Acknowledgements

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

Second, thanks to NVIDIA for all of their support!

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

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

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

- **2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders**

- **2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders**

- **2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders**

- **2017: OpenGL 4.6 / GLSL 4.60**

There is lots more detail at:

---

### History of Shaders

**2014: Khronos starts Vulkan effort**

- **2016: Vulkan 1.0**

- **2016: Vulkan 1.1**

- **2020: Vulkan 1.2**

There is lots more detail at:
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.
Why is it so important to keep the GPU Busy?

<table>
<thead>
<tr>
<th>Nvidia Titan V Specs vs. Titan Xp, 1080 Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Titan Y</strong></td>
</tr>
<tr>
<td><strong>GPU</strong></td>
</tr>
<tr>
<td><strong>Transistor Count</strong></td>
</tr>
<tr>
<td><strong>Flop Rate</strong></td>
</tr>
<tr>
<td><strong>CUDA Core/PES Cores (CUDA C1010/Q1000 / 1280 C1010/Q1000 / 1280)</strong></td>
</tr>
<tr>
<td><strong>TFLOPS</strong></td>
</tr>
<tr>
<td><strong>FP32</strong></td>
</tr>
<tr>
<td><strong>FP16</strong></td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
</tr>
<tr>
<td><strong>Total Power Budget (&quot;TDP&quot;)</strong></td>
</tr>
<tr>
<td><strong>Power Connectors</strong></td>
</tr>
<tr>
<td><strong>Base Price</strong></td>
</tr>
</tbody>
</table>

The Nvidia Titan V graphic card is targeted at gamers, but rather at scientific and machine learning applications. That does not, however, mean that the card is incapable of gaming, nor does it mean that we can't measure its performance metrics for gamers. The Tesla's in addition to its high-end graphics performance, are also part of the Tesla architecture and lineage. The key difference in the Tesla Y, the Tesla Y100, whereas the Tesla Y100 was available as part of a $15,000 workstation kit. The Tesla V100 adds general memory-V3 to the Tesla Y100 line, and has more memory capacity, a total of 32GB. The Tesla X100 line includes 24GB and has faster memory frequency, to be able to handle the increased memory bandwidth. Core count, for one, is 3136 CUDA cores on each GPU, with 448 tensor cores (used for Tensorflow deep learning training).
Who was the original Vulcan?

From WikiPedia:

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

Who is the Khronos Group?

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

Vulkan

- 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

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

Vertex, Normal, Color

**Model Transform** → **View Transform** → **Per-vertex Lighting** → **Projection Transform** → **Rasterization** → **Framebuffer**

Fragment Processing, Texturing, Per-fragment Lighting
The Basic Computer Graphics Pipeline, Shader-style

- **Vertex Shader**
  - `gl_Vertex`, `gl_Normal`, `gl_Color`
  - Per-vertex in variables
  - `gl_ModelViewMatrix`, `gl_ProjectionMatrix`, `gl_ModelViewProjectionMatrix`
  - Uniform Variables
  - `MC` = Model Vertex Coordinates
  - `WC` = World Vertex Coordinates
  - `EC` = Eye Vertex Coordinates

- **Fragment Shader**
  - Fragment Processing, Texturing, Per-fragment Lighting
  - Per-fragment in variables
  - Uniform Variables
  - `gl_FragColor`

The Basic Computer Graphics Pipeline, Vulkan-style

- **Vertex Shader**
  - Per-vertex in variables
  - Uniform Variables
  - `gl_Position`, Per-vertex out variables

- **Fragment Shader**
  - Output color(s)
  - Per-fragment in variables
  - Uniform Variables
Moving part of the driver into the application

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

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

Vulkan Highlights: 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
• In OpenGL, your “pipeline state” is the combination of whatever your current graphics attributes are: color, transformations, textures, shaders, etc.

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

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

• Think of the pipeline state as being immutable.

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

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

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

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

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

VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;
VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
 VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr ); // fills vmr
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = 0;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
Vulkan Quick Reference Card – I Recommend you Print This!


Vulkan Quick Reference Card

Vulkan Highlights: Overall Block Diagram

Vulkan Highlights: a More Typical Block Diagram
**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 "19" refers to the version of Visual Studio, not the year of development.

### Your Sample2019.zip File Contains This

**Table:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample.txt</td>
<td>04/13/2019 2:36 PM</td>
<td>File</td>
<td>432 KB</td>
</tr>
<tr>
<td>Sample.cpp</td>
<td>04/13/2019 2:36 PM</td>
<td>File</td>
<td>432 KB</td>
</tr>
<tr>
<td>Sample.exe</td>
<td>04/13/2019 2:36 PM</td>
<td>File</td>
<td>432 KB</td>
</tr>
</tbody>
</table>

### The Vulkan Sample Code Included with These Notes

The Vulkan sample code included with these notes is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/).

---

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

---

**Mike Bailey**
mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

---

SampleCode.pptx mjb – June 5, 2023
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
't', 'T': 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.
int 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'
" , DEBUGFILE );
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
" , Width, Height);
    Reset();
    InitGraphics();
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents();
        Time = glfwGetTime();          // elapsed time, in double-precision seconds
        UpdateScene();
        RenderScene();
    }
    fprintf(FpDebug, "Closing the GLFW window\n");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}

Main Program

void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool( );
    Init06CommandBuffers( );
InitGraphics(), II

Init07TextureSampler( &MyPuppyTexture.texSampler );
Init07TextureBufferAndFillFromFile("puppy.bmp", &MyPuppyTexture);

Init08Swapchain( );
Init09DepthStencilImage( );
Init10RenderPasses( );
Init11Framebuffers( );
Init12SpvShader( "sample-vert.spv", &ShaderModuleVertex );
Init12SpvShader( "sample-frag.spv", &ShaderModuleFragment );
Init13DescriptorSetPool( );
Init13DescriptorSetLayouts();
Init13DescriptorSets( );
Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );

A Colored Cube

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

static GLfloat CubeVertices[3][3] =
{ 
  { -1, -1, -1 }, 
  { -1, -1, 1 }, 
  { -1, 1, -1 }, 
  { -1, 1, 1 }, 
  {  1, -1, -1 }, 
  {  1, -1, 1 }, 
  {  1, 1, -1 }, 
  {  1, 1, 1 } };

static GLfloat CubeColors[3][3] =
{ 
  { 0, 0, 0 }, 
  { 1, 0, 0 }, 
  { 0, 1, 0 }, 
  { 1, 1, 0 }, 
  { 0, 0, 1 }, 
  { 1, 0, 1 }, 
  { 0, 1, 1 }, 
  { 1, 1, 1 } };
A Colored Cube

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

#include “SampleVertexData.cpp”

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

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

#include “SampleVertexData.cpp”

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

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

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

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

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

Vulkan Software Philosophy

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

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

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

Vulkan has chosen to do it like this:

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

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

**vkYyy( )** is a function call

**VK_ZZZ** is a constant

My Conventions

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

The number after “Init” gives you the ordering

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

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

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

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

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

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

```c
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, &physicalDevices[0] );
```
Double-click here to launch Visual Studio 2019 with this solution

Your Sample2019.zip File Contains This

Windows shader compiler

Linux shader compiler

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

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

ErrorCodes[] =
{
{ VK_NOT_READY, "Not Ready" },
{ VK_TIMEOUT, "Timeout" },
{ VK_EVENT_SET, "Event Set" },
{ VK_EVENT_RESET, "Event Reset" },
{ VK_INCOMPLETE, "Incomplete" },
{ VK_OUT_OF_HOST_MEMORY, "Out of Host Memory" },
{ VK_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" },
{ VK_INITIALIZATION_FAILED, "Initialization Failed" },
{ VK_DEVICE_LOST, "Device Lost" },
{ VK_MEMORY_MAP_FAILED, "Memory Map Failed" },
{ VK_LAYER_NOT_PRESENT, "Layer Not Present" },
{ VK_EXTENSION_NOT_PRESENT, "Extension Not Present" },
{ VK_FEATURE_NOT_PRESENT, "Feature Not Present" },
{ VK_INCOMPATIBLE_DRIVER, "Incompatible Driver" },
{ VK_TOO_MANY_OBJECTS, "Too Many Objects" },
{ VK_FORMAT_NOT_SUPPORTED, "Format Not Supported" },
{ VK_FRAGMENTED_POOL, "Fragmented Pool" },
{ VK_SURFACE_LOST_KHR, "Surface Lost" },
{ VK_NATIVE_WINDOW_IN_USE_KHR, "Native Window in Use" },
{ VK_SUBOPTIMAL_KHR, "Suboptimal" },
{ VK_ERROR_OUT_OF_DATE_KHR, "Error Out of Date" },
{ VK_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
{ VK_VALIDATION_FAILED_EXT, "Validation Failed" },
{ VK_INVALID_SHADER_NV, "Invalid Shader" },
{ VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" },
{ VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
};
void
PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose  &&  result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s\n", prefix.c_str(), "Successful");
        fflush(FpDebug);
        return;
    }

    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = "";
    for( int i = 0; i < numErrorCodes; i++ )
    {
        if( result == ErrorCodes[i].resultCode )
        {
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }

    fprintf( FpDebug, "\n%s: %s\n", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}

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

bool Paused;
bool Verbose;

#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
const int32_t OFFSET_ZERO = 0;
**Vulkan Program Flow – the Setup**

- Create a GLFW Vulkan Window
- 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 Vertices and Indices
- Allocate and Fill memory for the Uniform Buffers
- 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(s)
- Fill the Graphics Pipeline Data Structure(s)

**Vulkan Program Flow – the Rendering Loop**

```c
while( the GLFW Window should not close )
{
    UpdateScene();
    RenderScene();

    Create the Transformations
    Fill the Uniform Buffers
}
```

- Acquire the Next Swap Chain Image
- Begin its Command Buffer
- Create the RenderPass with the Framebuffer information
- for all the different Graphics Pipeline Data Structures being used
  - Bind that 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
- Submit the Command Buffer to a Queue
- Wait for the Queue to Finish Submitting
- Present the Image to the Viewer
```markdown
```
Vulkan Topologies

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST**
  - Graph showing point list topology with vertices $V_0$, $V_1$, $V_2$, and $V_3$.

- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST**
  - Graph showing line list topology with vertices $V_0$, $V_1$, $V_2$, and $V_3$.

- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**
  - Graph showing line strip topology with vertices $V_0$, $V_1$, $V_2$, $V_3$, $V_4$, and $V_5$.

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**
  - Graph showing triangle list topology with vertices $V_0$, $V_1$, $V_2$, and $V_3$.

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**
  - Graph showing triangle strip topology with vertices $V_0$, $V_1$, $V_2$, $V_3$, $V_4$, and $V_5$.

- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**
  - Graph showing triangle fan topology with vertices $V_0$, $V_1$, $V_2$, $V_3$, $V_4$, and $V_5$.
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;

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

A Colored Cube Example
This data is contained in the file SampleVertexData.cpp
Triangles Represented as an Array of Structures

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

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

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

Modeled in right-handed coordinates

Non-indexed Buffer Drawing

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

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

From the file `SampleVertexData.cpp`:

```
Stream of Vertices
```

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

Draw

Triangles
**Initializing and Filling the Vertex Buffer**

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

MyBuffer MyVertexDataBuffer;

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

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

VkResult Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    ... 
}
```

**A Preview of What Init05DataBuffer Does**

```c
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 ); // 0 is the offset
    return result;
}
```
We will come to the Pipeline later, but for now, know that a Vulkan pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its input.

### C/C++:
```c
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

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

### Vulkan Code:
```c
VkVertexInputBindingDescription vvibd[1];       // one of these per buffer data buffer
vvibd[0].binding = 0;                                           // which binding # this is
vvibd[0].stride = sizeof(struct vertex);            // bytes between successive structs
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX; // read one value per vertex
```

### Always use the C/C++ `offsetof()` construct rather than hardcoding the byte count!

This is because Vulkan uses the `offsetof()` macro to calculate the offset of a struct member, while C/C++ uses it to calculate the offset of a struct. This is different from OpenGL, where the offset is hardcoded in the shader.

### C/C++:
```c
glm::vec3 position;
glm::vec3 normal;
glm::vec3 color;
glm::vec2 texCoord;
```

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

### Vulkan Code:
```c
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
```

### Always use the C/C++ `offsetof()` construct rather than hardcoding the byte offset!
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 vertex input.

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

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

VkGraphicsPipelineCreateInfo vgpci;
    vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    vgpci.pNext = nullptr;
    vgpci.flags = 0;
    vgpci.stageCount = 2;                // number of shader stages in this pipeline
    vgpci.pStages = vpssci;
    vgpci.pVertexInputState = &vpvisci;
    vgpci.pInputAssemblyState = &vpiasci;
    vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci
    vgpci.pViewportState = &vpvsci;
    vgpci.pRasterizationState = &vprsci;
    vgpci.pMultisampleState = &vpmsci;
    vgpci.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 );

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

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

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

```cpp
struct vertex JustVertexData[] = 
{
    // vertex #0:
    // (-1.0, -1.0, -1.0),
    // (0.0, 0.0, 0.0),
    // (1.0, 0.0)
    // vertex #1:
    // (1.0, -1.0, -1.0),
    // (0.0, 0.0, -1.0),
    // (1.0, 0.0, 0.0),
    // (0.0, 0.0)
    // ...

    int JustIndexData[] =
    { 0, 2, 3, 0, 3, 1, 4, 5, 7, 4, 7, 6, 1, 3, 7, 1, 7, 5, 0, 4, 6, 0, 6, 2, 2, 6, 7, 2, 7, 3, 0, 1, 5, 0, 5, 4, 1, 3, 7, 1, 7, 5, 0, 4, 6, 0, 6, 2, 2, 6, 7, 2, 7, 3, 0, 1, 5, 0, 5, 4,
```

This data is contained in the file `SampleVertexData.cpp`
Drawing with an Index Buffer

```c
VkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets);
VkCmdBindIndexBuffer(commandBuffer, indexDataBuffer, indexOffset, indexType);

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

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

Drawing with an Index Buffer

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

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

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

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

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

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

Indirect Drawing (not to be confused with Indexed)

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

In Vulkan, "Indirect" means that you store the arguments in GPU memory and then give the 
vkCmdXxx call a pointer to those arguments

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

Compare this with:

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

In Vulkan, "Indirect" means that you store the arguments in GPU memory and then give the `vkCmdXxx` call a pointer to those arguments.

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

Compare this with:

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

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 (s,t) at these vertices to all be the same
“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!

We have a `vkLoadObjFile()` function to load an OBJ file into your Vulkan program!
Drawing an OBJ Object

```c
typedef struct MyBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory dstm;
    VkDeviceSize size; // in bytes
} MyBuffer;

MyBuffer MyObjBuffer; // global
...

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

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

VkBuffer objBuffer[1] = { MyObjBuffer.buffer };  
VkDeviceSize offsets[1] = { 0 }; 
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 );
```

Data Buffers

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
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.
Creating and Filling Vulkan Data Buffers

```
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'ded 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

```
"or" these bits together to specify how this buffer will be used
```
Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

VkDeviceMemory vdm;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset

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

Finding the Right Type of Memory

```cpp
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

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;
}

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

4 Memory Heaps:
Heap 0: size = 0xdbb00000 DeviceLocal
Heap 1: size = 0xfd504000
Heap 2: size = 0x0d600000 DeviceLocal
Heap 3: size = 0x02000000 DeviceLocal

These are the numbers for the Nvidia A6000 cards
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 I

struct vertex *vp;

vkMapMemory( LogicalDevice, IN 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 );

Memory-Mapped Copying to GPU Memory, Example II
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

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

VkBufferCreateInfo vbci;
...
VmaAllocationCreateInfo vaci;
  vaci.physicalDevice = PhysicalDevice;
  vaci.device = LogicalDevice;
  vaci.usage = VMA_MEMORY_USAGE_GPU_ONLY;
VmaAllocator var;
  vmaCreateAllocator( IN &vaci, OUT &var );
  ...
VkBuffer Buffer;
VmaAllocation van;
  vmaCreateBuffer( IN var, IN &vbci, IN &vaci, OUT &Buffer. OUT &van, nullptr );

void *mappedDataAddr;
  vmaMapMemory( 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:

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

// example:
```c
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.

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

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

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

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

```glsl
layout( std140, set = 1, binding = 0 ) uniform sceneBuf
{
    mat4 uProjection;
    mat4 uView;
    mat4 uSceneOrient;
    vec4 uLightPos;
    vec4 uLightColor;
    vec4 uLightKdKsKa;
    float uTime;
} Scene;

layout( std140, set = 2, binding = 0 ) uniform objectBuf
{
    mat4 uModel;
    mat4 uNormal;
    vec4 uColor;
    float uShininess;
} Object;
```

The uNormal is set to:

```c
glm::inverseTranspose( 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. ); // identity
Object.uNormal = glm::inverseTranspose( Scene.uView * Scene.uSceneOrient * Object.uModel );
```

This code assumes that this line:

```c
#define GLM_FORCE_RADIANS
```

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

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

\[
\text{struct objectBuf Object;}
\]

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

\[
\begin{align*}
\text{uniform objectBuf } & \text{ Object; } \\
\text{layout( } & \text{ std140, set = 2, binding = 0 ) uniform objectBuf } \\
& \{ \\
& \text{ mat4 } \text{ albedo; } \\
& \text{ mat4 } \text{ uNormal; } \\
& \text{ vec4 } \text{ uColor; } \\
& \text{ float } \text{ shininess; } \\
& \text{ Object; } \\
\end{align*}
\]

Filling the Data Buffer

\[
\text{typedef struct MyBuffer} \\
\{ \\
\text{ VIDDataBuffer } \text{ buffer; } \\
\text{ VIDDeviceMemory size; } & \text{ // in bytes } \\
\} \text{ MyBuffer; } \\
\ldots \\
\ldots \\
\text{ MyBuffer } \text{ MyObjectUniformBuffer; } \\
\text{Init05UniformBuffer( } \text{ sizeof(Object), } \text{ OUT } & \text{ MyObjectUniformBuffer ); } \\
\text{Fill05DataBuffer( MyObjectUniformBuffer, } \text{ IN } ( \text{ void } ) & \text{ Object ); } \\
\text{struct objectBuf } \\
\{ \\
\text{ glm: mat4 } \text{ uModel; } \\
\text{ glm: mat4 } \text{ uNormal; } \\
\text{ vec4 } \text{ uColor; } \\
\text{ float } \text{ shininess; } \\
\} \text{ Object; }
\]
Creating and Filling the Data Buffer – the Details

```c
 VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer ) {
  VkResult result = VK_SUCCESS;
  VkBufferCreateInfo vbci;
  vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  vbci.pNext = nullptr;
  vbci.flags = 0;
  vbci.size = pMyBuffer->size;
  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
  vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );

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

  VkDeviceMemory
  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

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

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 vs. Topology

**Geometry:**
Where things are (e.g., coordinates)

**Topology:**
How things are connected

Geometry = changed
Topology = same (1-2-3-4-1)

Geometry = same
Topology = changed (1-2-4-3-1)

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

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

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

V₀ – V₁ – V₂ – V₃

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

V₀ – V₁ – V₂

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

V₀ – V₁ – V₂ – V₃

Not Convex

V₀ – V₁ – V₂
What does “Convex Polygon” Mean?

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

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.
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] = {
  { 0, 2, 3 },
  { 0, 3, 1 },
  { 4, 5, 7 },
  { 4, 7, 6 },
  { 1, 3, 7 },
  { 1, 7, 5 },
  { 0, 4, 6 },
  { 0, 6, 2 },
  { 2, 6, 7 },
  { 2, 7, 3 },
  { 0, 1, 5 },
  { 0, 5, 4 }
};
```

```
static GLfloat CubeColors[9] = {
  { 0, 0, 0 },
  { 1, 0, 0 },
  { 0, 1, 0 },
  { 1, 1, 0 },
  { 0, 0, 1 },
  { 1, 0, 1 },
  { 0, 1, 1 },
  { 1, 1, 1 },
};
```

```
static GLfloat CubeVertices[8][3] = {
  { -1, -1, -1 },
  { 1, -1, -1 },
  { 1, 1, -1 },
  { -1, 1, -1 },
  { -1, -1, 1 },
  { 1, -1, 1 },
  { 1, 1, 1 },
  { -1, 1, 1 },
};
```
### Triangles in an Array of Structures

From the file SampleVertexData.cpp:

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

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

Modeled in right-handed coordinates

---

### 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**.

```cpp
VkPipelineRasterizationStateCreateInfo vprsci;
    ...;

vprsci.cullMode = VK_CULL_MODE_NONE;

vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```
Filling the Vertex Buffer

MyBuffer MyVertexDataBuffer;

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

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

A Reminder of What Init05DataBuffer Does

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

- Vertex Shader module
- Specialization info
- Vertex Input binding
- Vertex Input attributes
- Topology
- Tessellation Shaders, Geometry Shader
- Viewport
- Scissoring
- Depth Clamping
- DiscardEnable
- PolygonMode
- CullMode
- FrontFace
- LineWidth
- Which states are dynamic
- DepthTestEnable
- DepthWriteEnable
- DepthCompareOp
- StencilTestEnable
- PipelineLayoutCreateInfo

Pipeline Layout

- Vertex Shader module
- Specialization info
- Viewport
- Scissoring
- Depth Clamping
- DiscardEnable
- PolygonMode
- CullMode
- FrontFace
- LineWidth
- Which states are dynamic
- DepthTestEnable
- DepthWriteEnable
- DepthCompareOp
- StencilTestEnable

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

VkVertexInputBindingDescription vvid[1];  // one of these per buffer data buffer
vvid[0].binding = 0;  // which binding # this is
vvid[0].stride = sizeof(struct vertex);  // bytes between successive structs
vvid[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```
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.
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.

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

Telling the Command Buffer what Vertices to Draw

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

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

Don't ever hardcode the size of an array! Always get the compiler to generate it for you.

```c
const uint32_t vertexCount = 100;
```
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.

= Fixed Function

= Programmable
Vulkan Shader Stages

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

How Vulkan GLSL Differs from OpenGL GLSL

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

- In the compiler, there is an automatic
  `#define VULKAN 130`
or whatever the current version number is.
  Typically you use this like:
  ```c
  #ifdef VULKAN
     ...
  #endif
  ```

Vulkan Vertex and Instance indices:

- `gl_VertexIndex`
- `gl_InstanceIndex`

Both are 0-based

OpenGL uses:

- `gl_VertexID`
- `gl_InstanceID`

`gl_FragColor`:

- In OpenGL, `gl_FragColor` broadcasts to all color attachments
- In Vulkan, it just broadcasts to color attachment location #0
- Best idea: don’t use it at all – explicitly declare out variables to have specific location numbers:
  ```glsl
  layout ( location = 0 ) out vec4 fFragColor;
  ```
Shader combinations of separate texture data and samplers as an option:
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );

**Note:** our sample code doesn't use this.

**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

**Vulkan: Shaders’ use of Layouts for Uniform Variables**

```
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;

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

All non-sampler uniform variables must be in block buffers
**Vulkan Shader Compiling**

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

---

**Advantages:**

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

---

**SPIR-V:**

**Standard Portable Intermediate Representation for Vulkan**

```bash
```

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

(Can be overridden by the –S option)

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

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

You can run the glslangValidator program from the Windows Command Prompt, but I have found it easier to run the SPIR-V compiler from Windows-Bash.

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.

Or

BTW, within bash, if you want to list your files without that sometimes-hard-to-read filename coloring, do this:
`ls -l --color=none`

(ell-ess minus-ell minus-minus-color=none)
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:

```
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`:

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

---

Running glslangValidator.exe

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

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

Specify the SPIR-V output file

The input file. The compiler determines the shader type by the file extension:

- `.vert` Vertex shader
- `.tccs` Tessellation Control Shader
- `.tecs` Tessellation Evaluation Shader
- `.geom` Geometry shader
- `.frag` Fragment shader
- `.comp` Compute shader
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:

```
00000000 0203 0723 0000 0001 000a 0008 007e 0000
00000020 0000 0000 0011 0002 0001 0000 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
#define _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'
        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
        return VK_SHOULD_EXIT;
    }

    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );

    rewind( fp );

    unsigned char * code = new unsigned char [size];
    fread( code, size, 1, fp );

    fclose( fp );

    //...


### Reading a SPIR-V File into a Shader Module

```c
VkShaderModule ShaderModuleVertex;

VkShaderModuleCreateInfo vsmci;

vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;

vsmci.pNext = nullptr;

vsmci.flags = 0;

vsmci.codeSize = size;

vsmci.pCode = (uint32_t*)code;

VkResult result = vkCreateShaderModule(LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex);

fprintf(FpDebug, "Shader Module '%s' successfully loaded\n", filename.c_str());

delete[] code;

return result;
```

### Vulkan: Creating a Pipeline

```

vkCreateGraphicsPipeline()

```

**State Descriptors**

- **Shader stage (VERTEX, etc.)**
- **Input binding**
- **Input format**
- **Input rate**
- **Binding**
- **Location**
- **Pipeline layout**
- **RenderPass**
- **Base Pipeline Handle**
- **Base Pipeline Index**
- **Viewport State**
- **Scissor**
- **Depth stencil state**
- **Color blend state**
- **Dynamic state**
- **Pipeline color blend attachment state**
- **Pipeline dynamic state create info**
- **Array naming the states that can be set dynamically**

**State Descriptors**

- **Depth test enable**
- **Depth compare op**
- **Depth write enable**
- **Depth offset**
- **Color blend enable**
- **Src color blend factor**
- **Dst color blend factor**
- **Color blend op**
- **Src alpha blend factor**
- **Dst alpha blend factor**
- **Alpha blend op**
- **Color write mask**
You can also take a look at SPIR-V Assembly

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

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

For example, if this is your Shader Source

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

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

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

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

void main()
{
   mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
   gl_Position = PVM * vec4( aVertex, 1.);
   vNormal = Matrices.uNormalMatrix * aNormal;
   vColor = aColor;
   vTexCoord = aTexCoord;
}
```
This is the SPIR-V Assembly, Part I

Capability Shader

1. Extensons

Extension "GLSL.std.450"
MemoryModel Logical GLSL450
EntryPoint Vertex 4 "main" 34 37 48 53 56 57 61 63
Source GLSL
SourceExtension "GL_ARB_separate_shader_objects"
SourceExtension "GL_ARB_shading_language_420pack"
Name 4 "main"
Name 10 "PVM"
Name 15 "Matrices"
MemberName 13(matBuf) 0 "uModelMatrix"
MemberName 13(matBuf) 1 "uViewMatrix"
MemberName 13(matBuf) 2 "uProjectionMatrix"
MemberName 13(matBuf) 3 "uNormalMatrix"
Name 34 ""AABB"
Name 37 ""Vertices"
Name 48 ""Color"
Name 53 ""Normal"
MemberDecorate 13(matBuf) 0 ColMajor
MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 1 MatrixStride 16
MemberDecorate 13(matBuf) 1 Offset 64
MemberDecorate 13(matBuf) 2 MatrixStride 16
MemberDecorate 13(matBuf) 2 Offset 128
MemberDecorate 13(matBuf) 3 MatrixStride 16
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) Block
MemberDecorate 15(Matrices) DescriptorSet 0
MemberDecorate 15(Matrices) Binding 0
MemberDecorate 32(gl_PerVertex) 0 BuiltIn Position
MemberDecorate 32(gl_PerVertex) 1 BuiltIn PointSize
MemberDecorate 32(gl_PerVertex) 2 BuiltIn ClipDistance
MemberDecorate 32(gl_PerVertex) Block
MemberDecorate 37(aVertex) Location 0
MemberDecorate 48(vNormal) Location 0
MemberDecorate 53(aNormal) Location 1
MemberDecorate 56(vColor) Location 1
MemberDecorate 57(aColor) Location 2
MemberDecorate 61(vTexCoord) Location 2
MemberDecorate 63(aTexCoord) Location 3
MemberDecorate 65(lightBuf) Block
MemberDecorate 67(Light) DescriptorSet 1
MemberDecorate 67(Light) Binding 0

This is the SPIR-V Assembly, Part II

Decorate 15(Matrices) Binding 0
MemberDecorate 32(gl_PerVertex) 0 BuiltIn Position
MemberDecorate 32(gl_PerVertex) 1 BuiltIn PointSize
MemberDecorate 32(gl_PerVertex) 2 BuiltIn ClipDistance
MemberDecorate 32(gl_PerVertex) Block
Decorate 37(aVertex) Location 0
Decorate 48(vNormal) Location 0
Decorate 53(aNormal) Location 1
Decorate 56(vColor) Location 1
Decorate 57(aColor) Location 2
Decorate 61(vTexCoord) Location 2
Decorate 63(aTexCoord) Location 3
MemberDecorate 65(lightBuf) Block
MemberDecorate 67(Light) DescriptorSet 1
MemberDecorate 67(Light) Binding 0
2: TypeVoid
3: TypeFunction 2
6: TypeFloat 32
7: TypeVector 6(float) 4
8: TypeMatrix 7(float) 4
9: TypePointer Function 8
11: TypeVector 8(float) 3
12: TypeMatrix 11(float) 3
13(matBuf): TypeStruct 8 8 8 12
14: TypePointer Uniform 13(matBuf)
15(Matrices): 14(ptr) Variable Uniform
16: TypeInt 32 1
17: 16(int) Constant 2
18: 16(int) Constant 8
21: 16(int) Constant 1
25: 16(int) Constant 0
29: TypeInt 32 0
30: 29(int) Constant 1
31: TypeVector 8(float) 30
32(gl_PerVertex): TypeStruct 7(float) 6(float) 31
33: TypePointer Output 32(gl_PerVertex)
34: 33(ptr) Variable Output
36: TypePointer Input 11(float)
37(aVertex): TypeStruct 7(float) 6(float) 37
38: 37(ptr) Variable Input
39: 6(float) Constant 1.0
45: TypePointer Output 7(float)
47: TypePointer Output 11(float)
48(vNormal): 47(ptr) Variable Output
49: 16(int) Constant 3
50: 16(int) Constant 4
This is the SPIR-V Assembly, Part III

SPIR-V: Printing the Configuration
glslangValidator --c

6/5/2023
A Google-Wrapped Version of glslangValidator

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

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

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

1. You can #include files into your shader source

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

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

This causes a:

```
#define NUMPOINTS 4
```

to magically be inserted into the top of your source code.
Instancing – What and why?

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

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

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

But, this will only get us multiple instances of identical objects drawn on top of each other. How can we make each instance look differently?
Making each Instance look differently -- Approach #1

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

`gl_InstanceIndex` starts at 0

In the vertex shader:

```glsl
layout( std140, set = 0, binding = 0 ) uniform sporadicBuf {
    int uMode;
    int uUseLighting;
    int uNumInstances;
} Sporadic;

void main() {
    float DELTA = 3.0;
    float s = sqrt( float( Sporadic.uNumInstances ) );
    float c = ceil( float(s) );
    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;
}
```

In the vertex shader:

```glsl
int uNumInstances = 16
```
Making each Instance look differently -- Approach #2

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

Still uses \texttt{gl\_InstanceIndex}

In the vertex shader:

```cpp
layout( 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!\n" );
    exit( 1 );
  }

  glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );
  glfwWindowHint( GLFW_RESIZABLE, GLFW_FALSE );
 MainWindow = glfwCreateWindow( Width, Height, "Vulkan Sample", NULL, NULL );
  VkResult result = glfwCreateWindowSurface( Instance, MainWindow, NULL, &Surface );

  glfwSetErrorCallback( GLFWErrorCallback );
  glfwSetKeyCallback( MainWindow, GLFWKeyboard );
  glfwSetCursorPosCallback( MainWindow, GLFWMouseMotion );
  glfwSetMouseButtonCallback( MainWindow, GLFWMouseButton );
}
```
You Can Also Query What Vulkan Extensions GLFW Requires

```c
uint32_t count;
const char ** extensions = glfwGetRequiredInstanceExtensions (&count);

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

```

Found 2 GLFW Required Instance Extensions:
VK_KHR_surface
VK_KHR_win32_surface

GLFW Keyboard Callback

```c
void GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
  if( action == GLFW_PRESS )
  {
    switch( key )
    {
      //case GLFW_KEY_M:
      case 'm':
      case 'M':
      Mode++;
      if( Mode >= 2 )
      Mode = 0;
      break;
      default:
        fprintf( FpDebug, "Unknown key hit: 0x%04x = '%c\n", key, key );
        fflush(FpDebug);
    }
  }
}
```
void GLFWMouseButton( GLFWwindow *window, int button, int action, int mods )
{
  int b = 0;              // LEFT, MIDDLE, or RIGHT
  // get the proper button bit mask:
  switch( button )
  {
    case GLFW_MOUSE_BUTTON_LEFT:
      b = LEFT;               break;
    case GLFW_MOUSE_BUTTON_MIDDLE:
      b = MIDDLE;             break;
    case GLFW_MOUSE_BUTTON_RIGHT:
      b = RIGHT;              break;
    default:
      b = 0;
      fprintf( FpDebug, "Unknown mouse button: \%d\n", button );
  }
  // button down sets the bit, up clears the bit:
  if( action == GLFW_PRESS )
  {
    double xpos, ypos;
    glfwGetCursorPos( window, &xpos, &ypos);
    Xmouse = (int)xpos;
    Ymouse = (int)ypos;
    ActiveButton |= b;              // set the proper bit
  }
  else
  {
    ActiveButton &= ~b;             // clear the proper bit
  }
}

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;
}

GLFW Mouse Button Callback

GLFW Mouse Motion Callback
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    Time = glfwGetTime();          // elapsed time, in double-precision seconds
    UpdateScene( );
    RenderScene( );
}

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

Does not block – processes any waiting events, then returns

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

    glfwWaitEvents( );

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

    glfwWaitEventsTimeout( double secs );

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

    glfwPostEmptyEvent( );
What is GLM?

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

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

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

You invoke GLM like this:

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

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

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

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

```c
glMatrixMode( GL_MODELVIEW );
gLoadIdentity( );
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:

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

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

---

The Most Useful GLM Variables, Operations, and Functions

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

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

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

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

```cpp
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):

```cpp
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );
```
Here's what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

1. 

2. 
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.

```cpp
Telling Visual Studio about where the GLM folder is

GLM in the Vulkan sample.cpp Program

if( UseMouse )
{
    if( Scale < MINSIZE )
    {
        Scale = MINSIZE;
        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.uViewMatrix = glm::lookAt( eye, look, up );
    Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1f, 1000.f );
    Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL
    Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ); // note: inverseTransform!
    FillO5DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Misc.uTime = (float)Time;
    Misc.uMode = Mode;
    FillO5DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
}
```
How Does this Matrix Stuff Really Work?

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

Or, in matrix form:

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

Transformation Matrices

Translation

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

Scaling

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    \text{S}_x & 0 & 0 & 0 \\
    0 & \text{S}_y & 0 & 0 \\
    0 & 0 & \text{S}_z & 0 \\
    1 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

Rotation about X

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

Rotation about Y

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

Rotation about Z

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    \cos \theta & -\sin \theta & 0 & 0 \\
    \sin \theta & \cos \theta & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    1 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]
How it Really Works :-)  

\[
\begin{bmatrix}
\cos 90^\circ & \sin 90^\circ \\
-\sin 90^\circ & \cos 90^\circ
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2
\end{bmatrix} = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

http://xkcd.com

The Rotation Matrix for an Angle (\(\theta\)) about an Arbitrary Axis (Ax, Ay, Az)

\[
\begin{bmatrix}
A_A \cos \theta (1 - A_A) \\
A_A - \cos \theta A_A \\
A_A - \cos \theta A_A - \sin \theta A_A \\
A_A - \cos \theta A_A + \sin \theta A_A \\
A_A - \cos \theta (1 - A_A) \\
A_A - \cos \theta (A_A - \sin \theta A_A) \\
A_A - \cos \theta (A_A - \sin \theta A_A) - \sin \theta A_A \\
A_A - \cos \theta (A_A - \sin \theta A_A) + \sin \theta A_A \\
A_A - \cos \theta (A_A - \sin \theta A_A) + \sin \theta A_A + \cos \theta (1 - A_A)
\end{bmatrix}
\]

For this to be correct, \(\mathbf{A}\) must be a unit vector
Compound Transformations

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

A: We create more than one matrix.

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} = \begin{pmatrix}3 & 2 & 1 \\
\end{pmatrix} \cdot \begin{pmatrix}x \\
y \\
z
\end{pmatrix} = \begin{pmatrix}x \\
y \\
z
\end{pmatrix} \cdot \begin{pmatrix}3 \\
2 \\
1
\end{pmatrix} \cdot \begin{pmatrix}x \\
y \\
z
\end{pmatrix}
\]

Matrix Multiplication is not Commutative

Rotate, then translate

Translate, then rotate
Matrix Multiplication is Associative

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

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

From the Data Buffer Noteset

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;

vNormal = uNormalMatrix * aNormal;
gl_Position = uProjectionMatrix * uViewMatrix * uSceneMatrix * uModelMatrix * aVertex;
```

"CTM"
Why Isn't The Normal Matrix exactly the same as the Model Matrix?

\[
glm::mat4 \text{ Model } = u\text{ViewMatrix} * u\text{SceneMatrix} * u\text{ModelMatrix};
\]
\[
u\text{NormalMatrix } = \text{glm::inverseTranspose}(\text{glm::mat3(Model)});
\]

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.

Original object and normal:

Wrong!

Right!

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

```cpp
// Proposed, instead of above uniform variables
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( std140, 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.

```cpp
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
        }
    }
}
```
Descriptor Sets

Our example will assume the following shader uniform variables:

```c
// non-opaque must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform sporadicBuf
{
    int uMode;
    int uUseLighting;
    int uNumInstances;
} Sporadic;

layout( std140, set = 1, binding = 0 ) uniform sceneBuf
{
    mat4 uProjection;
    mat4 uView;
    mat4 uSceneOrient;
    vec4 uLightPos;
    vec4 uLightColor;
    vec4 uLightKaKdKs;
    float uTime;
} Scene;

layout( std140, set = 2, binding = 0 ) uniform objectBuf
{
    mat4 uModel;
    mat4 uNormal;
    vec4 uColor;
    float uShininess;
} Object;

layout( std140, set = 3, binding = 0 ) uniform sampler2D uSampler;
```

Descriptor Sets

**CPU:**
Uniform data created in a C++ data structure

```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;
```

**GPU:**
Uniform data in a "blob"*

```c
layout( std140, set = 0, binding = 0 ) uniform sporadicBuf
{
    int uMode;
    int uUseLighting;
    int uNumInstances;
} Sporadic;

layout( std140, set = 1, binding = 0 ) uniform sceneBuf
{
    mat4 uProjection;
    mat4 uView;
    mat4 uSceneOrient;
    vec4 uLightPos;
    vec4 uLightColor;
    vec4 uLightKaKdKs;
    float uTime;
} Scene;

layout( std140, set = 2, binding = 0 ) uniform objectBuf
{
    mat4 uModel;
    mat4 uNormal;
    vec4 uColor;
    float uShininess;
} Object;

layout( std140, set = 3, binding = 0 ) uniform sampler2D uSampler;
```

* "binary large object"
Step 1: Descriptor Set Pools

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

```
VkResult Init3DDescriptorSetPool ()
{
    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.

---

**SporadicSet DS Layout Binding:**

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER</td>
<td>1</td>
<td>VK_SHADER_STAGE_VERTEX_BIT</td>
</tr>
</tbody>
</table>

**SceneSet DS Layout Binding:**

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER</td>
<td>1</td>
<td>VK_SHADER_STAGE_VERTEX_BIT</td>
</tr>
</tbody>
</table>

**ObjectSet DS Layout Binding:**

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER</td>
<td>1</td>
<td>VK_SHADER_STAGE_VERTEX_BIT</td>
</tr>
</tbody>
</table>

**TexSamplerSet DS Layout Binding:**

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
<td>1</td>
<td>VK_SHADER_STAGE_FRAGMENT_BIT</td>
</tr>
</tbody>
</table>

---

```c
VkResult Init13DescriptorSetLayouts() {
    VkResult result;
    // 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_VERTEX_BIT | 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;
    uniform sampler2D uSampler;  
    vec4 rgba = texture(uSampler, vST);
```
Step 2: Define the Descriptor Set Layouts

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

SporadicSet DS Layout Binding:

binding descriptorType descriptorCount pipeline stage(s)

set = 0

bindingCount type number of that type pipeline stage(s)

vdslc0 DS Layout CI:

binding descriptorType descriptorCount pipeline stage(s)

set = 1

bindingCount type number of that type pipeline stage(s)

vdslc1 DS Layout CI:

binding descriptorType descriptorCount pipeline stage(s)

set = 2

bindingCount type number of that type pipeline stage(s)

vdslc2 DS Layout CI:

binding descriptorType descriptorCount pipeline stage(s)

set = 3

bindingCount type number of that type pipeline stage(s)

vdslc3 DS Layout CI:

Array of Descriptor Set Layouts

Pipeline Layout

VkDescriptorSetLayoutCreateInfo vdslc0

vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc0.pNext = nullptr;
vdslc0.flags = 0;
vdslc0.bindingCount = 1;
vdslc0.pBindings = &SporadicSet[0];

VkDescriptorSetLayoutCreateInfo vdslc1

vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc1.pNext = nullptr;
vdslc1.flags = 0;
vdslc1.bindingCount = 1;
vdslc1.pBindings = &SceneSet[0];

VkDescriptorSetLayoutCreateInfo vdslc2

vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc2.pNext = nullptr;
vdslc2.flags = 0;
vdslc2.bindingCount = 1;
vdslc2.pBindings = &ObjectSet[0];

VkDescriptorSetLayoutCreateInfo vdslc3

vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.pNext = nullptr;
vdslc3.flags = 0;
vdslc3.bindingCount = 1;
vdslc3.pBindings = &TexSamplerSet[0];

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

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

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

Step 4: Allocating the Memory for Descriptor Sets

```c
vkAllocateDescriptorSets( )
```
Step 4: Allocating the Memory for Descriptor Sets

```c
VkResult
Init13DescriptorSets( )
{
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;
    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );
}
```

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

- **VkDescriptorBufferInfo vdbi0:**
  - `vdbi0.buffer = MySporadicUniformBuffer.buffer;`
  - `vdbi0.offset = 0;`
  - `vdbi0.range = sizeof(Sporadic);`
  - This struct identifies what buffer it owns and how big it is.

- **VkDescriptorBufferInfo vdbi1:**
  - `vdbi1.buffer = MySceneUniformBuffer.buffer;`
  - `vdbi1.offset = 0;`
  - `vdbi1.range = sizeof(Scene);`
  - This struct identifies what buffer it owns and how big it is.

- **VkDescriptorBufferInfo vdbi2:**
  - `vdbi2.buffer = MyObjectUniformBuffer.buffer;`
  - `vdbi2.offset = 0;`
  - `vdbi2.range = sizeof(Object);`
  - This struct identifies what buffer it owns and how big it is.

- **VkDescriptorImageInfo vdii0:**
  - `vdii0.sampler = MyPuppyTexture.texSampler;`
  - `vdii0.imageView = MyPuppyTexture.texImageView;`
  - `vdii0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;`
  - This struct identifies what texture sampler and image view it owns.
Step 5: Tell the Descriptor Sets where their data is

```c
// ds 0:
VkWriteDescriptorSet vwds0;
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;
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds1.pNext = nullptr;
vwds1.dstSet = DescriptorSets[1];
vwds1.dstBinding = 0;
vwds1.dstArrayElement = 0;
vwds1.descriptorCount = 1;
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.pBufferInfo = IN &vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;

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

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

uint32_t copyCount = 0;
// this could have been done with one call and an array of VkWriteDescriptorSets:
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);
```

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.

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

This struct links a Descriptor Set to the buffer it is pointing to.
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

vkGraphicsPipelineCreateInfo

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

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

- `vgpci.stageCount = 2;`  
- `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;`  
- `vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;`  
- `vgpci.basePipelineIndex = 0;`

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

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

vkCmdBindDescriptorSets( )

- `cmdBuffer`  
- `descriptorSetCount`  
- `pipelineBindPoint`  
- `vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 4, DescriptorSets, (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
  - vkCreateDescriptorPool(): Create the pool of Descriptor Sets for future use

- VkDescriptorSetLayoutBinding
  - VkDescriptorSetLayoutCreateInfo
    - vkCreateDescriptorSetLayout(): Describe a particular Descriptor Set layout and use it in a specific Pipeline layout
    - vkCreatePipelineLayout(): Describe a particular Descriptor Set layout and use it in a specific Pipeline layout

- VkDescriptorSetAllocateInfo
  - vkAllocateDescriptorSets(): Allocate memory for particular Descriptor Sets

- VkDescriptorBufferInfo
  - VkDescriptorImageInfo
    - VkWriteDescriptorSet: Tell a particular Descriptor Set where its CPU data is
    - vkUpdateDescriptorSets(): Re-write CPU data into a particular Descriptor Set

- VkDescriptorSetAllocateInfo
  - 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.

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

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

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

Draw your polygons, specifying $s$ and $t$ at each vertex:

```c
 glBegin( GL_POLYGON );
   glVertex3f( x0, y0, z0 );
   glVertex3f( x1, y1, z1 );
   glVertex3f( x2, y2, z2 );
   glVertex3f( x3, y3, z3 );
 glEnd( );
```

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

Disable texture mapping:

```c
 glDisable( GL_TEXTURE_2D );
```
The easiest way to figure out what $s$ and $t$ are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane,

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

Or, for a sphere,

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

```cpp
s = ( lng + M_PI ) / ( 2.*M_PI );
t = ( lat + M_PI/2. ) / M_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

<table>
<thead>
<tr>
<th>CPU Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Host</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Visible</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GPU Memory</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Device</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Local</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GPU Memory</strong></td>
</tr>
</tbody>
</table>

memcpy( ) vkCmdCopyImage( )

Texture Sampling Hardware

RGBA to the Shader
## Memory Types

### NVIDIA A6000 Graphics:

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Memory 0</th>
<th>Memory 1: DeviceLocal</th>
<th>Memory 2: HostVisible HostCoherent</th>
<th>Memory 3: HostVisible HostCoherent HostCached</th>
<th>Memory 4: DeviceLocal HostVisible HostCoherent</th>
<th>Memory 5: DeviceLocal</th>
</tr>
</thead>
</table>

### Intel Integrated Graphics:

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Memory 0: DeviceLocal</th>
<th>Memory 1: DeviceLocal HostVisible HostCoherent</th>
<th>Memory 2: DeviceLocal HostVisible HostCoherent HostCached</th>
</tr>
</thead>
</table>

---

## Something I’ve Found Useful

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

```c
// 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 );

OpenGL

Vulkan

MyTexture MyPuppyTexture;
...
VkSamplerCreateInfo vsci;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    ...
result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &MyPuppyTexture->texSampler);

Textures' Undersampling Artifacts

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

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

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

*Latin: *multim in parvo*, "many things in a small place"*

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

  uint32_t texWidth = pMyTexture->width;
  uint32_t texHeight = pMyTexture->height;
  unsigned char *texture = pMyTexture->pixels;
  VkDeviceSize textureSize = texWidth * texHeight * 4;  // rgba, 1 byte each

  VkImage stagingImage;
  VkImage textureImage;

  // *******************************************************************************
  // this first {...} is to create the staging image:
  // *******************************************************************************
  VkImageCreateInfo imageCreateInfo;
  imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
  imageCreateInfo.pNext = nullptr;
  imageCreateInfo.flags = 0;
  imageCreateInfo.imageType = VK_IMAGE_TYPE_2D;
  imageCreateInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
  imageCreateInfo.extent.width = texWidth;
  imageCreateInfo.extent.height = texHeight;
  imageCreateInfo.extent.depth = 1;
  imageCreateInfo.mipLevels = 1;
  imageCreateInfo.arrayLayers = 1;
  imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
  imageCreateInfo.tiling = VK_IMAGE_TILING_LINEAR;
  #ifdef CHOICES
  VK_IMAGE_TILING_OPTIMAL
  VK_IMAGE_TILING_LINEAR
  #endif
  imageCreateInfo.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
  #ifdef CHOICES
  VK_IMAGE_USAGE_TRANSFER_SRC_BIT
  VK_IMAGE_USAGE_TRANSFER_DST_BIT
  VK_IMAGE_USAGE_SAMPLED_BIT
  VK_IMAGE_USAGE_STORAGE_BIT
  VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
  VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
  VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
  VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
  #endif
  imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

  VkMemoryRequirements memoryRequirements;
  vkGetImageMemoryRequirements(LogicalDevice, stagingImage, &memoryRequirements);

  VkMemoryAllocateInfo memoryAllocateInfo;
  memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  memoryAllocateInfo.pNext = nullptr;
  memoryAllocateInfo.allocationSize = memoryRequirements.size;

  VkDeviceMemory memory;
  VkPhysicalDeviceMemoryProperties memoryProperties;
  vkGetPhysicalDeviceMemoryProperties(LogicalDevice, &memoryProperties);
  VkMemoryPropertyFlags memoryPropertyFlags = 0;
  for (uint32_t i = 0; i < memoryProperties.memoryTypeCount; ++i)
  {
    if (memoryProperties.memoryTypes[i].propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT)
    {
      memoryPropertyFlags |= 1 << i;
    }
  }
  memoryAllocateInfo.memoryTypeIndex = FindMemoryThatIsHostVisible();

  result = vkAllocateMemory(LogicalDevice, &memoryAllocateInfo, allocator, &memory);

  stagingImage = memory;

  VkMemoryBarrier memoryBarrier;
  memoryBarrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
  memoryBarrier.pNext = nullptr;
  memoryBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
  memoryBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  memoryBarrier.srcQueueFamilyIndex = 0;
  memoryBarrier.dstQueueFamilyIndex = 0;
  memoryBarrier.srcStageMask = VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT;
  memoryBarrier.dstStageMask = VK_PIPELINE_STAGE_VERTEX_INPUT_BIT;
  memoryBarrier.srcObjectMask = 0;
  memoryBarrier.dstObjectMask = 0;

  vkUpdateMemory(LogicalDevice, stagingImage, memorySize, &memoryBarrier);

  VkImageSubresourceLayout subresourceLayout;
  vkGetImageSubresourceLayout(LogicalDevice, stagingImage, &imageCreateInfo, &subresourceLayout);

  VkImageTexelViewCreateInfo textureViewCreateInfo;
  textureViewCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_TEXEL_VIEW_CREATE_INFO;
  textureViewCreateInfo.pNext = nullptr;
  textureViewCreateInfo.flags = 0;
  textureViewCreateInfo.image = stagingImage;
  textureViewCreateInfo.mipLevel = 0;
  textureViewCreateInfo.arrayLayer = 0;
  textureViewCreateInfo.firstLayer = 0;
  textureViewCreateInfo.firstLevel = 0;
  textureViewCreateInfo.levelCount = 1;
  textureViewCreateInfo.layerCount = 1;
  textureViewCreateInfo.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

  VkImageTextureImageView textureImageView;
  textureImageView.sType = VK_STRUCTURE_TYPE_IMAGE_TEXTURE_VIEW_CREATE_INFO;
  textureImageView.pNext = nullptr;
  textureImageView.format = VK_FORMAT_R8G8B8A8_UNORM;
  textureImageView.imageAspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  textureImageView.mipLevel = 0;
  textureImageView.firstArrayLayer = 0;
  textureImageView.arrayLayerCount = 1;

  VkPipelineShaderStageCreateInfo shaderStageCreateInfo;
  shaderStageCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  shaderStageCreateInfo.pNext = nullptr;
  shaderStageCreateInfo.flags = 0;
  shaderStageCreateInfo.module = VK_NULL_HANDLE;
  shaderStageCreateInfo.entryPoint = "main";
  shaderStageCreateInfo.pipelineType = VK_PIPELINE_TYPE_GRAPHICS;
  shaderStageCreateInfo.stage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
  shaderStageCreateInfo.stages = VK_SHADER_STAGE_FRAGMENT_BIT;

  VkPipelineCreateInfo pipelineCreateInfo;
  pipelineCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_CREATE_INFO;
  pipelineCreateInfo.pNext = nullptr;
  pipelineCreateInfo.flags = 0;
  pipelineCreateInfo.layout = shaderStageCreateInfo.pipelineBindPoint;
  pipelineCreateInfo.renderPass = renderPassCreateInfo.renderPass;
  pipelineCreateInfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
  pipelineCreateInfo.lifetime = VK_PIPELINE_LIFETIME_PER:set
  pipelineCreateInfo.basePipelineHandle = VK_NULL_HANDLE;
  pipelineCreateInfo.basePipelineIndex = 0;

  VkCommandBuffer commandBuffer;
  commandBuffer = createCommandBuffer(outputSurfaceHandle, LogicalDevice, "main", VK_COMMAND_BUFFER_LEVEL_PRIMARY);

  VkGraphicsPipelineCreateInfo pipelineCreateInfo;
  pipelineCreateInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
  pipelineCreateInfo.pNext = nullptr;
  pipelineCreateInfo.flags = 0;
  pipelineCreateInfo.layout = shaderStageCreateInfo.pipelineBindPoint;
  pipelineCreateInfo.renderPass = renderPassCreateInfo.renderPass;
  pipelineCreateInfo.stages = shaderStageCreateInfo.pipelineStageFlags;

  VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo;
  pipelineLayoutCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOU

void *gpuMemory;

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

vkUnmapMemory(LogicalDevice, vdm);

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

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

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

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

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

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

    result = vkBindImageMemory(LogicalDevice, IN textureImage, IN vdm, 0);        // 0 = offset
}
// copy pixels from the staging image to the texture:
VkCommandBufferBeginInfo vcbbi;
  vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
  vcbbi.pNext = nullptr;
  vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
  vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

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

// transition the texture buffer layout:
  VkImageSubresourceRange visr;
  visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  visr.baseMipLevel = 0;
  visr.levelCount = 1;
  visr.baseArrayLayer = 0;
  visr.layerCount = 1;
  VkImageMemoryBarrier vimb;
  vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
  vimb.pNext = nullptr;
  vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
  vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
  vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
  vimb.image = textureImage;
  vimb.srcAccessMask = 0;
  vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  vimb.subresourceRange = visr;
  vkCmdPipelineBarrier(
    TextureCommandBuffer,
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
    0, (VkMemoryBarrier *)nullptr,
    0, (VkBufferMemoryBarrier *)nullptr,
    1, IN &vimb);

// now do the final image transfer:
  VkImageSubresourceLayers visl;
  visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  visl.baseArrayLayer = 0;
  visl.mipLevel = 0;
  visl.layerCount = 1;
  VkOffset3D vo3;
  vo3.x = 0;
  vo3.y = 0;
  vo3.z = 0;
  VkExtent3D ve3;
  ve3.width = texWidth;
  ve3.height = texHeight;
  ve3.depth = 1;
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 = vkQueue WaitForIdle( Queue );
// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)

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

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

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

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

The Actual
Image Data
Access to
an Image
8 bits Red
8 bits Green
8 bits Blue
8 bits Alpha

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

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

MyTexture MyPuppyTexture;

result = Init06TextureBufferAndFillFromBmpFile( "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 of 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 GPU and Driver specify the Pipeline Stages – the Vulkan Graphics Pipeline declares what goes in them.

**Vulkan Graphics Pipeline Stages and what goes into Them**

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

- **Topology**

- **Tessellation Shaders, Geometry Shader**

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

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

- **Fragment Shader module**
  - Specialization info

- **Texture Stage**
  - Rasterization
  - Dynamic State

- **Color Blending parameters**

---

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

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

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use (half-compiled SPIR-V modules)
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- 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
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, `srcColorBlendFactor`, `dstColorBlendFactor`, colorBlendOp, `srcAlphaBlendFactor`, `dstAlphaBlendFactor`, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

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

Creating a Graphics Pipeline from a lot of Pieces

```
vkCreatePipelineLayout()  // Creating a Graphics Pipeline from a lot of Pieces
vkCreatePipeline()       // Creating a Graphics Pipeline from a lot of Pieces
```

Array naming the states that can be set dynamically

```
vkGetPipelineDynamicStates()  // Creating a Graphics Pipeline from a lot of Pieces
```

Descriptor Set Layouts
- Push Constants
- `VkPipelineLayoutCreateInfo`
  - `Pipeline layout`
  - `RenderPass`
  - `basePipelineHandle`  // Creating a Graphics Pipeline from a lot of Pieces
  - `basePipelineIndex`  // Creating a Graphics Pipeline from a lot of Pieces

Shaders
- VertexInput State
  - `VkSpecializationInfo`
    - which stage (VERTEX, etc.)
  - `VkShaderModule`
  - `VkPipelineVertexInputStateCreateInfo`
    - `binding`
    - `stride`
    - `inputRate`
    - `location`
    - `binding`
    - `format`
    - `offset`

InputAssembly State
- `VkPipelineInputAssemblyStateCreateInfo`
  - `Topology`  // Creating a Graphics Pipeline from a lot of Pieces
  - `cullMode`
  - `polygonMode`
  - `frontFace`
  - `lineWidth`

Tessellation State
- `VkPipelineTessellationStateCreateInfo`

Viewport State
- `VkViewportStateCreateInfo`
  - `Viewport`
    - x, y, w, h, minDepth, maxDepth
  - `offset`
  - `extent`

Scissoring
- `VkPipelineRasterizationStateCreateInfo`
  - `Scissor`
    - x, y, w, h

Rasterization State
- `VkPipelineRasterizationStateCreateInfo`
  - `cullMode`
  - `polygonMode`
  - `frontFace`
  - `lineWidth`

MultiSample State
- `VkPipelineMultisampleStateCreateInfo`

DepthStencil State
- `VkPipelineDepthStencilStateCreateInfo`
  - `depthTestEnable`
  - `depthWriteEnable`
  - `depthCompareOp`
  - `stencilTestEnable`
  - `stencilOpStateFront`
  - `stencilOpStateBack`

ColorBlend State
- `VkPipelineColorBlendStateCreateInfo`
  - `blendEnable`
  - `srcColorBlendFactor`
  - `dstColorBlendFactor`
  - `colorBlendOp`
  - `srcAlphaBlendFactor`
  - `dstAlphaBlendFactor`
  - `alphaBlendOp`
  - `colorWriteMask`

Dynamic State
- `VkPipelineDynamicStateCreateInfo`
  - `Array naming the states that can be set dynamically`

Graphics Pipeline
- `VkPipelineColorBlendAttachmentState`
- `vkGetPipelineLayout()`  // Creating a Graphics Pipeline from a lot of Pieces
- `vkCreatePipeline()`       // Creating a Graphics Pipeline from a lot of Pieces
- `vkCreateGraphicsPipeline()`  // Creating a Graphics Pipeline from a lot of Pieces
- `vkCreatePipelineLayout()`  // Creating a Graphics Pipeline from a lot of Pieces
Creating a Typical Graphics Pipeline

```c
VkResult
Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader,
VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
    #ifdef ASSUMPTIONS
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
    vprsci.depthClampEnable = VK_FALSE;
    vprsci.polygonMode = VK_POLYGON_MODE_FILL;
    vprsci.cullMode = VK_CULL_MODE_NONE;    // best to do this because of the projectionMatrix[1][1] *= -1. ;
    vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    vprsci.rasterizerSamples = VK_SAMPLE_COUNT_ONE_BIT;
    vpcbas.blendEnable = VK_FALSE;
    vpdssci.depthEnable = VK_FALSE;
    vpdssci.depthTestEnable = VK_TRUE;
    vpdssci.depthWriteEnable = VK_TRUE;
    vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
    #endif
    . . .

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

The Shaders to Use

```c
    VkPipelineShaderStageCreateInfo
    vpssci[2];
    vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[0].pNext = nullptr;
    vpssci[0].flags = 0;
    vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
    #ifdef BITS
    VK_SHADER_STAGE_VERTEX_BIT
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
    VK_SHADER_STAGE_GEOMETRY_BIT
    VK_SHADER_STAGE_FRAGMENT_BIT
    VK_SHADER_STAGE_COMPUTE_BIT
    VK_SHADER_STAGE_ALL_GRAPHICS
    VK_SHADER_STAGE_ALL
    #endif
    vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[1].pNext = nullptr;
    vpssci[1].flags = 0;
    vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    vpssci[1].module = fragmentShader;
    vpssci[1].pName = "main";
    vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
    #ifdef BITS
    . . .
```

Use one `vpssci` array member per shader module you are using

Use one `vvibd` array member per vertex input array-of-structures you are using
VkVertexInputAttributeDescription *vviad[]; // an array containing one of these per vertex attribute in all bindings

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

#define EXTRAS_DEFINED_AT_THE_TOP

// these are here for convenience and readability:
#define VK_FORMAT_VEC4 VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_XYZW VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_VEC3 VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_STP VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_XYZ VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_VEC2 VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_ST VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_XY VK_FORMAT_R32G2_SFLOAT
#define VK_FORMAT_FLOAT VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_S VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_X VK_FORMAT_R32_SFLOAT

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

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

VkPipelineInputAssemblyStateCreateInfo *vpiasci; // declare the vertex topology
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
vpiasci.flags = 0;
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

#define CHOICES
VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY

#nife
vpiasci.pimitiveRestartEnable = VK_FALSE;

VkPipelineTessellationStateCreateInfo *vptsci; // Tessellation Shader info
vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
vptsci.pNext = nullptr;
vptsci.flags = 0;
vptsci.patchControlPoints = 0; // number of patch control points

VkPipelineGeometryStateCreateInfo *vpgsci; // Geometry Shader info
vpgsci.sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO;
vpgsci.pNext = nullptr;
vpgsci.flags = 0;
What is “Primitive Restart Enable”?  

`vpiasci.primitiveRestartEnable = VK_FALSE;`

“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.

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

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

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

![Original Image](image1)

**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.

![Scissoring Image](image2)

---

**Setting the Rasterizer State**

```c
VkPipelineRasterizationStateCreateInfo vprsci = { /* VkPipelineRasterizationStateCreateInfo structure */ }
vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
vprsci.pNext = nullptr;
vprsci.flags = 0;
vprsci.depthClampEnable = VK_FALSE;
vprsci.rasterizerDiscardEnable = VK_FALSE;
vprsci.polygonMode = VK_POLYGON_MODE_FILL;  // recommend this because of the projMatrix[1][1] *= -1.;
#ifdef CHOICES
    VK_POLYGON_MODE_FILL
    VK_POLYGON_MODE_LINE
    VK_POLYGON_MODE_POINT
#endif
vprsci.cullMode = VK_CULL_MODE_NONE;  // recommend this because of the projMatrix[1][1] *= -1.;
#ifdef CHOICES
    VK_CULL_MODE_NONE
    VK_CULL_MODE_FRONT_BIT
    VK_CULL_MODE_BACK_BIT
    VK_CULL_MODE_FRONT_AND_BACK_BIT
#endif
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
#ifdef CHOICES
    VK_FRONT_FACE_COUNTER_CLOCKWISE
    VK_FRONT_FACE_CLOCKWISE
#endif
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasClamp = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;
vprsci.lineWidth = 1.f;
```

Declare information about how the rasterization will take place.

---
What is “Depth Clamp Enable”?  

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

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

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

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

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

What is “Depth Bias Enable”?  

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

Depth Bias Enable allows scaling and translation of the Z-depth values as they come through the rasterizer to avoid Z-fighting.
VkPipelineMultisampleStateCreateInfo
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmci.pNext = nullptr;
vpmci.flags = 0;
vpmci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
vpmci.sampleShadingEnable = VK_FALSE;
vpmci.minSampleShading = 0;
vpmci.pSampleMask = (VkSampleMask *)nullptr;
vpmci.alphaToCoverageEnable = VK_FALSE;
vpmci.alphaToOneEnable = VK_FALSE;

We will discuss MultiSampling in a separate noteset.

VkPipelineColorBlendAttachmentState
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;

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

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

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

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

```cpp
VkPipelineColorBlendStateCreateInfo vpcbsci;
    vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    vpcbsci.pNext = nullptr;
    vpcbsci.flags = 0;
    vpcbsci.logicOpEnable = VK_FALSE;
    vpcbsci.logicOp = VK_LOGIC_OP_COPY;
#ifdef CHOICES
    VK_LOGIC_OP_CLEAR
    VK_LOGIC_OP_AND
    VK_LOGIC_OP_AND_REVERSE
    VK_LOGIC_OP_COPY
    VK_LOGIC_OP_AND_INVERTED
    VK_LOGIC_OP_NO_OP
    VK_LOGIC_OP_XOR
    VK_LOGIC_OP_OR
    VK_LOGIC_OP_NOR
    VK_LOGIC_OP_EQUIVALENT
    VK_LOGIC_OP_INVERT
    VK_LOGIC_OP_OR_REVERSE
    VK_LOGIC_OP_COPY_INVERTED
    VK_LOGIC_OP_OR_INVERTED
    VK_LOGIC_OP_NAND
    VK_LOGIC_OP_SET
#endif
    vpcbsci.attachmentCount = 1;
    vpcbsci.pAttachments = &vpcbas;
    vpcbsci.blendConstants[0] = 0;
    vpcbsci.blendConstants[1] = 0;
    vpcbsci.blendConstants[2] = 0;
    vpcbsci.blendConstants[3] = 0;
```

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

---

**Which Pipeline Variables can be Set Dynamically**

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

```cpp
VkPipelineDynamicStateCreateInfo vpdscci;
    vpdscci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
    vpdscci.pNext = nullptr;
    vpdscci.flags = 0;
    vpdscci.pDynamicStates = vdsi;
    vpdscci.pDynamicStatesCount = 1;
// leave turned off for now
 vpdscci.pDynamicStates = vdsi;
```

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*
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 Naive 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
}
Stencil Operations for Front and Back Faces

Using the Stencil Buffer to Perform Hidden Line Removal

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

    .compareOp = VK_COMPARE_OP_NEVER, // what to do if stencil operation succeeds

    .writeMask = 0xffffffff, // set all bits

    .reference = 0, // use zero as the reference value
};

const VkStencilOpState vsosb = { // back
    .depthFailOp = VK_STENCIL_OP_KEEP, // what to do if depth operation fails
    .failOp = VK_STENCIL_OP_KEEP, // what to do if stencil operation fails
    .passOp = VK_STENCIL_OP_KEEP, // what to do if stencil operation succeeds

    .compareOp = VK_COMPARE_OP_NEVER, // what to do if stencil operation succeeds

    .writeMask = 0xffffffff, // set all bits

    .reference = 0, // use zero as the reference value
};
```
Operations for Depth Values

```
VkPipelineDepthStencilStateCreateInfo
vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
VK_COMPARE_OP_NEVER  -- never succeeds
VK_COMPARE_OP_LESS   -- succeeds if new depth value is < the existing value
VK_COMPARE_OP_EQUAL  -- succeeds if new depth value is == the existing value
VK_COMPARE_OP_LESS_OR_EQUAL  -- succeeds if new depth value is <= the existing value
VK_COMPARE_OP_GREATER  -- succeeds if new depth value is > the existing value
VK_COMPARE_OP_NOT_EQUAL  -- succeeds if new depth value is != the existing value
VK_COMPARE_OP_GREATER_OR_EQUAL  -- succeeds if new depth value is >= the existing value
VK_COMPARE_OP_ALWAYS  -- always succeeds
#endif
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.0;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;

```

Putting it all Together! (finally…)

```
VkPipeline GraphicsPipeline; // global...

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.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 );

```
When Drawing, We will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer

```c
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:

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

 VkPipelineShaderStageCreateInfo
 VkPipelineVertexInputStateCreateInfo
 VkPipelineInputAssemblyStateCreateInfo
 VkViewportStateCreateInfo
 VkPipelineRasterizationStateCreateInfo
 VkPipelineColorBlendStateCreateInfo
 VkPipelineDepthStencilStateCreateInfo
 VkPipelineDynamicStateCreateInfo

 VkGraphicsPipelineCreateInfo

 VkPipelineShaderStageCreateInfo
 VkShaderModule
 VkVertexInputBindingDescription
 VkVertexInputAttributeDescription
 VkSpecializationInfo
 VkPipelineInputAssemblyStateCreateInfo
 VkPipelineViewportStateCreateInfo
 VkPipelineRasterizationStateCreateInfo
 VkPipelineColorBlendStateCreateInfo
 VkPipelineDepthStencilStateCreateInfo
 VkPipelineDynamicStateCreateInfo

 VkPipelineDynamicStateCreateInfo array naming the states that can be set dynamically

 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_DEPTH_BIAS
 VK_DYNAMIC_STATE_BLEND_CONSTANTS
 VK_DYNAMIC_STATE_DEPTH_BOUNDS
 VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK
 VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
 VK_DYNAMIC_STATE_STENCIL_REFERENCE
Creating a Pipeline

```c
VkDynamicState
{
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};
```

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

```c
 VkGraphicsPipelineCreateInfo vgpci;

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

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

If you declare certain state variables to be dynamic like this, then you must fill them in the command buffer! Otherwise, they are undefined.

---

Filling the Dynamic State Variables in the Command Buffer

First call:

```c
vkCmdBindPipeline( ... );
```

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

```c
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 );
```
This is from one of the Vulkan .h Files

Does this mean more Dynamic States are in the Works?

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_TEST_ENABLE = 1000267011,
Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Simplified Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer
Vulkan Queues and Command Buffers

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

```
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceFamilyProperties(PhysicalDevice, &count, OUT &vqfp, );
for( unsigned int i = 0; i < count; i++ )
{
    fprintf(FpDebug, "\t%d: Queue Family Count = %2d ; \t", 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");
}
```

For the Nvidia A6000 cards:

<table>
<thead>
<tr>
<th>Queue Family Count</th>
<th>Graphics Compute Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Similarly, we can write a function that finds the proper queue family.

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

Creating a logical device needs to know queue family information.

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

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

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

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

Creating the Command Buffers

```c
VkResult
Init06CommandBuffers()
{
    VkResult result;
    // allocate 2 command buffers for the double-buffered rendering:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 2; // 2, because of double-buffering
        result = vkAllocateCommandBuffers(LogicalDevice, &vcbai, &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, &vcbai, &TextureCommandBuffer);
    }
    return result;
}
```
Beginning a Command Buffer – One per Image

VkSemaphoreCreateInfo
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);

VkCommandBufferAllocateInfo
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
vcbbi.pNext = nullptr;

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

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

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

vkCmdCopyQueryPoolResults
vkCmdCuLaunchKernelX
vkCmdDebugMarkerBegin
vkCmdDebugMarkerEnd
vkCmdDebugMarkerInsert
vkCmdDispatch
vkCmdDispatchBase
vkCmdDispatchIndirect
vkCmdDraw
vkCmdDrawIndexed
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDispatchIndirect
vkCmdDispatch
vkCmdDispatchBase
vkCmdDraw
vkCmdDrawIndexed
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDispatchIndirect
vkCmdDispatch
vkCmdDispatchBase
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`
- `vkCmdSetDepthBoundsTestEnable`
- `vkCmdSetDepthCompareOp`
- `vkCmdSetDepthTestEnable`
- `vkCmdSetDepthWriteEnable`
- `vkCmdSetDeviceMask`
- `vkCmdSetDiscardRectangle`
- `vkCmdSetEvent`
- `vkCmdSetEvent2`
- `vkCmdSetExclusiveScissor`
- `vkCmdSetFragmentShadingRateEnum`
- `vkCmdSetFragmentShadingRate`
- `vkCmdSetFrontFace`
- `vkCmdSetLineStipple`
- `vkCmdSetLineWidth`
- `vkCmdSetLogicOp`
- `vkCmdSetPatchControlPoints`
- `vkCmdSetPrimitiveRestartEnable`
- `vkCmdSetPrimitiveTopology`
- `vkCmdSetRasterizerDiscardEnable`
- `vkCmdSetRayTracingPipelineStackSize`

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

- `vkCmdSetSampleLocations`
- `vkCmdSetScissor`
- `vkCmdSetScissorWithCount`
- `vkCmdSetStencilCompareMask`
- `vkCmdSetStencilOp`
- `vkCmdSetStencilReference`
- `vkCmdSetStencilTestEnable`
- `vkCmdSetStencilWriteMask`
- `vkCmdSetVertexInput`
- `vkCmdSetViewport`
- `vkCmdSetViewportShadingRatePalette`
- `vkCmdSetViewportWithCount`
- `vkCmdSetViewportWScaling`
- `vkCmdSubpassShading`
- `vkCmdTraceRaysIndirect2`
- `vkCmdTraceRaysIndirect`
- `vkCmdTraceRays`
- `vkCmdUpdateBuffer`
- `vkCmdWaitEvents`
- `vkCmdWaitEvents2`
- `vkCmdWriteAccelerationStructuresProperties`
- `vkCmdWriteBufferMarker2`
- `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,         // (float)Width,
  (float)Height,        // (float)Height,
  0.,                     // minDepth  
  1.                      // maxDepth
};
vkCmdSetViewport
( CommandBuffers[nextImageIndex], 0, 1, IN &viewport );         // 0=firstViewport, 1=viewportCount

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

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

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

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

vkCmdBindVertexBuffers
( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );               // 0, 1 = firstBinding, bindingCount

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

vkCmdEndRenderPass
( CommandBuffers[nextImageIndex] );

vkEndCommandBuffer
( CommandBuffers[nextImageIndex] );

Submitting a Command Buffer to a Queue for Execution

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

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

Mike Bailey
mjb@cs.oregonstate.edu

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

How OpenGL Thinks of Framebuffers
What is a Swap Chain?

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

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

- **Swap Chains are arranged as a ring buffer**
- **Swap Chains are tightly coupled to the window system.**

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

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

VulkanDebug.txt output for an Nvidia A6000:

***** Init08Swapchain *****

vkGetPhysicalDeviceSurfaceCapabilitiesKHR:

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

vkGetPhysicalDeviceSurfaceSupportKHR:

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

Found 3 Surface Formats:

0  44  0 VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
1  50  0 VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
2  64  0 VK_COLOR_SPACE_SRGB_NOLINEAR_KHR

Found 4 Present Modes:

0  2 VK_PRESENT_MODE_FIFO_KHR
1  3 VK_PRESENT_MODE_FIFO_RELAXED_KHR
2  1 VK_PRESENT_MODE_MAILBOX_KHR
3  0 VK_PRESENT_MODE_IMMEDIATE_KHR

***** Init08Swapchain *****
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 queuing of presentation requests is needed, as the requests are applied immediately.

`VkPresentModeMailboxKHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing 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 reuse by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.

`VkPresentModeFTFD_KHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing 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 in which the queue is non-empty. This is the only value of `VkPresentModeKHR` that is required to be supported.

`VkPresentModeFTFD_RelatedKHR` specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image then the presentation engine does not wait for another vertical blanking period for the update, meaning this mode may result in visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, 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.

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

`VkPresentModeKHR_S4_4_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_S8_8_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 its 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

- `vkGetDevicePhysicalSurfaceCapabilities()`: Gets the capabilities of the physical surface.
- `VkSurfaceCapabilities`: Represents the capabilities of a surface.
- `vkCreateSwapchainKHR`: Creates a swapchain.
- `VkSwapchainCreateInfoKHR`: Contains information about the swapchain.
- `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`: Gets capabilities of a physical device for a surface.
- `VkExtent2D`: Represents the size of an image.
- `vkCreateSwapchainKHR` call:
  ```c
  VkSurfaceCapabilitiesKHR vsc;
  vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
  VkExtent2D surfaceRes = vsc.currentExtent;
  VkSwapchainCreateInfoKHR vscci;
  vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
  vscci.pNext = nullptr;
  vscci.flags = 0;
  vscci.surface = Surface;
  vscci.minImageCount = 2; // double buffering
  vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
  vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
  vscci.imageExtent.width = surfaceRes.width;
  vscci.imageExtent.height = surfaceRes.height;
  vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
  vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
  vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
  vscci.imageArrayLayers = 1;
  vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
  vscci.queueFamilyIndexCount = 0;
  vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
  vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
  vscci.oldSwapchain = VK_NULL_HANDLE;
  vscci.clip = VK_TRUE;
  result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
  ```
- `vkGetSwapChainImages()`: Gets swapchain images.
- `vkCreateImageView()`: Creates an image view.

Creating a Swap Chain
Creating the Swap Chain Images and Image Views

```c
uint32_t imageCount; // # of display buffers – 2? 3?
result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr );
PresentImages = new VkImage[imageCount];
result = vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );
// present views for the double-buffering:
PresentImageViews = new VkImageView[imageCount];
for( unsigned int i = 0; i < imageCount; ++i ) {
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    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, &ALLOCATOR, OUT &PresentImageViews[i] );
}
```

Rendering into the Swap Chain, I

```c
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, &ALLOCATOR, OUT &imageReadySemaphore );
uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );
...
result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
...
vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );
vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
...
vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```
Rendering into the Swap Chain, II

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

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

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

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

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

Rendering into the Swap Chain, III

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

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

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```
Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Command Buffer

Command Buffer

Command Buffer

Querying the Number of Physical Devices

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

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

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

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

Which Physical Device to Use, I

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
    if( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not get the physical device properties of device %d
", 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.pipelineCacheSize[0] );
}
// need some logical here to decide which physical device to select:

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

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

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

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

fprintf( FpDebug, "Physical Device Features:\n" );
fprintf( FpDebug, "graphicsShader = %2d\n", PhysicalDeviceFeatures.graphicsShader );
fprintf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShader );
fprintf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect );
fprintf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines );
fprintf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport );
fprintf( FpDebug, "occlusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.occlusionQueryPrecise );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStatisticsQuery );
fprintf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64 );
fprintf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64 );
fprintf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16 );
### Here’s What the NVIDIA 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

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

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

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

Here's What I Got on the A6000's

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

4 Memory Heaps:
Heap 0: size = 0xdbb00000 DeviceLocal
Heap 1: size = 0xd504000
Heap 2: size = 0xd600000 DeviceLocal
Heap 3: size = 0x02000000 DeviceLocal
### Asking About the Physical Device's Queue Families

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

### Here's What I Got on the A6000's

<table>
<thead>
<tr>
<th>Queue Family</th>
<th>Graphics</th>
<th>Compute</th>
<th>Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Graphics</td>
<td>Compute Transfer</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Transfer</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Compute</td>
<td>Transfer</td>
<td></td>
</tr>
</tbody>
</table>
Logical Devices

Vulkan: Overall Block Diagram

Application

Instance
  Instance
    Physical Device
      Logical Device
        Queue

Physical Device
  Logical Device
    Queue

Physical Device
  Logical Device
    Queue

Physical Device
  Logical Device
    Queue

Logical Device
  Command Buffer
    Queue

Logical Device
  Command Buffer
    Queue

Logical Device
  Command Buffer
    Queue

Logical Device
  Command Buffer
    Queue
### Vulkan: a More Typical (and Simplified) Block Diagram

- **Application** → **Instance** → **Physical Device** → **Logical Device**
- **Queue** → **Command Buffer**

### Looking to See What Device Layers are Available

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

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

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

```c
// see what device extensions are available:
uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
    &extensionCount, (VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
    &extensionCount, deviceExtensions);
```

What Device Layers and Extensions are Available

```
4 physical device layers enumerated:

0x004030cd 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
    160 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'
    160 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':

0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
    160 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
```
VkDeviceCreateInfo vdci;
    vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
    vdci.pNext = nullptr;
    vdci.flags = 0;
    vdci.queueCreateInfoCount = 1;  // # of device queues
    vdci.pQueueCreateInfos = &vdqci; // array of VkDeviceQueueCreateInfo's
    vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
    vdci.ppEnabledLayerNames = myDeviceLayers;
    vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
    vdci.ppEnabledExtensionNames = myDeviceExtensions;
    vdci.pEnabledFeatures = &PhysicalDeviceFeatures;

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

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;

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

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

// get the queue for this logical device:
    vkGetDeviceQueue( LogicalDevice, 0, 0, &Queue );  // 0, 0 = queueFamilyIndex, queueIndex
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'
0x00400000 34 'VK_LAYER_RENDERDOC_Capture' 'Debugging capture layer for RenderDoc'
vkEnumerateInstanceExtensionProperties:

11 extensions enumerated:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_EXT_display_surface_counter'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_get_surface_capabilities2'
0x00000019  'VK_KHR_surface'
0x00000006  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_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'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':

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

0x00400033  1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
const char * instanceLayers[] =
{
    "VK_LAYER_LUNARG_api_dump",   // turn this on if want to see each function call and its arguments (very slow!)
    "VK_LAYER_LUNARG_core_validation",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_NV_optimus"
};

const char * instanceExtensions[] =
{
    "VK_KHR_surface",
#ifdef _WIN32
    "VK_KHR_win32_surface",
#endif
    "VK_EXT_debug_report",
};

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

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

// see what extensions are available:
uint32_t numExtensionsAvailable;
vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, (VkExtensionProperties *)nullptr );
InstanceExtensions = new VkExtensionProperties[ numExtensionsAvailable ];
result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, InstanceExtensions );

13 instance layers available:
0x00400033  2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
0x00400033  1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0x00400033  1  'VK_LAYER_LUNARG_OBJECT_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_monitor'  'Execution Monitoring Layer'
0x00400033  1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
0x00400033  1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
0x00400033  1  'VK_LAYER_GOOGLE_threading'  'Google Validation Layer'
0x00400033  1  'VK_LAYER_GOOGLE_unique_objects'  'Google Validation Layer'
0x00400033  1  'VK_LAYER_LUNARG_vktrace'  'Vktrace tracing library'
0x00400033  1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x00400033  1  'VK_LAYER_NV_night'  'NVIDIA Nsight interception layer'
0x00400033  4  'VK_LAYER_RENDERDOC_Capture'  'Debugging capture layer for RenderDoc'
11 instance extensions available:
- 0x00000008 'VK_EXT_debug_report'
- 0x00000001 'VK_EXT_display_surface_counter'
- 0x00000001 'VKKHR_get_physical_device_properties2'
- 0x00000001 'VK_KHR_get_surface_capabilities2'
- 0x00000019 'VK_KHR_surface'
- 0x00000006 'VK_KHR_win32_surface'
- 0x00000001 'VKKHR_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'

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

// create the instance, asking for the layers and extensions:
VkInstanceCreateInfo vici;
vici.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
vici.pNext = nullptr;
vici.flags = 0;
vici.pApplicationInfo = &vai;
vici.enabledLayerCount = sizeof(instanceLayers) / sizeof(char *);
vici.ppEnabledLayerNames = instanceLayers;
vici.enabledExtensionCount = extensionsWantedAndAvailable.size();
vici.ppEnabledExtensionNames = extensionsWantedAndAvailable.data();
result = vkCreateInstance(IN &vici, PALLOCATOR, OUT &Instance);
Will now ask for 3 instance extensions
   VK_KHR_surface
   VK_KHR_win32_surface
   VK_EXT_debug_report

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

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

int which = -1;
if ( discreteSelect >= 0 )
{
    which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
} else if ( integratedSelect >= 0 )
{
    which = integratedSelect;
    PhysicalDevice = physicalDevices[which];
} else
{
    fprintf( FpDebug, "Could not select a Physical Device\n" );
    return VK_SHOULD_EXIT;
}
delete[ ] physicalDevices;
vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );

vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );

vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp );

VkPhysicalDeviceMemoryProperties vpdmp;

vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );

uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( 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",
};
```c
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'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033   1  'VK_LAYER_LUNARG_core_validation'   'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00400033   1  'VK_LAYER_LUNARG_object_tracker'   'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'   'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':

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:

```cpp
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:

```cpp
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_TESSELATION_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`. 
Setting up the Push Constants for the Graphics Pipeline Data Structure

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

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

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

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

A Robotic Example using Push Constants

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

Where each arm is represented by:

```c
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

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

Positioning Part #1 With Respect to Ground

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

Code it

$$\left[ M_{1/G} \right] = \left[ T_{1/G} \right] \ast \left[ R_{\Theta_1} \right]$$

Say it
Why Do We Say it Right-to-Left?

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

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} = \begin{bmatrix} T_{1/G} \end{bmatrix} \begin{bmatrix} R_{\theta_1} \end{bmatrix} \begin{bmatrix} R_{\theta_2} \end{bmatrix}
\]

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 \(T_{1/G}\)

\[
\begin{bmatrix} M_{2/G} \end{bmatrix} = \begin{bmatrix} T_{1/G} \end{bmatrix} \begin{bmatrix} R_{\theta_1} \end{bmatrix} \begin{bmatrix} T_{2/1} \end{bmatrix} \begin{bmatrix} R_{\theta_2} \end{bmatrix}
\]

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

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

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

\[
\begin{bmatrix}
M_{3/G}
\end{bmatrix} = \begin{bmatrix}
M_{1/G}
\end{bmatrix} \ast \begin{bmatrix}
M_{2/1}
\end{bmatrix} \ast \begin{bmatrix}
M_{3/2}
\end{bmatrix}
\]

In the Reset Function

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

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

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

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

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

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

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

VkPipelineLayoutCreateInfo
vplci
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, IN &vplci, PALLOCATOR,
OUT &GraphicsPipelineLayout);
```

In the `UpdateScene()` Function

```cpp
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

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

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

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

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

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

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

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

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

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

```
in the Vertex Shader
layout( push_constant ) uniform arm
{
    mat4  armMatrix;
    vec3  armColor;
    float armScale;         // scale factor in x
} RobotArm;

layout( location = 0 ) in vec3 aVertex;

vec3 bVertex = aVertex;   // arm coordinate system is [-1., 1.] in X
bVertex.x += 1.;          // now is [0., 2.] 
bVertex.x /= 2.;          // now is [0., 1.]
bVertex.x *= (RobotArm.armScale);  // now is [0., RobotArm.armScale]
bVertex = vec3( RobotArm.armMatrix * vec4( bVertex, 1. ) );

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

Mike Bailey
mjb@cs.oregonstate.edu

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

Oregon State University
Computer Graphics
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.

Creating a Semaphore

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

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

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

```cpp
VkSemaphore imageReadySemaphore;

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

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

uint32_t nextImageIndex;
vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX,
  IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &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];
  vsi.signalSemaphoreCount = 0;
  vsi.pSignalSemaphores = (VkSemaphore) nullptr;

result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN 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, IN &vfci, PALLOCATOR, OUT &fence );
    // returns to the host right away:
    result = vkGetFenceStatus( LogicalDevice, IN fence );
        // result = VK_SUCCESS means it has signaled
        // result = VK_NOT_READY means it has not signaled

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

Fence Example

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence );
    . . .
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );
    . . .
result = vkQueuePresentKHR( presentQueue, IN &vpi );  // don't present the image until done rendering
```
Events

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

Controlling Events from the Host

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

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

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

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

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

Could be an array of events
Where signaled, where wait for the signal
Memory barriers get executed after events have been signaled

Pipeline Barriers

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Why Do We Need Pipeline Barriers?

A series of vkCmdxxx( ) calls are meant to run “flat-out”, that is, as fast as the Vulkan runtime can get them executing. But, many times, that is not desirable because the output of one command might be needed as the input to a subsequent command.

Pipeline Barriers solve this problem by declaring which stages of the hardware pipeline in subsequent vkCmdyyy( ) calls need to wait until which stages in previous vkCmdxxx( ) calls are completed.

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdBeginConditionalRendering</td>
</tr>
<tr>
<td>vkCmdBeginDebugUtilsLabel</td>
</tr>
<tr>
<td>vkCmdBeginQuery</td>
</tr>
<tr>
<td>vkCmdBeginQueryIndexed</td>
</tr>
<tr>
<td>vkCmdBeginRendering</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass2</td>
</tr>
<tr>
<td>vkCmdBeginTransformFeedback</td>
</tr>
<tr>
<td>vkCmdBindDescriptorSets</td>
</tr>
<tr>
<td>vkCmdBindIndexBuffer</td>
</tr>
<tr>
<td>vkCmdBindInvocationMask</td>
</tr>
<tr>
<td>vkCmdBindPipeline</td>
</tr>
<tr>
<td>vkCmdBindPipelineShaderGroup</td>
</tr>
<tr>
<td>vkCmdBindShadingRateImage</td>
</tr>
<tr>
<td>vkCmdBindTransformFeedbackBuffers</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffer</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffer2</td>
</tr>
<tr>
<td>vkCmdBindBlitImage</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdCopyQueryPoolResults</td>
</tr>
<tr>
<td>vkCmdCuLaunchKernelX</td>
</tr>
<tr>
<td>vkCmdDebugMarkerBegin</td>
</tr>
<tr>
<td>vkCmdDebugMarkerEnd</td>
</tr>
<tr>
<td>vkCmdDebugMarkerInsert</td>
</tr>
<tr>
<td>vkCmdDispatch</td>
</tr>
<tr>
<td>vkCmdDispatchBase</td>
</tr>
<tr>
<td>vkCmdDispatchIndirect</td>
</tr>
<tr>
<td>vkCmdDraw</td>
</tr>
<tr>
<td>vkCmdDrawIndexed</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawIndirect</td>
</tr>
<tr>
<td>vkCmdDrawIndirectByteCount</td>
</tr>
<tr>
<td>vkCmdDrawIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirect</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasks</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdBlitImage</td>
</tr>
<tr>
<td>vkCmdBuildAccelerationStructure</td>
</tr>
<tr>
<td>vkCmdBuildAccelerationStructuresIndirect</td>
</tr>
<tr>
<td>vkCmdBuildAccelerationStructures</td>
</tr>
<tr>
<td>vkCmdClearAttachments</td>
</tr>
<tr>
<td>vkCmdClearColorImage</td>
</tr>
<tr>
<td>vkCmdClearDepthStencilImage</td>
</tr>
<tr>
<td>vkCmdCopyAccelerationStructure</td>
</tr>
<tr>
<td>vkCmdCopyBuffer</td>
</tr>
<tr>
<td>vkCmdCopyBuffer2</td>
</tr>
<tr>
<td>vkCmdCopyBufferToImage</td>
</tr>
<tr>
<td>vkCmdCopyBufferToImage2</td>
</tr>
<tr>
<td>vkCmdCopyImage</td>
</tr>
<tr>
<td>vkCmdCopyImage2</td>
</tr>
<tr>
<td>vkCmdCopyImageToBuffer</td>
</tr>
<tr>
<td>vkCmdCopyImageToBuffer2</td>
</tr>
<tr>
<td>vkCmdCopyImageToAccelerationStructure</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdCopyQueryPoolResults</td>
</tr>
<tr>
<td>vkCmdCuLaunchKernelX</td>
</tr>
<tr>
<td>vkCmdDebugMarkerBegin</td>
</tr>
<tr>
<td>vkCmdDebugMarkerEnd</td>
</tr>
<tr>
<td>vkCmdDebugMarkerInsert</td>
</tr>
<tr>
<td>vkCmdDispatch</td>
</tr>
<tr>
<td>vkCmdDispatchBase</td>
</tr>
<tr>
<td>vkCmdDispatchIndirect</td>
</tr>
<tr>
<td>vkCmdDraw</td>
</tr>
<tr>
<td>vkCmdDrawIndexed</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawIndirect</td>
</tr>
<tr>
<td>vkCmdDrawIndirectByteCount</td>
</tr>
<tr>
<td>vkCmdDrawIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirect</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirectCount</td>
</tr>
<tr>
<td>vkCmdDrawMeshTasks</td>
</tr>
</tbody>
</table>
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
- vkCmdSetDepthBoundsTestEnable
- vkCmdSetDepthCompareOp
- vkCmdSetDepthTestEnable
- vkCmdSetDepthWriteEnable
- vkCmdSetDeviceMask
- vkCmdSetDiscardRectangle
- vkCmdSetEvent
- vkCmdSetEvent2
- vkCmdSetExclusiveScissor
- vkCmdSetFragmentShadingRateEnum
- vkCmdSetFragmentShadingRate
- vkCmdSetFrontFace
- vkCmdSetLineStipple
- vkCmdSetLineWidth
- vkCmdSetLogicOp
- vkCmdSetPatchControlPoints
- vkCmdSetPrimitiveRestartEnable
- vkCmdSetPrimitiveTopology
- vkCmdSetRasterizerDiscardEnable
- vkCmdSetRayTracingPipelineStackSize
- vkCmdSetSampleLocations
- vkCmdSetScissor
- vkCmdSetScissorWithCount
- vkCmdSetStencilCompareMask
- vkCmdSetStencilOp
- vkCmdSetStencilReference
- vkCmdSetStencilTestEnable
- vkCmdSetStencilWriteMask
- vkCmdSetVertexInput
- vkCmdSetViewport
- vkCmdSetViewportShadingRatePalette
- vkCmdSetViewportWithCount
- vkCmdSetViewportWScaling
- vkCmdSubpassShading
- vkCmdTraceRaysIndirect2
- vkCmdTraceRaysIndirect
- vkCmdTraceRays
- vkCmdUpdateBuffer
- vkCmdWaitEvents
- vkCmdWaitEvents2
- vkCmdWriteAccelerationStructuresProperties
- vkCmdWriteBufferMarker2
- 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( commandBuffer,
    srcStageMask, dstStageMask,
    VK_DEPENDENCY_BY_REGION_BIT,
    memoryBarrierCount, pMemoryBarriers,
    bufferMemoryBarrierCount, pBufferMemoryBarriers,
    imageMemoryBarrierCount, pImageMemoryBarriers
); 

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

The Scenario

src cars -> VERTEX_INPUT Street -> VERTEX_SHADER Street

dst cars -> FRAGMENT_SHADER Street -> COLOR_ATTACHMENT_OUTPUT Street

TOP_OF_PIPE Street

BOTTOM_OF_PIPE Street

mjb – June 5, 2023
### 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

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_TRANSFER_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

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

VK_ACCESS_INDIRECT_COMMAND_READ_BIT
VK_ACCESS_INDEX_READ_BIT
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
VK_ACCESS_UNIFORM_READ_BIT
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
VK_ACCESS_SHADER_READ_BIT
VK_ACCESS_SHADER_WRITE_BIT
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
VK_ACCESS_TRANSFER_READ_BIT
VK_ACCESS_TRANSFER_WRITE_BIT
VK_ACCESS_HOST_READ_BIT
VK_ACCESS_HOST_WRITE_BIT
VK_ACCESS_MEMORY_READ_BIT
VK_ACCESS_MEMORY_WRITE_BIT
### Pipeline Stages and what Access Operations are Allowed

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Access Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>VK_ACCESS_INDEX_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>VK_ACCESS_SHADER_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER</td>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>VK_ACCESS_HOST_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_MEMORY_READ_BIT</td>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_MEMORY_WRITE_BIT</td>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Access Operations</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>VK_PIPELINE_STAGE_MEMORY_READ_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_MEMORY_WRITE_BIT</td>
</tr>
</tbody>
</table>
Example #1: Be sure we are done writing an Output image before using it as a Fragment Shader Texture

Stages

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_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_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Access types

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

Example #1: The Scenario

src cars are generating the image

dst cars are waiting to use that image as a texture
Example #2: Setting a Pipeline Barrier so the Drawing Waits for the Compute Shader to Finish

```c
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 =
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_COMPUTE_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

```c
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 =
vbmb.offset = 0;
vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);
const uint32 bufferMemoryBarrierCount = 1;

vkCmdPipelineBarrier
{
    commandBuffer,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
    VK_DEPENDENCY_BY_REGION_BIT,
    0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0,nullptr
};
```
Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

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

Aliasing

The Display We Want

Too often, the Display We Get
"Aliasing" is a signal-processing term for "under-sampled compared with the frequencies in the signal".

What the signal really is:
what we want

Sampling Interval

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

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

#### Diagrams

- **VK_SAMPLE_COUNT_1_BIT**
- **VK_SAMPLE_COUNT_2_BIT**
- **VK_SAMPLE_COUNT_4_BIT**
- **VK_SAMPLE_COUNT_8_BIT**
- **VK_SAMPLE_COUNT_16_BIT**

#### Table

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.375, 0.125)</td>
<td>(0.375, 0.125)</td>
<td>(0.375, 0.125)</td>
<td>(0.375, 0.125)</td>
</tr>
<tr>
<td>(0.4375, 0.6875)</td>
<td>(0.4375, 0.6875)</td>
<td>(0.4375, 0.6875)</td>
<td>(0.4375, 0.6875)</td>
</tr>
<tr>
<td>(0.25, 0.25)</td>
<td>(0.25, 0.25)</td>
<td>(0.25, 0.25)</td>
<td>(0.25, 0.25)</td>
</tr>
<tr>
<td>(0.3125, 0.1875)</td>
<td>(0.3125, 0.1875)</td>
<td>(0.3125, 0.1875)</td>
<td>(0.3125, 0.1875)</td>
</tr>
<tr>
<td>(0.625, 0.875)</td>
<td>(0.625, 0.875)</td>
<td>(0.625, 0.875)</td>
<td>(0.625, 0.875)</td>
</tr>
<tr>
<td>(0.6875, 0.9375)</td>
<td>(0.6875, 0.9375)</td>
<td>(0.6875, 0.9375)</td>
<td>(0.6875, 0.9375)</td>
</tr>
<tr>
<td>(0.0, 0.5)</td>
<td>(0.0, 0.5)</td>
<td>(0.0, 0.5)</td>
<td>(0.0, 0.5)</td>
</tr>
<tr>
<td>(0.25, 0.125)</td>
<td>(0.25, 0.125)</td>
<td>(0.25, 0.125)</td>
<td>(0.25, 0.125)</td>
</tr>
<tr>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
</tr>
<tr>
<td>(0.9375, 0.25)</td>
<td>(0.9375, 0.25)</td>
<td>(0.9375, 0.25)</td>
<td>(0.9375, 0.25)</td>
</tr>
<tr>
<td>(0.875, 0.1875)</td>
<td>(0.875, 0.1875)</td>
<td>(0.875, 0.1875)</td>
<td>(0.875, 0.1875)</td>
</tr>
<tr>
<td>(0.8575, 0.8125)</td>
<td>(0.8575, 0.8125)</td>
<td>(0.8575, 0.8125)</td>
<td>(0.8575, 0.8125)</td>
</tr>
<tr>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
<td>(0.5, 0.0625)</td>
</tr>
<tr>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
</tr>
<tr>
<td>(0.0625, 0.25)</td>
<td>(0.0625, 0.25)</td>
<td>(0.0625, 0.25)</td>
<td>(0.0625, 0.25)</td>
</tr>
<tr>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
<td>(0.9375, 0.0625)</td>
</tr>
<tr>
<td>(0.0625, 0.0)</td>
<td>(0.0625, 0.0)</td>
<td>(0.0625, 0.0)</td>
<td>(0.0625, 0.0)</td>
</tr>
</tbody>
</table>
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.

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

# Fragment Shader calls = 8
Consider Two Triangles Who Pass Through the Same Pixel

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

<table>
<thead>
<tr>
<th>Number of Fragment Shader Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
</tr>
</tbody>
</table>

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

\# Fragment Shader calls = 2
Consider Two Triangles Who Pass Through the Same Pixel

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

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

Consider Two Triangles Who Pass Through the Same Pixel

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

<table>
<thead>
<tr>
<th>Number of Fragment Shader Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multisampling</strong></td>
</tr>
<tr>
<td>Blue fragment shader calls</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
</tr>
</tbody>
</table>
VkPipelineMultisampleStateCreateInfo
vpmsci;

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

VkGraphicsPipelineCreateInfo
vgpci;

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

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

---

At least this fraction of samples will get their own fragment shader calls (as long as they pass the depth and stencil tests).

0. produces simple multisampling
(0. - 1.) produces partial supersampling
1. Produces complete supersampling
Setting up the Image

```c
VkAttachmentDescription vad[];
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;
    colorReference.attachment = 0;
    colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

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

Setting up the Image

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

VkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.pAttachments = vad;
vrpci.subpassCount = 1;
vrpci.pSubpasses = IN &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkSubpassDependency *)nullptr;

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

```c
VIOffset3D
  vo3.x = 0;
  vo3.y = 0;
  vo3.z = 0;

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

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

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

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

For the ImageLayout, use VK_IMAGE_LAYOUT_GENERAL.
Multipass Rendering uses Attachments -- What is a Vulkan Attachment Anyway?

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

-- Vulkan Programming Guide

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

For example:

```
Attachment

Attachment

Subpass  Subpass  Subpass  Framebuffer
```

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.

```
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;

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

VkAttachmentReference depthReference;
  depthReference.attachment = 1;
  depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```
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.
Multipass, I

VkAttachmentDescription

<table>
<thead>
<tr>
<th>Attachment</th>
<th>Flags</th>
<th>Format</th>
<th>Samples</th>
<th>LoadOp</th>
<th>StoreOp</th>
<th>StencilLoadOp</th>
<th>StencilStoreOp</th>
<th>InitialLayout</th>
<th>FinalLayout</th>
</tr>
</thead>
<tbody>
<tr>
<td>vad[0]</td>
<td>0</td>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_DONT_CARE</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
</tr>
<tr>
<td>vad[1]</td>
<td>0</td>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_DONT_CARE</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
</tr>
<tr>
<td>vad[2]</td>
<td>0</td>
<td>VK_FORMAT_R8G8B8A8_SRGB</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_OPAQUE</td>
<td>VK_ATTACHMENT_LOAD_OP_DONT_CARE</td>
<td>VK_ATTACHMENT_STORE_OP_OPAQUE</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
<td>VK_IMAGE_LAYOUT_PRESENT_SRC</td>
</tr>
</tbody>
</table>

Multiplicity, II

VkAttachmentReference

- depthOutput
  - attachment = 0
  - layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL

- gBufferInput
  - attachment = 0
  - layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

- gBufferOutput
  - attachment = 1
  - layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

- lightingInput[2]
  - attachment = 0
  - layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL

- lightingInput[1]
  - attachment = 1
  - layout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

- lightingOutput
  - attachment = 2
  - layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
VkSubpassDescription vsd[3];
vsd[0].flags = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[0].inputAttachmentCount = 0;
vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
vsd[0].colorAttachmentCount = 0;
vsd[0].pColorAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pDepthStencilAttachment = &depthOutput;
vsd[0].preserveAttachmentCount = 0;
vsd[0].pPreserveAttachments = (uint32_t *)nullptr;
vsd[1].flags = 0;
vsd[1].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[1].inputAttachmentCount = 0;
vsd[1].pInputAttachments = (VkAttachmentReference *)nullptr;
vsd[1].colorAttachmentCount = 1;
vsd[1].pColorAttachments = &gBufferOutput;
vsd[1].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[1].pDepthStencilAttachment = (VkAttachmentReference *)nullptr;
vsd[1].preserveAttachmentCount = 0;
vsd[1].pPreserveAttachments = (uint32_t *)nullptr;
vsd[2].flags = 0;
vsd[2].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[2].inputAttachmentCount = 2;
vsd[2].pInputAttachments = &lightingInput[0];
vsd[2].colorAttachmentCount = 1;
vsd[2].pColorAttachments = &lightingOutput;
vsd[2].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[2].pDepthStencilAttachment = (VkAttachmentReference *)nullptr;
vsd[2].preserveAttachmentCount = 0;
vsd[2].pPreserveAttachments = (uint32_t *)nullptr;

VkSubpassDependency vsdp[2];
vsdp[0].srcSubpass = 0; // depth rendering →
vsdp[0].dstSubpass = 1; // → gbuffer
vsdp[0].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[0].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vsdp[0].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[0].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
vsdp[1].srcSubpass = 1; // gbuffer →
vsdp[1].dstSubpass = 2; // → color output
vsdp[1].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
vsdp[1].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vsdp[1].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
vsdp[1].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vsdp[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

Notice how similar this is to creating a Directed Acyclic Graph (DAG).
VkRenderPassCreateInfo
vrpci

vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrcip.pNext = nullptr;
vrcip.flags = 0;
vrcip.attachmentCount = 3; // depth, gbuffer, output
vrpci.pAttachments = vad;
vrcip.subpassCount = 3;
vrcip.pSubpasses = vsd;
vrcip.dependencyCount = 2;
vrcip.pDependencies = vsdp;

result = vkCreateRenderPass(LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass);

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

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

vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);

// subpass #1 is started here
vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);

// subpass #2 is started here
vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);