An Introduction to the Vulkan Computer Graphics API

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

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

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! The GeForce 1080ti cards are what made this course possible.

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

Ali Alsalehy
Natasha Anisimova
Jianchang Bi
Christopher Cooper
Richard Cunard
Braxton Cuneo
Benjamin Fields
Trevor Hammock
Zach Lerew
Victor Li

Alan Neads
Raja Petroff
Bei Rong
Lance Roy
Lily Shellhammer
Hannah Solorzano
Jian Tang
Glenn Upthagrove
Logan Wingard

What Prompted the Move to Vulkan?

1. Performance
2. Performance
3. Performance

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

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

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

**NVIDIA Titan V Specifications 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 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>GV100</td>
<td>GV100</td>
<td>GV100</td>
<td>GV100</td>
<td>GV100</td>
</tr>
<tr>
<td><strong>Transformer Count</strong></td>
<td>211.1T</td>
<td>211.1T</td>
<td>192.2T</td>
<td>192.2T</td>
<td>192.2T</td>
</tr>
<tr>
<td><strong>Core Clock</strong></td>
<td>1330MHz</td>
<td>1330MHz</td>
<td>1330MHz</td>
<td>1330MHz</td>
<td>1330MHz</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>167GB/s</td>
<td>167GB/s</td>
<td>167GB/s</td>
<td>167GB/s</td>
<td>167GB/s</td>
</tr>
<tr>
<td><strong>Memory Capacity</strong></td>
<td>12GB</td>
<td>12GB</td>
<td>12GB</td>
<td>12GB</td>
<td>12GB</td>
</tr>
<tr>
<td><strong>Memory Type</strong></td>
<td>HBM2</td>
<td>HBM2</td>
<td>HBM2</td>
<td>HBM2</td>
<td>HBM2</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>26GB/s</td>
<td>26GB/s</td>
<td>26GB/s</td>
<td>26GB/s</td>
<td>26GB/s</td>
</tr>
</tbody>
</table>

The NVIDIA Titan V is used to test targets at games, but rather at scientific and research/learning applications. That does not, however, mean that the cards are capable of gaming, nor does it mean that we can extrapolate those performance metrics to mobile GPUs. The Titan V is a significant product of the high-end NVIDIA GPU, part of the Tesla architecture. The key differentiator is that its Faraday chips, which run at 250W, whereas the Tesla V100 is available as part of $30,000 developer kit. The Tesla V100 chip offers greater memory capacity, but 1080 Ti's 12GB HBM2 versus 12GB HBM2, and ample storage capacity, is a bit of a jock than what is required in many algorithms. Core count, for instance, is 12GB CUDA cores across both GPUs, with 1440 Tensor cores used for transfer or machine learning environments in each GPU.
Who is the Khronos Group?

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

Vulkan

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

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

Moving part of the driver into the application

Complex drivers lead to driver overhead and 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
Vulkan Highlights: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Vulkan Highlights: a More Typical Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer
Your Sample2017.zip File Contains This

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>12/27/2017 14:05 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Debug</td>
<td>12/27/2017 14:33 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>12/27/2017 14:44 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>glfw.h</td>
<td>12/27/2017 15:04 AM</td>
<td>C/C++ Header</td>
<td>493 KB</td>
</tr>
<tr>
<td>gpl_license</td>
<td>12/27/2017 15:06 AM</td>
<td>Object File Library</td>
<td>240 KB</td>
</tr>
<tr>
<td>VulkanLoader.h</td>
<td>12/27/2017 15:08 AM</td>
<td>C/C++ Header</td>
<td>159 KB</td>
</tr>
<tr>
<td>VulkanLoader.cc</td>
<td>12/27/2017 15:09 AM</td>
<td>C/C++ Header</td>
<td>163 KB</td>
</tr>
<tr>
<td>VulkanLoader/clock</td>
<td>12/27/2017 15:10 AM</td>
<td>C/C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>VulkanLoader/lib</td>
<td>12/27/2017 15:11 AM</td>
<td>C/C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>VulkanLoader/texture</td>
<td>12/27/2017 15:12 AM</td>
<td>C/C++ Header</td>
<td>1 KB</td>
</tr>
<tr>
<td>image.bmp</td>
<td>12/27/2017 15:13 AM</td>
<td>BMP File</td>
<td>1 KB</td>
</tr>
<tr>
<td>image.jpg</td>
<td>12/27/2017 15:14 AM</td>
<td>JPG File</td>
<td>1 KB</td>
</tr>
<tr>
<td>sample.cpp</td>
<td>12/27/2017 15:15 AM</td>
<td>C/C++ Source</td>
<td>10 KB</td>
</tr>
<tr>
<td>SampleProject</td>
<td>12/27/2017 15:16 AM</td>
<td>Visual Studio 7.0</td>
<td>2 KB</td>
</tr>
<tr>
<td>SampleProject/Files</td>
<td>12/27/2017 15:17 AM</td>
<td>Visual Studio 7.0</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple_flag.h</td>
<td>12/27/2017 15:18 AM</td>
<td>Plain Text</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple_fg.cpp</td>
<td>12/27/2017 15:19 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple tease.cpp</td>
<td>12/27/2017 15:20 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple tease.h</td>
<td>12/27/2017 15:21 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple tease.s</td>
<td>12/27/2017 15:22 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>Sample/coating</td>
<td>12/27/2017 15:23 AM</td>
<td>Project File</td>
<td>7 KB</td>
</tr>
<tr>
<td>Sample/coating/Files</td>
<td>12/27/2017 15:24 AM</td>
<td>Project File</td>
<td>1 KB</td>
</tr>
<tr>
<td>sample_ng.cpp</td>
<td>12/27/2017 15:25 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>sample_set.cpp</td>
<td>12/27/2017 15:26 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>sample_set.h</td>
<td>12/27/2017 15:27 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple_set.s</td>
<td>12/27/2017 15:28 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>simple_set.s</td>
<td>12/27/2017 15:29 AM</td>
<td>SPV File</td>
<td>1 KB</td>
</tr>
<tr>
<td>SimpleSaveData.cpp</td>
<td>12/27/2017 15:30 AM</td>
<td>C/C++ Source</td>
<td>4 KB</td>
</tr>
<tr>
<td>vk_surfaces.h</td>
<td>12/27/2017 15:31 AM</td>
<td>C/C++ Header</td>
<td>2 KB</td>
</tr>
<tr>
<td>vk_surfaces.h</td>
<td>12/27/2017 15:32 AM</td>
<td>C/C++ Header</td>
<td>2 KB</td>
</tr>
<tr>
<td>vk_platform.h</td>
<td>12/27/2017 15:33 AM</td>
<td>C/C++ Header</td>
<td>2 KB</td>
</tr>
<tr>
<td>vk_platform.c</td>
<td>12/27/2017 15:34 AM</td>
<td>C/C++ Source</td>
<td>4 KB</td>
</tr>
<tr>
<td>VulkanDebug.bit</td>
<td>12/27/2017 15:35 AM</td>
<td>Text Document</td>
<td>2 KB</td>
</tr>
</tbody>
</table>

Drawing

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

typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;
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 }
};
```

Triangles Represented as an Array of Structures

```
struct vertex
{
    glm::vec3       position;
    glm::vec3       normal;
    glm::vec3       color;
    glm::vec2       texCoord;
};
struct vertex VertexData[  ] =
{
    // triangle 0-2-3:
    // vertex #0:
    {  // vertex #0:
        { -1., -1., -1. },
        {  0.,  0., -1. },
        { 0.,  0.,  0. },
        { 1., 0. },
    },
    // vertex #2:
    {  // vertex #2:
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        { 0.,  1.,  0. },
        { 1., 1. },
    },
    // vertex #3:
    {  // vertex #3:
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        { 1.,  1.,  0. },
        { 0., 1. },
    },

    // triangle 0-1-5:
    // vertex #0:
    {  // vertex #0:
        { -1., -1., -1. },
        {  0.,  0., -1. },
        { 0.,  0.,  1. },
        { 1., 0. },
    },
    // vertex #1:
    {  // vertex #1:
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        { 1.,  1.,  0. },
        { 0., 1. },
    },
    // vertex #5:
    {  // vertex #5:
        {  1., -1., -1. },
        {  0.,  0., -1. },
        { 0.,  0.,  1. },
        { 1., 0. },
    },
};
```

From the file SampleVertexData.cpp:
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[7] =
{
    // 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. } },
};
```

Transmission Order:

- Triangle 0-2-3
- Triangle 0-3-1
- Triangle 4-5-7

**From the file `SampleVertexData.cpp`:**

**Vertex Data:**
- Vertex 7
- Vertex 5
- Vertex 4
- Vertex 3
- Vertex 1
- Vertex 0
- Vertex 3
- Vertex 2
- Vertex 0

**Triangles:**
- Draw

Filling the Vertex Buffer:

```cpp
MyBuffer MyVertexDataBuffer;

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

**VkResult**

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

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

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.

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

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

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

VkPipelineInputStateCreateInfo 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;
```
### Telling the Pipeline about its Input

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

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

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

### 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] );

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

Better to do `sizeof` than to hard-code a number
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,
};

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 );

Remember that integer-indexed buffers are just BLOBs too.
Drawing with an Indexed Buffer

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

```c
Init05MyVertexDataBuffer( sizeof(JustVertexData), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
Init05MyIndexDataBuffer( sizeof(JustIndexData), &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );
```

Note that there is no vertex-count! It is up to you to not exceed the number of vertices with your index numbers!
Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

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

Thus, when using index-ed buffer drawing, you need to create a new vertex struct if any of (position, normal, color, texCoords) changes from what was previously-stored at those coordinates.
Data Buffers

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

A **Data Buffer** is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a “Binary Large Object”, or “BLOB”.)

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

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

```c
typedef VkBuffer VkDataBuffer;
```

**Terminology Issues**
**Vulkan: Buffers**

- `vkCreateBuffer()`
  - `VkBufferCreateInfo`
  - `bufferUsage`
  - `queueFamilyIndices`
  - `size (bytes)`
  - `LogicalDevice`

- `vkGetBufferMemoryRequirements()`
  - `Buffer`
  - `VkMemoryAllocateInfo`
  - `memoryType`

- `vkAllocateMemory()`
  - `LogicalDevice`

- `vkBindBufferMemory()`
  - `bufferMemoryHandle`

- `vkMapMemory()`
  - `gpuAddress`

---

**Vulkan: Creating a Data Buffer**

```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 = << 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 uint32_t *)nullptr;

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

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

```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

<< do the memory copy >>
result = vkUnmapMemory( LogicalDevice, IN vdm );
```

Finding the Right Type of Memory

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

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

Finding the Right Type of Memory

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

- 2 Memory Heaps:
  - Heap 0: size = 0xb7c00000 DeviceLocal
  - Heap 1: size = 0xfac00000
Something I've Found Useful

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

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.

Initializing a Data Buffer

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

VkResult Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer)
{
    
    vbci.size = pMyBuffer->size = size;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    pMyBuffer->vdm = vdm;
}
Here’s the C struct to hold some uniform variables

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

Here’s the shader code to access those uniform variables

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

Filling those Uniform Variables

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

CPU: MyBuffer MyMatrixUniformBuffer;

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

```
MyBuffer MyMatrixUniformBuffer;
```

This C struct is holding the actual data. It is writeable by the application.

```
struct matBuf Matrices;
```

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

```
uniform matBuf Matrices;
```

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

CPU:
```
struct matBuf Matrices;
```

The Descriptor Set for the Buffer

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

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

VkWriteDescriptorSet wds0;
    // ds 0:
    wds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    wds0.pNext = nullptr;
    wds0.dstSet = DescriptorSets[0];
    wds0.dstBinding = 0;
    wds0.dstArrayElement = 0;
    wds0.descriptorCount = 1;
    wds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    wds0.pBufferInfo = &vdbi0;
    wds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;

vkUpdateDescriptorSets( LogicalDevice, 1, IN &wds0, IN 0, (VkCopyDescriptorSet *)nullptr );
```
Filling the Data Buffer

typedef struct myBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    } MyBuffer;

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

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, 0 );  // 0 is the offset
    return result;
}
VkResult
Fill05DataBuffer(IN MyBuffer myBuffer, IN void * data)
{
    // the size of the data had better match the size that was used to init the buffer!
    void * pGpuMemory;
    vkMapMemory(LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory);
    // 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.

---

**Vulkan**

**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.

Vulkan Shader Stages

Shader stages

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

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

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

Vertex and Instance indices:

- gl_VertexIndex
- gl_InstanceIndex

  These are called:
  - gl_VertexID
  - gl_InstanceID

  In OpenGL. The Vulkan names make more sense.

- Both are 0-based

**gl_FragColor:**

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

Shader combinations of separate texture data and samplers:

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

Descriptor Sets:

layout( set=0, binding=0 ) . . .

Push Constants:

layout( push_constant) . . .

Specialization Constants:

- layout( constant_id = 3 ) const int N = 5;
- Can only use basic operators, declarations, and constructors
- Only for scalars, but a vector can be constructed from specialization constants

Specialization Constants for Compute Shaders:

- layout( local_size_x_id = 8, local_size_y_id = 16 );

  - gl_WorkGroupSize.z is still as it was
Vulkan: Shaders’ use of Layouts for Uniform Variables

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

Vulkan Shader Compiling

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

**Advantages:**

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

```
```

**Shader file extensions:**
- `.vert` Vertex
- `.tesc` Tessellation Control
- `.tese` Tessellation Evaluation
- `.geom` Geometry
- `.frag` Fragment
- `.comp` Compute
(Can be overridden by the –S option)

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

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

---

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

1. Click on the Microsoft Start icon
2. Type `word bash`
You Can Run the SPIR-V Compiler on Windows from a Bash Shell

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

Running glslangValidator.exe

```
MINGW64:/y/Vulkan/Sample2017
ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 185
glslangValidator.exe -V sample-vert.vert -o sample-vert.spv sample-vert.vert
ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 186
glslangValidator.exe -V sample-frag.frag -o sample-frag.spv sample-frag.frag
ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 
```
You can also run SPIR-V from a Linux Shell

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

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

- The input file. The compiler determines the shader type by the file extension:
  - .vert  Vertex shader
  - .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
VkResult Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule ) {
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb" );
    if( fp == NULL ) {
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str() );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC ) {
        fprintf( FpDebug, "Magic number for spir-v file '%s is 0x%08x -- should be 0x%08x
", filename.c_str() , magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char *code = new unsigned char [size];
    fread( code, size, 1, fp );
    fclose( fp );
    return VK_SUCCESS;
}
```
Reading a SPIR-V File into a Shader Module

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

VkResult result = vkCreateShaderModule( LogicalDevice, IN &vsmci, PALLOCATOR, pShaderModule );
fprintf( FpDebug, "Shader Module '%s' successfully loaded\n", filename.c_str() );
delete [ ] code;
return result;

Vulkan: Creating a Pipeline

VkShaderStageCreateInfo
  which stage (VERTEX, etc.)

VkShaderModule
  binding, inputRate

VkPipelineShaderStageCreateInfo
  which stage (VERTEX, etc.)

VkPipelineInputAssemblyStateCreateInfo
  Topology

VkPipelineInputStateCreateInfo
  Viewport
  Scissor
  stencilTestEnable
  stencilOpStateFront
  stencilOpStateBack

VkPipelineTessellationStateCreateInfo
  cullMode
  polygonMode
  frontFace
  lineWidth

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

VkPipelineColorBlendAttachmentState
  alphaBlendEnable
  depthWriteEnable
  depthCompareOp

VkPipelineColorBlendStateCreateInfo
  stencilWriteMask
  stencilReadMask
  srcColorBlendFactor
  dstColorBlendFactor
  colorBlendOp
  srcAlphaBlendFactor
  dstAlphaBlendFactor
  alphaBlendOp
  colorWriteMask

VkPipelineDynamicStateCreateInfo
  Array naming the states that can be set dynamically

VkPipelineInputAssemblyStateCreateInfo
  MultiSample State
  Dynamic State
  Pipeline layout
  RenderPass
  basePipelineHandle
  basePipelineIndex

VkPipelineViewportStateCreateInfo
  x, y, w, h,
  minDepth,
  maxDepth
  offset
  extent

VkPipelineDepthStencilStateCreateInfo
  depthTestEnable
  depthWriteEnable
  depthCompareOp
  stencilTestEnable
  stencilOpStateFront
  stencilOpStateBack

VkPipelineColorBlendAttachmentState
  blendEnable
  srcColorBlendFactor
  dstColorBlendFactor
  colorBlendOp
  srcAlphaBlendFactor
  dstAlphaBlendFactor
  alphaBlendOp
  colorWriteMask

VkPipelineDynamicStateCreateInfo
  Array naming the states that can be set dynamically
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

1. Capability Shader
   - 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 32  "gl_PerVertex"
   Name 32(gl_PerVertex) 0  "gl_Position"
   Name 32(gl_PerVertex) 1  "gl_PointSize"
   Name 32(gl_PerVertex) 2  "gl_ClipDistance"

   Name 37  "aVertex"
   Name 48  "vNormal"
   Name 53  "aNormal"
   Name 56  "vColor"
   Name 57  "aColor"
   Name 61  "fTexCoord"
   Name 63  "aTexCoord"
   Name 66  "lightBuf"
   Name 65(lightBuf) 0  "uLightPos"

   Decorate 13(matBuf) Block
   Decorate 15(Matrices) DescriptorSet 0
   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
   MemberDecorate 32(gl_PerVertex) 3 BuiltIn Color

   Decorate 37(aVertex) Location 0
   Decorate 48(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
   MemberDecorate 65(lightBuf) 1 Offset 128
   MemberDecorate 65(lightBuf) 2 Offset 192

   Decorate 13(matBuf) Block
   Decorate 15(Matrices) DescriptorSet 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 0
22:     16(int) Constant 1
25:     16(int) Constant 3

32(gl_PerVertex):             TypeStruct 7(fvec4) 6(float) 30
33:             TypePointer Output 32(gl_PerVertex)
34:             TypePointer Input 11(fvec3)
36:             TypePointer Input 11(fvec3)
37(Vectors):     36(ptr) Variable Output
38:     16(int) Constant 3
40:     16(int) Constant 3
46(bMain):     47(ptr) Variable Output
47:             TypePointer Output 11(fvec3)
This is the SPIR-V Assembly, Part III

```
50:         TypePointer Uniform 12
53a(Normal): 36(ptr) Variable Input
56(Color): 47(ptr) Variable Output
57(Color): 36(ptr) Variable Input
59:         TypeVector 6(float) 2
60:         TypePointer Output 59(fvec2)
61:         Variable Input 59(fvec2)
62:         TypePointer Uniform 65(lightBuf)
63:         Variable Uniform 65(lightBuf)
4(main): 2 Function None 3
5:         Label
10:         Variable Function
19: 18(ptr) AccessChain 15(Matrices) 21
20: 8 Load 19
22: 8 Load 21
24: 8 MatrixTimesMatrix 20 22
26: 8 Load 25
28: 8 Load 27
29: 8 MatrixTimesMatrix 25 28
30: 8 Load 29
32: 8 Load 31
34: 8 Load 33
36: 8 Load 35
38: 8 Load 37
40: 8 Load 38
42: 8 Load 41
44: 8 Load 43
46: 8 Load 45
48: 8 Load 47
50: 8 Load 49
52: 8 Load 51
54: 8 Load 53
56: 8 Load 55
58: 8 Load 57
60: 8 Load 59
62: 8 Load 61
64: 8 Load 63
```

SPIR-V: Printing the Configuration

```
glslangValidator –c
```

```
MacLights 32
MacClampFns 6
MacControlChars 32
MacTextureCounts 32
MacTextureUnits 64
MacTextureUniformComponents 4096
MacKeywordIndex 64
MacTextureTableTextureImageUnits 16
MacCombinedTextureImageUnits 32
MacControlCombinedTextureImageUnits 80
MacFragmentUniformComponents 4096
MacCombinedBuffers 32
MacCombinedVertexBuffers 32
MacVertexUniformBuffers 32
MacCombinedUniformBuffers 32
MacProgramControlBuffers 4
MacFunctionMetaBuffers 4
MacGlobalControlBuffers 4
MacGlobalFunctionBuffers 4
MacDirectControlBuffers 1
MacDirectFunctionBuffers 1
MacCombinedControlBuffers 1
MacCombinedFunctionBuffers 1
MacCombinedDirectBuffers 1
MacBuiltInBuffers 4
MacBuiltInFunctionBuffers 4
MacBuiltInDirectBuffers 4
MacBuiltInControlBuffers 4
MacCombinedBuiltInBuffers 4
MacUniformBuffers 4
MacControlBuffers 4
MacFunctionBuffers 4
MacDirectBuffers 4
MacBuiltInBuffers 4
```

```
MacLights 32
MacClampFns 6
MacControlChars 32
MacTextureCounts 32
MacTextureUnits 64
MacTextureUniformComponents 4096
MacKeywordIndex 64
MacTextureTableTextureImageUnits 16
MacCombinedTextureImageUnits 32
MacControlCombinedTextureImageUnits 80
MacFragmentUniformComponents 4096
MacCombinedBuffers 32
MacCombinedVertexBuffers 32
MacVertexUniformBuffers 32
MacCombinedUniformBuffers 32
MacProgramControlBuffers 4
MacFunctionMetaBuffers 4
MacGlobalControlBuffers 4
MacGlobalFunctionBuffers 4
MacDirectControlBuffers 1
MacDirectFunctionBuffers 1
MacCombinedControlBuffers 1
MacCombinedFunctionBuffers 1
MacCombinedDirectBuffers 1
MacBuiltInBuffers 4
MacBuiltInFunctionBuffers 4
MacBuiltInDirectBuffers 4
MacBuiltInControlBuffers 4
MacCombinedBuiltInBuffers 4
MacUniformBuffers 4
MacControlBuffers 4
MacFunctionBuffers 4
MacDirectBuffers 4
MacBuiltInBuffers 4
```

```
MacLights 32
MacClampFns 6
MacControlChars 32
MacTextureCounts 32
MacTextureUnits 64
MacTextureUniformComponents 4096
MacKeywordIndex 64
MacTextureTableTextureImageUnits 16
MacCombinedTextureImageUnits 32
MacControlCombinedTextureImageUnits 80
MacFragmentUniformComponents 4096
MacCombinedBuffers 32
MacCombinedVertexBuffers 32
MacVertexUniformBuffers 32
MacCombinedUniformBuffers 32
MacProgramControlBuffers 4
MacFunctionMetaBuffers 4
MacGlobalControlBuffers 4
MacGlobalFunctionBuffers 4
MacDirectControlBuffers 1
MacDirectFunctionBuffers 1
MacCombinedControlBuffers 1
MacCombinedFunctionBuffers 1
MacCombinedDirectBuffers 1
MacBuiltInBuffers 4
MacBuiltInFunctionBuffers 4
MacBuiltInDirectBuffers 4
MacBuiltInControlBuffers 4
MacCombinedBuiltInBuffers 4
MacUniformBuffers 4
MacControlBuffers 4
MacFunctionBuffers 4
MacDirectBuffers 4
MacBuiltInBuffers 4
```
SPIR-V: More Information

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

Installing bash on Windows

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

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

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

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

https://www.windowscentral.com/how-install-bash-shell-command-line-windows-10
Vulkan Sample Code

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

Sample Program Output
### Caveats on the Sample Code

1. I’ve written everything out in **appalling longhand**.

2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.

3. At times, I could have hidden complexity, but I didn’t. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that’s a secret to you.

4. I’ve setup Vulkan structs every time they are used, even though, in most cases, they could have been partially or completely setup once and then re-used.

5. At times, I’ve setup things that didn’t need to be setup just to show you what could go there.

6. There are good uses for C++ classes and methods here to hide some complexity, but I’ve not done that.

7. I’ve typedef’d a couple things to make the Vulkan phraseology more consistent.

8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.

9. At times, I have copied lines from vulkan.h into the code as comments to show you how certain options could be set.

10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

---

### Main Program

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

---

11/24/2018
# InitGraphics()

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

The Vertex Data is in a Separate File

```cpp
#include "SampleVertexData.cpp"

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

struct vertex VertexData[] =
{
    // triangle 0-2-3:
    // vertex #0:
    {
        { -1., -1., -1. },
        {  0.,  0., -1. },
        {  0.,  0.,  0. },
        {  1., 0. }
    },
    // vertex #2:
    {
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. }
    },
    // vertex #3:
    {
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        {  1.,  1.,  0. },
        {  0., 1. }
    },
    ...
};
```
What if you don’t need all of this information?

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

---

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums

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

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

   we would actually say in C:

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

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

My Conventions

“Init” in a function call name means that something is being setup that only needs to be setup once
The number after “Init” gives you the ordering
In the source code, after main( ) comes InitGraphics( ), then all of the InitxxYYY( ) functions in numerical order. After that comes the helper functions
“Find” in a function call name means that something is being looked for
“Fill” in a function call name means that some data is being supplied to Vulkan
“IN” and “OUT” ahead of pointer (address) arguments are just there to let you know how a pointer is used by the function. Otherwise, they have no significance.
#define IN
#define OUT

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

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 Vulkan (and OpenCL) pattern (get used to it):

result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
Your Sample2017.zip File Contains This

Window shader compiler

Linux shader compiler

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

```
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, "Surface Lost" },
    { VK_ERROR_NATIVE_WINDOW_IN_USE, "Native Window in Use" },
    { VK_SUBOPTIMAL, "Suboptimal" },
    { VK_ERROR_OUT_OF_DATE, "Out of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_POOL_MEMORY, "Out of Pool Memory" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR, "Invalid External Handle" },
};
```

Reporting Error Results, I

mjb – November 24, 2018

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

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

bool Paused;
bool Verbose;

#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
void InitGLFW()
{
    glfwInit();
    glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );
    glfwWindowHint( GLFW_RESIZABLE, GLFW_FALSE );
    MainWindow = glfwCreateWindow( Width, Height, "Vulkan Sample", NULL, NULL );
    VkResult result = glfwCreateWindowSurface( Instance, MainWindow, NULL, &Surface );
    glfwSetErrorCallback( GLFWErrorCallback );
    glfwSetKeyCallback( MainWindow, GLFWKeyboard );
    glfwSetCursorPosCallback( MainWindow, GLFWMouseMotion );
    glfwSetMouseButtonCallback( MainWindow, GLFWMouseMotion );
}

Setting Up GLFW
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:
                Mode++;
                if( Mode >= 2 )
                    Mode = 0;
                break;
            default:
                fprintf( FpDebug, "Unknow key hit: 0x%04x = \"%c\"\", key, key );
                fflush(FpDebug);
                break;
        }
    }
}
```

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
    }
}
```
### GLFW Mouse Motion Callback

```c
void
GLFWMouseMotion( GLFWwindow *window, double xpos, double ypos )
{
    int dx = (int)xpos - Xmouse;            // change in mouse coords
    int dy = (int)ypos - Ymouse;

    if( ( ActiveButton & LEFT ) != 0 )
    {
        Xrot += ( ANGFACT*dy );
        Yrot += ( ANGFACT*dx );
    }

    if( ( ActiveButton & MIDDLE ) != 0 )
    {
        Scale += SCLFACT * (float) ( dx - dy );
        // keep object from turning inside-out or disappearing:
        if( Scale < MINSCALE )
            Scale = MINSCALE;
    }

    Xmouse = (int)xpos;                     // new current position
    Ymouse = (int)ypos;
}
```

### Looping and Closing GLFW

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

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

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

What is GLM?

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

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

You can find it at:

http://glm.g-truc.net/0.9.8.5/

You invoke GLM like this:

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

If GLM is not installed in a system place, put it somewhere you can get access to.
Why are we even talking about this?

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

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

you would now have to say:

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

The Most Useful GLM Variables, Operations, and Functions

---

// constructor:

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

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

// emulating OpenGL transformations with concatenation:

```cpp
    glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
    glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
    glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```
// viewing volume (assign, not concatenate):

glm::mat4 glm::ortho( 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 );

GLM in the Vulkan sample.cpp Program

if( UseMouse )
{
    if( Scale < MINSCALE )
    {
        Scale = MINSCALE;
        Matrices.uModelMatrix = glm::mat4();           // identity
        Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
        Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0.) );
        Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0.) );
    }
}
else
{
    if( ! Paused )
    {
        const glm::vec3 axis = glm::vec3( 0., 1., 0. );
        Matrices.uModelMatrix = glm::rotate( glm::mat4( ), (float)glm::radians( 360.f*Time/SECONDS_PER_CYCLE ),   axis );
    }
}
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan’s projected Y is inverted from OpenGL.
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
Your Sample2017.zip File Contains GLM Already

Matrix Multiplication is not Commutative

Rotate, then translate

Translate, then rotate
Matrix Multiplication is Associative

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

One matrix —
the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

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

```cpp
// glm::mat4 Model = glm::mat4();
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::mat4 Matrix = Projection * View * Model;
// glm::mat3 NormalMatrix = glm::inverseTranspose( glm::mat3(Model) );
```
Why Isn't The Normal Matrix just the same as the Model Matrix?

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

Wrong!

Original object and normal

Right!

Original object and normal

\[
\text{glm::mat3 NormalMatrix} = \text{glm::inverseTranspose(glm::mat3(Model))};
\]
Instancing – What and why?

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

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

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

Making each instance look differently -- Approach #1

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

`gl_InstanceIndex` starts at 0

In the vertex shader:

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

```
gl_Position = PVM * vertex;
```
Making each Instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer and reference them

Still uses `gl_InstanceIndex`

In the vertex shader:

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

Read a new characteristic for each instance

Internally uses `gl_InstanceIndex`, but you don't

This is just the Vertex Input State Portion of the Graphics Pipeline Structure
This definition says that we should advance through the input buffer by this much every time we hit a new vertex.
How We Constructed the Graphics Pipeline Structure Before

```cpp
VkPipelineVertexInputStateCreateInfo vpvisci;

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

VkGraphicsPipelineCreateInfo vgpci;

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

How We Construct the Graphics Pipeline Structure Now

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

```cpp
VkVertexInputBindingDescription vvibd[2];

vvibd[0].binding = 0;       // which binding # this is
vvibd[0].stride = sizeof( struct vertex );  // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

vvibd[1].binding = 1;       // which binding # this is
vvibd[1].stride = sizeof( glm::vec3 );  // bytes between successive entries
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;
```

This definition says that we should advance through the input buffer by this much every time we hit a new instance.
How We Construct the Graphics Pipeline Structure Now

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

```cpp
VkVertexInputAttributeDescription vviad[5];
    // an array containing one of these per vertex attribute in all bindings
    // 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[5].location = 0;                  // location in the layout decoration
vviad[5].binding = 1;                   // which binding description this is part of
vviad[5].format = VK_FORMAT_VEC3;       // r, g, b
vviad[5].offset = 0;                      // just one element, so offset is 0
```

How We Construct the Graphics Pipeline Structure Now

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

```cpp
VkPipelineVertexInputStateCreateInfo vpvisci;
    vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
    vpvisci.pNext = nullptr;
    vpvisci.flags = 0;
    vpvisci.vertexBindingDescriptionCount = 2;
    vpvisci.pVertexBindingDescriptions = vvibd;
    vpvisci.vertexAttributeDescriptionCount = 5;
    vpvisci.pVertexAttributeDescriptions = vviad;
```

```cpp
VkGraphicsPipelineCreateInfo vgpci;
    vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    vgpci.pNext = nullptr;
    vgpci.flags = 0;
    vgpci.pVertexInputState = &vpvisci;
    ...
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
                                      PALLOCATOR, OUT pGraphicsPipeline );
```
How We Write the Vertex Shader Now

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

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

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

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

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 bunch of related uniform variables all at once?

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

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

**What are Descriptor Sets?**

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

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

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

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

Our example will assume the following shader uniform variables:

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

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    float uKa, uKd, uKs, uShininess;
    vec4 uLightPos;
    vec4 uLightSpecularColor;
    vec4 uEyePos;
} Light;

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

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

**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 used in the shader
- Knows the shader data structure
- Doesn't know where each piece of data starts
- Knows where the data starts
- Knows the data's size

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

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

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

**GPU:**
- Uniform data used in a “blob”*
- Knows where the data starts
- Doesn’t know the CPU or GPU data structure

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

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    float uKa, uKd, uKs, uShininess;
    vec4 uLightPos;
    vec4 uLightSpecularColor;
    vec4 uEyePos;
} Light;

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

```cpp
void 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.

Matrices Set DS Layout Binding:
- binding: 0
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
- descriptorCount: 1
- pipeline stage(s): VK_SHADER_STAGE_VERTEX_BIT
- pImmutableSamplers = (VkSampler *)nullptr

Light Set DS Layout Binding:
- binding: 0
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
- descriptorCount: 1
- stageFlags: VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT
- pImmutableSamplers = (VkSampler *)nullptr

Misc Set DS Layout Binding:
- binding: 0
- descriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
- descriptorCount: 1
- stageFlags: VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT
- pImmutableSamplers = (VkSampler *)nullptr

Sampler Set DS Layout Binding:
- binding: 0
- descriptorType: VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
- descriptorCount: 1
- stageFlags: VK_SHADER_STAGE_FRAGMENT_BIT
- pImmutableSamplers = (VkSampler *)nullptr

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;
Step 2: Define the Descriptor Set Layouts

**Pipeline Layout**

Array of Descriptor Set Layouts

---

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

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

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

VkDescriptorSetLayoutCreateInfo vdslc3;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.pNext = nullptr;
vdslc3.flags = 0;
vdslc3.bindingCount = 1;
vdslc3.pBindings = &TexSamplerSet[0];
result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc3, PALLOCATOR, OUT &DescriptorSetLayouts[3]);
return result;
```
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

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

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;`
- `vdbi0.offset = 0;`
- `vdbi0.range = sizeof(Matrices);`

**VkDescriptorBufferInfo**
- `vdbi1.buffer = MyLightUniformBuffer.buffer;`
- `vdbi1.offset = 0;`
- `vdbi1.range = sizeof(Light);`

**VkDescriptorBufferInfo**
- `vdbi2.buffer = MyMiscUniformBuffer.buffer;`
- `vdbi2.offset = 0;`
- `vdbi2.range = sizeof(Misc);`

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

Good to use `sizeof`

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

**VkWriteDescriptorSet**
- `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;`

**VkWriteDescriptorSet**
- `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;`
Step 5: Tell the Descriptor Sets where their data is

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

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

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

// number of stages in this pipeline = vertex + fragment
vgpci.stageCount = 2;

result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vgpci, PALLOCATOR, OUT &GraphicsPipeline);
```
Step 7: Bind Descriptor Sets into the Command Buffer when Drawing

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

The Graphics Pipeline

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".
2. There's a lot that goes into it.
3. For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

There is also a Vulkan Compute Pipeline.

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.
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;
}
```

Vulkan: A Pipeline Records the Following Items:

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

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

Creating a Typical Graphics Pipeline

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

```cpp
// VkPipelineShaderStageCreateInfo
vpssci[2];
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pNext = nullptr;
vpssci[0].flags = 0;
vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
#endif
VK_SHADER_STAGE_VERTEX_BIT
VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
VK_SHADER_STAGE_GEOMETRY_BIT
VK_SHADER_STAGE_FRAGMENT_BIT
VK_SHADER_STAGE_COMPUTE_BIT
VK_SHADER_STAGE_ALL_GRAPHICS
VK_SHADER_STAGE_ALL
#endif
vpssci[0].module = vertexShader;
vpssci[0].pName = "main";
vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[1].pNext = nullptr;
vpssci[1].flags = 0;
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
VkVertexInputBindingDescription vvibd[1]; // an array containing one of these per buffer being used
vvibd[0].binding = 0;           // which binding # this is
vvibd[0].stride = sizeof( struct vertex );              // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
#endif
VK_VERTEX_INPUT_RATE_VERTEX
VK_VERTEX_INPUT_RATE_INSTANCE
#endif
VkVertexInputAttributeDescription vviad[4];               // an array containing one of these per vertex attribute in all bindings
// 4 = vertex, normal, color, texture coord
vviad[0].location = 0;                  // location in the layout
vviad[0].binding = 0;                   // which binding 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_R32G32B32_SFLOAT
#define VK_FORMAT_STP           VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_XYZ           VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_VEC2          VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_ST            VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_XY            VK_FORMAT_R32G32_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
```

**Link in the Per-Vertex Attributes**

```cpp
// VkVertexInputBindingDescription
vvibd[1]; // an array containing one of these per buffer being used
vvibd[0].binding = 0; // which binding # this is
vvibd[0].stride = sizeof( struct vertex ); // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
#endif
VK_VERTEX_INPUT_RATE_VERTEX
VK_VERTEX_INPUT_RATE_INSTANCE
#endif
```
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;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
#ifdef CHOICES
VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
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
#endif
vpvisci.primitiveRestartEnable = VK_FALSE;

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

VkPipelineGeometryStateCreateInfo vpgsci;
vpptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
vpptsci.pNext = nullptr;
vpptsci.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”? 

```cpp
vpiasci.primitiveRestartEnable = VK_FALSE;
```

“Restart Enable” is used with:
- Indexed drawing.
- Triangle Fan and *Strip topologies

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

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

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

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

```cpp
short int restartIndex = ~0;
```
or,

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

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

Triangle Strip #0:
Triangle Strip #1:
Triangle Strip #2:
...
What is the Difference Between Changing the Viewport and Changing the Scissoring?

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

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

```c
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;
#endif

vprsci.cullMode = VK_CULL_MODE_NONE;   // recommend this because of the projMatrix[1][1] *= -1.;
#endif
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
#endif
vprsci.lineWidth = 1.f;
```

Declare information about how the rasterization will take place

What is “Depth Clamp Enable”??

```c
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.
**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**

```c
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.

```
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.

```
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_INVERTED
VK_LOGIC_OP_COPY_INVERTED
VK_LOGIC_OP_OPAQUE
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?

```cpp
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };
#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport()
VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor()
VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth()
VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias()
VK_DYNAMIC_STATE_BLEND_CONSTANTS        -- vkCmdSetBlendConstants()
VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthBounds()
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompareMask()
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK     -- vkCmdSetStencilWriteMask()
VK_DYNAMIC_STATE_STENCIL_REFERENCE     -- vkCmdSetStencilReference()
#endif

class VkPipelineDynamicStateCreateInfo

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;
```

Stencil Operations for Front and Back Faces

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

// FRONT 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

// FRONT 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

// FRONT
vsosf.compareOp = VK_COMPARE_OP_GREATER;
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;
```
Uses for Stencil Operations

Polygon edges without Z-fighting

Magic Lenses

Operations for Depth Values

```cpp
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;
#ifdef CHOICES
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
#endif
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…)

VkGraphicsPipelineCreateInfo  
v pci;


gpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;

gpci.pNext = nullptr;

gpci.flags = 0;

#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

gpci.stageCount = 2;                           // number of stages in this pipeline

gpci.pStages = vpssci;

gpci.pVertexInputState = &vpvisci;

gpci.pInputAssemblyState = &vpiasci;

gpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;

gpci.pViewportState = &vpvsci;

gpci.pRasterizationState = &vprsci;

gpci.pMultisampleState = &vpmsci;

gpci.pDepthStencilState = &vpdssci;

gpci.pColorBlendState = &vpcbsci;

gpci.pDynamicState = &vpdsci;

gpci.layout = IN GraphicsPipelineLayout;

gpci.renderPass = IN RenderPass;

gpci.subpass = 0;                              // subpass number

gpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;

gpci.basePipelineIndex = 0;

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

return result;

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

vkCmdBindPipeline( CommandBuffers[nextImageIndex], 
VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
Queues and Command Buffers

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

Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Command Buffer

Command Buffer
Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

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;
    uint32_t count = -1;
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    for( unsigned int i = 0; i < count; i++ )
    for( unsigned int i = 0; i < count; i++ )
        if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
            if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
                return i;
                return i;
    return -1;
    return -1;
}
```

Creating a Logical Device Queue Needs to Know Queue Family Information

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

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

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

VkQueue Queue:
VkQueue Queue:
    uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
    uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
    uint32_t queueIndex = 0;
    uint32_t queueIndex = 0;
result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
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;
  vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
  result = vkCreateCommandPool
          ( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool );
  return result;
}
```

Creating the Command Buffers

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

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

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

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

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

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

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>vkSemaphoreCreateInfo</th>
<th>vkCreateSemaphore()</th>
<th>vkAllocateCommandBuffer()</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkCommandBufferBeginInfo</td>
<td>vkBeginCommandBuffer()</td>
<td></td>
</tr>
</tbody>
</table>
```

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>VkSemaphoreCreateInfo</th>
<th>vkCreateSemaphore()</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkCommandBufferAllocateInfo</td>
<td>AllocateCommandBuffer()</td>
</tr>
<tr>
<td>VkCommandBufferBeginInfo</td>
<td>BeginCommandBuffer()</td>
</tr>
</tbody>
</table>
```
These are the Commands that could be entered into the Command Buffer, I

vkCmdBeginQuery(commandBuffer, flags);
vkCmdBeginRenderPass(commandBuffer, const contents);
vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
vkCmdBindIndirectBuffer(commandBuffer, indexType);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertextBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdCopyBuffer(commandBuffer, pRegions);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);
vkCmdDrawIndirect(commandBuffer, stride);
vkCmdDrawIndirectCountAMD(commandBuffer, stride);
vkCmdEndQuery(commandBuffer, query);
vkCmdEndRenderPass(commandBuffer);
vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);

vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
vkCmdNextSubpass(commandBuffer, contents);
vkCmdPixelBarrier(commandBuffer, srcImage, dstImage, dependencyFlags, memoryBarrierCount, memoryBarriers, bufferMemoryBarriers, imageMemoryBarriers);
vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);
vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, size, pValues);
vkCmdPushDescriptorSet(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);
vkCmdPushDescriptorSetWithTemplate(commandBuffer, descriptorUpdateTemplate, layout, set, pData);
vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo);
vkCmdResetEvent(commandBuffer, event, stageMask);
vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);
vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
vkCmdSetDeviceMaskKHX(commandBuffer, deviceMask);
vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, discardRectangles);
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, pWaitStages, pWaitDstStageMask, pMemBarriers, pMemoryBarrierCount, pMemoryBarriers, imageMemoryBarriers, pimageMemoryBarriers);
vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query);

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

vkCmdBeginQuery(commandBuffer, flags);
vkCmdBeginRenderPass(commandBuffer, const contents);
vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
vkCmdBindIndirectBuffer(commandBuffer, indexType);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertextBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdCopyBuffer(commandBuffer, pRegions);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);
vkCmdDrawIndirect(commandBuffer, stride);
vkCmdDrawIndirectCountAMD(commandBuffer, stride);
vkCmdEndQuery(commandBuffer, query);
vkCmdEndRenderPass(commandBuffer);
vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);
VkResult

RenderScene()
{
VkResult result;
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

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

uint32_t nextImageIndex;
vkAcquireNextImage(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;
    vccv.float32[0] = 0.0;
    vccv.float32[1] = 0.0;
    vccv.float32[2] = 0.0;
    vccv.float32[3] = 1.0;

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

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

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

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

vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], 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 );

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

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

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

Create fence

Fill in the queue information

Submit the queue

Wait for the fence

Create fence

Get the queue

Fill in the queue information

Submit the queue

Wait for the fence

The Swap Chain

Mike Bailey

mj@cs.oregonstate.edu

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

Vulkan.

SIGGRAPH
Tokyo

mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan
How We Think of OpenGL Framebuffers

Vulkan Thinks of it as a Ring Buffer
What is a Swap Chain?

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

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

Swap Chains are arranged as a ring buffer.

Swap Chains are tightly coupled to the window system.

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

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

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

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

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

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

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

We Need to Find Out What our Display Capabilities Are

```
VkSurfaceCapabilities vsc;
vkGetPhysicalDeviceSurfaceCapabilities( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "%s\n\n\nvkGetPhysicalDeviceSurfaceCapabilities:
" );
...
VkBool32 supported;
result = vkGetPhysicalDeviceSurfaceSupport( PhysicalDevice, FindQueueFamilyThatDoesGraphics(), Surface, &supported );
if( supported == VK_TRUE )
  fprintf( FpDebug, "** This Surface is supported by the Graphics Queue **\n" );

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

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

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

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

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

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

We Need to Find Out What our Display Capabilities Are

Creating a Swap Chain

vkGetDevicePhysicalSurfaceCapabilities( )

VkSurfaceCapabilities

surface imageFormat
  imageColorSpace
  imageExtent
  imageArrayLayers
  imageUsage
  imageSharingMode
  preTransform
  compositeAlpha
  presentMode
  clipped

VkSwapchainCreateInfo

minImageCount
  currentExtent
  minImageExtent
  maxImageExtent
  maxImageArrayLayers
  supportedTransforms
  currentTransform
  supportedCompositeAlpha

vkCreateSwapchain( )

vkGetSwapChainImages( )

vkCreateImageView( )
Creating a Swap Chain

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

VkSwapchainCreateInfo vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO;
vscci.pNext = nullptr;
vscci.flags = 0;
vscci.surface = Surface;
vscci.minImageCount = 2;                 // double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR;
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;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT;
vscci.colorbarLayers = 1;
vscci.clipped = VK_TRUE;
result = vkCreateSwapchain( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
```

Creating the Swap Chain Images and Image Views

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

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

// present views for the double-buffering:
PresentImageViews = new VkImageView[ imageCount ];
for( unsigned int i = 0; i < imageCount; i++ )
{
    VkImageViewInitInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    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, &vsci, PALLOCATOR, OUT &imageReadySemaphore );

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

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

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

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

Rendering into the Swap Chain, II

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

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

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

VkSubmitInfo vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.waitSemaphoreCount = 1;
    vsi.pWaitSemaphores = &imageReadySemaphore;
    vsi.pWaitDstStageMask = &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 );

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

result = vkQueuePresent( presentQueue, IN &vpi );
VkResult
RenderScene()
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, &vsci, nullptr, OUT &imageReadySemaphore);
    uint32_t nextImageIndex;
    vkAcquireNextImage(LogicalDevice, SwapChain, UINT64_MAX, imageReadySemaphore, VK_NULL_HANDLE, OUT &nextImageIndex);

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

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

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

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

    VkRect2D r2d = { 0, 0 }, e2d = { Width, Height };
    VkRenderPassBeginInfo vrpbi;
    vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    vrpbi.pNext = nullptr;
    vrpbi.renderPass = RenderPass;
    vrpbi.framebuffer = Framebuffers[nextImageIndex];
    vrpbi.renderArea = r2d;
    vrpbi.clearValueCount = 2;
    // used for VK_ATTACHMENT_LOAD_OP_CLEAR
    vrpbi.pClearValues = vcv;
    vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], &vrpbi, VK_SUBPASS_CONTENTS_INLINE);
    vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
VkViewport viewport = {
    0.,  // x
    0.,  // y
    (float)Width,  // minDepth
    (float)Height,  // maxDepth
};

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

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

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

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

//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] );

VkFenceCreateInfo vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

vfci.pNext = nullptr;

vfci.flags = 0;

VkFence renderFence;

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

vkFenceCreateInfo
vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

vfci.pNext = nullptr;

vfci.flags = 0;

VkFence renderFence;

vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );
VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

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

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

vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );

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

result = vkQueuePresent( presentQueue, IN &vpi );

vkDestroySemaphore( LogicalDevice, imageReadySemaphore, PALLOCATOR );
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 );
```

...  
```c
glEnd( );
```

Disable texture mapping:

```c
glDisable( GL_TEXTURE_2D );
```

**Triangles in an Array of Structures**

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

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

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

Or, for a sphere,

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

From the Sphere code:

\[
s = (\ \text{lng} + \ M_{\text{PI}} \ ) / (2.*M_{\text{PI}}); \\
t = (\ \text{lat} + \ M_{\text{PI}/2.}) / M_{\text{PI}};
\]
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

\[ s = ? \quad t = ? \]

You really are at the mercy of whoever did the modeling...
VkDescriptorSetLayoutBinding TexSamplerSet[1];
TexSamplerSet[0].binding = 0;
TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
// uniform sampler2D uSampler
// v4d rgb = texture(uSampler, vST);
TexSamplerSet[0].descriptorCount = 1;
TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;

VkDescriptorImageInfo vdi0;
vdi0.sampler = MyPuppyTexture.texSampler;
vdi0.imageView = MyPuppyTexture.texImageView;
vdi0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

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

---

**Memory Types**

CPU Memory

You create your texture here

GPU Memory

Host Visible GPU Memory
(the "Staging Buffer")

Device Local GPU Memory

Texture Sampling Hardware

RGBA to the Shader
Memory Types

NVIDIA Discrete Graphics:

11 Memory Types:
Memory 0: HostVisible HostCoherent HostCached
Memory 0: HostVisible HostCoherent HostCached
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

```
gTexParameter( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
gTexParameter( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
gTexParameter( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
gTexParameter( 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 influenced the colors in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets filtered into in resolution-level textures.

Texture Mip*-mapping

- Total texture storage is ~ 2x what it was without mip-mapping
- Graphics hardware determines which level to use based on the texels : pixels ratio.
- In addition to just picking one mip-map level, the rendering system can sample from two of them, one less 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: mult 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 use 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_COLOR_ATTACHMENT_BIT
        VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
        VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
        #endif
        vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage );
    }
    VkImageCreateInfo viici;
    viici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    viici.pNext = nullptr;
    viici.flags = 0;
    viici.imageType = VK_IMAGE_TYPE_2D;
    viici.format = VK_FORMAT_R8G8B8A8_UNORM;
    viici.extent.width = texWidth;
    viici.extent.height = texHeight;
    viici.extent.depth = texDepth;
    viici.mipLevels = 1;
    viici.arrayLayers = 1;
    viici.samples = VK_SAMPLE_COUNT_1_BIT;
    viici.tiling = VK_IMAGE_TILING_LINEAR;
    #ifdef CHOICES
    VK_IMAGE_TILING_LINEAR
    #endif
    viici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT;
    #ifdef CHOICES
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT
    VK_IMAGE_USAGE_TRANSFER_DST_BIT
    VK_IMAGE_USAGE_SAMPLED_BIT
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
    VK_IMAGE_USAGE_TRANSFER_DST_BIT
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
    #endif
    viici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    result = vkCreateImage( LogicalDevice, IN &viici, PALLOCATOR, OUT &textureImage );
    return result;
}
```c
#ifdef CHOICES
VK_IMAGE_LAYOUT_UNDEFINED
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");
    fprintf(FpDebug, "	offset = %lld\n", vsl.offset);
    fprintf(FpDebug, "	size = %lld\n", vsl.size);
    fprintf(FpDebug, "	rowPitch = %lld\n", vsl.rowPitch);
    fprintf(FpDebug, "	arrayPitch = %lld\n", vsl.arrayPitch);
    fprintf(FpDebug, "	depthPitch = %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) {
memorycopy(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();

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;
vcbi.pNext = nullptr;
vcbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

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 = 0;
vimb.dstAccessMask = 0;
vimb.subresourceRange = visr;

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, 0, 0, 0, 0, 0);

result = vkEndCommandBuffer(TextureCommandBuffer);
// transition the texture buffer layout:

// VkImageSubresourceRange
vk::imageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

// VkImageMemoryBarrier
vk::imageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = 0;
vimb.dstAccessMask = 0;
vimb.subresourceRange = visr;

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

// now do the final image transfer:

vk::imageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.baseMipLevel = 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;

vk::imageCopy vic;
vic.srcSubresource = visl;
vic.srcOffset = vo3;
vic.dstSubresource = visl;
vic.dstOffset = vo3;
vic.extent = ve3;

vkCmdCopyImage(textureCommandBuffer,
stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &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);
}

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

    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.image = textureImage;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    vivci.subresourceRange = visr;

    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
    return result;
}

Note that, at this point, the CPU buffer and the GPU Staging Buffer are no longer needed, and can be destroyed.
Reading in a Texture from a BMP File

```c
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.

---

Vulkan

Physical Devices

Mike Bailey

mjb@cs.oregonstate.edu

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

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Command Buffer

Command Buffer

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: Identifying the Physical Devices

```
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, \n%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;
}
```

Which Physical Device to Use, I

```
int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    if ( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not get the physical device properties of device %d\n", i );
        return VK_SHOULD_EXIT;
    }
    fprintf( FpDebug, \nDevice %2d:\n", i);
    fprintf( FpDebug, "API version: %d\n", vpdp.apiVersion );
    fprintf( FpDebug, "Driver version: %d\n", vpdp.driverVersion );
    fprintf( FpDebug, "Vendor ID: 0x%04x\n", vpdp.vendorID );
    fprintf( FpDebug, "Device ID: 0x%04x\n", vpdp.deviceID );
    printf( FpDebug, "Physical Device Type: %d =\n", 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, "Device Name: %s\n", vpdp.deviceName );
    fprintf( FpDebug, "Pipeline Cache Size: %d\n", vpdp.pipelineCacheSize );
```
Asking About the Physical Device’s Features

```cpp
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 1080ti Produced

```cpp
vkEnumeratePhysicalDevices:

Device 0:
    API version: 4194360
    Driver version: 4194360
    Vendor ID: 0x10de
    Device ID: 0xb06
    Physical Device Type: 2 = (Discrete GPU)
    Device Name: GeForce GTX 1080 Ti
    Pipeline Cache Size: 13

Device #0 selected (GeForce GTX 1080 Ti):

Physical Device Features:
    geometryShader = 1
    tessellationShader = 1
    multiDrawIndirect = 1
    wideLines = 1
    largePoints = 1
    multiViewport = 1
    occlusionQueryPrecise = 1
    pipelineStatisticsQuery = 1
    shaderFloat64 = 1
    shaderInt64 = 1
    shaderInt16 = 0
```
Here’s What the Intel HD Graphics 520 Produced

vkEnumeratePhysicalDevices:

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

Device #0 selected (Intel(R) HD Graphics 520)

Physical Device Features:
   geometryShader = 1
   tessellationShader = 1
   multiDrawIndirect = 1
   wideLines = 1
   largePoints = 1
   multiViewport = 1
   occlusionQueryPrecise = 1
   pipelineStatisticsQuery = 1
   shaderFloat64 = 1
   shaderInt64 = 1
   shaderInt16 = 1

Which Physical Device to Use, II

// need some logical here to decide which physical device to select:
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
   discreteSelect = i;
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
   integratedSelect = i;
int which = -1;
if( discreteSelect >= 0 )
{
   which = discreteSelect;
   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 Different Memories

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

fprintf( FpDebug, "%d Memory Types:
", vpdmp.memoryTypeCount );
for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
{
    VkMemoryType vmt = vpdmp.memoryTypes[i];
    fprintf( FpDebug, "Memory %2d: ", i );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTYDEVICE_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_MEMORYPROPERTY_HOST_CACHED_BIT ) != 0 )    fprintf( FpDebug, "HostCached" );
    if( ( vmt.propertyFlags & VK_MEMORYPROPERTY_LAZILY_ALLOCATED_BIT ) != 0 )    fprintf( FpDebug, "LazilyAllocated" );
    fprintf(FpDebug, ";
" );
}

fprintf( FpDebug, "%d Memory Heaps:
", vpdmp.memoryHeapCount );
for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ )
{
    VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
    fprintf( FpDebug, "Heap %d: ", i);
    if( ( vmh.flags & VK_MEMORY_HEAPDEVICE_LOCAL_BIT ) != 0 )     fprintf( FpDebug, "DeviceLocal" );
    fprintf(FpDebug, "size = 0x%08lx
", (unsigned long int)vmh.size);
}
```

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
Asking About the Physical Device's Queue Families

```c
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
fprintf( FpDebug, "Found %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 =  1 ; Transfer
2: queueCount =  8 ; Compute
Logical Devices

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

Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Command Buffer

Command Buffer
Looking to See What Device Layers are Available

```cpp
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_surface",
    "VK_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, deviceLayers[i].layerName, &layerCount, deviceLayers);
```

Looking to See What Device Extensions are Available

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

3 physical device layers enumerated:

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

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

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

Vulkan: Specifying a Logical Device Queue

```c
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;
```
Vulkan: Creating a Logical Device

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

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

Vulkan: Creating the Logical Device’s Queue

```c
// get the queue for this logical device:

vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue ); // 0, 0 = queueFamilyIndex, queueIndex
```
Layers and Extensions

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

They are not always necessary, but when you need them, you will be really glad they are there!
const char * instanceLayers[ ] =
{
    "VK_LAYER_LUNARG_api_dump", // turn this on if want to see each function call and its arguments (very slow!)
    "VK_LAYER_LUNARG_core_validation",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_NV_optimus"
};

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

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

// see what layers are available:
vkEnumerateInstanceLayerProperties( &numLayersAvailable, (VkLayerProperties *)nullptr );
InstanceLayers = new VkLayerProperties[ numLayersAvailable ];
result = vkEnumerateInstanceLayerProperties( &numLayersAvailable, InstanceLayers );

// see what extensions are available:

13 instance layers available:
0x00400033  2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
0x00400033  1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0x00400033  1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
0x00400033  1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00400033  1  'VK_LAYER_LUNARG_parameter_validation'  'LunarGValidation Layer'
0x00400033  1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
0x00400033  1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
0x00400033  1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
0x00400033  1  'VK_LAYER_GOOGLE_threading'  'Google Validation Layer'
0x00400033  1  'VK_LAYER_GOOGLE_unique_objects'  'Google Validation Layer'
0x00400033  1  'VK_LAYER_LUNARG_vktrace'  'Vktrace tracing library'
0x00400033  1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x00400033  1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x00400033  34  'VK_LAYER_RENDERDOC_Capture'  'Debugging capture layer for RenderDoc'
vkEnumerateInstanceExtensionProperties:

11 extensions enumerated:
0x00000008 'VK_EXT_debug_report'
0x00000001 'VK_EXT_display_surface_counter'
0x00000001 'VK_get_physical_device_properties2'
0x00000001 'VK_get_surface_capabilities2'
0x00000019 'VK_surface'
0x00000006 'VK_win32_surface'
0x00000001 'VK_KHR_device_group_creation'
0x00000001 'VK_external_fence_capabilities'
0x00000001 'VK_external_memory_capabilities'
0x00000001 'VK_external_semaphore_capabilities'
0x00000001 'VK_NV_external_memory_capabilities'

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

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

Looking to See What Extensions are Both Wanted and Available
Will now ask for 3 instance extensions
   VK_surface
   VK_win32_surface
   VK_EXT_debug_report

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

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    // need some logical here to decide which physical device to select:
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        discreteSelect = i;
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        integratedSelect = i;
}

int which = -1;
if( discreteSelect >= 0 )
{
    which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
}
else if( integratedSelect >= 0 )
{
    which = integratedSelect;
    PhysicalDevice = physicalDevices[which];
}
else
{
    fprintf( FpDebug, "Could not select a Physical Device
" );
    return VK_SHOULD_EXIT;
}
delete[ ] physicalDevices;
```
vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp );
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceQueueFamilyProperties( PhysicalDevice, &count, OUT vqfp );
delete[ ] vqfp;

VkResult result;
float   queuePriorities[NUM_QUEUES_WANTED] =
(  
  1.,
);
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];
vdqci[i].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci[i].pNext = nullptr;
vdqci[i].flags = 0;
vdqci[i].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[i].queueCount = 1;                // how many queues to create
vdqci[i].pQueuePriorities = queuePriorities;    // array of queue priorities [0.,1.]
const char * myDeviceLayers[] =  
(  
  "VK_LAYER_LUNARG_api_dump",
  "VK_LAYER_LUNARG_core_validation",
  "VK_LAYER_LUNARG_image",
  "VK_LAYER_LUNARG_object_tracker",
  "VK_LAYER_LUNARG_parameter_validation",
  "VK_LAYER_NV_optimus"
);  
const char * myDeviceExtensions[] =
(  
  "VK_swapchain",
);
uint32_t layerCount;
voidEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);
for (unsigned int i = 0; i < layerCount; i++)
{
  // see what device extensions are available:
  uint32_t extensionCount;
  voidEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
  (VkExtensionProperties *)nullptr);
  VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
  result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
deviceExtensions);
}
delete[] deviceLayers;

4 physical device layers enumerated:
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
voidEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_NV_optimus':

0x00400033 1 'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
voidEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':

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

0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
voidEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
vkEnumerateDeviceLayerProperties:

3 physical device layers enumerated:
0x00400038 1 `VK_LAYER_NV_optimus` `NVIDIA Optimus layer`
  0 device extensions enumerated for `VK_LAYER_NV_optimus`:

0x00400033 1 `VK_LAYER_LUNARG_object_tracker` `LunarG Validation Layer`
  0 device extensions enumerated for `VK_LAYER_LUNARG_object_tracker`:

0x00400033 1 `VK_LAYER_LUNARG_parameter_validation` `LunarG Validation Layer`
  0 device extensions enumerated for `VK_LAYER_LUNARG_parameter_validation`:

---

**Synchronization**

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

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

Ask for Something  →  Try to Use the Something  →  Semaphore
Creating a Semaphore

```cpp
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 );
```

Semaphores Example during the Render Loop

```cpp
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;
vkAcquireNextImage( 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 );
```

Could be an array of semaphores
Fences

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

#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

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

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

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

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

Could be an array of fences
### Fence Example

```cpp
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;
vs.i.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vs.i.pNext = nullptr;
vs.i.waitSemaphoreCount = 1;
vs.i.pWaitSemaphores = &imageReadySemaphore;
vs.i.pWaitDstStageMask = &waitAtBottom;
vs.i.commandBufferCount = 1;
vs.i.pCommandBuffers = &CommandBuffers[nextImageIndex];

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

### Events

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

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

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

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

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

Controlling Events from the Device

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

Note: the GPU cannot test for an event, but it can block waiting for one
Pipeline Barriers: A case of Gate-ing and Wait-ing

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

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

We don't any one of these commands to have to wait on a previous command unless you say so. In general, we want all of these commands to be able to run “flat-out”. But, if we do that, surely there will be nasty race conditions!
From the Command Buffer Notes:
These are the Commands that can be entered into the Command Buffer, II

vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, data);
vkCmdNextSubpass(commandBuffer, contents);
vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdProcessCommandNVX(commandBuffer, pProcessCommandInfo);
vkCmdPushConstant(commandBuffer, layout, stageFlags, offset, size, pValues);
vkCmdPushDescriptorSetWithTemplate(commandBuffer, descriptorUpdateTemplate, layout, set, pDescriptorWrites);
vkCmdResetEvent(commandBuffer, event, stageMask);
vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
vkCmdSetDeviceMaskKHX(commandBuffer, deviceMask);
vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);
vkCmdSetEvent(commandBuffer, event, stageMask);
vkCmdSetEventVisibility(commandBuffer, event, stageMask);
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);

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

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

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

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

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

vkCmdPipelineBarrier( commandBuffer,
srcStageMask, // Guarantee that this pipeline stage has completely generated one set of data before …
dstStageMask, // ... allowing this pipeline stage to consume it
VK_DEPENDENCY_BY_REGION_BIT,
memoryBarrierCount, pMemoryBarriers,
bufferMemoryBarrierCount, pBufferMemoryBarriers,
imageMemoryBarrierCount, pImageMemoryBarriers
);

The Scenario

src cars

dst cars

TOP_OF_PIPE Street

VERTEX_INPUT Street

VERTEX_SHADER Street

BOTTOM_OF_PIPE Street

TRANSFER_BIT Street

COLOR_ATTACHMENT_OUTPUT Street

FRAGMENT_SHADER Street

mjb – November 24, 2018
The Scenario Rules

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

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

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

**Pipeline Stages**

**Access Masks** – What are you interested in generating or consuming this memory for?
### Pipeline Stages and what Access Operations can Happen There

<table>
<thead>
<tr>
<th>Stage</th>
<th>Access Indirect Command Read Bit</th>
<th>Access Index Read Bit</th>
<th>Access Vertex Attribute Read Bit</th>
<th>Access Uniform Read Bit</th>
<th>Access Input Attachment Read Bit</th>
<th>Access Shader Read Bit</th>
<th>Access Shader Write Bit</th>
<th>Access Color Attachment Read Bit</th>
<th>Access Color Attachment Write Bit</th>
<th>Access Depth stencil Attachment Read Bit</th>
<th>Access Depth stencil Attachment Write Bit</th>
<th>Access Transfer Read Bit</th>
<th>Access Transfer Write Bit</th>
<th>Access Host Read Bit</th>
<th>Access Host Write Bit</th>
<th>Access Memory Read Bit</th>
<th>Access Memory Write Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<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_PIPELINE_STAGE_TRANSFER_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>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Access Indirect Command Read Bit</th>
<th>Access Index Read Bit</th>
<th>Access Vertex Attribute Read Bit</th>
<th>Access Uniform Read Bit</th>
<th>Access Input Attachment Read Bit</th>
<th>Access Shader Read Bit</th>
<th>Access Shader Write Bit</th>
<th>Access Color Attachment Read Bit</th>
<th>Access Color Attachment Write Bit</th>
<th>Access Depth stencil Attachment Read Bit</th>
<th>Access Depth stencil Attachment Write Bit</th>
<th>Access Transfer Read Bit</th>
<th>Access Transfer Write Bit</th>
<th>Access Host Read Bit</th>
<th>Access Host Write Bit</th>
<th>Access Memory Read Bit</th>
<th>Access Memory Write Bit</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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example: Be sure we are done writing an output image before using it for something else**

**Stages**
- VK_PIPELINE_STAGE_TOP_OFPIPE_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

---

**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

Stages

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

Access types

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

The Scenario

cars

src
dst

src cars

dst cars

src (no access setting needed)
VkImageLayout – How an Image gets Laid Out in Memory depends on how it will be Used

Here, the use of vkCmdPipelineBarrier() is to simply change the layout of an image

Push Constants

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

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

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

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:

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

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

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

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

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

```cpp
VkPushConstantRange vpcr[1];
    vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
                       VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
    vpcr[0].offset = 0;
    vpcr[0].size = sizeof(glm::mat4);

VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = DescriptorSetLayouts;
    vplci.pushConstantRangeCount = 1;
    vplci.pPushConstantRanges = &vpcr;
    result = vkCreatePipelineLayout( LogicalDevice, &vplci, PALLOCATOR,
                                    &GraphicsPipelineLayout );
```

Creating a Pipeline

Diagram showing the creation of a pipeline with various stages and bindings. The diagram includes nodes for Shader, Input Assembly State, Viewport State, Color Blend State, Depth Stencil State, Dynamic State, and Pipeline Layout. The pipeline creation function `vkCreateGraphicsPipeline()` is highlighted, along with related functions and structures such as `VkPipelineShaderStageCreateInfo`, `VkPipelineInputAssemblyStateCreateInfo`, `VkPipelineColorBlendAttachmentState`, and `VkPipelineDynamicStateCreateInfo`. The diagram also indicates locations where specializations can be set.
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)

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

Locations?
Positioning Part #1 With Respect to Ground

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

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

---

Positioning Part #2 With Respect to Ground

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

$$[M_{1/G}] = [T_{1/G}] \cdot [R_{\theta_1}] \cdot [T_{2/1}] \cdot [R_{\theta_2}]$$

$$[M_{2/G}] = [M_{1/G}] \cdot [M_{2/1}]$$
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 \( T_{1/G} \)

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

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

In the Reset Function

```c
struct arm Arm1;
struct arm Arm2;
struct arm Arm3;

Arm1.armMatrix = glm::mat4( );
Arm1.armColor = glm::vec3( 0.f, 1.f, 0.f );
Arm1.armScale = 6.f;
Arm2.armMatrix = glm::mat4( );
Arm2.armColor = glm::vec3( 1.f, 0.f, 0.f );
Arm2.armScale = 4.f;
Arm3.armMatrix = glm::mat4( );
Arm3.armColor = glm::vec3( 0.f, 0.f, 1.f );
Arm3.armScale = 2.f;
```

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

```c
VkPushConstantRange
vpcr[1];

vpcr[0].stageFlags =
   VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
| VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

vpcr[0].size = sizeof(struct arm);

VkPipelineLayoutCreateInfo
vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

result = vkCreatePipelineLayout( LogicalDevice, 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();

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

m1g = glm::rotate(m1g, rot1, zaxis);

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

m1g = glm::rotate(m1g, rot2, zaxis);

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

m1g = glm::rotate(m1g, rot3, zaxis);

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

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

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

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

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

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

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

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

Setting Up Global Memory Pipeline Barriers

```c
VkMemoryBarrier vmb;  
vmb.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;  
vmb.pNext = nullptr;  
vmb.srcAccessMask = 0;  
vmb.dstAccessMask = 0;
```

```c
vkCmdPipelineBarrier(commandBuffer,  
    srcStageMask,  
    dstStageMask,  
    VK_DEPENDENCY_BY_REGION_BIT,  
    1, IN &vmb,  
    0, nullptr,  
    0, nullptr);
```
Setting Up Buffer Memory Pipeline Barriers

```c
VkBufferMemoryBarrier vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessMask =
vbmb.dstAccessMask =
vbmb.srcQueueFamilyIndex =
vbmb.dstQueueFamilyIndex =
vbmb.buffer =
vbmb.offset =
vbmb.size =
vkCmdPipelineBarrier( commandBuffer,
srcStageMask,
dstStageMask,
VK_DEPENDENCY_BY_REGION_BIT,
0, NULL,
1, IN &vbmb,
0, nullptr );
```

Setting Up Image Memory Pipeline Barriers

```c
VkImageMemoryBarrier vimb;
  vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  vimb.pNext = nullptr;
  vimb.srcAccessMask =
  vimb.dstAccessMask =
  vimb.oldLayout =
  vimb.newLayout =
  vimb.srcQueueFamilyIndex =
  vimb.dstQueueFamilyIndex =
  vimb.image =
  vimb.subResourceRange =
vkCmdPipelineBarrier( commandBuffer,
srcStageMask,
dstStageMask,
VK_DEPENDENCY_BY_REGION_BIT,
0, NULL,
0, NULL,
1, IN &vimb );
```
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);

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

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

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

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

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

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

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

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

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

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```
In the Vertex Shader

```glsl
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
```

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

Anti-aliasing by Multisampling

4x

16x
MultiSampling

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

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

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

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.

---

Vulkan Distribution of Sampling Points within a Pixel

---

160
Vulkan Distribution of Sampling Points within a Pixel

Consider Two Triangles Whose Edges Pass Through the Same Pixel
Supersampling

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

# Fragment Shader calls = 8

Multisampling

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

# Fragment Shader calls = 2
Setting up the Image

```c
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;
    vgpci.pMultisampleState = &vpmsci;

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

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;
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;
```

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

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

```
Setting up the Image

```c
VkSubpassDescription vsd;
vsd.flags = 0;
vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd.inputAttachmentCount = 0;
vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
vsd.colorAttachmentCount = 1;
vsd.pColorAttachments = &colorReference;
vsd.pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd.pDepthStencilAttachment = &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 = &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkDependency *nullptr);

result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
```
Resolving the Image:
Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

```c
VLOffset3D vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;

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

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

VkImageResolve vir;
vir.srcSubresource = visl;
vir.srcOffset = vo3;
vir.dstSubresource = visl;
vir.dstOffset = vo3;
vir.extent = ve3;

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

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:

```
Attachment

Attachment

Subpass -- Subpass -- Subpass

Framebuffer
```

**Back in Our Single-pass Days**

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

```
Render

Framebuffer
```

Here comes a quick reminder of how we did that.

Afterwards, we will extend that.
 VkAttachmentDescription
  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;
  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;

 VkAttachmentReference
  colorReference.attachment = 0;
  colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
  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 some image processing on the outcome.

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

Multipass Algorithm to Render and then Image Process

Original Sharpened Edge Detected

No Noise Noise Noise

Original Sharpened Edge Detected

No Noise Noise Noise

Original Sharpened Edge Detected

No Noise Noise Noise

Original Sharpened Edge Detected

No Noise Noise Noise

Original Sharpened Edge Detected

No Noise Noise Noise
### Multipass, I

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

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

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

### Multipass, II

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

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

VkAttachmentReference outputReference;
outputReference.attachment = 2;
outputReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```
Multipass, III

```c
VkSubpassDescription vsd[2];
vsd[0].flags = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[0].inputAttachmentCount = 0;
vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
vsd[0].colorAttachmentCount = 1;
vsd[0].pColorAttachments = colorReference;
vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pDepthStencilAttachment = &depthReference;
vsd[0].preserveAttachmentCount = 0;
vsd[0].pPreserveAttachments = (uint32_t *) nullptr;
vsd[1].flags = 0;
vsd[1].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[1].inputAttachmentCount = 1;
vsd[1].pInputAttachments = colorReference;
vsd[1].colorAttachmentCount = 1;
vsd[1].pColorAttachments = &outputReference;
vsd[1].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[1].pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
vsd[1].preserveAttachmentCount = 0;
vsd[1].pPreserveAttachments = (uint32_t *) nullptr;
```

Multipass, IV

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

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

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

```c
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_OUTPUT_BIT;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vimb.subresourceRange = visr;
vkCmdPipelineBarrier(TextureCommandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
    0, (VkMemoryBarrier *)nullptr,
    0, (VkBufferMemoryBarrier *)nullptr,
    1, IN &vimb);
```

Multipass, V

```c
vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpb, IN VK_SUBPASS_CONTENTS_INLINE );

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

```c
vkCmdNextSubpass( CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE );

// second subpass is started here – doesn’t need any new drawing vkCmd’s

```

```c
vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
```
Creating a Pipeline with Dynamically Changeable State Variables

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

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

This is useful for managing state information that needs to change frequently. This also creates possible optimization opportunities for the Vulkan driver.
Which Pipeline State Variables can be Changed Dynamically

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

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

Creating a Pipeline

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

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

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

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

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

Array naming the states that can be set dynamically

Filling State Variables in the Command Buffer

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

- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);`
- `vkCmdSetLineWidth(commandBuffer, lineWidth);`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference);`
There are 3 types of Queries: **Occlusion, Pipeline Statistics, and Timestamp**

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

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

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

Resetting, Filling, and Examining a Query Pool

```
Occlusion Queries count the number of fragments drawn between the vkCmdBeginQuery and the vkCmdEndQuery that pass both the Depth and Stencil tests.

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

Some hints:

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

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

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

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

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

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

- 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);

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

**Timestamp Query**

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

Even though the stages are “bits”, you are supposed to only specify one of them.

```c
vkCmdWriteTimeStamp( CommandBuffer, pipelineStages, timestampQueryPool, 0 );
```
An Introduction to the Vulkan Computer Graphics API

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

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