Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

SIGGRAPH 2020 Abridged Version

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

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

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

Mike Bailey

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

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

http://cs.oregonstate.edu/~mjb/vulkan
Section titles that have been greyed-out have not been included in the ABRIDGED noteset, i.e., the one that has been made to fit in SIGGRAPH's reduced time slot. These topics are in the FULL noteset, however, which can be found on the web page: http://cs.oregonstate.edu/~mjb/vulkan

My Favorite Vulkan Reference

Introduction

Mike Bailey
mjb@cs.oregonstate.edu

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

Everything You Need to Know is Right Here ... Somewhere 😊
Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

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

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

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

• More low-level information must be provided (by you!) in the application, rather than the driver
• Screen coordinate system is Y-down
• No “current state”, at least not one maintained by the driver

• All of the things that we have talked about being deprecated in OpenGL are really deprecated in Vulkan: built-in pipeline transformations, begin-vertex-end, fixed-function, etc.
• You must manage your own transformations.
• All transformation, color and texture functionality must be done in shaders.
• Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the runtime pipeline-building process.
• In OpenGL, your “pipeline state” is the combination of whatever your current graphics attributes are: color, transformations, textures, shaders, etc.

• Changing the state on-the-fly one item at-a-time is very expensive

• Vulkan forces you to set all your state variables at once into a “pipeline state object” (PSO) data structure and then invoke the entire PSO at once whenever you want to use that state combination

• Think of the pipeline state as being immutable.

• Potentially, you could have thousands of these pre-prepared pipeline state objects
Querying the Number of Something

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

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

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

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

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

VkBufferCreateInfo
vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;
VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );

VkMemoryRequirements

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

VkMemoryAllocateInfo

vkAllocateMemory( LogicalDevice, IN &vbci, PALLOCATOR, OUT &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
Vulkan Quick Reference Card – I Recommend you Print This!


Vulkan Quick Reference Card

Vulkan Highlights: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Queue

Queue

Queue

Queue

Queue

Queue

Queue

Queue

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Vulkan Highlights: a More Typical Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Queue

Queue

Queue

Command Buffer

Command Buffer

Command Buffer
Steps in Creating Graphics using Vulkan

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

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjb@cs.oregonstate.edu

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

Sample Program Keyboard Inputs

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

'i', 'I': Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)
'1', '4', '9': Set the number of instances (in the instancing version)
Caveats on the Sample Code, I

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

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

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

4. I’ve setup Vulkan structs every time they are used, even though, in many cases (most?), they could have been setup once and then re-used each time.

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

Caveats on the Sample Code, II

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

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

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

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

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

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

Vulkan Conventions

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

**vkYyy( )** is a function call

**VK_ZZZ** is a constant

My Conventions

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

The number after “Init” gives you the ordering

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

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

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

“IN” and “OUT” ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are actually #define’d to nothing.
Your Sample2019.zip File Contains This

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

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

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

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

bool Paused;
bool Verbose;

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

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST**
  - Points: $V_0, V_1, V_2, V_3$

- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST**
  - Lines: $V_0V_1, V_1V_2, V_2V_3$

- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**
  - Strips: $V_0V_1V_2V_3$

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**
  - Triangles: $V_0V_1V_2, V_1V_2V_3$

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**
  - Strips: $V_0V_1V_2V_3V_4$

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**
  - Fans: $V_0V_1V_2V_3V_4$
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;

static GLuint CubeTriangleIndices[3][3] = {
    {0, 2, 3},
    {0, 3, 1},
    {1, 2, 3},
    {0, 1, 3},
    {1, 0, 2},
    {1, 2, 0},
    {0, 1, 2},
    {0, 0, 1}
};
Triangles Represented as an Array of Structures

From the file SampleVertexData.cpp:

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

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

Modeled in right-handed coordinates

From the file SampleVertexData.cpp:

Non-indexed Buffer Drawing

Stream of Vertices

```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. }
};
```
struct vertex VertexData[ ] =
{
    ...
};

MyBuffer MyVertexDataBuffer;

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

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

A Preview of What Init05DataBuffer Does

VkResult Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer)
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *) nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr ); // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 ); // 0 is the offset
    return result;
}
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;
```

Always use the C/C++ construct `offsetof`, rather than hardcoding the value!
We will come to the Pipeline later, but for now, know that a Vulkan Pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its vertex input.

```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;

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

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

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
                                    PALLOCATOR, OUT &GraphicsPipeline);
```
We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

```cpp
VkBuffer buffers[1] = MyVertexDataBuffer.buffer;
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );
const uint32_t vertexCount = sizeof( VertexData ) / sizeof( VertexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

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

```cpp
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 );

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

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

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

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

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

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

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

**Sometimes the Same Point Needs Multiple Attributes**

Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

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

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

Where values match at the corners (color)

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

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

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

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

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

**Vulkan Vertex and Instance indices:**

- gl_VertexIndex
- gl_InstanceIndex

**OpenGL uses:**

- gl_VertexID
- gl_InstanceID

- Both are 0-based

**gl_FragColor:**

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

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

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

**Descriptor Sets:**

```glsl
layout( set=0, binding=0 ) . . . ;
```

**Push Constants:**

```glsl
layout( push_constant ) . . . ;
```

**Specialization Constants:**

```glsl
layout( constant_id = 3 ) const int N = 5;
```

- Only for scalars, but a vector’s components can be constructed from specialization constants

**Specialization Constants for Compute Shaders:**

```glsl
layout( local_size_x_id = 8, local_size_y_id = 16 );
```

- This sets gl_WorkGroupSize.x and gl_WorkGroupSize.y
- gl_WorkGroupSize.z is set as a constant

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

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

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

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

Vulkan Shader Compiling

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

<table>
<thead>
<tr>
<th>GLSL Source</th>
<th>External GLSL Compiler</th>
<th>SPIR-V</th>
<th>Compiler in driver</th>
<th>Vendor-specific code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develop Time</td>
<td>Run Time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```
```

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

(Can be overridden by the –S option)

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

Windows: glslangValidator.exe
Linux: glslangValidator

---

**Running glslangValidator.exe**

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

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

Specify the output file

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

```
MINGW64:/y/Vulkan/Sample2017

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

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

ONID+mjb@pooh MINGW64 /y/Vulkan/Sample2017
$ 
```

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 ... 
```
#define SPIRV_MAGIC 0x07230203


VkResult
Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb");
    if( fp == NULL )
    {
        fprintf( FpDebug, "Cannot open shader file \"%s\n", filename.c_str( ) );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
    {
        fprintf( FpDebug, "Magic number for spir-v file \"%s\n\" 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 );

Reading a SPIR-V File into a Vulkan Shader Module

VkShaderModule ShaderModuleVertex;

...
**Vulkan: Creating a Pipeline**

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:

```cpp
#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:
This is the SPIR-V Assembly, Part II

```spirvasm
declare 15(matrices) binding 0
member declare 32(gl_pervertex) 0 built-in position
member declare 32(gl_pervertex) 1 built-in point size
member declare 32(gl_pervertex) 2 built-in clip distance
declare 32(gl_pervertex) block

declare 32(gl_pervertex) location 0

declare 48(vnormal) location 0

declare 53(a_normal) location 1


declare 56(v_color) location 1

declare 57(a_color) location 2


declare 61(v_texture_coord) location 2

declare 63(a_texture_coord) location 3


declare 65(lightbuf) 0 offset 0

declare 65(lightbuf) block

declare 67(light) descriptor set 1

declare 67(light) binding 0

2: type void
3: type function 2
6: type float 32
7: type vector 6(float) 4
8: type matrix 7(fvec4) 4
9: type pointer function 8
11: type vector 6(float) 3
12: type matrix 11(fvec3) 3

(matrices): type struct 8 8 8 12

[]

14: type pointer uniform 13(matrices)
15(matrices): 14(ptr) variable uniform
16: type int 32 1
17: 16(int) constant 2
18: type pointer uniform 8
19: 16(int) constant 1
20: 16(int) constant 0
21: 20(int) constant 1
22: 21(int) constant 0
23: 22(int) constant 1
24: 23(int) constant 0
25: 24(int) constant 1
26: 25(int) constant 0
27: 26(int) constant 1
28: 27(int) constant 0
29: 28(int) constant 1
30: 29(int) constant 0
31: 30(int) constant 1
32(gl_pervertex): type struct 7(fvec4) 6(float)
33: type pointer output 32(gl_pervertex)
34: 33(ptr) variable output
36: type pointer uniform 12
37(a_vertex): 36(ptr) variable input
39: 6(float) constant 1065353216
40: 39(ptr) variable output
44: 7(fvec4) matrix times vector
46: 45(ptr) access chain 34 25
47: 46(ptr) variable output
48(v_normal): 47(ptr) variable output
49: 48(v_normal) variable output
50: type pointer uniform 12
51: 50(ptr) variable input
52: 51(ptr) variable output
53(a_color): 52(ptr) variable input
54: 53(a_color) variable output
55: 54(ptr) variable input
56(v_color): 55(ptr) variable output
57(a_texture_coord): 56(v_color) pointer input
58: 57(a_texture_coord) variable input
60: type pointer uniform 8
61(a_vertex): 60(ptr) variable input
62: 61(a_vertex) pointer input
63(a_normal): 62(a_vertex) pointer input
64: 63(a_normal) variable input
65(lightbuf): type struct 7(fvec4)
66: type pointer uniform 65(lightbuf)
67(light): 66(ptr) variable uniform
40(main): 2 function none 3
5: label

10(pvm): type int 32
19: 10(pvm) access chain 15(matrices) 17
20: 19(ptr) variable input
22: 18(ptr) access chain 15(matrices) 21
23: 22(ptr) variable input
24: 23(ptr) variable input
25: 24(ptr) variable input
26: 25(ptr) variable input
27: 26(ptr) variable input
28: 27(ptr) variable input
30: 28(ptr) variable input
32: 30(ptr) variable input
33: 32(ptr) variable input
34: 33(ptr) variable input
35: 34(ptr) variable input
36: 35(ptr) variable input
37: 36(ptr) variable input
38: 37(ptr) variable input
39: 38(ptr) variable input
40: 39(ptr) variable input
41: 40(ptr) variable input
42: 41(ptr) variable input
43: 42(ptr) variable input
44: 43(ptr) variable input
45: 44(ptr) variable input
46: 45(ptr) variable input
47: 46(ptr) variable input
48: 47(ptr) variable input
49: 48(ptr) variable input
50: 49(ptr) variable input
51: 50(ptr) variable input
52: 51(ptr) variable input
53: 52(ptr) variable input
54: 53(ptr) variable input
55: 54(ptr) variable input
56: 55(ptr) variable input
57: 56(ptr) variable input
58: 57(ptr) variable input
59: 58(ptr) variable input
60: 59(ptr) variable input
61: 60(ptr) variable input
62: 61(ptr) variable input
63: 62(ptr) variable input
64: 63(ptr) variable input
65: 64(ptr) variable input
66: 65(ptr) variable input
67: 66(ptr) variable input
40(main): 2 function none 3
5: label

10(pvm): type int 32
19: 10(pvm) access chain 15(matrices) 17
20: 19(ptr) variable input
22: 18(ptr) access chain 15(matrices) 21
23: 22(ptr) variable input
24: 23(ptr) variable input
25: 24(ptr) variable input
26: 25(ptr) variable input
27: 26(ptr) variable input
28: 27(ptr) variable input
30: 28(ptr) variable input
32: 30(ptr) variable input
33: 32(ptr) variable input
34: 33(ptr) variable input
35: 34(ptr) variable input
36: 35(ptr) variable input
37: 36(ptr) variable input
38: 37(ptr) variable input
39: 38(ptr) variable input
40: 39(ptr) variable input
41: 40(ptr) variable input
42: 41(ptr) variable input
43: 42(ptr) variable input
44: 43(ptr) variable input
45: 44(ptr) variable input
46: 45(ptr) variable input
47: 46(ptr) variable input
48: 47(ptr) variable input
49: 48(ptr) variable input
50: 49(ptr) variable input
51: 50(ptr) variable input
52: 51(ptr) variable input
53: 52(ptr) variable input
54: 53(ptr) variable input
55: 54(ptr) variable input
56: 55(ptr) variable input
57: 56(ptr) variable input
58: 57(ptr) variable input
59: 58(ptr) variable input
60: 59(ptr) variable input
61: 60(ptr) variable input
62: 61(ptr) variable input
63: 62(ptr) variable input
64: 63(ptr) variable input
65: 64(ptr) variable input
66: 65(ptr) variable input
67: 66(ptr) variable input
40(main): 2 function none 3
5: label
```

This is the SPIR-V Assembly, Part III
A Google-Wrapped Version of glslangValidator

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

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

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

1. You can #include files into your shader source

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

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

glslc is included in your Sample .zip file

---

Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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

```c
typedef VkBuffer VkDataBuffer;
```

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

**vkCreateBuffer( )**

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

**vkGetBufferMemoryRequirements( )**

- `VkMemoryAllocateInfo` size, memoryType
- `LogicalDevice`

**vkAllocateMemory( )**

- `LogicalDevice`

**vkBindBufferMemory( )**

- `bufferMemoryHandle`

**vkMapMemory( )**

- `gpuAddress`

---

Creating a Vulkan Data Buffer

```c
VkBuffer Buffer;
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << 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 int32_t) nullptr;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```
Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

. . .

VkDeviceMemory vdm;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset
. . .
result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );
<< do the memory copy >>
result = vkUnmapMemory( LogicalDevice, IN vdm );
```

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;
}
```
```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:
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 = 0x8c7c00000 DeviceLocal
Heap 1: size = 0xfac00000
Sidebar: The Vulkan Memory Allocator (VMA)

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

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

This repository includes a smattering of documentation.

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

VkBufferCreateInfo vbci;
... VmaAllocationCreateInfo vaci;
... vaci.physicalDevice = PhysicalDevice;
... vaci.device = LogicalDevice;
... vaci.usage = VMA_MEMORY_USAGE_GPU_ONLY;

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

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

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

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

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

MyBuffer MyMatrixUniformBuffer;
```

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

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

---

Initializing a Data Buffer

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

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

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

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

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

Filling those Uniform Variables

```c
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle in radians
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( 1. );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan's LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

This code assumes that this line:

```c
#define    GLM_FORCE_RADIANS
```

is listed before GLM is included!
The Parade of Buffer Data

MyBuffer  MyMatrixUniformBuffer;

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

This C struct is holding the original data, written by the application.

struct matBuf
{
    glm::vec3 eye(0, 0, EYEDIST);
    glm::vec3 lookAt(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.uNormalMatrix = glm::inverse(glm::transpose( glm::mat3( Matrices.uModelMatrix ) ));
};

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

uniform matBuf  Matrices;

Filling the Data Buffer

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

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

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

Matrices.uModelMatrix = glm::mat4( );  // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uNormalMatrix = glm::inverse(glm::transpose( glm::mat3( Matrices.uModelMatrix ) ));
```
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );         // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```

```
VkResult Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to Init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
    memcp( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}
```

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

```c
#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"

uint32_t Width, Height;
VkSurfaceKHR Surface;

void InitGLFW() {
  glfwInit();
  if( !glfwVulkanSupported() ) {
    fprintf( stderr, "Vulkan is not supported on this system!
" );
    exit( 1 );
  }
  glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );
  glfwWindowHint( GLFW_RESIZABLE, GLFW_FALSE );
  MainWindow = glfwCreateWindow( Width, Height, "Vulkan Sample", NULL, NULL );
  VkResult result = glfwCreateWindowSurface( Instance, MainWindow, NULL, OUT &Surface );
  glfwSetErrorCallback(GLFWErrorCallback);
  glfwSetKeyCallback( MainWindow, GLFWKeyboard);
  glfwSetCursorPosCallback( MainWindow, GLFWMouseMotion);
  glfwSetMouseButtonCallback( MainWindow, GLFWMouseButton);
}
```
You Can Also Query What Vulkan Extensions GLFW Requires

```c
uint32_t count;
const char ** extensions = glfwGetRequiredInstanceExtensions (&count);
fprintf( FpDebug, "Found %d GLFW Required Instance Extensions:\n", count );
for( uint32_t i = 0; i < count; i++ )
{
    fprintf( FpDebug, "t%s\n", extensions[ i ] );
}
```

Found 2 GLFW Required Instance Extensions:
- VK_KHR_surface
- VK_KHR_win32_surface

GLFW Keyboard Callback

```c
void GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
    if( action == GLFW_PRESS )
    {
        switch( key )
        {
            //case GLFW_KEY_M:
            case 'm':
            case 'M':
                Mode++;
                if( Mode >= 2 )
                    Mode = 0;
                break;

            default:
                fprintf( FpDebug, "Unknown key hit: 0x%x = '%c'\n", key, key );
                fflush(FpDebug);
                break;
        }
    }
}
```
GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

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

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

    if( ( ActiveButton & MIDDLE ) != 0 )
    {
        Scale += SCLFACT * (float) ( dx - dy );
        // keep object from turning inside-out or disappearing:
        if( Scale < MINSCALE )
        {
            Scale = MINSCALE;
        }
        Xmouse = (int)xpos;                     // new current position
        Ymouse = (int)ypos;
    }
```
Looping and Closing GLFW

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

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

Does not block – processes any waiting events, then returns

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

```cpp
glfwWaitEvents( );
```

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

```cpp
glfwWaitEventsTimeout( double secs );
```

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

```cpp
glfwPostEmptyEvent( );
```
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.

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/

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

You invoke GLM like this:

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

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

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

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

You would now say:

```cpp
glm::mat4 modelview = glm::mat4( 1. ); // identity
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
modelview = glm::lookAt( eye, look, up ); // \{x',y',z'} = \{v\}*[x,y,z]
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) ); // \{x',y',z'} = \{v\}*[yr]*[x,y,z]
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) ); // \{x',y',z'} = \{v\}*[yr][xr]*[x,y,z]
modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) ); // \{x',y',z'} = \{v\}*[yr]*[sr]*[x,y,z]
```

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

---

The Most Useful GLM Variables, Operations, and Functions

---

// constructor:

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

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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

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

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

```cpp
glm::mat4 glm::perspective( float fovy, float aspect, float near, float far);
```

// viewing (assign, not concatenate):

```cpp
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );
```

---

**Installing GLM into your own space**

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

GLM in the Vulkan sample.cpp Program

```cpp
if( UseMouse )
{
    if( Scale < MINSCALE )
    {
        Scale = MINSCALE;
        Matrices.uModelMatrix = glm::mat4( 1. );           // identity
        Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0.) );
        Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0.) );
        Matrices.uModelMatrix = glm::scale(  Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
        // done this way, the Scale is applied first, then the Xrot, then the Yrot
    }
    else
    {
        if( ! Paused )
        {
            const glm::vec3 axis = glm::vec3( 0., 1., 0. );
            Matrices.uModelMatrix = glm::rotate( glm::mat4( 1. ), (float)glm::radians( 360.f*Time/SECONDS_PER_CYCLE ), axis );
        }
    }
}

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

Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1f, 1000.f );
Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL

Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix );   // note: inverseTransform !

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

Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
```
Sidebar: Why Isn't The Normal Matrix exactly the same as the Model Matrix?

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

\[
glm::mat3 \text{ NormalMatrix} = \text{glm::mat3(Model)};\]

Wrong!

Original object and normal

\[
glm::mat3 \text{ NormalMatrix} = \text{glm::inverseTranspose(glm::mat3(Model))};\]

Right!

Instancing

Mike Bailey

mjb@cs.oregonstate.edu

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

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

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

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

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

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:

```c
out vec3 vColor;
const int NUMINSTANCES = 16;
const float DELTA = 3.0;

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

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

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

Still uses `gl_InstanceIndex`

In the vertex shader:
```
layout( std140, set = 3, binding = 0 ) uniform colorBuf
{
   vec3  uColors[1024];
} Colors;
out vec3 vColor;
...
int index = gl_InstanceIndex % 1024;  // or "& 1023" – gives 0 - 1023
vColor = Colors.uColors[ index ];
vec4 vertex = . . .
gl_Position = PVM * vertex;  // [p][v][m]```
What is the Vulkan Graphics Pipeline?

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

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

3. For the most part, the Vulkan Graphics Pipeline Data Structure is immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.

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

Don’t worry if this is too small to read – a larger version is coming up.

There is also a Vulkan **Compute Pipeline Data Structure**.
The First Step: Create the Graphics Pipeline Layout

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

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

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

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

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

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

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

```c
VkResult
Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader, VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
    #ifdef ASSUMPTIONS
        vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
        vprsci.depthClampEnable = VK_FALSE;
        vprsci.rasterizerDiscardEnable = VK_FALSE;
        vprsci.polygonMode = VK_POLYGON_MODE_FILL;
        vprsci.cullMode = VK_CULL_MODE_NONE; // best to do this because of the projectionMatrix[1][1] *= -1.;
        vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
        vprsci.rasterizationSamples = VK_SAMPLE_COUNT_ONE_BIT;
        vpbas.blendEnable = VK_FALSE;
        vpbcsci.logicOpEnable = VK_FALSE;
        vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_ONE_BIT;
        #endif
    
    // ..

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

The Shaders to Use

```c
 use one vpssci array member per shader module you are using

 use one vvibd array member per vertex input array-of-structures you are using
```
Link in the Per-Vertex Attributes

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

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

Declare the binding descriptions and attribute descriptions.

Declare the vertex topology.

Declare Tessellation Shader info.

Declare Geometry Shader info.
Options for vpiasci.topology

VK_PRIMITIVE_TOPOLOGY_POINT_LIST

VK_PRIMITIVE_TOPOLOGY_LINE_LIST

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

What is “Primitive Restart Enable”?

vpiasci.primitiveRestartEnable = VK_FALSE;

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

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

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

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 0xffff.
If your VkIndexType is VK_INDEX_TYPE_UINT32, it is 0xffffffff.
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

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

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

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

```
VkPipelineRasterizationStateCreateInfo vprsci;
    vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    vprsci.pNext = nullptr;
    vprsci.flags = 0;
    vprsci.depthClampEnable = VK_FALSE;
    vprsci.rasterizerDiscardEnable = VK_FALSE;
    vprsci.polygonMode = VK_POLYGON_MODE_FILL;
    vprsci.cullMode = VK_CULL_MODE_NONE;   // recommend this because of the projMatrix[1][1] *= -1.;
    vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    vprsci.depthBiasEnable = VK_FALSE;
    vprsci.depthBiasConstantFactor = 0.f;
    vprsci.depthBiasClamp = 0.f;
    vprsci.depthBiasSlopeFactor = 0.f;
    vprsci.lineWidth = 1.f;
    // Declare information about how the rasterization will take place
```

Setting the Rasterizer State
**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
- The gray area shows what would happen with depthClampEnable (except it would have been red).

---

**What is “Depth Bias Enable”?**

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

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

We will discuss MultiSampling in a separate noteset.

VkPipelineColorBlendAttachmentState
vpcbas = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ATTACHMENT_CREATE_INFO;

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

Color Blending State for each Color Attachment *

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

Color_{new} = (1.-\alpha) \times Color_{existing} + \alpha \times Color_{incoming}

0 \leq \alpha \leq 1.

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

```c
VkPipelineColorBlendStateCreateInfo vpcbsci = {  
    .sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO,  
    .pNext = nullptr,  
    .flags = 0,  
    .logicOpEnable = VK_FALSE,  
    .logicOp = VK_LOGIC_OP_COPY,  
    .attachmentCount = 1,  
    .pAttachments = &vpcbas,  
    .blendConstants[0] = 0,  
    .blendConstants[1] = 0,  
    .blendConstants[2] = 0,  
    .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**

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

```c
#ifdef CHOICES  
VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport( )  
VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor( )  
VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth( )  
VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias( )  
VK_DYNAMIC_STATE_BLEND_CONSTANTS        -- vkCmdSetBendConstants( )  
VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthZBounds( )  
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompareMask( )  
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK -- vkCmdSetStencilWriteMask( )  
VK_DYNAMIC_STATE_STENCIL_REFERENCE -- vkCmdSetStencilReferencce( )  
#endif
```

```c
VkPipelineDynamicStateCreateInfo vpdsici = {  
    .sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO,  
    .pNext = nullptr,  
    .flags = 0,  
    .dynamicStateCount = 0;  
    .pDynamicStates = vds;  
    .dynamicStateCount = 0;  // leave turned off for now
};
```
The Stencil Buffer

Here's how the Stencil Buffer works:

1. While drawing into the Render Buffer, you can write values into the Stencil Buffer at the same time.

2. While drawing into the Render Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.

3. When drawing into the Render Buffer, you can write-protect certain parts of the Render Buffer based on values that are in the Stencil Buffer.

Using the Stencil Buffer to Create a Magic Lens
Using the Stencil Buffer to Create a Magic Lens

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

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting
Using the Stencil Buffer to Better Outline Polygons

Clear the SB = 0
for( each polygon )
{
    Draw the edges, setting SB = 1
    Draw the polygon wherever SB != 1
    Draw the edges, setting SB = 0
}
Using the Stencil Buffer to Perform Hidden Line Removal

```c
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
#endif
    vsosf.compareOp = VK_COMPARE_OP_NEVER;
#ifdef CHOICES
    VK_STENCIL_OP_KEEP -- keep the stencil value as it is
    VK_STENCIL_OP_ZERO -- set stencil value to 0
    VK_STENCIL_OP_REPLACE -- replace stencil value with the reference value
    VK_STENCIL_OP_INCREMENT_AND_CLAMP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_CLAMP -- decrement stencil value
    VK_STENCIL_OP_INVERT -- bit-invert stencil value
    VK_STENCIL_OP_INCREMENT_AND_WRAP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_WRAP -- decrement stencil value
#endif
    vsosf.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;

Stencil Operations for Front and Back Faces
Operations for Depth Values

```
 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;
 vpdssci.depthBoundsTestEnable = VK_FALSE;
 vpdssci.front = vsosf;
 vpdssci.back = vsosb;
 vpdssci.minDepthBounds = 0.0f;
 vpdssci.maxDepthBounds = 1.0f;
 vpdssci.stencilTestEnable = VK_FALSE;
``` 

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 = 0.0f;
vpdssci.maxDepthBounds = 1.0f;
vpdssci.stencilTestEnable = VK_FALSE;
``` 

Putting it all Together! (finally...)

```
VkPipeline GraphicsPipeline;

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

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT &GraphicsPipeline );
return result;
```
Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

\[\text{vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );}\]

Sidebar: What is the Organization of the Pipeline Data Structure?

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

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

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

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

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

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

```glsl
layout( std140, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, binding = 1 ) uniform mat4 uViewMatrix;
layout( std140, binding = 2 ) uniform mat4 uProjectionMatrix;
layout( std140, binding = 3 ) uniform mat3 uNormalMatrix;
layout( std140, binding = 4 ) uniform vec4 uLightPos;
layout( std140, binding = 5 ) uniform float uTime;
layout( std140, binding = 6 ) uniform int uMode;
layout( binding = 7 ) uniform sampler2D uSampler;
```
Descriptor Sets are an intermediate data structure that tells shaders how to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders. There are three advantages in doing things this way:

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

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

What are Descriptor Sets?

Our example will assume the following shader uniform variables:

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

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

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

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

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

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

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

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

struct lightBuf
{
  glm::vec4 uLightPos;
};

struct miscBuf
{
  float uTime;
  int uMode;
};

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

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

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

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

Step 1: Descriptor Set Pools

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

```
 DescriptorSetPool
```

```
 device

 VkCreateDescriptorPool( )

 VkDescriptorPoolCreateInfo

 flags

 maxSets

 poolSizeCount

 poolSizes

```

* “binary large object”*
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.
VkResult Init13DescriptorSetLayouts()
{
  VkResult result;

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

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

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

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

  uniform sampler2D uSampler;
  vec4 rgba = texture( uSampler, vST );

---

**Step 2: Define the Descriptor Set Layouts**

- MatrixSet DS Layout Binding
- LightSet DS Layout Binding
- MiscSet DS Layout Binding
- TexSamplerSet DS Layout Binding

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

**VkResult**
**Init14GraphicsPipelineLayout()**
{
    
    **VkPipelineLayoutCreateInfo**
    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

```c
VkResult Init13DescriptorSets()
{
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;

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

This struct identifies what buffer it owns and how big it is

```
VkDescriptorBufferInfo
vdbi0;

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

This struct identifies what buffer it owns and how big it is

```
VkDescriptorBufferInfo
vdbi1;

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

This struct identifies what buffer it owns and how big it is

```
VkDescriptorBufferInfo
vdbi2;

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

This struct identifies what texture sampler and image view it owns

```
VkDescriptorImageInfo
vdii0;

vdii.sampler = MyPuppyTexture.texSampler;
vdzi.imageView = MyPuppyTexture.texImageView;
vdzi.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```

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

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

```
VkWriteDescriptorSet
vwds0;

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

// ds 1:
VkWriteDescriptorSet
vwds1;

vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds1.pNext = nullptr;
vwds1.dstSet = DescriptorSets[1];
vwds1.dstBinding = 0;
vwds1.dstArrayElement = 0;
vwds1.descriptorCount = 1;
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.pBufferInfo = &vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
```
Step 5: Tell the Descriptor Sets where their data is

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

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

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

```
VkWriteDescriptorSet vwds3;
// ds 3:
vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds3.pNext = nullptr;
vwds3.dstSet = DescriptorSets[3];
vwds3.dstBinding = 0;
vwds3.dstArrayElement = 0;
vwds3.descriptorCount = 1;
vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
vwds3.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
vwds3.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds3.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
vgpci.stageCount = 2;
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
```

```
#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif
```

```
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
```

```
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 );
```

So, the Pipeline Layout contains the **structure** of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.

Sidebar: The Entire Collection of Descriptor Set Paths

- **vkCreateDescriptorPool( )**
  - Create the pool of Descriptor Sets for future use
- **vkCreateDescriptorSetLayout( )**
  - Describe a particular Descriptor Set layout and use it in a specific Pipeline layout
- **vkCreatePipelineLayout( )**
- **vkAllocateDescriptorSets( )**
  - Allocate memory for particular Descriptor Sets
- **VkWriteDescriptorSet**
  - Tell a particular Descriptor Set where its CPU data is
- **vkUpdateDescriptorSets( )**
  - Re-write CPU data into a particular Descriptor Set
- **vkCmdBindDescriptorSets( )**
  - Make a particular Descriptor Set "current" for rendering
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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

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

The Pipeline Data Structure

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

Mike Bailey
mjb@cs.oregonstate.edu

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

Triangles in an Array of Structures

```c
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.  }},
};
```
Memory Types

**NVIDIA Discrete Graphics:**

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

**Intel Integrated Graphics:**

3 Memory Types:
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal HostVisible HostCoherent
- Memory 2: DeviceLocal HostVisible HostCoherent HostCached
Texture Sampling Parameters

```gl
// OpenGL
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

```vulkan
// Vulkan
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, &vsci, PALLOCATOR, pTextureSampler);
```

Textures' Undersampling Artifacts

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

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

Average 4 pixels to make a new one

Average 4 pixels to make a new one

Average 4 pixels to make a new one

• Total texture storage is ~ 2x what it was without mip-mapping

• Graphics hardware determines which level to use based on the texels : pixels ratio.

• In addition to just picking one mip-map level, the rendering system can sample from two of them, one less that the T:P ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin: multum 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.anisotropy = 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;

vsci.unnormalizedCoordinates = VK_FALSE; // VK_TRUE means we are using raw texels as the index
// VK_FALSE means we are using the usual 0. - 1.

result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler );
VkResult
Init07TextureBuffer( INOUT MyTexture * pMyTexture)
{
VkResult result = VK_SUCCESS;
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;
#endif
vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
#if define CHOICES
VK_IMAGE_USAGE_TRANSFER_SRC_BIT
VK_IMAGE_USAGE_TRANSFER_DST_BIT
VK_IMAGE_USAGE_SAMPLED_BIT
VK_IMAGE_USAGE_STORAGE_BIT
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
#endif
vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
#endif
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
", vmr.size);
fprintf(FpDebug, "Image vmr.alignment = %lld
", vmr.alignment);
fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
fflush(FpDebug);
}

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible(); // because we want to mmap it
VkDeviceMemory vdm;
vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
pMyTexture->vdm = vdm;

vkBindImageMemory(LogicalDevice, IN stagingImage, IN &vdm, 0); // 0 = offset
// we have now created the staging image -- fill it with the pixel data:

// this second {...} is to create texture image:
{
VkImageCreateInfo visi;
visi.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
visi.pNext = nullptr;
visi.flags = 0;
visi.imageType = VK_IMAGE_TYPE_2D;
visi.format = VK_FORMAT_R8G8B8A8_UNORM;
visi.extent.width = texWidth;
visi.extent.height = texHeight;
visi.extent.depth = 1;
visi.mipLevels = 1;
visi.arrayLayers = 1;
visi.samples = VK_SAMPLE_COUNT_1_BIT;
visi.tiling = VK_IMAGE_TILING_LINEAR;
#endif
visi.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
#if define CHOICES
VK_IMAGE_USAGE_TRANSFER_SRC_BIT
VK_IMAGE_USAGE_TRANSFER_DST_BIT
VK_IMAGE_USAGE_SAMPLED_BIT
VK_IMAGE_USAGE_STORAGE_BIT
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
#endif
visi.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
#endif
vkCreateImage(LogicalDevice, IN &visi, PALLOCATOR, OUT &textureImage); // allocated, but not filled

VkMemoryRequirements vmm;
vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmm);
if (Verbose)
{
fprintf(FpDebug, "Texture vmr.size = %lld
", vmm.size);
fprintf(FpDebug, "Texture vmr.alignment = %lld
", vmm.alignment);
fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmm.memoryTypeBits);
fflush(FpDebug);
}

VkMemoryAllocateInfo vma;
vma.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vma.pNext = nullptr;
vma.allocationSize = vmm.size;
vma.memoryTypeIndex = FindMemoryThatIsHostVisible(); // because we want to mmap it
VkDeviceMemory vmt;
vkAllocateMemory(LogicalDevice, IN &vma, PALLOCATOR, OUT &vmt);
pMyTexture->vmt = vmt;

vkBindImageMemory(LogicalDevice, IN textureImage, IN &vmt, 0); // 0 = offset

if (Verbose)
{
fprintf(FpDebug, "SubresourceLayout:
" RED); // subresource layout
fprintf(FpDebug, "Format = %d
", textureFormat);
fprintf(FpDebug, "ViewFormat = %d
", textureViewFormat);
fprintf(FpDebug, "AspectMask = %d
", textureAspectMask);
fprintf(FpDebug, "MipLevels = %d
", textureMipLevels);
fprintf(FpDebug, "ArrayLayers = %d
", textureArrayLayers);
fprintf(FpDebug, "SampleCount = %d
", textureSampleCount);
fprintf(FpDebug, "Format = %d
", textureFormat);
fflush(FpDebug);
}

VkImageSubresource vis;
vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
vis.mipLevel = 0;
vis.arrayLayer = 0;
VkSubresourceLayout vsl;
vkGetImageSubresourceLayout(LogicalDevice, textureImage, IN &vis, OUT &vsl);
if (Verbose)
{
fprintf(FpDebug, "Subresource Layout:
" RED); // subresource layout
fprintf(FpDebug, "Format = %d
", textureFormat);
fprintf(FpDebug, "ViewFormat = %d
", textureViewFormat);
fprintf(FpDebug, "AspectMask = %d
", textureAspectMask);
fprintf(FpDebug, "MipLevels = %d
", textureMipLevels);
fprintf(FpDebug, "ArrayLayers = %d
", textureArrayLayers);
fprintf(FpDebug, "SampleCount = %d
", textureSampleCount);
fprintf(FpDebug, "Format = %d
", textureFormat);
fflush(FpDebug);
}
}
```c
void * gpuMemory;

vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
// 0 and 0 = offset and memory map flags
if (vsl.rowPitch == 4 * texWidth)
{
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
} else
{
    unsigned char *gpuBytes = (unsigned char *)gpuMemory;
    for (unsigned int y = 0; y < texHeight; y++)
    {
        memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
    }
}

vkUnmapMemory(LogicalDevice, vdm);

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

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

VkImageCreateInfo vici;
vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
vici.pNext = nullptr;
vici.flags = 0;
vici.imageType = VK_IMAGE_TYPE_2D;
vici.format = VK_FORMAT_R8G8B8A8_UNORM;
vici.extent.width = texWidth;
vici.extent.height = texHeight;
vici.extent.depth = 1;
vici.mipLevels = 1;
vici.arrayLayers = 1;
vici.samples = VK_SAMPLE_COUNT_1_BIT;
vici.tiling = VK_IMAGE_TILING_OPTIMAL;
vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
// because we are transferring into it and will eventually sample from it
vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

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

VkMemoryRequirements vmr;

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

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

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

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

// copy pixels from the staging image to the texture:
VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

// ******************************************************************************
// transition the staging buffer layout:
// ******************************************************************************
{
    VkImageSubresourceRange visr;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = stagingImage;
    vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = visr;
    vkCmdPipelineBarrier
    (TextureCommandBuffer,
     VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0,
     0, (VkMemoryBarrier *)nullptr,
     0, (VkBufferMemoryBarrier *)nullptr,
     1, IN &vimb);
}

// ******************************************************************************
// transition the texture buffer layout:
// ******************************************************************************
{
    VkImageSubresourceRange visr;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = 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 = VK_ACCESS_TRANSFER_WRITE_BIT;
    vimb.subresourceRange = visr;
    vkCmdPipelineBarrier
    (TextureCommandBuffer,
     VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
     0, (VkMemoryBarrier *)nullptr,
     0, (VkBufferMemoryBarrier *)nullptr,
     1, IN &vimb);

    // now do the final image transfer:
    VkImageSubresourceLayers visl;
    visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visl.baseArrayLayer = 0;
    visl.mipLevel = 0;
    visl.layerCount = 1;
    VkOffset3D vo3;
    vo3.x = 0;
    vo3.y = 0;
    vo3.z = 0;
    VkExtent3D ve3;
    ve3.width = texWidth;
    ve3.height = texHeight;
    ve3.depth = 1;
vkCmdCopyImageTextureCommandBuffer

stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);

result = vkEndCommandBuffer( TextureCommandBuffer );
VkSubmitInfo vsi;

vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &TextureCommandBuffer;
vsi.waitSemaphoreCount = 0;
vsi.pWaitSemaphores = (VkSemaphore *)nullptr;
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;

result = vkQueueSubmit( Queue, 1, IN &vsi, VK_NULL_HANDLE );
result = vkQueueWaitIdle( Queue );
// create an image view for the texture image:
// (an “image view” is used to indirectly access an image)

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

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

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

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

typedef struct MyTexture
{
  uint32_t width;
  uint32_t height;
  VkImage texImage;
  VkImageView texImageView;
  VkSampler texSampler;
  VkDeviceMemory vdm;
} MyTexture;

MyTexture MyPuppyTexture;

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

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

Mike Bailey
mjb@cs.oregonstate.edu

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

Simplified Block Diagram
Vulkan Queues and Command Buffers

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

```
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr);

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceFamilyProperties(PhysicalDevice, &count, OUT &vqfp,);

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

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

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

Creating a logical device needs to know queue family information:

```c
float queuePriorities[] = {1.0f};
VkDeviceQueueCreateInfo vdqci[1];
vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1;
vdqci[0].queuePriorities = (float *)queuePriorities;

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

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

```
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

```
VkResult Init06CommandBuffers( )
{
    VkResult result;
    // allocate 2 command buffers for the double-buffered rendering:
    VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 2;           // 2, because of double-buffering
    result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
}

// allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 1;
    result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );
}
return result;
}
vkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

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

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

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

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

vkEndCommandBuffer( CommandBuffers[nextImageIndex]);
These are the Commands that could be entered into the Command Buffer, I

- vkCmdBeginQuery(commandBuffer, flags);
- vkCmdBeginRenderPass(commandBuffer, const contents);
- vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
- vkCmdBindIndexBuffer(commandBuffer, indexType);
- vkCmdBindPipeline(commandBuffer, pipeline);
- vkCmdBindVertexBuffer(commandBuffer, firstBinding, bindingCount, const pOffsets);
- vkCmdBindPipelineExt(commandBuffer, const info);
- vkCmdBlurImage(commandBuffer, filter);
- vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
- vkCmdClearColorImage(commandBuffer, pRanges);
- vkCmdClearDepthStencilImage(commandBuffer, pRanges);
- vkCmdCopyBuffer(commandBuffer, pRegions);
- vkCmdCopyImage(commandBuffer, pRegions);
- vkCmdCopyImageToBuffer(commandBuffer, pRegions);
- 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);
- vkCmdEndQuery(commandBuffer, query);
- vkCmdEndRenderPass(commandBuffer);
- vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);
- vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
- vkCmdNextSubpass(commandBuffer, contents);
- vkCmdPipelineBarrier(srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, memoryBarrierInfo);
- vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);
- vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, size, pValues);
- vkCmdPushDescriptorSet(pPipelineBarrier);
- vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo);
- vkCmdResetEvent(commandBuffer, event, stageMask);
- vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
- vkCmdResolveImage(commandBuffer, dstImage, dstImageLayout, srcImage, srcImageLayout, regionCount, regionCount, regionCount, regionCount);
- vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
- vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);
- vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
- vkCmdSetDeviceMaskKHR(commandBuffer, deviceMask);
- vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);
- vkCmdSetEvent(commandBuffer, event, stageMask);
- vkCmdSetLineWidth(commandBuffer, line_width);
- vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);
- vkCmdSetStencilCompareMask(commandBuffer, compareMask);
- vkCmdSetStencilReference(commandBuffer, faceMask, reference);
- vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
- vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);
- vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewports);
- vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);
- vkCmdWriteTimestamp(commandBuffer, pipelineStage, stageMask, queryPool, query);

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

- vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
- vkCmdNextSubpass(commandBuffer, contents);
- vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarrierCount, pBufferMemoryBarrier, pMemoryBarrierCount, pImageMemoryBarrierCount, pImageMemoryBarrierCount);
- vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);
- vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, size, pValues);
- vkCmdPushDescriptorSet(pPipelineBarrier);
- vkCmdReserveSpaceForCommandsNVX(commandBuffer, pReserveSpaceInfo);
- vkCmdResetEvent(commandBuffer, event, stageMask);
- vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
- vkCmdResolveImage(commandBuffer, dstImage, dstImageLayout, srcImage, srcImageLayout, regionCount, regionCount, regionCount, regionCount);
- vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
- vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);
- vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
- vkCmdSetDeviceMaskKHR(commandBuffer, deviceMask);
- vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);
- vkCmdSetEvent(commandBuffer, event, stageMask);
- vkCmdSetLineWidth(commandBuffer, line_width);
- vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);
- vkCmdSetStencilCompareMask(commandBuffer, compareMask);
- vkCmdSetStencilReference(commandBuffer, faceMask, reference);
- vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
- vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);
- vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewports);
- vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);
- vkCmdWriteTimestamp(commandBuffer, pipelineStage, stageMask, queryPool, query);
VkResult RenderScene()
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, OUT &imageReadySemaphore);

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

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

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

    VkClearDepthStencilValue vcdsv;
    vcdsv.depth = 1.0f;
    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; // used for VK_ATTACHMENT_LOAD_OP_CLEAR
    vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
```cpp
VkViewport viewport = { 0., 0., (float)Width, (float)Height, 0., 1. }; // x, y, width, height, minDepth, 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 };
VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets); // 0, 1 = firstBinding, bindingCount

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

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

---

**Submitting a Command Buffer to a Queue for Execution**

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

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

VkFence renderFence;
    vkCreateFence(logicalDevice, &vfci, allocator, &renderFence);
    result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkQueue presentQueue;
    vkGetDeviceQueue(logicalDevice, FindQueueFamilyThatDoesGraphics(), 0, &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, &vsi, renderFence); // 1 = submitCount
result = vkWaitForFences(logicalDevice, 1, renderFence, VK_TRUE, UINT64_MAX); // waitAll, timeout
vkDestroyFence(logicalDevice, renderFence, allocator);

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

result = vkQueuePresentKHR(presentQueue, &vpi);

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

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

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

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

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

Mike Bailey
mjb@cs.oregonstate.edu

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

How OpenGL Thinks of Framebuffers
What is a Swap Chain?

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

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

Swap Chains are arranged as a ring buffer

Swap Chains are tightly coupled to the window system.

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

1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to “front buffer”)
We Need to Find Out What our Display Capabilities Are

VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );

VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "vkGetPhysicalDeviceSurfaceCapabilitiesKHR:\n" );

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

uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
fprintf( FpDebug, "Found %d Surface Formats:\n", formatCount );

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "Found %d Present Modes:\n", presentModeCount );

VulkanDebug.txt output:

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

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

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

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

- `vkGetDevicePhysicalSurfaceCapabilities()`
  - `VkSurfaceCapabilities`
    - `surface`
    - `imageFormat`
    - `imageColorSpace`
    - `imageExtent`
    - `imageArrayLayers`
    - `imageUsage`
    - `imageSharingMode`
    - `preTransform`
    - `compositeAlpha`
    - `presentMode`
    - `clipped`

- `vkCreateSwapchain()`
- `vkGetSwapChainImages()`
- `vkCreateImageView()`

```cpp
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr;
vscci.flags = 0;
vscci.surface = Surface;
vscci.minImageCount = 2;  // double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.imageArrayLayers = 1;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.queueFamilyIndexCount = 0;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
```
Creating the Swap Chain Images and Image Views

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

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

Rendering into the Swap Chain, I

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

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

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

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

vkCmdDraw( CommandBuffers[ nextImageIndex ], IN &vcbi, IN VK_SUBPASS_CONTENTS_INLINE );
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 );

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 );

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

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

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

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

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

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

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

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

where:

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

---

Creating a Pipeline

A pipeline is a series of steps that are needed to render a scene. You can consider them in a similar way to the states of a pipeline seen in the diagram, starting with vertex input, passing to a view port, building a render pass, then building a pipeline layout, and finally building the pipeline:

```cpp
vkCreateGraphicsPipeline( );
```
Setting up the Push Constants for the Pipeline Structure

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

```
VkPushConstantRange
vpcr[0].stageFlags = 
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

vpcr[0].size = sizeof(glm::mat4);

VkPipelineLayoutCreateInfo
vplci

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

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

An Robotic Example using Push Constants

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

Where each arm is represented by:

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

struct arm  Arm1;
struct arm  Arm2;
struct arm  Arm3;
```
Forward Kinematics:
You Start with Separate Pieces, all Defined in their Own Local Coordinate System

Hook the Pieces Together, Change Parameters, and Things Move
(All Young Children Understand This)
Forward Kinematics:
Given the Lengths and Angles, Where do the Pieces Move To?

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

Positioning Part #1 With Respect to Ground

$$[M_{1/G}] = [T_{1/G}] \ast [R_{\Theta_1}]$$
Positioning Part #2 With Respect to Ground

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

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

\[
\begin{bmatrix}
M_{2/G}
\end{bmatrix} = \begin{bmatrix}
M_{1/G}
M_{2/1}
\end{bmatrix}
\]

Positioning Part #3 With Respect to Ground

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

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

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

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

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

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

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

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

Setup the Push Constant for the Pipeline Structure

```cpp
VkPushConstantRange vpcr[1];

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

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

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

glm::vec3 zaxis = glm::vec3(0., 0., 1.);

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

glm::mat4 m21 = glm::mat4(1.); // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));

m21 = glm::rotate(m21, rot2, zaxis); // [T][R]

m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm

glm::mat4 m32 = glm::mat4(1.); // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));

m32 = glm::rotate(m32, rot3, zaxis); // [T][R]

m32 = glm::translate(m32, glm::vec3(0., 0., 2.)); // z-offset from previous arm

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

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

In the *RenderScene* Function

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

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1 );

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

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

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

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

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

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

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, &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 );
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
" );
    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
", PhysicalDeviceCount );
    return VK_SHOULD_EXIT;
}
Vulkan: Identifying the Physical Devices

int discreteSelect = -1;
int integratedSelect = -1;
for(unsigned int i = 0; i < PhysicalDeviceCount; i++)
{
    VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    if( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not get the physical device properties of device %d\n", i );
        return VK_SHOULD_EXIT;
    }
    fprintf( FpDebug, "\n
Device %2d:\n", i );
    fprintf( FpDebug, "\tAPI version: %d\n", vpdp.apiVersion );
    fprintf( FpDebug, "\tDriver version: %d\n", vpdp.apiVersion );
    fprintf( FpDebug, "\tDevice ID: 0x%04x\n", vpdp.deviceID );
    fprintf( FpDebug, "\tPhysical Device Type: %d =", vpdp.deviceType );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        fprintf( FpDebug, " (Discrete GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        fprintf( FpDebug, " (Integrated GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
        fprintf( FpDebug, " (Virtual GPU)\n" );
    if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
        fprintf( FpDebug, " (CPU)\n" );
    fprintf( FpDebug, "\tDevice Name: %s\n", vpdp.deviceName );
    fprintf( FpDebug, "\tPipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0] );
Which Physical Device to Use, I
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
    printf( FpDebug, "Could not select a Physical Device\n" );
    return VK_SHOULD_EXIT;

Asking About the Physical Device’s Features

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

printf( FpDebug, \"Physical Device Features:\n\n\"");
printf( FpDebug, "geometryShader = %2d\n", PhysicalDeviceFeatures.geometryShader);
printf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShader );
printf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect );
printf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines );
printf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints );
printf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport );
printf( FpDebug, "ocl\lusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.ocl\lusionQueryPrecise );
printf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStatisticsQuery );
printf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64 );
printf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64 );
printf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16 );
Here's What the NVIDIA RTX 2080 Ti Produced

vkEnumeratePhysicalDevices:

Device 0:
- API version: 4198499
- Driver version: 4198499
- Vendor ID: 0x10de
- Device ID: 0x1e04
- Physical Device Type: 2 = (Discrete GPU)
- Device Name: RTX 2080 Ti
- Pipeline Cache Size: 206

Device #0 selected ('RTX 2080 Ti')

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

Here's What the Intel HD Graphics 520 Produced

vkEnumeratePhysicalDevices:

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

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

Physical Device Features:
- geometryShader = 1
- tessellationShader = 1
- multiDrawIndirect = 1
- wideLines = 1
- largePoints = 1
- multiViewport = 1
- occlusionQueryPrecise = 1
- pipelineStatisticsQuery = 1
- shaderFloat64 = 1
- shaderInt64 = 1
- shaderInt16 = 1
Asking About the Physical Device's Different Memories

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

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

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

Here's What I Got

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

2 Memory Heaps:
Heap 0:  size = 0xb7c00000 DeviceLocal
Heap 1:  size = 0xfac00000
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
fprintf( FpDebug, "InFound %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, "%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");
}

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

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: a More Typical (and Simplified) Block Diagram
const char * myDeviceLayers[] =
{
    // "VK_LAYER_LUNARG_api_dump",
    // "VK_LAYER_LUNARG_core_validation",
    // "VK_LAYER_LUNARG_image",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    // "VK_LAYER_NV_optimus"
};

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

// 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, deviceLayers);

// see what device extensions are available:

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

VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];

result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, deviceLayers, deviceExtensions);
What Device Layers and Extensions are Available

4 physical device layers enumerated:

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

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

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

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

Vulkan: Creating a Logical Device

```c
float queuePriorities[1] = {
    1.0f,
};
VkDeviceQueueCreateInfo vdqci;
vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci.pNext = nullptr;
vdqci.flags = 0;
vdqci.queueFamilyIndex = 0;
vdqci.queueCount = 1;
vdqci.pQueueProperties = queuePriorities;

VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1;                    // # of device queues
vdqci.pQueueCreateInfos = IN vdqci;              // array of VkDeviceQueueCreateInfo's
vdqci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdqci.pEnabledLayerNames = myDeviceLayers;
vdqci.enabledExtensionCount = 0;
vdqci.ppEnabledExtensionNames = (const char **)nullptr; // no extensions
vdqci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdqci.pEnabledExtensionNames = myDeviceExtensions;
vdqci.pEnabledFeatures = IN &PhysicalDeviceFeatures;
result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
```
// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue ); // 0, 0 = queueFamilyIndex, queueIndex