The Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

Course Goals

• Better understand what Vulkan is and when it would be a good application solution (and when it wouldn’t)
• Leave you with working, understandable Vulkan code
• Understand the inner workings of Vulkan, especially the parts that are different from OpenGL and Direct X 11

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 always for your courage and patience!

Second, thanks to NVIDIA for all of your support! These courses could not have ever happened without you!

Third, thanks to the Khronos Group for the great Vulkan teaching materials and other swag! (Look at those happy faces in the photo holding the reference cards.)
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?

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 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

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.

Vulkan Reference Card

Even though we are up to Vulkan 1.3, the Reference Card is 1.1


Vulkan Highlights: Overall Block Diagram

Khronos Group
Vulkan Highlights: a More Typical Block Diagram

Application
  ↓
Instance
  ↓
Physical Device
  ↓
Logical Device
  ↓
Command Buffer
  ↓
Command Buffer
  ↓
Command Buffer

Steps in Creating Graphics using Vulkan

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

The Vulkan Sample Code Included with These Notes

Sample Program Output

Sample Program Keyboard Inputs

Y (‘ll), ‘L’: Toggle lighting off and on
Y’, ‘M’: Toggle display mode (textures vs. colors, for now)
Y’, ‘P’: Pause the animation
Y’, ‘Q’: quit the program
Esc: quit the program
Y’, ‘R’: Toggle rotation-animation and using the mouse
Y’, ‘T’: Toggle using a vertex buffer only vs. an index buffer (in the index buffer 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’d a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan_core.h into the code as comments to show you what certain options could be.
10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

```c
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
");
    }
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```

InitGraphics( ), I

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

InitGraphics( ), II

Vulkan Software Philosophy

Vulkan has lots of typedefs that define C/C++ structs and enums.
Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary information into a function. For example, where we might normally say, using C++ class methods:
```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 InitXXXYYY( ) functions in numerical order. After that comes the helper functions

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

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

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

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &, count,
result = vkEnumeratePhysicalDevices( Instance, &count,
result = vkEnumeratePhysicalDevices( Instance, &count, &physicalDevices[0],

Where to put them
How many total
there are

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

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

Your Sample2019-COLORED-CUBE.zip File Contains This

Linux shader compiler
Windows shader compiler
Double-click here to launch Visual Studio 2019 with this solution

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

Vulkan Program Flow – the Setup

Create a GLFW Vulkan Window
Create the Queue(s) (1 in our case)
Create the Graphics Pipeline Data Structure Layout(s)

Create the Logical Device
Allocate and Fill memory for the Uniform Buffers
Allocate and Fill memory for the Vertices and Indices
Create the Command Buffers (3 in our case)

If using Textures, create the Sampler, Read the Texture, and move it to Device Local Memory
Create the Swap Chain (2 images in our case)

Be sure you have Compiled the Shaders into .spv files
Create the Descriptor Set Data Structures
Fill the Graphics Pipeline Data Structure(s)

Vulkan Program Flow – the Rendering Loop

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

Create the Transformations
Fill the Uniform Buffers
Begin its Command Buffer
Create the RenderPass with the Framebuffer information
Acquire the Next Swap Chain Image
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

Drawing
Vulkan Topologies

**VK_PRIMITIVE_TOPOLOGY_POINT_LIST**

```
V0 V1 V2 V3
```

**VK_PRIMITIVE_TOPOLOGY_LINE_LIST**

```
V0 V1 V3 V2
```

**VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**

```
V0 V2 V1 V3
```

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**

```
V0 V1 V3 V4
```

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**

```
V0 V5 V4 V6
```

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**

```
V0 V1 V2 V3 V4
```


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


### A Colored Cube Example

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

### Triangles Represented as an Array of Structures

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

### Non-indexed Buffer Drawing

Stream of Vertices

```
<table>
<thead>
<tr>
<th>Vertex 0</th>
<th>Vertex 1</th>
<th>Vertex 2</th>
<th>Vertex 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.7, 0.7, 0.7)</td>
<td>(0.7, 0.3, 0.3)</td>
<td>(0.7, 0.3, 0.7)</td>
<td>(0.7, 0.3, 0.7)</td>
</tr>
<tr>
<td>(1.0, 0.7, 0.7)</td>
<td>(1.0, 0.7, 0.3)</td>
<td>(1.0, 0.7, 0.7)</td>
<td>(1.0, 0.7, 0.7)</td>
</tr>
<tr>
<td>(0.7, 0.7, 0.3)</td>
<td>(0.7, 0.3, 0.7)</td>
<td>(0.7, 0.3, 0.7)</td>
<td>(0.7, 0.3, 0.7)</td>
</tr>
</tbody>
</table>
```

### Initializing and Filling the Vertex Buffer

```
struct vertex VertexData[] =
{
    // triangle 0-2-3:
    // vertex #0:
    { (0.7, 0.7, 0.7), (0.7, 0.3, 0.3), (0.7, 0.3, 0.7), (0.7, 0.3, 0.7) },
    // vertex #2:
    { (0.7, 0.7, 0.3), (0.7, 0.3, 0.7), (0.7, 0.3, 0.7), (0.7, 0.3, 0.7) },
    // vertex #3:
    { (0.7, 0.7, 0.7), (0.7, 0.3, 0.7), (0.7, 0.3, 0.7), (0.7, 0.3, 0.7) },
};
```
A Preview of What Init05DataBuffer Does

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.

GLSL Shader:

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

C/C++:

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

Always use the C/C++ struct rather than hardcoding the byte count!

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ struct rather than hardcoding the byte count!
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.
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.

But, to Draw that Terrain Surface, You Need “Primitive Restart”

“Primitive Restart” is used with:
- Indexed drawing
- TRIANGLE_FAN and TRIANGLE_STRIP topologies

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

```
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 `vkUloadObjFile()` function to load an OBJ file into your Vulkan program!

Drawing an OBJ Object

```
MyBuffer MyObjBuffer; // global

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

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 );
```

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

From the Reference Card

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

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:

```c
typedef VkBuffer VkDataBuffer;
```

This is probably a bad idea in the long run.

### Terminology Issues

**Creating and Filling Vulkan Data Buffers**

```c
void* 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 = (VK_USAGE_TRANSFER_SRC_BIT |
vkUsageTransferDstBit |
vkUsageUniformTexelBufferBit |
vkUsageStorageTexelBufferBit |
vkUsageUniformBufferBit |
vkUsageStorageBufferBit |
vkUsageIndexBufferBit |
vkUsageVertexBufferBit |
vkUsageIndirectBufferBit)
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const int32_t) nullptr;

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

**Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer**

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

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

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

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

**Finding the Right Type of Memory**

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

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

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

Memory Heaps:

- Heap 0: size = 0x00b00000 DeviceLocal
- Heap 1: size = 0x00500000 DeviceLocal
- Heap 2: size = 0x00600000 DeviceLocal
- Heap 3: size = 0x02000000 DeviceLocal

Memory-Mapped Copying to GPU Memory, Example I

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

Memory-Mapped Copying to GPU Memory, Example II

```c
struct vertex *vp;
vkMapMemory(LogicalDevice, 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);
```

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. I am including its Github link here and a little sample code in case you want to take a peek.

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

This repository also includes a smattering of documentation.

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

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

void *mappedDataAddr;
void *mappedDataAddr;
void *mappedDataAddr;
```

Something I've Found Useful

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

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

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

The uNormal is set to:

```
glm::inverseTranspose(uView * uSceneOrient * uModel)
```

In the vertex shader, each object vertex gets transformed by:
```
uProjection* uView * uSceneOrient * uModel
```

Filling those Uniform Variables

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

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

This C struct is holding the data buffer

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

Filling the Data Buffer

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

Creating and Filling the Data Buffer – the Details
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);
    // 0 and 0 are offset and flags
    memcpy(pGpuMemory, data, (size_t)myBuffer.size);
    vkUnmapMemory(LogicalDevice, IN myBuffer.vdm);
    return VK_SUCCESS;
}
```

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

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
- `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:
    ```c
    layout (location = 0) out vec4 fFragColor;
    ```

The Shaders’ View of the Basic Computer Graphics Pipeline

Shader stages
- `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_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`

Vulkan Shader Stages

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

Not: our sample code doesn’t use this.
**SPIR-V:** Standard Portable Intermediate Representation for Vulkan

You can 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 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.

Advantages:

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

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

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


Or, follow these instructions:

1. Head to the Start menu search bar, type in 'terminal,' and launch the Windows Terminal as administrator. (On some systems, this is called 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.

Advantages:

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

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

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


Or, follow these instructions:

1. Head to the Start menu search bar, type in 'terminal,' and launch the Windows Terminal as administrator. (On some systems, this is called 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.

Advantages:

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

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

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


Or, follow these instructions:

1. Head to the Start menu search bar, type in 'terminal,' and launch the Windows Terminal as administrator. (On some systems, this is called 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.
Vulkan: Creating a Graphics Pipeline Data Structure

SPIR-V: More Information

A Google-Wrapped Version of glslangValidator

Instancing

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:

```
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;
}
```
Making each instance look differently -- Approach #2

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

Still uses gl_InstanceIndex

In the vertex shader:

```glsl
layout( std140, set = 4, binding = 0 ) uniform colorBuf {
vec3 uColors[1024];
}

out vec3 vColor;

int index = gl_InstanceIndex % 1024; // gives 0 - 1023
vColor = colorBuf.uColors[ index ];

vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
```

In the vertex shader:

```glsl
struct atom {
vec3 position;
int atomicNumber;
}
layout( std140, set = 4, binding = 0 ) uniform moleculeBuf {
atom atoms[24];
}

void main( ) {
mat4  P = Scene.uProjection;
mat4  V = Scene.uView;
mat4  SO = Scene.uSceneOrient;
mat4  M = Object.uModel;
mat4 VM = V * SO * M;
mat4 PVM = P * VM;

vColor = aColor;
vTexCoord = aTexCoord;
vec4 ECposition = VM * vec4(aVertex, 1.);
vec4 lightPos = vec4(Scene.uLightPos.xyz, 1.);             // light source in fixed location because not transformed
vL = normalize(lightPos.xyz - ECposition.xyz);             // vector from the point to the light
vec4 eyePos = vec4(0., 0., 0., 1.);                                  // eye position after applying the viewing matrix
vE = normalize(eyePos.xyz - ECposition.xyz);              // vector from the point to the eye

int atomicNumber = atoms[gl_InstanceIndex].atomicNumber;
vec3 position = atoms[gl_InstanceIndex].position;
float radius;
if( atomicNumber == 1 ) {
    radius = 0.37;
    vColor = vec3(1., 1., 1.);
}
else if( atomicNumber == 6 ) {
    radius = 0.77;
    vColor = vec3(0., 1., 0.);
}
else if( atomicNumber == 7 ) {
    radius = 0.70;
    vColor = vec3(0., 0., 1.);
}
else if( atomicNumber == 8 ) {
    radius = 0.66;
    vColor = vec3(1., 0., 0.);
}
else {
    radius = 0.75;
    vColor = vec3(1., 0., 1.);    // big magenta ball to tell us something is wrong
}
vec3 bVertex = aVertex;
bVertex.xyz *= radius;
bVertex.xyz += position;

gl_Position = PVM * vec4(bVertex, 1.);
```
You Can Also Query What Vulkan Extensions GLFW Requires

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

Found 2 GLFW Required Instance Extensions:
VK_KHR_surface
VK_KHR_win32_surface

GLFW Keyboard Callback

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'
", key );
                fflush(FpDebug);
        }
    }
}

GLFW Mouse Button Callback

void
GLFWMouseButton( GLFWwindow *window, int button, int action, int mods )
{
    int b = 0;              // LEFT, MIDDLE, or RIGHT
    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
", button );
    }
    if( action == GLFW_PRESS ) {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;
        Ymouse = (int)ypos;
        ActiveButton |= b;              // set the proper bit
    } else {
        ActiveButton &= ~b;             // clear the proper bit
    }
}

GLFW Mouse Motion Callback

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

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 return

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( );

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 invoke GLM like this:
#define    GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include  <glm/gtc/matrix_transform.hpp>
#include  <glm/gtc/matrix_inverse.hpp>

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

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

OpenGL treats all angles as given in degrees. This line forces GLM to treat all angles as given in radians.

I recommend this so that all angles you create in all programming will be in radians.

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:
glm::mat4 modelview = glm::mat4(). // 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}

You would now say:
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}

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

The Most Useful GLM Variables, Operations, and Functions

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

If multiplications:
glm::mat4 * glm::mat4;     // promote a vec3 to a vec4 via a constructor

If emulating OpenGL transformations with concatenation:
glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
The Most Useful GLM Variables, Operations, and Functions

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

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

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

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

// viewing (assign, not concatenate):
glm::mat4 glm::lookAt(glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up);

GLM in the Vulkan sample.cpp Program

vNormal = uNormalMatrix * aNormal;

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

From the Data Buffer Noteset

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

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

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

Here are the UBO variables that connect to the shader data:

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

Glsl code for the vertex shader:

descriptorSet = descriptorSet[scene];

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?

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

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.

What are Descriptor Sets?

std140
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.

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

GPU: Uniform data used in the shader

- Float
- Mat
- Vec
- Struct
- Sampler

GPU: Uniform data in a “blob”

- * “binary large object”
- * “blob”

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.

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.

**Example code: C++ data structure**

```cpp
struct UniformData {
    float shininess;
    glm::mat4 model;
    float time;
    glm::vec4 lightKaKdKs;
    glm::vec4 lightColor;
    glm::mat4 view;
    glm::mat4 projection;
    int numInstances;
    int useLighting;
    int mode;
};
```

**Example code: Creating a Descriptor Set Layout**

```cpp
vk::DescriptorSetLayoutBuilder layout;
{ // DS #0:
    layout.setBinding(0, vk::DescriptorType::UniformBuffer, 1);
    layout.setFlags(vk::DescriptorSetLayoutFlagBits::NonOpaque);
    layout.setPImmutableSamplers(nullptr);
}
```

**Example code: Creating a Descriptor Pool**

```cpp
vk::DescriptorPoolBuilder poolBuilder;
{ // poolSizeCount:
    poolBuilder.setPoolSize(0, 1, vk::DescriptorType::UniformBuffer);
    poolBuilder.setPoolSize(1, 1, vk::DescriptorType::StorageBuffer);
}
```

**Example code: Creating a Descriptor Pool with the layout**

```cpp
vk::DescriptorPoolBuilder poolBuilder;
{ // poolSizeCount:
    poolBuilder.setPoolSize(0, 1, vk::DescriptorType::UniformBuffer);
    poolBuilder.setPoolSize(1, 1, vk::DescriptorType::StorageBuffer);
    poolBuilder.setPoolSize(2, 1, vk::DescriptorType::UniformBuffer);
    poolBuilder.setPoolSize(3, 1, vk::DescriptorType::UniformBuffer);
}
```

**Example code: Creating a Descriptor Set**

```cpp
vk::DescriptorSetBuilder setBuilder;
setBuilder.withLayout(layout); // Use the layout from the pool
setBuilder.withPool(pool); // Use the pool
{ // set 0:
    setBuilder.setBinding(0, 1); // 1 descriptor
}
```

**Example code: Using the Descriptor Set**

```cpp
vk::PipelineBuilder pipelineBuilder;
{ // useUniform
    pipelineBuilder.useUniformBufferSet(set, 0); // Use the uniform buffer set
}
```
Step 2: Define the Descriptor Set Layouts

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

void Init14GraphicsPipelineLayout( )
{
    // SporadicSet DS Layout Binding:
    result = vkCreateDescriptorSetLayout( LogicalDevice,
        IN &vdsai, PALLOCATOR, OUT &DescriptorSetLayouts[0] );
    DescriptorSetPool = DescriptorSetLayouts[0];

    // SceneSet DS Layout Binding:
    result = vkCreateDescriptorSetLayout( LogicalDevice,
        IN &vdsai, PALLOCATOR, OUT &DescriptorSetLayouts[1] );

    // ObjectSet DS Layout Binding:
    result = vkCreateDescriptorSetLayout( LogicalDevice,
        IN &vdsai, PALLOCATOR, OUT &DescriptorSetLayouts[2] );

    // TexSamplerSet DS Layout Binding:
    result = vkCreateDescriptorSetLayout( LogicalDevice,
        IN &vdsai, PALLOCATOR, OUT &DescriptorSetLayouts[3] );

    // Pipeline Layout
    vplci = new VkPipelineLayoutCreateInfo; // malloc
    vplci.pNext = nullptr;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.setLayoutCount = 4;
    vplci.flags = 0;

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

    // globals:
}

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

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

void Init14GraphicPipelineLayout( )
{
    // results:
    result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, OUT &PipelineLayout );

    // SporadicSet DS Layout Binding:
    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );

    // SceneSet DS Layout Binding:
    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[1] );

    // ObjectSet DS Layout Binding:
    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[2] );

    // TexSamplerSet DS Layout Binding:
    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[3] );

    // globals:
}

Step 4: Allocating the Memory for Descriptor Sets

// globals:
VkDescriptorPool DescriptorPool;

void Init14DescriptorSets( )
{
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.pNext = nullptr;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pSetLayouts = &DescriptorSetLayouts[0];
    vdsai.descriptorSetCount = 1;
    vdsai.descriptorPool = DescriptorPool;

    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );

    // globals:
}

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

// globals:
VkDescriptorPool DescriptorPool;

void Init14DescriptorSets( )
{
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.pNext = nullptr;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pSetLayouts = &DescriptorSetLayouts[0];
    vdsai.descriptorSetCount = 1;
    vdsai.descriptorPool = DescriptorPool;

    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );

    // globals:
}
Step 5: Tell the Descriptor Sets where their CPU data is

```c
// this could have been done with one call and an array of VkWriteDescriptorSets:
VmaBufferMemoryBarrierEXT
voladin, 1, IN
vdbi2, IN
flushes, IN
flushes, IN
copyCount, IN
(VkCopyDescriptorSet *)nullptr );
and use it in a specific
Pipeline layout

Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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

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

The Pipeline Data Structure

Sidebar: The Entire Descriptor Set Journey

Create the pool of Descriptor Sets for future use

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

Allocate memory for particular Descriptor Sets

Tell a particular Descriptor Set where its CPU data is

Re-write CPU data into a particular Descriptor Set

Make a particular Descriptor Set current for rendering.
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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

Textures

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.

Enable texture mapping:

```glEnable(GL_TEXTURE_2D);```

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

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

```glEnd();```

Disable texture mapping:

```glDisable(GL_TEXTURE_2D);```

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

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

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

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

Or, for a sphere,

\[
s = \frac{\theta - (-\pi)}{2\pi}
\]

\[
t = \frac{\phi - (-\frac{\pi}{2})}{\pi}
\]

\[
s = (\text{lng} + \text{M\_PI}) / (2\times\text{M\_PI})
\]

\[
t = (\text{lat} + \text{M\_PI} / 2) / \text{M\_PI}
\]

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…,

\[
s = ?
\]

\[
t = ?
\]

Memory Types

CPU Memory

GPU Memory

Host Visible

Device Local

GPU Memory

CUDA to the Shader

GPU Memory

Host Visible

Host Coherent

Memory Types

NVIDIA A6000 Graphics:

6 Memory Types:

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

Intel Integrated Graphics:

3 Memory Types:

Memory 0: DeviceLocal
Memory 1: DeviceLocal HostVisible HostCoherent
Memory 2: DeviceLocal HostVisible HostCoherent HostCached

Texture Sampling Parameters

OpenGL

Vulkan

// holds all the information about a data buffer so it can be encapsulated in one variable:

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

// holds all the information about a texture so it can be encapsulated in one variable:

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

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

Something I’ve Found Useful

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);

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 left over in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

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

Texture Mip-mapping

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

\* Latin: rufid in pavo, "many things in a small place"
// transition the texture buffer layout a second time:

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

result = vkEndCommandBuffer( TextureCommandBuffer );

VkMemoryAllocateInfo vmai;
result = vkAllocateMemory( LogicalDevice, IN textureImage, OUT &vmr);
fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );
fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment );

vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
vici.queueFamilyIndexCount = 0;
vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
vici.tiling = VK_IMAGE_TILING_OPTIMAL;
vici.samples = VK_SAMPLE_COUNT_1_BIT;
vici.arrayLayers = 1;
vici.mipLevels = 1;
vici.extent.depth = 1;
vici.extent.height = texHeight;
vici.extent.width = texWidth;
vici.flags = 0;
vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;

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

visl.layerCount = 1;
visl.baseArrayLayer = 0;
visl.baseMipLevel = 0;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vimb.subresourceRange = visr;
visr.layerCount = 1;
visr.baseArrayLayer = 0;
visr.baseMipLevel = 0;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vkCreateImageMemoryBarrier( TextureCommandBuffer,
    VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0,
    1, IN &vimb);

vkCreateImageMemoryBarrier( TextureCommandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
    1, IN &vimb);

vkBindImageMemory( LogicalDevice, IN textureImage, IN vdm, 0 );

vkGetImageMemoryRequirements( LogicalDevice, IN textureImage, OUT &vmr);
Reading in a Texture from a BMP File

```
MyTexture MyPuppyTexture;

result = InitTextureBufferAndFillFromBmpFile( "puppy1.bmp", &MyPuppyTexture);
```

The 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 GIMP’s GIMP.

The 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. You know the OpenGL state, so the Vulkan Graphics Pipeline is not the processes that OpenGL
would call "the graphics pipeline".
4. The Vulkan Graphics Pipeline is the combination of state variables into a Pipeline, that Pipeline never gets changed. To make
new combinations of state variables, create a new GPDS.
5. The shaders get compiled in the way when their Graphics Pipeline Data Structure gets created.

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

A Graphics Pipeline Data Structure Contains the Following State Items:

- **Pipeline Layout**: The layout of the Descriptor Sets and information on the Push Constants need to be supplied.
- **Vertex Input Stage**: x, y, w, h
- **Input Assembly**: x, y, w, h
- **Vertex Shader module**: x, y, w, h
- **Geometry Shader**: x, y, w, h
- **Tessellation Shaders, Geometry Shader**: x, y, w, h
- **Fragment Shader stage**: x, y, w, h
- **Dynamic State**: which states can be set dynamically (bound to the command buffer, outside the Pipeline)
- **Blending**: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorWriteMask
- **Depth**: depthTestEnable, depthWriteEnable, depthCompareOp
- **Rasterization**: cullMode, polygonMode, frontFace, lineWidth
- **Assembly**: topology (e.g., VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST)
- **Viewport**: x, y, w, h
- **Color Clamping**: x, y, w, h
- **Viewport**: x, y, w, h
- **Scissoring**: x, y, w, h
- **Color**: x, y, w, h
- **Depth Clamping**: x, y, w, h
- **StencilEnable, stencilFront, stencilBack**: x, y, w, h
- **AlphaEnable, lineWidth, colorWriteMask**: x, y, w, h
- **DepthWriteEnable, stencilCompareOpFront, stencilCompareOpBack**: x, y, w, h
- **AlphaBlendEnable, depthTestEnable, depthWriteEnable, depthCompareOp**: x, y, w, h
- **BlendEnable, alphaBlendFactor, dstAlphaBlendFactor, srcAlphaBlendFactor, colorBlendFactor**: x, y, w, h
- **Dynamic State**: 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 Typical Graphics Pipeline

The Shaders to Use

Options for vplsci.topology
vpiasci.primitiveRestartEnable = VK_FALSE;

**What is “Primitive Restart Enable”?**

“Restart Enable” is used with:
- Indexed drawing.
- TRAPEZOID, FAN and TRAPEZOID_STRIP topologies

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

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

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 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:
Viewpointing operates on vertices and takes place right before the rasterizer. Changing the vertical part of the viewport causes the entire scene to get scaled (squished) into the viewport area.

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

---

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
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.

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Operations for Depth Values

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

```
VkPipeline GraphicsPipeline; // global

vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
```

Queues and Command Buffers

Application

Instance

Physical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Physical Device

Physical Device

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Vulkan Queues and Command Buffers

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

```
CPU Thread
CPU Thread
CPU Thread
CPU Thread

Cmd buffer

queue

GPU Thread
GPU Thread
GPU Thread
GPU Thread

Cmd buffer

queue
```

Querying what Queue Families are Available

```
uint32_t count;

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

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];

vkGetPhysicalDeviceFamilyProperties(PhysicalDevice, &count, OUT &vqfp);

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

Found 3 Queue Families:
0: Queue Family Count = 16 ; Graphics Compute Transfer
1: Queue Family Count = 2 ; Transfer
2: Queue Family Count = 8 ; Compute Transfer

For the Nvidia A6000 cards:
Similarly, we can write a function that finds the proper queue family:

```c
int FindQueueFamilyThatDoesGraphics()
{
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
    vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, &count, 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
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pQueueCreateInfos = &vdqci[0];
vdci.queueCreateInfoCount = 1;  // one entry per queueCount
vdci.pEnabledFeatures = PhysicalDeviceFeatures;    // already created
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

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

Creating the Command Pool as part of the Logical Device:

```c
VkCommandPoolCreateInfo vcpci;
vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
                 | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
vcpci.pNext = nullptr;
```

Creating the Command Buffers:

```c
VkResult result = vkCreateSemaphore(LogicalDevice, &vsci, pAllocator, &imageReadySemaphore);
```

Beginning a Command Buffer – One per Image:

```c
vkBeginCommandBuffer(CommandBuffers[0], &vcbbi);
```

Beginning a Command Buffer

```c
vkBeginCommandBuffer(CommandBuffers[1], &vcbbi);
```
These are the Commands that could be entered into a Command Buffer, I

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

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

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

How the RenderScene() Function Works
VkViewport viewport = {
    0., // x
    0., // y
    (float)Width,
    (float)Height,
    0., // minDepth
    1. // maxDepth
};

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

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

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

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

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

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

VkDeviceSize offsets[1] = { 0 };

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

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

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);

VkSubmitInfo vsi;

vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

vsi.pNext = nullptr;

vsi.commandBufferCount = 1;

vsi.pCommandBuffers = &CommandBuffer;

vsi.waitSemaphoreCount = 1;

vsi.pWaitSemaphores = imageReadySemaphore;

vsi.signalSemaphoreCount = 0;

vsi.pSignalSemaphores = (VkSemaphore *)nullptr;

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

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

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

VkPresentInfoKHR vpi;

vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

vpi.pNext = nullptr;

vpi.waitSemaphoreCount = 0;

vpi.pWaitSemaphores = (VkSemaphore *)nullptr;

vpi.swapchainCount = 1;

vpi.pSwapchains = &SwapChain;

vpi.pImageIndices = &nextImageIndex;

vpi.pResults = (VkResult *)nullptr;

vkQueuePresentKHR(presentQueue, &vpi);

The Entire Submission / Wait / Display Process

Submitting a Command Buffer to a Queue for Execution

The Swap Chain

How OpenGL Thinks of Framebuffers

How Vulkan Thinks of Framebuffers – the Swap Chain
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

We Need to Find Out What our Display Capabilities Are

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

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

Creating a Swap Chain
Creating a Swap Chain

```c
vkCreateSwapchainKHR
```

```
result = vkCreateSwapchainKHR( LogicalDevice, Surface, OUT &SwapChain );
```

```
VkSurfaceCapabilitiesKHR vscc;
```

```
vsc.currentExtent = vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
```

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

```
VkSemaphoreCreateInfo vsci;
```

```
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
```

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

```
VkSwapchainCreateInfoKHR vscci;
```

```
vscci.clipped = VK_TRUE;
```

```
vscci.oldSwapchain = VK_NULL_HANDLE;
```

```
vscci.presentMode = VK_PRESENT_MODE_FIFO_KHR;
```

```
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
```

```
vscci.queueFamilyIndexCount = 0;
```

```
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

```
vscci.imageArrayLayers = 1;
```

```
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

```
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
```

```
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

```
vscci.imageExtent.width = surfaceRes.width;
```

```
vscci.imageExtent.height = surfaceRes.height;
```

```
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
```

```
vscci.imageFormat = VK_FORMAT_RGB8_SRGB;
```

```
vscci.minImageCount = 3;
```

```
vscci.surface = Surface;
```

```
vscci.flags = 0;
```

```
vscci.pNext = nullptr;
```

```
vpi.pResults = (VkResult *) nullptr;
```

```
vpi.pImageIndices = &imageCount;
```

```
vpi.pSwapchains = &SwapChain;
```

```
vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
```

```
vpi.waitSemaphoreCount = 0;
```

```
vpi.pNext = nullptr;
```

```
```
Rendering into the Swap Chain, I

```
Rendering into the Swap Chain, II

```
Rendering into the Swap Chain, III

```
```
Logical Device

Queue
Instance
Physical Device

Querying the Number of Physical Devices

```
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );
```

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

```
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
    int integratedSelect = -1;
    int discreteSelect = -1;
```

Which Physical Device to Use, I

```
if( result != VK_SUCCESS )
    result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
```

Which Physical Device to Use, II

```
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
    fprintf( FpDebug, " (CPU)\n" );
else if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
    fprintf( FpDebug, " (Virtual GPU)\n" );
else if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
    fprintf( FpDebug, " (Integrated GPU)\n" );
else if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
    fprintf( FpDebug, " (Discrete GPU)\n" );
```

Vulkan: Overall Block Diagram

```
Vulkan: a More Typical (and Simplified) Block Diagram
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```

```
Vulkan: Identifying the Physical Devices
```
Asking About the Physical Device's Features

Here's What I Got on the Nvidia A6000

6/5/2023
Found 3 Queue Families:
0: Queue Family Count = 16  ;   Graphics Compute Transfer
1: Queue Family Count =  2  ;   Transfer
2: Queue Family Count =  8  ;   Compute Transfer

Here's What I Got on the Nvidia A6000

Logical Devices

Application
  Instance
    Physical Device
      Logical Device
        Queue
          Command Buffer

Vulkan: Overall Block Diagram

Application
  Instance
    Physical Device
      Logical Device
        Queue
          Command Buffer

Looking to See What Device Layers are Available

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

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

```
// see what device layers are available:
uint32_t  layerCount;
if (vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr) == VK_SUCCESS)
{
  VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
  uint32_t result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);
  // see what device layers are available
  myDeviceLayers = new const char *[layerCount];
  myDeviceLayers[0] = "VK_LAYER_LUNARG_api_dump";
  myDeviceLayers[1] = "VK_LAYER_LUNARG_core_validation";
  myDeviceLayers[2] = "VK_LAYER_LUNARG_image";
  myDeviceLayers[3] = "VK_LAYER_LUNARG_object_tracker";
  myDeviceLayers[4] = "VK_LAYER_LUNARG_parameter_validation";
}
```

```
// see what device extensions are available:
uint32_t  extensionCount;
if (vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr) == VK_SUCCESS)
{
  VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
  uint32_t result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
  // see what device extensions are available
  myDeviceExtensions = new const char *[extensionCount];
  myDeviceExtensions[0] = "VK_KHR_surface";
  myDeviceExtensions[1] = "VK_KHR_win32_surface";
  myDeviceExtensions[2] = "VK_EXT_debug_report";
  myDeviceExtensions[3] = "VK_KHR_swapchains";
}
```

```
Vulkan: Overall Block Diagram

Application
  Instance
    Physical Device
      Logical Device
        Queue
          Command Buffer

Vulkan: a More Typical (and Simplified) Block Diagram

Application
  Instance
    Physical Device
      Logical Device
        Command Buffer
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'
160 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033   1  'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
160 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'

Vulkan: Creating a Logical Device

float   queuePriorities[1] =
{
1.
}

Vulkan: Creating the Logical Device's Queue

// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0,  OUT &Queue );               // 0, 0 = queueFamilyIndex, queueIndex

Layers and Extensions

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_LUNARG_vktrace' 'Vktrace tracing library'
0x00400033   1  'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x0040000d   1  'VK_LAYER_NV_nsight'  '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'
0x00000019  'VK_KHR_surface'
0x00000003  'VK_KHR_surfaceTools'
0x00000003  'VK_KHR_external_fence_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR external_memory_capabilities'
0x00000001  'VK_KHR_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':

vkEnumerateDeviceExtensionProperties:

11 instance layers available:
0x00400033   2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
0x00400033   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  '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'
0x00400038   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x0040000d   1  'VK_LAYER_NV_nsight'  'NVIDIA Nsight interception layer'
0x00400000  34  '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  '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_NV_external_memory_capabilities'

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 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;
}

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] = {
    1.0f,
};
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_LUNARG_validation",
};

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

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

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

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

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

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

In an effort to expand flexibility and retain efficiency, Vulkan provides something called Push Constants. Like the name implies, these let you “push” constant values out to the shaders. These are typically used for small, frequently-updated data values, such as matrix 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 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.

Push Constants are pushed at the shaders by giving them to pValues size stageFlags where:

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

Here is a void function that establishes the push constants structure:

```c
VkPushConstantRange
vpconst[0].size = sizeof(glm::mat4);
vpconst[0].offset = 0;
vpconst[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT;

vpconst[1].size = sizeof(glm::mat4);
vpconst[1].offset = 112;
vpconst[1].stageFlags = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
```

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
};
```

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
vpconst[0].size = sizeof(glm::mat4);
vpconst[0].offset = 0;
vpconst[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT;

vpconst[1].size = sizeof(glm::mat4);
vpconst[1].offset = 112;
vpconst[1].stageFlags = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
```
In the `Reset()` Function

```c
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

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

In the `UpdateScene()` Function

```c
float rot1 = (float)(2.*M_PI*Time); // rotation for arm1, in radians
float rot2 = 2.f * rot1; // rotation for arm2, in radians
float rot3 = 2.f * rot2; // rotation for arm3, in radians
glm::vec3 zaxis =  glm::vec3(0., 0., 1.);
glm::mat4 m1g = glm::mat4( 1. ); // identity
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis); // [T]*[R]
glm::mat4 m21 = glm::mat4( 1. ); // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis); // [T]*[R]
m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm
glm::mat4 m32 = glm::mat4( 1. ); // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis); // [T]*[R]
m32 = glm::translate(m32, glm::vec3(0., 0., 2.)); // z-offset from previous arm
Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m1g * m21; // m2g
Arm3.armMatrix = m1g * m21 * m32; // m3g
```

In the Vertex Shader

```c
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
```

In the `RenderScene()` Function

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

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

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

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

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

Where Synchronization Fits in the Overall Block Diagram

- 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.

Semaphores Example during the Render Loop

Creating a Semaphore

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

#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

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

Events

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

Controlling Events from the Host

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

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

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

Note: the host cannot 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, IN event, pipelineStageBits );

Could be an array of events

where signaled, where wait for the signal

Memory barriers get executed after events have been signaled

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

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.

**Why Do We Need Pipeline Barriers?**

**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.

**vkCmdPipelineBarrier() Function Call**

 pourquoi le passage de dépendance entre les commandes qui ont été soumises avant le barrière et les commandes qui sont soumises après le barrière... permet d’assurer que... 


**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 cars continue.
Pipeline Stages

- Vertex Shader
- Fragment Shader
- Computed Shader
- Geometry Shader
- Tesselation Evaluation Shader
- Tesselation Control Shader
- Primitive Assembly
- Register

Example: Be sure we are done writing an Output image before using it.

Access Operations and what Pipeline Stages they can be used in

- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT

Example: The Scenario

src cars are generating the image
dst cars are waiting to use that image as a texture
Aliasing

The Display We Want

Too often, the Display We Get

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, 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.

One pixel

Sub-pixels

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.

Consider Two Triangles That Pass Through the Same Pixel

Let's assume (for now) that the two triangles don't overlap—that is, they look this way because they butt up against each other.
Consider Two Triangles Who Pass Through the Same Pixel

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

### Number of Fragment Shader Calls

<table>
<thead>
<tr>
<th></th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Red fragment</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

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?

### Number of Fragment Shader Calls

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

Setting up the Image

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

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

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

VK_TRUE means to allow some sort of multisampling to take place.
VkPipelineMultisampleStateCreateInfo vpmisci;

vpmsci.minSampleShading = 0.5;

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

0. produces simple multisampling
(0. - 1.) produces partial supersampling
1. Produces complete supersampling

VkAttachmentDescription vad[2];

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[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;

Setting up the Image

Summary

Vulkan Program Flow – the Setup

Vulkan

307

308

309

310

311

312
So What Do We All Do Now?

- It doesn't see Vulkan replacing OpenGL ever.
- However, I wonder if Khronos will become less and less excited about adding new extensions to OpenGL.
- And, I also wonder if vendors will become less and less excited about improving OpenGL drivers.

So What Do We All Do Now?

- Performance-critical
- Performance-uncritical
- Need ray-tracing

The Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).
SIGGRAPH '23 Courses, August 08-10, 2023, Los Angeles, CA, USA
ACM 978-1-4503-8742-2/23/08.
DOI: 10.1145/3587423.3595529

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