_t    firstIndex;
    int32_t     vertexOffset;
    uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;

Compare this with:

vkCmdDrawIndexedIndirect( commandBuffer, buffer, offset, drawCount, stride );

Indexed Indirect Drawing (i.e., both Indexed and Indirect)


typedef struct
{
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;

Sometimes the Same Point Needs Multiple Attributes

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

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.
The OBJ File Format – a triple-indexed way of Drawing

From the Quick Reference Card

Vulkan: Buffers

Vulkan: Creating a Data Buffer

Data Buffers

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 so far as to override some of Vulkan’s own terminology:

typedef VkBuffer VkDataBuffer;

Vulkan: Buffers

Vulkan: Creating a Data Buffer
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 );
```

Finding the Right Type of Memory

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

Finding the Right Type of Memory

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

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

### 2 Memory Heaps:
- Heap 0: size = 0xb7c00000 DeviceLocal
- Heap 1: size = Default20000

Something I’ve Found Useful

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

```c
MyBuffer MyMatrixUniformBuffer;
```

It’s the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need.
Here’s the C struct to hold some uniform variables:

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

Here’s the shader code to access those uniform variables:

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

Filling those Uniform Variables

```c
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

Filling the Data Buffer

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

Creating and Filling the Data Buffer – the Details

```c
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &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 );
    return result;
}
```
Copy to GPU Memory via Memory Mapping

```c
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 );
    // 0 and 0 are offset and flags
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}
```

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

### The Shaders’ View of the Basic Computer Graphics Pipeline

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

### Vulkan Shader Stages

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

### Vulkan: GLSL Differences from OpenGL

- In the complete, there is an automatic
  `#define VULKAN 100`

#### Vertex and Instance indices:

- `gl_VertexID` and `gl_InstanceID` in OpenGL. The Vulkan names make more sense.

#### gl_FragColor:

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

### Shader combinations of separate texture data and samplers:

```c
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );
```

#### Descriptor Sets:

```c
layout( set=0, binding=0 )
```

#### Specialization Constants:

- `const int N = 5;`
- Only for scalars, but a vector can be constructed from specialization constants

```c
#layout( local_size_x_id = 8, local_size_y_id = 16 )
```

### Push Constants:

- `layout(push_constant) . . .`

### Vulkan Shader Stages

```c
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
```
The first open standard intermediate language for parallel compute and graphics:

- SPIR-V (Standard Portable Intermediate Representation) was initially developed for use by OpenCL and SPIR versions 1.2 and 2.0 were based on LLVM. SPIR-V was then developed as an open API standard that is fully defined by Khronos with active support from the compiler and runtime community. It is called SPIR-V.

- SPIR-V is the first open standard, cross-API intermediate language for vertex and fragment processors and is comprised of four increasingly expressive subsets: OpenCL 2.1 and OpenCL 2.2 and the new Vulkan graphics and compute API.

- SPIR-V is an intermediate representation that can be used to represent parallel compute and graphics kernels. It is based on LLVM and is designed to be a standard portable intermediate representation for use by Vulkan and OpenCL. SPIR-V is a cross-API standard that is fully defined by Khronos with active support from the compiler and runtime community. It is known as SPIR-V.

- SPIR-V 1.1, launched in parallel with OpenCL 2.2, now supports the full language features of OpenCL 1.2.2 and OpenCL 2.2 and is compatible with the OpenGL shading language. It also enhances the expressive capabilities of kernel programs by supporting named barriers, subgroup execution, and program scope variables.

- SPIR-V is a modular and scalable language that can be split into multiple modules, each corresponding to a specific feature. This allows for the creation of custom language extensions and makes it easy to add new features to the language.

- For developers using SPIR-V, it means that kernel source code no longer needs to be directly expressed. Instead, high-level languages such as RIBC can be used to generate SPIR-V code, which is then compiled to final binary code.

- 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 specific 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

Vulkan Shader Compiling

- You do:
  - External Compiler
  - Driver

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

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

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

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

Pick one:
- Can get to your personal folders
- Does not have make
- Cannot get to your personal folders
- Does have make
Running glslangValidator.exe

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

You can also run SPIR-V from a Linux Shell

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

You can also run SPIR-V from a Linux Shell

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

The input file. The compiler determines the shader type by the file extension.

- .vert Vertex shader
- .tcs Tessellation Control Shader
- .tecs Tessellation Evaluation Shader
- .geom Geometry shader
- .frag Fragment shader
- .comp Compute shader

Specify the output file

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 . . .

How do you know if SPIR-V compiled successfully?

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, IN &vsmci, PALLOCATOR, pShaderModule );
    fprintf( FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );
    delete [] code;
    return result;
}
You can also take a look at SPIR-V Assembly

This prints out the SPIR-V “assembly” to standard output. Other than nerd interest, there is no graphics-programming reason to look at this.

For example, if this is your Shader Source

```glsl
void main( )
{
    layout ( location = 2 ) out vec2 vTexCoord;
    layout ( location = 0 ) out vec3 vNormal;
    layout( location = 3 ) in vec2 aTexCoord;
    layout( location = 2 ) in vec3 aColor;
    layout( location = 1 ) in vec3 aNormal;
}
```

```glsl
layout( std140, set = 1, binding = 0 ) uniform lightBuf
// non-opaque must be in a uniform block:
```
Installing Bash on Windows

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

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

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

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

Sample Program Output

- I've written everything out in appalling long hand.
- Everything is in one .c 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've 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 be done.
- There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
- I've typedef'd a couple things to make the Vulkan phrasing 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've copied lines from vulkan.h into the code as comments to show you what certain options could be.
- I've decided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.
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'
    );
    FpDebug = stderr;
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
    );
    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()

```c
void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices),   &MyMatrixUniformBuffer);
    Fill05DataBuffer( MyMatrixUniformBuffer,     (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light),      &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer(  sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer,                   (void *) VertexData );
    Init06CommandPool( );
    Init06CommandBuffers( );
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init08Swapchain( );
    Init09DepthStencilImage( );
    Init10RenderPasses( );
    Init11Framebuffers( );
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorSetLayouts();
    Init13DescriptorSets( );
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
          VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

A Colored Cube

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

The Vertex Data is in a Separate File

```c
#include "SampleVertexData.cpp"
```
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};

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

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

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

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums
2. Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary information into a function. For example, we might normally say in C++:
   ```
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex,  OUT &Queue );
   ```
   we would actually say in C:
   ```
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex,  OUT &Queue );
   ```

Vulkan Conventions

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

My Conventions

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

The number after "Init" gives you the ordering

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

"Find" in a function call name means that something is being looked for

"Fill" in a function call name means that some data is being supplied to Vulkan

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

(define IN)
(define OUT)

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

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

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

#define REPORT(s)               PrintVkError( result, s );  fflush(FpDebug);
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****\n", s );  fflush(FpDebug); }

bool Paused;
bool Verbose;
#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );

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 Mouse Button Callback

```c
void GLFWMouseButton( GLFWwindow *window, int button, int action, int mods )
{
    int b = 0; // LEFT, MIDDLE, or RIGHT
    // get the proper button bit mask:
    switch( button )
    {
        case GLFW_MOUSE_BUTTON_LEFT:
            b = LEFT; break;
        case GLFW_MOUSE_BUTTON_MIDDLE:
            b = MIDDLE; break;
        case GLFW_MOUSE_BUTTON_RIGHT:
            b = RIGHT; break;
        default:
            fprintf( FpDebug, "Unknown mouse button: %d
", button);
            b = 0;
    }

    // button down sets the bit, up clears the bit:
    if( action == GLFW_PRESS )
    {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;
        Ymouse = (int)ypos;
        ActiveButton |= b; // set the proper bit
    }
    else
    {
        ActiveButton &= ~b; // clear the proper bit
    }
}
```

GLFW Mouse Motion Callback

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

Looping and Closing GLFW

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

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

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

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

you would now have to say:

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

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

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
### The Most Useful GLM Variables, Operations, and Functions

**// constructor:**

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

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

**// multiplications:**

```cpp
glm::mat4 * glm::mat4
glm::mat4 * glm::vec4
glm::mat4 * glm::vec4(glm::vec3)  // promote vec3 to a vec4 via a constructor
```

**// emulating OpenGL transformations with concatenation:**

```cpp
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):**

```cpp
glm::mat4 glm::ortho(float left, float right, float bottom, float top, float near, float far);
```

**// viewing (assign, not concatenate):**

```cpp
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.

---

**Here’s what that GLM folder looks like**

---

**Telling Visual Studio about where the GLM folder is**

1. 
2. 
3. 
4. 
5. 

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.
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 Xrot is applied first, then the Yrot, 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 ) );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );

Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
```

How Does this Matrix Stuff Really Work?

This is called a “Linear Transformation” because all of the coordinates are raised to the 1st power, that is, there are no x^2, x^3, etc. terms.

Or, in matrix form:

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    A & B & C & D \\
    E & F & G & H \\
    I & J & K & L \\
    0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

Translation

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

Rotation about X

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & \cos\theta & -\sin\theta & 0 \\
    0 & \sin\theta & \cos\theta & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

Rotation about Y

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    \cos\theta & 0 & \sin\theta & 0 \\
    0 & 1 & 0 & 0 \\
    -\sin\theta & 0 & \cos\theta & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

Rotation about Z

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    \cos\theta & -\sin\theta & 0 & 0 \\
    \sin\theta & \cos\theta & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

The Rotation Matrix for an Angle (θ) about an Arbitrary Axis (Ax, Ay, Az)

\[
\begin{bmatrix}
    \cos\theta & \sin\theta & 0 & 0 \\
    -\sin\theta & \cos\theta & 0 & 0 \\
    A_x - \cos\theta(A_y - \sin\theta A_z) & A_y - \cos\theta(A_z - \sin\theta A_x) - \sin\theta A_z & A_z - \cos\theta(A_x - \sin\theta A_y) + \sin\theta A_y & 0 \\
    A_x - \cos\theta(A_z + \sin\theta A_y) & A_z - \cos\theta(A_y - \sin\theta A_x) - \sin\theta A_y & A_y - \cos\theta(A_x + \sin\theta A_z) + \sin\theta A_z & 0 \\
\end{bmatrix}
\]

For this to be correct, A must be a unit vector.
Compound Transformations

Q: Our rotation matrices only work around the origin? What if we want to rotate about an arbitrary point (A, B)?

A: We create more than one matrix.

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

Matrix Multiplication is not Commutative

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

Matrix Multiplication is Associative

One matrix – the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

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

Wrong!

Right!

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
 x' \\
 y' \\
 z'
\end{bmatrix} =
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
 x \\
 y \\
 z
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & A \\
 0 & 1 & 0 & B \\
 0 & 0 & 1 & 0
\end{bmatrix}^{-1}
\]
Instancing – What and why?

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

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:**

```gl
int NUMINSTANCES = 16;
float DELTA = 3.0;
float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vec3 vColor = vec3(1., 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.);
```

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:**

```gl
layout(std140, set = 3, binding = 0) uniform colorBuf{
    vec3 uColors[1024];
} Colors;
```

```gl
out vec3 vColor;
```

```gl
int index = gl_InstanceIndex % 1024; // 0 - 1023
vColor = Colors.uColors[index];
```

Making each Instance look differently -- Approach #3

Put a series of unique characteristics in a data buffer, one element per instance.

Read a new characteristic for each instance

Internally uses `gl_InstanceIndex`, but you don’t
How We Constructed the Graphics Pipeline Structure Before

```
result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, IN &

vpvisci

vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.vertexBindingDescriptionCount = 2;
vpvisci.flags = 0;
vpvisci.pNext = nullptr;

vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = &vviad[0];

vpvisci.pVertexInputState = &vgpci;
vgpci.flags = 0;
vgpci.pNext = nullptr;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

// Assume we have a struct vertex,
// an array containing one of these per forum being used
// an array containing one of these per vertex attribute in all bindings
// 4 = vertex, normal, color, texture coord
vvibd[0].binding = 0; // which binding this is
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].stride = sizeof( struct vertex ); // bytes between successive entries
vvibd[1].location = 0; // location in the layout decoration
vvibd[1].offset = offsetof( struct vertex, position ); // offset from beginning of struct

vvibd[2].binding = 0; // which binding this is
vvibd[2].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;
vvibd[2].stride = sizeof( glm::vec3 ); // bytes between successive entries
vvibd[2].location = 2; // location in the layout decoration
vvibd[2].offset = offsetof( struct vertex, color ); // offset from beginning of struct

vvibd[3].binding = 0; // which binding this is
vvibd[3].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;
vvibd[3].stride = sizeof( glm::vec2 ); // bytes between successive entries
vvibd[3].location = 3; // location in the layout decoration
vvibd[3].offset = offsetof( struct vertex, texcoord ); // offset from beginning of struct

// 2 = vertex, color
// an array containing one of these per buffer being used
// an array containing one of these per vertex attribute in all bindings
// 4 = vertex, normal, color, texture coord
vviad[0].binding = 0; // which binding this is
vviad[0].location = 0; // location in the layout decoration
vviad[0].offset = offsetof( struct vertex, position ); // offset from beginning of struct
vviad[0].format = VK_FORMAT_VEC3; // r, g, b
vviad[0].binding = 0; // which binding this is part of
vviad[0].location = 0; // location in the layout decoration
vviad[0].offset = 0; // past one element, so offset is 0

vviad[1].binding = 0; // which binding this is
vviad[1].location = 0; // location in the layout decoration
vviad[1].offset = offsetof( struct vertex, normal ); // offset from beginning of struct
vviad[1].format = VK_FORMAT_VEC3; // nx, ny, nz
vviad[1].binding = 0; // which binding this is part of
vviad[1].location = 1; // location in the layout decoration
vviad[1].offset = 0; // past one element, so offset is 0

vviad[2].binding = 0; // which binding this is
vviad[2].location = 1; // location in the layout decoration
vviad[2].offset = offsetof( struct vertex, color ); // offset from beginning of struct
vviad[2].format = VK_FORMAT_VEC3; // r, g, b
vviad[2].binding = 0; // which binding this is part of
vviad[2].location = 2; // location in the layout decoration
vviad[2].offset = 0; // past one element, so offset is 0

vviad[3].binding = 0; // which binding this is
vviad[3].location = 3; // location in the layout decoration
vviad[3].offset = offsetof( struct vertex, texcoord ); // offset from beginning of struct
vviad[3].format = VK_FORMAT_VEC2; // s, t
vviad[3].binding = 0; // which binding this is part of
vviad[3].location = 4; // location in the layout decoration
vviad[3].offset = 0; // past one element, so offset is 0
```

How We Constructed the Graphics Pipeline Structure Now

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

VkVertexInputAttributeDescription vviad[4];
// an array containing one of these per vertex attribute in all bindings
// 4 = vertex, normal, color, texture coord
vviad[0].binding = 0; // which binding this is
vviad[0].location = 0; // location in the layout decoration
vviad[0].offset = offsetof( struct vertex, position ); // offset from beginning of struct
vviad[0].format = VK_FORMAT_VEC3; // x, y, z
vviad[0].binding = 0; // which binding this is part of
vviad[0].location = 0; // location in the layout decoration
vviad[0].offset = 0; // past one element, so offset is 0

vviad[1].binding = 0; // which binding this is
vviad[1].location = 0; // location in the layout decoration
vviad[1].offset = offsetof( struct vertex, normal ); // offset from beginning of struct
vviad[1].format = VK_FORMAT_VEC3; // nx, ny, nz
vviad[1].binding = 0; // which binding this is part of
vviad[1].location = 1; // location in the layout decoration
vviad[1].offset = 0; // past one element, so offset is 0

vviad[2].binding = 0; // which binding this is
vviad[2].location = 1; // location in the layout decoration
vviad[2].offset = offsetof( struct vertex, color ); // offset from beginning of struct
vviad[2].format = VK_FORMAT_VEC3; // r, g, b
vviad[2].binding = 0; // which binding this is part of
vviad[2].location = 2; // location in the layout decoration
vviad[2].offset = 0; // past one element, so offset is 0

vviad[3].binding = 0; // which binding this is
vviad[3].location = 3; // location in the layout decoration
vviad[3].offset = offsetof( struct vertex, texcoord ); // offset from beginning of struct
vviad[3].format = VK_FORMAT_VEC2; // s, t
vviad[3].binding = 0; // which binding this is part of
vviad[3].location = 4; // location in the layout decoration
vviad[3].offset = 0; // past one element, so offset is 0
```

Note: same names as before, but
different sizes
In OpenGL

OpenGL uses 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?

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

Descriptor Sets

Our example will assume the following shader uniform variables:

```glsl
main( )
void
layout ( location = 2 ) out vec2 vTexCoord;
layout ( location = 1 ) out vec3 vColor;
layout( location = 4 ) in vec3 aInstanceColor;
layout( location = 3 ) in vec2 aTexCoord;
layout( location = 2 ) in vec3 aColor;
layout( location = 1 ) in vec3 aNormal;

#extension GL_ARB_shading_language_420pack : enable

#version 400

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

// non-opaque must be in a uniform block:
layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
layout( set = 1, binding = 0 ) uniform mat4 uModelMatrix;
layout( set = 0, binding = 0 ) uniform mat4 uProjectionMatrix;

uniform mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;

vTexCoord = aTexCoord;
vColor = aInstanceColor;
vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );

float uKa, uKd, uKs, uShininess;
vec4 uLightSpecularColor;
vec4 uLightPos;
vec4 uEyePos;
float uKa, uKd, uKs, uShininess;
vec4 uLightSpecularColor;
vec4 uLightPos;
vec4 uEyePos;

// non-opaque must be in a uniform block:
struct lightBuf
{
  float  uKa, uKd, uKs, uShininess;
  vec4 uLightSpecularColor;
  vec4 uLightPos;
  vec4 uEyePos;
}

// non-opaque must be in a uniform block:
struct matBuf
{
  mat3 uNormalMatrix;
  mat4 uProjectionMatrix;
  mat4 uViewMatrix;
  mat4 uModelMatrix;
}

// non-opaque must be in a uniform block:
struct miscBuf
{
  int uLighting;
  float uTime;
}

structDescriptors
{
  uniform mat4 uModelMatrix;
  uniform mat4 uProjectionMatrix;
  uniform mat4 uViewMatrix;
  uniform mat4 uModelMatrix;
  uniform int uMode;
  uniform vec4 uLightPos;
  uniform float uKa, uKd, uKs, uShininess;
  uniform vec4 uLightSpecularColor;
  uniform vec4 uLightPos;
  uniform vec4 uEyePos;
}

Do the drawing

Bind Descriptor Set #2

for( each draw )

Bind Descriptor Set #1

for( each object )

Bind Descriptor Set #0

for( each scene )

Bird Descriptor Set #0

for each object

Bird Descriptor Set #1


What are Descriptor Sets?

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

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

Uniform data in a "blob"*

* "binary object"
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.

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.

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

```c
VkResult InitGraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout)
    return result;
}
```

Step 4: Allocating the Memory for Descriptor Sets

```c
VkWriteDescriptorSet

vkAllocateDescriptorSets

void Init13DescriptorSets()
{
    result = vkAllocateDescriptorSets(LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
}
```

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

```c
void Init14GraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
}
```

Array of Descriptor Set Layouts

```c
vkUpdateDescriptorSets
```

```c
vkUpdateDescriptorSets
```

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

```c
void Init12GraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
}
```

Good to use sizeof

```c
void Init14GraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
}
```

Step 5: Tell the Descriptor Sets where their data is

```c
void Init12GraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
}
```

```c
void Init14GraphicsPipelineLayout()
{
    VkPipelineLayoutCreateInfo vplci
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);
}
```
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

```cpp
// step 6
vkResult vgpci = 0;
vgpci.basePipelineIndex = 0;
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.subpass = 0;                              // subpass number
vgpci.renderPass = IN RenderPass;
vgpci.layout = IN
vgpci.pDynamicState = &vpdsci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pViewportState = &vpvsci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pStages = vpssci;
vgpci.stageCount = 2;                          // number of stages in this pipeline = vertex + fragment
vgpci.pNext = nullptr;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

// step 7
vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &GraphicsPipeline, PALLOCATOR, OUT &GraphicsPipeline );
```

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

```cpp
// step 7
vkCmdBindDescriptorSets( cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, DescriptorSets, 0, (uint32_t *)nullptr );
```

### What is the Vulkan Graphics Pipeline?

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

### The First Step: Create the Graphics Pipeline Layout

The Graphics Pipeline Layout is fairly static. Only the layout of the Descriptor Sets and information on the Push Constants need to be supplied.
Vulkan: A Pipeline Records the Following Items:

- Pipeline Layout: DescriptorSet, PushConstants
- Which Shaders are going to be used
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- VertexFormat: x, y, z, w, r, g, b, a, m
- Scissoring: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp,
- Link in the Shaders
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

Push Constants

Create a typical Graphics Pipeline

Vertex

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y, z, w</td>
<td>Vertex attributes</td>
</tr>
<tr>
<td>r, g, b, a</td>
<td>Color attributes</td>
</tr>
<tr>
<td>m</td>
<td>Material attributes</td>
</tr>
</tbody>
</table>

Link in the Per-Vertex Attributes

- Use one vuid array member per element in the object for the array-of-structures element you are using as vertex input
- These are defined at the top of the sample code as that you don’t need to define them for positions, normals, and tex coords

Link in the Shaders

- Use one vuid array member per shader module you are using

Creating a Typical Graphics Pipeline

Dynamic State

Viewport State

DepthStencil State

Rasterization State

Geometry Shader info

Tessellation Shader info

Declare the binding descriptions and attribute descriptions

Declare the vertex topology

State Descriptions

• Pipeline: Name, type, shader stage

• Vertex attributes: location, binding, format, offset

• Vertex bindings: binding, stride, inputRate

• Assembly: topology

• VertexFormat: x, y, z, w, r, g, b, a, m

• Scissoring: x, y, w, h

• Rasterization: cullMode, polygonMode, frontFace, lineWidth

• Depth: depthTestEnable, depthWriteEnable, depthCompareOp

• Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack

• Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp

• DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)
Options for `vpiasci.topology`

- 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

What is “Primitive Restart Enable”?

- `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.

```cpp
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`

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

```cpp````c
short int restartIndex = ~0;
``````c```

What is the Difference Between Changing the Viewport and Changing the Scissoring?

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

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

Setting the Rasterizer State

```cpp
class GameEngine::PipelineInfo

// Setup a pipeline
```

Declarations about how the rasterization will take place
vprsci.depthClampEnable = VK_FALSE;

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

A good use for this is Polygon Capping:

What is “Depth Clamp Enable”?

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

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

What is “Depth Bias Enable”?

Multisampling State

vkPipelineMultisampleStateCreateInfo vpmsci;
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
vpmsci.sampleShadingEnable = VK_FALSE;
vpmsci.minSampleShading = 0;
vpmsci.pSampleMask = (VkSampleMask *)nullptr;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;

Declare information about how the multisampling will take place

MultiSampling State

VkPipelineColorBlendAttachmentState vpcbas;

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

Color Blending State for each Color Attachment

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

Color Blending State for each Color Attachment

VkPipelineColorBlendStateCreateInfo vpcbsci;

vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;

vpcbsci.pNext = nullptr;

vpcbsci.flags = 0;

vpcbsci.logicOpEnable = VK_FALSE;

vpcbsci.logicOp = VK_LOGIC_OP_COPY;

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

vpcbsci.attachmentCount = 1;

vpcbsci.pAttachments = &vpcbas;

vpcbsci.blendConstants[0] = 0;

vpcbsci.blendConstants[1] = 0;

vpcbsci.blendConstants[2] = 0;

vpcbsci.blendConstants[3] = 0;

This controls blending between the output of the fragment shader and the input to the color attachments.

Which Pipeline Variables can be Set Dynamically?

VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };

#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport( )
VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor( )
VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth( )
VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias( )
VK_DYNAMIC_STATE_BLEND_CONSTANTS        -- vkCmdSetBendConstants( )
VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthZBounds( )
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompare Mask( )
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK     -- vkCmdSetStencilWriteM ask( )
VK_DYNAMIC_STATE_STENCIL_REFERENCE     -- vkCmdSetStencilReferen ces( )
#endif

VkPipelineDynamicStateCreateInfo vpdsci;

vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;

vpdsci.pNext = nullptr;

vpdsci.flags = 0;

vpdsci.dynamicStateCount = 0;                   // leave turned off for now

vpdsci.pDynamicStates = vds;

Which Pipeline Variables can be Set Dynamically?
Stencil Operations for Front and Back Faces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkCmdBindPipeline</td>
<td>Bind the Graphics Pipeline to the Command Buffer when Drawing</td>
</tr>
<tr>
<td>VkPipelineDepthStencilStateCreateInfo</td>
<td>Create pipeline state for depth and stencil operations.</td>
</tr>
<tr>
<td>VkStencilOpState</td>
<td>Define operations for front and back faces.</td>
</tr>
<tr>
<td>VkGraphicsPipelineCreateInfo</td>
<td>Create graphics pipeline with specified state.</td>
</tr>
</tbody>
</table>

Operations for Depth Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkCmdBindPipeline</td>
<td>Bind the Graphics Pipeline to the Command Buffer when Drawing</td>
</tr>
<tr>
<td>VkPipelineDepthStencilStateCreateInfo</td>
<td>Create pipeline state for depth and stencil operations.</td>
</tr>
<tr>
<td>VkStencilOpState</td>
<td>Define operations for front and back faces.</td>
</tr>
<tr>
<td>VkGraphicsPipelineCreateInfo</td>
<td>Create graphics pipeline with specified state.</td>
</tr>
</tbody>
</table>

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

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

Putting it all Together! (finally…)

```cpp
bool CreateGraphicsPipeline(VkDevice device, VkRenderPass renderPass, VkPipelineLayout pipelineLayout, const VkPipelineCreateInfo *pPipelineCreateInfo, void **ppGraphicsPipeline)
{
    VkGraphicsPipelineCreateInfo vgpci = *pPipelineCreateInfo;
    return result;
}
```

Uses for Stencil Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_STENCIL_OP_REPLACE</td>
<td>Replace stencil value with the reference value.</td>
</tr>
<tr>
<td>VK_STENCIL_OP_KEEP</td>
<td>Keep the stencil value as it is.</td>
</tr>
<tr>
<td>VK_STENCIL_OP_ZERO</td>
<td>Set the stencil value to 0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_STENCIL_OP_REPLACE</td>
<td>Replace stencil value with the reference value.</td>
</tr>
<tr>
<td>VK_STENCIL_OP_KEEP</td>
<td>Keep the stencil value as it is.</td>
</tr>
<tr>
<td>VK_STENCIL_OP_ZERO</td>
<td>Set the stencil value to 0.</td>
</tr>
</tbody>
</table>

Polygon edges without Z-fighting

```
vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```
Vulkan: a More Typical (and Simplified) Block Diagram

- **Application**
  - **Instance**
  - **Physical Device**
    - **Logical Device**
    - **Queue**
  - **Command Buffer**

Vulkan Queues and Command Buffers

- Graphic commands are recorded in command buffers, e.g., `vkCmdDrawSomething();`
- 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 already has them
- Each Queue belongs to a Queue Family
- We don’t create Queue Families – the Physical Device already has them
- Command Buffers can be filled by different CPU Threads
- You can have as many simultaneous Command Buffers as you want
- Graphics commands are recorded in command buffers, e.g.,

```c
return -1;
```

```c
for (unsigned int i = 0; i < count; i++)
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
uint32_t count = -1;
```

Similarly, we can write a function that finds the proper Queue Family

```c
uint32_t count;
void *queueFamilies = NULL;
uint32_t queueCount = 0;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
```

Querying what Queue Families are Available

```c
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT &vqfp, );
```

Creating a Logical Device Queue Needs to Know Queue Family Information

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

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

```c
VkResult
vkCreateCommandPool( LogicalDevice, IN &vsci, PALLOCATOR, OUT &CommandPool );
```

Creating the Command Buffers

```c
vkBeginCommandBuffer( commandBuffer, IN &vcbbi );
```

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

```c
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdBlitImage( commandBuffer, filter );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdBeginQuery( commandBuffer, flags );
vkCmdCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
```
VkResult offsets[1] = { 0 };

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

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

vkCmdEndRenderPass

vkCmdBindPushConstants

vkCmdSetViewport

vkCmdBindDescriptorSets

vkCmdBindPushConstants

vkCmdBindDescriptorSets

The Entire Submission / Wait / Display Process

Submit the queue

Fill in the queue information

Submit the queue

Get the queue

Create fence

Wait for the fence

Submitting a Command Buffer to a Queue for Execution
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")

We Need to Find Out What our Display Capabilities Are

VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "vkGetPhysicalDeviceSurfaceCapabilitiesKHR:
" );
.
VkBool32 supported;
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics( ), Surface, &supported );
if( supported == VK_TRUE )
fprintf( FpDebug, "** This Surface is supported by the Graphics Queue **
" );
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
fprintf( FpDebug, "Found %d Surface Formats:
", formatCount )
.
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "Found %d Present Modes:
", presentModeCount )
.

This is a pretty good analogy, except that there can be many more images in the ring buffer than are being shown here.
We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

Creating a Swap Chain

Creating the Swap Chain Images and Image Views

Rendering into the Swap Chain, I

Rendering into the Swap Chain, II
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );

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 );
Textures

9/19/2018

The Basic Idea

Texture mapping is a computer graphics operation in which a separate image, referred to as the texture, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a texture map. This can be any image. At one time, some graphics hardware required the image's pixel dimensions to be a power of two. This restriction has been lifted on most (all?) graphics cards, but just to be safe—The X and Y dimensions did not need to be the same power of two. So, a 128x32 image would have been OK; a 129x511 image might not have been.

Also, to prevent confusion, the texture pixels are not called pixels. A pixel is a dot in the final screen image. A dot in the texture image is called a texture element, or texel. Similarly, to avoid terminology confusion, a texture's width and height dimensions are not called X and Y. They are called S and T. A texture map is not generally indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always $S=0$, the right side is $S=1$, the bottom is $T=0$, and the top is $T=1$. Thus, you do not need to be aware of the texture's resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.

In OpenGL terms: assigning an (s,t) to each vertex

Enable texture mapping:

```cpp
glEnable(GL_TEXTURE_2D);
```

Draw your polygons, specifying a and t at each vertex:

```cpp
gBegin(GL_POLYGON);
gTexCoord2f(s0, t0);
gNormal3f(nx0, ny0, nz0);
gVertex3f(x0, y0, z0);
gTexCoord2f(s1, t1);
gNormal3f(nx1, ny1, nz1);
gVertex3f(x1, y1, z1);
```

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

Triangles in an Array of Structures

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

$$s = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \quad t = \frac{y - y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}$$
Using a Texture: How do you know what \((s,t)\) to assign to each vertex?

Or, for a sphere,

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

From the Sphere code:

\[ s = \frac{\text{lng} + \pi/2}{\pi} \]
\[ t = \frac{\text{lat} + \pi/2}{\pi} \]

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

Be careful where \(s\) abruptly transitions from 1. back to 0.

Memory Types

You create your texture here

Host Visible
GPU Memory

Texture
Sampling

GPU Memory

Device Local
GPU Memory

RGBA to the Shader
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 left over 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.

Textures’ Undersampling Artifacts

Texels

Pixels

Average 4 pixels to make a new one

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_ADDRESS_MODE_LINEAR.

* Latin: mult in parvo, “many things in a small place”
// *******************************************************************************
// this second {...} is to create the actual texture image:
// transition the texture buffer layout:
{
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

    // initializes the image
    ve3.depth = 1;
    ve3.height = texHeight;
    ve3.width = texWidth;

    ve3.offset.x = 0;
    ve3.offset.y = 0;
    ve3.offset.z = 0;

    ve3.extent.width = texWidth;
    ve3.extent.height = texHeight;
    ve3.extent.depth = 1;

    vsl.layerCount = 1;
    vsl.baseArrayLayer = 0;
    vsl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

    // because we want to mmap it
    ve3.offset.x = 0;
    ve3.offset.y = 0;
    ve3.offset.z = 0;

    ve3.extent.width = texWidth;
    ve3.extent.height = texHeight;
    ve3.extent.depth = 1;

    vsl.layerCount = 1;
    vsl.baseArrayLayer = 0;
    vsl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, IN &vsl, OUT &vmr);

    ve3.extent.height = texHeight;
    ve3.extent.width = texWidth;
    ve3.extent.depth = 1;

    vsl.layerCount = 1;
    vsl.baseArrayLayer = 0;
    vsl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

    // because we are transferring into it and will eventual sample from it
    visr.baseArrayLayer = 0;
    visr.baseMipLevel = 0;
    visr.layerCount = 1;
    visr.levelCount = 1;

    VkImageSubresourceRange visr = {
        .baseSubresource = 0,
        .layerCount = 1,
        .levelCount = 1,
        .aspectMask = VK_IMAGE_ASPECT_COLOR_BIT,
    };

    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, IN &vis, OUT &vsl);
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

    vkBindImageMemory(LogicalDevice, stagingImage, IN &vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);

    vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

    vimb.subresourceRange = visr;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.srcAccessMask = 0;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_TRANSFER_BIT;

    VkBufferMemoryBarrier vbbi;
    vbbi.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
    vbbi.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vbbi.srcAccessMask = vkAccessMask(vdm);
    vbbi.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vbbi.dstAccessMask = 0;

    VkImageMemoryBarrier imb;
    imb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    imb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    imb.srcAccessMask = 0;
    imb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    imb.dstAccessMask = VK_ACCESS_TRANSFER_BIT;

    VkMemoryBarrier memb;
    memb.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
    memb.srcAccessMask = VK_ACCESS_TRANSFER_BIT;
    memb.dstAccessMask = VK_ACCESS_TRANSFER_BIT;

    VkPipelineStageFlags vss = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    VkPipelineStageFlags vs = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

    TextureImage imbTextureImage = { lock, (PALLOCATOR) },
    TextureCommandBuffer vbcb = { lock, (PALLOCATOR) },
    TextureCameraBuffer vcbb = { lock, (PALLOCATOR) },
    TextureFacade tf = { lock, (PALLOCATOR) };

    vbcbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    vbcbi.pNext = nullptr;

    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    vcbbi.pNext = nullptr;

    TextureFacade tf = { lock, (PALLOCATOR) };

    // map the actual image to the staging image
    vkMapMemory(LogicalDevice, stagingImage, IN &vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

    // reset the staging image
    VkCommandBufferInheritanceInfo vcbi;
    vcbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO;
    vcbi.pNext = nullptr;
    vcbi.srcCommandBuffer = (TextureCommandBuffer *)nullptr;
    vcbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

    // because we are transferring into it and will eventual sample from it
    visr.baseArrayLayer = 0;
    visr.baseMipLevel = 0;
    visr.layerCount = 1;
    visr.levelCount = 1;

    VkImageSubresourceRange visr = {
        .baseSubresource = 0,
        .layerCount = 1,
        .levelCount = 1,
        .aspectMask = VK_IMAGE_ASPECT_COLOR_BIT,
    };

    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, IN &vis, OUT &vsl);
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

    vkBindImageMemory(LogicalDevice, stagingImage, IN &vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);

    vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

    vimb.subresourceRange = visr;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.srcAccessMask = 0;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstAccessMask = 0;

    VkBufferMemoryBarrier vbbi;
    vbbi.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
    vbbi.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vbbi.srcAccessMask = 0;
    vbbi.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vbbi.dstAccessMask = 0;

    VkImageMemoryBarrier imb;
    imb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    imb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    imb.srcAccessMask = 0;
    imb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    imb.dstAccessMask = VK_ACCESS_TRANSFER_BIT;

    VkMemoryBarrier memb;
    memb.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
    memb.srcAccessMask = VK_ACCESS_TRANSFER_BIT;
    memb.dstAccessMask = VK_ACCESS_TRANSFER_BIT;

    VkPipelineStageFlags vss = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    VkPipelineStageFlags vs = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

    TextureImage imbTextureImage = { lock, (PALLOCATOR) },
    TextureCommandBuffer vbcb = { lock, (PALLOCATOR) },
    TextureCameraBuffer vcbb = { lock, (PALLOCATOR) },
    TextureFacade tf = { lock, (PALLOCATOR) };

    vbcbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    vbcbi.pNext = nullptr;

    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    vcbbi.pNext = nullptr;

    TextureFacade tf = { lock, (PALLOCATOR) };

    // map the actual image to the staging image
    vkMapMemory(LogicalDevice, stagingImage, IN &vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

    // reset the staging image
    VkCommandBufferInheritanceInfo vcbi;
    vcbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO;
    vcbi.pNext = nullptr;
    vcbi.srcCommandBuffer = (TextureCommandBuffer *)nullptr;
    vcbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
Reading in a Texture from a BMP File

```cpp
// Read in a texture from a BMP file.
// (puppy.bmp

typedef struct MyTexture
{
    uint32_t width;
    uint32_t height;
    VkSampler texSampler;
    VkDevices texDevice;
} MyTexture;

MyTexture MyPuppyTexture;

result = Init06TextureSampler( &MyPuppyTexture.texSampler );
```

This function can be found in the `sample.cpp` file. The BMP file needs to be created by something that writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick's convert, Adobe Photoshop, or GNU's GIMP.

https://en.wikipedia.org/wiki/Anisotropic_filtering

Note that, at this point, the CPU buffer and the GPU Staging Buffer are no longer needed, and can be destroyed.

Vulkan: Overall Block Diagram

```
Vulkan: Overall Block Diagram
```

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

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
Vulkan: a More Typical (and Simplified) Block Diagram

Vulkan: Identifying the Physical Devices

Vulkan: Asking About the Physical Device’s Features

Vulkan: Querying the Number of Physical Devices

Which Physical Device to Use, I

Here’s What the NVIDIA 1080ti Produced
### Here's What the Intel HD Graphics 520 Produced

```c
vkEnumeratePhysicalDevices:
Device 0:
  API version: 4194360
  Driver version: 4194360
  Vendor ID: 0x8086
  Device ID: 0x1916
  Physical Device Type: 1 = (Integrated GPU)
  Device Name: Intel(R) HD Graphics 520
  Pipeline Cache Size: 213

Device #0 selected ('Intel(R) HD Graphics 520')
Physical Device Features:
  geometryShader = 1
  tessellationShader = 1
  multiDrawIndirect = 1
  wideLines = 1
  largePoints = 1
  multiViewport = 1
  occlusionQueryPrecise = 1
  pipelineStatisticsQuery = 1
  shaderFloat64 = 1
  shaderInt64 = 1
  shaderInt16 = 1
```

### Which Physical Device to Use, II

```c
vkGetPhysicalDeviceMemoryProperties:

```Memory Types:
```Memory 0:  DeviceLocal
Memory 1:  HostVisible HostCoherent
Memory 2:  HostVisible HostCoherent HostCached

```Memory Heaps:
```Heap 0:  size = 0xb7c00000 DeviceLocal
Heap 1:  size = 0xfac00000

```Asking About the Physical Device's Queue Families

```
```

### Here's What I Got

```
```

### Asking About the Physical Device's Queue Families

```
```

### Here's What I Got

```
```

```c
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, &count, nullptr);

for( unsigned int i = 0; i < count; i++ )
{

```
Vulkan: Overall Block Diagram

Instance

Application

Physical Device

Logical Device

Command Buffer

Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Command Buffer

Looking to See What Device Layers are Available

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

const char * myDeviceExtensions[] = {
    VK_KHR_surface,
    VK_KHR_win32_surface
};

// see what device layers are available:
uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);

VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, deviceLayers);

Looking to See What Device Extensions are Available

// 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, deviceExtensions);

What Device Layers and Extensions are Available

2 physical device layers enumerated:

0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus'

0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker'

0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation'
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;

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

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

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

Layers and Extensions

Layers are code that can be installed between the Application and Vulkan. Normally, Vulkan is meant to run "flat out". Layers can take the extra time to perform useful functions like printing debugging messages, printing function calls, etc. They are not always necessary, but when you need them, you will be really glad they are there!
int integratedSelect = -1;
int discreteSelect = -1;
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );
delete[] physicalDevices;

else if( integratedSelect >= 0 )

else if( discreteSelect >= 0 )

result = vkCreateInstance( IN &vici, PALLOCATOR, OUT &Instance );

// need some logical here to decide which physical device to select:
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp);
VkPhysicalDeviceProperties vpdp;

PhysicalDevice = physicalDevices[which];
which = integratedSelect;
PhysicalDevice = physicalDevices[which];
which = discreteSelect;

vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
uint32_t count = -1;

vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp);
VkPhysicalDeviceMemoryProperties vpdmp;

vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp );
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );

vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x0000000f 'VK_EXT_debug_report'
0x00000001 'VK_KHR_external_memory_capabilities'
0x00000001 'VK_KHR_external_semaphore_capabilities'
0x00000001 'VK_KHR_external_fence_capabilities'
0x00000001 'VK_KHR_device_group_creation'
0x00000006 'VK_KHR_win32_surface'
0x00000019 'VK_KHR_surface'
0x00000001 'VK_KHR_get_surface_capabilities2'
0x00000001 'VK_KHR_get_physical_device_properties2'
0x00000001 'VK_EXT_display_surface_counter'
0x00000001 'VK_KHR_get_physical_device_properties2'

looking to see what extensions are both wanted and available:

for( uint32_t wanted = 0; wanted < numExtensionsWanted; wanted++ )

for( uint32_t available = 0; available < numExtensionsAvailable; available++ )

if( strcmp( instanceExtensions[wanted], InstanceExtensions[available].extensionName ) == 0 )

break;

extensionsWantedAndAvailable.push_back( InstanceExtensions[available].extensionName );

vici.ppEnabledExtensionNames = extensionsWantedAndAvailable.data();;
vici.enabledExtensionCount = extensionsWantedAndAvailable.size();
vici.ppEnabledLayerNames = instanceLayers;
vici.enabledLayerCount = sizeof(instanceLayers) / sizeof(char *);
vici.pApplicationInfo = &vai;
vici.flags = 0;
vici.pNext = nullptr;
vici.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;

Will now ask for 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report

looking to see what extensions are both wanted and available:

Will now ask for 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report

looking to see what extensions are both wanted and available:

Will now ask for 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report
```c
VkResult result;
float queuePriorities[NUM_QUEUES_WANTED] = {
    1.0f};
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];
vdqci[0].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
v dqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
v dqci[0].queueCount = 1; // how many queues to create
vdqci[0].pQueuePriorities = queuePriorities; // array of queue priorities [0.0f, 1.0f]

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

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

uint32_t layerCount;
vkEnumerateDeviceLayerProperties(pDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(pDevice, &layerCount, deviceLayers);

for (unsigned int i = 0; i < layerCount; i++)
{
    // see what device extensions are available:
    uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(pDevice, deviceLayers[i].layerName, &extensionCount,
                                         (VkExtensionProperties *)nullptr);
    VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
    result = vkEnumerateDeviceExtensionProperties(pDevice, deviceLayers[i].layerName, &extensionCount,
                                                  deviceExtensions);
    delete[] deviceLayers;
}
```
### 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.

![Vulkan Highlights: Overall Block Diagram](image)

#### Creating a Semaphore

```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
// vsci.pNext = nullptr;  // Optional next structure
// vsci.flags = 0;  // Not used in this example

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

#### Semaphores Example during the Render Loop

```c
// ... Use imageReadySemaphore...

VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
// vsci.pNext = nullptr;  // Optional next structure
// vsci.flags = 0;  // Not used in this example

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

uint32_t nextImageIndex;
result = vkAcquireNextImageKHR( LogicalDevice, SwapChain, UINT64_MAX, imageReadySemaphore, VK_NULL_HANDLE, &nextImageIndex );

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkSubmitInfo vsi;
// vci.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;  // Both a structure and array look valid, but only an array looks valid
// vci.pNext = nullptr;  // Optional next structure
// vci.waitSemaphoreCount = 1;  // Optional: waits for one semaphore
vci.waitSemaphoreCount = 1;
// vci.pWaitSemaphores = &imageReadySemaphore;  // Only one semaphore
vci.pWaitSemaphores = &imageReadySemaphore;
// vci.pWaitDstStageMask = &waitAtBottom;  // Optional:
vci.pWaitDstStageMask = &waitAtBottom;

vsi.commandBufferCount = 1;
// vci.pCommandBuffers = CommandBuffers[nextImageIndex];  // Optional:
vsi.pCommandBuffers = CommandBuffers[nextImageIndex];
// vci.signalSemaphoreCount = 0;
// vci.pSignalSemaphores = (VkSemaphore) nullptr;

result = vkQueueSubmit( presentQueue, 1, &vsi, renderFence );
```

#### 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.

```c
#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

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

// 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, VK_TRUE, timeout );
// timeout is a uint64_t timeout in nanoseconds (could be 0, which means to return immediately)
// timeout can be up to UINT64_MAX = 0xffffffffffffffff (= 580+ years)
// result = VK_SUCCESS means it returned because a fence (or all fences) signaled
// result = VK_TIMEOUT means it returned because the timeout was exceeded
```

---

57
**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

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

---

**Controlling Events from the Device**

result = vkCmdSetEvent( CommandBuffer, IN event, pipelineStageBits, srcPipelineStageBits, dstPipelineStageBits );
result = vkCmdResetEvent( CommandBuffer, IN event, pipelineStageBits, srcPipelineStageBits, dstPipelineStageBits );
result = vkCmdSetEvent( CommandBuffer, IN event, pipelineStageBits, nullptr, nullptr );
result = vkCmdResetEvent( CommandBuffer, IN event, nullptr, nullptr, nullptr );

---

**From the Command Buffer Notes:**

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

- Can enter these commands in any order
- In some commands, the GPU is blocked threatening race conditions

---

**Pipeline Barriers:**

A case of Gate-ing and Wait-ing

---

**Pipeline Barriers:**

A case of Gate-ing and Wait-ing

---

**Fence Example**

```cpp
VkQueue presentQueue;
VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

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

---

**Controlling Events from the Host**

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

result = vkCreateEvent( LogicalDevice, IN &veci, PALLOCATOR, OUT &event );

vsi.signalSemaphoreCount = 0;

vsi.pNext = nullptr;
```

---

**Event Example:**

```cpp
VkSubmitInfo vsi;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.commandBufferCount = 1;
vsi.pWaitDstStageMask = &waitAtBottom;
```

---

**Note:** The CPU cannot block waiting for an event, but it can test for one.
4. There are connections from these sensors to the traffic lights so that when the first car in the src group enters its intersection, the dst traffic light will be turned red.
5. When the last car in the src group completely makes it through its intersection, the dst traffic light can be turned back to green.
6. The Vulkan command pipeline ordering is this: (1) the src cars get released, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released (which end up being stopped by a red light somewhere), (4) the src cars clear their intersection, (5) the dst cars get released.
Example: Be sure we are done writing an output image before using it for something else.

The Scenario:

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

The Scenario:

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

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

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:

```glsl
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:

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

where:
- stageFlags are one or more 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.

### VkImageLayout – How an Image gets Laid Out in Memory

VkImageLayout – How an Image gets Laid Out in Memory depends on how it will be Used

Here, the use of vkCmdPipelineBarrier() is to simply change the layout of an image to a new set of VK_IMAGE_LAYOUT_ flags.

#### Access types

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

#### Stages

- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
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);
```

```c
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;
```

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

### An Robotic Example using Push Constants

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

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

struct armArm1;
struct armArm2;
struct armArm3;
```

Where each arm is represented by:

- armMatrix
- armColor
- armScale

Forward Kinematics:

- Hook the Pieces Together, Change Parameters, and Things Move (All Young Children Understand This)

- Given the Lengths and Angles, Where do the Pieces Move To?
Positioning Part #1 With Respect to Ground
1. Rotate by $\Theta_1$
2. Translate by $T_{U/G}$

Write it
$$[M_{1/G}] = [T_{1/G}] \cdot [R_{\Theta_1}]$$

Say it

Positioning Part #2 With Respect to Ground
1. Rotate by $\Theta_2$
2. Translate the length of part 1
3. Rotate by $\Theta_1$
4. Translate by $T_{U/G}$

Write it
$$[M_{2/G}] = [T_{1/G}] \cdot [R_{\Theta_1}] \cdot [T_{2/G}] \cdot [R_{\Theta_2}]$$
$$[M_{2/G}] = [M_{1/G}] \cdot [M_{2/1}]$$

Say it

Positioning Part #3 With Respect to Ground
1. Rotate by $\Theta_3$
2. Translate the length of part 2
3. Rotate by $\Theta_2$
4. Translate the length of part 1
5. Rotate by $\Theta_1$
6. Translate by $T_{U/G}$

Write it
$$[M_{3/G}] = [T_{1/G}] \cdot [R_{\Theta_3}] \cdot [T_{2/G}] \cdot [R_{\Theta_2}] \cdot [T_{3/G}] \cdot [R_{\Theta_1}]$$
$$[M_{3/G}] = [M_{1/G}] \cdot [M_{2/1}] \cdot [M_{3/2}]$$

Say it

In the Reset Function
```
struct arm          Arm1;
struct arm          Arm2;
struct arm          Arm3;
...                

Arm1.armMatrix = glm::mat4();
Arm1.armColor  = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale  = 6.f;
Arm2.armMatrix = glm::mat4();
Arm2.armColor  = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale  = 4.f;
Arm3.armMatrix = glm::mat4();
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
```
VkPipelineLayoutCreateInfo vplci;
    .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    .pNext = nullptr;
    .flags = 0;
    .setLayoutCount = 4;
    .pSetLayouts = DescriptorSetLayouts;
    .pushConstantRangeCount = 1;
    .pPushConstantRanges = vpcr;

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```
In the UpdateScene Function

```c
float rot1 = (float)Time;
float rot2 = 2 * rot1;
float rot3 = 2 * 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);

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);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));

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

In the RenderScene Function

```c
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

In the Vertex Shader

```c
layout( push_constant ) uniform arm {
    mat4 armMatrix;
    vec3 armColor;
    float armScale;  // scale factor in x
} RobotArm;
layout( location = 0 ) in vec3 aVertex;

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

Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
"Aliasing" is a signal-processing term for "under-sampled compared with the frequencies in the signal".

What the signal really is: what we want

Sampling Interval

Sampled Points

What we think the signal is: too often, what we get

"The Nyquist [sampling] rate is twice the maximum component frequency of the function [i.e., signal] being sampled." -- WikiPedia

MultiSampling

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

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

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

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.
Vulkan Distribution of Sampling Points within a Pixel

Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

Multisampling

Setting up the Image

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

VkGraphicsPipelineCreateInfo vgpci;
vpmci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vpmci.pNext = nullptr;

. . .
vpmci.pMultisampleState = &vpmsci;

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

Final Pixel Color = \( \sum \) Color sample from subpixel.

# Fragment Shader calls = 8

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

(8,1) produces partial supersampling

1. Produces complete supersampling
## Setting up the Image

```c
VkAttachmentDescription vad[2];

vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[0].flags = 0;

vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
vad[1].flags = 0;
```

**Color Reference**
```
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

**Depth Reference**
```
depthReference.attachment = 1;
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

## Resolving the Image:

### Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

```c
V1Offset3D 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;
visl.extent = ve3;

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

## 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.

![Diagram of Subpass, Attachment, and Framebuffer](diag.png)
Back in Our Single-pass Days, I

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

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

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

Multipass, I

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.

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

Multipass Algorithm to Render and then Image Process

Original | Sharpened | Edge Detected

No Noise | No Noise | No Noise

Original Sharpened Edge Detected
Placing a Pipeline Barrier so an Image is not used before it is Ready

```
vkCmdPipelineBarrier(TextureCommandBuffer, VkImageMemoryBarrier
vimb
vimb.subresourceRange = visr;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vimb.image = textureImage;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vimb.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Dynamic State Variables

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

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

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

Creating a Pipeline with Dynamically Changeable State Variables
Which Pipeline State Variables can be Changed Dynamically

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

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

Creating a Pipeline

```cpp
VkDynamicState vds[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};
VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState);
vpdsci.pDynamicStates = &vds;

VkGraphicsPipelineCreateInfo vgpci;

vgpci.pDynamicState = &vpdsci;

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

Filling State Variables in the Command Buffer

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

- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports );`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors );`
- `vkCmdSetLineWidth(commandBuffer, linewidth );`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );`
- `vkCmdSetBlendConstants(commandBuffer, blendConstants[4] );`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds );`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask );`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask );`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference );`

Setting up Query Pools

There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp

Vulkan requires you to first setup “Query Pools”, one for each specific type.

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

**Pipeline Statistics Query**

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

### Resetting, Filling, and Examining a Query Pool

**Resetting, Filling, and Examining a Query Pool**

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

Here is how you create a much-simpler Compute Pipeline

This is a Particle System application, so we need Positions, Velocities, and (possibly) Colors

A Reminder about Data Buffers

Creating a Shader Storage Buffer
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

```c
VkMemoryRequirements
vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );

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

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );

result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 );

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );
```

And, since we have Data Buffers, we will need Descriptor Sets to Create the Pipeline Layout

```c
#define BUFFER_SLOT_BINDING 0
#define BUFFER_SLOT_COUNT 1
#define BUFFER_PADDING 0

VkDescriptorSetLayoutCreateInfo
vdslc.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vkCreateDescriptorSetLayout( LogicalDevice, &vdslc, PALLOCATOR, OUT &DescriptorLayout );

VkPipelineLayoutCreateInfo
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &PipelineLayout );
```

The Particle System Compute Shader -- Setup

```c
#define POSITION_BINDING 0
#define VELOCITY_BINDING 1
#define COLOR_BINDING 2

VkDescriptorSetLayoutCreateInfo
vdslc.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vkCreateDescriptorSetLayout( LogicalDevice, &vdslc, PALLOCATOR, OUT &DescriptorLayout );

VkPipelineLayoutCreateInfo
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &PipelineLayout );
```
\begin{verbatim}
const VECTOR G = VECTOR(0., -9.8, 0.);
const float DT = 0.1;
const SPHERE Sphere = vec4(-100., -800., 0., 600.); // x, y, z, r

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POIINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;

POINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;

if( IsInsideSphere(pp, Sphere) )
{
    vp = BounceSphere(p, v, S);
    pp = p + vp*DT + 0.5*DT*DT*G;
}

Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;
\end{verbatim}
A Specialization Constant is a way of injecting an integer or Boolean constant into an .spv-compiled version of a shader right before the final compilation. That final compilation happens when you call `vkCreateComputePipelines()` without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago.

In the compute shader:

```c
layout( constant_id = 0 ) const int numXworkItems = 32;
layout( local_size_x = numXworkItems, local_size_y = 1, local_size_z = 1 ) in;

VkSpecializationMapEntry vsme[1]; // one array element for each Specialization Constant
vsme.constantID = 0;
vsme.offset = 0; // # bytes into the Specialization Constant array this one item is
vsme.size = sizeof(int); // size of just this Specialization Constant

int numXworkItems = 64;

VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(int); // size of all the Specialization Constants together
vsii.pData = &numXworkItems; // array of all the Specialization Constants
```

In the C/C++ program:

```c
VkSpecializationMapEntry vsme[1];
vsme.constantID = 0;
vsme.offset = 0;
vsme.size = sizeof(int);

int numXworkItems;

VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(int);
vsii.pData = &numXworkItems;
```

In the compute shader:

```c
int numXworkItems = 64;
```

In the C/C++ program:

```c
int numXworkItems = 64;
```

In the compute shader:

```c
result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vpssci[0], &pAllocator, &ComputePipeline);```