Course Goals

- Give a sense of how Vulkan is different from OpenGL
- Show how to do basic drawing in Vulkan
- Leave you with working, documented, understandable sample code

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

Welcome! I'm happy to be here. I hope you are too!
Introduction

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Top Three Reasons that Prompted the Development of 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.
Who's Been Specifically Working on Vulkan?

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-vertex-end, fixed-function, etc.
- You must manage your own transformations.
- All transformation, color and texture functionality must be done in shaders.
- Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the runtime pipeline-building process.

Vulkan Highlights: Pipeline State Data Structure

- In OpenGL, your “pipeline state” is the combination of 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 variables at once into a “pipeline state object” (PSO) data structure and then invoke the entire PSO at once 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 pipeline state objects

Vulkan: Creating a Pipeline


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

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

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

---

**Vulkan Code has a Distinct “Style” of Setting Information in structs and then Passing that Information as a pointer-to-the-struct**

```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, IN &vbci, PALLOCATOR, OUT &Buffer );
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );      // fills vmr
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = 0;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```

---

**Vulkan Quick Reference Card – I Recommend you Print This!**

Steps in Creating Graphics using Vulkan

1. Create the Vulkan 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 Framebuffer(s)
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- …

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Sample Program Output

Sample Program Keyboard Inputs

'I', 'L': Toggle lighting off and on
'm', 'M': Toggle display mode (textures vs. colors, for now)
'p', 'P': Pause the animation
'q', 'Q': quit the program
Esc: quit the program
'y', 'R': Toggle rotation-animation and using the mouse

'l', '1', '4', '9' Set the number of instances (in the instancing version)

Caveats on the Sample Code, I

1. I've written everything out in appalling longhand.
2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything easily.
3. At times, I could have hidden complexity, but I didn't. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that's kept a secret from you.
4. I've setup Vulkan structs every time they are used, even though, in many cases (most?), they could have been setup once and then re-used each time.
5. At times, I've setup things that didn't need to be setup just to show you what could go there.

Caveats on the Sample Code, II

6. There are great uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'ed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure. I hope it is clearer this way.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.
Main Program

```c
int main( int argc, char * argv[] )
{
    Width = 800;
    Height = 600;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE);
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d\n", Width, Height);
    Reset( );
    InitGraphics( );
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents( );
        Time = glfwGetTime( );
        UpdateScene( ); RenderScene( );
        fprintf(FpDebug, "Closing the GLFW window\n");
        vkQueueWaitIdle( Queue );
        vkDeviceWaitIdle( LogicalDevice );
        DestroyAllVulkan( );
        glfwDestroyWindow( MainWindow );
        glfwTerminate( );
        return 0;
    }
}
```

Vulkan Conventions

- `Vk`Xxx is a typedef, probably a struct
- `vk`Yyy( ) is a function call
- `VK`_ZZZ 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 function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are actually #define’d to nothing.

Struct `errorcode`

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

Error Codes:

- `VK_NOT_READY`, "Not Ready"
- `VK_TIMEOUT`, "Timeout"
- `VK_EVENT_SET`, "Event Set"
- `VK_EVENT_RESET`, "Event Reset"
- `VK_ERROR_OUT_OF_HOST_MEMORY`, "Out of Host Memory"
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`, "Out of Device Memory"
- `VK_ERROR_INITIALIZATION_FAILED`, "Initialization Failed"
- `VK_ERROR_DEVICE_LOST`, "Device Lost"
- `VK_ERROR_MEMORY_MAP_FAILED`, "Memory Map Failed"
- `VK_ERROR_LAYER_NOT_PRESENT`, "Layer Not Present"
- `VK_ERROR_EXTENSION_NOT_PRESENT`, "Extension Not Present"
- `VK_ERROR_FEATURE_NOT_PRESENT`, "Feature Not Present"
- `VK_ERROR_INCOMPATIBLE_DRIVER`, "Incompatible Driver"
- `VK_ERROR_TOO_MANY_OBJECTS`, "Too Many Objects"
- `VK_ERROR_FORMAT_NOT_SUPPORTED`, "Format Not Supported"
- `VK_ERROR_FRAGMENTED_POOL`, "Fragmented Pool"
- `VK_ERROR_SURFACE_LOST_KHR`, "Surface Lost"
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`, "Native Window in Use"
- `VK_SUBOPTIMAL_KHR`, "Suboptimal"
- `VK_ERROR_OUT_OF_DATE_KHR`, "Out of Date"
- `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`, "Incompatible Display"
- `VK_ERROR_INITIALIZATION_FAILED_KHR`, "Initialization Failed"
- `VK_ERROR_INCOMPATIBLE_DRIVER_KHR`, "Incompatible Driver"
- `VK_ERROR_OUT_OF_POOL_MEMORY_KHR`, "Out of Pool Memory"
- `VK_ERROR_INCOMPATIBLE_EXTERNAL_HANDLE_KHR`, "Incompatible External Handle"
void PrintVkError( VkResult result, std::string prefix ){
    if (Verbose && result == VK_SUCCESS){
        fprintf(FpDebug, "%s: %s
", prefix.c_str(), "Successful");
        fflush(FpDebug);
        return;
    }
    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = "";
    for( int i = 0; i < numErrorCodes; i++ ){
        if ( result == ErrorCodes[i].resultCode){
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }
    fprintf(FpDebug, "%s: %s
", prefix.c_str(), meaning.c_str());
    fflush(FpDebug);
}

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

bool Paused;
bool Verbose;
#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
const int32_t OFFSET_ZERO = 0;

Vulkan Topologies

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

Extras in the Code

-Warns
- HERE_I_AM
- DEBUGFILE

Reporting Error Results, II
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;
struct vertex VertexData[] = {
    ...
};
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer(sizeOf(VertexData), OUT &MyVertexDataBuffer);
Fill05DataBuffer(MyVertexDataBuffer, (void *)VertexData);

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

Telling the Pipeline about its Input

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

C/C++:

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

GLSL Shader:

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

Always use the C/C++ construct `offsetof`, rather than hardcoding the value!
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 vertex input.

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

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 &GraphicsPipeline );
```

Telling the Command Buffer what Vertices to Draw

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

```c
VkBuffer buffers[1] = {MyVertexDataBuffer.buffer};
vkCmdBindVertexBuffer(CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets);
collection size[1] = {sizeof(JustVertexData[0])};
const uint32_t vertexCount = sizeof(JustVertexData[0]) / sizeof(JustVertexData[0].x);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDrawCommandBuffers(nextImageIndex), vertexCount, instanceCount, firstVertex, firstInstance;
```

Drawing with an Index Buffer

Stream of Vertices

<table>
<thead>
<tr>
<th>Vertex 7</th>
<th>Vertex 5</th>
<th>Vertex 4</th>
<th>Vertex 3</th>
<th>Vertex 1</th>
<th>Vertex 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1, -1, -1,</td>
<td>-1, -1, -1,</td>
<td>-1, -1, -1,</td>
<td>-1, -1, -1,</td>
<td>-1, -1, -1,</td>
<td>-1, -1, -1,</td>
</tr>
</tbody>
</table>

Stream of Indices

<table>
<thead>
<tr>
<th>Vertex Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Triangles Draw

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

<table>
<thead>
<tr>
<th>Triangle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, -2, 1</td>
</tr>
<tr>
<td>-3, -4, 5</td>
</tr>
<tr>
<td>0, 1, -2</td>
</tr>
</tbody>
</table>
Drawing with an Index Buffer

```c
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
```

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

```c
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```

---

Sometimes the Same Point Needs Multiple Attributes

```
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), IN &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
Init05MyIndexDataBuffer( sizeof(JustIndexData), IN &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );
```

```
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;
```

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

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 indexed buffer drawing, you need to create a new vertex struct if any of {position, normal, color, texCoords} changes from what was previously-stored at those coordinates.
Sometimes the Same Point Needs Multiple Attributes

Where values match at the corners (color)

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

The Shaders’ View of the Basic Computer Graphics Pipeline

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

Vulkan Shader Stages

```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;
```
How Vulkan GLSL Differs from OpenGL GLSL

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

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

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

**OpenGL uses:**
- `gl_VertexID`
- `gl_InstanceID`

---

**Vulkan Shader Compiling**

- You half-precompile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V, which stands for Standard Portable Intermediate Representation.
- SPIR-V gets turned into fully-compiled code at runtime, when the pipeline structure is finally created
- The SPIR-V spec has been public for a few years – new shader languages are surely being developed
- OpenGL and OpenCL have now adopted SPIR-V as well

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

---

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

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

**Descriptor Sets:**
```glsl
layout( set=0, binding=0 ) . . .
```

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

- Only for scalars, but a vector’s components can be constructed from specialization constants
- Specialization Constants for Compute Shaders:
  ```glsl
  layout( local_size_x_id = 8, local_size_y_id = 16 );
  ```
  - This sets `gl_WorkGroupSize.x` and `gl_WorkGroupSize.y`
  - `gl_WorkGroupSize.z` is set as a constant

**Push Constants:**
```glsl
layout( push_constant ) . . .
```
SPIR-V:
Standard Portable Intermediate Representation for Vulkan

`glslangValidator shaderFile -V [-I <dir>] [-S <stage>] -o shaderBinaryFile.spv`

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

(Can be overridden by the -S option)

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

Windows: `glslangValidator.exe`
Linux: `glslangValidator`

Running `glslangValidator.exe`

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?

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

Specify the output file

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

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

Running `glslangValidator.exe`

MINGW64 /y/Vulkan/Example2017

```
 MinhD3pooh MINGW64 /y/Vulkan/Example2017
 % $ 185
 glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
 sample-vert.vert

 MinhD3pooh MINGW64 /y/Vulkan/Example2017
 % $ 185
 glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
 sample-frag.frag

 MinhD3pooh MINGW64 /y/Vulkan/Example2017
 % $ 185
```
# define SPIRV_MAGIC 0x07230203

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 );
    VkShaderModule ShaderModuleVertex;
    VkShaderModuleCreateInfo vsmci;
    vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    vsmci.pNext = nullptr;
    vsmci.flags = 0; vsmci.codeSize = size; vsmci.pCode = (uint32_t *)code;
    VkResult result = vkCreateShaderModule(LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex);
    fprintf(FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );
    delete [] code;
    return result;
}

You can also take a look at SPIR-V Assembly

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

This prints out the SPIR-V "assembly" to standard output. Other than nerd interest, there is no graphics-programming reason to look at this. ☺
For example, if this is your Shader Source

```glsl
#version 400

// Non-opaque must be in a uniform block;
// input, gl_PerVertex, set 1, binding 0; union significant

layout( std140, set = 0, binding = 0 ) uniform matBuf {
  MemberName 13(matBuf) 0  "uModelMatrix" MemberName 13(matBuf) 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix"
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
}

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 0 ColMajor MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 2 ColMajor MemberDecorate 13(matBuf) 2 Offset 128

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

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) 3 MatrixStride 16

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

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

This is the SPIR-V Assembly, Part I

```spirv
#version 400

// Non-opaque must be in a uniform block;
// input, gl_PerVertex, set 1, binding 0; union significant

layout( std140, set = 0, binding = 0 ) uniform matBuf {
  MemberName 13(matBuf) 0  "uModelMatrix" MemberName 13(matBuf) 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix"
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
}

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 0 ColMajor MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 2 ColMajor MemberDecorate 13(matBuf) 2 Offset 128

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

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) 3 MatrixStride 16

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

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

This is the SPIR-V Assembly, Part II

```spirv
// Non-opaque must be in a uniform block;
// input, gl_PerVertex, set 1, binding 0; union significant

layout( std140, set = 0, binding = 0 ) uniform matBuf {
  MemberName 13(matBuf) 0  "uModelMatrix" MemberName 13(matBuf) 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix"
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
}

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 0 ColMajor MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 2 ColMajor MemberDecorate 13(matBuf) 2 Offset 128

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

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) 3 MatrixStride 16

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

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

This is the SPIR-V Assembly, Part III

```spirv
// Non-opaque must be in a uniform block;
// input, gl_PerVertex, set 1, binding 0; union significant

layout( std140, set = 0, binding = 0 ) uniform matBuf {
  MemberName 13(matBuf) 0  "uModelMatrix" MemberName 13(matBuf) 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix"
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
}

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 0 ColMajor MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 2 ColMajor MemberDecorate 13(matBuf) 2 Offset 128

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

// non-opaque must be in a uniform block:
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) 3 MatrixStride 16

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

void
main( )
{
  mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
  gl_Position = PVM * vec4( aVertex, 1.);
  vNormal = Matrices.uNormalMatrix * aNormal;
  vColor = aColor;
  vTexCoord = aTexCoord;
}
```
A Google-Wrapped Version of glslangValidator

The shaderc project from Google (https://github.com/google/shaderc) provides a glslangValidator wrapper program called \texttt{glslc} that has a much improved command-line interface. You use, basically, the same way:

\texttt{glslc.exe \--target-env=vulkan \ sample-vert.vert \o sample-vert.spv}

There are several really nice features. The two I really like are:

1. You can \texttt{#include} files into your shader source
2. You can \texttt{#define} definitions on the command line like this:
   
   \texttt{glslc.exe \--target-env=vulkan \-DNUMPONTS=4 \ sample-vert.vert \o sample-vert.spv}

\texttt{glslc} is included in your Sample .zip file

Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

From the Quick Reference Card

A Vulkan 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 sometimes have taken to calling these things “Data Buffers” and have even gone to far as to override some of Vulkan’s own terminology:

\texttt{typedef VkBuffer VkDataBuffer;}

This is probably a bad idea in the long run.
Creating and Filling Vulkan Data Buffers

Creating a Vulkan Data Buffer

Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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;
}
```

Finding the Right Type of Memory

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

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

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. I don’t have experience using it (yet), so I’m not in a position to confidently comment on it. But, I am including its github link here and a little sample code in case you want to take a peek.

https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

This repository includes a smattering of documentation.

```c
#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"

VkBufferCreateInfo vbci;
VmaAllocationCreateInfo vaci;

vaci.physicalDevice = PhysicalDevice;
vaci.device = LogicalDevice;
vaci.usage = VMA_MEMORY_USAGE_GPU_ONLY;

VmaAllocator var;
vmaCreateAllocator( IN &vaci, OUT &var );

VkBuffer Buffer;
VmaAllocation van;
vmaCreateBuffer( IN var, IN &vbci, IN &vaci, OUT &Buffer, OUT &van, nullptr );

void *mappedDataAddr;
vmaMapMemory( IN var, IN van, OUT &mappedDataAddr );
memcpy( mappedDataAddr, &MyData, sizeof(MyData) );
vmaUnmapMemory( IN var, IN van );
```
### Something I’ve Found Useful

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

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

```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.

It also makes it impossible to accidentally associate the wrong 

VkDeviceMemory and/or VkDeviceSize with the wrong data buffer.

### Initializing a Data Buffer

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

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

### Here’s a C struct used by the Sample Code to hold some uniform variables

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

Here’s the associated GLSL shader code to access those uniform variables

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

### Filling those Uniform Variables

```c
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle in radians
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( 1. );              // 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.; // account for Vulkan’s LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

This code assumes that this line:

```c
#define GLM_FORCE_RADIANS
```

is listed before GLM is included!
This C struct is holding the original data, written by the application.

```c
struct matBuf Matrices;
```

The MyBuffer does not hold any actual data itself. It just information about what is in the data buffer.

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

```c
uniform matBuf Matrices;
```

---

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

```c
VkResult Init05UniformBuffer( size_t size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );  // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```

```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.
Setting Up GLFW

```c
#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"
...
uint32_t Width, Height;
VkSurfaceKHR Surface;
...
void
InitGLFW()
{
    glfwInit();
    if( !glfwVulkanSupported() )
    {
        fprintf(stderr, "Vulkan is not supported on this system!
");
        exit(1);
    }
    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, OUT(Surface));
    glfwSetErrorCallback(
        GLFWErrorCallback);
    glfwSetKeyCallback( MainWindow,
        GLFWKeyboard);
    glfwSetCursorPosCallback( MainWindow,
        GLFWMouseMotion);
    glfwSetMouseButtonCallback( MainWindow,
        GLFWMouseButton);}
```

You Can Also Query What Vulkan Extensions GLFW Requires

```c
uint32_t count;
const char ** extensions =
    glfwGetRequiredInstanceExtensions(&count);
for( uint32_t i = 0; i < count; i++ )
{
    fprintf(FpDebug, ": %s
", extensions[i]);
}
```

```
Found 2 GLFW Required Instance Extensions:
VK_KHR_surface
VK_KHR_win32_surface
```

GLFW Keyboard Callback

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

```
```
### GLFW Mouse Button Callback

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

### GLFW Mouse Motion Callback

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

### Looping and Closing GLFW

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

VkQueueWaitIdle(Queue);
VkDeviceWaitIdle(LogicalDevice);
DestroyVulkan();
glfwDestroyWindow(MainWindow);
glfwTerminate();
```

If you would like to block waiting for events, use:

```c
glfwWaitEvents();
```

You can have the blocking wake up after a timeout period with:

```c
glfwWaitEventsTimeout(double secs);
```

You can wake up one of these blocks from another thread with:

```c
glfwPostEmptyEvent();
```
GLM

Mike Bailey
mjb@cs.oregonstate.edu

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.

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:

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

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

Why are we even talking about this?

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

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
gluLookAt( 0., 0., 3., 0., 0., 0., 0., 1., 0. );
glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );

you would now say:

glm::mat4 modelview = glm::mat4( 1. ); // identity
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::scale( modelview, glm::vec3(Scale,Scale,Scale) );

This is exactly the same concept as OpenGL, but a different expression of it. Read on for details …

The Most Useful GLM Variables, Operations, and Functions

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

// constructor:
glm::mat4 t1; // identity matrix
glm::vec3 c;

// multiplications:

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

// emulating OpenGL transformations with concatenation:

glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
The Most Useful GLM Variables, Operations, and Functions

// viewing volume (assign, not concatenate):
glm::mat4 glm::ortho(float left, float right, float bottom, float top, float near, float far);
glm::mat4 glm::ortho(float left, float right, float bottom, float top);

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

Installing GLM into your own space

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

Installing GLM into your own space

Here's what that GLM folder looks like

GLM in the Vulkan sample.cpp Program

if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    glm::mat4 Matrices.uModelMatrix = glm::mat4( 1. );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0. ) );
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0. ) );
    // done this way, the Scale is applied first, then the Xrot, then the Yrot
}
else
{
    if( ! Paused )
    {
        const glm::vec3 axis = glm::vec3( 0., 1., 0. );
        Matrices.uModelMatrix = glm::rotate( glm::mat4( 1. ), (float)glm::radians( 360.f*Time/SECONDS_PER_CYCLE ), axis );
    }
    glm::vec3 eye(0.,0.,EYEDIST );
    glm::vec3 look(0.,0.,0.);
    glm::vec3 up(0.,1.,0.);
    Matrices.uVewMatrix = glm::lookAt( eye, look, up );
    Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1f, 1000.f );
    Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL
    Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
    // note: inverseTransform!
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Misc.uTime = (float)Time;
    Misc.uMode = Mode;
    Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
Sidebar: Why Isn’t The Normal Matrix exactly the same as the Model Matrix?

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

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

Wrong!

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

Right!

Original object and normal

Instancing

Mike Bailey
mjb@cs.oregonstate.edu

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

Instancing – What and why?

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

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

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

BTW, when not using instancing, be sure the instanceCount is 1, not 0!

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:

out vec3 vColor;
const int NUMINSTANCES = 16;
const float DELTA = 3.0;
float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vColor = vec3(1., float((1.+gl_InstanceIndex) / float(NUMINSTANCES)), 0.);

xdelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.;
ydelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.;
vec4 vertex = vec4( aVertex.xyz + vec3( xdelta, ydelta, 0. ), 1.);

gl_Position = PVM * vertex;  // [p][v][m]
Put the unique characteristics in a uniform buffer array and reference them

Still uses gl_InstanceIndex

In the vertex shader:

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

out vec3 vColor;

int index = gl_InstanceIndex % 1024; // or "& 1023" – gives 0 - 1023
vColor = Colors.uColors[ index ];
```

```
vec4 vertex = ...

PVM * vertex; // [p][v][m]
```

What is the Vulkan Graphics Pipeline?

Here’s what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context". It is a data structure.

2. The Vulkan Graphics Pipeline is not the processes that OpenGL would call "the graphics pipeline".

3. For the most part, the Vulkan Graphics Pipeline Data Structure is 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.

There is also a Vulkan Compute Pipeline Data Structure.
A Pipeline Data Structure Contains the Following State Items:

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

**Bold/Italics** indicates that this state item can also be set with Dynamic State Variables
Creating a Typical Graphics Pipeline

VkResult
Init14GraphicsVertexFragmentPipeline(VkShaderModule vertexShader, VkShaderModule fragmentShader, 
  VKPipelineTopology topology, OUT VkPipeline *pGraphicsPipeline)
{
    VkPipelineShaderStageCreateInfo vpssci[2];
    vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[0].pNext = nullptr;
    vpssci[0].flags = 0;
    vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
    vpssci[1].pNext = nullptr;
    vpssci[1].flags = 0;
    vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    vpssci[1].module = fragmentShader;
    vpssci[1].pName = "main";
    vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

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

    Link in the Per-Vertex Attributes

    VKVertexInputAttributeDescription vviad[3];
    // an array containing one of these per buffer being used
    vviad[0].location = 0;                      // location in the layout
    vviad[0].binding = 0;                       // which binding # this is
    vviad[0].stride = sizeof( struct vertex );  // bytes between successive
    vviad[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
    vviad[1].location = 1;                      // nx, ny, nz
    vviad[1].binding = 0;                       // which binding # this is
    vviad[1].format = VK_FORMAT_VEC3;          // nx, ny, nz
    vviad[1].offset = offsetof( struct vertex, normal );                    // 12
    vviad[2].location = 2;                      // r, g, b
    vviad[2].binding = 0;                       // which binding # this is
    vviad[2].format = VK_FORMAT_VEC3;          // nx, ny, nz
    vviad[2].offset = offsetof( struct vertex, color );                       // 24

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

    Use one vpvisci array member per shader module you are using

    Use one vptsci array member per vertex
    topology, and always uses the settings in red above.
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.

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

If your **VkIndexType** is **VK_INDEX_TYPE_UINT16**, then the special index is `0xffff`.

If your **VkIndexType** is **VK_INDEX_TYPE_UINT32**, it is `0xffffffff`.

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

Declare the viewport information

Declare the scissoring information

Group the viewport and scissoring information together.
What is the Difference Between Changing the Viewport and Changing the Scissoring?

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

What is “Depth Clamp Enable”?  

```
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:**

![Original Image](image1)

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

![Original Image](image2)

The gray area shows what would happen with depthClampEnable (except it would have been red).

Setting the Rasterizer State

```
VkPipelineRasterizationStateCreateInfo
vprsci;

vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;

vprsci.pNext = nullptr;

vprsci.flags = 0;

vprsci.depthClampEnable = VK_FALSE;

vprsci.rasterizerDiscardEnable = VK_FALSE;

vprsci.polygonMode = VK_POLYGON_MODE_FILL;

#ifdef CHOICES
VK_POLYGON_MODE_LINE
VK_POLYGON_MODE_POINT
#endif

vprsci.cullMode = VK_CULL_MODE_NONE;  // recommend this because of the projMatrix[1][1] *= -1;

#ifdef CHOICES
VK_CULL_MODE_NONE
VK_CULL_MODE_FRONT_BIT
VK_CULL_MODE_BACK_BIT
VK_CULL_MODE_FRONT_AND_BACK_BIT
#endif

vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;

#ifdef CHOICES
VK_FRONT_FACE_COUNTER_CLOCKWISE
VK_FRONT_FACE_CLOCKWISE
#endif

vprsci.depthBiasEnable = VK_FALSE;

vprsci.depthBiasConstantFactor = 0.f;

vprsci.depthBiasClamp = 0.f;

vprsci.depthBiasSlopeFactor = 0.f;

vprsci.lineWidth = 1.f;
```

What is “Depth Bias Enable”?

```
vprsci.depthBiasEnable = VK_FALSE;
```

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

![Z-fighting](image3)
Declare information about how the multisampling will take place.

*MultiSampling State*

We will discuss MultiSampling in a separate noteset.

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.

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

\[
\text{Color}_{\text{new}} = (1 - \alpha) \times \text{Color}_{\text{existing}} + \alpha \times \text{Color}_{\text{incoming}}
\]

\[0 \leq \alpha \leq 1.\]

*A “Color Attachment” is a framebuffer to be rendered into. You can have as many of these as you want.*

Which Pipeline Variables can be Set Dynamically

Just used as an example in the Sample Code

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

Here’s how the Stencil Buffer works:

1. While drawing into the Render Buffer, you can write values into the Stencil Buffer at the same time.
2. While drawing into the Render Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.
3. When drawing into the Render Buffer, you can write-protect certain parts of the Render Buffer based on values that are in the Stencil Buffer.

Using the Stencil Buffer to Create a Magic Lens

1. Clear the SB = 0
2. Write protect the color buffer
3. Fill a square, setting SB = 1
4. Write-enable the color buffer
5. Draw the solids wherever SB == 0
6. Draw the wireframes wherever SB == 1

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting
Using the Stencil Buffer to Better Outline Polygons

Clear the SB = 0

for( each polygon )
{
    Draw the edges, setting SB = 1
    Draw the polygon wherever SB != 1
    Draw the edges, setting SB = 0
}

Using the Stencil Buffer to Perform Hidden Line Removal

```cpp
VkStencilOpState vsosf; // front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds

// front
vsosf.compareOp = VK_COMPARE_OP_NEVER;

// back
vsosf.compareOp = VK_COMPARE_OP_NEVER;
```

Stencil Operations for Front and Back Faces

```cpp
// front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds

// back
vsosf.compareOp = VK_COMPARE_OP_NEVER;
```
vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
Descriptor Sets

Mike Bailey
mjb@cs.oregonstate.edu

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

In OpenGL

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

Each uniform variable gets updated one-at-a-time.

 Wouldn’t it be nice if we could update a collection of related uniform variables all at once, without having to update the uniform variables that are not related to this collection?

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

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

• Related uniform variables can be updated as a group, gaining efficiency.

• Descriptor Sets are activated when the Command Buffer is filled. Different values for the uniform buffer variables can be toggled by just swapping out the Descriptor Set that points to GPU memory, rather than re-writing the GPU memory.

• Values for the shaders’ uniform buffer variables can be compartmentalized into what quantities change often and what change seldom (scene-level, model-level, draw-level), so that uniform variables need to be re-written no more often than is necessary.

for( each scene )
{
    Bind Descriptor Set #0
    for( each object )
    {
        Bind Descriptor Set #1
        for( each draw )
        {
            Bind Descriptor Set #2
            Do the drawing
        }
    }
}

Our example will assume the following shader uniform variables:

```c
// non-opaque must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
} Matrices;

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

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

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```

What are Descriptor Sets?

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

• Related uniform variables can be updated as a group, gaining efficiency.

• Descriptor Sets are activated when the Command Buffer is filled. Different values for the uniform buffer variables can be toggled by just swapping out the Descriptor Set that points to GPU memory, rather than re-writing the GPU memory.

• Values for the shaders’ uniform buffer variables can be compartmentalized into what quantities change often and what change seldom (scene-level, model-level, draw-level), so that uniform variables need to be re-written no more often than is necessary.

for( each scene )
{
    Bind Descriptor Set #0
    for( each object )
    {
        Bind Descriptor Set #1
        for( each draw )
        {
            Bind Descriptor Set #2
            Do the drawing
        }
    }
}
Descriptor Sets

CPU:
- Uniform data created in a C++ data structure
- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size
- Doesn't know where each piece of data starts
- Doesn't know the CPU or GPU data structure

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

GPU:
- Uniform data used in the shader
- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size
- Doesn't know the CPU or GPU data structure

* "binary large object"

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

struct lightBuf
{
  glm::vec4 uLightPos;
};

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

```c
layout( std140, set = 0, binding = 0 ) uniform matBuf Matrices;
layout( std140, set = 1, binding = 0 ) uniform lightBuf Light;
layout( std140, set = 2, binding = 0 ) uniform miscBuf Misc;
layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```

Step 1: Descriptor Set Pools

You don't allocate Descriptor Sets on the fly -- that is too slow. Instead, you allocate a "pool" of Descriptor Sets and then pull from that pool later.

```c
result = vkCreateDescriptorPool( LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
```

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 2: Define the Descriptor Set Layouts

```c
VkResult Init13DescriptorSetLayouts()
{
    VkResult result;

    // DS #1: Matrix Set
    VkDescriptorSetLayoutBinding MatrixSet[1];
    MatrixSet[0].binding = 0;
    MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MatrixSet[0].descriptorCount = 1;
    MatrixSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    result = vkCreateDescriptorSetLayout(LogicalDevice, &MatrixSet[0], PALLOCATOR, &DescriptorSetLayouts[0]);

    // DS #2: Light Set
    VkDescriptorSetLayoutBinding LightSet[1];
    LightSet[0].binding = 0;
    LightSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    LightSet[0].descriptorCount = 1;
    LightSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    result = vkCreateDescriptorSetLayout(LogicalDevice, &LightSet[0], PALLOCATOR, &DescriptorSetLayouts[1]);

    // DS #3: Misc Set
    VkDescriptorSetLayoutBinding MiscSet[1];
    MiscSet[0].binding = 0;
    MiscSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MiscSet[0].descriptorCount = 1;
    MiscSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    result = vkCreateDescriptorSetLayout(LogicalDevice, &MiscSet[0], PALLOCATOR, &DescriptorSetLayouts[2]);

    // DS #4: Tex Sampler Set
    VkDescriptorSetLayoutBinding TexSamplerSet[1];
    TexSamplerSet[0].binding = 0;
    TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    TexSamplerSet[0].descriptorCount = 1;
    TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
    result = vkCreateDescriptorSetLayout(LogicalDevice, &TexSamplerSet[0], PALLOCATOR, &DescriptorSetLayouts[3]);

    return result;
}
```

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

```c
VkResult Init14GraphicsPipelineLayout()
{
    VkResult result;

    VkDescriptorSetLayoutCreateInfo vdslc0;
    vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc0.pNext = nullptr;
    vdslc0.flags = 0;
    vdslc0.bindingCount = 1;
    vdslc0.pBindings = &MatrixSet[0];
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc0, PALLOCATOR, &DescriptorSetLayouts[0]);

    VkDescriptorSetLayoutCreateInfo vdslc1;
    vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc1.pNext = nullptr;
    vdslc1.flags = 0;
    vdslc1.bindingCount = 1;
    vdslc1.pBindings = &LightSet[0];
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc1, PALLOCATOR, &DescriptorSetLayouts[1]);

    VkDescriptorSetLayoutCreateInfo vdslc2;
    vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc2.pNext = nullptr;
    vdslc2.flags = 0;
    vdslc2.bindingCount = 1;
    vdslc2.pBindings = &MiscSet[0];
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc2, PALLOCATOR, &DescriptorSetLayouts[2]);

    VkDescriptorSetLayoutCreateInfo vdslc3;
    vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc3.pNext = nullptr;
    vdslc3.flags = 0;
    vdslc3.bindingCount = 1;
    vdslc3.pBindings = &TexSamplerSet[0];
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc3, PALLOCATOR, &DescriptorSetLayouts[3]);

    return result;
}
```
Step 4: Allocating the Memory for Descriptor Sets

vkAllocateDescriptorSets()

- DescriptorSetLayouts
- VkDescriptorSetAllocateInfo
  - descriptorPool
  - descriptorSetCount
  - descriptorSetLayouts

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

- VkDescriptorBufferInfo
  - vdbi0
    - buffer = MyMatrixUniformBuffer.buffer
    - offset = 0
    - range = sizeof(Matrices)
  - vdbi1
    - buffer = MyLightUniformBuffer.buffer
    - offset = 0
    - range = sizeof(Light)
  - vdbi2
    - buffer = MyMiscUniformBuffer.buffer
    - offset = 0
    - range = sizeof(Misc)

- VkDescriptorImageInfo
  - vdii0
    - sampler = MyPuppyTexture.texSampler
    - imageView = MyPuppyTexture.texImageView
    - imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

- VkWriteDescriptorSet
  - vwds0
    - dstSet = DescriptorSets[0]
    - dstBinding = 0
    - descriptorCount = 1
    - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    - pBufferInfo = &vdbi0
  - vwds1
    - dstSet = DescriptorSets[1]
    - dstBinding = 0
    - descriptorCount = 1
    - descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    - pBufferInfo = &vdbi1

This struct identifies what buffer it owns and how big it is.
This struct identifies what texture sampler and image view it owns.
This struct links a Descriptor Set to the buffer it is pointing to.
**Step 5: Tell the Descriptor Sets where their data is**

- **VkWriteDescriptorSet**
  - `sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;`
  - `pNext = nullptr;`
  - `dstSet = DescriptorSets[2];`
  - `dstBinding = 0;`
  - `dstArrayElement = 0;`
  - `descriptorCount = 1;`
  - `descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;`
  - `pBufferInfo = &vdbi2;`
  - `pImageInfo = (VkDescriptorImageInfo *)nullptr;`
  - `pTexelBufferView = (VkBufferView *)nullptr;`

- **VkWriteDescriptorSet**
  - `sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;`
  - `pNext = nullptr;`
  - `dstSet = DescriptorSets[3];`
  - `dstBinding = 0;`
  - `dstArrayElement = 0;`
  - `descriptorCount = 1;`
  - `descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;`
  - `pBufferInfo = (VkDescriptorBufferInfo *)nullptr;`
  - `pImageInfo = &vdii0;`
  - `pTexelBufferView = (VkBufferView *)nullptr;`

- `uint32_t copyCount = 0;` // this could have been done with one call and an array of VkWriteDescriptorSets:

```
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr );
```

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

- **VkGraphicsPipelineCreateInfo**
  - `sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;`
  - `pNext = nullptr;`
  - `flags = 0;` // Flags
  - `stageCount = 2;` // number of stages in this pipeline
  - `pStages = vpssci;`
  - `pVertexInputState = &vpvisci;`
  - `pInputAssemblyState = &vpiasci;`
  - `pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;`
  - `pViewportState = &vpvsci;`
  - `pRasterizationState = &vprsci;`
  - `pMultisampleState = &vpmsci;`
  - `pDepthStencilState = &vpdssci;`
  - `pColorBlendState = &vpcbsci;`
  - `pDynamicState = &vpdsci;`
  - `layout = IN GraphicPipelineLayout;`
  - `renderPass = IN RenderPass;`
  - `subpass = 0;`
  - `basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;`
  - `basePipelineIndex = 0;`

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

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

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

So, the Pipeline Layout contains the structure of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

The pieces of the Pipeline Data Structure are fixed in size – with the exception of the Descriptor Sets and the Push Constants. Each of these two can be any size, depending on what you allocate for them. So, the Pipeline Data Structure needs to know how these two are configured before it can set its own total layout.

Think of the DS layout as being a particular-sized hole in the Pipeline Data Structure. Any data you have that matches this hole’s shape and size can be plugged in there.

The Pipeline Data Structure

Fixed Pipeline Elements

Specific Descriptor Set Layout

Any set of data that matches the Descriptor Set Layout can be plugged in there.

Textures

Mike Bailey
mjb@cs.oregonstate.edu

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

Triangles in an Array of Structures

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

struct vertex VertexData[3] =
{
    // triangle 0-2-3:
    // vertex #0:
    {
        { -1., -1., -1. },
        {  0.,  0., -1. },
        {  0.,  0.,  0. },
        {  1., 0. }
    },
    // vertex #2:
    {
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. }
    },
    // vertex #3:
    {
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        {  1.,  1.,  0. },
        {  0., 1. }
    }
};
GPU Memory Host Visible GPU Memory Device Local GPU Memory

Texture Sampling Hardware

memcpy() vkCmdCopyImage()

NVIDIA Discrete Graphics:
11 Memory Types:
Memory 0:
Memory 1:
Memory 2:
Memory 3:
Memory 4:
Memory 5:
Memory 6:
Memory 7: DeviceLocal
Memory 8: DeviceLocal
Memory 9: HostVisible HostCoherent HostCached
Memory 10: HostVisible HostCoherent HostCached

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

Textures' Undersampling Artifacts

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

Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all white, with the red texel having never been included in the final image. The solution is to create lower-resolution textures of the same texture so that the red texel gets included somehow in all resolution-level textures.
• 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 mipmap level, the rendering system can sample from two of them, one less that the T:P ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin: multum in parvo, "many things in a small place"
// ******************************************************************************
// to create the actual texture image:
// ******************************************************************************
{

VkImageCreateInfo vici;

vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;

vici.pNext = nullptr;

vici.flags = 0;

void * gpuMemory;

vici.imageType = VK_IMAGE_TYPE_2D;

vici.format = VK_FORMAT_R8G8B8A8_UNORM;

vkMapMemory
(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);

vici.extent.height = texHeight;

vici.extent.depth = 1;

if (vsl.rowPitch == 4 * texWidth)

vici.mipLevels = 1;

{

vici.arrayLayers = 1;

memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

vici.samples = VK_SAMPLE_COUNT_1_BIT;

{

vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

unsigned char *gpuBytes = (unsigned char *)gpuMemory;

for (unsigned int y = 0; y < texHeight; y++)

memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );

vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

vici.queueFamilyIndexCount = 0;

vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

vkUnmapMemory
(LogicalDevice, vdm);

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

VkMemoryRequirements vmr;

vkGetImageMemoryRequirements
(LogicalDevice, IN textureImage, OUT &vmr);

if( Verbose )

{

fprintf( FpDebug, "Texture vmr.size = %lld
", vmr.size );

fprintf( FpDebug, "Texture vmr.alignment = %lld
", vmr.alignment) ;

fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits );

fflush( FpDebug );

}

VkMemoryAllocateInfo vmai;

vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;

vmai.pNext = nullptr;

vmai.allocationSize = vmr.size;

vmai.memoryTypeIndex =
);

VkDeviceMemory vdm;

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

result =
vkBindImageMemory
(LogicalDevice, IN textureImage, IN vdm, 0 );

// copy pixels from the staging image to the texture:

// transition the texture buffer layout:// ******************************************************************************

VkCommandBufferBeginInfo vcbbi;

{

VkImageSubresourceRange visr;

vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

vcbbi.pNext = nullptr;

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

visr.layerCount = 1;

VkImageMemoryBarrier vimb;

result =
vkBeginCommandBuffer
(TextureCommandBuffer, IN &vcbbi);

vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;vimb.pNext = nullptr;vimb.oldLayout = ... = VK_QUEUE_FAMILY_IGNORED;vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;vimb.image = textureImage;

vimb.srcAccessMask = 0;

vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;vimb.subresourceRange = visr;

// transition the staging buffer layout:// ******************************************************************************

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

// now do the final image transfer:

VkImageMemoryBarrier vimb;

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

} // because we want to sample from it

// transition the staging buffer layout:// ******************************************************************************

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

// transition the texture buffer layout:// ******************************************************************************

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

} // because we want to sample from it

} // because we want to sample from it
VkImageCopy vic;
vic.srcSubresource = visl; vic.srcOffset = vo3; vic.dstSubresource = visl; vic.dstOffset = vo3; vic.extent = ve3;

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

// *******************************************************************************

// transition the texture buffer layout a second time:// *******************************************************************************
{
    VkImageSubresourceRange visr;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT; visr.baseMipLevel = 0; visr.levelCount = 1; visr.baseArrayLayer = 0;
    visr.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER; vimb.pNext = nullptr; vimb.oldLayout = ... = VK_QUEUE_FAMILY_IGNORED; vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED; vimb.image = textureImage;
    vimb.srcAccessMask = 0; vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT; vimb.subresourceRange = visr;
    vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);
}

result = vkEndCommandBuffer( TextureCommandBuffer );
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO; vsi.pNext = nullptr; vsi.commandBufferCount = 1; vsi.pCommandBuffers = &TextureCommandBuffer; vsi.waitSemaphoreCount = 0;
vsii.pWaitSemaphores = (VkSemaphore *)nullptr;
vsii.signalSemaphoreCount = 0; vsii.pSignalSemaphores = (VkSemaphore *)nullptr; vsii.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
result = vkQueueSubmit( Queue, 1, IN &vsi, VK_NULL_HANDLE );
result = vkQueueWaitIdle( Queue );

// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT; visr.baseMipLevel = 0; visr.levelCount = 1; visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO; vivci.pNext = nullptr; vivci.flags = 0; vivci.image = textureImage; vivci.viewType = VK_IMAGE_VIEW_TYPE_2D; vivci.format = VK_FORMAT_R8G8B8A8_UNORM; vivci.components.r = VK_COMPONENT_SWIZZLE_R; vivci.components.g = VK_COMPONENT_SWIZZLE_G; vivci.components.b = VK_COMPONENT_SWIZZLE_B; vivci.components.a = VK_COMPONENT_SWIZZLE_A; vivci.subresourceRange = visr;
result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
return result;

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

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

Found 3 Queue Families:
0: Queue Family Count = 16; Graphics Compute Transfer
1: Queue Family Count = 1; Transfer
2: Queue Family Count = 8; Compute
Similarly, we can write a function that finds the proper queue family:

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

Similarly, we can create a logical device that needs to know queue family information:

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

Creating the command pool as part of the logical device:

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

Creating the command buffers:

```c
VkResult Init06CommandBuffers()
{
    VkResult result;
    VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.commandBufferCount = 2;
    result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
    return result;
}
```
Beginning a Command Buffer – One per Image

```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
imageReadySemaphore = vkCreateSemaphore( LogicalDevice, &vsci, &pAllocator, &imageReadySemaphore);

vkCommandBufferAllocateInfo cbinfo = {.
    sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO,
    pNext = nullptr,
    level = VK_COMMAND_BUFFER_LEVEL_PRIMARY,
    commandBufferCount = 1
};
result = vkCommandBufferAllocate( LogicalDevice, &cbinfo, &pAllocator, CommandBuffers);

for (uint32_t nextImageIndex = 0; nextImageIndex < imageReadySemaphore;)
{
    VkCommandBufferBeginInfo cbinfo = {.
        sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
        pNext = nullptr,
        flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT
    };
    cbinfo.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    CommandBuffers[1] = vkBeginCommandBuffer( CommandBuffers[1], &cbinfo);
    // These are the Commands that could be entered into the Command Buffer, I
    // ...
    vkCmdBeginRenderPass( commandBuffer, const contents );
    vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
    vkCmdBindIndexBuffer( commandBuffer, indexType );
    vkCmdBindPipeline( commandBuffer, pipeline );
    vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
    vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
    ...
    vkCmdEndRenderPass( commandBuffer );
    CommandBuffers[1] = vkEndCommandBuffer( CommandBuffers[1]);
    vkQueueSubmit( LogicalDevice, 1, &submitInfo, VK_NULL_HANDLE);
    
    nextImageIndex = vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX, VK_NULL_HANDLE, OUT &&imageReadySemaphore, OUT &nextImageIndex);
}
```
VkResult RenderScene()
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

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

    uint32_t nextImageIndex;
    vkAcquireNextImageKHR(
        LogicalDevice, SwapChain, UINT64_MAX, VK_NULL_HANDLE,
        VK_NULL_HANDLE, OUT &nextImageIndex);

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

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

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

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

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

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

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

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

    VkViewport viewport = {
        0.,                     // x
        0.,                     // y
        (float)Width,          // width
        (float)Height,         // height
        0.,                     // minDepth
        1.                      // maxDepth
    };

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

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

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

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

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

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

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

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

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

    vkCmdEndRenderPass(
        CommandBuffers[nextImageIndex]);

    vkEndCommandBuffer(
        CommandBuffers[nextImageIndex]);
}

Submitting a Command Buffer to a Queue for Execution
The Entire Submission / Wait / Display Process

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

“Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences.”

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there.

But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.
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

VulkanDebug.txt output:

```
vkGetPhysicalDeviceSurfaceCapabilitiesKHR:
  - minImageCount = 2 ; maxImageCount = 8
  - currentExtent = 1024 x 1024
  - maxImageExtent = 1024 x 1024
  - maxImageArrayLayers = 1
  - supportedTransforms = 0x0001
  - currentTransform = 0x0001
  - supportedCompositeAlpha = 0x0001
  - supportedUsageFlags = 0x009f

** This Surface is supported by the Graphics Queue **

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

Found 3 Present Modes:
  0:        2 ( VK_PRESENT_MODE_FIFO_KHR )
  1:        3 ( VK_PRESENT_MODE_FIFO_RELAXED_KHR )
  2:        1 ( VK_PRESENT_MODE_MAILBOX_KHR )
```
Creating a Swap Chain

vkCreateSwapchainKHR

VkSwapchainCreateInfo

surface
imageFormat
imageColorSpace
imageExtent
imageArrayLayers
imageUsage
imageSharingMode
preTransform
compositeAlpha
presentMode
clipped

vkGetDevicePhysicalSurfaceCapabilitiesKHR

VkSurfaceCapabilities

minImageCount
maxImageCount
currentExtent
minImageExtent
maxImageExtent
maxImageArrayLayers
supportedTransforms
currentTransform
supportedCompositeAlpha

vkGetSwapChainImagesKHR

vkCreateImageView

VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(PhysicalDevice, Surface, OUT &vsc);
surfaceRes = vsc.currentExtent;

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr; vscci.flags = 0;
vscci.surface = Surface; vscci.minImageCount = 2;
// double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.imageArrayLayers = 1;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.queueFamilyIndexCount = 0;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;

result = vkCreateSwapchainKHR(LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain);

Creating the Swap Chain Images and Image Views

uint32_t imageCount; // # of display buffers – 2? 3?
result = vkGetSwapchainImagesKHR(LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr);
PresentImages = new VkImage[imageCount];
result = vkGetSwapchainImagesKHR(LogicalDevice, SwapChain, OUT &imageCount, PresentImages);

// present views for the double-buffering:
PresentImageViews = new VkImageView[imageCount];
for(unsigned int i = 0; i < imageCount; i++)
{
VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vivci.subresourceRange.baseLayer = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.image = PresentImages[i];
result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
}

Rendering into the Swap Chain, I

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

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

uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);
result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
    vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
    vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
    vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
    vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
Push Constants

Mike Bailey
mjb@cs.oregonstate.edu

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.
Push Constants

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

```cpp
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:

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

where:

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

An Robotic Example using Push Constants

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

Where each arm is represented by:

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

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

```cpp
VkPushConstantRange
  vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
                      VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
  vpcr[0].offset = 0;
  vpcr[0].size = sizeof( glm::mat4 );

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

result = vkCreatePipelineLayout( LogicalDevice, &vplci, PALLOCATOR, &GraphicsPipelineLayout );
```
Forward Kinematics:
You Start with Separate Pieces, all Defined in their Own Local Coordinate System

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

Forward Kinematics:
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 $T_{1/G}$

$[M_{1/G}] = [T_{1/G}] \ast [R_{\theta_1}]$
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 $T_{1/G}$

Write it

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

\[
[M_{2/G}] = [M_{1/G}] \cdot [M_{2/1}]
\]

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 $T_{1/G}$

Write it

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

\[
[M_{3/G}] = [M_{1/G}] \cdot [M_{2/1}] \cdot [M_{3/2}]
\]

Say it

In the Reset Function

```c
struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
...
Arm1.armMatrix = glm::mat4(1,);
Arm1.armColor = glm::vec3(0.0f, 1.0f, 0.0f);
Arm1.armScale = 6.0f;
Arm2.armMatrix = glm::mat4(1,);
Arm2.armColor = glm::vec3(1.0f, 0.0f, 0.0f);
Arm2.armScale = 4.0f;
Arm3.armMatrix = glm::mat4(1,);
Arm3.armColor = glm::vec3(0.0f, 0.0f, 1.0f);
Arm3.armScale = 2.0f;
```

The constructor `glm::mat4(1,)` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`.

Setup the Push Constant for the Pipeline Structure

```c
VkPushConstantRange vpcr1;
vpcr1.stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vpcr1.offset = 0;
vpcr1.size = sizeof(struct arm);
...
VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = &vpcr1;
result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
```
In the UpdateScene Function

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

glm::vec3 zaxis = glm::vec3(0., 0., 1.);
glm::mat4 m1g = glm::mat4(1.); // identity
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis); // [T]*[R]

glm::mat4 m21 = glm::mat4(1.); // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis); // [T]*[R]
m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm

glm::mat4 m32 = glm::mat4(1.); // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis); // [T]*[R]
m32 = glm::translate(m32, glm::vec3(0., 0., 2.)); // z-offset from previous arm

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

In the RenderScene Function

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

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

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

The strategy is to draw each link using the same vertex buffer, but modified with a unique color, length, and matrix transformation

In the Vertex Shader

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

layout(location = 0) in vec3 aVertex;

... 

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

... 

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

Physical Devices

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Vulkan: Identifying the Physical Devices

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

Where to put them

How many total there are

To put them (use it):

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

Which Physical Device to Use, I

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

Device %2d:
", i );
    fprintf( FpDebug, "API version: %d
", vpdp.apiVersion );
    fprintf( FpDebug, "Driver version: %d
", vpdp.apiVersion );
    fprintf( FpDebug, "Vendor ID: 0x%04x
", vpdp.vendorID );
    fprintf( FpDebug, "Device ID: 0x%04x
", vpdp.deviceID );
    fprintf( FpDebug, "Physical Device Type: %d = " , vpdp.deviceType);
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        fprintf( FpDebug, "(Discrete GPU)
" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        fprintf( FpDebug, "(Integrated GPU)
" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
        fprintf( FpDebug, "(Virtual GPU)
" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
        fprintf( FpDebug, "(CPU)
" );
    fprintf( FpDebug, "Device Name: %s
", vpdp.deviceName );
    printf( FpDebug, "Pipeline Cache Size: %u
", vpdp.pipelineCacheSize);
}
```
Which Physical Device to Use, II

// need some logical here to decide which physical device to select:
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
  discreteSelect = i;
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
  integratedSelect = i;

int which = -1;
if( discreteSelect >= 0 )
  which = discreteSelect;
else if( integratedSelect >= 0 )
  which = integratedSelect;
else
  fprintf( FpDebug, "Could not select a Physical Device\n" );
return VK_SHOULD_EXIT;

Asking About the Physical Device's Features

VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );

fprintf( FpDebug, "Physical Device Features:\n" );
fprintf( FpDebug, "geometryShader = %2d\n", PhysicalDeviceFeatures.geometryShader);
fprintf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShader);
fprintf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect);
fprintf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines);
fprintf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints);
fprintf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport);
fprintf( FpDebug, "occlusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.occlusionQueryPrecise);
fprintf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStatisticsQuery);
fprintf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64);
fprintf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64);
fprintf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16);

Here's What the NVIDIA RTX 2080 Ti Produced

vkEnumeratePhysicalDevices:
Device 0:
  API version: 419849
  Driver version: 419849
  Vendor ID: 0x10de
  Device ID: 0xe04
  Physical Device Type: 2 (Discrete GPU)
  Device Name: RTX 2080 Ti
  Pipeline Cache Size: 206
Device #0 selected ('RTX 2080 Ti')
Physical Device Features:
geometryShader = 1
tessellationShader = 1
multiDrawIndirect = 1
wideLines = 1
largePoints = 1
multiViewport = 1
occlusionQueryPrecise = 1
pipelineStatisticsQuery = 1
shaderFloat64 = 1
shaderInt64 = 1
shaderInt16 = 1

Here's What the Intel HD Graphics 520 Produced

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

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

Asking About the Physical Device's Queue Families

Here's What I Got

11 Memory Types:
Memory 0:
Memory 1:
Memory 2:
Memory 3:
Memory 4:
Memory 5:
Memory 6:
Memory 7: DeviceLocal
Memory 8: DeviceLocal
Memory 9: HostVisible HostCoherent
Memory 10: HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0: size = 0xb7c00000 DeviceLocal
Heap 1: size = 0xfac00000

Here's What I Got

Found 3 Queue Families:
0: queueCount = 16   ; Graphics Compute Transfer
1: queueCount = 2   ; Transfer
2: queueCount = 8   ; Compute
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",
    "VK_EXT_debug_report",
    // "VK_KHR_swapchain"
};

// see what device layers are available:
uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, layerCount, deviceLayers);

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

4 physical device layers enumerated:
0x00401063 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x00401072 2 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00000001 'VK_EXT_validation_cache'
0x00000004 'VK_EXT_debug_marker'

0x00401072 1 'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00000001 'VK_EXT_validation_cache'
0x00000004 'VK_EXT_debug_marker'
0x00401072 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00000001 'VK_EXT_validation_cache'
0x00000004 'VK_EXT_debug_marker'
0x00401072 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
0x00000001 'VK_EXT_validation_cache'
0x00000004 'VK_EXT_debug_marker'