Acknowledgements

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

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

Second, thanks to NVIDIA for all of their support!

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

2004: OpenGL 2.0 / GLSL 1.10 includes Vertex and Fragment Shaders

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

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

2016: Vulkan 1.1

2020: Vulkan 1.2

There is lots more detail at:
https://en.wikipedia.org/wiki/Vulkan_(API)
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 was the original Vulcan?

From WikiPedia:

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


Why Name it after the God of the Forge?

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

Playing “Where’s Waldo” with Khronos Membership
Who's Been Specifically Working on Vulkan?

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

Vulkan Differences from OpenGL

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

The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

MC → Model Transform → View Transform → Per-vertex Lighting → Projection Transform → Fragment Processing, Texturing, Per-Fragment Lighting → Rasterization

Framebuffer

MC → WC → EC
The Basic Computer Graphics Pipeline, Shader-style

- gl_Vertex, gl_Normal, gl_Color
- Per-vertex in variables
- gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix
- Uniform Variables

Vertex Shader

- MC = Model Vertex Coordinates
- WC = World Vertex Coordinates
- EC = Eye Vertex Coordinates

Fragment Shader

- gl_Position, Per-vertex out variables
- Uniform Variables

Rasterization

- gl_FragColor
- Per-fragment in variables
- Uniform Variables

The Basic Computer Graphics Pipeline, Vulkan-style

- gl_Position, Per-vertex out variables
- Uniform Variables

Vertex Shader

- gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix
- Uniform Variables

Fragment Shader

- gl_FragColor
- Per-fragment in variables
- Uniform Variables

Rasterization

- Output color(s)

Vulkan Highlights: Command Buffers

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

Moving part of the driver into the application

- Complex drivers lead to driver overhead and cross vendor unpredictability
- Error management is always active
- Driver processes full shading language source
- Separate APIs for desktop and mobile markets

Vulkan

- Simpler drivers for low-overhead efficiency and cross vendor portability
- Layered architecture so validation and debug layers can be unloaded when not needed
- Run-time only has to ingest SPIR-V intermediate language
- Unified API for mobile, desktop, console and embedded platforms
Vulkan Highlights: Pipeline State Objects

- In OpenGL, your "pipeline state" is the combination of whatever your current graphics attributes are: color, transformations, textures, shaders, etc.
- Changing the state on-the-fly one item at-a-time is very expensive.
- Vulkan forces you to set all your state variables at once into a "pipeline state object" (PSO) data structure and then invoke the entire PSO 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

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

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

Vulkan Code has a Distinct "Style" of Setting Information in structs

This "style" of setting information is 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 = NULL;
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 = NULL;
VK_RESULT result = vkCreateBuffer ( LogicalDevice, &vbci, PALLOCATOR, &Buffer );
```

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

```
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = NULL;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = 0;
result = vkAllocateMemory( LogicalDevice, &vmai, PALLOCATOR, &MatrixBufferMemoryHandle );
```

```
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```
### Steps in Creating Graphics using Vulkan

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

### Vulkan GPU Memory

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

### Vulkan Render Passes

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

### Vulkan Compute Shaders

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

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

Vulkan Shaders

- GLSL is the same as before ... almost
- For places it’s not, an implied `#define VULKAN 100`
  is automatically supplied by the compiler
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)
- SPIR-V gets turned into fully-compiled code at runtime
- The SPIR-V spec has been public for years – new shader languages are surely being developed
- OpenCL and OpenGL have adopted SPIR-V as well

Advantages:

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

The Vulkan Sample Code Included with These Notes
Sample Program Output

Sample Program Keyboard Inputs

'I' (ell), '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', ..., '9', 'a', ..., 'g': 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.

9. At times, I have copied lines from vulkan_core.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
void main( int argc, char * argv[] )
{
    Width = 1024;
    Height = 1024;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
    FpDebug = stderr;
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
    Reset();
    InitGraphics();
    /\ loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents( );
        Time = glfwGetTime( ); // elapsed time, in double-precision
        UpdateScene( );
        RenderScene( );
    }
    fprintf(FpDebug, "Closing the GLFW window
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```

InitGraphics( ), I

```c
void InitGraphics ( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    Init02Instance();
    Init03PhysicalDeviceAndDeviceProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices),           &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light),      &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer(  sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer,                   (void *) VertexData );
    Init06CommandPool( );
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init08Swapchain( );
    Init09DepthStencilImage( );
    Init10RenderPasses( );
    Init11Framebuffer( );
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorSetLayouts();
    Init13DescriptorSets( );
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
                                        VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

A Colored Cube

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

InitGraphics( ), II

```c
int Init07TextureSampler( &MyPuppyTexture.texSampler );
int Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
int Init08Swapchain( );
int Init09DepthStencilImage( );
int Init10RenderPasses( );
int Init11Framebuffer( );
int Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
int Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
int Init13DescriptorSetPool( );
int Init13DescriptorSetLayouts();
int Init13DescriptorSets( );
int Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
                                        VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
```
A Colored Cube

The Vertex Data is in a Separate File that is #include'd into sample.cpp

# include "SampleVertexData.cpp"

Vulkan Software Philosophy

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

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

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

Vulkan has chosen to do it like this:

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

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

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

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

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

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

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

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

Reporting Error Results, I

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

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

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

Vulkan Program Flow – the Setup

Query the Physical Devices and Choose (1 in our case)
Decide on the Extensions and Layers You Want
Create the Logical Device
Create the Queue(s) (1 in our case)
Allocate and Fill memory for the Uniform Buffers
Allocate and Fill memory for the Vertices and Indices
Create the Command Buffers (3 in our case)
If using Textures, create the Sampler, Read the Texture, and move it to Device Local Memory
Create the Swap Chain (2 images in our case)
Be sure you have Compiled the Shaders into .spv files
Create the Descriptor Set Data Structures
Create the Graphics Pipeline Data Structure (in our case)

Vulkan Program Flow – the Rendering Loop

while (the GLFW Window should not close) {
    UpdateScene();
    RenderScene();
    for (all the different Graphics Pipeline Data Structures being used) {
        Bind the Graphics Pipeline Data Structure
        Set any Dynamic State Variables
        Bind the Proper Descriptor Set Values
        Do the Drawing
    }
    End the RenderPass
    End the Command Buffer
}

Extras in the Code

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

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

The same as OpenGL topologies, with a few left out.

```c
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,
    // additional topologies
} VKPrimitiveTopology;
```

#### A Colored Cube Example

This data is contained in the file `SampleVertexData.cpp`
Triangles Represented as an Array of Structures

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

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

This data is contained in the file `SampleVertexData.cpp`

Modeled in right-handed coordinates

Non-indexed Buffer Drawing

```
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.}
    }
};
```

From the file `SampleVertexData.cpp`:

```
Triangles

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

Stream of Vertices

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

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

A Preview of What `Init05DataBuffer` Does

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

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

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++ constructs rather than hardcoding the byte count!

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.

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

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++ sizeof() construct rather than hardcoding the byte count!

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.

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

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++ sizeof() construct rather than hardcoding the byte count!
Telling the Command Buffer what Vertices to Draw

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

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

Always use the C/C++ construct `sizeof`, rather than hardcoding a byte count!

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

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

Drawing with an Index Buffer

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

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

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

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

```c
Init05MyVertexDataBuffer( sizeof(JustVertexData), IN &MyJustVertexDataBuffer );  
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );  
Init05MyIndexDataBuffer( sizeof(JustIndexData), IN &MyJustIndexDataBuffer );  
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );
```
Drawing with an Index Buffer

```c
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer }; // 0, 1 = firstBinding, bindingCount
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
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;
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```

Indirect Drawing (not to be confused with Indexed)

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

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

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

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

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

Sometimes the Same Vertex Needs Multiple Attributes

Sometimes a vertex 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. Vertex #7 above has the same color, regardless of what face it is in. However, Vertex #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 Vertex Needs Multiple Attributes

Where values match at the corners (color)

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

Terrain Surfaces are a Great Application of Indexed Drawing

Triangle Strip #0
Triangle Strip #1
Triangle Strip #2

There is no question that it is OK for the (u,v) at these vertices to all be the same

But, to Draw that Terrain Surface, You Need "Primitive Restart"

"Primitive Restart" is used with:

- Indexed drawing
- TRIANGLE_FAN and TRIANGLE_STRIP topologies

A special "index" is used to indicate that the triangle strip should start over. This is more efficient than explicitly ending the current triangle strip and explicitly starting a new one.

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

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the restart index is 0xffff.

If your VkIndexType is VK_INDEX_TYPE_UINT32, then the restart index is 0xffffffff.

That is, a one in all available bits

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

Note: The OBJ file format uses 1-based indexing for faces!
Drawing an OBJ Object

MyBuffer MyObjBuffer; // global

MyObjBuffer = VkOsuLoadObjFile("filename.obj"); // initializes and fills the buffer with
// triangles defined in GPU memory with an array of struct vertex

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

const uint32_t firstInstance = 0;
const uint32_t firstVertex = 0;
const uint32_t instanceCount = 1;
const uint32_t vertexCount = MyObjBuffer.size / sizeof(struct vertex);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec4 color;
    glm::vec2 texCoord;
};

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

```cpp
typedef VkBuffer VkDataBuffer;
```

This is probably a bad idea in the long run.

**From the Quick Reference Card**

Even though Vulkan is up to 1.3, the most current Vulkan Reference card is version 1.1

Creating and Filling Vulkan Data Buffers

**Creating a Vulkan Data Buffer**

```c
VkBuffer Buffer;  
// or "VkDataBuffer Buffer"

VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or'ed bits of: >>
    VK_USAGE_TRANSFER_SRC_BIT
    VK_USAGE_TRANSFER_DST_BIT
    VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
    VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
    VK_USAGE_UNIFORM_BUFFER_BIT
    VK_USAGE_STORAGE_BUFFER_BIT
    VK_USAGE_INDEX_BUFFER_BIT
    VK_USAGE_VERTEX_BUFFER_BIT
    VK_USAGE_INDIRECT_BUFFER_BIT
vbci.sharingMode = << one of: >>
    VK_SHARING_MODE_EXCLUSIVE
    VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;
result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```

Creating a Vulkan Data 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 );
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;
}
```
**Finding the Right Type of Memory**

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

**Memory-Mapped Copying to GPU Memory, Example I**

```c
void *mappedDataAddr;
vkMapMemory( LogicalDevice, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&mappedDataAddr );
memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );
vkUnmapMemory( LogicalDevice, myBuffer.vdm );
```

**Memory-Mapped Copying to GPU Memory, Example II**

```c
struct vertex *vp;
vkMapMemory( LogicalDevice, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&vp );
for( int i = 0; i < numTrianglesInObjFile; i++ ) // number of triangles
{
    for( int j = 0; j < 3; j++ ) // 3 vertices per triangle
    {
        vp->position = glm::vec3( . . . );
        vp->normal = glm::vec3( . . . );
        vp->color = glm::vec3( . . . );
        vp->texCoord = glm::vec2( . . . );
        vp++;
    }
    vkUnmapMemory( LogicalDevice, myBuffer.vdm );
}
```
Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. 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 also includes a smattering of documentation.

See our class VMA noteset for more VMA details

```
#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( var, van, OUT &mappedDataAddr );
memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );
vmaUnmapMemory( var, van );
See our class VMA noteset for more VMA details
```

Something I've Found Useful

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

```
typedef struct MyBuffer
{
  VkDataBuffer buffer;
  VkDeviceMemory vdm;
  VkDeviceSize size;  // in bytes
} MyBuffer;
...
// example:
MyBuffer MyObjectUniformBuffer;
```

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.

```
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 are C/C++ structs used by the Sample Code to hold some uniform variables:

```c
// Scene struct
struct sceneBuf
{
    glm::mat4 uProjection;
    glm::mat4 uView;
    glm::mat4 uSceneOrient;
    glm::vec4 uLightPos;
    glm::vec4 uLightColor;
    float uTime;
};

// Object struct
struct objectBuf
{
    glm::mat4 uModel;
    glm::mat4 uSceneOrient;
    float uTime;
};
```

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

```glsl
The uNormal is set to:

```c
glm::inverseTransposes( uView * uSceneOrient * uModel )
```

In the vertex shader, each object vertex gets transformed by:

```c
uProjection* uView * uSceneOrient * uModel
```

In the vertex shader, each surface normal vector gets transformed by the uNormal.

```
const float EYEDIST = 3.0f;
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.);
Scene.uProjection = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Scene.uProjection[1][1] *= -1.; // account for Vulkan's LH screen coordinate system
Scene.uView = glm::lookAt( eye, look, up );
Scene.uSceneOrient = glm::mat4( 1. );
Object.uModelOrient = glm::mat4( 1. );
Object.uNormal = glm::inverseTransposes( Scene.uView * Scene.uSceneOrient * Object.uModel );
```

This code assumes that this line:

```
#define    GLM_FORCE_RADIANS
```

is listed before GLM is #included!

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

```
struct objectBuf Object;
```

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

```
MyBuffer MyObjectUniformBuffer;
```

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

```c
# example:
MyBuffer MyObjectUniformBuffer;
Init05UniformBuffer( sizeof(Object), OUT &MyObjectUniformBuffer );
Fill05DataBuffer( MyObjectUniformBuffer, IN (void *) &Object );
```
Creating and Filling the Data Buffer – the Details

**VkResult**

`Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer)`

```c
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);
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;
```

Creating and Filling the Data Buffer – the Details

**VkResult**

`Fill05DataBuffer(IN MyBuffer myBuffer, IN void * data)`

```c
// the size of the data had better match the size that was used to Init the buffer!
void * pGpuMemory;
vkMapMemory(LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory);
// 0 and 0 are offset and flags
memcpy(pGpuMemory, data, (size_t)myBuffer.size);
vkUnmapMemory(LogicalDevice, IN myBuffer.vdm);
return VK_SUCCESS;
```

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

What is a Vertex Buffer?

Vertex Buffers are how you draw things in Vulkan. They are very much like Vertex Buffer Objects in OpenGL, but more detail is exposed to you (a lot more…).

But, the good news is that Vertex Buffers are really just ordinary Data Buffers, so some of the functions will look familiar to you.

First, a quick review of computer graphics geometry . . .
Geometry: Where things are (e.g., coordinates)

Topology: How things are connected

Vulkan Topologies

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

Vulkan Topologies – Requirements and Orientation

Polygons must be:
- Convex and
- Planar

Polygons are traditionally:
- CCW when viewed from outside the solid object

It's not absolutely necessary, but there are possible optimizations if you are consistent
OpenGL Topologies – Vertex Order Matters

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

What does “Convex Polygon” Mean?

We could go all mathematical here, but let’s go visual instead. In a convex polygon, a line between any two points inside the polygon never leaves the inside of the polygon.

Convex

Not Convex

OK, now let’s go all mathematical. In a convex polygon, every interior angle is between 0° and 180°.

Convex

Not Convex

Why is there a Requirement for Polygons to be Convex?

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

Convex

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

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

https://github.com/ivanfratric/polypartition

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

---

Why is there a Requirement for Polygons to be Planar?

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

OK

OK

Not OK

---

Vertex Orientation Issues

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system.

Internally, however, the Vulkan pipeline uses a left-handed system:

The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the GLM projection matrix, like this:

```
ProjectionMatrix[ 1 ][ 1 ] *= -1.;
```

This is like saying “Y’ = -Y”.

---

A Colored Cube Example

```
static GLuint CubeTriangleIndices[3][3] = {
  { 0, 2, 3 },
  { 0, 3, 1 },
  { 4, 5, 7 },
  { 4, 7, 6 },
  { 1, 3, 7 },
  { 1, 7, 5 },
  { 0, 4, 6 },
  { 0, 6, 2 },
  { 2, 6, 7 },
  { 2, 7, 3 },
  { 0, 1, 5 },
  { 0, 5, 4 }
};
```
Triangles in an Array of Structures

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

# Array of vertex data
struct vertex VertexData[3] = {
    // Triangle 0-2-3:
    // Vertex #0:
    { -1., -1., -1. },
    { 0., 0., -1. },
    { 0., 0., 0. },
    { 1., 0. },

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

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

Vertex Orientation Issues

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

Because this 3D object is closed, Vulkan can save rendering time by not even bothering with triangles whose vertices look like they are oriented CW. This is called backface culling.

Vulkan's change in coordinate systems can mess up the backface culling.

So I recommend, at least at first, that you do **no culling**.

```
VkPipelineRasterizationStateCreateInfo vprsci;
...

vprsci.cullMode = VK_CULL_MODE_NONE
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

Filling the Vertex Buffer

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

VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
```

A Reminder of What Init05DataBuffer Does

```
VkResult Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    MyBuffer MyVertexDataBuffer;
    int MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    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.

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

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

```cpp
VkVertexInputAttributeDescription vviad[4]; // array per vertex input attribute
    // 4 = vertex, normal, color, texture coord
    vviad[0].location = 0; // location in the layout decoration
    vviad[0].binding = 0; // which binding description this is part of
    vviad[0].format = VK_FORMAT_VEC3; // x, y, z
    vviad[0].offset = offsetof( struct vertex, position ); // 0
    vviad[1].location = 1;
    vviad[1].binding = 0;
    vviad[1].format = VK_FORMAT_VEC3; // nx, ny, nz
    vviad[1].offset = offsetof( struct vertex, normal ); // 12
    vviad[2].location = 2;
    vviad[2].binding = 0;
    vviad[2].format = VK_FORMAT_VEC3; // r, g, b
    vviad[2].offset = offsetof( struct vertex, color ); // 24
    vviad[3].location = 3;
    vviad[3].binding = 0;
    vviad[3].format = VK_FORMAT_VEC2; // s, t
    vviad[3].offset = offsetof( struct vertex, texCoord ); // 36
```

```cpp
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;
```

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

```
```
Telling the Pipeline Data Structure about its Input

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

```cpp
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
vgpci.stageCount = 2; // number of shader stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr; // &vptsci
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0; // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );
```

Telling the Command Buffer what Vertices to Draw

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

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

Shaders and SPIR-V

The Shaders' View of the Basic Computer Graphics Pipeline

- You need 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.
Shader stages

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

How Vulkan GLSL Differs from OpenGL GLSL

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

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

Push Constants:
layout( push_constant ) ... ;

Specialization Constants:
layout( constant_id = 3 ) const int N = 5;
• Only for scalars, but a vector's components can be constructed from specialization constants

For example, Specialization Constants can be used with Compute Shaders:
layout( local_size_x_id = 8, local_size_y_id = 16 );
• This sets gl_WorkGroupSize.x and gl_WorkGroupSize.y
• gl_WorkGroupSize.z is set as a constant

Note: our sample code doesn't use this.
Vulkan Shader Compiling

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

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

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

To install the bash shell on your own Windows machine, go to this URL:

Or, follow these instructions:
1. Head to the Start menu search bar, type in ‘terminal,’ and launch the Windows Terminal as administrator. (On some systems, this is called the Command Prompt.)
2. Type in the following command in the administrator: wsl --install
3. Restart your PC once the installation is complete.

As soon as your PC boots up, the installation will begin again. Your PC will start downloading and installing the Ubuntu software. You’ll soon get asked to set up a username and password. This can be the same as your system’s username and password, but doesn’t have to be. The installation will automatically start off from where you left it.
Running glslangValidator.exe in bash

As long as I am on bash, I like using the `make` utility. To do that, put these shader compile lines in a file called `Makefile`:

```bash
ALLSHADERS: sample-vert.vert sample-frag.frag
glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```

Then type `make ALLSHADERS`:

```
ibpC:/mnt/c/MJB/Vulkan/Sample2019-CREASEDCUBE$ make ALLSHADERS:
ibpC:/mnt/c/MJB/Vulkan/Sample2019-CREASEDCUBE$ glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
ibpC:/mnt/c/MJB/Vulkan/Sample2019-CREASEDCUBE$ glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```

Running glslangValidator.exe

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

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

How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages, it just prints the name of the source file you just compiled.

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

So, if you use the Linux command `od -x` on the .spv file, like this:

```
od -x sample-vert.spv
```

the magic number shows up like this:

```
0000000 0203 0723 0000 0001 000a 0008 007e 0000
0000020 0000 0000 0011 0002 0001 0000 000b 0006
```

"od" stands for "octal dump", even though it can format the raw bits as most anything: octal, hexadecimal, bytes, characters, etc. "-x" means to format in hexadecimal.

Reading a SPIR-V File into a Vulkan Shader Module

```c
#ifndef _WIN32
typedef int errno_t;
int fopen_s( FILE**, const char *, const char * );
#endif
#define SPIRV_MAGIC             0x07230203

VkResult Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
    FILE *fp;
    #ifdef WIN32
        errno_t err = fopen_s( &fp, filename.c_str( ), "rb" );
        if( err != 0 )
    #else
        fp = fopen( filename.c_str( ), "rb" );
        if( fp == NULL )
    #endif
    {
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str( ) );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
    {
        fprintf( FpDebug, "Magic number for spir-v file '%s is 0x%08x -- should be 0x%08x
", filename.c_str( ), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char * 
```
VkShaderModule ShaderModuleVertex;
...
VkShaderModuleCreateInfo vsmci;
   vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
   vsmci.pNext = nullptr;
   vsmci.flags = 0;
   vsmci.codeSize = size;
   vsmci.pCode = (uint32_t *)code;
VkResult result = vkCreateShaderModule(LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex);
fprintf(FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str());
delete[] code;
return result;
}

Reading a SPIR-V File into a Shader Module

VkGraphicsPipelineCreateInfo

Shader stages
- Vertex
- Tesselation
- InputAssembly
- Viewport
- Rasterization
- MultiSample
- DepthStencil
- ColorBlend
- Dynamic
- Pipeline layout
- RenderPass

Vulkan: Creating a Pipeline

You can also take a look at SPIR-V Assembly

You can also take a look at SPIR-V Assembly

For example, if this is your Shader Source

```
#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;
}
```

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.
SPIR-V: More Information

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

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:

```bash
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:

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

glslc is included in your Sample .zip file

This causes a:

```cpp
#define NUMPOINTS 4
```

to magically be inserted into the top of your source code.

Instancing

Instancing is the ability to draw the same object multiple times.

1. It uses all the same vertices and the same graphics pipeline data structure each time.
2. It avoids the overhead of the program asking to have the object drawn again, letting the GPU/driver handle all of that.

```cpp
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?

Instancing – What and why?
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:

```glsl
class std140, set = 0, binding = 0) uniform sporadicBuf {
  int uMode;
  int uUseLighting;
  int uNumInstances;
} Sporadic;
.
void main() {
  
  float DELTA = 3.0;
  
  float a = sqrt(float(Sporadic.uNumInstances));
  float c = ceil(float(a));
  int cols = int(c);
  int fullRows = gl_InstanceIndex / cols;
  int remainder = gl_InstanceIndex %% cols;
  
  float xdelta = DELTA * float(remainder);
  float ydelta = DELTA * float(fullRows);
  
  vColor = vec3(1., float((1. + gl_InstanceIndex)) / float(Sporadic.uNumInstances), 0.);
  
  vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
  
  gl_Position = PVM * vertex;
}
```

Making each Instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer array and reference them.

Still uses `gl_InstanceIndex`

In the vertex shader:

```glsl
class std140, set = 4, binding = 0) uniform colorBuf {
  vec3 uColors[1024];
} Colors;
.
out vec3 vColor;
.
int index = gl_InstanceIndex %% 1024; // gives 0 - 1023
vColor = Colors.uColors[index];
.
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);

gl_Position = PVM * vertex;
```
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, "%s\n", extensions[i]);
}
```

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%04x = \"%c\"\n", key );
                fflush(FpDebug);
        }
    }
```
### GLFW Mouse Button Callback

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

### GLFW Mouse Motion Callback

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

### Looping and Closing GLFW

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

### Does not block – processes any waiting events, then returns

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

---

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

```c
glfwWaitEvents();
```

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

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

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

```c
glfwPostEmptyEvent();
```
GLM is a set of C++ classes and functions to fill in the programming gaps in writing the basic vector and matrix mathematics for OpenGL applications. However, even though it was written for OpenGL, it works fine with Vulkan.

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

You can find it at:
http://glm.g-truc.net/0.9.8.5/

You invoke GLM like this:

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

If GLM is not installed in a system place, put it somewhere you can get access to. Later on, these notes will show you how to use it from there.

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.

---

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 matrix
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
modelview = glm::lookAt( eye, look, up );
```

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

---

The Most Useful GLM Variables, Operations, and Functions

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

- **// constructor:**

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

- **// multiplications:**

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

- **// 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 );
  ```
The Most Useful GLM Variables, Operations, and Functions

// viewing volume (assign, not concatenate):

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

// viewing (assign, not concatenate):

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

Installing GLM into your own space

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

Here’s what that GLM folder looks like

Telling Visual Studio about where the GLM folder is
A period, indicating that the project folder should also be searched when a #include <xxx> is encountered. If you put it somewhere else, enter that full or relative path instead.

How Does this Matrix Stuff Really Work?

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

Or, in matrix form:

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

GLM in the Vulkan sample.cpp Program

```
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.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix );
// note: inverseTransform!
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
```

Transformation Matrices

### Translation

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

### Rotation about X

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

### Scaling

\[
\begin{pmatrix}
{x'} \\
{y'} \\
{z'}
\end{pmatrix} =
\begin{pmatrix}
S & 0 & 0 & 0 \\
0 & S & 0 & 0 \\
0 & 0 & S & 0
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

### Rotation about Y

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

### Rotation about Z

\[
\begin{pmatrix}
{x'} \\
{y'} \\
{z'}
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & \sin \theta & 0 & 0 \\
-\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]
The Rotation Matrix for an Angle ($\theta$) about an Arbitrary Axis ($Ax$, $Ay$, $Az$)

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

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

Matrix Multiplication is not Commutative

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

A: We create more than one matrix.
Matrix Multiplication is Associative

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \begin{pmatrix}
  T_{rA,sB} & \cdot & \begin{pmatrix}
  R_q & \cdot & T_{rA,sB}
\end{pmatrix} & \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\end{pmatrix}
\]

One matrix to rule them all – the **Current Transformation Matrix**, or **CTM**

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

 glm::mat4 Model = uViewMatrix * uSceneMatrix * uModelMatrix;
 uNormalMatrix = glm::inverseTranspose( glm::mat3(Model) );

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

From the Data Buffer Notset

Here’s the vertex shader shader code to use the matrices:

```glsl
layout( std140, set = 0, binding = 0 ) uniform sceneMatBuf
{
  mat4 uProjectionMatrix;
  mat4 uViewMatrix;
  mat4 uSceneMatrix;
} SceneMatrices;

layout( std140, set = 1, binding = 0 ) uniform objectMatBuf
{
  mat4 uModelMatrix;
  mat4 uNormalMatrix;
} ObjectMatrices;

Here’s the vertex shader shader code to use the matrices:

```glsl
vNormal = uNormalMatrix * aNormal;

```glsl
gl_Position = uProjectMatrix * uViewMatrix * uSceneMatrix * uModelMatrix * aVertex;
```

"CTM"
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;
```

*std140* has to do with the alignment of the different data types. It is the simplest, and so we use it in class to give everyone the highest probability that their system will be compatible with the alignment.

What are Descriptor Sets?

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

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

```glsl
for( sporadically )
{
    Bind Descriptor Set #0
    for( the entire scene )
    {
        Bind Descriptor Set #1
        for( each object in the scene )
        {
            Bind Descriptor Set #2
            Do the drawing
        }
    }
}
```

Our example will assume the following shader uniform variables:

```c++
struct sporadicBuf
{
    int uMode;
    int uUseLighting;
    int uNumInstances;
} Sporadic;

struct sceneBuf
{
    glm::mat4 uProjection;
    glm::mat4 uView;
    glm::mat4 uSceneOrient;
    glm::vec4 uLightPos;
    glm::vec4 uLightColor;
    glm::vec4 uLightKaKdKs;
    float uTime;
} Scene;

struct objectBuf
{
    glm::mat4 uModel;
    glm::mat4 uNormal;
    glm::vec4 uColor;
    float uShininess;
} Object;

layout( std140, set = 0, binding = 0 ) uniform sporadicBuf uSporadic;
layout( std140, set = 1, binding = 0 ) uniform sceneBuf uScene;
layout( std140, set = 2, binding = 0 ) uniform objectBuf uObject;
layout( binding = 7 ) 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 during initialization and then pull from that pool later.

```
void Init13DescriptorSetPool()
{
    VkResult result;
    VkDescriptorPoolSize vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;
    #ifdef CHOICES
    VK_DESCRIPTOR_TYPE_SAMPLER
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
    #endif
    VkDescriptorPoolCreateInfo vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;
    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = &vdps[0];
    result = vkCreateDescriptorPool(LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
    return result;
}
```

Step 2: Define the Descriptor Set Layouts

I think of Descriptor Set Layouts as a kind of "Rosetta Stone" that allows the Graphics Pipeline data structure to allocate room for the uniform variables and to access them.

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

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

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

// DS #3:
VkDescriptorSetLayoutBinding TexSamplerSet[1];
TexSamplerSet[0].binding = 0;
TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
TexSamplerSet[0].descriptorCount = 1;
TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;
```
Step 2: Define the Descriptor Set Layouts

```cpp
// globals:
VkDescriptorPool DescriptorPool;
VkDescriptorSetLayout DescriptorSetLayouts[4];
VkDescriptorSet

binding
descriptorType
descriptorCount
pipeline stage(s)

SporadicSet DS Layout Binding:
SceneSet DS Layout Binding:
ObjectSet DS Layout Binding:
TexSamplerSet DS Layout Binding:

bindingCount
type
number of that type
pipeline stage(s)

vdslc0 DS Layout CI:
vdslc1 DS Layout CI:
vdslc2 DS Layout CI:
vdslc3 DS Layout CI:

// globals:
VkDescriptorPool DescriptorPool;
VkDescriptorSetLayout DescriptorSetLayouts[4];
VkDescriptorSet

// Define the Descriptor Set Layouts

// Define the Descriptor Set Layouts

VkDescriptorSetLayoutCreateInfo
vdslc0
vdslc1
vdslc2
vdslc3;

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

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

```cpp
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];

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

Step 4: Allocating the Memory for Descriptor Sets

```cpp
VkResult Init15AllocateDescriptorSets()
{
    VkResult result;
    VkDescriptorSetAllocateInfo

result = vkAllocateDescriptorSets(LogicalDevice, IN &DescriptorSetPool, IN &DescriptorPool, IN DescriptorSetLayoutsCount, OUT DescriptorSet);
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: Allocating the Memory for Descriptor Sets

```c
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 = IN &vdbi0;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pTexelBufferView = (VkBufferView *)nullptr;

// ds 1:
VkWriteDescriptorSet vwds1;
// ds 2:
VkWriteDescriptorSet vwds2;
// ds 3:
VkWriteDescriptorSet vwds3;
```

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

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

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

```c
uint32_t copyCount = 0;
// this could have been done with one call and an array of VkWriteDescriptorSets:
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);
```
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

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

#define CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

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

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

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

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

```cpp
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 Descriptor Set Journey

- **VkDescriptorPoolCreateInfo**
  - Create the pool of Descriptor Sets for future use

- **VkCreateDescriptorPool( )**

- **VkDescriptorSetLayoutBinding**
  - Describe a particular Descriptor Set layout and use it in a specific Pipeline layout

- **VkCreateDescriptorSetLayout( )**

- **VkCreatePipelineLayout( )**

- **VkAllocateDescriptorSets( )**
  - Allocate memory for particular Descriptor Sets

- **VkCreateDescriptorSet( )**
  - 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.
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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

The Basic Idea

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

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

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

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that the rasterizer is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when it goes to draw each pixel, it uses that pixel’s interpolated (s,t) to lookup a color in the texture image.
Enable texture mapping:
```c
glEnable( GL_TEXTURE_2D );
```

Draw your polygons, specifying \( s \) and \( t \) at each vertex:
```c
glBegin( GL_POLYGON );
glTexCoord2f( s0, t0 );
glNormal3f( nx0, ny0, nz0 );
glVertex3f( x0, y0, z0 );
glTexCoord2f( s1, t1 );
glNormal3f( nx1, ny1, nz1 );
glVertex3f( x1, y1, z1 );
. . .
glEnd( );
```

Disable texture mapping:
```c
glDisable( GL_TEXTURE_2D );
```

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

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

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

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

Or, for a sphere,

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

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

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

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

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

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

Memory Types

- CPU Memory
  - Host
  - Visible
  - GPU Memory
- GPU Memory
  - Device
  - Local
  - GPU Memory

memcpy() vkCmdCopyImage()
Memory Types

NVIDIA A6000 Graphics:
6 Memory Types:
Memory 0:
Memory 1: DeviceLocal
Memory 2: HostVisible HostCoherent
Memory 3: HostVisible HostCoherent HostCached
Memory 4: DeviceLocal HostVisible HostCoherent
Memory 5: DeviceLocal

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

Something I've Found Useful

I find it handy to encapsulate texture information in a struct, just like I do with buffer information:

// holds all the information about a data buffer so it can be encapsulated in one variable:
typedef struct MyBuffer
{
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;

// holds all the information about a texture so it can be encapsulated in one variable:
typedef struct MyTexture
{
    uint32_t                        width;
    uint32_t                        height;
    unsigned char *            pixels;
    VkImage texImage;
    VkImageView texImageView;
    VkSampler texSampler;
    VkDeviceMemory vdm;
} MyTexture;

I find it handy to encapsulate texture information in a struct, just like I do with buffer information:

// holds all the information about a data buffer so it can be encapsulated in one variable:
typedef struct MyBuffer
{
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;

// holds all the information about a texture so it can be encapsulated in one variable:
typedef struct MyTexture
{
    uint32_t                        width;
    uint32_t                        height;
    unsigned char *            pixels;
    VkImage texImage;
    VkImageView texImageView;
    VkSampler texSampler;
    VkDeviceMemory vdm;
} MyTexture;

Texture Sampling Parameters

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 );

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

*Latin: multo in parvo, “many things in a small place”*
void * gpuMemory;

vkMapMemory(

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

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

VkImageMemoryBarrier vimb;

vimb.subresourceRange = visr;

vimb.dstAccessMask = 0;

vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;

vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;

vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

vimb.pNext = nullptr;

vkCmdPipelineBarrier

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

// transition the texture buffer layout:

// now do the final image transfer:

// because we are transferring into it and will eventual sample from it

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

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

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

    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = textureImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
    vimb.subresourceRange = visr;

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

    // *******************************************************************************
result = vkEndCommandBuffer( TextureCommandBuffer );

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

result = vkQueueSubmit( Queue, 1, IN &vsi, VK_NULL_HANDLE );
result = vkQueueWaitIdle( Queue );

// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

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

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

// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)

Access to an Image
Image View
The Actual Image Data

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

Reading in a Texture from a BMP File

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

MyTexture MyPuppyTexture;

result = Init06TextureBufferAndFillFromBmpFile( "puppy1.bmp", &MyPuppyTexture);
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.
What is the Vulkan Graphics Pipeline Data Structure (GPDS)?

Here’s what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”. It is a data structure.
2. Since you know the OpenGL state, a lot of the Vulkan GPDS will seem familiar to you.
3. The current shader program is part of the state. (It was in OpenGL too, we just didn’t make a big deal off it.)
4. The Vulkan Graphics Pipeline is not the processes that OpenGL would call “the graphics pipeline”.
5. 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 GPDS.
6. The shaders get compiled the rest of the way when their Graphics Pipeline Data Structure gets created.

There are also a Vulkan Compute Pipeline Data Structure and a Raytrace Pipeline Data Structure – we will get to those later.

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
VkPipelineLayout GraphicsPipelineLayout; // global
-
VkResult Init14GraphicsPipelineLayout( )
{
    VkResult result;
    VkPipelineLayoutCreateInfo vplci = {
        .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,
        .pNext = nullptr,
        .flags = 0,
        .setLayoutCount = 4,
        .pSetLayouts = &DescriptorSetLayouts[0],
        .pushConstantRangeCount = 0,
        .pPushConstantRanges = (VkPushConstantRange *)nullptr,
    };
    result = vkCreatePipelineLayout(LogicalDevice, &vplci, PALLOCATOR, &GraphicsPipelineLayout);
    return result;
}
```

Let the Pipeline Layout know about the Descriptor Set and Push Constant layouts.

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 Graphics Pipeline Data Structure Contains the Following State Items:

- Dynamic State: which states can be set dynamically (bound to the command buffer).
- Blending: blendEnable.
- Rasterization: cullMode, polygonMode, frontFace.
- Assembly: topology (e.g., VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST).
- Viewport: x, y, w, h, minDepth, maxDepth.
- Scissoring: x, y, w, h.
- Rasterization: cullMode, polygonMode, frontFace, lineWidth.
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp.
- DepthClampEnable: whether to clamp the depth.
- MultiSample: sampleCount, sampleMask, minSampleShadingValue.
- VertexInput: inputRate, binding, stride, inputRate.
- Per-vertex input bindings: binding, stride, inputRate.
- Color Write Mask: srcColorBlendFactor, dstColorBlendFactor.
- VertexShader: shaderStage.
- Pipeline Layout: Descriptor Sets, Push Constants.

**Bold/Italics** indicates that this state item can be changed with Dynamic State Variables.

### Creating a Typical Graphics Pipeline

```c
VkResult Init4GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader, VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
    #ifdef ASSUMPTIONS
    #endif
    #ifdef CHOICES
    
    #endif
    #ifdef BITS
    
    #endif
    VkPipelineShaderStageCreateInfo vertexShaderCreateInfo, fragmentShaderCreateInfo;    // an array containing one of these per shader module you are using
    VkSpecializationInfo specializationInfo;    // an array containing one of these per shader module you are using
    VkVertexInputBindingDescription vvibd[1];       // an array containing one of these per buffer being used
    VkVertexInputAttributeDescription vviasd[1];       // an array containing one of these per buffer being used
    VkPipelineColorBlendAttachmentState srcColorBlends[1];    // an array containing one of these per colorWriteMask
    VkPipelineColorBlendAttachmentState dstColorBlends[1];    // an array containing one of these per colorWriteMask
    VkPipelineColorBlendStateCreateInfo colorBlends[1];    // an array containing one of these per colorWriteMask
    VkPipelineDepthStencilStateCreateInfo depthStencils[1];    // an array containing one of these per stencilTestEnable
    VkPipelineRasterizationStateCreateInfo rasterizations[1];    // an array containing one of these per cullMode
    VkPipelineInputAssemblyStateCreateInfo inputAssemblies[1];    // an array containing one of these per topology
    VkPipelineViewportStateCreateInfo viewports[1];    // an array containing one of these per viewport
    VkPipelineDynamicStateCreateInfo dynamicStates[1];    // an array containing one of these per dynamicState
    VkPipelineLayoutCreateInfo layoutCreateInfo;
    
    // Create the shader modules
    vertexShaderCreateInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
    vertexShaderCreateInfo.pSpecializationInfo = nullptr;
    vertexShaderCreateInfo.pName = "main";
    vkCreateShaderModule(&vertexShader, &vertexShaderCreateInfo, nullptr, OUT &vertexShader);
    
    fragmentShaderCreateInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    fragmentShaderCreateInfo.pSpecializationInfo = nullptr;
    fragmentShaderCreateInfo.pName = "main";
    vkCreateShaderModule(&fragmentShader, &fragmentShaderCreateInfo, nullptr, OUT &fragmentShader);
    
    // Create the layout
    layoutCreateInfo.setLayoutCount = 2;
    layoutCreateInfo.pSetLayouts = &descriptorSetLayouts[0], &pipelineLayouts[0];
    layoutCreateInfo.pPushConstantRanges = nullptr;
    vkCreatePipelineLayout( 1, &layoutCreateInfo, nullptr, OUT &pipelineLayouts[0]);
    
    // Create the pipeline
    VkGraphicsPipelineCreateInfo pipelineCreateInfo;
    pipelineCreateInfo.stageCount = 2;
    pipelineCreateInfo.pStages = &vertexShaderCreateInfo, &fragmentShaderCreateInfo;
    pipelineCreateInfo.pVertexInputState = nullptr;
    pipelineCreateInfo.pTessellationState = nullptr;
    pipelineCreateInfo.pGeometryState = nullptr;
    pipelineCreateInfo.pRasterizationState = &rasterizations[0];
    pipelineCreateInfo.pColorBlendState = &colorBlends[0];
    pipelineCreateInfo.pDepthStencilState = &depthStencils[0];
    pipelineCreateInfo.pMultisampleState = &multisampleStates[0];
    pipelineCreateInfo.pViewportState = &viewports[0];
    pipelineCreateInfo.pInputAssemblyState = &inputAssemblies[0];
    pipelineCreateInfo.pDynamicState = &dynamicStates[0];
    pipelineCreateInfo.layout = pipelineLayouts[0];
    pipelineCreateInfo.basePipelineIndex = -1;    // best to do this because of the projectionMatrix[1][1] *= -1.;
    
    // Create the pipeline
    vkCreateGraphicsPipelines( 1, &pipelineCache, 1, &pipelineCreateInfo, nullptr, OUT &pGraphicsPipeline);
}
```

### The Shaders to Use

The shaders to use are specified in the code snippet above. The shaders can be created as separate modules or as half-compiled SPIR-V modules.

### Creating a Graphics Pipeline from a Lot of Pieces

The code snippet above demonstrates how to create a typical graphics pipeline using a variety of state items. The pipeline is created as a series of VkPipeline* objects, each representing a component of the pipeline (e.g., shader modules, layout, pipeline state). The pipeline is assembled by calling `vkCreateGraphicsPipelines()`.

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.
If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is indicated that the primitive should start over. This is more efficient than explicitly ending the current triangle strip and explicitly starting a new one.

• TRIANGLE_FAN and TRIANGLE_STRIP topologies
• Indexed drawing.

"Restart Enable" is used with:

- Indexed drawing.
- TRIANGLE_FAN and TRIANGLE_STRIP topologies

If vpiasci.primitiveRestartEnable is VK_TRUE, then a special “index” can be used to indicate that the primitive should start over. This is more efficient than explicitly ending the current triangle strip and explicitly starting a new one.

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 available:

If your VkIndexType is VK_INDEX_TYPE_UINT32, then the special index is not available:

That is, a one in all available bits
One Really Good use of Indexed Drawing and 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 (squeezed) 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.

Setting the Rasterizer State

Declare the viewport information

Declare the scissoring information

Group the viewport and scissoring information together

Declare information about how the rasterization will take place
**What is “Depth Clamp Enable”?**

```
vprsci.depthClampEnable = VK_FALSE;
```

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

A good use for this is **Polygon Capping**:

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

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

---

**What is “Depth Bias Enable”?**

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

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

---

**MultiSampling State**

Declare information about how the multisampling will take place

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

We will discuss MultiSampling in a separate noteset.

---

**Color Blending State for each Color Attachment**

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

```
VkPipelineColorBlendAttachmentState vpcbas;
```

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

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

```
Color\text{new} = (1-\alpha) \ast \text{Colorexisting} + \alpha \ast \text{Colorincoming}
```

0 ≤ α ≤ 1.

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

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

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

**VkDynamicState**

```cpp
vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR, ...
```

Which Pipeline Variables can be Set Dynamically

Just used as an example in the Sample Code

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

This allows you to give the graphics a full Graphics Pipeline Data Structure and then change some elements of it.

**The Stencil Buffer**

Here's what the Stencil Buffer can do for you:

1. While drawing into the Back Buffer, you can write values into the Stencil Buffer at the same time.
2. While drawing into the Back Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.
3. The Stencil Buffer can be used to write-protect certain parts of the Back Buffer.

You Can Think of the Stencil Buffer as a Separate Framebuffer, or, You Can Think of it as being Per-Pixel

Both are correct, but I like thinking of it "per-pixel" better.
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

I Once Used the Stencil Buffer to Create a Magic Lens for Volume Data

In this case, the scene inside the lens was created by drawing the same object, but drawing it with its near clipping plane being farther away from the eye position.
Using the Stencil Buffer to Perform Polygon Capping

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

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting

Using the Stencil Buffer to Better Outline Polygons

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
}

Before After
Using the Stencil Buffer to Perform Hidden Line Removal

Operations for Depth Values

Stencil Operations for Front and Back Faces

Putting it all Together! (finally...)

Group all of the individual state information and create the pipeline
When Drawing, We will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer

```cpp
VkPipeline GraphicsPipeline; // global
...
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
```

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

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

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

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

```
Fixed-layout Pipeline Blocks
Variable-layout Pipeline Blocks
```

Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

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

Vulkan Pipeline Creation

- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkVertexInputBindingDescription
- VkViewportStateCreateInfo
- Viewport
- x, y, w, h,
- minDepth,
- maxDepth
- offset
- extent
- Scissor

- VkPipelineRasterizationStateCreateInfo
- cullMode
- polygonMode
- frontFace
- lineWidth

- VkSpecializationInfo
- which stage (VERTEX, etc.)
- VkShaderModule

- VkPipelineInputAssemblyStateCreateInfo
- Topology

- VkVertexInputAttributeDescription
- binding
- stride
- inputRate
- location
- binding
- format
- offset

- VkPipelineDepthStencilStateCreateInfo

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

- VkPipelineColorBlendAttachmentState

- VkPipelineDynamicStateCreateInfo

Which Pipeline State Variables can be Changed Dynamically

The possible dynamic variables are shown in the VkDynamicState enum:

- VK_DYNAMIC_STATE_VIEWPORT
- VK_DYNAMIC_STATE_SCISSOR
- VK_DYNAMIC_STATE_LINE_WIDTH
- VK_DYNAMIC_STATE_TESSELATION_CONTROL
- VK_DYNAMIC_STATE_TESSELATION_EVALUATION_INPUTS
- VK_DYNAMIC_STATE_TESSELATION_EVALUATION_OUTPUTS
- VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK
- VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
- VK_DYNAMIC_STATE_STENCIL_REFERENCE
- VK_DYNAMIC_STATE_BLEND_CONSTANTS
- VK_DYNAMIC_STATE_DEPTH_BOUNDS
- VK_DYNAMIC_STATE_BLEND_FACTOR
- VK_DYNAMIC_STATE_BLEND_OP
- VK_DYNAMIC_STATE_STENCIL_FRONT_OPERATIONS
- VK_DYNAMIC_STATE_STENCIL_BACK_OPERATIONS
- VK_DYNAMIC_STATE_BLEND_EQUATION
- VK_DYNAMIC_STATE_COLOR_WRITE_MASK
- VK_DYNAMIC_STATE_DEPTH_BIAS
- VK_DYNAMIC_STATE_DEPTH_BIAS_CONSTANTS

Creating a Pipeline

- VkDynamicState
- VkPipelineDynamicStateCreateInfo
- VkGraphicsPipelineCreateInfo

Filling the Dynamic State Variables in the Command Buffer

First call:

```c
vkCmdBindPipeline( … );
```

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

- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports );`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors );`
- `vkCmdSetLineWidth(commandBuffer, linewidth );`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );`
- `vkCmdSetBlendConstants(commandBuffer, blendConstants[4] );`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds );`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask );`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask );`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference );`
VK_DYNAMIC_STATE_VIEWPORT = 0,
VK_DYNAMIC_STATE_SCISSOR = 1,
VK_DYNAMIC_STATE_LINE_WIDTH = 2,
VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
VK_DYNAMIC_STATE_CULL_MODE = 1000267000,
VK_DYNAMIC_STATE_FRONT_FACE = 1000267001,
VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY = 1000267002,
VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT = 1000267003,
VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT = 1000267004,
VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE = 1000267005,
VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE = 1000267006,
VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE = 1000267007,
VK_DYNAMIC_STATE_DEPTH_COMPARE_OP = 1000267008,
VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE = 1000267009,
VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE = 1000267010,
VK_DYNAMIC_STATE_STENCIL_OP = 1000267011,
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, but doesn't have to be
- 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 already has them
- Each Queue belongs to a Queue Family
- We don't create Queue Families – the Physical Device already has them

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;
}
```

Querying what Queue Families are Available

```c
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT &vqfp);
for(unsigned int i = 0; i < count; i++)
{
    fprintf(FpDebug, "Found %d Queue Families:
        0: Queue Family Count = %2d  ;  
        1: Queue Family Count = %2d  ;
        2: Queue Family Count = %2d  ;
        
        0: Queue Family Count = 16  ;    Graphics Compute Transfer
        1: Queue Family Count =   2  ;    Transfer
        2: Queue Family Count =   8  ;    Compute Transfer
        
        For the Nvidia A6000 cards:
    
    Similarly, we Can Write a Function that Finds the Proper Queue Family
```

Creating a Logical Device Needs to Know Queue Family Information

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

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

```cpp
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

```cpp
VkResult Init06CommandBuffers()
{
    VkResult result;
    // allocate 2 command buffers for the double-buffered rendering:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 2;           // 2, because of double-buffering
        result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &CommandBuffers[0]);
    }
    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 1;
        result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer);
    }
    return result;
}
```

Beginning a Command Buffer – One per Image

```cpp
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
uint32_t nextImageIndex;
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;
result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
```

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

- vkCmdBeginConditionalRendering
- vkCmdBeginDebugUtilsLabel
- vkCmdBeginQuery
- vkCmdBeginQueryIndexed
- vkCmdBeginRendering
- vkCmdBeginRenderPass
- vkCmdBeginRenderPass2
- vkCmdBeginTransformFeedback
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindInvocationMask
- vkCmdBindPipeline
- vkCmdBindPipelineShaderGroup
- vkCmdBindShadingRateImage
- vkCmdBindTransformFeedbackBuffers
- vkCmdBindVertexBuffers
- vkCmdBindVertexBuffers2
- vkCmdBlitImage
- vkCmdBlitImage2
- vkCmdBuildAccelerationStructure
- vkCmdBuildAccelerationStructureIndirect
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearColorImageToBuffer
- vkCmdClearColorImageToBuffer2
- vkCmdClearDepthStencilImage
- vkCmdClearDepthStencilImageToBuffer
- vkCmdClearDepthStencilImageToBuffer2
- vkCmdClearDepthStencilImageToAccelerationStructure

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

- vkCmdCopyQueryPoolResults
- vkCmdCuLaunchKernelX
- vkCmdDebugMarkerBegin
- vkCmdDebugMarkerEnd
- vkCmdDispatch
- vkCmdDispatchBase
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndexedIndirectCount
- vkCmdDrawIndirect
- vkCmdDrawIndirectByteCount
- vkCmdDrawIndirectCount
- vkCmdDrawMeshTasksIndirect
- vkCmdDrawMeshTasksIndirectCount
- vkCmdDrawMeshTasksIndirectCount
- vkCmdDrawMeshTasks

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

- vkCmdPreprocessGeneratedCommands
- vkCmdPushConstants
- vkCmdPushDescriptorSet
- vkCmdPushDescriptorSetWithTemplate
- vkCmdResetEvent
- vkCmdResetEvent2
- vkCmdResetQueryPool
- vkCmdResolveImage
- vkCmdResolveImage2
- vkCmdSetBlendConstants
- vkCmdSetCheckpoint
- vkCmdSetCoarseSampleOrder
- vkCmdSetCullMode
- vkCmdSetDepthBias
- vkCmdSetDepthBiasEnable
- vkCmdSetDepthBounds
- vkCmdSetDepthBoundsEnable
- vkCmdSetDepth CompareOp
- vkCmdSetDepthTestEnable
- vkCmdSetDeviceMask
- vkCmdSetDiscardRectangle
- vkCmdSetExclusiveScissor
- vkCmdSetFragmentShadingRate
- vkCmdSetFragmentShadingRateEnable
- vkCmdSetFrontFace
- vkCmdSetLineStipple
- vkCmdSetLineWidth
- vkCmdSetLogicOp
- vkCmdSetMulti
- vkCmdSetPipeline
- vkCmdSetPrimitiveTopology
- vkCmdSetRayTracingPipelineStackSize
- vkCmdSetSampleLocations
- vkCmdSetScissor
- vkCmdSetScissorWithCount
- vkCmdSetStencilCompareMask
- vkCmdSetStencilOp
- vkCmdSetStencilReference
- vkCmdSetStencilTestEnable
- vkCmdSetStencilWriteMask
- vkCmdSetViewport
- vkCmdSetViewportShadingRate
- vkCmdSetViewportWithCount
- vkCmdSetViewportWScaling
- vkCmdSubpassShading
- vkCmdTraceRaysIndirect
- vkCmdTraceRays
- vkCmdUpdateBuffer
- vkCmdUpdateBufferMarker
- vkCmdWriteAccelerationStructure
- vkCmdWriteBufferMarker
- vkCmdWriteTimestamp
- vkCmdWriteTimestamp2
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.f;
  vcdsv.stencil = 0;

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

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

  VkRenderPassBeginInfo vrpbi;
  vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  vrpbi.pNext = nullptr;
  vrpbi.renderPass = RenderPass;
  vrpbi.framebuffer = Framebuffers[nextImageIndex];
  vrpbi.renderArea = r2d;
  vrpbi.clearValueCount = 2;
  vrpbi.pClearValues = vcv;               // used for VK_ATTACHMENT_LOAD_OP_CLEAR
  vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);

  VkViewport viewport = {
    0,                     // x
    0,                     // y
    (float)Width,          // width
    (float)Height,         // height
    0,                     // minDepth
    1.                      // maxDepth
  };
  vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &viewport);

  VkRect2D scissor = {
    0, 0, Width, Height
  };
  vkCmdSetScissor(CommandBuffers[nextImageIndex], 0, 1, IN &scissor);

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

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

  VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
  VkDeviceSize offsets[1] = { 0 };
  vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);               // 0, 1 = firstBinding, bindingCount

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

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

  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;

  vkQueueSubmit(LogicalQueue, 1, &vsi, VK_NULL_HANDLE);
  vkQueueWaitIdle(LogicalQueue);

  return result;
}
The Entire Submission / Wait / Display Process

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

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

// 0 = queueIndex

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

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

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

VkPresentInfoKHR vpi;
vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
vpi.pImageIndices = &nextImageIndex;

result = vkQueuePresentKHR(presentQueue, IN &vpi);

What Happens After a Queue has Been Submitted?

As the Vulkan 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

How OpenGL Thinks of Framebuffers

Update → Depth → Back → Front

Refresh
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")
Here's What the Vulkan Spec Has to Say About Present Modes, I

`VkPresentModeKHR` specifies that the presentation engine does not wait for a vertical blanking period to update the current image. Meaning this mode may result in visible tearing. No internal querying of presentation requests is needed, as the requests are applied immediately.

`VkPresentModeKHR_MITIGATION_KHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. This mode cannot be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for re-use by the application. One request is removed from the queue and processed during each vertical blanking period when the queue is non-empty.

`VkPresentModeKHR_FIFO_KHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. This mode cannot be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period when the queue is non-empty. This is the only value of `presentMode` that is required to be supported.

`VkPresentModeKHR_FIFOLIMITED_KHR` specifies that the presentation engine generally waits for the next vertical blanking period to update the current image, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.

Creating a Swap Chain

```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 );
```

Here's What the Vulkan Spec Has to Say About Present Modes, II

`VkPresentModeKHR_MITIGATION_KHR` specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a shared presentable image. The presentation engine is only required to update the current image after a new presentation request is received. Therefore, the application must make a presentation request whenever an update is required. However, the presentation engine may update the current image at any point, meaning this mode may result in visible tearing.

`VkPresentModeKHR_MITIGATION_KHR` specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a shared presentable image. The presentation engine periodically updates the current image on a regular refresh cycle. The application is only required to make one initial presentation request, after which the presentation engine must update the current image without any need for further presentation requests. The application can indicate the image contents have been updated by making a presentation request, but this does not guarantee the timing of when it will be updated. This mode may result in visible tearing if rendering to the image is not timed correctly.

Creating a Swap Chain

```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?
result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, PresentImages ); // present views for the double-buffering:
PresentImages = new VkImage[ imageCount ];
for( unsigned int i = 0; i < imageCount; i++ )
{
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vivci.subresourceRange.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;
Vsemaphore imageReadySemaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
uint32_t nextImageIndex = 0;
vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );
```

Rendering into the Swap Chain, II

```c
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
VkFence renderFence;
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue );
```

Rendering into the Swap Chain, III

```c
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );
VkPresentInfoKHR
vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
```

Physical Devices

Vulkan: Overall Block Diagram

Instance

Physical Device

Logical Device

Queue

Command Buffer

Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Querying the Number of Physical Devices

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

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

result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
```c
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, "
%d physical devices found.
", 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;
}

VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
if( result != VK_SUCCESS )
{
    fprintf( FpDebug, "Could not get the physical device properties of device %d
", i );
    return VK_SHOULD_EXIT;
}
fprintf( FpDebug, " 

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

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

int which = -1;
if( discreteSelect >= 0 )
{
    which = discreteSelect;
}
else if( integratedSelect >= 0 )
{
    which = integratedSelect;
}
else
{
    fprintf( FpDebug, "Could not select a Physical Device
" );
    return VK_SHOULD_EXIT;
}
VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
```

```
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, "
%d physical devices found.
", 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;
}
```
Here’s What the NVIDIA A6000 Produced

```
Init03PhysicalDeviceAndGetQueueFamilyProperties
Device 0:
  API version: 4206797
  Driver version: 4206797
  Vendor ID: 0x10de
  Device ID: 0x2230
  Physical Device Type: 2 = (Discrete GPU)
  Device Name: NVIDIA RTX A6000
  Pipeline Cache Size: 72
Device #0 selected ('NVIDIA RTX A6000')

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

```
Init03PhysicalDeviceAndGetQueueFamilyProperties
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

std::vector<VkPhysicalDeviceMemoryProperties> phymem;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &phymem );

for( unsigned int i = 0; i < phymem.size(); i++ )
{
  VkPhysicalDeviceMemoryProperties p = phymem[i];
  printf( FpDebug, "Memory %d: ", i );
  if( ( p.memoryProperties.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT       )   != 0 )    printf( FpDebug, " DeviceLocal" );
  if( ( p.memoryProperties.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT       )     != 0 )    printf( FpDebug, " HostVisible" );
  if( ( p.memoryProperties.propertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT )    != 0 )    printf( FpDebug, " HostCoherent" );
  if( ( p.memoryProperties.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT       )   != 0 )    printf( FpDebug, " HostCached" );
  if( ( p.memoryProperties.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT ) != 0 )    printf( FpDebug, " LazilyAllocated" );
}
```

Here’s What I Got on the A6000’s

```
6 Memory Types:
  Memory 0: Memory
  Memory 1: DeviceLocal
  Memory 2: HostVisible HostCoherent
  Memory 3: HostVisible HostCoherent HostCached
  Memory 4: DeviceLocal HostVisible HostCached
  Memory 5: DeviceLocal

4 Memory Heaps:
  Heap 0: size = 0x09bb0000 DeviceLocal
  Heap 1: size = 0x0d504000
  Heap 2: size = 0x0a0b00000 DeviceLocal
  Heap 3: size = 0x000000000 DeviceLocal
```
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
fprintf(FpDebug, "Found %d Queue Families:
", 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: Queue Family Count = 16  ;   Graphics Compute Transfer
1: Queue Family Count = 2  ;   Transfer
2: Queue Family Count = 8  ;   Compute Transfer

Here's What I Got on the A6000's

Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Command Buffer

Logical Device

Command Buffer

Logical Device

Command Buffer

Logical Device

Command Buffer

Logical Device
Vulkan: a More Typical (and Simplified) Block Diagram

Looking to See What Device Layers are Available

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

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

Looking to See What Device Extensions are Available

```c
uint32_t  layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, deviceLayers[i].layerName, &layerCount, (VkLayerProperties *)nullptr);
```

What Device Layers and Extensions are Available

```c
4 physical device layers enumerated:
0x004030cd 1 "VK_LAYER_NV_optimus" 'NVIDIA Optimus layer'
160 device extensions enumerated for 'VK_LAYER_NV_optimus':
```

```c
0x00400033 1 "VK_LAYER_LUNARG_core_validation" 'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
```

```c
0x00400033 1 "VK_LAYER_LUNARG_object_tracker" 'LunarG Validation Layer'
160 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
```

```c
0x00400033 1 "VK_LAYER_LUNARG_parameter_validation" 'LunarG Validation Layer'
160 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
```
float queuePriorities[1] =
{  1.   
};

VkDeviceQueueCreateInfo vdqci;
vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci.pNext = nullptr;
vdqci.flags = 0;
vdqci.queueFamilyIndex = 0;
vdqci.queueCount = 1;
vdqci.pQueueProperties = queuePriorities;

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

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

// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue );               // 0, 0 = queueFamilyIndex, queueIndex
vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_EXT_display_surface_counter'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_get_surface_capabilities2'
0x00000009  'VK_KHR_surface'
0x00000002  'VK_KHR_win32_surface'
0x00000001  'VK_KHX_device_group_creation'
0x00000001  'VK_KHR_external_fence_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_NV_external_memory_capabilities'

vkEnumerateDeviceLayerProperties:
3 physical device layers enumerated:
0x00400038   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'

// see what layers are available:
uint32_t numLayersAvailable = sizeof(instanceLayers) / sizeof(char *);
result = vkEnumerateInstanceLayerProperties( &numLayersAvailable, (VkLayerProperties *)nullptr );
// see what extensions are available:
uint32_t numExtensionsWanted = sizeof(instanceExtensions) / sizeof(char *);
result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsWanted, (VkExtensionProperties *)nullptr );
Will now ask for 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report
vkGetPhysicalDeviceProperties(PhysicalDevice, OUT &PhysicalDeviceProperties);
vkGetPhysicalDeviceFeatures(IN PhysicalDevice, OUT &PhysicalDeviceFeatures);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp);

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

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

VkResult result;
float queuePriorities[NUM_QUEUES_WANTED] = {
  1.}
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];
vdqci[0].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1; // how many queues to create
vdqci[0].pQueuePriorities = queuePriorities; // array of queue priorities [0..1]

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

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

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

4 physical device layers enumerated:
0x00400038   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':

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, such as mat4 transformation matrices. This is a good feature, since Vulkan, at times, makes it cumbersome to send changes to the graphics.

By "small", Vulkan specifies that there will be at least 128 bytes that can be used, 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 do not live in their own GPU memory. They are actually included inside the Vulkan graphics pipeline data structure.
Creating a Graphics Pipeline Data Structure

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:

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

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

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

where:

- `stageFlags` are or’ed bits of:
  - `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`
  - `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`
  - `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
  - `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
  - `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`

- `size` is in bytes
- `pValues` is a void * pointer to the data, which, in this 4x4 matrix example, would be of type `glm::mat4`.

A Robotic Example using Push Constants

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

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

1

2

3

Ground

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

\[ \theta_1 \]

\[ \theta_2 \]

\[ \theta_3 \]

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

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

1. Rotate by \( \theta_1 \)
2. Translate by \( T_{1/G} \)
Why Do We Say it Right-to-Left?

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

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

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

Positioning Part #2 With Respect to Ground

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

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

Positioning Part #3 With Respect to Ground

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

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

In the Reset Function

struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
...

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

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

Arm3.armMatrix = glm::mat4( 1. ); // blue
Arm3.armColor = glm::vec3( 0.f, 0.f, 1.f );
Arm3.armScale = 2.f;
Set the Push Constant for the Graphics Pipeline Data Structure

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

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

VkPipelineLayoutCreateInfo
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;

vplci.setLayoutCount = 5;
vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = vpcr;

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

In the `UpdateScene( )` Function

```c
float rot1 = (float)(2.*M_PI*Time); // rotation for arm1, in radians
float rot2 = 2.f * rot1; // rotation for arm2, in radians
float rot3 = 2.f * rot2; // rotation for arm3, in radians

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

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

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

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

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

In the `RenderScene( )` Function

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

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

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

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

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

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

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

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

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

Where Synchronization Fits in the Overall Block Diagram
Semaphores

- Indicates that a batch of commands has been processed from a queue. Basically announces "I am finished!".
- You create one and give it to a Vulkan function which sets it. Later on, you tell another Vulkan function to wait for this semaphore to be signaled.
- You don't end up setting, resetting, or checking the semaphore yourself.
- Semaphores must be initialized ("created") before they can be used.

```
Ask for Something → Your program continues → Try to Use that Something
```

Creating a Semaphore

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

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

This doesn't actually do anything with the semaphore – it just sets it up.

Semaphores Example during the Render Loop

```
VkSemaphore imageReadySemaphore;

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

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

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

VkPipelineStageFlags waitAtBottomOfPipe = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

Fences

- Used to synchronize CPU-GPU tasks.
- Used when the host needs to wait for the device to complete something big.
- Announces that queue-submitted work is finished.
- You can un-signal, signal, test or block-while-waiting.
Fences

```c
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

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

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

Set the fence
Wait on the fence(s)
```

Events

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

Fence Example

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

VkSubmitInfo vsi;

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

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

result = vkQueuePresentKHR(presentQueue, IN &vsi);
```

Controlling Events from the Host

```c
vkEventCreateInfo veci;
veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;
veci.pNext = nullptr;
veci.flags = 0;

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

result = vkSetEvent(LogicalDevice, IN event);

result = vkResetEvent(LogicalDevice, IN event);

result = vkGetEventStatus(LogicalDevice, IN event);

// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled

Note: the host cannot block waiting for an event, but it can test for it
```
Controlling Events from the Device

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

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

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

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

Note: there is no problem with Read-after-Read (R-a-R) as no data gets changed.
These are the Commands that could be entered into a Command Buffer, I

vkCmdBeginConditionalRendering
vkCmdBeginDebugUtilsLabel
vkCmdBeginQuery
vkCmdBeginQueryIndexed
vkCmdBeginRendering
vkCmdBeginRenderPass
vkCmdBeginRenderPass2
vkCmdBeginTransformFeedback
vkCmdBindDescriptorSets
vkCmdBindIndexBuffer
vkCmdBindInvocationMask
vkCmdBindPipeline
vkCmdBindPipelineShaderGroup
vkCmdBindShadingRateImage
vkCmdBindTransformFeedbackBuffers
vkCmdBindVertexBuffers
vkCmdBindVertexBuffers2
vkCmdBlitImage

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

vkCmdBlitImage2
vkCmdBuildAccelerationStructure
vkCmdBuildAccelerationStructureIndirect
vkCmdClearAttachments
vkCmdClearAttachments
vkCmdClearColorImage
vkCmdClearColorImage
vkCmdCopyAccelerationStructure
vkCmdCopyAccelerationStructureToMemory
vkCmdCopyBuffer
vkCmdCopyBuffer
vkCmdCopyBufferToImage2
vkCmdCopyBufferToImage2
vkCmdCopyImage
vkCmdCopyImage
vkCmdCopyMemoryToAccelerationStructure

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

vkCmdCopyQueryPoolResults
vkCmdCuLaunchKernelX
vkCmdDebugMarkerBegin
vkCmdDebugMarkerEnd
vkCmdDispatch
vkCmdDispatchBase
vkCmdDispatchIndirect
vkCmdDraw
vkCmdDrawIndexed
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDrawIndirectByteCount
vkCmdDrawIndirectCount
vkCmdDrawMeshTasksIndirect
vkCmdDrawMulti
vkCmdDrawMultiIndexed
vkCmdEndConditionalRendering
vkCmdEndDebugUtilsLabel
vkCmdEndQuery
vkCmdEndQueryIndexed
vkCmdEndRenderPass
vkCmdEndRenderPass2
vkCmdEndTransformFeedback
vkCmdExecuteCommands
vkCmdExecuteGeneratedCommands
vkCmdFillBuffer
vkCmdInsertDebugUtilsLabel
vkCmdNextSubpass
vkCmdNextSubpass2
vkCmdPipelineBarrier
vkCmdPipelineBarrier2

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

vkCmdPreprocessGeneratedCommands
vkCmdPushConstants
vkCmdPushDescriptorSet
vkCmdPushDescriptorSetWithTemplate
vkCmdResetEvent
vkCmdResetEvent2
vkCmdResetQueryPool
vkCmdResolveImage
vkCmdResolveImage2
vkCmdSetBlendConstants
vkCmdSetCheckpoint
vkCmdSetCoarseSampleOrder
vkCmdSetCullMode
vkCmdSetDepthBias
vkCmdSetDepthBiasEnable
vkCmdSetDepthBounds
vkCmdSetDepthBoundsTestEnable
vkCmdSetDepthCompareOp
vkCmdSetDepthTestEnable
vkCmdSetDepthWriteEnable
vkCmdSetDeviceMask
vkCmdSetDiscardRectangle
vkCmdSetEvent
vkCmdSetEvent2
vkCmdSetExclusiveScissor
vkCmdSetFragmentShadingRate
vkCmdSetFrontFace
vkCmdSetLineWidth
vkCmdSetLogicOp
vkCmdSetLineStipple
vkCmdSetSampleLocations
vkCmdSetScissor
vkCmdSetScissorWithCount
vkCmdSetStencilCompareMask
vkCmdSetStencilOp
vkCmdSetStencilReference
vkCmdSetStencilTestEnable
vkCmdSetStencilWriteMask
vkCmdSetViewport
vkCmdSetViewportWithCount
vkCmdSetViewportWScaling
vkCmdSetViewportShadingRatePalette
vkCmdSetViewportWithCount
vkCmdSetViewportWScaling
vkCmdSetViewportShadingRatePalette
vkCmdSetViewportWithCount
vkCmdSubpassShading
vkCmdTraceRaysIndirect
vkCmdTraceRays
vkCmdUpdateBuffer
vkCmdWriteAccelerationStructureProperties
vkCmdWriteBufferMarker
vkCmdWriteBufferMarker
vkCmdWriteTimestamp
vkCmdWriteTimestamp2
A Pipeline Barrier is a way to establish a dependency between commands that were submitted before the barrier and commands that are submitted after the barrier.

\[
vkCmdPipelineBarrier(\text{commandBuffer}, \\
\text{srcStageMask}, \text{dstStageMask}, \\
\text{VK\_DEPENDENCY\_BY\_REGION\_BIT}, \\
\text{memoryBarrierCount}, p\text{MemoryBarriers}, \\
\text{bufferMemoryBarrierCount}, p\text{BufferMemoryBarriers}, \\
\text{imageMemoryBarrierCount}, p\text{ImageMemoryBarriers})
\]

The hope is to maximize the number of unblocked stages: produce data early and consume data late.

The Scenario

1. The cross-streets are named after pipeline stages
2. All traffic lights start out green
3. There are special sensors at all intersections that will know when any car in the src group is in that intersection
4. There are connections from those sensors to the traffic lights so that when any car in the src group is in the intersection, the proper dst traffic lights will be turned red
5. When the last car in the src group completely makes it through its intersection, the proper dst traffic lights are turned back to green
6. The Vulkan command pipeline ordering is this: (1) the src cars get released by the previous vkCmdxxx, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released by the next vkCmdyyy, (4) the dst cars stop at the red light, (5) the src cars clear the intersection, (6) the dst lights turn green, (6) the dst cars continue.

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

- VK\_PIPELINE\_STAGE\_TOP\_OF\_PIPE\_BIT
- VK\_PIPELINE\_STAGE\_DRAW\_INDIRECT\_BIT
- VK\_PIPELINE\_STAGE\_VERTEX\_INPUT\_BIT
- VK\_PIPELINE\_STAGE\_VERTEX\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_TESSELLATION\_CONTROL\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_TESSELLATION\_EVALUATION\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_GEOMETRY\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_FRAGMENT\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_EARLY\_FRAGMENT\_TESTS\_BIT
- VK\_PIPELINE\_STAGE\_LATE\_FRAGMENT\_TESTS\_BIT
- VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT
- VK\_PIPELINE\_STAGE\_COMPUTE\_SHADER\_BIT
- VK\_PIPELINE\_STAGE\_TRANSFER\_BIT
- VK\_PIPELINE\_STAGE\_BOTTOM\_OF\_PIPE\_BIT
- VK\_PIPELINE\_STAGE\_HOST\_BIT
- VK\_PIPELINE\_STAGE\_ALL\_GRAPHICS\_BIT
- VK\_PIPELINE\_STAGE\_ALL\_COMMANDS\_BIT
Pipeline Stages and what Access Operations are Allowed

Access Masks – What are you Interested in Generating or Consuming this Memory for?

Access Operations and what Pipeline Stages they can be used In
Example #1: Be sure we are done writing an Output image before using it as a Fragment Shader Texture

<table>
<thead>
<tr>
<th>Stages</th>
<th>Access types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>src _ read</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>src _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_SHADER_BIT</td>
<td>dst _ read</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>src _ read</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>src _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>src _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>src _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>src _ read</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>dst _ write</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>src _ read</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>dst _ write</td>
</tr>
</tbody>
</table>

Example #2: Setting a Pipeline Barrier so the Drawing Waits for the Compute Shader to Finish

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

const uint32 bufferMemoryBarrierCount = 1;

vkCmdPipelineBarrier(
    commandBuffer,              
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,  
    VK_DEPENDENCY_BY_REGION_BIT,  
    0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0,nullptr
);
```

Example #2: Setting a Pipeline Barrier so the Compute Shader Waits for the Drawing to Finish

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

const uint32 bufferMemoryBarrierCount = 1;

vkCmdPipelineBarrier(
    commandBuffer,              
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,  
    VK_DEPENDENCY_BY_REGION_BIT,  
    0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0, nullptr
);
```
Aliasing

"Aliasing" is a signal-processing term for "under-sampled compared with the frequencies in the signal."

What the signal really is:
what we want

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

Sampling Interval

Sampled Points
The Nyquist Criterion

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

MultiSampling

Oversampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there.

There are two approaches to this:

1. Supersampling: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. Render the image at each of these sub-pixels. **Results in the best image, but the most rendering time.**

2. Multisampling: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. If any of them pass, then perform a single color render for the one pixel and assign that single color to all the sub-pixels that passed the depth and stencil tests. **Results in a good image, with less rendering time.**

The final step is to average those sub-pixels' colors to produce one final color for this whole pixel. This is called **resolving** the pixel.

Vulkan Specification Distribution of Sampling Points within a Pixel
Consider Two Triangles That Pass Through the Same Pixel

Let's assume (for now) that the two triangles don't overlap — that is, they look this way because they butt up against each other.

Consider Two Triangles Who Pass Through the Same Pixel

Let's assume (for now) that the two triangles don't overlap — that is, they look this way because they butt up against each other.

Supersampling

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

\# Fragment Shader calls = 8

Multisampling

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

\# Fragment Shader calls = 2

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Consider Two Triangles Who Pass Through the Same Pixel

Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

A: The ideas are all still the same, but the blue one had to deal with 8 sub-pixels (instead of 5 like before). But, the red triangle came along and obsoleted 3 of those blue sub-pixels. Note that the "resolved" image will still turn out the same as before.

Consider Two Triangles Who Pass Through the Same Pixel

Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

A: The ideas are all still the same, but the blue one had to deal with 8 sub-pixels (instead of 5 like before). But, the red triangle came along and obsoleted 3 of those blue sub-pixels. Note that the "resolved" image will still turn out the same as before.

Number of Fragment Shader Calls

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Red fragment</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Multisampling and Supersampling

VkPipelineMultisampleStateCreateInfo

- `sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO`
- `pNext = nullptr`
- `flags = 0`
- `rasterizationSamples = VK_SAMPLE_COUNT_8_BIT`
- `sampleShadingEnable = VK_TRUE`
- `minSampleShading = 0.5f`
- `pSampleMask = (VkSampleMask *)nullptr`
- `alphaToCoverageEnable = VK_FALSE`
- `alphaToOneEnable = VK_FALSE`

VkGraphicsPipelineCreateInfo

- `sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO`
- `pNext = nullptr`
- `pMultisampleState = &vpmsci`

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

- vad[0].format = VK_FORMAT_B8G8R8A8_SRGB; // 24-bit color
- vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
- vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
- vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
- vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
- vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
- vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
- vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
- vad[0].flags = 0;

- vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT; // 32-bit floating-point depth
- vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
- vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
- vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
- vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
- vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
- vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
- vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
- vad[1].flags = 0;

VkAttachmentReference 

- colorReference.attachment = 0;
- colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

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

VkSubpassDescription 

- vsd.flags = 0;
- vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
- vsd.inputAttachmentCount = 0;
- vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
- vsd.colorAttachmentCount = 1;
- vsd.pColorAttachments = &colorReference;
- vsd.pResolveAttachments = (VkAttachmentReference *)nullptr;
- vsd.pDepthStencilAttachment = &depthReference;
- vsd.preserveAttachmentCount = 0;
- vsd.pPreserveAttachments = (uint32_t *)nullptr;

VkRenderPassCreateInfo 

- vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
- vrpci.pNext = nullptr;
- vrpci.flags = 0;
- vrpci.attachmentCount = 2; // color and depth/stencil
- vrpci.pAttachments = vad;

Vulkan

- vkCmdResolveImage(cmdBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, 1, &vir);
- result = vkCreateRenderPass( LogicalDevice, &vrpci, PALLOCATOR, &RenderPass );

Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

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

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

Multipass Rendering

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

mjb@cs.oregonstate.edu
Multipass Rendering uses Attachments — What is a Vulkan Attachment Anyway?

"An attachment is an image associated with a renderpass that can be used as the input or output of one or more of its subpasses."

— Vulkan Programming Guide

An attachment can be written to, read from, or both.

For example:

![Diagram showing subpass and attachments](image)

What is an Example of Wanting to do This?

There is a process in computer graphics called **Deferred Rendering**. The idea is that a game-quality fragment shader takes a long time (relatively) to execute, but, with all the 3D scene detail, a lot of the rendered fragments are going to get z-buffered away anyhow. So, why did we invoke the fragment shaders so many times when we didn’t need to?

Here’s the trick:

Let’s create a grossly simple fragment shader that writes out (into multiple framebuffers) each fragment’s:
- position (x,y,z)
- normal (nx,ny,nz)
- material color (r,g,b)
- texture coordinates (s,t)

As well as:
- the current light source positions and colors
- the current eye position

When we write these out, the final framebuffers will contain just information for the pixels that can be seen. We then make a second pass running the expensive lighting model just for those pixels. This known as the **G-buffer Algorithm**.

Back in Our Single-pass Days

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

Here comes a quick reminder of how we did that.

Afterwards, we will extend it.

![Diagram showing single-pass rendering](image)

Back in Our Single-pass Days, I

```c
VkAttachmentDescription vad[2];
vad[0].flags = 0;
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[1].flags = 0;
vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

![Code showing Vulkan attachment description](image)
## Multipass Rendering

So far, we've only performed single-pass rendering, but within a single Vulkan RenderPass, we can also have several subpasses, each of which is feeding information to the next subpass or subpasses. In this case, we will look at following up a 3D rendering with Gbuffer operations.

The Gbuffer algorithm is where you render just the depth in the first pass and use that to limit the number of calls to time-consuming fragment shaders in the second or subsequent passes.

### Attachment #1
- **Depth Attachment**
- **Gbuffer Attachments**

### Attachment #2
- **3D Rendering Pass**
- **Gbuffer Pass**
- **Lighting Pass**
- Output

### Subpass #0
- **Attachment #1**
- **Attachment #2**

### Subpass #1
- **Attachment #0**
- **Attachment #1**
- **Attachment #2**

### Subpass #2
- **Attachment #0**
- **Attachment #1**
- **Attachment #2**
VkSubpassDescription

vsd[0]
- flags = 0;
- pAttachments = (VkAttachmentReference *) nullptr;
- colorAttachmentCount = 0;
- pColorAttachments = (VkAttachmentReference *) nullptr;
- pResolveAttachments = (VkAttachmentReference *) nullptr;
- pDepthStencilAttachment = &depthOutput;
- preserveAttachmentCount = 0;
- pPreserveAttachments = (uint32_t *) nullptr;

vsd[1]
- flags = 0;
- pAttachments = (VkAttachmentReference *) nullptr;
- colorAttachmentCount = 1;
- pColorAttachments = &gBufferOutput;
- pResolveAttachments = (VkAttachmentReference *) nullptr;
- pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
- preserveAttachmentCount = 0;
- pPreserveAttachments = (uint32_t *) nullptr;

vsd[2]
- flags = 0;
- pAttachments = (VkAttachmentReference *) nullptr;
- colorAttachmentCount = 2;
- pColorAttachments = &lightingOutput;
- pResolveAttachments = (VkAttachmentReference *) nullptr;
- pDepthStencilAttachment = (VkAttachmentReference *) nullptr;
- preserveAttachmentCount = 0;
- pPreserveAttachments = (uint32_t *) nullptr;

VkSubpassDependency

vsdp[0]
- srcSubpass = 0; // depth rendering
- dstSubpass = 1; // gbuffer
- srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
- dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
- srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
- dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
- dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

vsdp[1]
- srcSubpass = 1; // gbuffer
- dstSubpass = 2; // color output
- srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
- dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
- srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
- dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
- dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;