Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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

http://cs.oregonstate.edu/~mjb/vulkan
Welcome! I'm happy to be here. I hope you are too!

- Professor of Computer Science, Oregon State University
- Has been in computer graphics for over 30 years
- Has had over 8,000 students in his university classes
- mjb@cs.oregonstate.edu

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

1. Introduction
2. Sample Code
3. Drawing
4. Shaders and SPIR-V
5. Dats Buffers
6. GLFW
7. GLM
8. Instancing
9. Graphics Pipeline Data Structure
10. Descriptor Sets
11. Textures
12. Queues and Command Buffers
13. Swap Chain
14. Push Constants
15. Physical Devices
16. Logical Devices
17. Dynamic State Variables
18. Getting Information Back
19. Compute Shaders
20. Specialization Constants
21. Synchronization
22. Pipeline Barriers
23. Multisampling
24. Multipass
25. Ray Tracing
My Favorite Vulkan Reference

Introduction

Mike Bailey
mjb@cs.oregonstate.edu

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

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

Second, thanks to NVIDIA for all of their support!

Third, thanks to the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding them.)
2004: OpenGL 2.0 / GLSL 1.10 includes Vertex and Fragment Shaders

2008: OpenGL 3.0 / GLSL 1.30 adds features left out before

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

History of Shaders

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

2016: Vulkan 1.1

2020: Vulkan 1.2

There is lots more detail at:

https://en.wikipedia.org/wiki/Vulkan_(API)
Everything You Need to Know is Right Here … Somewhere 😊
Top Three Reasons that Promoted 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.
OpenGL 4.2 Pipeline Flowchart
**Why is it so important to keep the GPU Busy?**

### NVidia Titan V Specs vs. Titan Xp, 1080 Ti

<table>
<thead>
<tr>
<th></th>
<th>Titan V</th>
<th>Tesla V100</th>
<th>Tesla P100</th>
<th>GTX 1080 Ti</th>
<th>GTX 1060</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>GV100</td>
<td>GV100</td>
<td>GP100 Cut-Down Pascal</td>
<td>GP102 Pascal</td>
<td>GP104-400 Pascal</td>
</tr>
<tr>
<td><strong>Transistor Count</strong></td>
<td>21.1B</td>
<td>21.1B</td>
<td>15.3B</td>
<td>12B</td>
<td>7.2B</td>
</tr>
<tr>
<td><strong>Fab Process</strong></td>
<td>12nm FFN</td>
<td>12nm FFN</td>
<td>16nm FinFET</td>
<td>16nm FinFET</td>
<td>16nm FinFET</td>
</tr>
<tr>
<td><strong>CUDA Cores / Tensor Cores</strong></td>
<td>5120 / 640</td>
<td>5120 / 640</td>
<td>3584 / 0</td>
<td>3584 / 0</td>
<td>2560 / 0</td>
</tr>
<tr>
<td><strong>TMUs</strong></td>
<td>320</td>
<td>224</td>
<td>224</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td><strong>ROPs</strong></td>
<td>7</td>
<td>96 (7)</td>
<td>88</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td><strong>Core Clock</strong></td>
<td>1200MHz</td>
<td>1328MHz</td>
<td>-</td>
<td>1607MHz</td>
<td></td>
</tr>
<tr>
<td><strong>Boost Clock</strong></td>
<td>1455MHz</td>
<td>1370MHz</td>
<td>1480MHz</td>
<td>1600MHz</td>
<td>1733MHz</td>
</tr>
<tr>
<td><strong>FP32 TFLOPs</strong></td>
<td>15.6TFLOPs</td>
<td>14TFLOPs</td>
<td>10.6TFLOPs</td>
<td>-11.4TFLOPs</td>
<td>9TFLOPs</td>
</tr>
<tr>
<td><strong>Memory Type</strong></td>
<td>HBM2</td>
<td>HBM2</td>
<td>HBM2</td>
<td>GDDR5X</td>
<td>GDDR5X</td>
</tr>
<tr>
<td><strong>Memory Capacity</strong></td>
<td>12GB</td>
<td>16GB</td>
<td>16GB</td>
<td>11GB</td>
<td>8GB</td>
</tr>
<tr>
<td><strong>Memory Clock</strong></td>
<td>1.7Gbps HBM2</td>
<td>1.75Gbps HBM2</td>
<td>?</td>
<td>11Gbps</td>
<td>10Gbps GDDR5X</td>
</tr>
<tr>
<td><strong>Memory Interface</strong></td>
<td>3072-bit</td>
<td>4096-bit</td>
<td>4096-bit</td>
<td>352-bit</td>
<td>256-bit</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>653GB/s</td>
<td>900GB/s</td>
<td>-</td>
<td>~484GB/s</td>
<td>320.32GB/s</td>
</tr>
<tr>
<td><strong>Total Power Budget (“TDP”)</strong></td>
<td>250W</td>
<td>250W</td>
<td>300W</td>
<td>250W</td>
<td>180W</td>
</tr>
<tr>
<td><strong>Power Connectors</strong></td>
<td>1x 8-pin</td>
<td>1x 8-pin</td>
<td>?</td>
<td>1x 8-pin</td>
<td>1x 8-pin</td>
</tr>
<tr>
<td><strong>Release Date</strong></td>
<td>12/07/2017</td>
<td>4Q16-1Q17</td>
<td>TBD</td>
<td>5/27/2016</td>
<td></td>
</tr>
<tr>
<td><strong>Release Price</strong></td>
<td>$3000</td>
<td>$10000</td>
<td>-</td>
<td>$700</td>
<td>Reference: $700 MSRP, $600 Now: $500</td>
</tr>
</tbody>
</table>

The NVidia Titan V graphics card is not targeted at gamers, but rather at scientific and machine/deep learning applications. That does not, however, mean that the card is incapable of gaming, nor does it mean that we can't extrapolate future key performance metrics for Volta. The Titan V is a derivative of the earlier-released GV100 GPU, part of the Tesla accelerator card series. The key differentiator is that the Titan V ships at $3000, whereas the Tesla V100 was available as part of a $10,000 developer kit. The Tesla V100 still offers greater memory capacity by 4GB – 16GB HBM2 versus 12GB HBM2 – and has a wider memory interface, but other core features remain matched or nearly matched. Core count, for one, is 5120 CUDA cores on each GPU, with 640 Tensor cores (used for Tensorflow deep/machine learning workloads) on each GPU.
Who was the original Vulcan?

From WikiPedia:

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

Why Name it after the God of the Forge?
The Khronos Group, Inc. is a non-profit member-funded industry consortium, focused on the creation of open standard, royalty-free application programming interfaces (APIs) for authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. Khronos members may contribute to the development of Khronos API specifications, vote at various stages before public deployment, and accelerate delivery of their platforms and applications through early access to specification drafts and conformance tests.
Playing “Where’s Waldo” with Khronos Membership
Who's Been Specifically Working on Vulkan?
Vulkan

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

• More low-level information must be provided (by you!) in the application, rather than the driver

• Screen coordinate system is Y-down

• No “current state”, at least not one maintained by the driver

• All of the things that we have talked about being *deprecated* in OpenGL are *really deprecated* in Vulkan: built-in pipeline transformations, begin-end, fixed-function, etc.

• You must manage your own transformations.

• All transformation, color 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.
The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

MC → WC → EC → EC → Projection Transform

Model Transform → View Transform → Per-vertex Lighting

Framebuffer

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Shader-style

**Vertex Shader**
- **Per-vertex in variables**: `gl_Vertex, gl_Normal, gl_Color`
- **Uniform Variables**: `gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix`
- **Per-vertex out variables**: `gl_Position`

**Fragment Shader**
- **Per-fragment in variables**: `gl_FragColor`
- **Uniform Variables**: `gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix`
- **Per-fragment out variables**: Light Processing, Texturing, Per-fragment Lighting

**Rasterization**
- Output from Fragment Shader

**Framebuffer**
- Final output of the pipeline

**Notes**
- **MC** = Model Vertex Coordinates
- **WC** = World Vertex Coordinates
- **EC** = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Vulkan-style

Vertecl Shader

Fragment Shader

Framebuffer

Per-vertex in variables

Uniform Variables

gl_Position, Per-vertex out variables

Per-fragment in variables

Uniform Variables

Output color(s)
Moving part of the driver into the application

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets

Simpler drivers for low-overhead efficiency and cross vendor portability

Layered architecture so validation and debug layers can be unloaded when not needed

Run-time only has to ingest SPIR-V intermediate language

Unified API for mobile, desktop, console and embedded platforms

Khronos Group
Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- E.g., `vkCmdDoSomething(cmdBuffer, …);`
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed to Queues when the application wants them to be flushed
- Each command buffer can be filled from a different thread
Vulkan Highlights: Pipeline State Objects

- 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

VkGraphicsPipelineCreateInfo

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

VkSpecializationInfo

which stage (VERTEX, etc.)

VkShaderModule

binding
- stride
- inputRate

VkVertexInputBindingDescription

location
- binding
- format
- offset

VkVertexInputAttributeDescription

Viewports
- x, y, w, h,
- minDepth, maxDepth

 VkViewportsStateCreateInfo

Topology

Viewport

Scissor

x, y, w, h,
- offset
- extent

VkPipelineRasterizationStateCreateInfo

cullMode
- polygonMode
- frontFace
- lineWidth

VkPipelineInputAssemblyStateCreateInfo

VkPipelineColorBlendAttachmentState

VkPipelineColorBlendStateCreateInfo

blendEnable
- srcColorBlendFactor
dstColorBlendFactor
colorBlendOp
srcAlphaBlendFactor
dstAlphaBlendFactor
alphaBlendOp
colorWriteMask

depthTestEnable
- depthWriteEnable
depthCompareOp
stencilTestEnable
stencilOpStateFront
stencilOpStateBack

colorBlendEnable

VkPipelineDynamicStateCreateInfo

Array naming the states that can be set dynamically

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

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

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

result = vkEnumeratePhysicalDevices( Instance, &count, 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!

Vulkan Quick Reference Card

Vulkan Highlights: Overall Block Diagram
Vulkan Highlights: a More Typical Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Command Buffer

Command Buffer
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- …
Vulkan GPU Memory

• Your application allocates GPU memory for the objects it needs

• To write and read that GPU memory, you map that memory to the CPU address space

• Your application is responsible for making sure that what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.
Vulkan Render Passes

- Drawing is done inside a render pass
- Each render pass contains what framebuffer attachments to use
- Each render pass is told what to do when it begins and ends
Vulkan Compute Shaders

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

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

- GLSL is the same as before … almost

- For places it’s not, an implied
  \texttt{#define VULKAN 100}
  is automatically supplied by the compiler

- You pre-compile your shaders with an external compiler

- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)

- SPIR-V gets turned into fully-compiled code at runtime

- The SPIR-V spec has been public for years – new shader languages are surely being developed

- OpenCL and OpenGL have adopted SPIR-V as well

\begin{center}
\begin{tikzpicture}
  \node[draw,rectangle,minimum width=2cm,minimum height=1cm] (a) at (0,0) {GLSL Source};
  \node[draw,rectangle,minimum width=2cm,minimum height=1cm] (b) at (2,0) {External GLSL Compiler};
  \node[draw,rectangle,minimum width=2cm,minimum height=1cm] (c) at (4,0) {SPIR-V};
  \node[draw,rectangle,minimum width=2cm,minimum height=1cm] (d) at (6,0) {Compiler in driver};
  \node[draw,rectangle,minimum width=2cm,minimum height=1cm] (e) at (8,0) {Vendor-specific code};

  \path[->] (a) edge node {} (b);
  \path[->] (b) edge node {} (c);
  \path[->] (c) edge node {} (d);
  \path[->] (d) edge node {} (e);
\end{tikzpicture}
\end{center}

**Advantages:**

1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends
Your Sample2019.zip File Contains This

The “19” refers to the version of Visual Studio, not the year of development.
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'l', 'L'</td>
<td>Toggle lighting off and on</td>
</tr>
<tr>
<td>'m', 'M'</td>
<td>Toggle display mode (textures vs. colors, for now)</td>
</tr>
<tr>
<td>'p', 'P'</td>
<td>Pause the animation</td>
</tr>
<tr>
<td>'q', 'Q'</td>
<td>Quit the program</td>
</tr>
<tr>
<td>Esc</td>
<td>Quit the program</td>
</tr>
<tr>
<td>'r', 'R'</td>
<td>Toggle rotation-animation and using the mouse</td>
</tr>
<tr>
<td>'i', 'I'</td>
<td>Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)</td>
</tr>
<tr>
<td>‘1’, ‘4’, ‘9’</td>
<td>Set the number of instances (in the instancing version)</td>
</tr>
</tbody>
</table>
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.
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.

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.
```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\n", 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( );              // elapsed time, in double-precision seconds
        UpdateScene( );
        RenderScene( );
    }

    fprintf(FpDebug, "Closing the GLFW window\n");

    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```
void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );

    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool( );
    Init06CommandBuffers( );
Init07TextureSampler( &MyPuppyTexture.texSampler );
Init07TextureBufferAndFillFromFile("puppy.bmp", &MyPuppyTexture);

Init08Swapchain( );

Init09DepthStencilImage( );

Init10RenderPasses( );

Init11Framebuffers( );

Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );

Init13DescriptorSetPool( );
Init13DescriptorSetLayouts();
Init13DescriptorSets( );

Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
A Colored Cube

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

```c
static GLfloat CubeColors[ ][3] = {
    { 0., 0., 0. },
    { 1., 0., 0. },
    { 0., 1., 0. },
    { 1., 1., 0. },
    { 0., 0., 1. },
    { 1., 0., 1. },
    { 0., 1., 1. },
    { 1., 1., 1. },
};
```

```c
static GLfloat CubeVertices[ ][3] = {
    { -1., -1., -1. },
    { 1.,  -1., -1. },
    { -1.,  1., -1. },
    { 1.,   1., -1. },
    { -1., -1.,  1. },
    { 1.,  -1.,  1. },
    { -1.,  1.,  1. },
    { 1.,   1.,  1. }
};
```
struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};

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

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

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

    // vertex #2:
    {
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. }
    },

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

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

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

As best as I can tell, the only costs for retaining vertex attributes that you aren’t going to use are some GPU memory space and possibly some inefficient uses of the cache, but not gross performance. So, I recommend keeping this struct intact, and, if you don’t need texturing, simply don’t use the texCoord values in your vertex shader.
Vulkan Software Philosophy

Vulkan has lots of typedefs that define C/C++ structs and enums

Vulkan takes a non-C++ object-oriented approach in that those typedef'ed structs pass all the necessary information into a function. For example, where we might normally say in C++:

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

we would actually say in C:

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

**VkXxx** is a typedef, probably a struct

**vkYyy( )** 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 InitxxYYY( ) functions in numerical order. After that comes the helper functions

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

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

“IN” and “OUT” ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are actually #define’d to nothing.
Querying the Number of Something and Allocating Enough Structures to Hold Them All

```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT &physicalDevices[0] );
```

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[0] );
```
Your Sample2019.zip File Contains This

Double-click here to launch Visual Studio 2019 with this solution

The “19” refers to the version of Visual Studio, not the year of development.
struct errorcode
{
    VkResult resultCode;
    std::string meaning;
}

ErrorCodes[ ] =
{
    { VK_NOT_READY, "Not Ready" },
    { VK_TIMEOUT, "Timeout" },
    { VK_EVENT_SET, "Event Set" },
    { VK_EVENT_RESET, "Event Reset" },
    { VK_INCOMPLETE, "Incomplete" },
    { VK_ERROR_OUT_OF_HOST_MEMORY, "Out of Host Memory" },
    { VK_ERROR_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" },
    { VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" },
    { VK_ERROR_DEVICE_LOST, "Device Lost" },
    { VK_ERROR_MEMORY_MAP_FAILED, "Memory Map Failed" },
    { VK_ERROR_LAYER_NOT_PRESENT, "Layer Not Present" },
    { VK_ERROR_EXTENSION_NOT_PRESENT, "Extension Not Present" },
    { VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" },
    { VK_ERROR_INCOMPATIBLE_DRIVER, "Incompatible Driver" },
    { VK_ERROR_TOO_MANY_OBJECTS, "Too Many 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, "Error Out of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
};
void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s\n", 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, "\n%s: %s\n", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```c
#define REPORT(s)               { PrintVkError( result, s );  fflush(FpDebug); }

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

bool          Paused;

bool          Verbose;

#define DEBUGFILE               "VulkanDebug.txt"

errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );

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

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

static GLfloat CubeColors[ ][3] = 
{ 
  { 0., 0., 0. },
  { 1., 0., 0. },
  { 0., 1., 0. },
  { 1., 1., 0. },
  { 0., 0., 1. },
  { 1., 0., 1. },
  { 0., 1., 1. },
  { 1., 1., 1. },
};

static GLfloat CubeVertices[ ][3] = 
{ 
  { -1., -1., -1. },
  { 1., -1., -1. },
  { -1., 1., -1. },
  { 1., 1., -1. },
  { -1., -1., 1. },
  { 1., -1., 1. },
  { -1., 1., 1. },
  { 1., 1., 1. }
};
Triangles Represented as an Array of Structures

From the file `SampleVertexData.cpp`:

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

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

    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1.,  1. }
},

    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0.,  1. }
};
```

Modeled in right-handed coordinates
Non-indexed Buffer Drawing

From the file SampleVertexData.cpp:

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

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

Stream of Vertices

```
Stream of Vertices:

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

Triangles

Draw
```
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;
}
A Preview of What `Init05DataBuffer` Does

```c
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = (size);
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );

    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr ); // fills vmr

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

    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;

    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 ); // 0 is the offset
    return result;
}
```
Telling the Pipeline about its Input

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

C/C++:

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

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;
```

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

**struct vertex**

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

**VkVertexInputAttributeDescription** `vviad[4]`; // array per vertex input attribute

// 4 = vertex, normal, color, texture coord
`vviad[0].location = 0;` // location in the layout decoration
`vviad[0].binding = 0;` // which binding description this is part of
`vviad[0].format = VK_FORMAT_VEC3;` // x, y, z
`vviad[0].offset = offsetof( struct vertex, position );` // 0

`vviad[1].location = 1;`
`vviad[1].binding = 0;`
`vviad[1].format = VK_FORMAT_VEC3;` // nx, ny, nz
`vviad[1].offset = offsetof( struct vertex, normal );` // 12

`vviad[2].location = 2;`
`vviad[2].binding = 0;`
`vviad[2].format = VK_FORMAT_VEC3;` // r, g, b
`vviad[2].offset = offsetof( struct vertex, color );` // 24

`vviad[3].location = 3;`
`vviad[3].binding = 0;`
`vviad[3].format = VK_FORMAT_VEC2;` // s, t
`vviad[3].offset = offsetof( struct vertex, texCoord );` // 36

```cpp
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!
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.

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

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

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

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

Always use the C/C++ construct `sizeof`, rather than hardcoding a count!
struct vertex JustVertexData[ ] = {
    // vertex #0:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. } },
    // vertex #1:
    {  1., -1., -1. },
    {  0.,  0., -1. },
    {  1.,  0.,  0. },
    {  0., 0. } },
...
int JustIndexData[ ] = {
    0, 2, 3,
    0, 3, 1,
    4, 5, 7,
    4, 7, 6,
    1, 3, 7,
    1, 7, 5,
    0, 4, 6,
    0, 6, 2,
    2, 6, 7,
    2, 7, 3,
    0, 1, 5,
    0, 5, 4,};
Drawing with an Index Buffer

```c
vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets);

vkCmdBindIndexBuffer(commandBuffer, indexDataBuffer, indexOffset, indexType);

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

vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);
```
Drawing with an Index Buffer

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

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

Init05MyIndexDataBuffer(sizeof(JustIndexData), IN &MyJustIndexDataBuffer);
Fill05DataBuffer(MyJustIndexDataBuffer, (void *) JustIndexData);
```
Drawing with an Index Buffer

```
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };  
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer }; 

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );  
// 0, 1 = firstBinding, bindingCount  
vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32 ); 

const uint32_t vertexCount = sizeof( JustVertexData ) / sizeof( JustVertexData[0] );  
const uint32_t indexCount = sizeof( JustIndexData ) / sizeof( JustIndexData[0] );  
const uint32_t instanceCount = 1;  
const uint32_t firstVertex = 0;  
const uint32_t firstIndex = 0;  
const uint32_t firstInstance = 0;  
const uint32_t vertexOffset = 0; 

vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,  
vertexOffset, firstInstance ); 
```
Indirect Drawing (not to be confused with Indexed)

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

vkCmdDrawIndirect( CommandBuffers[nextImageIndex], buffer, offset, drawCount, stride);

Compare this with:

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
Indexed Indirect Drawing (i.e., both Indexed and Indirect)

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

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

Compare this with:

```c
vkCmdDrawIndexed( commandBuffer, 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 OBJ File Format – a triple-indexed way of Drawing

Note: The OBJ file format uses 1-based indexing for faces!
Shaders and SPIR-V

Mike Bailey
mjb@cs.oregonstate.edu

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

= Fixed Function

= Programmable
Shader stages

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

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:

• In the compiler, there is an automatic
  #define VULKAN 100

**Vulkan Vertex and Instance indices:**

<table>
<thead>
<tr>
<th>Vulkan</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gl_VertexIndex</td>
<td>gl_VertexID</td>
</tr>
<tr>
<td>gl_InstanceIndex</td>
<td>gl_InstanceID</td>
</tr>
</tbody>
</table>

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

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

Push Constants:
layout( push_constant ) . . . ;

Specialization Constants:
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:
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

Note: our sample code doesn’t use this.
Vulkan: Shaders’ use of Layouts for Uniform Variables

// non-sampler variables must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-sampler variables must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;

// non-sampler variables must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-sampler variables must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;

// non-sampler variables must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-sampler variables must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;
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
SPIR-V:  
Standard Portable Intermediate Representation for Vulkan

```
```

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

(Can be overridden by the –S option)

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

Windows:  glslangValidator.exe
Linux:    glslangValidator
You Can Run the SPIR-V Compiler on Windows from a Bash Shell

1. Click on the Microsoft Start icon

2. Type the word *bash*

This is only available within 64-bit Windows 10.
You Can Run the SPIR-V Compiler on Windows from a Bash Shell

This is only available within 64-bit Windows 10.

Pick one:

- Can get to your personal folders
- Does not have make

- Can get to your personal folders
- Does have make
Running glslangValidator.exe

```
ONID+mjb@pooh MINGW64 /y/Vulkan/Sample2017
$ !85
```
glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
```
ONID+mjb@pooh MINGW64 /y/Vulkan/Sample2017
$ !86
```
glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```
ONID+mjb@pooh MINGW64 /y/Vulkan/Sample2017
$
Running `glslangValidator.exe`

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

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

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

Specify the output file
How do you know if SPIR-V compiled successfully?

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

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

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

```
0203 0723 . . .
```
Reading a SPIR-V File into a Vulkan Shader Module

```c
#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''\n", 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\n",
                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\n", filename.c_str() );
    delete [ ] code;
return result;
You can also take a look at SPIR-V Assembly

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

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

```glsl
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-opaque must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

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

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

void main( )
{

    mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
    gl_Position = PVM * vec4( aVertex, 1. );

    vNormal  = Matrices.uNormalMatrix * aNormal;
    vColor   = aColor;
    vTexCoord = aTexCoord;
}
```
This is the SPIR-V Assembly, Part I

```spirv
#define version 400
#pragma enable

#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

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

// non-opaque must be in a uniform block
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

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, 1.0 );

    vNormal = Matrices.uNormalMatrix * aNormal;
    vColor = aColor;
    vTexCoord = aTexCoord;
}
```

**Capability Shader**

1:

- `ExtInstImport "GLSL.std.450"
- `MemoryModel Logical GLSL450
- `EntryPoint Vertex 4 "main" 34 37 48 53 56 57 61 63
- `Source GLSL 400
- `SourceExtension "GL_ARB_separate_shader_objects"
- `SourceExtension "GL_ARB_shading_language_420pack"
- `Name 4 "main"
- `Name 10 "PVM"
- `Name 13 "matBuf"
- `MemberName 13(matBuf) 0 "uModelMatrix"
- `MemberName 13(matBuf) 1 "uViewMatrix"
- `MemberName 13(matBuf) 2 "uProjectionMatrix"
- `MemberName 13(matBuf) 3 "uNormalMatrix"
- `Name 15 "Matrices"
- `Name 32 "gl_PerVertex"
- `MemberName 32(gl_PerVertex) 0 "gl_Position"
- `MemberName 32(gl_PerVertex) 1 "gl_PointSize"
- `MemberName 32(gl_PerVertex) 2 "gl_ClipDistance"
- `Name 34 """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
- `Name 37 "aVertex"
- `Name 48 "vNormal"
- `Name 53 "aNormal"
- `Name 56 "vColor"
- `Name 57 "aColor"
- `Name 61 "vTexCoord"
- `Name 63 "aTexCoord"
- `Name 65 "lightBuf"
- `MemberName 65(lightBuf) 0 "uLightPos"
- `Name 67 "Light"
- `MemberDecorate 13(matBuf) 0 ColMajor
- `MemberDecorate 13(matBuf) 0 Offset 0
- `MemberDecorate 13(matBuf) 0 MatrixStride 16
- `MemberDecorate 13(matBuf) 1 ColMajor
- `MemberDecorate 13(matBuf) 1 Offset 64
- `MemberDecorate 13(matBuf) 1 MatrixStride 16
- `MemberDecorate 13(matBuf) 2 ColMajor
- `MemberDecorate 13(matBuf) 2 Offset 128
- `MemberDecorate 13(matBuf) 2 MatrixStride 16
- `MemberDecorate 13(matBuf) 3 ColMajor
- `MemberDecorate 13(matBuf) 3 Offset 192
- `MemberDecorate 13(matBuf) 3 MatrixStride 16
- `Decorate 13(matBuf) Block
- `Decorate 15(Matrices) DescriptorSet 0
```
This is the SPIR-V Assembly, Part II

```
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform mat4 Buf
{
  mat4 uModeMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat3 uNormalMatrix;
}

if ( non-opaque must be in a uniform block) {
  layout( std140, set = 1, binding = 0 ) uniform lightBuf
  {
    vec4 uLightPos;
  }
}

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.uModeMatrix;
  gl_Position = PVM * vec4( aVertex, 1.0 );
  vNormal = Matrices.uNormalMatrix * aNormal;
  vColor = aColor;
  vTexCoord = aTexCoord;
}
```

Decorate 15(Matrices) Binding 0
MemberDecorate 32(gl_PerVertex) 0 BuiltIn Position
MemberDecorate 32(gl_PerVertex) 1 BuiltIn PointSize
MemberDecorate 32(gl_PerVertex) 2 BuiltIn ClipDistance
Decorate 32(gl_PerVertex) Block
Decorate 37(aVertex) Location 0
Decorate 46(vNormal) Location 0
Decorate 53(aNormal) Location 1
Decorate 56(vColor) Location 1
Decorate 57(aColor) Location 2
Decorate 61(vTexCoord) Location 2
Decorate 63(aTexCoord) Location 3
MemberDecorate 65(lightBuf) 0 Offset 0
Decorate 65(lightBuf) Block
Decorate 67(Light) DescriptorSet 1
Decorate 67(Light) Binding 0

2:     TypeVoid
3:     TypeFunction 2
6:     TypeFloat 32
7:     TypeVector 6(float) 4
8:     TypeMatrix 7(fvec4) 4
9:     TypePointer Function 8
11:    TypeVector 6(float) 3
12:    TypeMatrix 11(fvec3) 3
13(matBuf):  TypeStruct 8 8 8 12
14:    TypePointer Uniform 13(matBuf)
15(Matrices):  14(ptr) Variable Uniform
16:    TypeInt 32 1
17:    16(int) Constant 2
18:    TypePointer Uniform 8
21:    16(int) Constant 1
25:    16(int) Constant 0
29:    TypeInt 32 0
30:    29(int) Constant 1
31:    TypeArray 6(float) 30
32(gl_PerVertex):  TypeStruct 7(fvec4) 6(float) 31
33:    TypePointer Output 32(gl_PerVertex)
34:    33(ptr) Variable Output
36:    TypePointer Input 11(fvec3)
37(aVertex):  36(ptr) Variable Input
39:    6(float) Constant 1065353216
45:    TypePointer Output 7(fvec4)
47:    TypePointer Output 11(fvec3)
48(vNormal):  47(ptr) Variable Output
49:    16(int) Constant 3
This is the SPIR-V Assembly, Part III

```spirv-assembly
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform mat4 lightBuf
{
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat4 uNormalMatrix;
} Matrices;

// non-opaque must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
  vec4 uLightPos;
} Light;

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, 1.0 );

  vNormal = Matrices.uNormalMatrix * aNormal;
  vColor = aColor;
  vTexCoord = aTexCoord;
}
```

50: TypePointer Uniform 12
53(aNormal): 36(ptr) Variable Input
56(vColor): 47(ptr) Variable Output
57(aColor): 36(ptr) Variable Input
59: TypeVector 6(float) 2
60: TypePointer Output 59(fvec2)
61(vTexCoord): 60(ptr) Variable Output
62: TypePointer Input 59(fvec2)
63(aTexCoord): 62(ptr) Variable Input
65(lightBuf): TypeStruct 7(fvec4)
66: TypePointer Uniform 65(lightBuf)
67(Light): 66(ptr) Variable Uniform
4(main): 2 Function None 3
5: Label
10(PVM): 9(ptr) Variable Function
19: 18(ptr) AccessChain 15(Matrices) 17
20: 8 Load 19
22: 18(ptr) AccessChain 15(Matrices) 21
23: 8 Load 22
24: 8 MatrixTimesMatrix 20 23
26: 18(ptr) AccessChain 15(Matrices) 25
27: 8 Load 26
28: 8 MatrixTimesMatrix 24 27
29: Store 10(PVM) 28
35: 8 Load 10(PVM)
38: 11(fvec3) Load 37(aVertex)
40: 6(float) CompositeExtract 38 0
41: 6(float) CompositeExtract 38 1
42: 6(float) CompositeExtract 38 2
43: 7(fvec4) CompositeConstruct 40 41 42 39
44: 7(fvec4) MatrixTimesVector 35 43
46: 45(ptr) AccessChain 34 25
47: Store 46 44
51: 50(ptr) AccessChain 15(Matrices) 49
52: 12 Load 51
54: 11(fvec3) Load 53(aNormal)
55: 11(fvec3) MatrixTimesVector 52 54
56: Store 48(vNormal) 55
58: 11(fvec3) Load 57(aColor)
59: Store 56(vColor) 58
64: 59(fvec2) Load 63(aTexCoord)
65: Store 61(vTexCoord) 64
66: Return
67: FunctionEnd
```
## SPIR-V: Printing the Configuration

```
glslangValidator --c
MaxLights 32
MaxClipPlanes 6
MaxTextureUnits 32
MaxTextureCoords 32
MaxVertexAttribs 64
MaxVertexUniformComponents 4096
MaxVaryingFloats 64
MaxVertexTextureImageUnits 32
MaxCombinedTextureImageUnits 32
MaxFragmentUniformComponents 4096
MaxDrawBuffers 32
MaxVertexUniformVectors 128
MaxVaryingVectors 8
MaxFragmentUniformVectors 16
MaxVertexOutputVectors 16
MaxFragmentInputVectors 15
MinProgramTexelOffset -8
MaxProgramTexelOffset 7
MaxClipDistances 8
MaxComputeWorkGroupCountX 65535
MaxComputeWorkGroupCountY 65535
MaxComputeWorkGroupCountZ 65535
MaxComputeWorkGroupSizeX 1024
MaxComputeWorkGroupSizeY 1024
MaxComputeWorkGroupSizeZ 64
MaxComputeUniformComponents 1024
MaxComputeTextureImageUnits 16
MaxComputeImageUniforms 8
MaxComputeAtomicCounters 8
MaxComputeAtomicCounterBuffers 1
MaxVaryingComponents 60
MaxVertexOutputComponents 64
MaxGeometryInputComponents 64
MaxGeometryOutputComponents 128
MaxFragmentInputComponents 128
MaxImageUnits 8
MaxCombinedImageUnitsAndFragmentOutputs 8
MaxCombinedShaderOutputResources 8
MaxImageSamples 0
MaxVertexImageUniforms 0
MaxTessControlImageUniforms 0
MaxTessEvaluationImageUniforms 0
MaxGeometryImageUniforms 0
MaxFragmentImageUniforms 81
MaxGeometryTextureImageUnits 16
MaxGeometryOutputVertices 256
MaxGeometryTotalOutputComponents 1024
MaxGeometryUniformComponents 1024
MaxGeometryVaryingComponents 64
MaxTessControlInputComponents 128
MaxTessControlOutputComponents 128
MaxTessControlTextureImageUnits 16
MaxTessControlUniformComponents 1024
MaxTessControlTotalOutputComponents 4096
MaxTessEvaluationInputComponents 128
MaxTessEvaluationOutputComponents 128
MaxTessEvaluationTextureImageUnits 16
MaxTessEvaluationUniformComponents 1024
MaxTessPatchComponents 120
MaxPatchVertices 32
MaxTessGenLevel 64
MaxViewports 16
MaxVertexAtomicCounters 0
MaxTessControlAtomicCounters 0
MaxTessEvaluationAtomicCounters 0
MaxGeometryAtomicCounters 0
MaxFragmentAtomicCounters 8
MaxCombinedAtomicCounters 8
MaxAtomicCounterBindings 1
MaxVertexAtomicCounterBuffers 0
MaxTessControlAtomicCounterBuffers 0
MaxTessEvaluationAtomicCounterBuffers 0
MaxGeometryAtomicCounterBuffers 0
MaxFragmentAtomicCounterBuffers 1
MaxCombinedAtomicCounterBuffers 1
MaxAtomicCounterBufferSize 16384
MaxTransformFeedbackBuffers 4
MaxTransformFeedbackInterleavedComponents 64
MaxCullDistances 8
MaxCombinedClipAndCullDistances 8
MaxSamples 4
nonInductiveForLoops 1
whileLoops 1
doWhileLoops 1
generalUniformIndexing 1
generalAttributeMatrixVectorIndexing 1
generalVaryingIndexing 1
generalSamplerIndexing 1
generalVariableIndexing 1
generalConstantMatrixVectorIndexing 1
```
SPIR-V Tools:
http://github.com/KhronosGroup/SPIRV-Tools
The shaderc project from Google (https://github.com/google/shaderc) provides a glslangValidator wrapper program called **glslc** that has a much improved command-line interface. You use, basically, the same way:

```
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 `#include` files into your shader source

2. You can “`#define`” definitions on the command line like this:

```
glslc.exe   --target-env=vulkan   -DNUMPONTS=4   sample-vert.vert   -o   sample-vert.spv
```

*glslc* is included in your Sample .zip file
Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline or a compute pipeline.
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:

```c
typedef VkBuffer VkDataBuffer;
```

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

1. **VkBufferCreateInfo**
   - `bufferUsage`
   - `queueFamilyIndices`
   - `size (bytes)`

2. **vkCreateBuffer( )**

3. **Buffer**
   - `vkGetBufferMemoryRequirements( )`
     - `memoryType`
     - `size`
   - **VkMemoryAllocateInfo**

4. **vkAllocateMemory( )**
   - `LogicalDevice`
   - `bufferMemoryHandle`

5. **vkBindBufferMemory( )**

6. **vkMapMemory( )**
   - `LogicalDevice`
   - `gpuAddress`
Creating a Vulkan Data Buffer

**VkBuffer Buffer;**

VkBufferCreateInfo vbci;

- vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
- vbci.pNext = nullptr;
- vbci.flags = 0;
- vbci.size = << buffer size in bytes >>
- vbci.usage = <<or’ed bits of: >>
  - VK_USAGE_TRANSFER_SRC_BIT
  - VK_USAGE_TRANSFER_DST_BIT
  - VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
  - VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
  - VK_USAGE_UNIFORM_BUFFER_BIT
  - VK_USAGE_STORAGE_BUFFER_BIT
  - VK_USAGE_INDEX_BUFFER_BIT
  - VK_USAGE_VERTEX_BUFFER_BIT
  - VK_USAGE_INDIRECT_BUFFER_BIT
- vbci.sharingMode = << one of: >>
  - VK_SHARING_MODE_EXCLUSIVE
  - VK_SHARING_MODE_CONCURRENT
- vbci.queueFamilyIndexCount = 0;
- vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

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

result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset

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

result = vkUnmapMemory( LogicalDevice, IN vdm );
```
int FindMemoryThatIsHostVisible() {
    VkPhysicalDeviceMemoryProperties vpdmp;
    vkGetPhysicalDeviceMemoryProperties(PhysicalDevice, OUT &vpdmp);
    for(unsigned int i = 0; i < vpdmp.memoryTypeCount; i++) {
        VkMemoryType vmt = vpdmp.memoryTypes[i];
        if((vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT) != 0) {
            return i;
        }
    }
    return -1;
}
Finding the Right Type of Memory

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

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

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
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.
Sidebar: The Vulkan Memory Allocator (VMA)

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

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

... MyBuffer MyMatrixUniformBuffer;
```

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

It also makes it impossible to accidentally associate the wrong VkDeviceMemory and/or VkDeviceSize with the wrong 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.

```
struct matBuf Matrices;

glm::vec3 eye(0,0,EYEDIST);
glm::vec3 look(0,0,0);
glm::vec3 up(0,1,0);
Matrices.uModelMatrix = glm::mat4();  // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.1;
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

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

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

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

MyBuffer MyMatrixUniformBuffer;

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

glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);

Matrices.uModelMatrix = glm::mat4( ); // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );

Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;

Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```
Creating and Filling the Data Buffer – the Details

```
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );

    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );  // fills vmr

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

    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;

    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```
VkResult
Fill05DataBuffer(IN MyBuffer myBuffer, IN void * data)
{
    // the size of the data had better match the size that was used to Init the buffer!
    void * pGpuMemory;
    vkMapMemory(LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0_OUT &pGpuMemory);
    // 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.
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.
GLFW

Mike Bailey
mjb@cs.oregonstate.edu

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

http://cs.oregonstate.edu/~mjb/vulkan
GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating window contexts and surfaces, receiving input and events.

GLFW is written in C and has native support for Windows, macOS and many Unix-like systems using the X Window System, such as Linux and FreeBSD.

GLFW is licensed under the zlib/libpng license.

- Gives you a window and OpenGL context with just two function calls
- Support for OpenGL, OpenGL ES, Vulkan and related options, flags and extensions
- Support for multiple windows, multiple monitors, high-DPI and gamma ramps
- Support for keyboard, mouse, gamepad, time and window event input, via polling or callbacks
- Comes with guides, a tutorial, reference documentation, examples and test programs
- Open Source with an OSI-certified license allowing commercial use
- Access to native objects and compile-time options for platform specific features
- Community-maintained bindings for many different languages

No library can be perfect for everyone. If GLFW isn’t what you’re looking for, there are alternatives.

http://www.glfw.org/
#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!\n" );
        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);

fprintf( FpDebug, "\nFound %d GLFW Required Instance Extensions:\n", count );

for( uint32_t i = 0; i < count; i++ )
{
    fprintf( FpDebug, "\t%s\n", 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:
                printf( FpDebug, "Unknown 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 );
            break;
    }

    // button down sets the bit, up clears the bit:
    if( action == GLFW_PRESS )
    {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;
        Ymouse = (int)ypos;
        ActiveButton |= b; // set the proper bit
    }
    else
    {
        ActiveButton &= ~b; // clear the proper bit
    }
}
```
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;
}
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );
    Time = glfwGetTime( );       // elapsed time, in double-precision seconds
    UpdateScene( );
    RenderScene( );
}

vkQueueWaitIdle( Queue );
vkDeviceWaitIdle( LogicalDevice );
DestroyAllVulkan( );
glfwDestroyWindow( MainWindow );
glfwTerminate( );

Does not block – processes any waiting events, then returns
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();
```
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:

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

OpenGL treats all angles as given in degrees. This line forces GLM to treat all angles as given in radians. I recommend this so that all angles you create in all programming will be in radians.

If GLM is not installed in a system place, put it somewhere you can get access to. Later on, these notes will show you how to use it from there.
Why are we even talking about this?

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

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

you would now say:

```cpp
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 );  // \{x',y',z'\} = [v]*\{x,y,z\}
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0. ) );  // \{x',y',z'\} = [v]*[yr]*\{x,y,z\}
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0. ) );  // \{x',y',z'\} = [v]*[yr]*[xr]*\{x,y,z\}
modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) );  // \{x',y',z'\} = [v]*[yr]*[xr]*[s]*\{x,y,z\}
```

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

glm::mat4( 1. ); // identity matrix
glm::vec4( );
glm::vec3( );

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

// multiplications:

glm::mat4 * glm::mat4
glm::mat4 * glm::vec4
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);

glm::mat4 glm::frustum(float left, float right, float bottom, float top, float near, float far);
glm::mat4 glm::perspective(float fovy, float aspect, float near, float far);

// viewing (assign, not concatenate):

glm::mat4 glm::lookAt(glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up);
Installing GLM into your own space

I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.
Here’s what that GLM folder looks like
Telling Visual Studio about where the GLM folder is

1. Telling Visual Studio about where the GLM folder is
2. Refreshing the Project Toolbox Items

```
// This is a sample OpenGL / GLUT program
// The objective is to draw a 3D object and change the color of the axes
// with a glut menu
//
// The left mouse button does rotation
// The middle mouse button does scaling
// The user interface allows:
// 1. The axes to be turned on and off
// 2. The color of the axes to be changed
// 3. Debugging to be turned on and off
// 4. Depth cueing to be turned on and off
// 5. The projection to be changed
// 6. The transformations to be reset
```
A *period*, indicating that the **project folder** should also be searched when a `#include <xxx>` is encountered. If you put it somewhere else, enter that full or relative path instead.
if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4( 1. );           // identity
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0. ) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0. ) );
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
        // 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 );
How Does this Matrix Stuff Really Work?

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

Or, in matrix form:

\[
\begin{align*}
x' &= Ax + By + Cz + D \\
y' &= Ex + Fy + Gz + H \\
z' &= Ix + Jy + Kz + L
\end{align*}
\]
Transformation Matrices

Translation

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

Rotation about X

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

Rotation about Y

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

Rotation about Z

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

Translation

Scaling

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

Rotation about X

Rotation about Y

Rotation about Z
How it Really Works :-)

\[
\begin{bmatrix}
\cos 90^\circ & \sin 90^\circ \\
-\sin 90^\circ & \cos 90^\circ
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2
\end{bmatrix} = \begin{bmatrix} \theta \\ \phi \end{bmatrix}
\]

http://xkcd.com
The Rotation Matrix for an Angle ($\theta$) about an Arbitrary Axis ($Ax, Ay, Az$)

$$[M] = \begin{bmatrix}
A_x A_x + \cos \theta(1 - A_x A_x) & A_x A_y - \cos \theta(A_x A_y) - \sin \theta A_z & A_x A_z - \cos \theta(A_x A_z) + \sin \theta A_y \\
A_y A_x - \cos \theta(A_y A_x) + \sin \theta A_z & A_y A_y + \cos \theta(1 - A_y A_y) & A_y A_z - \cos \theta(A_y A_z) - \sin \theta A_x \\
A_z A_x - \cos \theta(A_z A_x) - \sin \theta A_y & A_z A_y - \cos \theta(A_z A_y) + \sin \theta A_x & A_z A_z + \cos \theta(1 - A_z A_z)
\end{bmatrix}$$

For this to be correct, $A$ must be a unit vector.
Q: Our rotation matrices only work around the origin? What if we want to rotate about an arbitrary point (A,B)?

A: We create more than one matrix.

\[
\begin{pmatrix}
x' \\
y' \\
z' \\
1
\end{pmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} R_\theta \end{bmatrix} \cdot \begin{bmatrix} T_{-A,-B} \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
\]
Matrix Multiplication is not Commutative

Y

Rotate, then translate

X

Y

Translate, then rotate

X

Y
Matrix Multiplication is Associative

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \left[ T_{+A,+B} \right] \cdot \left[ R_{\theta} \right] \cdot \left[ T_{-A,-B} \right] \cdot \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \left( \left[ T_{+A,+B} \right] \cdot \left[ R_{\theta} \right] \cdot \left[ T_{-A,-B} \right] \right) \cdot \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

One matrix – the Current Transformation Matrix, or CTM
One Matrix to Rule Them All

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \left( \begin{bmatrix}
  T_{A,+B} & 0 & 0 & 0 \\
  0 & R_\theta & 0 & 0 \\
  0 & 0 & T_{-A,-B} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \right) \cdot \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

```cpp
glm::mat4 Model = glm::mat4(1.);
Model = glm::translate(Model, glm::vec3(A, B, 0.));
Model = glm::rotate(Model, thetaRadians, glm::vec3(Ax, Ay, Az));
Model = glm::translate(Model, glm::vec3(-A, -B, 0.));

glm::vec3 eye(0., 0., EYEDIST);
glm::vec3 look(0., 0., 0.);
glm::vec3 up(0., 1., 0.);
glm::mat4 View = glm::lookAt(eye, look, up);

glm::mat4 Projection = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Projection[1][1] *= -1.;

glm::mat3 Matrix = Projection * View * Model;
glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(Model));
```
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.

Original object and normal

Wrong!

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

Right!

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

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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

`gl_InstanceIndex` starts at 0

In the vertex shader:

```glsl
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:

```cpp
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 = . . .

gl_Position = PVM * vertex;  // [p]*[v]*[m]
```
The Graphics Pipeline Data Structure

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
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.
Graphics Pipeline Stages and what goes into Them

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

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

- **Topology**

- **Viewport Scissoring**

- **Tessellation Shaders, Geometry Shader**

- **Depth Clamping**
  - DiscardEnable
  - PolygonMode
  - CullMode
  - FrontFace
  - LineWidth

- **Which states are dynamic**
  - DepthTestEnable
  - DepthWriteEnable
  - DepthCompareOp
  - StencilTestEnable

- **Fragment Shader module**
  - Specialization info

- **Color Blending parameters**

**Graphics Pipeline Stages**

1. **Input Assembly**
2. **Viewport**
3. **Tessellation, Geometry Shaders**
4. **Rasterization**
5. **Dynamic State**
6. **Depth/Stencil**
7. **Fragment Shader Stage**
8. **Color Blending Stage**
The First Step: Create the Graphics Pipeline Layout

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

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

    VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4; vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0; vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

    return result;
}
```

Why is this necessary? It is because the Descriptor Sets and Push Constants data structures have different sizes depending on how many of each you have. So, the exact structure of the Pipeline Layout depends on you telling Vulkan about the Descriptor Sets and Push Constants that you will be using.
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
- 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

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.
The Shaders to Use

Use one **vpssci** array member per shader module you are using.

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

// 4 = vertex, normal, color, texture coord
vviad[0].location = 0; // location in the layout
vviad[0].binding = 0; // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3; // x, y, z
vviad[0].offset = offsetof( struct vertex, position ); // 0

#ifdef EXTRAS_DEFINED_AT_THE_TOP
// these are here for convenience and readability:
#define VK_FORMAT_VEC4 VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_XYZW VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_VEC3 VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_FLOAT VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_S VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_X VK_FORMAT_R32_SFLOAT
#endif

vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3; // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal ); // 12

vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3; // r, g, b
vviad[2].offset = offsetof( struct vertex, color ); // 24

vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2; // s, t
vviad[3].offset = offsetof( struct vertex, texCoord ); // 36

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

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

Link in the Per-Vertex Attributes
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;
   #ifdef _WITH_ADJACENCY
      VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
      VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
   #endif
   vpiasci.primitiveRestartEnable = VK_FALSE;

VkPipelineTessellationStateCreateInfo `vptsci`;
   vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
   vptsci.pNext = nullptr;
   vptsci.flags = 0;
   vptsci.patchControlPoints = 0; // number of patch control points

VkPipelineGeometryStateCreateInfo `vpgsci`;
   vpgsci.sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO;
   vpgsci.pNext = nullptr;
   vpgsci.flags = 0;
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”?  

```c
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

Triangle Strip #0:

Triangle Strip #1:

Triangle Strip #2:

...
VkViewport
vv.x = 0;
vv.y = 0;
vv.width = (float)Width;
vv.height = (float)Height;
vv.minDepth = 0.0f;
vv.maxDepth = 1.0f;

VkRect2D
vr.offset.x = 0;
vr.offset.y = 0;
vr.extent.width = Width;
vr.extent.height = Height;

VkPipelineViewportStateCreateInfo vpvsci;
pvsci.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
pvsci.pNext = nullptr;
pvsci.flags = 0;
pvsci.viewportCount = 1;
pvsci.pViewports = &vv;
pvsci.scissorCount = 1;
pvsci.pScissors = &vr;

Declare the viewport information

Declare the scissoring information

Group the viewport and scissor 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 (scrunching) 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.
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_FILL
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;

Declare information about how the rasterization will take place
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**:

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

The gray area shows what would happen with depthClampEnable (except it would have been red).
What is “Depth Bias Enable”? 

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

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

We will discuss MultiSampling in a separate noteset.

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

Declare information about how the multisampling will take place
Color Blending State for each Color Attachment *

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

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

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

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

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

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

```c
VkPipelineColorBlendStateCreateInfo vpcbsci;
  vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
  vpcbsci.pNext = nullptr;
  vpcbsci.flags = 0;
  vpcbsci.logicOpEnable = VK_FALSE;
  vpcbsci.logicOp = VK_LOGIC_OP_COPY;

#ifdef CHOICES
  VK_LOGIC_OP_CLEAR
  VK_LOGIC_OP_AND
  VK_LOGIC_OP_AND_REVERSE
  VK_LOGIC_OP_COPY
  VK_LOGIC_OP_AND_INVERTED
  VK_LOGIC_OP_NO_OP
  VK_LOGIC_OP_XOR
  VK_LOGIC_OP_OR
  VK_LOGIC_OP_NOR
  VK_LOGIC_OP_EQUIVALENT
  VK_LOGIC_OP_INVERT
  VK_LOGIC_OP_OR_REVERSE
  VK_LOGIC_OP_COPY_INVERTED
  VK_LOGIC_OP_OR_INVERTED
  VK_LOGIC_OP_NAND
  VK_LOGIC_OP_SET
#endif
  vpcbsci.attachmentCount = 1;
  vpcbsci.pAttachments = &vpcbas;
  vpcbsci.blendConstants[0] = 0;
  vpcbsci.blendConstants[1] = 0;
  vpcbsci.blendConstants[2] = 0;
  vpcbsci.blendConstants[3] = 0;
```

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

Just used as an example in the Sample Code

```c
VkDynamicState

#define CHOICES

VkPipelineDynamicStateCreateInfo vpdsci
    vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
    vpdsci.pNext = nullptr;
    vpdsci.flags = 0;
    vpdsci.dynamicStateCount = 0; // leave turned off for now
    vpdsci.pDynamicStates = vds;
```

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

```c
    VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport( )
    VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor( )
    VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth( )
    VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias( )
    VK_DYNAMIC_STATE_BLEND_CONSTANTS-- vkCmdSetBlendConstants( )
    VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthZBounds( )
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompareMask( )
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK -- vkCmdSetStencilWriteMask( )
    VK_DYNAMIC_STATE_STENCIL_REFERENCE  -- vkCmdSetStencilReferences( )
```
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*
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
Using the Stencil Buffer to Perform Polygon Capping
Using the Stencil Buffer to Perform *Polygon Capping*

1. Clear the SB = 0
2. Draw the polygons, setting SB = ~ SB
3. Draw a large gray polygon across the entire scene wherever SB != 0
Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting
Using the Stencil Buffer to Better Outline Polygons
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*
Stencil Operations for Front and Back Faces

```
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

#ifdef CHOICES
VK_STENCIL_OP_KEEP -- keep the stencil value as it is
VK_STENCIL_OP_ZERO -- set stencil value to 0
VK_STENCIL_OP_REPLACE -- replace stencil value with the reference value
VK_STENCIL_OP_INCREMENT_AND_CLAMP -- increment stencil value
VK_STENCIL_OP_DECREMENT_AND_CLAMP -- decrement stencil value
VK_STENCIL_OP_INVERT -- bit-invert stencil value
VK_STENCIL_OP_INCREMENT_AND_WRAP -- increment stencil value
VK_STENCIL_OP_DECREMENT_AND_WRAP -- decrement stencil value
#endif

vsosf.compareOp = VK_COMPARE_OP_NEVER;

#ifdef CHOICES
VK_COMPARE_OP_NEVER -- never succeeds
VK_COMPARE_OP_LESS -- succeeds if stencil value is < the reference value
VK_COMPARE_OP_EQUAL -- succeeds if stencil value is == the reference value
VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if stencil value is <= the reference value
VK_COMPARE_OP_GREATER -- succeeds if stencil value is > the reference value
VK_COMPARE_OP_NOT_EQUAL -- succeeds if stencil value is != the reference value
VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if stencil value is >= the reference value
VK_COMPARE_OP_ALWAYS -- always succeeds
#endif

vsosf.compareMask = ~0;
vsosf.writeMask = ~0;
vsosf.reference = 0;

VkStencilOpState vsosb; // back
vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
vsosb.failOp = VK_STENCIL_OP_KEEP;
vsosb.passOp = VK_STENCIL_OP_KEEP;
vsosb.compareOp = VK_COMPARE_OP_NEVER;
vsosb.compareMask = ~0;
vsosb.writeMask = ~0;
vsosb.reference = 0;
```
VkPipelineDepthStencilStateCreateInfo vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;

VK_COMPARE_OP_NEVER -- never succeeds
VK_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
VK_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
VK_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
VK_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
VK_COMPARE_OP_ALWAYS -- always succeeds

vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;
Putting it all Together! (finally…)

```c
VkPipeline GraphicsPipeline;

VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#endif

#define CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

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

return result;
```
Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

```cpp
vkCmdBindPipeline( CommandBuffers[nextImageIndex], 
    VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```
Sidebar: What is the Organization of the Pipeline Data Structure?

If you take a close look at the pipeline data structure creation information, you will see that almost all the pieces have a *fixed size*. For example, the viewport only needs 6 pieces of information – ever:

```c
VkViewport vv;
vv.x = 0;
vv.y = 0;
vv.width = (float)Width;
vv.height = (float)Height;
vv.minDepth = 0.0f;
vv.maxDepth = 1.0f;
```

There are two exceptions to this -- the Descriptor Sets and the Push Constants. Each of these two can be almost any size, depending on what you allocate for them. So, I think of the Pipeline Data Structure as consisting of some fixed-layout blocks and 2 variable-layout blocks, like this:
Descriptor Sets

Mike Bailey
mjb@cs.oregonstate.edu

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

```
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;
```
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.

```c
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

Our example will assume the following shader uniform variables:

```glsl
// non-opaque must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

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

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

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

**GPU:**
Uniform data in a “blob”*
- Knows where the data starts
- Knows the data’s size

**GPU:**
Uniform data used in the shader
- Knows the shader data structure
- Doesn’t know where each piece of data starts

```cpp
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;
};
```

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

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

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

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

* “binary large object”
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.
VkResult Init13DescriptorSetPool()
{
    VkResult result;
    VkDescriptorPoolSize vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;

    #ifdef CHOICES
    VK_DESCRIPTOR_TYPE_SAMPLER
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
    #endif

    VkDescriptorPoolCreateInfo vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;
    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = &vdps[0];

    result = vkCreateDescriptorPool(LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
    return result;
}
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.

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

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

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

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```
VkResult
Init13DescriptorSetLayouts( )
{
    VkResult result;

    // DS #0:
    VkDescriptorSetLayoutBinding       MatrixSet[1];
        MatrixSet[0].binding      = 0;
        MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        MatrixSet[0].descriptorCount = 1;
        MatrixSet[0].stageFlags     = VK_SHADER_STAGE_VERTEX_BIT;
        MatrixSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #1:
    VkDescriptorSetLayoutBinding       LightSet[1];
        LightSet[0].binding        = 0;
        LightSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        LightSet[0].descriptorCount = 1;
        LightSet[0].stageFlags      = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
        LightSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #2:
    VkDescriptorSetLayoutBinding       MiscSet[1];
        MiscSet[0].binding         = 0;
        MiscSet[0].descriptorType  = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        MiscSet[0].descriptorCount = 1;
        MiscSet[0].stageFlags      = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
        MiscSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #3:
    VkDescriptorSetLayoutBinding       TexSamplerSet[1];
        TexSamplerSet[0].binding   = 0;
        TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        TexSamplerSet[0].descriptorCount  = 1;
        TexSamplerSet[0].stageFlags     = VK_SHADER_STAGE_FRAGMENT_BIT;
        TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;

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

MatrixSet DS Layout Binding:
- binding
- descriptorType
- descriptorCount
- pipeline stage(s)

set = 0

vdslc0 DS Layout CI:
- bindingCount
- type
- number of that type
- pipeline stage(s)

LightSet DS Layout Binding:
- binding
- descriptorType
- descriptorCount
- pipeline stage(s)

set = 1

vdslc1 DS Layout CI:
- bindingCount
- type
- number of that type
- pipeline stage(s)

MiscSet DS Layout Binding:
- binding
- descriptorType
- descriptorCount
- pipeline stage(s)

set = 2

vdslc2 DS Layout CI:
- bindingCount
- type
- number of that type
- pipeline stage(s)

TexSamplerSet DS Layout Binding:
- binding
- descriptorType
- descriptorCount
- pipeline stage(s)

set = 3

vdslc3 DS Layout CI:
- bindingCount
- type
- number of that type
- pipeline stage(s)

Array of Descriptor Set Layouts

Pipeline Layout
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];

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];

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];

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, IN &vdslc0, PALLOCATOR, OUT &DescriptorSetLayouts[0] );
result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc1, PALLOCATOR, OUT &DescriptorSetLayouts[1] );
result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc2, PALLOCATOR, OUT &DescriptorSetLayouts[2] );
result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc3, PALLOCATOR, OUT &DescriptorSetLayouts[3] );

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

```c
VkResult
Init14GraphicsPipelineLayout( )
{
    VkResult result;
    VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

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

![Diagram](image.png)
Step 4: Allocating the Memory for Descriptor Sets

```cpp
VkResult
Init13DescriptorSets( )
{
    VkResult result;

    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;

    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );
}
```
Step 5: Tell the Descriptor Sets where their CPU Data is

VkDescriptorBufferInfo  
- `vdbi0`: buffer = MyMatrixUniformBuffer.buffer;  
- `vdbi1`: buffer = MyLightUniformBuffer.buffer;  
- `vdbi2`: buffer = MyMiscUniformBuffer.buffer;

Each of these structs identifies what buffer it owns and how big it is.

VkDescriptorImageInfo  
- `vdii0`: sampler = MyPuppyTexture.texSampler;  
- `vdii1`: imageView = MyPuppyTexture.texImageView;  
- `vdii2`: imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

This struct identifies what texture sampler and image view it owns.
Step 5: Tell the Descriptor Sets where their CPU Data is

```
VkWriteDescriptorSet vwds0;
    // ds 0:
    vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds0.pNext = nullptr;
    vwds0.dstSet = DescriptorSets[0];
    vwds0.dstBinding = 0;
    vwds0.dstArrayElement = 0;
    vwds0.descriptorCount = 1;
    vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vwds0.pBufferInfo = &vdbi0;
    vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
    vwds0.pTexelBufferView = (VkBufferView *)nullptr;

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

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

This struct links a Descriptor Set to the buffer it is pointing to.
Step 5: Tell the Descriptor Sets where their data is

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

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

```
VkWriteDescriptorSet vwds2;
    // ds 2:
    vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds2.pNext = nullptr;
    vwds2.dstSet = DescriptorSets[2];
    vwds2.dstBinding = 0;
    vwds2.dstArrayElement = 0;
    vwds2.descriptorCount = 1;
    vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vwds2.pBufferInfo = &vdbi2;
    vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
    vwds2.pTexelBufferView = (VkBufferView *)nullptr;

    // ds 3:
    VkWriteDescriptorSet vwds3;
    vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds3.pNext = nullptr;
    vwds3.dstSet = DescriptorSets[3];
    vwds3.dstBinding = 0;
    vwds3.dstArrayElement = 0;
    vwds3.descriptorCount = 1;
    vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vwds3.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
    vwds3.pImageInfo = &vdii0;
    vwds3.pTexelBufferView = (VkBufferView *)nullptr;

    uint32_t copyCount = 0;

    // this could have been done with one call and an array of VkWriteDescriptorSets:
    vkUpdateDescriptorSets( LogicalDevice, 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

```c
VkGraphicsPipelineCreateInfo vgpci;

vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;

#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

vgpci.stageCount = 2;  // number of stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.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 );
```
Step 7: Bind Descriptor Sets into the Command Buffer when Drawing

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

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: The Entire Collection of Descriptor Set Paths

- VkCreateDescriptorPool( )
  Create the pool of Descriptor Sets for future use

- VkCreateDescriptorSetLayout( )
  Describe a particular Descriptor Set layout and use it in a specific Pipeline layout

- VkAllocateDescriptorSets( )
  Allocate memory for particular Descriptor Sets

- VkUpdateDescriptorSets( )
  Tell a particular Descriptor Set where its CPU data is

- VkCmdBindDescriptorSets( )
  Re-write CPU data into a particular Descriptor Set

- VkCmdBindDescriptorSets( )
  Make a particular Descriptor Set “current” for rendering

VkDescriptorPoolCreateInfo
VkDescriptorSetLayoutBinding
VkDescriptorSetLayoutCreateInfo
VkDescriptorPoolCreateInfo
VkDescriptorBufferInfo
VkDescriptorImageInfo
VkDescriptorSetAllocateInfo
VkDescriptorSetLayoutCreateInfo
VkWriteDescriptorSet
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**
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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
The Basic Idea

Texture mapping is a computer graphics operation in which a separate image, referred to as the texture, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a texture map.

Also, to prevent confusion, the texture pixels are not called pixels. A pixel is a dot in the final screen image. A dot in the texture image is called a texture element, or texel.

Similarly, to avoid terminology confusion, a texture’s width and height dimensions are not called X and Y. They are called S and T. A texture map is not generally indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always S=0., the right side is S=1., the bottom is T=0., and the top is T=1. Thus, you do not need to be aware of the texture’s resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.
The Basic Idea

The mapping between the geometry of the 3D object and the S and T of the texture image works like this:

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that the rasterizer is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when it goes to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.
Enable texture mapping:

```c
glEnable( GL_TEXTURE_2D );
```

Draw your polygons, specifying \( s \) and \( t \) at each vertex:

```c
glBegin( GL_POLYGON );
    glTexCoord2f( s0, t0 );
    glNormal3f( nx0, ny0, nz0 );
    glVertex3f( x0, y0, z0 );
    glTexCoord2f( s1, t1 );
    glNormal3f( nx1, ny1, nz1 );
    glVertex3f( x1, y1, z1 );
    ...
    glEnd( );
```

Disable texture mapping:

```c
glDisable( GL_TEXTURE_2D );
```
Triangles in an Array of Structures

```cpp
// Define a vertex structure
struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};

// Define the array of vertex data
struct vertex VertexData[] =
{
    // Triangle 0-2-3:
    // Vertex #0:
    {
        { -1., -1., -1. },
        {  0.,  0., -1. },
        {  0.,  0.,  0. },
        {  1., 0. }
    },
    // Vertex #2:
    {
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. }
    },
    // Vertex #3:
    {
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        {  1.,  1.,  0. },
        {  0.,  1. }
    }
};
```
Using a Texture: How do you know what \((s,t)\) to assign to each vertex?

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

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

Or, for a sphere,

\[
s = \frac{\Theta - (-\pi)}{2\pi} \quad \quad \quad \quad \quad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}
\]

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

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

\[ s = ? \quad \text{and} \quad t = ? \]
You really are at the mercy of whoever did the modeling…
Be careful where $s$ abruptly transitions from 1. back to 0.
Memory Types

CPU Memory

GPU Memory

Host
Visible
GPU Memory

Device
Local
GPU Memory

memcpy( )

vkCmdCopyImage( )

Texture Sampling Hardware

RGBA to the Shader
## Memory Types

### 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
- 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
Texture Sampling Parameters

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

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

  ... 

result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, pTextureSampler );
```
Textures’ Undersampling Artifacts

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

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

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

* Latin: *multim in parvo*, “many things in a small place”
VkResult
Init07TextureSampler( MyTexture * pMyTexture )
{
    VkResult result;
    VkSamplerCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    #ifdef CHOICES
    VK_SAMPLER_ADDRESS_MODE_REPEAT
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
    #endif
    vsci.mipLodBias = 0.;
    vsci.anisotropyEnable = VK_FALSE;
    vsci.maxAnisotropy = 1.;
    vsci.compareEnable = VK_FALSE;
    vsci.compareOp = VK_COMPARE_OP_NEVER;
    #ifdef CHOICES
    VK_COMPARE_OP_NEVER
    VK_COMPARE_OP_LESS
    VK_COMPARE_OP_EQUAL
    VK_COMPARE_OP_LESS_OR_EQUAL
    VK_COMPARE_OP_GREATER
    VK_COMPARE_OP_NOT_EQUAL
    VK_COMPARE_OP_GREATER_OR_EQUAL
    VK_COMPARE_OP_ALWAYS
    #endif
    vsci.minLod = 0.;
    vsci.maxLod = 0.;
    vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
    #ifdef CHOICES
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
    VK_BORDER_COLOR_INT_OPAQUE_BLACK
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE
    VK_BORDER_COLOR_INT_OPAQUE_WHITE
    #endif
    vsci.unnormalizedCoordinates = VK_FALSE;  // VK_TRUE means we are using raw texels as the index
    // VK_FALSE means we are using the usual 0. - 1.
    result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler );
}
VkResult
Init07TextureBuffer(INOUT MyTexture * pMyTexture)
{
    VkResult result;

    uint32_t texWidth = pMyTexture->width;
    uint32_t texHeight = pMyTexture->height;
    unsigned char *texture = pMyTexture->pixels;

    VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each

    VkImage stagingImage;
    VkImage textureImage;

    // *******************************************************************************
    // this first {...} is to create the staging image:
    // *******************************************************************************
    {
        VkImageCreateInfo vici;
        vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
        vici.pNext = nullptr;
        vici.flags = 0;
        vici.imageType = VK_IMAGE_TYPE_2D;
        vici.format = VK_FORMAT_R8G8B8A8_UNORM;
        vici.extent.width = texWidth;
        vici.extent.height = texHeight;
        vici.extent.depth = 1;
        vici.mipLevels = 1;
        vici.arrayLayers = 1;
        vici.samples = VK_SAMPLE_COUNT_1_BIT;
        vici.tiling = VK_IMAGE_TILING_LINEAR;

#ifdef CHOICES
    VK_IMAGE_TILING_OPTIMAL
    VK_IMAGE_TILING_LINEAR
#endif
        vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;

#ifdef CHOICES
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT
    VK_IMAGE_USAGE_TRANSFER_DST_BIT
    VK_IMAGE_USAGE_SAMPLED_BIT
    VK_IMAGE_USAGE_STORAGE_BIT
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
#endif
        vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    }
```c
#ifdef CHOICES
#define VK_IMAGE_LAYOUT_UNDEFINED
#define VK_IMAGE_LAYOUT_PREINITIALIZED
#endif

vici.queueFamilyIndexCount = 0;
vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

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

if (Verbose)
{
    fprintf(FpDebug, "Image vmr.size = %lld\n", vmr.size);
    fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
    fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    fflush(FpDebug);
}

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible(); // because we want to mmap it

VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
pMyTexture->vdm = vdm;

result = vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0); // 0 = offset

// we have now created the staging image -- fill it with the pixel data:

VkImageSubresource vis;
vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
vis.mipLevel = 0;
vis.arrayLayer = 0;

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

if (Verbose)
{
    fprintf(FpDebug, "Subresource Layout:\n\n");
    fprintf(FpDebug, "\toffset = %lld\n", vsl.offset);
    fprintf(FpDebug, "\tsize = %lld\n", vsl.size);
    fprintf(FpDebug, "\trowPitch = %lld\n", vsl.rowPitch);
    fprintf(FpDebug, "\tarrayPitch = %lld\n", vsl.arrayPitch);
    fprintf(FpDebug, "\tdepthPitch = %lld\n", vsl.depthPitch);
    fflush(FpDebug);
}
```

void * gpuMemory;

vkMapMemory( LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
// 0 and 0 = offset and memory map flags

if (vsl.rowPitch == 4 * texWidth)
{
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
}
else
{
    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) );
    }
}

vkUnmapMemory( LogicalDevice, vdm);

// ******************************************************************************
// this second {...} is to create the actual texture image:

{
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_OPTIMAL;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
    // because we are transferring into it and will eventual sample from it
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

    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\n", vmr.size );
        fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment );
        fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );
        fflush( FpDebug );
    }

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

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

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

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

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

result = vkBeginCommandBuffer( TextureCommandBuffer, IN &vcbbi);

// *******************************************************************************
// transition the staging buffer layout:
// *******************************************************************************
{
  VkImageSubresourceRange visr;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;

  VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = stagingImage;
    vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = visr;

  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:
// **************************************************************************
{
  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 = ... = textureImage;
  vimb.srcAccessMask = 0;
  vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  vimb.subresourceRange = visr;

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

  VkCmdPipelineBarrier( TextureCommandBuffer,
                        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
                        0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr,
                        1, IN &vimb);

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

  VkOffset3D vo3;
  vo3.x = 0;
  vo3.y = 0;
  vo3.z = 0;

  VkExtent3D ve3;
  ve3.width = texWidth;
  ve3.height = texHeight;
  ve3.depth = 1;
VkImageCopy
    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_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = textureImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
    vimb.subresourceRange = visr;
    
    vkCmdPipelineBarrier(TextureCommandBuffer,  
        VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,  0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

    result = vkEndCommandBuffer( TextureCommandBuffer );
}

VkSubmitInfo vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &TextureCommandBuffer;
    vsi.waitSemaphoreCount = 0;
    vsi.pWaitSemaphores = (VkSemaphore *)nullptr;
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
    vsi.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.
typedef struct MyTexture
{
    uint32_t width;
    uint32_t height;
    VkImage texImage;
    VkImageView texImageView;
    VkSampler texSampler;
    VkDeviceMemory vdm;
} MyTexture;

MyTexture MyPuppyTexture;

result = Init06TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyTexturePuppy);
Init06TextureSampler( &MyPuppyTexture.texSampler );

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

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Vulkan: Overall Block Diagram
Vulkan Queues and Command Buffers

- Graphics commands are recorded in command buffers, e.g., `vkCmdDoSomething(cmdBuffer, …);`
- You can have as many simultaneous Command Buffers as you want
- Each command buffer can be filled from a different thread
- Command Buffers record 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
uint32_t  count;

vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr );

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];

vkGetPhysicalDeviceFamilyProperties( 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;
    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;
}
```
float queuePriorities[] =
{
    1. // one entry per queueCount
};

VkDeviceQueueCreateInfo vdqci[1];
    vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
    vdqci[0].pNext = nullptr;
    vdqci[0].flags = 0;
    vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    vdqci[0].queueCount = 1;
    vdqci[0].queuePriorities = (float *) queuePriorities;

VkDeviceCreateInfo vdci;
    vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
    vdci.pNext = nullptr;
    vdci.flags = 0;
    vdci.queueCreateInfoCount = 1; // # of device queues wanted
    vdci.pQueueCreateInfos = (const VkDeviceQueueCreateInfo *&)vdqci[0]; // array of VkDeviceQueueCreateInfo's
    vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
    vdci.ppEnabledLayerNames = myDeviceLayers;
    vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
    vdci.ppEnabledExtensionNames = myDeviceExtensions;
    vdci.pEnabledFeatures = &PhysicalDeviceFeatures; // already created

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

VkQueue Queue;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
uint32_t queueIndex = 0;

result = vkGetDeviceQueue(LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue);
Creating the Command Pool as part of the Logical Device

```c
VkResult
Init06CommandPool( )
{
    VkResult result;

    VkCommandPoolCreateInfo vcpci;
    vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcpci.pNext = nullptr;
    vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT |
                  VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;

    #ifdef CHOICES
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
    #endif
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );

    result = vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool );

    return result;
}
```
Creating the Command Buffers

```cpp
VkResult Init06CommandBuffers()
{
    VkResult result;

    // allocate 2 command buffers for the double-buffered rendering:

    VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 2; // 2, because of double-buffering

    result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &CommandBuffers[0]);
}

// allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:

VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 1;

    result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer);
}

return result;
}
```
Beginning a Command Buffer – One per Image

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

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

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

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

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

...  
vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```
Beginning a Command Buffer

- `vkBeginCommandBuffer()`
  - `VkCommandBufferAllocateInfo`
    - `vkCreateCommandBufferPool()`
      - `VkCommandBufferPoolCreateInfo`
      - `VkCommandBufferAllocateInfo`
        - `vkAllocateCommandBuffer()`
          - `VkCommandBufferBeginInfo`
These are the Commands that could be entered into the Command Buffer, 1

vkCmdBeginQuery( commandBuffer, flags );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexType );
vkCmdBindPipeline( commandBuffer, pipeline );
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
vkCmdBlitImage( commandBuffer, filter );
vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );
vkCmdClearColorImage( commandBuffer, pRanges );
vkCmdClearDepthStencilImage( commandBuffer, pRanges );
vkCmdClearColorImageToBuffer( commandBuffer, pRegions );
vkCmdClearDepthStencilImageToBuffer( commandBuffer, pRegions );
vkCmdCopyBufferToImage( commandBuffer, pRanges );
vkCmdCopyQueryPoolResults( commandBuffer, flags );
vkCmdDebugMarkerBeginEXT( commandBuffer, pMarkerInfo );
vkCmdDebugMarkerEndEXT( commandBuffer );
vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );
vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );
vkCmdDispatchIndirect( commandBuffer, offset );
vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );
vkCmdDrawIndexedIndirect( commandBuffer, stride );
vkCmdDrawIndexedIndirectCountAMD( commandBuffer, stride );
vkCmdDrawIndirect( commandBuffer, stride );
vkCmdDrawIndirectCountAMD( commandBuffer, stride );
vkCmdEndQuery( commandBuffer, query );
vkCmdEndRenderPass( commandBuffer );
vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );
These are the Commands that could be entered into the Command Buffer, II

vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pBufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pBufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
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, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

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

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

    result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
VkClearColorValue
   vccv.float32[0] = 0.0;
vccv.float32[1] = 0.0;
vccv.float32[2] = 0.0;
vccv.float32[3] = 1.0;

VkClearDepthStencilValue
   vcdsv.depth = 1.f;
vcdsv.stencil = 0;

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

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

VkRenderPassBeginInfo
   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; // used for VK_ATTACHMENT_LOAD_OP_CLEAR
   vrpbi.pClearValues = vcv;

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

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

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

vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
                         GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
                         // dynamic offset count, dynamic offsets

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);               // 0, 1 = firstBinding, bindingCount

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

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
Submitting a Command Buffer to a Queue for Execution

VkSubmitInfo vsi;
  vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  vsi.pNext = nullptr;
  vsi.commandBufferCount = 1;
  vsi.pCommandBuffers = &CommandBuffer;
  vsi.waitSemaphoreCount = 1;
  vsi.pWaitSemaphores = imageReadySemaphore;
  vsi.signalSemaphoreCount = 0;
  vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
  vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
The Entire Submission / Wait / Display Process

```
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;

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

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

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

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

vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );

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

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```
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.
The Swap Chain

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
How OpenGL Thinks of Framebuffers

Update

Depth

Back

Front

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

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

Swap Chains are arranged as a ring buffer.

Swap Chains are tightly coupled to the window system.

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

1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to “front buffer”)
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "\nvkGetPhysicalDeviceSurfaceCapabilitiesKHR:\n" );

VkBool32 supported;
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics( ), Surface, &supported );
if( supported == VK_TRUE )
  fprintf( FpDebug, "** This Surface is supported by the Graphics Queue **\n" );

uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
fprintf( FpDebug, "\nFound %d Surface Formats:\n", formatCount )

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "\nFound %d Present Modes:\n", presentModeCount );
## VulkanDebug.txt output:

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

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

Found 2 Surface Formats:

<table>
<thead>
<tr>
<th>Format Index</th>
<th>Format</th>
<th>Sample Format</th>
<th>Color Space</th>
<th>Non-Linear Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
<td>0</td>
<td>VK_FORMAT_B8G8R8A8_UNORM</td>
<td>VK_COLOR_SPACE_SRGB_NONLINEAR_KHR</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>0</td>
<td>VK_FORMAT_B8G8R8A8_SRGB</td>
<td>VK_COLOR_SPACE_SRGB_NONLINEAR_KHR</td>
</tr>
</tbody>
</table>

Found 3 Present Modes:

<table>
<thead>
<tr>
<th>Present Mode Index</th>
<th>Present Mode</th>
<th>Color Space</th>
<th>Non-Linear Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>VK_PRESENT_MODE_FIFO_KHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>VK_PRESENT_MODE_FIFO_RELAXED_KHR</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>VK_PRESENT_MODE_MAILBOX_KHR</td>
<td></td>
</tr>
</tbody>
</table>
Creating a Swap Chain

vkGetDevicePhysicalSurfaceCapabilities()

VkSurfaceCapabilities

surface
imageFormat
imageColorSpace
imageExtent
imageArrayLayers
imageUsage
imageSharingMode
preTransform
compositeAlpha
presentMode
clipped

VkSwapchainCreateInfo

minImageCount
maxImageCount
currentExtent
minImageExtent
maxImageExtent
maxImageArrayLayers
supportedTransforms
currentTransform
supportedCompositeAlpha

vkCreateSwapchain()

vkGetSwapChainImages()

vkCreateImageView()
Creating a Swap Chain

```c
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;

VkSwapchainCreateInfoKHR vscci;
    vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
    vscci.pNext = nullptr;
    vscci.flags = 0;
    vscci.surface = Surface;
    vscci.minImageCount = 2; // double buffering
    vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
    vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
    vscci.imageExtent.width = surfaceRes.width;
    vscci.imageExtent.height = surfaceRes.height;
    vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
    vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
    vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
    vscci.imageArrayLayers = 1;
    vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vscci.queueFamilyIndexCount = 0;
    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 );
```
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 ... = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;
    vivci.image = PresentImages[ i ];

    result = vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[ i ] );
}
Rendering into the Swap Chain, I

```cpp
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 ] );
```
VkFenceCreateInfo \( \text{vfci}; \)
\begin{verbatim}
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;
\end{verbatim}

VkFence renderFence;
\text{vkCreateFence}( \text{LogicalDevice}, &\text{vfci}, \text{PALLOCATOR}, \text{OUT} &renderFence );

VkQueue presentQueue;
\text{vkGetDeviceQueue}( \text{LogicalDevice}, \text{FindQueueFamilyThatDoesGraphics( )}, 0,
\text{OUT} &\text{presentQueue} );

\ldots

VkSubmitInfo \( \text{vsi}; \)
\begin{verbatim}
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &\text{imageReadySemaphore};
vsi.pWaitDstStageMask = &\text{waitAtBottom};
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &\text{CommandBuffers[ nextImageIndex ]};
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = &\text{SemaphoreRenderFinished};
\end{verbatim}

\text{result} = \text{vkQueueSubmit}( \text{presentQueue}, 1, \text{IN} &\text{vsi}, \text{IN} \text{renderFence} ); \quad // 1 = \text{submitCount}
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );

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

result = vkQueuePresentKHR( presentQueue, IN &vpi );
Push Constants

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
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.
Creating a Pipeline

**vkCreatePipelineLayout()**

- VkPipelineLayoutCreateInfo
  - basePipelineHandle
  - basePipelineIndex

**vkCreateGraphicsPipeline()**

- VkGraphicsPipelineCreateInfo
  - shaderStageCreateInfos
  - vertexInputStateCreateInfo
  - viewportStateCreateInfo
  - rasterizationStateCreateInfo
  - inputAssemblyStateCreateInfo
  - tesselationStateCreateInfo
  - colorBlendStateCreateInfo
  - multiSampleStateCreateInfo
  - depthStencilStateCreateInfo
  - dynamicStateCreateInfo

- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkPipelineInputAssemblyStateCreateInfo
- VkPipelineTessellationStateCreateInfo
- VkPipelineViewportStateCreateInfo
- VkPipelineRasterizationStateCreateInfo
- VkPipelineColorBlendStateCreateInfo
- VkPipelineDepthStencilStateCreateInfo
- VkPipelineDynamicStateCreateInfo

- VkSpecializationInfo
- VkShaderModule

- VkPipelineBindPoint
- VkPipelineLayout
- VkRenderPass

- VkPipelineColorBlendAttachmentState
- VkPipelineColorBlendStateCreateInfo

- VkDevice
- VkPipeline

**Push Constants**

- VkPushConstantRange

**Descriptor Set Layouts**

- VkDescriptorSetLayoutCreateInfo
- VkDescriptorSetLayout

**Push Constants**

- VkPushConstantShape

**Graphics Pipeline**

- VkPipeline

**Array naming the states that can be set dynamically**

- VkPipelineDynamicStateCreateInfo
  - depthTestEnable
  - depthWriteEnable
  - depthCompareOp
  - stencilTestEnable
  - stencilOpStateFront
  - stencilOpStateBack
  - blendEnable
  - srcColorBlendFactor
  - dstColorBlendFactor
  - colorBlendOp
  - srcAlphaBlendFactor
  - dstAlphaBlendFactor
  - alphaBlendOp
  - colorWriteMask

- VkViewport
- VkScissor

- VkDevice
- VkPipeline

**Location binding format offset**

- VkSampler
- VkSamplerCreateInfo

**VkVertexInputBindDescription**

- binding
- location
- format
- offset

**VkVertexInputAttributeDescription**

- binding
- location
- stride
- inputRate

**Viewport**

- x, y, w, h
- minDepth, maxDepth

**Scissor**

- x, y, w, h
- offset
- extent

**Topology**

- cullMode
- polygonMode
- frontFace
- lineWidth
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`. 
Setting up the Push Constants for the Pipeline Structure

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, IN &vplci, PALLOCATOR,
                                 OUT &GraphicsPipelineLayout);
```
An Robotic Example using Push Constants

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

Where each arm is represented by:

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

struct armArm1;
struct armArm2;
struct armArm3;
```
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 $T_{1/G}$

\[
\begin{bmatrix}
M_{1/G}
\end{bmatrix} = \begin{bmatrix}
T_{1/G}
\end{bmatrix} \ast \begin{bmatrix}
R_{\theta_1}
\end{bmatrix}
\]
Why Do We Say it Right-to-Left?

We adopt the convention that the coordinates are multiplied on the right side of the matrix:

\[
\left[ \begin{array}{c}
    x' \\
    y' \\
    z' \\
    1
\end{array} \right] = \left[ \begin{array}{cccc}
    T_{1/G} \\
    R_{\theta_1}
\end{array} \right] \cdot \left[ \begin{array}{c}
    x \\
    y \\
    z \\
    1
\end{array} \right]
\]

So the right-most transformation in the sequence multiplies the \((x,y,z,1)\) *first* and the left-most transformation multiples it *last*.
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

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

Say it

$\begin{bmatrix} M_{2/G} \end{bmatrix} = \begin{bmatrix} M_{1/G} \end{bmatrix} * \begin{bmatrix} M_{2/1} \end{bmatrix}$
Positioning Part #3 With Respect to Ground

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

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

\[
[M_{3/G}] = [M_{1/G}] \ast [M_{2/1}] \ast [M_{3/2}]
\]
In the Reset Function

```cpp
struct arm                      Arm1;
struct arm                      Arm2;
struct arm                      Arm3;

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

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

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

The constructor `glm::mat4( 1. )` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`.
Setup the Push Constant for the Pipeline Structure

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

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

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

```c++
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

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

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

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

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

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

The strategy is to draw each link using the same vertex buffer, but modified with a unique color, length, and matrix transformation.
layout( push_constant ) uniform arm
{
    mat4 armMatrix;
    vec3 armColor;
    float armScale; // scale factor in x
}
RobotArm;

layout( location = 0 ) in vec3 aVertex;

vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
bVertex.x += 1.; // now is [0., 2.]
bVertex.x /= 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: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Queue

Queue

Queue

Queue

Queue

Queue

Command Buffer

Command Buffer

Command Buffer

mjb – July 24, 2020
Querying the Number of Physical Devices

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

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

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

<table>
<thead>
<tr>
<th>How many total there are</th>
<th>Where to put them</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, &amp;count, nullptr );</td>
<td></td>
</tr>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, &amp;count, physicalDevices );</td>
<td></td>
</tr>
</tbody>
</table>
VkResult result = VK_SUCCESS;

result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 )
{
    fprintf( FpDebug, "Could not count the physical devices\n" );
    return VK_SHOULD_EXIT;
}

fprintf(FpDebug, "%d physical devices found.\n", PhysicalDeviceCount);

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ PhysicalDeviceCount ];
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );
if( result != VK_SUCCESS )
{
    fprintf( FpDebug, "Could not enumerate the %d physical devices\n", PhysicalDeviceCount );
    return VK_SHOULD_EXIT;
}
int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    if( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not get the physical device properties of device %d\n", i );
        return VK_SHOULD_EXIT;
    }
    fprintf( FpDebug, \n\nDevice %2d:\n", i );
    fprintf( FpDebug, "\tAPI version: %d\n", vpdp.apiVersion );
    fprintf( FpDebug, "\tDriver version: %d\n", vpdp.apiVersion );
    fprintf( FpDebug, "\tVendor ID: 0x%04x\n", vpdp.vendorID );
    fprintf( FpDebug, "\tDevice ID: 0x%04x\n", vpdp.deviceID );
    fprintf( FpDebug, "\tPhysical Device Type: %d =", vpdp.deviceType );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        fprintf( FpDebug, " (Discrete GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        fprintf( FpDebug, " (Integrated GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
        fprintf( FpDebug, " (Virtual GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
        fprintf( FpDebug, " (CPU)\n" );
    fprintf( FpDebug, "\tDevice Name: %s\n", vpdp.deviceName );
    fprintf( FpDebug, "\tPipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0] );
}
// need some logical here to decide which physical device to select:

if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
    discreteSelect = i;

if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
    integratedSelect = i;

int which = -1;
if( discreteSelect >= 0 )
    { which = discreteSelect;
      PhysicalDevice = physicalDevices[which];
    }
else if( integratedSelect >= 0 )
    { which = integratedSelect;
      PhysicalDevice = physicalDevices[which];
    }
else
    { fprintf( FpDebug, "Could not select a Physical Device\n" );
      return VK_SHOULD_EXIT;
    }
Asking About the Physical Device’s Features

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

fprintf( FpDebug, "\nPhysical 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: 4198499
  Driver version: 4198499
  Vendor ID: 0x10de
  Device ID: 0x1e04
  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

```java
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

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

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

fprintf( FpDebug, "\n%d Memory Heaps:\n", vpdmp.memoryHeapCount );
for( unsigned int  i = 0; i < vpdmp.memoryHeapCount; i++ )
{
    fprintf(FpDebug, "Heap %d: ", i);
    VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
    fprintf( FpDebug, " size = 0x%08lx", (unsigned long int)vmh.size );
    if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT  ) != 0 )     fprintf( FpDebug, " DeviceLocal" );  // only one in use
    fprintf(FpDebug, "\n");
}
```
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
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
fprintf(FpDebug, "\nFound %d Queue Families:\n", count);

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT vqfp);
for( unsigned int i = 0; i < count; i++ )
{
    fprintf(FpDebug, "\t%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, "\n");
}
Here’s What I Got

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

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Vulkan: Overall Block Diagram
Vulkan: a More Typical (and Simplified) Block Diagram

1. Application
2. Instance
3. Physical Device
4. Logical Device
5. Queue
   - Command Buffer
   - Command Buffer
   - Command Buffer
Looking to See What Device Layers are Available

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

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

// see what device layers are available:
uint32_t  layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, deviceLayers);
```
Looking to See What Device Extensions are Available

```c
// see what device extensions are available:

uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
                                      &extensionCount, (VkExtensionProperties *)nullptr);

VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];

result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
                                                  &extensionCount, deviceExtensions);
```
4 physical device layers enumerated:

0x00401063  1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
  0 device extensions enumerated for 'VK_LAYER_NV_optimus':

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'
float queuePriorities[1] =
{
    1.
};
VkDeviceQueueCreateInfo vdqci;
    vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    vdqci.pNext = nullptr;
    vdqci.flags = 0;
    vdqci.queueFamilyIndex = 0;
    vdqci.queueCount = 1;
    vdqci.pQueueProperties = queuePriorities;

result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
// get the queue for this logical device:

vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue ); // 0, 0 = queueFamilyIndex, queueIndex
Dynamic State Variables

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

- VkGraphicsPipelineCreateInfo
  - Shader stages (VERTEX, etc.)
  - binding
  - stride
  - inputRate
- VkShaderModule
  - VkPipelineShaderStageCreateInfo
  - VkVertexInputBindingDescription
    - which stage (VERTEX, etc.)
    - binding
    - stride
    - inputRate
- VkVertexInputAttributeDescription
  - which stage (VERTEX, etc.)
  - binding
  - stride
  - inputRate
  - location
- VkPipelineInputAssemblyStateCreateInfo
  - VkPipelineVertexInputStateCreateInfo
  - VkViewportStateCreateInfo
    - x, y, w, h
    - minDepth
    - maxDepth
    - offset
    - extent
- VkPipelineRasterizationStateCreateInfo
  - VkPipelineDepthStencilStateCreateInfo
  - VkPipelineColorBlendStateCreateInfo
    - depthTestEnable
    - depthWriteEnable
    - depthCompareOp
    - stencilTestEnable
    - stencilOpStateFront
    - stencilOpStateBack
    - blendEnable
    - srcColorBlendFactor
    - dstColorBlendFactor
    - colorBlendOp
    - srcAlphaBlendFactor
    - dstAlphaBlendFactor
    - alphaBlendOp
    - colorWriteMask
- VkPipelineColorBlendAttachmentState
  - VkPipelineDynamicStateCreateInfo
  - Array naming the states that can be set dynamically
  - VkPipelineDynamicStateCreateInfo
  - VkPipelineInputAssemblyStateCreateInfo
  - Viewport
  - Scissor
  - Topology
  - cullMode
  - polygonMode
  - frontFace
  - lineWidth

vkCreateGraphicsPipeline()
Which Pipeline State Variables can be Changed Dynamically

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

```c
VK_DYNAMIC_STATE_VIEWPORT
VK_DYNAMIC_STATE_SCISSOR
VK_DYNAMIC_STATE_LINE_WIDTH
VK_DYNAMIC_STATE_DEPTH_BIAS
VK_DYNAMIC_STATE_BLEND_CONSTANTS
VK_DYNAMIC_STATE_DEPTH_BOUNDS
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
VK_DYNAMIC_STATE_STENCIL_REFERENCE
```
Creating a Pipeline

```cpp
VkDynamicState
{
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo vpdsci;
    vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
    vpdsci.pNext = nullptr;
    vpdsci.flags = 0;
    vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState); // i.e., 2
    vpdsci.pDynamicStates = &vds;

VkGraphicsPipelineCreateInfo vgpci;
    . . .
    vgpci.pDynamicState = &vpdsci;
    . . .

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

If you declare certain state variables to be dynamic like this, then you **must** fill them in the command buffer! Otherwise, they are **undefined**.
Filling the Dynamic State Variables in the Command Buffer

First call:

\[
\text{vkCmdBindPipeline( \ldots );}
\]

Then, the command buffer-bound function calls to set these dynamic states are:

\[
\text{vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );}
\]
\[
\text{vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );}
\]
\[
\text{vkCmdSetLineWidth( commandBuffer, linewidth );}
\]
\[
\text{vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );}
\]
\[
\text{vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );}
\]
\[
\text{vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );}
\]
\[
\text{vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );}
\]
\[
\text{vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );}
\]
\[
\text{vkCmdSetStencilReference( commandBuffer, faceMask, reference );}
\]
Getting Information Back from the Graphics System

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
• There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp

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

• This indicates that Vulkan thinks that Queries are time-consuming (relatively) to setup, and thus better to set them up in program-setup than in program-runtime
 VkQueryPoolCreateInfo  
   vqpci;  
   vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;  
   vq pci.pNext = nullptr;  
   vqpci.flags = 0;  
   vqpci.queryType = << one of: >>  
       VK_QUERY_TYPE_OCCLUSION  
       VK_QUERY_TYPE_PIPELINE_STATISTICS  
       VK_QUERY_TYPE_TIMESTAMP  
   vqpci.queryCount = 1;  
   vqpci.pipelineStatistics = 0;  
   \( \text{bitmask of what stats you are querying for if you are doing a pipeline statistics query} \)  

 VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT  
 VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT  
 VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT  
 VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT  
 VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT  
 VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT  
 VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT  
 VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT  
 VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT  
 VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT  
 VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT  

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

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

 VkQueryPool timestampQueryPool;  
 result = vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &timestampQueryPool );
vkCmdResetQueryPool( CommandBuffer, \textit{occlusionQueryPool}, 0, 1 );

vkCmdBeginQuery( CommandBuffer, \textit{occlusionQueryPool}, 0, VK_QUERY_CONTROL_PRECISE_BIT );

\ldots

vkCmdEndQuery( CommandBuffer, \textit{occlusionQueryPool}, 0 );

\#define DATASIZE 128
uint32_t data[DATASIZE];

result = vkGetQueryPoolResults( LogicalDevice, \textit{occlusionQueryPool}, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags );

// or'ed combinations of:
// VK_QUERY_RESULT_64_BIT
// VK_QUERY_RESULT_WAIT_BIT
// VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
// VK_QUERY_RESULT_PARTIAL_BIT
// stride is \# of bytes in between each result
Occlusion Queries count the number of fragments drawn between the `vkCmdBeginQuery` and the `vkCmdEndQuery` that pass both the Depth and Stencil tests.

This is commonly used to see what level-of-detail should be used when drawing a complicated object.

**Some hints:**

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

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

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

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

// vqpci.pipelineStatistics = or’ed bits of:
// VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT
// VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT
// VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT
```
Timestamp Query

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

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

The `vkCmdWriteTimeStam` function produces the time between when this function is called and when the first thing reaches the specified pipeline stage.

Even though the stages are “bits”, you are supposed to only specify one of them, not “or” multiple ones together.

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

// VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
// VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
// VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
// VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
// ... VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
// VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
// VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
// VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
// VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
// VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
// VK_PIPELINE_STAGE_TRANSFER_BIT
// VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
// VK_PIPELINE_STAGE_HOST_BIT
Compute Shaders

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
The Example We Are Going to Use Here is a *Particle System*

The Compute Shader Moves the Particles by Recomputing the Position and Velocity Buffers

The Rendering Draws the Particles by Reading the Position and Color Buffers
#define NUM_PARTICLES (1024*1024) // total number of particles to move
#define NUM_WORK_ITEMS_PER_GROUP 64 // # work-items per work-group
#define NUM_X_WORK_GROUPS (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP )

struct pos
{
    glm::vec4; // positions
};

struct vel
{
    glm::vec4; // velocities
};

struct col
{
    glm::vec4; // colors
};

Note that .w and .vw are not actually needed. But, by making these structure sizes a multiple of 4 floats, it doesn’t matter if they are declared with the std140 or the std430 qualifier. I think this is a good thing.
The Data in your Compute Shader will look like This

```cpp
layout( std140, set = 0, binding = 0 ) buffer Pos
{
    vec4 Positions[ ];  // array of structures
};

layout( std140, set = 0, binding = 1 ) buffer Vel
{
    vec4 Velocities[ ];  // array of structures
};

layout( std140, set = 0, binding = 2 ) buffer Col
{
    vec4 Colors[ ];  // array of structures
};
```

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when Vulkan examines the size of this buffer’s data store.
Remember the Graphics Pipeline Data Structure?

- VkPipelineLayoutCreateInfo
- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkPipelineInputAssemblyStateCreateInfo
- VkPipelineRasterizationStateCreateInfo
- VkPipelineDepthStencilStateCreateInfo
- VkPipelineColorBlendStateCreateInfo
- VkPipelineDynamicStateCreateInfo

Wall of Text: (content not included in this representation due to the nature of the image)
Here is how you create a Compute Pipeline Data Structure

- **VkPipelineLayoutCreateInfo**
  - `vkCreatePipelineLayout()`
  - `VkPipelineLayoutCreateInfo`
  - `VkPipelineLayoutCreateInfo`
  - `Push Constants`
  - `Descriptor Set Layouts`
  - `VkPipelineShaderStageCreateInfo`
  - `VkSpecializationInfo`
  - `VkShaderModule`
  - `which stage (COMPUTE)`

- **VkComputePipelineCreateInfo**
  - `vkCreateComputePipelines()`
  - `VkComputePipelineCreateInfo`

Highlighted boxes are ones that the Graphics Pipeline Data Structure also has

Note how less complicated this is!
A Reminder about Data Buffers

vkCreateBuffer(
LogicalDevice, VkBufferCreateInfo
bufferUsage, queueFamilyIndices, size (bytes)
)

vkGetBufferMemoryRequirements(
Buffer, VkMemoryAllocateInfo
memoryType, size
)

vkAllocateMemory(
LogicalDevice, VkMemoryAllocateInfo
)

vkBindBufferMemory(
bufferMemoryHandle
)

vkMapMemory(
gpuAddress
)
Creating a Shader Storage Buffer

```c
VkBuffer PosBuffer;

...  

VkBufferCreateInfo vbc;
  vbc.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  vbc.pNext = nullptr;
  vbc.flags = 0;
  vbc.size = NUM_PARTICLES * sizeof(glm::vec4);
  vbc.usage = VK_USAGE_STORAGE_BUFFER_BIT;
  vbc.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
  vbc.queueFamilyIndexCount = 0;
  vbc.pQueueFamilyIndices = (const int32_t)nullptr;

result = vkCreateBuffer ( LogicalDevice, IN &vbc, PALLOCATOR, OUT &PosBuffer );
```
Allocating Memory for a Buffer, Binding a Buffer to Memory, and Filling the Buffer

```c
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, PosBuffer, OUT &vmr );

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

  

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

result = vkBindBufferMemory( LogicalDevice, PosBuffer, IN vdm, 0 ); // 0 is the offset
```
Create the Compute Pipeline Layout

```c
VkDescriptorSetLayoutBinding ComputeSet[3];
ComputeSet[0].binding = 0;
ComputeSet[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
ComputeSet[0].descriptorCount = 1;
ComputeSet[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[0].pImmutableSamplers = (VkSampler *)nullptr;
ComputeSet[1].binding = 1;
ComputeSet[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
ComputeSet[1].descriptorCount = 1;
ComputeSet[1].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[1].pImmutableSamplers = (VkSampler *)nullptr;
ComputeSet[2].binding = 2;
ComputeSet[2].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
ComputeSet[2].descriptorCount = 1;
ComputeSet[2].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[2].pImmutableSamplers = (VkSampler *)nullptr;

VkDescriptorSetLayoutCreateInfo vdslc;
vdslc.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc.pNext = nullptr;
vdslc.flags = 0;
vdslc.bindingCount = 3;
vdslc.pBindings = &ComputeSet[0];
```
Create the Compute Pipeline Layout

```
VkPipelineLayout  ComputePipelineLayout;
VkDescriptorSetLayout  ComputeSetLayout;

... 

result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc, PALLOCATOR, OUT &ComputeSetLayout );

VkPipelineLayoutCreateInfo  vplci;
   vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
   vplci.pNext = nullptr;
   vplci.flags = 0;
   vplci.setLayoutCount = 1;
   vplci.pSetLayouts = ComputeSetLayout;
   vplci.pushConstantRangeCount = 0;
   vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout );
```
Create the Compute Pipeline

```
VkPipeline
ComputePipeline;

VkPipelineShaderStageCreateInfo vpssci;
vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0;
vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;

result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline );
```
### Creating a Vulkan Data Buffer

```c
VkBuffer Buffer;

VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = NUM_PARTICLES * sizeof(glm::vec4);
    vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const int32_t *)nullptr;

result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &posBuffer );
```
Allocating Memory and Binding the Buffer

```c
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, posBuffer, OUT &vmr );

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

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

result = vkBindBufferMemory( LogicalDevice, posBuffer, IN vdm, 0 ); // 0 is the offset

MyBuffer myPosBuffer;
    myPosBuffer.size   = vbci.size;
    myPosBuffer.buffer = PosBuffer;
    myPosBuffer.vdm   = vdm;
```
struct pos * positions;

vkMapMemory( LogicalDevice, IN myPosBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &positions );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    positions[ i ].x = Ranf( XMIN, XMAX );
    positions[ i ].y = Ranf( YMIN, YMAX );
    positions[ i ].z = Ranf( ZMIN, ZMAX );
    positions[ i ].w = 1.;
}

vkUnmapMemory( LogicalDevice, IN myPosBuffer.vdm );

struct vel * velocities;

vkMapMemory( LogicalDevice, IN myVelBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &velocities );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    velocities[ i ].x = Ranf( VXMIN, VXMAX );
    velocities[ i ].y = Ranf( VYMIN, VYMAX );
    velocities[ i ].z = Ranf( VZMIN, VZMAX );
    velocities[ i ].w = 0.;
}

vkUnmapMemory( LogicalDevice, IN myVelBuffer.vdm );

struct col * colors;

vkMapMemory( LogicalDevice, IN myColBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &colors );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    colors[ i ].r = Ranf( .3f, 1. );
    colors[ i ].g = Ranf( .3f, 1. );
    colors[ i ].b = Ranf( .3f, 1. );
    colors[ i ].a = 1.;
}

vkUnmapMemory( LogicalDevice, IN myColBuffer.vdm );
```
#include <stdlib.h>
#define TOP     2147483647.             // 2^31 - 1

float
Ranf( float low, float high )
{
    long random( );           // returns integer 0 - TOP
    float r = (float)rand( );
    return low + r * ( high - low ) / (float)RAND_MAX ;
}
```
The Particle System Compute Shader

```c
layout( std140, set = 0, binding = 0 ) buffer Pos
{
    vec4 Positions[ ]; // array of structures
};

layout( std140, set = 0, binding = 1 ) buffer Vel
{
    vec4 Velocities[ ]; // array of structures
};

layout( std140, set = 0, binding = 2 ) buffer Col
{
    vec4 Colors[ ]; // array of structures
};

layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```

This is the number of **work-items per work-group**, set in the compute shader. The number of work-groups is set in the

```c
vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY, workGroupCountZ );
```

function call in the application program.
The Data gets Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

20 total items to compute:

The Invocation Space can be 1D, 2D, or 3D. This one is 1D.

\[
\text{#WorkGroups} = \frac{\text{GlobalInvocationSize}}{\text{WorkGroupSize}}
\]

\[
5 \times 4 = \frac{20}{4}
\]
The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

20x12 (=240) total items to compute:

The Invocation Space can be 1D, 2D, or 3D. This one is 2D.

\[
\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}
\]

\[
5 \times 4 = \frac{20 \times 12}{4 \times 3}
\]
A Mechanical Equivalent...

“Work Group”

“Work Items”

http://news.cision.com

mjb – July 24, 2020
```c
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4          // xc, yc, zc, r
#define PLANE vec4           // a, b, c, d

const VECTOR G = VECTOR( 0., -9.8, 0. );
const float DT = 0.1;

const SPHERE Sphere = vec4( -100., -800., 0., 600. ); // x, y, z, r

uint gid = gl_GlobalInvocationID.x; // where I am in the global dataset (6 in this example)
// (as a 1d problem, the .y and .z are both 1)

POINT p = Positions[ gid ].xyz;
VELOCITY v = Velocities[ gid ].xyz;

POINT pp = p + v*DT + .5*DT*DT*G;
VELOCITY vp = v + G*DT;

Positions[ gid ].xyz = pp;
Velocities[ gid ].xyz = vp;
```

The Particle System Compute Shader – The Physics

\[
p' = p + v \cdot t + \frac{1}{2} G \cdot t^2
\]

\[
v' = v + G \cdot t
\]
VELOCITY
Bounce( VELOCITY vin, VECTOR n )
{
    VELOCITY vout = reflect( vin, n );
    return vout;
}

// plane equation:  Ax + By + Cz + D = 0
// ( it turns out that (A,B,C) is the normal )

VELOCITY
BouncePlane( POINT p, VELOCITY v, PLANE pl)
{
    VECTOR n = normalize( VECTOR( pl.xyz ) );
    return Bounce( v, n );
}

bool
IsUnderPlane( POINT p, PLANE pl )
{
    float r = pl.x*p.x + pl.y*p.y + pl.z*p.z + pl.w;
    return ( r < 0. );
}

Note: a surface in the x-z plane has the equation: 0x + 1y + 0z + 0 = 0
and thus its normal vector is (0,1,0)
VELOCITY
BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
    VECTOR n = normalize( p - s.xyz );
    return Bounce( v, n );
}

bool
IsInsideSphere( POINT p, SPHERE s )
{
    float r = length( p - s.xyz );
    return ( r < s.w );
}
uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case

POINT p = Positions[ gid ].xyz;
VELOCITY v = Velocities[ gid ].xyz;

POINT pp = p + v*DT + .5*DT*DT*G;
VELOCITY vp = v + G*DT;

if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + .5*DT*DT*G;
}

Positions[ gid ].xyz = pp;
Velocities[ gid ].xyz = vp;

\[
\begin{align*}
    p' &= p + v \cdot t + \frac{1}{2} G \cdot t^2 \\
    v' &= v + G \cdot t
\end{align*}
\]

**Graphics Trick Alert:** Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough (and it is), nobody will ever know…
Dispatching the Compute Shader from the Command Buffer

```
#define NUM_PARTICLES (1024*1024)
#define NUM_WORK_ITEMS_PER_GROUP 64
#define NUM_X_WORK_GROUPS (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)
...
vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);
vkCmdDispatch(CommandBuffer, NUM_X_WORK_GROUPS, 1, 1);
```

This is the number of work-groups, set in the application program. The number of work-items per work-group is set in the layout in the compute shader:

```
layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```
Displaying the Particles

```c
VkVertexInputBindingDescription vvibd[3];                  // one of these per buffer data buffer
    vvibd[0].binding = 0;                                           // which binding # this is
    vvibd[0].stride = sizeof(struct pos);                      // bytes between successive structs
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

    vvibd[1].binding = 1;
    vvibd[1].stride = sizeof(struct vel);
    vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

    vvibd[2].binding = 2;
    vvibd[2].stride = sizeof(struct col);
    vvibd[2].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

layout( location = 0 ) in vec4 aPosition;
layout( location = 1 ) in vec4 aVelocity;
layout( location = 2 ) in vec4 aColor;
```
### Displaying the Particles

```
VkVertexInputAttributeDescription vviad[3];  // array per vertex input attribute
  // 3 = position, velocity, color
  vviad[0].location = 0;  // location in the layout decoration
  vviad[0].binding = 0;   // which binding description this is part of
  vviad[0].format = VK_FORMAT_VEC4;  // x, y, z, w
  vviad[0].offset = offsetof( struct pos, pos );  // 0

  vviad[1].location = 1;
  vviad[1].binding = 0;
  vviad[1].format = VK_FORMAT_VEC4;  // nx, ny, nz
  vviad[1].offset = offsetof( struct vel, vel );  // 0

  vviad[2].location = 2;
  vviad[2].binding = 0;
  vviad[2].format = VK_FORMAT_VEC4;  // r, g, b, a
  vviad[2].offset = offsetof( struct col, col );  // 0
```
Telling the Pipeline about its 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 = 3;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexAttributeDescriptionCount = 3;
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_POINT_LIST;
```

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
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 );
```
VkBufferMemoryBarrier
    vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessFlags = VK_ACCESS_SHADER_WRITE_BIT;
vbmb.dstAccessFlags = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer =
vbmb.offset = 0;
vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);

const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier
    (commandBuffer,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,
    VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount
    IN &vbmb, 0, nullptr
);
VkBuffer buffers[ ] = MyPosBuffer.buffer, MyVelBuffer.buffer, MyColBuffer.buffer;
size_t offsets[ ] = { 0, 0, 0 };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 3, buffers, offsets );

const uint32_t vertexCount = NUM_PARTICLES;
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

vkCmdDraw( CommandBuffers[nextImageIndex], NUM_PARTICLES, 1, 0, 0 );
    // vertexCount, instanceCount, firstVertex, firstInstance
Setting a Pipeline Barrier so the Compute Waits for the Drawing

```cpp
VkBufferMemoryBarrier vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessFlags = 0;
vbmb.dstAccessFlags = VK_ACCESS_UNIFORM_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer =
vbmb.offset = 0;
vbmb.size = ??

const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
    VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount
    IN &vbmb, 0, nullptr
);
```
Specialization Constants

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Remember the Compute Pipeline?

- **Descriptor Set Layouts**
- **Push Constants**
- ** VkPipelineLayoutCreateInfo**
- **which stage (COMPUTE)**
- ** VkSpecializationInfo**
- ** VkShaderModule**
- ** VkPipelineShaderStageCreateInfo**
- **Shaders**
  - Pipeline layout
  - basePipelineHandle
  - basePipelineIndex
- ** VkComputePipelineCreateInfo**
- ** vkCreatePipelineLayout( )**
- ** VkPipelineLayoutCreateInfo**
- ** Shaders**
  - Pipeline layout
  - basePipelineHandle
  - basePipelineIndex
- ** VkComputePipelineCreateInfo**
- ** vkCreateComputePipelines( )**

**Compute Pipeline**
What Are Specialization Constants?

In Vulkan, all shaders get halfway-compiled into SPIR-V and then the rest-of-the-way compiled by the Vulkan driver.

Normally, the half-way compile finalizes all constant values and compiles the code that uses them.

But, it would be nice every so often to have your Vulkan program sneak into the halfway-compiled binary and manipulate some constants at runtime. This is what Specialization Constants are for. A Specialization Constant is a way of injecting an integer, Boolean, uint, float, or double constant into a halfway-compiled version of a shader right before the rest-of-the-way compilation.

That final compilation happens when you call `vkCreateComputePipelines()`

Without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago.
Specialization Constants could be used for:

- Setting the work-items per work-group in a compute shader
- Setting a Boolean flag and then eliminating the if-test that used it
- Setting an integer constant and then eliminating the switch-statement that looked for it
- Making a decision to unroll a for-loop because the number of passes through it are small enough
- Collapsing arithmetic expressions into a single value
- Collapsing trivial simplifications, such as adding zero or multiplying by 1
Specialization Constants are Described in the Compute Pipeline
Specialization Constant Example -- Setting an Array Size

In the compute shader

```cpp
layout( constant_id = 7 ) const int ASIZE = 32;

int array[ASIZE];
```

In the Vulkan C/C++ program:

```cpp
int asize = 64;

VkSpecializationMapEntry vsme[1]; // one array element for each // Specialization Constant

vsme[0].constantID = 7; // # bytes into the Specialization Constant
vsme[0].offset = 0; // array this one item is
vsme[0].size = sizeof(asize); // size of just this Specialization Constant

VkSpecializationInfo vsi;
vs.i.mapEntryCount = 1;
vs.i.pMapEntries = &vsme[0];
vs.i.dataSize = sizeof(asize); // size of all the Specialization Constants together
vs.i.pData = &asize; // array of all the Specialization Constants
```
Linking the Specialization Constants into the Compute Pipeline

```c
int asize = 64;

VkSpecializationMapEntry vsme[1];
    vsme[0].constantID = 7;
    vsme[0].offset = 0;
    vsme[0].size = sizeof(asize);

VkSpecializationInfo vsi;
    vsi.mapEntryCount = 1;
    vsi.pMapEntries = &vsme[0];
    vsi.dataSize = sizeof(asize);
    vsi.pData = &asize;

VkPipelineShaderStageCreateInfo vpssci;
    vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci.pNext = nullptr;
    vpssci.flags = 0;
    vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
    vpssci.module = computeShader;
    vpssci.pName = "main";
    vpssci.pSpecializationInfo = &vsi;

VkComputePipelineCreateInfo vcpci[1];
    vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
    vcpci[0].pNext = nullptr;
    vcpci[0].flags = 0;
    vcpci[0].stage = vpssci;
    vcpci[0].layout = ComputePipelineLayout;
    vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
    vcpci[0].basePipelineIndex = 0;

result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, OUT &ComputePipeline );
```
Specialization Constant Example – Setting Multiple Constants

In the compute shader:

```glsl
layout( constant_id = 9 ) const int a = 1;
layout( constant_id = 10 ) const int b = 2;
layout( constant_id = 11 ) const float c = 3.14;
```

In the C/C++ program:

```c
struct abc { int a, int b, float c; } abc;

VkSpecializationMapEntry vsme[3];
    vsme[0].constantID = 9;
    vsme[0].offset = offsetof( abc, a );
    vsme[0].size = sizeof(abc.a);
    vsme[1].constantID = 10;
    vsme[1].offset = offsetof( abc, b );
    vsme[1].size = sizeof(abc.b);
    vsme[2].constantID = 11;
    vsme[2].offset = offsetof( abc, c );
    vsme[2].size = sizeof(abc.c);

VkSpecializationInfo vsi;
    vsi.mapEntryCount = 3;
    vsi.pMapEntries = &vsme[0];
    vsi.dataSize = sizeof(abc); // size of all the Specialization Constants together
    vsi.pData = &abc; // array of all the Specialization Constants
```

It’s important to use sizeof( ) and offsetof( ) instead of hardcoding numbers!
Specialization Constants – Setting the Number of Work-items Per Work-Group in the Compute Shader

In the compute shader

```cpp
layout( local_size_x_id=12 ) in;

layout( local_size_x = 32, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:

```cpp
int numXworkItems = 64;

VkSpecializationMapEntry vsme[1];
    vsme[0].constantID = 12;
    vsme[0].offset = 0;
    vsme[0].size = sizeof(int);

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

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Remember the Overall Block Diagram?

Application

Instance

Instance

Physical Device

Physical Device

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Queue

Queue

Queue

Queue

Queue

Queue

Queue

Command Buffer

Command Buffer

Command Buffer
Where Synchronization Fits in the Overall Block Diagram
Semaphores

- Used to synchronize work executing on difference queues within the same logical device
- You create them, and give them to a Vulkan function which sets them. Later on, you tell a Vulkan function to wait on this particular semaphore
- You don’t end up setting, resetting, or checking the semaphore yourself
- Semaphores must be initialized (“created”) before they can be used

Diagram:

- **Ask for Something**
- **Semaphore**
- **Try to Use that Something**
- **Your program continues**
Creating a Semaphore

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

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

This doesn’t actually do anything with the semaphore – it just sets it up
Semaphores Example during the Render Loop

```c
VkSemaphore imageReadySemaphore;

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

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

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

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

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

Set the semaphore

Wait on the semaphore

You do this to wait for an image to be ready to be rendered into
Fences

- Used when the host needs to wait for the device to complete something big
- Used to synchronize the application with commands submitted to a queue
- Announces that queue-submitted work is finished
- Much finer control than semaphores
- You can un-signal, signal, test or block-while-waiting
Fences

#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

VkFenceCreateInfo vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT;  // = 0
    // VK_FENCE_CREATE_SIGNALED_BIT is only other option

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

, , ,

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

// blocks the host from executing:
result = vkWaitForFences( LogicalDevice, 1, IN \&fence, waitForAll, timeout );
    // waitForAll = VK_TRUE: wait for all fences in the list
    // waitForAll = VK_FALSE: wait for any one fence in the list
    // timeout is a uint64_t timeout in nanoseconds (could be 0, which means to return immediately)
    // timeout can be up to UINT64_MAX = 0xffffffffffffffff (= 580+ years)
    // result = VK_SUCCESS means it returned because a fence (or all fences) signaled
    // result = VK_TIMEOUT means it returned because the timeout was exceeded
Fence Example

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue );

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

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

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

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```
Events

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

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

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

result = vkSetEvent( LogicalDevice, IN event );
result = vkResetEvent( LogicalDevice, IN event );
result = vkGetEventStatus( LogicalDevice, IN event );
// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled

Note: the host cannot \textit{block} waiting for an event, but it can test for it
Controlling Events from the Device

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

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

Could be an array of events

Where signaled, where wait for the signal

Memory barriers get executed after events have been signaled
Pipeline Barriers

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
From the Command Buffer Notes:
These are the Commands that can be entered into the Command Buffer, I

```c
vkCmdBeginQuery( commandBuffer, flags );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexType );
vkCmdBindPipeline( commandBuffer, pipeline );
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
vkCmdBlitImage( commandBuffer, filter );
vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );
vkCmdClearColorImage( commandBuffer, pRanges );
vkCmdClearDepthStencilImage( commandBuffer, pRanges );
vkCmdCopyBuffer( commandBuffer, pRegions );
vkCmdCopyBufferToImage( commandBuffer, pRanges );
vkCmdCopyImage( commandBuffer, pRegions );
vkCmdCopyImageToBuffer( commandBuffer, pRegions );
vkCmdCopyQueryPoolResults( commandBuffer, flags );
vkCmdDebugMarkerBeginEXT( commandBuffer, pMarkerInfo );
vkCmdDebugMarkerEndEXT( commandBuffer );
vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );
vvkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );
vkCmdDispatchIndirect( commandBuffer, offset );
vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );
vkCmdDrawIndexedIndirect( commandBuffer, stride );
vkCmdDrawIndexedIndirectCountAMD( commandBuffer, stride );
vkCmdDrawIndirect( commandBuffer, stride );
vkCmdDrawIndirectCountAMD( commandBuffer, stride );
vkCmdEndQuery( commandBuffer, query );
vkCmdEndRenderPass( commandBuffer );
vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );
```
From the Command Buffer Notes:

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

```c
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```
Potential Memory Race Conditions that Pipeline Barriers can Prevent

1. Write-then-Read (WtR) – the memory write in one operation starts overwriting the memory that another operation’s read needs to use

2. Read-then-Write (RtW) – the memory read in one operation hasn’t yet finished before another operation starts overwriting that memory

3. Write-then-Write (WtW) – two operations start overwriting the same memory and the end result is non-deterministic

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

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

### `vkCmdPipelineBarrier()` Function Call

```c
vkCmdPipelineBarrier( commandBuffer,
                      srcStageMask,  // Guarantee that *this* pipeline stage is completely done being used before …
                      dstStageMask,  // … allowing *this* pipeline stage to be used
                      VK_DEPENDENCY_BY_REGION_BIT,
                      memoryBarrierCount, pMemoryBarriers,
                      bufferMemoryBarrierCount, pBufferMemoryBarriers,
                      imageMemoryBarrierCount, pImageMemoryBarriers
);  // Defines what data we will be blocking on or un-blocking on
```
The Scenario

src cars

TOP_OF_PIPE Street

VERTEX_INPUT Street

VERTEX_SHADER Street

BOTTOM_OF_PIPE Street

dst cars

COLOR_ATTACHMENT_OUTPUT Street

FRAGMENT_SHADER Street

TRANSFER_BIT Street

SIGGRAPH THINK BEYOND
The Scenario

1. The cross-streets are named after pipeline stages

2. All traffic lights start out green

3. There are special sensors at all intersections that will know when *any car in the src group* is in that intersection

4. There are connections from those sensors to the traffic lights so that when *any car in the src group* is in the intersection, the proper *dst* traffic light will be turned red

5. When the *last car in the src group* completely makes it through its intersection, the proper *dst* traffic light is turned back to green

6. The Vulkan command pipeline ordering is this: (1) the *src* cars get released, (2) the pipeline barrier is invoked (which turns some light red), (3) the *dst* cars stop at the red light, (4) the *src* intersection clears, (5) all lights are now green, (6) the *dst* cars continue.
Pipeline Stage Masks –
Where in the Pipeline is this Memory Data being Generated or Consumed?

<table>
<thead>
<tr>
<th>Stage Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
</tr>
</tbody>
</table>
Pipeline Stages

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
Access Masks –
What are you Interested in Generating or Consuming this Memory for?

<table>
<thead>
<tr>
<th>Access Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
</tr>
</tbody>
</table>
### Pipeline Stages and what Access Operations are Allowed

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>VK.ACCESS_INDIRECT_COMMAND_READ_BIT</th>
<th>VK.ACCESS_INDEX_READ_BIT</th>
<th>VK.ACCESS_VERTEX_ATTRIBUTE_READ_BIT</th>
<th>VK.ACCESS_UNIFORM_READ_BIT</th>
<th>VK.ACCESS_INPUT_ATTACHMENT_READ_BIT</th>
<th>VK.ACCESS_SHADER_READ_BIT</th>
<th>VK.ACCESS_SHADER_WRITE_BIT</th>
<th>VK.ACCESS_COLOR_ATTACHMENT_READ_BIT</th>
<th>VK.ACCESS_COLOR_ATTACHMENT_WRITE_BIT</th>
<th>VK.ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</th>
<th>VK.ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</th>
<th>VK.ACCESS_TRANSFER_READ_BIT</th>
<th>VK.ACCESS_TRANSFER_WRITE_BIT</th>
<th>VK.ACCESS_HOST_READ_BIT</th>
<th>VK.ACCESS_HOST_WRITE_BIT</th>
<th>VK.ACCESS_MEMORY_READ_BIT</th>
<th>VK.ACCESS_MEMORY_WRITE_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelineStageTopOfPipeBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageDrawIndirectBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageVertexInputBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageVertexShaderBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageTessellationControlShaderBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageTessellationEvaluationShaderBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageGeometryShaderBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageEarlyFragmentTestsBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageFragmentShaderBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageLateFragmentTestsBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageColorAttachmentOutputBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageBottomOfPipeBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageComputeShader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageTransferBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelineStageHostBit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Access Operations and what Pipeline Stages they can be used In

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Be sure we are done writing an output image before using it for something else

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

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

Stages:
- `src`
- `dst`

Access types:
- `src`
- `dst` (no access setting needed)
The Scenario

src cars are generating the image

dst cars are doing something with that image
Example: Don’t read a buffer back to the host until a shader is done writing it.

<table>
<thead>
<tr>
<th>Stages</th>
<th>Access types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>VK_ACCESS_INDEX_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>VK_ACCESS_SHADER_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>VK_ACCESS_HOST_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
</tr>
</tbody>
</table>

(src) (no access setting needed)
The Scenario

src cars

TOP_OF_PIPE Street

VERTEX_INPUT Street

dst cars

VERTEX_SHADER Street

FRAGMENT_SHADER Street

BOTTOM_OF_PIPE Street

TRANSFER_BIT Street

COLOR_ATTACHMENT_OUTPUT Street

src cars

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

<table>
<thead>
<tr>
<th>VkImageMemoryBarrier</th>
<th>vimb’</th>
</tr>
</thead>
<tbody>
<tr>
<td>vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;</td>
<td></td>
</tr>
<tr>
<td>vimb.pNext = nullptr;</td>
<td></td>
</tr>
<tr>
<td>vimb.srcAccessMask = ??;</td>
<td></td>
</tr>
<tr>
<td>vimb.dstAccessMask = ??;</td>
<td></td>
</tr>
<tr>
<td><strong>vimb.oldLayout = ??;</strong></td>
<td></td>
</tr>
<tr>
<td>vimb.newLayout = ??;</td>
<td></td>
</tr>
<tr>
<td>vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;</td>
<td></td>
</tr>
<tr>
<td>vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;</td>
<td></td>
</tr>
<tr>
<td>vimb.image = ??;</td>
<td></td>
</tr>
<tr>
<td>vimb.subresourceRange = visr;</td>
<td></td>
</tr>
</tbody>
</table>

**VK_IMAGE_LAYOUT_UNDEFINED**
**VK_IMAGE_LAYOUT_GENERAL**
**VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL**
**VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL**
**VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL**
**VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL**
**VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL**
**VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**
**VK_IMAGE_LAYOUT_PREINITIALIZED**
**VK_IMAGE_LAYOUT_PRESENT_SRC_KHR**
**VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

Here, the use of vkCmdPipelineBarrier( ) is to simply change the layout of an image.
Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

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

The Display We Want

Too often, the Display We Get
“Aliasing” is a signal-processing term for “under-sampled compared with the frequencies in the signal”.

What the signal really is: what we want

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

Sampling Interval

Sampled Points
Aliasing
The Nyquist Criterion

“The Nyquist [sampling] rate is twice the maximum component frequency of the function [i.e., signal] being sampled.” -- WikiPedia
Oversampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there.

There are two approaches to this:

1. **Supersampling**: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. Render the image at each of these sub-pixels.

2. **Multisampling**: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. If any of them pass, then perform a single color render for the one pixel and assign that single color to all the sub-pixels that passed the depth and stencil tests.

The final step will be to average those sub-pixels’ colors to produce one final color for this whole pixel. This is called **resolving** the pixel.
Vulkan Specification Distribution of Sampling Points within a Pixel
Vulkan Specification Distribution of Sampling Points within a Pixel

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.375, 0.125)</td>
<td>(0.5625, 0.3125)</td>
<td>(0.5625, 0.5625)</td>
<td></td>
</tr>
<tr>
<td>(0.4375, 0.6875)</td>
<td>(0.4375, 0.3125)</td>
<td>(0.3125, 0.625)</td>
<td></td>
</tr>
<tr>
<td>(0.25, 0.25)</td>
<td>(0.8125, 0.5625)</td>
<td>(0.1875, 0.375)</td>
<td></td>
</tr>
<tr>
<td>(0.875, 0.375)</td>
<td>(0.3125, 0.1875)</td>
<td>(0.8125, 0.6875)</td>
<td></td>
</tr>
<tr>
<td>(0.125, 0.625)</td>
<td>(0.1875, 0.8125)</td>
<td>(0.6875, 0.1875)</td>
<td></td>
</tr>
<tr>
<td>(0.75, 0.75)</td>
<td>(0.0625, 0.4375)</td>
<td>(0.5, 0.0625)</td>
<td></td>
</tr>
<tr>
<td>(0.625, 0.875)</td>
<td>(0.6875, 0.9375)</td>
<td>(0.125, 0.75)</td>
<td></td>
</tr>
<tr>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.25)</td>
<td>(0.0625, 0.0)</td>
<td></td>
</tr>
</tbody>
</table>
Consider Two Triangles Who Pass Through the Same Pixel

Let’s assume (for now) that the two triangles don’t overlap – that is, they look this way because they butt up against each other.
Supersampling

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

# Fragment Shader calls = 8
Multisampling

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

# Fragment Shader calls = 2
Consider Two Triangles Who Pass Through the Same Pixel

Let’s assume (for now) that the two triangles don’t overlap – that is, they look this way because they butt up against each other.

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Consider Two Triangles Who Pass Through the Same Pixel

Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

A: The ideas are all still the same, but the blue one had to deal with 8 sub-pixels (instead of 5 like before). But, the red triangle came along and obsoleted 3 of those blue sub-pixels. Note that the “resolved” image will still turn out the same as before.
Consider Two Triangles Who Pass Through the Same Pixel

What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>shader calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red fragment</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>shader calls</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Setting up the Image

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

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

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

**How dense is the sampling**

**VK_TRUE means to allow some sort of multisampling to take place**
VkPipelineMultisampleStateCreateInfo  

```
vpmcsi;
```

```
    vpmcsi.minSampleShading = 0.5;
```

```
    ...
```

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

- 0. produces simple multisampling
- (0., 1.) produces partial supersampling
- 1. Produces complete supersampling
-setting up the image

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

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

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

VkAttachmentReference depthReference;
  depthReference.attachment = 1;
  depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```
Setting up the Image

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

VkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.attachmentCount = 2; // color and depth/stencil
vrpci.pAttachments = vad;
vrpci.subpassCount = 1;
vrpci.pSubpasses = IN &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkSubpassDependency *)nullptr;

result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
```
Resolving the Image:
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

VIOffset3D
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;

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

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

VkImageResolve
vir.srcSubresource = visl;
vir.srcOffset = vo3;
vir.dstSubresource = visl;
vir.dstOffset = vo3;
vir.extent = ve3;

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

For the *ImageLayout, use VK_IMAGE_LAYOUT_GENERAL
Multipass Rendering

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Multipass Rendering uses Attachments -- What is a Vulkan Attachment Anyway?

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

-- Vulkan Programming Guide

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

For example:
What is an Example of Wanting to do This?

There is a process in computer graphics called **Deferred Rendering**. The idea is that a game-quality fragment shader takes a long time (relatively) to execute, but, with all the 3D scene detail, a lot of the rendered fragments are going to get z-buffered away anyhow. So, why did we invoke the fragment shaders so many times when we didn’t need to?

Here’s the trick:

Let’s create a grossly simple fragment shader that writes out (into multiple framebuffers) each fragment’s:
- position (x,y,z)
- normal (nx,ny,nz)
- material color (r,g,b)
- texture coordinates (s,t)

As well as:
- the current light source positions and colors
- the current eye position

When we write these out, the final framebuffers will contain just information for the pixels that *can be seen*. We then make a second pass running the expensive lighting model *just* for those pixels. This known as the **G-buffer Algorithm**.
Back in Our Single-pass Days

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

Here comes a quick reminder of how we did that.

Afterwards, we will extend it.
Back in Our Single-pass Days, I

```c
VkAttachmentDescription vad[2];
vad[0].flags = 0;
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

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

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

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

 VkRenderPassCreateInfo
    vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
    vrpci.pNext = nullptr;
    vrpci.flags = 0;
    vrpci.attachmentCount = 2;  // color and depth/stencil
    vrpci.pAttachments = vad;
    vrpci.subpassCount = 1;
    vrpci.pSubpasses = &vsd;
    vrpci.dependencyCount = 0;
    vrpci.pDependencies = (VkSubpassDependency *)nullptr;

result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
Multipass Rendering

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

In this case, we will look at following up a 3D rendering with Gbuffer operations.
VkAttachmentDescription

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

vad[1].flags = 0;
vad[1].format = VK_FORMAT_R32G32B32A32_UINT;
vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_UNDEFINED;

vad[2].flags = 0;
vad[2].format = VK_FORMAT_R8G8B8A8_SRGB;
vad[2].samples = VK_SAMPLE_COUNT_1_BIT;
vad[2].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[2].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC;
VkAttachmentReference depthOutput;
depthOutput.attachment = 0; // depth
depthOutput.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference gbufferInput;
gBufferInput.attachment = 0; // depth
gBufferInput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference gbufferOutput;
gBufferOutput.attachment = 1; // gbuffer
gBufferOutput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference lightingInput[2];
lightingInput[0].attachment = 0; // depth
lightingInput[0].layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL;
lightingInput[1].attachment = 1; // gbuffer
lightingInput[1].layout = VK_IMAGE_LAYOUT_SHADER_READ_OPTIMAL;

VkAttachmentReference lightingOutput;
lightingOutput.attachment = 2; // color rendering
lightingOutput = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
VkSubpassDescription vsd[3];
    vsd[0].flags = 0;
    vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
    vsd[0].inputAttachmentCount = 0;
    vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
    vsd[0].colorAttachmentCount = 0;
    vsd[0].pColorAttachments = (VkAttachmentReference *)nullptr;
    vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
    vsd[0].pDepthStencilAttachment = &depthOutput;
    vsd[0].preserveAttachmentCount = 0;
    vsd[0].pPreserveAttachments = (uint32_t *) nullptr;
    vsd[0].preserveAttachmentCount = 0;
    vsd[0].pPreserveAttachments = (uint32_t *) nullptr;

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

    vsd[2].flags = 0;
    vsd[2].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
    vsd[2].inputAttachmentCount = 2;
    vsd[2].pInputAttachments = &lightingInput[0];
    vsd[2].colorAttachmentCount = 1;
    vsd[2].pColorAttachments = &lightingOutput;
    vsd[2].pResolveAttachments = (VkAttachmentReference *)nullptr;
    vsd[2].pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
    vsd[2].preserveAttachmentCount = 0;
    vsd[2].pPreserveAttachments = (uint32_t *) nullptr;
VkSubpassDependency vsdp[2];
vsdp[0].srcSubpass = 0; // depth rendering  
vsdp[0].dstSubpass = 1; //  
gbuffer
vsdp[0].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[0].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vsdp[0].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[0].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

vsdp[1].srcSubpass = 1; // gbuffer  
vsdp[1].dstSubpass = 2; //  
color output
vsdp[1].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[1].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vsdp[1].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[1].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

Notice how similar this is to creating a Directed Acyclic Graph (DAG).
VkRenderPassCreateInfo
  vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  vrpci.pNext = nullptr;
  vrpci.flags = 0;
  vrpci.attachmentCount = 3;  // depth, gbuffer, output
  vrpci.pAttachments = vad;
  vrpci.subpassCount = 3;
  vrpci.pSubpasses = vsd;
  vrpci.dependencyCount = 2;
  vrpci.pDependencies = vsdp;

result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );

// subpass #0 is automatically started here

vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
    GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *) nullptr );
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdNextSubpass( CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE );
// subpass #1 is started here

vkCmdNextSubpass( CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE );
// subpass #2 is started here

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
Analog Ray Tracing Example
Digital Ray Tracing Examples
The Rasterization Shader Pipeline

- **Vertex Shader**
  - Primitive Assembly
  - **Tessellation Control Shader**
  - **Tessellation Primitive Generator**
  - **Tessellation Evaluation Shader**
  - Primitive Assembly
  - **Geometry Shader**
  - Primitive Assembly
  - **Rasterizer**
  - **Fragment Shader**

- **Fixed Function**
- **Programmable**
The Ray-trace Pipeline Involves Five New Shader Types

- A Ray Generation Shader runs on a 2D grid of threads. It begins the entire ray-tracing operation.
- An Intersection Shader implements ray-primitive intersections.
- An Any Hit Shader is called when the Intersection Shader finds a hit.
- The Closest Hit Shader is called with the information about the hit that happened closest to the viewer. Typically lighting is done here, or firing off new rays to handle reflection and refraction.
- A Miss Shader is called when no intersections are found for a given ray. Typically it just sets its pixel color to the background color.

Note: none of this lives in the graphics hardware pipeline. This is all built on top of the compute functionality.
The Ray Intersection Process for a Sphere

1. Sphere equation: \((x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2\)

2. Ray equation: \((x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz)\)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying-through and simplifying gives:

\[At^2 + Bt + C = 0\]

Solve for \(t_1, t_2\)

If both \(t_1\) and \(t_2\) are complex, then the ray missed the sphere.
If \(t_1 == t_2\), then the ray brushed the sphere at a tangent point.
If both \(t_1\) and \(t_2\) are real and different, then the ray entered and exited the sphere.

In Vulkan terms:

\[
gl_{\text{WorldRayOriginKHR}} = (x_0,y_0,z_0)
gl_{\text{HitKHR}} = t\ngl_{\text{WorldRayDirectionKHR}} = (dx,dy,dz)
\]
The Ray Intersection Process for a Cube

1. Plane equation: \( Ax + By + Cz + D = 0 \)

2. Ray equation: \( (x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz) \)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying-through and simplifying gives:

\[ At + B = 0 \]
Solve for \(t\)

A cube is actually the intersection of 6 half-space planes (just 4 are shown here). Each of these will produce its own \(t\) intersection value. Treat them as pairs: \((t_{x1}, t_{x2})\), \((t_{y1}, t_{y2})\), \((t_{z1}, t_{z2})\)

The ultimate entry and exit values are:

\[
\begin{align*}
    t_{\text{min}} &= \max(\ \min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2}) ) \\
    t_{\text{max}} &= \min(\ \max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2}) )
\end{align*}
\]
In a Raytracing, each ray typically hits a lot of Things
Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers

- The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.

- Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.

- The BLAS is used as a Model Coordinate bounding box.

- The TLAS is used as a World Coordinate bounding box.

- A TLAS can instance multiple BLAS’s.
Creating *Bottom* Level Acceleration Structures

```c
vkCreateAccelerationStructureKHR BottomLevelAccelerationStructure;

VkAccelerationStructureInfoKHR vasi;
vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
vasi.flags = 0;
vasi.pNext = nullptr;
vasi.instanceCount = 0;
vasi.geometryCount = << number of vertex buffers >>
vasi.pGeometries = << vertex buffer pointers >>

VkAccelerationStructureCreateInfoKHR vasci;
vasci.sType = VK_STRUCTURE_TYPE ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
vasci.pNext = nullptr;
vasci.info = &vasi;
vasci.compactedSize = 0;

result = vkCreateAccelerationStructureKHR( LogicalDevice, IN &vasci, PALLOCATOR, OUT &BottomLevelAccelerationStructure );
```

![Diagram of Bottom Level Acceleration Structures](image)
Creating Top Level Acceleration Structures

```c
vkCreateAccelerationStructureKHR TopLevelAccelerationStructure;

VkAccelerationStructureInfoKHR vasi;
vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
vasi.flags = 0;
vasi.pNext = nullptr;
vasi.instanceCount = << number of bottom level acceleration structure instances >>;
vasi.geometryCount = 0;
vasi.pGeometries = VK_NULL_HANDLE;

VkAccelerationStructureCreateInfoKHR vasci;
vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
vasci.pNext = nullptr;
vasci.info = &vasi;
vasci.compactedSize = 0;

result = vkCreateAccelerationStructureKHR( LogicalDevice, &vasci, PALLOCATOR, &TopLevelAccelerationStructure );
```

![Diagram of Top Level Acceleration Structure with three bottom level acceleration structures](image)
Ray Generation Shader

A “payload” is information that keeps getting passed through the process. Different stages can add to it. It is finally consumed at the very end, in this case by writing \texttt{color} into the pixel being worked on.
A New Built-in Function

```c
void traceKHR(
    accelerationStructureKHR topLevel,
    uint rayFlags,
    uint cullMask,
    uint sbtRecordOffset,
    uint sbtRecordStride,
    uint missIndex,
    vec3 origin,
    float tmin,
    vec3 direction,
    float tmax,
    int payload
);
```

In Vulkan terms:

- `gl_WorldRayOriginKHR = (x₀, y₀, z₀)`
- `gl_HitKHR = t`
- `gl_WorldRayDirectionKHR = (dx, dy, dz)`
Intersection Shader

```cpp
hitAttributeKHR vec3 attribs

void main()
{
    SpherePrimitive sph = spheres[gl_PrimitiveID];
    vec3 orig = gl_WorldRayOriginKHR;
    vec3 dir = normalize(gl_WorldRayDirectionKHR);

    float discr = b*b - 4.*a*c;
    if( discr < 0. )
        return;

    float tmp = ( -b - sqrt(discr) ) / (2.*a);
    if( gl_RayTminKHR < tmp && tmp < gl_RayTmaxKHR )
    {
        vec3 p = orig + tmp * dir;
        attribs = p;
        reportIntersectionKHR( tmp, 0 );
        return;
    }

    tmp = ( -b + sqrt(discr) ) / (2.*a);
    if( gl_RayTminKHR < tmp && tmp < gl_RayTmaxKHR )
    {
        vec3 p = orig + tmp * dir;
        attribs = p;
        reportIntersectionKHR( tmp, 0 );
        return;
    }
}
```

Intersect a ray with an arbitrary 3D object. Passes data to the Any Hit shader. There is a built-in ray-triangle Intersection Shader.
Miss Shader

Handle a ray that doesn’t hit *any* objects

```cpp
rayPayloadKHR myPayload
{
    vec4 color;
};

void main( )
{
    color = vec4( 0., 0., 0., 1. );
}
```

![Diagram of shader handling](image)
Any Hit Shader

Handle a ray that hits *anything*. Store information on each hit. Can reject a hit.

```glsl
layout( binding = 4, set = 0) buffer outputProperties
{
    float outputValues[ ];
} outputData;

layout(location = 0) rayPayloadInKHR uint outputId;
layout(location = 1) rayPayloadInKHR uint hitCounter;
hitAttributeKHR vec 3 attribs;

void main()
{
    outputData.outputValues[ outputId + hitCounter ] = gl_PrimitiveID;
    hitCounter = hitCounter + 1;
}
```
Closest Hit Shader

Handle the intersection closest to the viewer. Collects data from the Any Hit shader. Can spawn more rays.

```glsl
rayPayloadKHR myPayLoad
{
    vec4 color;
};

void main()
{
    vec3 stp = gl_WorldRayOriginKHR + gl_HitKHR * gl_WorldRayDirectionKHR;
    color = texture( MaterialUnit, stp ); // material properties lookup
}
```

In Vulkan terms:
- `gl_WorldRayOriginKHR = (x₀, y₀, z₀)`
- `gl_HitKHR = t`
- `gl_WorldRayDirectionKHR = (dx, dy, dz)`
Other New Built-in Functions

void `terminateRayKHR`();

void `ignoreIntersectionKHR`();

Loosely equivalent to “discard”

void `reportIntersectionKHR`( float hit, uint hitKind );
Ray Trace Pipeline Data Structure

```c
// Ray Trace Pipeline Data Structure
VkPipeline      RaytracePipeline;
VkPipelineLayout PipelineLayout;

// VkPipelineLayoutCreateInfo
VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 1;
vplci.pSetLayouts = &descriptorSetLayout;
vplci.pushConstantRangeCount = 0;
vplci.pPushConstantRanges = nullptr;

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, nullptr, OUT &PipelineLayout);

// VkRayTracingPipelineCreateInfoKHR
VkRayTracingPipelineCreateInfoKHR vrtpci;
vrtpci.sType = VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
vrtpci.pNext = nullptr;
vrtpci.flags = 0;
vrtpci.stageCount = << # of shader stages in the ray-trace pipeline >>
vrtpci.pStages = << what those shader stages are >>
vrtpci.groupCount = << # of shader groups >>
vrtpci.pGroups = << pointer to the groups (a group is a combination of shader programs >>
vrtpci.maxRecursionDepth = << how many recursion layers deep the ray tracing is allowed to go >>;
vrtpci.layout = PipelineLayout;
vrtpci.basePipelineHandle = VK_NULL_HANDLE;
vrtpci.basePipelineIndex = 0;

result = vkCreateRayTracingPipelinesKHR( LogicalDevice, PALLOCATOR, 1, IN &rvrtpci, nullptr, OUT &RaytracePipeline);
```
The Trigger comes from the Command Buffer: 
vlCmdBindPipeline( ) and vkCmdTraceRaysKHR( )

```
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_RAYTRACING_KHR, RaytracePipeline );

vkCmdTraceRaysKHR( CommandBuffer, raygenShaderBindingTableBuffer, raygenShaderBindingOffset, missShaderBindingTableBuffer, missShaderBindingOffset, missShaderBindingStride, hitShaderBindingTableBuffer, hitShaderBindingOffset, hitShaderBindingStride, callableShaderBindingTableBuffer, callableShaderBindingOffset, callableShaderBindingStride, width, height, depth );,
```
https://www.youtube.com/watch?v=QL7sXc2iNJ8
Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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