Introduction

Vulkan

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

Acknowledgements

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

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

Second, thanks to NVIDIA! The GeForce 1080ti cards are what made this course possible.

What Prompted the Move to Vulkan?

1. Performance
2. Performance
3. Performance

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

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

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

OpenGL 4.2 Pipeline Flowchart
Why is it so important to keep the GPU Busy?

<table>
<thead>
<tr>
<th>GPU</th>
<th>Titan V</th>
<th>Tesla V100</th>
<th>Nvidia P100</th>
<th>G100 2020</th>
<th>V100 2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>12 TFLOP</td>
<td>31 TFLOP</td>
<td>34 TFLOP</td>
<td>14 TFLOP</td>
<td>13 TFLOP</td>
</tr>
<tr>
<td>Data Transfer Rate</td>
<td>47 GB/s</td>
<td>119 GB/s</td>
<td>145 GB/s</td>
<td>58.6 GB/s</td>
<td>58.6 GB/s</td>
</tr>
<tr>
<td>Texture Memory</td>
<td>32 GB</td>
<td>32 GB</td>
<td>32 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>VRAM</td>
<td>64 GB</td>
<td>16 GB</td>
<td>16 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Who was the original Vulcan?

From WikiPedia:

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


Why Name it after the God of the Forge?

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

Who’s Been Specifically Working on Vulkan?

Vulkan

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

Vulkan Differences from OpenGL

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

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

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

Fragment Processing, Texturing, Per-Fragment Lighting

Framebuffer

The Basic Computer Graphics Pipeline, Shader-style

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

gl_Vertex, gl_Normal, gl_Color

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

Fragment Processing, Texturing, Per-Fragment Lighting

Framebuffer

The Basic Computer Graphics Pipeline, Vulkan-style

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

Per-vertex in variables

Vertex Shader

gl_Position, Per-vertex out variables

Fragment Shader

Framebuffer

A Complete API Redesign

Originally architected for graphics workstations with direct renderers and split memory
Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering

Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance
Explicit API – the application has direct, predictable control over the operation of the GPU

Threading model doesn’t enable generation of graphics commands in parallel to command execution
Multi-core friendly with multiple command buffers that can be created in parallel

Syntax evolved over twenty years – complex API choices can obscure optimal performance paths
Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance

Shader language compiler built into driver. Only GLSL supported. Have to ship shader source
SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability

Despite conformance testing, developers must often handle implementation variability between vendors
Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability

Khronos Group
Moving part of the driver into the application

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

**Vulkan**

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

Vulkan Highlights: Command Buffers

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

Vulkan Highlights: Pipelines

- In OpenGL, your “pipeline state” is whatever your current graphics attributes are: color, transformations, textures, shaders, etc.
- Changing the state on-the-fly one item at-a-time is very expensive.
- Vulkan forces you to set all your state at once into a “pipeline state object” (PSO) and then invoke the entire PSO whenever you want to use that state combination.
- Think of the pipeline state as being immutable.
- Potentially, you could have thousands of these pre-prepared states.
- This is a good time to talk about how game companies view Vulkan…

Querying the Number of Things

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

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

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

```c
How many total there are Where to put them
result = vkEnumeratePhysicalDevices( Instance, &count, NULLptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
```
Vulkan Code has a Distinct "Style"

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>;
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;

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

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

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

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```

Vulkan Quick Reference Card

Vulkan Highlights: Overall Block Diagram

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

Vulkan: Creating a Pipeline

Vulkan GPU Memory

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

From the OpenGL Shader Storage Buffer notes:
```c
void createBuffer( VkBuffer &buffer, uint32_t dataSize ) { 
  glGenBuffers( 1, &buffer ); 
  glBindBuffer( GL_SHADER_STORAGE_BUFFER, _buffer ); 
  glBufferData( GL_SHADER_STORAGE_BUFFER, dataSize, NULL, GL_DYNAMIC_DRAW ); 
  GLint bufMask= GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT; // the invalidate makes a big difference when re-writing
  struct pos *points = ( struct pos * ) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, dataSize, bufMask );
  // you need to testMapBufferRange before access
  glBindBuffer( GL_SHADER_STORAGE_BUFFER, 0 );
  // remember to unmap the buffer
}
```
Vulkan Render Passes

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

Vulkan Compute Shaders

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

Vulkan Synchronization

- Vulkan tries to run “flat out”
- Therefore, synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan does not ever lock – that’s the application’s job
- Threads can concurrently read from the same object
- Threads can concurrently write to different objects

Vulkan Shaders

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

Advantages:

1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends
Vertex Buffers are how you draw things in Vulkan. They are very much like Vertex Buffer Objects in OpenGL, but more detail is exposed to you (a lot more…).

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

First, a quick review of computer graphics geometry . . .
typedef enum VkPrimitiveTopology
{
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;

Vulkan Topologies – Some OpenGL Topologies are Missing

OpenGL Topologies – Polygon Requirements

Polygons must be:

• Convex and
• Planar

Vulkan Topologies – Requirements and Orientation

Polygons must be:

• Convex and
• Planar

Polygons are traditionally:

• CCW when viewed from outside the solid object

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

VK_LINE_STRIP

V0
V1
V2
V3

VK_LINE_STRIP

V0
V1
V2
V3

What does “Convex Polygon” Mean?

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

Convex

V0
V1
V2
V3

Not Convex

V0
V1
V2
V3

Why is there a Requirement for Polygons to be Convex?

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

Convex

Not Convex

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

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

https://github.com/ivanfratric/polypartition

If you ever need to do this, contact me. I have working code …
Why is there a Requirement for Polygons to be Planar?

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

Vertex Orientation Issues

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system. Internally, however, the Vulkan pipeline uses a left-handed system:

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

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

This is like saying "Y' = -Y".

A Colored Cube Example

Triangles in an Array of Structures

From the file SampleVertexData.cpp:

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

struct vertex VertexData[] = {
    // triangle 0-2-3:
    // vertex #0:
    {
        { -1., -1., -1. },
        { 0., 0., -1. },
        { 0., 0., 0. },
        { 1., 0. }
    },
    // vertex #2:
    {
        { -1.,  1., -1. },
        { 0., 0., -1. },
        { 0., 1., 0. },
        { 1., 1. }
    },
    // vertex #3:
    {
        {  1.,  1., -1. },
        { 0., 0., -1. },
        { 1., 1.,  0. },
        { 0.,1. }
    },
    ...,
    // triangle 0-1-5:
    // vertex #0:
    {
        { -1., -1., -1. },
        { 0., 0., -1. },
        { 0., 0.,  0. },
        { 1., 0. }
    },
};
```
Vertex Orientation Issues

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

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

Vulkan’s change in coordinate systems can mess up the backface culling. So I recommend, at least at first, that you do no culling.

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

Filling the Vertex Buffer

```cpp
MyBuffer MyVertexDataBuffer;
MyVertexDataBuffer.size = sizeof(VertexData);
Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *)VertexData );
Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
```

What Init05DataBuffer Does

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

The Vulkan Pipeline

The Vulkan Pipeline consists of several stages, each with its own responsibilities. Here is a summary of the stages:

1. **Vertex Input Stage**
   - **Vertex Input Binding**
   - **Vertex Input Attributes**

2. **Input Assembly**

3. **Topology**
   - **Fragment Shader Stage**
   - **Depth Stencil**

4. **Viewport Scissor**

5. **Rasterization**
   - **Depth Clamping**
   - **Discard Enable**

6. **Color Blending**
   - **Color Blending Parameters**

Each stage has specific input and output parameters, which are set using corresponding state objects.

```
#define vertexShader_module
#define specializationInfo
#define vertexInputBinding
#define vertexInputAttributes
```

```
#define fragmentShaderStage
#define depthStencil

#define colorBlending
```

```
#define vertexInputStage
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

```
#define topology
```

```
#define fragmentShaderStage
```

```
#define depthStencil
```

```
#define colorBlending
```

```
#define vertexInputStage
```

```
#define inputAssembly
```

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

Telling the Pipeline about its Input

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

C/C++

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

GLSL

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

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

Telling the Pipeline about its Input

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

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

VkPipelineInputAssemblyStateCreateInfo vpiasci;  // used to describe the Pipeline's input to the pipeline
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;vpiasci.flags = 0;vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;;

VkGraphicsPipelineCreateInfo vgpci;                // used to create the Pipeline
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;vgpci.pNext = nullptr;vgpci.flags = 0;
vgpci.stageCount = 2;                // number of shader stages in this pipeline
vgpci.pStages = vpssci;vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pViewportState = &vpvsci;vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = ... GraphicsPipelineLayout;vgpci.renderPass = IN RenderPass;vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );

Telling the Pipeline about its Input

We will come to the Pipeline later, but for now, know that a Vulkan Pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its input.
VkBuffer buffers[1] = MyVertexDataBuffer.buffer;
    vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
    const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
    const uint32_t instanceCount = 1;
    const uint32_t firstVertex = 0;
    const uint32_t firstInstance = 0;
    vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

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

Better to do this than to hard-code a number

Vulkan Topologies

typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;
A Colored Cube Example

Triangles Represented as an Array of Structures

Non-indexed Buffer Drawing

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

A Reminder 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:

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

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


C/C++:

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

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

VkPipelineVertexInputStateCreateInfo vpvisci;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.vertexBindingDescriptionCount = 1;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = vviad;

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

Telling the Pipeline about its Input

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

VkPipelineInputStateCreateInfo

Telling the Pipeline about its Input

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

VkPipelineInputAssemblyStateCreateInfo

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.
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.

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

### Telling the Pipeline about its Input

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

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

### Telling the Command Buffer what Vertices to Draw

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

```
struct vertex JustVertexData[] = {
  // vertex #0:
  { -1., -1., -1. },
  {  0.,  0., -1. },
  {  0.,  0.,  0. },
  {  1., 0. },
  // vertex #1:
  {  1., -1., -1. },
  {  0.,  0., -1. },
  {  1.,  0.,  0. },
  {  0., 0. },
  // ...}

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

### Drawing with an Indexed Buffer

Transmission Order

<table>
<thead>
<tr>
<th>Index</th>
<th>Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4-5-7</td>
</tr>
<tr>
<td>7</td>
<td>5-6-4</td>
</tr>
<tr>
<td>6</td>
<td>1-4-5</td>
</tr>
<tr>
<td>5</td>
<td>1-5-6</td>
</tr>
<tr>
<td>4</td>
<td>2-3-4</td>
</tr>
<tr>
<td>3</td>
<td>3-0-4</td>
</tr>
<tr>
<td>2</td>
<td>0-3-1</td>
</tr>
<tr>
<td>1</td>
<td>2-0-1</td>
</tr>
<tr>
<td>0</td>
<td>0-0-0</td>
</tr>
</tbody>
</table>

Actual Index Data

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

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

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

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

Drawing with an Indexed Buffer

```c
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32 );
const uint32_t vertexCount = sizeof(JustVertexData) / sizeof(JustVertexData[0]);
const uint32_t indexCount = sizeof(JustIndexData) / sizeof(JustIndexData[0]);
const uint32_t instanceCount = 1; const uint32_t firstVertex = 0; const uint32_t firstIndex = 0; const uint32_t firstInstance = 0;
const uint32_t vertexOffset = 0;
#ifdef VERTEX_BUFFER
    vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
#endif
#ifdef INDEX_BUFFER
    vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
#endif
```

Indirect Drawing (not to be confused with Indexed)

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

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

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

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

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

Compare this with:

```c
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```
Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn't.

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

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

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

```
v 1.710541 1.283360 -0.040860
v 1.714593 1.273043 -0.041939
v 1.722786 1.267216 -0.041939
v 1.727156 1.271963 -0.041795
v 1.732121 1.265378 -0.037323
v 1.745131 1.286599 -0.037101
v 1.761568 1.297970 -0.037073
v 1.762207 1.290297 -0.040704
v 1.698431 1.285652 -0.040489
v 1.705163 1.268445 -0.029861
v 1.715231 1.283344 -0.029807
...```

```
vn 0.1725 0.2557 -0.9512
vn -0.1979 -0.1899 -0.9616
vn -0.2050 -0.2127 -0.9554
vn 0.1664 0.3020 -0.9387
vn -0.2040 -0.1718 -0.9638
vn 0.1465 0.5030 -0.9292
vn -0.2055 -0.1698 -0.9638
vn 0.4419 0.6438 -0.6249
vn 0.4573 0.5682 -0.6841
vn 0.5166 0.5538 -0.6353
vn 0.1791 0.2082 -0.9616
vn -0.2167 -0.2250 -0.9499
vn 0.4624 0.6871 -0.2987
```

```
vt 0.816406 0.955536
vt 0.822754 0.959168
vt 0.815918 0.959442
vt 0.829501 0.958662
vt 0.829501 0.958662
vt 0.834494 0.958818
vt 0.824219 0.951283
vt 0.817383 0.951538
vt 0.810590 0.951385
vt 0.809070 0.953833
vt 0.809070 0.953833
vt 0.810350 0.948381
...```

```
f 73/73/75 65/65/67 66/66/68
f 66/66/68 74/74/76 73/73/75
f 74/74/76 66/66/68 67/67/69
f 67/67/69 75/75/77 74/74/76
f 75/75/77 67/67/69 69/69/71
f 69/69/71 76/76/78 75/75/77
f 71/71/73 72/72/74 77/77/79
f 72/72/74 78/78/80 77/77/79
f 78/78/80 72/72/74 73/73/75
f 73/73/75 79/79/81 78/78/80
f 79/79/81 73/73/75 74/74/76
f 74/74/76 80/80/82 79/79/81
f 80/80/82 74/74/76 75/75/77
f 75/75/77 81/81/83 80/80/82
```

Note: The OBJ file format uses 1-based indexing for faces!
A **Data Buffer** is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a "Binary Large Object", or "BLOB").

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

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

```c
typedef VkBuffer VkDataBuffer;
```

### Terminology Issues

**Vulkan: Buffers**

The `VkBufferCreateInfo` struct defines the properties of a buffer.

- `sType`: `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
- `pNext`: Pointer to next structure
- `flags`: Buffer flags
- `size`: Buffer size in bytes
- `usage`: Buffer usage flags
- `sharingMode`: `VK_SHARING_MODE_EXCLUSIVE` or `VK_SHARING_MODE_CONCURRENT`
- `queueFamilyIndexCount`: Number of queue family indices
- `pQueueFamilyIndices`: Pointer to array of queue family indices

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = <buffer size in bytes>;
vbci.usage = <buffer usage flags>;
vbci.sharingMode = <sharing mode>;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const uint32_t*)nullptr;

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

### Vulkan: Creating a Data Buffer

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

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

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

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

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

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

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

2 Memory Heaps:
- Heap 0: size = 0xb7c00000 DeviceLocal
- Heap 1: size = 0x90000000
typedef struct MyBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;

MyBuffer MyMatrixUniformBuffer;

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

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

```c
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer ) {
    ... vbci.size = pMyBuffer->size = size;
    ... result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    ... pMyBuffer->vdm = vdm;
}
```

Here’s the C struct to hold some uniform variables

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

Here’s the shader code to access those uniform variables

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

Filling those Uniform Variables

```c
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4();              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```
This C struct is holding the actual data. It is writeable by the application.

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

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

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

CPU:

```c
struct matBuf Matrices;
```

The Parade of Data

Creating and Filling the Data Buffer – the Details

```
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.size = pMyBuffer->size;
  vbci.usage = usage;
  vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
  vbci.queueFamilyIndexCount = 0;
  vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
  result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer);
  VkMemoryRequirements vmr;
  vkGetBufferMemoryRequirements(LogicalDevice, IN pMyBuffer->buffer, OUT &vmr);
  VkMemoryAllocateInfo vmai;
  vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  vmai.pNext = nullptr;
  vmai.allocationSize = vmr.size;
  vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
  VkDeviceMemory vdm;
  result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
  pMyBuffer->vdm = vdm;
  result = vkBindBufferMemory(LogicalDevice, pMyBuffer->buffer, IN vdm, 0);
  return result;
}
```

Creating and Filling the Data Buffer

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

The Descriptor Set for the Buffer

```
VkWriteDescriptorSet vws0;
// ds 0
vws0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vws0.pNext = nullptr;
vws0.dstSet = ...;
vws0.dstBinding = 0;
vws0.dstArrayElement = 0;
vws0.descriptorCount = 1;
vws0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vws0.pBufferInfo = &vdbi0;
vws0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
```

Vulkan code snippet for creating a descriptor set and writing to it.
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.

The Shaders' View of the Basic Computer Graphics Pipeline

- In general, you want to have a vertex and fragment shader as a minimum.
- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.
- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shaders.

Shader stages

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

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:
- In the compiler, there is an automatic
  #define VKLVL 100

Vertex and Instance indices:
- gl_VertexIndex
- gl_InstanceIndex
- Both are 0-based

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

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:
- In the compiler, there is an automatic
  #define VULKAN 100

These are
- gl_VertexID
- gl_InstanceID
- In OpenGL. The Vulkan names make more sense.

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

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

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

Specialization Constants for Compute Shaders:
layout(local_size_x_id = 8, local_size_y_id = 16);
- gl_WorldGroupSize.z is still as it was

Push Constants:
layout(push_constant) . . .

Vulkan Shaders’ use of Layouts for Uniform Variables

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

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

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

All opaque (non-sampler) uniform variables must be in block buffers

Vulkan Shader Compiling

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

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

The first open standard intermediate language for parallel compute and graphics:

- SPIR (Standard Portable Intermediate Representation) was initially developed for use by OpenGL and SPIR versions 1.2 and 2.0 were based on LLVM. SPIR has now evolved into a true cross-API standard that is fully defined by Khronos with native support for shader and kernel features – called SPIR-V.
- SPIR-V is the first open standard, cross-API intermediate language for natively representing parallel compute and graphics and is incorporated as part of the core specification of both OpenCL 2.1 and OpenCL 2.2 and the new Vulkan graphics and compute API.

- SPIR-V exposes the machine model for OpenCL 1.2, 2.0, 2.1, 2.2 and Vulkan - including full flow control, and graphics and parallel constructs not supported in LLVM. SPIR-V also supports OpenCL C and OpenCL C++ kernel languages as well as the GLSL shader language for Vulkan.
- SPIR-V 1.1, launched in parallel with OpenCL 2.2, now supports all the kernel language features of OpenCL C++ in OpenCL 2.2, including initializer and finalizer function execution modes to support constructors and destructors. SPIR-V 1.1 also enhances the expressiveness of kernel programs by supporting named barriers, subgroup execution, and program scope pipes.

- SPIR-V is catalyzing a revolution in the language compiler ecosystem – it can split the compiler chain across multiple vendors' products, enabling high-level language front-ends to emit programs in a standardized intermediate form to be ingested by Vulkan or OpenGL drivers. For hardware vendors, ingesting SPIR-V eliminates the need to build a high-level language source compiler into device drivers, significantly reducing driver complexity, and will enable a broader range of language and framework front-ends to run on diverse hardware architectures.

- For developers, using SPIR-V means that kernel source code no longer has to be directly exposed, kernel load times can be accelerated and developers can choose the use of a common language front-end, improving kernel reliability and portability across multiple hardware implementations.

https://www.khronos.org/spir

SPIR-V:

Standard Portable Intermediate Representation for Vulkan


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

(Can be overridden by the -S option)

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

Windows: glslangValidator.exe
Linux: setenv LD_LIBRARY_PATH /usr/local/common/gcc-6.3.0/lib64/
You can also run SPIR-V from a Linux Shell

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

How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages.

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

So, if you do an `od -x` on the .spv file, the magic number looks like this:

```
0203 0723 . . .
```
VkResult
InitSpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb" );
    if( fp == NULL )
    {
        fprintf( FpDebug, "Cannot open shader file %s\n", filename.c_str() );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
    {
        fprintf( FpDebug, "Magic number for spir-v file %s is 0x%08x -- should be 0x%08x\n", filename.c_str(), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char *code = new unsigned char [size];
    fread( code, size, 1, fp );
    fclose( fp );

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

    VkResult result = vkCreateShaderModule( LogicalDevice, IN &vsmci, PALLOCATOR, pShaderModule );
    fprintf( FpDebug, "Shader Module %s successfully loaded\n", filename.c_str() );
    delete[ ] code;
    return result;
}
For example, if this is your Shader Source

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

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

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

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

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

This is the SPIR-V Assembly, Part I

```assembly
1:           ExtInstImport "GLSL.std.450"
MemoryModel Logical GLSL450EntryPoint Vertex 4  "main" 34 37 48 53 56 57 61 63
Source GLSL 400
SourceExtension ... 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix" Name 15  "Matrices"
Name 32  "gl_PerVertex"
MemberName 32(gl_PerVertex) 0  "gl_Position" MemberName 32(gl_PerVertex) 1  "gl_PointSize" MemberName 32(gl_PerVertex) 2  ... 0 MemberDecorate 13(matBuf) 0 MatrixStride 16 MemberDecorate 13(matBuf) 1 ColMajor MemberDecorate 13(matBuf) 1 Offset 64 MemberDecorate 13(matBuf) 1 MatrixStride 16 Decorate 13(matBuf) BlockDecorate 15(Matrices) DescriptorSet 0
2:             TypeVoid
3:             TypeFunction 26:             TypeFloat 327:             TypeVector 6(float) 48:             TypeMatrix 7(fvec4) 49:             TypePointer Function 8 11:             TypeVector 6(float) 312:             TypeMatrix 11(fvec3) 313(matBuf):             TypeStruct 8 8 8 12 14:             TypePointer Uniform 13(matBuf) 15(Matrices):     14(ptr) Variable Uniform 16:             TypeInt 32 117:     16(int) Constant 2 30:             TypePointer Uniform 821:     16(int) Constant 125:     16(int) Constant 0 30:             TypeArray 6(float) 3 32(gl_PerVertex):             TypeStruct 7(fvec4) 6(float) 33:             TypePointer Output 32(gl_PerVertex) 34:     33(ptr) Variable Output 36:             TypePointer Input 11(fvec3) 37(aVertex):     36(ptr) Variable Input 39:    6(float) Constant 10 65:           8 Store 35 38:   11(fvec3) Load 37(aVertex) 40:    6(float) CompositeExtract 38 0 41:    6(float) ... 24 43:    7(fvec4) CompositeConstruct 40 41 42 39 44:    7(fvec4) MatrixTimesVector 35 43 46:     45(ptr) AccessChain 34 25 Store 46 44
51:     50(ptr) AccessChain 15(Matrices) 49 52:          12 Load 51 54:   11(fvec3) Load 53(aNormal) 55:   11(fvec3) MatrixTimesVector 52 54 Store 48(vNormal) 55 58:   11(fvec3) Load 57(aColor) Store 56(vColor) 58 64:   59(fvec2) Load 63(aTexCoord) Store 61(vTexCoord) 64 ReturnFunctionEnd
```

This is the SPIR-V Assembly, Part II

```assembly
11:           ExtInstImport "GLSL.std.450"
MemoryModel Logical GLSL450EntryPoint Vertex 4  "main" 34 37 48 53 56 57 61 63
Source GLSL 400
SourceExtension ... 1  "uViewMatrix" MemberName 13(matBuf) 2  "uProjectionMatrix" MemberName 13(matBuf) 3  "uNormalMatrix" Name 15  "Matrices"
Name 32  "gl_PerVertex"
MemberName 32(gl_PerVertex) 0  "gl_Position" MemberName 32(gl_PerVertex) 1  "gl_PointSize" MemberName 32(gl_PerVertex) 2  ... 0 MemberDecorate 13(matBuf) 0 MatrixStride 16 MemberDecorate 13(matBuf) 1 ColMajor MemberDecorate 13(matBuf) 1 Offset 64 MemberDecorate 13(matBuf) 1 MatrixStride 16 Decorate 13(matBuf) BlockDecorate 15(Matrices) DescriptorSet 0
2:             TypeVoid
3:             TypeFunction 26:             TypeFloat 327:             TypeVector 6(float) 48:             TypeMatrix 7(fvec4) 49:             TypePointer Function 8 11:             TypeVector 6(float) 312:             TypeMatrix 11(fvec3) 313(matBuf):             TypeStruct 8 8 8 12 14:             TypePointer Uniform 13(matBuf) 15(Matrices):     14(ptr) Variable Uniform 16:             TypeInt 32 117:     16(int) Constant 2 30:             TypePointer Uniform 821:     16(int) Constant 125:     16(int) Constant 0 30:             TypeArray 6(float) 3 32(gl_PerVertex):             TypeStruct 7(fvec4) 6(float) 33:             TypePointer Output 32(gl_PerVertex) 34:     33(ptr) Variable Output 36:             TypePointer Input 11(fvec3) 37(aVertex):     36(ptr) Variable Input 39:    6(float) Constant 10 65:           8 Store 35 38:   11(fvec3) Load 37(aVertex) 40:    6(float) CompositeExtract 38 0 41:    6(float) ... 24 43:    7(fvec4) CompositeConstruct 40 41 42 39 44:    7(fvec4) MatrixTimesVector 35 43 46:     45(ptr) AccessChain 34 25 Store 46 44
51:     50(ptr) AccessChain 15(Matrices) 49 52:          12 Load 51 54:   11(fvec3) Load 53(aNormal) 55:   11(fvec3) MatrixTimesVector 52 54 Store 48(vNormal) 55 58:   11(fvec3) Load 57(aColor) Store 56(vColor) 58 64:   59(fvec2) Load 63(aTexCoord) Store 61(vTexCoord) 64 ReturnFunctionEnd
```
Installing bash on Windows

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

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

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

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

- I’ve written everything out in **appalling longhand**.
- Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.
- At times, I could have hidden complexity, but I didn’t. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that’s a secret to you.
- I’ve setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
- At times, I’ve setup things that didn’t need to be setup just to show you what could go there.
- There are good uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
- I’ve typedef’d a couple things to make the Vulkan phraseology more consistent.
- Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure
- At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
- I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

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

Init07TextureSampler( &MyPuppyTexture.texSampler );
Init07TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyPuppyTexture );
Init08Swapchain();
Init09DepthStencilImage();
Init10RenderPasses();
Init11Framebuffers();
Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
Init13DescriptorSetPool();
Init13DescriptorSetLayouts();
Init13DescriptorSets();
Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );

A Colored Cube

A Colored Cube

A Colored Cube

A Colored Cube

The Vertex Data is in a Separate File

#include "SampleVertexData.cpp"

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

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

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

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

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

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums
2. Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary information into a function. For example, where we might normally say in C++:
   ```
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex, OUT &Queue );
   ```
   we would actually say in C:
   ```
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```

Vulkan Conventions

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

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

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

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

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

Linux shader compiler

Windows shader compiler

Linux shader compiler

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

Reporting Error Results, I

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

ErrorCodes[] = {
    { VK_NOT_READY, "Not Ready" },
    { VK_TIMEOUT, "Timeout" },
    { VK_EVENT_RESET, "Event Reset" },
    { VK_INCOMPLETE, "Incomplete" },
    { VK_ERROR_OUT_OF_HOST_MEMORY, "Out of Host Memory" },
    { VK_ERROR_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" },
    { VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" },
    { VK_ERROR_DEVICE_LOST, "Device Lost" },
    { VK_ERROR_MEMORY_MAP_FAILED, "Memory Map Failed" },
    { VK_ERROR_LAYER_NOT_PRESENT, "Layer Not Present" },
    { VK_ERROR_EXTENSION_NOT_PRESENT, "Extension Not Present" },
    { VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" },
    { VK_ERROR_INCOMPATIBLE_DRIVER, "Incompatible Driver" },
    { VK_ERROR_TOO_MANY_OBJECTS, "Too Many Objects" },
    { VK_ERROR_FORMAT_NOT_SUPPORTED, "Format Not Supported" },
    { VK_ERROR_FRAGMENTED_POOL, "Fragmented Pool" },
    { VK_ERROR_SURFACE_LOST_KHR, "Surface Lost" },
    { VK_ERROR_NATIVE_WINDOW_IN_USE_KHR, "Native Window In Use" },
    { VK_SUBOPTIMAL_KHR, "Suboptimal" },
    { VK_ERROR_OUT_OF_DATE_KHR, "Out Of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out Of Pool Memory" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR, "Invalid External Handle" },
};
```

Reporting Error Results, II

```c
void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s
", prefix.c_str(), "Successful" );
        fflush(FpDebug);
        return;
    }

    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = "";
    for( int i = 0; i < numErrorCodes; i++ )
    {
        if( result == ErrorCodes[i].resultCode )
        {
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }
    fprintf( FpDebug, "
%s: %s
", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```

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

Extras in the Code

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

boolPaused;
bool Verbose;

# define DEBUGFILE "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
```
Setting Up GLFW

```c
void InitGLFW() {
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
    MainWindow = glfwCreateWindow(Width, Height, "Vulkan Sample", NULL, NULL);
    VkResult result = glfwCreateWindowSurface(Instance, MainWindow, NULL, &Surface);
    glfwSetErrorCallback(GLFWErrorCallback);
    glfwSetKeyCallback(MainWindow, GLFWKeyboard);
    glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
    glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

GLFW Keyboard Callback

```c
void GLFWKeyboard(GLFWwindow * window, int key, int scancode, int action, int mods) {
    if (action == GLFW_PRESS) {
        switch(key) {
        case GLFW_KEY_M:
            if (Mode == 1)
                Mode = 0;
            else
                Mode = 1;
            break;
        default:
            fprintf(FpDebug, "Unknow key hit: 0x%04x = '%c'", key, key);
            fflush(FpDebug);
        }
    }
}
```
GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

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

Looping and Closing GLFW

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

GLM.pptx

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

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

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

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

You invoke GLM like this:
#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.

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:

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

you would now have to say:

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

Exactly the same concept, 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.

```gl
// constructor:
glm::mat4( );
glm::vec4( ); // identity matrix
glm::vec3( );

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

// emulating OpenGL transformations with concatenation:
glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```

The Most Useful GLM Variables, Operations, and Functions

```gl
// viewing volume (assign, not concatenate):
glm::mat4 glm::ortho( float left, float right, float bottom, float top, float near, float far );
glm::mat4 glm::frustum( float left, float right, float bottom, float top, float near, float far );

// viewing (assign, not concatenate):
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );
```
Installing GLM into your own space

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

Here’s what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

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

5. "Your GLM folder location"
GLM in the Vulkan sample.cpp Program

```cpp
if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4();  // identity
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale, Scale, Scale) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3(0., 1., 0.) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3(1., 0., 0.) );
    // done this way, the Yrot is applied first, then the Xrot, then the Scale
}
else
{
    if( !Paused )
    {
        const glm::vec3 axis = glm::vec3(0., 1., 0.);
        Matrices.uModelMatrix = glm::rotate( glm::mat4(), (float)glm::radians(360.f*Time/SECONDS_PER_CYCLE), axis );
    }
}
Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uProjectionMatrix[1][1] *= -1.;  // Vulkan's projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uProjectionMatrix[1][1] *= -1.;
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
Fill05DataBuffer(MyMatrixUniformBuffer, (void*) &Matrices);
Misc.uTime = (float)Time;
Misc.uMode = Mode; Fill05DataBuffer(MyMiscUniformBuffer, (void*) &Misc);
```

Your Sample2017.zip File Contains GLM Already

How Does this Matrix Stuff Really Work?

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

Or, in matrix form:

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

Transformation Matrices

**Translation**

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

**Scaling**

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} =
\begin{bmatrix}
  S_x & 0 & 0 & 0 \\
  0 & S_y & 0 & 0 \\
  0 & 0 & S_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

**Rotation about X**

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

**Rotation about Y**

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

**Rotation about Z**

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

\[
\begin{bmatrix}
\cos 90^\circ & \sin 90^\circ \\
-\sin 90^\circ & \cos 90^\circ
\end{bmatrix}
\begin{bmatrix}
\alpha_1 \\
\alpha_2
\end{bmatrix} = \text{rotation}
\]

http://xkcd.com

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

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

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

Compound Transformations

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

A: We create more than one matrix.

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

Matrix Multiplication is not Commutative

Rotate, then translate

Translate, then rotate
Matrix Multiplication is Associative

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \left( \begin{bmatrix}
  T_{A,B} & 0 & 0 & 0 \\
  0 & T_{A,B} & 0 & 0 \\
  0 & 0 & T_{A,B} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \right) \left( \begin{bmatrix}
  R_y & 0 & 0 & 0 \\
  0 & R_y & 0 & 0 \\
  0 & 0 & R_y & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \right) \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

One matrix – the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{pmatrix} = \left( \begin{bmatrix}
  T_{A,B} & 0 & 0 & 0 \\
  0 & T_{A,B} & 0 & 0 \\
  0 & 0 & T_{A,B} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \right) \left( \begin{bmatrix}
  R_y & 0 & 0 & 0 \\
  0 & R_y & 0 & 0 \\
  0 & 0 & R_y & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix} \right) \begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

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

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

Wrong!

Right!

```cpp
glm::mat3 NormalMatrix = glm::mat3(Model);
```
Instancing – What and why?

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

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

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

Making each instance look differently -- Approach #1

Use the built-in vertex shader variable \texttt{gl\_InstanceIndex} to define a unique display property, such as position or color.

\texttt{gl\_InstanceIndex} starts at 0

In the vertex shader:

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

Making each instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer and reference them

Still uses \texttt{gl\_InstanceIndex}

In the vertex shader:

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

Read a new characteristic for each instance

Internally uses `gl_InstanceIndex`, but you don’t

This is just the Vertex Input State Portion of the Graphics Pipeline Structure

VkVertexInputBindingDescription

- `binding`
- `stride`
- `inputRate`

 vinegar [1]

// an array containing one of these per buffer being used

- `binding`
- `stride`
- `inputRate`

// bytes between successive

- `binding`
- `format`
- `offset`

This definition says that we should advance through the input buffer by this much every time we hit a new vertex.
vkPipelineVertexInputStateCreateInfo
  vpvisci;
  // used to describe the input vertex attributes
  vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vvisci.pNext = nullptr;vpvisci.flags = 0;
vvisci.vertexBindingDescriptionCount = 2;
vvisci.pVertexBindingDescriptions = vvibd;
vvisci.vertexAttributeDescriptionCount = 5;
vvisci.pVertexAttributeDescriptions = vviad;

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

How We Constructed the Graphics Pipeline Structure Before

VkVertexInputBindingDescription vvibd[2];
vvibd[0].binding = 0;           // which binding # this is
vvibd[0].stride = sizeof( struct vertex );              // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].binding = 1;           // which binding # this is
vvibd[1].stride = sizeof( glm::vec3 );             // bytes between successive entries
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;

This definition says that we should advance through the input buffer by this much every time we hit a new instance.

How We Construct the Graphics Pipeline Structure Now

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

VkVertexInputAttributeDescription vviad[5];
// an array containing one of these per vertex attribute in all bindings
// 4 = vertex, normal, color, texture coord
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position );                  // 0
...
vviad[5].location = 0;                  // location in the layout decoration
vviad[5].binding = 1;                   // which binding description this is part of
vviad[5].format = VK_FORMAT_VEC3;       // r, g, b
vviad[5].offset = 0;                      // just one element, so offset is 0

How We Construct the Graphics Pipeline Structure Now

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

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

Note: same names as before, but different sizes

This definition says that we should advance through the input buffer by this much every time we hit a new instance.

How We Construct the Graphics Pipeline Structure Now

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

VkVertexInputBindingDescription vvibd[2];
vvibd[0].binding = 0;       // which binding # this is
vvibd[0].stride = sizeof(struct vertex); // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].binding = 1;       // which binding # this is
vvibd[1].stride = sizeof(glm::vec3); // bytes between successive entries
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;

...
How We Write the Vertex Shader Now

```glsl
version 400
extension GL_ARB_separate_shader_objects : enable
extension GL_ARB_shading_language_420pack : enable
layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
layout( location = 4 ) in vec3 aInstanceColor;
layout ( location = 0 ) out vec3 vNormal;
layout ( location = 1 ) out vec3 vColor;
layout ( location = 2 ) out vec2 vTexCoord;
void main()
{
    mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
    vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );
    vColor = aInstanceColor;
    vTexCoord = aTexCoord;
    gl_Position = PVM * vec4( aVertex, 1. );
}
```

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 bunch of related uniform variables all at once?

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

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

What are Descriptor Sets?

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

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

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

Our example will assume the following shader uniform variables:

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

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

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

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

**Step 1: Descriptor Set Pools**

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

```plaintext
void VkResult
vkCreateDescriptorSetPool()
{
    VkDescriptorSet pool;
    VkSampler sampler;
    VkShaderStageFlagBits stages = VK_SHADER_STAGE_FRAGMENT_BIT;
    VkDescriptorSetLayoutCreateInfo
    { stage = stages,
      count = 1,
      pSampler = &sampler,
      pBindings = &bindings[0]
    };
    result = vkCreateDescriptorSetPool(Device, sampler, stages, &bindings[0], DescriptorPool);
}
```
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.

```cpp
VkResult Init13DescriptorSetLayouts()
{
    Pipeline data structure to allocate room for the uniform variables and to access them.

    // DS #0:
    layout( std140, set = 0, binding = 0 ) uniform matBuf
    MatrixSet[1]
    VkDescriptorSetLayoutBinding
    MatrixSet[0].binding            = 0;
    mat4 uModelMatrix;mat4 uViewMatrix;mat4 uProjectionMatrix;mat3 uNormalMatrix;
    MatrixSet[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MatrixSet[0].descriptorCount = 1;

    // DS #1:
    layout( std140, set = 1, binding = 0 ) uniform lightBuf
    LightSet[1]
    VkDescriptorSetLayoutBinding
    LightSet[0].binding            = 0;
    vec4 uLightPos;vec4 uLightSpecularColor;vec4 uEyePos;
    LightSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT |
    VK_SHADER_STAGE_FRAGMENT_BIT;
    LightSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #2:
    layout( std140, set = 2, binding = 0 ) uniform miscBuf
    MiscSet[1]
    VkDescriptorSetLayoutBinding
    MiscSet[0].binding            = 0;
    vec4 rgba = texture( uSampler, vST );
    MiscSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    MiscSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc0, PALLOCATOR, OUT &DescriptorSetLayouts[0]);
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc1, PALLOCATOR, OUT &DescriptorSetLayouts[1]);
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc2, PALLOCATOR, OUT &DescriptorSetLayouts[2]);
    result = vkCreateDescriptorSetLayout(LogicalDevice, &vdslc3, PALLOCATOR, OUT &DescriptorSetLayouts[3]);
    return result;
}
```
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```c
VkResult Init14GraphicsPipelineLayout()
{
    VkResult result;
    VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

Array of Descriptor Set Layouts

---

Step 4: Allocating the Memory for Descriptor Sets

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

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

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

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

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

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

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

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

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

This struct identifies what texture sampler and image view it owns

Good to use `sizeof`
Step 5: Tell the Descriptor Sets where their CPU Data is

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

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

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

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

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

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

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

```
```
The Vulkan Graphics Pipeline

What is the Vulkan Graphics Pipeline?

- The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”.
- There’s a lot that goes into it.
- For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.
- The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

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

There is also a Vulkan Compute Pipeline – we will get to that later.

Here’s what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”.
2. There’s a lot that goes into it.
3. For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

The First Step: Create the Graphics Pipeline Layout

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

VkResult Init14GraphicsPipelineLayout() {
  VkResult result;

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

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

Let the Pipeline Layout know about the Descriptor Set and Push Constant layouts.
Vulkan: A Pipeline Records the Following Items:

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

**BoldText** indicates that this state item can also be set with Dynamic Variables

Creating a Typical Graphics Pipeline

```c
VkPipelineShaderStageCreateInfo vpssci[2];
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pNext = nullptr;
vpssci[0].flags = 0;
vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
vpssci[0].module = vertexShader;
vpssci[0].pName = "main";
vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[1].pNext = nullptr;
vpssci[1].flags = 0;
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
```

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

Link in the Shaders

```c
VkPipelineShaderStageCreateInfo vpssci[2];
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pNext = &vpmsci;
vpssci[0].flags = 0;
vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
vpssci[0].module = vertexShader;
vpssci[0].pName = "main";
vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[1].pNext = &vpmsci;
vpssci[1].flags = 0;
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
```

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

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

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

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

Options for \texttt{vpiasci.topology}

What is "Primitive Restart Enable"?

"Restart Enable" is used with:

- Indexed drawing.
- Triangle Fan and "Strip" topologies.

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

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

If your \texttt{VkIndexType} is \texttt{VK_INDEX_TYPE_UINT16}, then the special index is 0xffff.
If your \texttt{VkIndexType} is \texttt{VK_INDEX_TYPE_UINT32}, it is 0xffffffff.

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

```
short int restartIndex = ~0;
```
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

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

What is the Difference Between Changing the Viewport and Changing the Scissoring?

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

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

Setting the Rasterizer State

```cpp
VkPipelineRasterizationStateCreateInfo vprsci;
    vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    vprsci.pNext = nullptr;
    vprsci.flags = 0;
    vprsci.depthClampEnable = VK_FALSE;
    vprsci.rasterizerDiscardEnable = VK_FALSE;
    vprsci.polygonMode = VK_POLYGON_MODE_FILL;
    #ifdef CHOICES
    VK_POLYGON_MODE_FILL
    VK_POLYGON_MODE_LINE
    VK_POLYGON_MODE_POINT
    #endif
    vprsci.cullMode = VK_CULL_MODE_NONE;   // recommend this because of the projMatrix[1][1] *= -1.;
    #ifdef CHOICES
    VK_CULL_MODE_NONE
    VK_CULL_MODE_FRONT_BIT
    VK_CULL_MODE_BACK_BIT
    VK_CULL_MODE_FRONT_AND_BACK_BIT
    #endif
    vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    #ifdef CHOICES
    VK_FRONT_FACE_COUNTER_CLOCKWISE
    VK_FRONT_FACE_CLOCKWISE
    #endif
    vprsci.depthBiasEnable = VK_FALSE;
    vprsci.depthBiasConstantFactor = 0.f;
    vprsci.depthBiasSlopeFactor = 0.f;
    vprsci.lineWidth = 1.f;
```
**What is “Depth Clamp Enable”?**

vprsci.depthClampEnable = VK_FALSE;

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

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

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

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

**What is “Depth Bias Enable”?**

vprsci.depthBiasEnable = VK_FALSE;

vprsci.depthBiasConstantFactor = 0.f;

vprsci.depthBiasClamp = 0.f;

vprsci.depthBiasSlopeFactor = 0.f;

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

**MultiSampling State**

VkPipelineMultisampleStateCreateInfo vpmsci;

vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

vpmsci.pNext = nullptr;

vpmsci.flags = 0;

vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

vpmsci.sampleShadingEnable = VK_FALSE;

vpmsci.minSampleShading = 0;

vpmsci.pSampleMask = (VkSampleMask *)nullptr;

vpmsci.alphaToCoverageEnable = VK_FALSE;

vpmsci.alphaToOneEnable = VK_FALSE;

Declare information about how the multisampling will take place

**Color Blending State for each Color Attachment**

VkPipelineColorBlendAttachmentState vpcbas;

vpcbas.blendEnable = VK_FALSE;

vpcbas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;

vpcbas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;

vpcbas.colorBlendOp = VK_BLEND_OP_ADD;

vpcbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;

vpcbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;

vpcbas.alphaBlendOp = VK_BLEND_OP_ADD;

vpcbas.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT |
VK_COLOR_COMPONENT_A_BIT;

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

This controls blending between the output of each color attachment and its image memory.
Color Blending State for each Color Attachment

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

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

Which Pipeline Variables can be Set Dynamically?

```c
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR, VK_DYNAMIC_STATE_LINE_WIDTH, VK_DYNAMIC_STATE_DEPTH_BIAS, VK_DYNAMIC_STATE_BLEND_CONSTANTS, VK_DYNAMIC_STATE_DEPTH_BOUNDS, VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK, VK_DYNAMIC_STATE_STENCIL_WRITE_MASK, VK_DYNAMIC_STATE_STENCIL_REFERENCE, VK_DYNAMIC_STATE_COLOR_WRITE_MASK_OPS, VK_DYNAMIC_STATE_COLOR_INDEX_TYPE, VK_DYNAMIC_STATE_DEPTH_STENCIL_OPS, VK_DYNAMIC_STATE_DEPTH_STENCIL抽奖表(REFERENCE) };

This will control which pipeline variables can be set dynamically.
```

Stencil Operations for Front and Back Faces

```c
VkStencilOpState vsosf;  // front
    vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
    vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
    vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds
    vsosf.compareOp = VK_COMPARE_OP_NEVER; // compare mask
    vsosf.compareMask = ~0;
    vsosf.writeMask = ~0;
    vsosf.reference = 0;

VkStencilOpState vsosb;  // back
    vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
    vsosb.failOp = VK_STENCIL_OP_KEEP;
    vsosb.passOp = VK_STENCIL_OP_KEEP;
    vsosb.compareOp = VK_COMPARE_OP_NEVER;
    vsosb.compareMask = ~0;
    vsosb.writeMask = ~0;
    vsosb.reference = 0;
```

Uses for Stencil Operations

- Magic Lenses
- Polygon edges without Z-fighting
Operations for Depth Values

```cpp
VkPipelineDepthStencilStateCreateInfo vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
VP_COMPARE_OP_NEVER -- never succeeds
VP_COMPARE_OP_LESS -- succeeds if new depth value is < existing value
VP_COMPARE_OP_EQUAL -- succeeds if new depth value is == existing value
VP_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= existing value
VP_COMPARE_OP_GREATER -- succeeds if new depth value is > existing value
VP_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != existing value
VP_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= existing value
VP_COMPARE_OP_ALWAYS -- always succeeds#endif
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;
#endif
```

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

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

Putting it all Together! (finally…)

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

Queues and Command Buffers

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

Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

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

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

We can use this function to create a command pool and command buffers as part of the logical device.
Beginning a Command Buffer

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

```c
textIndex = imageReadySemaphore;
result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, OUT &imageReadySemaphore);
```

```c
VkCommandBufferAllocateInfo cmdBufInfo;
uint32_t nextImageIndex;
```

```c
result = vkCreateCommandBufferPool(LogicalDevice, VK_COMMAND_BUFFER_TYPE网投型, OUT &cmdBufInfo);
```

```c
vkAcquireNextImageKHR(LogicalDevice, SwapChain, UINT64_MAX, imageReadySemaphore, VK_NULL_HANDLE, OUT &nextImageIndex);
```

```c
VkCommandBufferBeginInfo vcbbi;
```

```c
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
```

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

```c
result = vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
```

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

```c
vkCmdBeginQuery(commandBuffer, flags);
vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
vkCmdBeginRenderPass(commandBuffer, const contents);
vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
vkCmdBindIndexBuffer(commandBuffer, indexType);
vkCmdNextSubpass(commandBuffer, contents);
vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);
vkCmdBlitImage(commandBuffer, filter);
vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdPushDescriptorSetKHR(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);
vkCmdCopyBuffer(commandBuffer, pRegions);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyImage(commandBuffer, pRegions);
vkCmdCopyImageToBuffer(commandBuffer, pRegions);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
vkCmdSetDeviceMaskKHX(commandBuffer, deviceMask);
vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);
vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);
vkCmdSetEvent(commandBuffer, event, stageMask);
vkCmdSetLineWidth(commandBuffer, lineWidth);
vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);
vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);
vkCmdSetStencilReference(commandBuffer, faceMask, reference);
vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);
vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewportWScalings);
vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);
```

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

```c
vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
```
VkResult RenderScene()
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

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

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

    VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

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

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

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

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

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

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

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

    VkViewport viewport = {
        0., (float)Width, 0., (float)Height, 0., 1.0, 0., 1.0
    };
    vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, &viewport);

    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);

    VkDeviceSize offset = 0;
    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);

    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 = 1;
    vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
    Submitting a Command Buffer to a Queue for Execution

    VkClearValue vcc;
    vcc.color = vcc;
    vcc.depth = vcc;
    Submitting a Command Buffer to a Queue for Execution
The Entire Submission / Wait / Display Process

Create fence

Get the queue

Fill in the queue information

Submit the queue

Wait for the fence

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

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

result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

// 0 = queueIndex

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

vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;

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

result = vkQueuePresentKHR(presentQueue, IN &vpi);

The Entire Submission / Wait / Display Process

How We Think of OpenGL Framebuffers

Video Driver

Update

Double-buffered Color Framebuffers

Refresh

Depth-Buffer

Front

Back

Video Driver

Vulkan Thinks of it as a Ring Buffer

Swap Chain

Back

Depth-Buffer

Back
What is a Swap Chain?

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

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

Swap Chains are arranged as a ring buffer

Swap Chains are tightly coupled to the window system.

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

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

Because it has the word "chain" in it, let's try to visualize the Swap Chain as a physical chain.

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

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

In the same way that bicycle chain links are "re-used", Swap Chain images get re-used too.

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

We Need to Find Out What our Display Capabilities Are

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

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

Swap Chains are arranged as a ring buffer

Swap Chains are tightly coupled to the window system.

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

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

Because it has the word "chain" in it, let's try to visualize the Swap Chain as a physical chain.

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

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

In the same way that bicycle chain links are "re-used", Swap Chain images get re-used too.

This is a pretty good analogy, except that there can be many more images in the ring buffer than are being shown here.
We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

vkGetPhysicalDeviceSurfaceCapabilitiesKHR:
    minImageCount = 2 ; maxImageCount = 8
    currentExtent = 1024 x 1024
    minImageExtent = 1024 x 1024
    maxImageExtent = 1024 x 1024
    maxImageArrayLayers = 1
    supportedTransforms = 0x0001
    currentTransform = 0x0001
    supportedCompositeAlpha = 0x0001
    supportedUsageFlags = 0x009f
    ** This Surface is supported by the Graphics Queue **

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

Found 3 Present Modes:
0:        2 ( VK_PRESENT_MODE_FIFO_KHR )
1:        3 ( VK_PRESENT_MODE_FIFO_RELAXED_KHR )
2:        1 ( VK_PRESENT_MODE_MAILBOX_KHR )

We Need to Find Out What our Display Capabilities Are

Creating a Swap Chain

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

Creating the Swap Chain Images and Image Views

uint32_t imageCount = 0; // # of display buffers – 2? 3?
result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr );
PresentImages = new VkImage[ imageCount ];
result = vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );

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

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

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

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

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

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

VkSubmitInfo vsi;
  vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;vsi.pNext = nullptr;vsi.waitSemaphoreCount = 1;vsi.pWaitSemaphores = &imageReadySemaphore;
  vsi.pWaitDstStageMask = &waitAtBottom;vsi.commandBufferCount = 1;vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
  vsi.signalSemaphoreCount = 0;vsi.pSignalSemaphores = &SemaphoreRenderFinished;
result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence ); // 1 = submitCount

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

VkPresentInfoKHR vpi;
  vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  vpi.pNext = nullptr;
  vpi.waitSemaphoreCount = 0;
  vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
  vpi.pWaitDstStageMask = &waitAtBottom;
  vpi.commandBufferCount = 1;
  vpi.pCommandBuffers = &CommandBuffers[nextImageIndex];
  vpi.swapchainCount = 1;
  vpi.pSwapchains = &SwapChain;
  vpi.pImageIndices = &nextImageIndex;
  vpi.pResults = (VkResult *) nullptr;
result = vkQueuePresentKHR( presentQueue, IN &vpi );
VkPipelineRasterizationStateCreateInfo vprsci;

vprsci.cullMode = VK_CULL_MODE_NONE
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;

Matrices.uProjectionMatrix[1][1] *= -1.;
Vulkan: Submitting to a Queue

VkSubmitInfo
pipelineStageFlags
cmdBufferCount
cmdBuffers[]
waitSemaphores
vkQueueSubmit()
vkGetDeviceQueue()

VkResult
RenderScene() {
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);
    VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
    VkClearColorValue vccv;
    vccv.float32[0] = 0.0; vccv.float32[1] = 0.0;
    vccv.float32[2] = 0.0; vccv.float32[3] = 1.0;
    VkClearDepthStencilValue vcdsv;
    vcdsv.depth = 1.f;
    vcdsv.stencil = 0;
    VkClearValue vcv[2];
    vcv[0].color = vccv; vcv[1].depthStencil = vcdsv;
    VkOffset2D o2d = { 0, 0 }; VkExtent2D e2d = { Width, Height }; VkRect2D r2d = { o2d, e2d };
    VkRenderPassBeginInfo vrpbi;
    vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    vrpbi.pNext = nullptr;
    vrpbi.renderPass = RenderPass; vrpbi.framebuffer = Framebuffers[nextImageIndex]; vrpbi.renderArea = r2d;
    vrpbi.clearValueCount = 2;
    vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
    vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
    vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &viewport);
    vkCmdSetScissor(CommandBuffers[nextImageIndex], 0, 1, &scissor);
    vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
    //vkCmdBindPushConstants(CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values);
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };  
VkDeviceSize offsets[1] = { 0 };  
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );  
const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);  
const uint32_t instanceCount = 1;  
const uint32_t firstVertex = 0;  
const uint32_t firstInstance = 0;  
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );  
vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );  
vkEndCommandBuffer( CommandBuffers[nextImageIndex] );  

VkFenceCreateInfo vfci;  
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;  
vfci.pNext = nullptr;  
vfci.flags = 0;  
VkFence renderFence;  
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );  

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;  
VkQueue presentQueue;  
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue );  
VkSubmitInfo vsi;  
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;  
vsi.pNext = nullptr;  
vsi.waitSemaphoreCount = 1;  
vsi.pWaitSemaphores = &imageReadySemaphore;  
vsi.pWaitDstStageMask = &waitAtBottom;  
vsi.commandBufferCount = 1;  
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];  
vsi.signalSemaphoreCount = 0;  
vsi.pSignalSemaphores = &SemaphoreRenderFinished;  
result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence );  
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );  
vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );  

result = vkQueuePresentKHR( presentQueue, IN &vpi);  
vkDestroySemaphore( LogicalDevice, imageReadySemaphore, PALLOCATOR );  

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. This can be most any image. At one time, some graphics hardware required the image's pixel dimensions to be a power of two. This restriction has been lifted on most (all?) graphics cards, but just to be safe... The X and Y dimensions did not need to be the same power of two, just a power of two. So, a 128x512 image would have been OK: a 129x511 image might not have.

Also, to prevent confusion, the texture pixels are not called pixels. A pixel is a dot in the final screen image. A dot in the texture image is called a texture element, or texel. Similarly, to avoid terminology confusion, a texture's width and height dimensions are not called X and Y. They are called S and T. A texture map is not generally indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always S=0, the right side is S=1, the bottom is T=0, and the top is T=1. Thus, you do not need to be aware of the texture's resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.
The mapping between the geometry of the 3D object and the S and T of the texture image works like this:

\[ T = 1 \]
\[ S = 0 \]
\[ T = 0 \]
\[ S = 1 \]

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

Enable texture mapping:
```c
glEnable( GL_TEXTURE_2D );
```

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

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

To figure out what (s,t) to assign to each vertex, you specify an (s,t) pair at each vertex, along with the vertex coordinate. The same time that the rasterizer is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when it goes to draw each pixel, it uses that pixel’s interpolated (s,t) to lookup a color in the texture image.

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

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

Struct VertexData

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

Triangles in an Array of Structures

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

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

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

Or, for a sphere,

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

From the Sphere code:

\[
s = \frac{(\text{lng} + M\_\pi)}{(2\cdot M\_\pi)}; \\
t = \frac{(\text{lat} + M\_\pi/2)}{M\_\pi};
\]

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

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

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

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

Be careful where \(s\) abruptly transitions from 1. back to 0.
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbs
Textures' Undersampling Artifacts

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

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

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

- Graphics hardware determines which level to use based on the texels : pixels ratio.
- In addition to just picking one mip-map level, the rendering system can sample from two of them, one less that the T:P ratio and one more, and then blend the two RGBA's returned. This is known as Vk_SAMPLER_MIPMAP_MODE_LINEAR.

```c
void Init07TextureSampler( MyTexture * pMyTexture )
{
    Init07TextureBuffer( INOUT MyTexture * pMyTexture )
    {
        VkResult result;
        VkSamplerCreateInfo vsci;
        uint32_t texWidth = pMyTexture->width;
        uint32_t texHeight = pMyTexture->height;
        unsigned char *texture = pMyTexture->pixels;
        VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each

        vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
        vsci.pNext = nullptr;
        vsci.flags = 0;
        vsci.magFilter = VK_FILTER_LINEAR;
        vsci.minFilter = VK_FILTER_LINEAR;
        vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;

        // ************************************************************ *******************
        // this first {...} is to create the staging image:
        // ************************************************************ *******************

        VkImageCreateInfo vici;
        vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
        vici.pNext = nullptr;
        vici.flags = 0;
        vici.imageType = VK_IMAGE_TYPE_2D;
        vici.format = VK_FORMAT_R8G8B8A8_SRGB;
        vici.tiling = VK_IMAGE_TILING_LINEAR;
        vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
        vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vici.mipLevels = 1;
        vici.arrayLayers = 1;
        vici.samples = VK_SAMPLE_COUNT_1_BIT;

        result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, OUT &pMyTexture->textureImage );
        
        VkSamplerCreateInfo vsci;
        vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
        vsci.pNext = nullptr;
        vsci.flags = 0;
        vsci.magFilter = VK_FILTER_LINEAR;
        vsci.minFilter = VK_FILTER_LINEAR;
        vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
        vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.mipLodBias = 0.;
        vsci.anisotropyEnable = VK_FALSE;
        vsci.maxAnisotropy = 1.;
        vsci.compareEnable = VK_FALSE;
        vsci.compareOp = VK_COMPARE_OP_NEVER;

        VkImageCreateInfo vici;
        vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
        vici.pNext = nullptr;
        vici.flags = 0;
        vici.imageType = VK_IMAGE_TYPE_2D;
        vici.format = VK_FORMAT_R8G8B8A8_SRGB;
        vici.tiling = VK_IMAGE_TILING_LINEAR;
        vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
        vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vici.mipLevels = 1;
        vici.arrayLayers = 1;
        vici.samples = VK_SAMPLE_COUNT_1_BIT;

        result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, OUT &pMyTexture->stagingImage );
        
        VkCommandBufferCreateInfo vcci;
        vcci.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_CREATE_INFO;
        vcci.pNext = nullptr;
        vcci.flags = VK_COMMAND_BUFFER_CREATE_RESET_COMMAND_BUFFER_BIT;
        
        VkCommandBufferBeginInfo vcci;
        vcci.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
        vcci.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

        Cmd_FillWithRed();
        Cmd_FillWithRed();
        Cmd_FillWithRed();
        Cmd_FillWithRed();
        Cmd_FillWithRed();
        
        enable comparison against a reference value during lookups
```
```c
#ifdef CHOICES
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_PREINITIALIZED#endif

vici.queueFamilyIndexCount = 0;
vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

VkMemoryRequirements vmr;

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

if (Verbose)
// 0 and 0 = offset and memory map flags
    fprintf(FpDebug, "Image vmr.size = %lld\n", vmr.size);
    fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
    fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    fflush(FpDebug);

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

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

// ************************************************************ *******************
result = vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
if (Verbose)
// 0 and 0 = offset and memory map flags
    fprintf(FpDebug, "Texture vmr.size = %lld\n", vmr.size);
    fprintf(FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment);
    fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    fflush(FpDebug);

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

VkImageCreateInfo vici;
vici.pNext = nullptr;
vici.flags = 0;
vici.imageType = VK_IMAGE_TYPE_2D;
vici.format = VK_FORMAT_R8G8B8A8_UNORM;
vici.extent.width = texWidth;
vici.extent.height = texHeight;

// ************************************************************ *******************
// transition the staging buffer layout:// *******************************************************************************
// because we are transferring into it and will eventual sample from it
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

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

VkImageMemoryBarrier vimb;

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

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

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

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

VkDeviceMemory vdm;

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

// *******************************************************************************
// copy pixels from the staging image to the texture:// this second {...} is to create the actual texture image:
// because we are transferring into it and will eventual sample from it
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

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

VkImageMemoryBarrier vimb;

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

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

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

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
```

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

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

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

// now do the final image transfer:
VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;visl.baseArrayLayer = 0;visl.mipLevel = 0;
visl.layerCount = 1;

VkOffset3D                              vo3;
vo3.x = 0;vo3.y = 0;vo3.z = 0;

VkExtent3D                              ve3;
ve3.width = texWidth;ve3.height = texHeight;ve3.depth = 1;

VkImageCopy vic;
vic.srcSubresource = visl;vic.srcOffset = vo3;vic.dstSubresource = visl;vic.dstOffset = vo3;vic.extent = ve3;

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

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

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

result = vkCreateImageView(
LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
return result;
}
Reading in a Texture from a BMP File

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

MyTexture MyPuppyTexture;
```

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

---

Anisotropic Texture Filtering

https://en.wikipedia.org/wiki/Anisotropic_filtering

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

Application
→ Instance
→ Physical Device
→ Logical Device
  Command Buffer
  Command Buffer
  Command Buffer

Vulkan: Identifying the Physical Devices

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr);
if( result != VK_SUCCESS || count <= 0 ){
    fprintf( FpDebug, "Could not count the physical devices
    return VK_SHOULD_EXIT;
}
VkPhysicalDeviceProperties vdp;
VkPhysicalDeviceProperties * physicalDevices = new VkPhysicalDeviceProperties[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );
if( result != VK_SUCCESS ){
    fprintf( FpDebug, "Could not enumerate the %d physical devices
    return VK_SHOULD_EXIT;
}

Vulkan: Querying the Number of Physical Devices

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

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, NULL);

Which Physical Device to Use, I

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    VkPhysicalDeviceProperties vdp;
    vdp = physicalDevices[i];
    if( result != VK_SUCCESS ){
        fprintf( FpDebug, "Could not get the physical device properties of device %d
        return VK_SHOULD_EXIT;
    }
    fprintf( FpDebug, "Device %2d:
    	API version: %d
    	Driver version: %d
    	Vendor ID: 0x%04x
    	Device ID: 0x%04x
    	Physical Device Type: %d = "
    	if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        fprintf( FpDebug, "(Discrete GPU)
    	if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        fprintf( FpDebug, "(Integrated GPU
    	if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
        fprintf( FpDebug, "(Virtual GPU
    	if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
        fprintf( FpDebug, "(CPU
    	if( vdp.deviceName == "Software" || vdp.deviceName == "SoftwareRenderer" )
        fprintf( FpDebug, "(Software Renderer
    
    VkResult result = VK_SUCCESS;
    result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
    if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 ){
        fprintf( FpDebug, "Could not count the physical devices
        return VK_SHOULD_EXIT;
    }
    VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ PhysicalDeviceCount ];
    result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );
    if( result != VK_SUCCESS ){
        fprintf( FpDebug, "Could not enumerate the %d physical devices
        return VK_SHOULD_EXIT;
    }
    VkPipelineCacheSize = "Software" || vdp.pipelineCacheUUID[0];
Asking About the Physical Device’s Features

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

fprintf( FpDebug, "Physical Device Features:
" );
fprintf( FpDebug, "geometryShader = %2d
", PhysicalDeviceFeatures.geometryShader );
fprintf( FpDebug, "tessellationShader = %2d
", PhysicalDeviceFeatures.tessellationShader );
fprintf( FpDebug, "multiDrawIndirect = %2d
", PhysicalDeviceFeatures.multiDrawIndirect );
fprintf( FpDebug, "wideLines = %2d
", PhysicalDeviceFeatures.wideLines );
fprintf( FpDebug, "largePoints = %2d
", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "multiViewport = %2d
", PhysicalDeviceFeatures.multiViewport );
fprintf( FpDebug, "occlusionQueryPrecise = %2d
", PhysicalDeviceFeatures.occlusionQueryPrecise );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d
", PhysicalDeviceFeatures.pipelineStatisticsQuery );
```

Here’s What the NVIDIA 1080Ti Produced

```c
vkEnumeratePhysicalDevices:
Device 0:
  API version: 4194360
  Driver version: 4194360
  Vendor ID: 0x10de
  Device ID: 0x1b06
  Physical Device Type: 2 = (Discrete GPU)
  Device Name: GeForce GTX 1080 Ti
  Pipeline Cache Size: 13
Device #0 selected (GeForce GTX 1080 Ti)

Physical Device Features:
  geometryShader = 1
  tessellationShader = 1
  multiDrawIndirect = 1
  wideLines = 1
  largePoints = 1
  multiViewport = 1
  occlusionQueryPrecise = 1
  pipelineStatisticsQuery = 1
  shaderFloat64 = 1
  shaderInt64 = 1
  shaderInt16 = 0
```

Here’s What the Intel HD Graphics 520 Produced

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

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

Which Physical Device to Use, II

```c
// need some logical here to decide which physical device to select:
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
  discreteSelect = i;
else if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
  integratedSelect = i;
else
  fprintf( FpDebug, "Could not select a Physical Device
" );
  return VK_SHOULD_EXIT;
```
Asking About the Physical Device's Different Memories

VkPhysicalDeviceMemoryProperties

vkGetPhysicalDeviceMemoryProperties(PhysicalDevice, OUT &vpdmp);

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

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

Here's What I Got

11 Memory Types:
Memory 0: Memory 1: Memory 2: Memory 3: Memory 4: Memory 5: Memory 6: DeviceLocal Memory 7: DeviceLocal HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0: size = 0x07c00000 DeviceLocal
Heap 1: size = 0x0fac00000

Asking About the Physical Device's Queue Families

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

Here's What I Got

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

Vulkan: Overall Block Diagram

Vulkan: a More Typical (and Simplified) Block Diagram

Looking to See What Device Layers are Available

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

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

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

```c
uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
```

What Device Layers and Extensions are Available

3 physical device layers enumerated:

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

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

- 0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarGValidation Layer'
- 0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':

Vulkan: Specifying a Logical Device Queue

```c
float queuePriorities[1] = {
1.0
};
```

Vulkan: Creating a Logical Device

```c
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1; // # of device queues
vdci.pQueueCreateInfos = &vdqci; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.ppEnabledLayerNames = myDeviceLayers;
vdci.enabledExtensionCount = 0;
vdci.ppEnabledExtensionNames = (const char **)nullptr; // no extensions
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = &PhysicalDeviceFeatures;
result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
```
Vulkan: Creating the Logical Device’s Queue

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

Vulkan Layers

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

They are not always necessary, but when you need them, you will be really glad they are there!

Layers and Extensions

Looking to See What Instance Layers and Instance Extensions are Available

```c
const char * instanceLayers[] =
{
    //VK_LAYER_LUNARG_api_dump", // turn this on if want to see each function call and its arguments (very slow!)
    "VK_LAYER_LUNARG_core_validation",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_NV_optimus",
};
const char * instanceExtensions[] =
{
    "VK_KHR_surface",
#ifdef _WIN32
    "VK_KHR_win32_surface",
#endif
    "VK_EXT_debug_report",
};
uint32_t numExtensionsWanted = sizeof(instanceExtensions) / sizeof(char *);
// see what layers are available:
vkEnumerateInstanceLayerProperties( &numLayersAvailable, (VkLayerProperties *)nullptr );
InstanceLayers = new VkLayerProperties[numLayersAvailable];
result = vkEnumerateInstanceLayerProperties( &numLayersAvailable, InstanceLayers );
// see what extensions are available:
uint32_t numExtensionsAvailable;
vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, (VkExtensionProperties *)nullptr );
InstanceExtensions = new VkExtensionProperties[numExtensionsAvailable];
result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable, InstanceExtensions );
```
Looking to See What Extensions are Both Wanted and Available

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

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

vkGetPhysicalDeviceMemoryProperties( IN PhysicalDevice, OUT &vpdmp );
uint32_t count = -1;
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:
    unit32_t extensionCount;
    vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr);
    VkExtensionProperties * extensions = new VkExtensionProperties[extensionCount];
    result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
    deviceExtensions = new VkExtensionProperties[extensionCount];
    result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
    //
    delete[ ] deviceLayers;
    delete[ ] deviceExtensions;
}
4 physical device layers enumerated:
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
vkEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033 1 'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
vkEnumerateDeviceExtensionProperties: Successful
  0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':

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

Remember the Overall Block Diagram?
### Semaphores

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

#### Semaphores Example during the Render Loop

```cpp
VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;

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

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

uint32_t nextImageIndex;

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkSubmitInfo vsi;

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

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

Could be an array of semaphores.
Fences

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

```plaintext
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

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

// VK_FENCE_CREATE_SIGNALED_BIT is only other option

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

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

// blocks:
result = vkWaitForFences( LogicalDevice, 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 = 0xffffffffffffffffff (= 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 even signal at one place in the pipeline and wait for it at another place in the pipeline
- Signaling in the pipeline means "signal as the last piece of this draw command passes that point in the pipeline".
- You can signal, un-signal, or test from a vk function or from a vkCmd function
- Can wait from a vkCmd function
### Controlling Events from the Host

```
VkEventCreateInfo veci;
veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;
veci.pNext = nullptr;
veci.flags = 0;
VkEvent event;
result = vkCreateEvent(LogicalDevice, IN &veci, PALLOCATOR, OUT &event);
result = vkSetEvent(LogicalDevice, IN event);
result = vkResetEvent(LogicalDevice, IN event);
result = vkGetEventStatus(LogicalDevice, IN event);
// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled
```

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

### Controlling Events from the Device

```
result = vkCmdSetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdResetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdWaitEvents(CommandBuffer, IN &event, pipelineStageBits);
result = vkCmdWaitEvents(CommandBuffer, IN &event, pipelineStageBits);
```

**Note:** the GPU cannot test for an event, but it can block waiting for one.

### From the Command Buffer Notes:

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

```c
vkCmdBeginQuery(commandBuffer, flags);
vkCmdBeginRenderPass(commandBuffer, const contents);
vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
vkCmdBindIndexBuffer(commandBuffer, indexType);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdBlitImage(commandBuffer, filter);
vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdCopyBuffer(commandBuffer, pRegions);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyImage(commandBuffer, pRegions);
vkCmdCopyImageToBuffer(commandBuffer, pRegions);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vvkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);
vkCmdDrawIndirect(commandBuffer, stride);
vkCmdDrawIndirectCountAMD(commandBuffer, stride);
vkCmdEndQuery(commandBuffer, query);
vkCmdEndRenderPass(commandBuffer);
vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);
```

**Could be an array of events**

**Where signaled, where wait for the signal**

**Memory barriers get executed after events have been signaled**

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

---

**PipeLine Barriers:**

A case of Gate-ing and Wait-ing

---

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

vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask) ;
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );

We don’t any one of these commands to have to wait on a previous command unless you say so. In general, we want all of these commands to be able to run “flat-out”. But, if we do that, surely there will be nasty race conditions!

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

Note: there is no problem with Read-then-Read (RtR) as no data has been changed
### The Scenario Rules

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

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

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

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

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_VERTEX_SHADER_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_MEMORY_WRITE_BIT
Pipeline Stages and what Access Operations can Happen There

1. VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
2. VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
3. VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
4. VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
5. VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
6. VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
7. VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
8. VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
9. VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
10. VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
11. VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
12. VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
13. VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
14. VK_PIPELINE_STAGE_TRANSFER_BIT
15. VK_PIPELINE_STAGE_HOST_BIT
16. VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
17. VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Access Operations and what Pipeline Stages they can be used In

Example: Be sure we are done writing an output image before using it for something else

The Scenario

src cars are generating the image
dst cars are doing something with that image
Example: Don’t read a buffer back to the host until a shader is done writing it

Example: Don’t read a buffer back to the host until a shader is done writing it

Stages

<table>
<thead>
<tr>
<th>Access types</th>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDEX_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Access types

<table>
<thead>
<tr>
<th>Access types</th>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VkImageLayout – How an Image gets Laid Out in Memory depends on how it will be Used

<table>
<thead>
<tr>
<th>VkImageMemoryBarrier</th>
<th>Used as a color attachment</th>
<th>Read into a shader as a texture</th>
<th>Copy from</th>
<th>Copy to</th>
<th>Show Image to viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>img</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.pNext = nullptr;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.srcAccessMask = ??;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.dstAccessMask = ??;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.oldLayout = ??;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.newLayout = ??;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.image = ??;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vimb.subresourceRange = visr;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

The Scenario

src cars
dst cars

Push Constants

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

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

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

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

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

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

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

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

Setting up the Push Constants for the Pipeline Structure

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

```glsl
VkPushConstantRange vpcr[1];

vpcr[0].stageFlags = Vk_PIPELINE_STAGE_VERTEX_SHADER_BIT |
  VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

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

VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

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

Creating a Pipeline

```glsl
vkCreateGraphicsPipeline( );
```

Array of states that can be set dynamically
An Robotic Example using Push Constants

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

Where each arm is represented by:

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

Forward Kinematics:
You Start with Separate Pieces, all Defined in their Own Local Coordinate System

1 2 3

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

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

Locations?
Positioning Part #1 With Respect to Ground

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

\[
M_{1/G} = T_{1/G} \cdot R_{\Theta_1}
\]

Write it
Say it

Positioning Part #2 With Respect to Ground

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

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

Write it
Say it

Positioning Part #3 With Respect to Ground

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

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

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

Write it
Say it

Why Do We Say it Right-to-Left?

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

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

\[
\begin{bmatrix}
x' \\
y' \\
z'\\
1
\end{bmatrix} = \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} \cdot \begin{bmatrix}
R \\
T \\
M
\end{bmatrix}
\]

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

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

Arm1.armMatrix = glm::mat4();
Arm1.armColor  = glm::vec3(0.0f, 1.0f, 0.0f);
Arm1.armScale  = 6.0f;
Arm2.armMatrix = glm::mat4();
Arm2.armColor  = glm::vec3(1.0f, 0.0f, 0.0f);
Arm2.armScale  = 4.0f;
Arm3.armMatrix = glm::mat4();
Arm3.armColor  = glm::vec3(0.0f, 0.0f, 1.0f);
Arm3.armScale  = 2.0f;

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

In the UpdateScene Function

```cpp
float rot1 = (float)Time;
float rot2 = 2.0f * rot1;
float rot3 = 2.0f * rot2;

glm::vec3 zaxis = glm::vec3(0.0f, 0.0f, 1.0f);
glm::mat4 m1g = glm::mat4();
m1g = glm::translate(m1g, glm::vec3(0.0f, 0.0f, 0.0f));
m1g = glm::rotate(m1g, rot1, zaxis);

glm::mat4 m21 = glm::mat4();
m21 = glm::translate(m21, glm::vec3(2.0f * Arm1.armScale, 0.0f, 0.0f));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0.0f, 0.0f, 2.0f));

glm::mat4 m32 = glm::mat4();
m32 = glm::translate(m32, glm::vec3(2.0f * Arm2.armScale, 0.0f, 0.0f));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0.0f, 0.0f, 2.0f));

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

In the RenderScene Function

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```
In the Vertex Shader

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

layout( location = 0 ) in vec3 aVertex;

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

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

Antialiasing and Multisampling

```
Aliasing

The Display We Want

Too often, the Display We Get
```

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

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

What the signal really is: what we want

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

**Nyquist Criterion**

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

**Anti-aliasing**

4x 16x
Multisampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there. There are two approaches:

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

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

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.
Supersampling:
- Final Pixel Color = \sum_1^B \text{Color sample from subpixel}_i
- # Fragment Shader calls = B

Multisampling:
- Final Pixel Color = 3 \cdot \text{One color sample from A} + 5 \cdot \text{One color sample from B}
- # Fragment Shader calls = 2

Setting up the Image:
- VkPipelineMultisampleStateCreateInfo `vpmsci`:
  - `vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;`
  - `vpmsci.pNext = nullptr;`
  - `vpmsci.flags = 0;`
  - `vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;`
  - `vpmsci.sampleShadingEnable = VK_TRUE;`
  - `vpmsci.minSampleShading = 0.5f;`
  - `vpmsci.pSampleMask = (VkSampleMask *)nullptr;`
  - `vpmsci.alphaToCoverageEnable = VK_FALSE;`
  - `vpmsci.alphaToOneEnable = VK_FALSE;`
- VkGraphicsPipelineCreateInfo `vgpci`:
  - `vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;`
  - `vgpci.pNext = nullptr;`
  - `vgpci.pMultisampleState = &vpmsci;`

Result:
- `vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );`

Setting up the Image:
- VkPipelineMultisampleStateCreateInfo `vpmsci`:
  - `vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;`
  - `vpmsci.pNext = nullptr;`
  - `vpmsci.flags = 0;`
  - `vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;`
  - `vpmsci.sampleShadingEnable = VK_TRUE;`
  - `vpmsci.minSampleShading = 0.5f;`
  - `vpmsci.pSampleMask = (VkSampleMask *)nullptr;`
  - `vpmsci.alphaToCoverageEnable = VK_FALSE;`
  - `vpmsci.alphaToOneEnable = VK_FALSE;`
- VkGraphicsPipelineCreateInfo `vgpci`:
  - `vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;`
  - `vgpci.pNext = nullptr;`
  - `vgpci.pMultisampleState = &vpmsci;`

Result:
- `vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );`

Setting up the Image:
- VkPipelineMultisampleStateCreateInfo `vpmsci`:
  - `vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;`
  - `vpmsci.pNext = nullptr;`
  - `vpmsci.flags = 0;`
  - `vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;`
  - `vpmsci.sampleShadingEnable = VK_TRUE;`
  - `vpmsci.minSampleShading = 0.5f;`
  - `vpmsci.pSampleMask = (VkSampleMask *)nullptr;`
  - `vpmsci.alphaToCoverageEnable = VK_FALSE;`
  - `vpmsci.alphaToOneEnable = VK_FALSE;`
- VkGraphicsPipelineCreateInfo `vgpci`:
  - `vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;`
  - `vgpci.pNext = nullptr;`
  - `vgpci.pMultisampleState = &vpmsci;`

Result:
- `vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );`

At least this fraction of samples will get their own fragment shader calls (as long as they pass the depth and stencil tests):
- 0. produces simple multisampling
- 0.1 produces partial supersampling
- 1. Produces complete supersampling
Setting up the Image

VkAttachmentDescription vad[2];
  vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
  vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
  vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
  vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
  vad[0].flags = 0;
  vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
  vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
  vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
  vad[1].flags = 0;

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

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

Resolving the Image:
Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

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

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

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

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

vkCmdResolveImage(cmdBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, 1, &vir);
Multipass Rendering uses Attachments --
What is a Vulkan Attachment Anyway?

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

– Vulkan Programming Guide

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

---

Back in Our Single-pass Days

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

Here comes a quick reminder of how we did that.

Afterwards, we will extend that.

---

Back in Our Single-pass Days, I

---

Back in Our Single-pass Days, II

---
So far, we’ve only performed single-pass rendering, but within a single Vulkan RenderPass, we can also have several subpasses, each of which is feeding information to the next subpass or subpasses.

In this case, we will look at following up a 3D rendering with some image processing on the outcome.

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

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

Multipass, IV

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

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

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

Multipass, V

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

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

```
vkCmdBeginRenderPass(CommandBuffer[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// first subpass is automatically started here
vkCmdBindPipeline(CommandBuffer[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
GraphicsPipeline);
vkCmdBindDescriptorSets(CommandBuffer[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
vkCmdBindVertexBuffers(CommandBuffer[nextImageIndex], 0, 1, vBuffers, offsets);
vkCmdDraw(CommandBuffer[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
. . .
vkCmdNextSubpass(CommandBuffer[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// second subpass is started here – doesn’t need any new drawing vkCmd’s
. . .
vkCmdEndRenderPass(CommandBuffer[nextImageIndex]);
```
Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

Which Pipeline State Variables can be Changed Dynamically

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

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

Creating a Pipeline

```c
VkDynamicState vds[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};
VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState);
vpdsci.pDynamicStates = &vds;
VkGraphicsPipelineCreateInfo vgpci;

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

If you declare certain state variables to be dynamic like this, then you must fill them in the command buffer! Otherwise, they are undefined and bad things are likely to happen.
Getting Information Back from the Graphics System

Vulkan

There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp

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

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

VkQueryPoolCreateInfo vqpci;
    vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
    vqpci.pNext = nullptr;
    vqpci.flags = 0;
    vqpci.queryType = << one of: >>
        VK_QUERY_TYPE_OCCLUSION, VK_QUERY_TYPE_PIPELINE_STATISTICS, VK_QUERY_TYPE_TIMESTAMP
    vqpci.queryCount = 3;
    vqpci.pipelineStatistics = 0; // bitmask of what stats you are querying for if you
        are doing a pipeline statistics query
    // VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT
    // VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT
    // VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT
    // VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT
    // VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT
    // VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT
    // VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT
    // VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT
    // VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT
    // VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT
    // VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT

vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &occlusionQueryPool );

vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &statisticsQueryPool );

vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &timestampQueryPool );

Occlusion Query

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

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

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

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

Pipeline Statistics Query

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

uint32_t _ counts[NUM_STATS];
result = vkGetQueryPoolResults( LogicalDevice, statisticsQueryPool, 0, 1,
    sizeof(uint32_t), &counts, 0, VK_QUERY_RESULT_WAIT_BIT );

vkCmdCopyQueryPoolResults( CommandBuffer, occlusionQueryPool, 0, 1,
    buffer, 0, 0, VK_QUERY_RESULT_WAIT_BIT );
Timestamp Query

Timestamp Queries count how many nanoseconds of time elapsed between the vkCmdBeginQuery and the vkCmdEndQuery.

```cpp
uint64_t nanosecondsCount;
result = vkGetQueryPoolResults( LogicalDevice, timestampQueryPool, 0, 1, sizeof(uint64_t), &nanosecondsCount, 0, VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT);

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

Timestamp Query

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

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

```cpp
vkCmdWriteTimeStamp( CommandBuffer, pipelineStages, timestampQueryPool, 0 );
```

**Compute Shaders**

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
Here is how you create a much-simpler Compute Pipeline

Start with Creating the Data Buffers

This is a Particle System application, so we need Positions, Velocities, and (possibly) Colors

```
layout( std140, set = 0, binding = 0 ) buffer Pos
{
    vec4 Positions[   ]; // array of structures
};

layout( std140, set = 0, binding = 1 ) buffer Vel
{
    vec4 Velocities[   ]; // array of structures
};

layout( std140, set = 0, binding = 2 ) buffer Col
{
    vec4 Colors[   ]; // array of structures
};
```

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when OpenGL examines the size of this buffer’s data store.

A Reminder about Data Buffers

```
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_STORAGE_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

Creating a Shader Storage 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_SHADER_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

VkBuffer Buffer;
result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

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

Fill the Data Buffer

```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;
}
```

Create the Compute Pipeline Layout

```c
VkDescriptorSetLayoutBinding ComputeSet[1];
  ComputeSet[0].binding = 0;
  ComputeSet[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
  ComputeSet[0].descriptorCount = 3;
  ComputeSet[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
  ComputeSet[0].pImmutableSamplers = (VkSampler *)nullptr;

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

VkPipelineLayoutCreateInfo vplci;
  vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  vplci.pNext = nullptr;
  vplci.flags = 0;
  vplci.setLayoutCount = 1;
  vplci.pSetLayouts = &ComputeSetLayout;
  vplci.pushConstantRangeCount = 0;
  vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout );
```
Create the Compute Pipeline

```cpp
VkPipelineShaderStageCreateInfo vpssci;
vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0;
vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;

result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline);
```

The Particle System Compute Shader -- Setup

```cpp
#version 430
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 )  buffer  Pos {
vec4  Positions[ ]; // array of structures
};
layout( std140, set = 0, binding = 1 )  buffer Vel {
vec4  Velocities[ ]; // array of structures
};
layout( std140, set = 0, binding = 2 )  buffer  Col {
vec4  Colors[ ]; // array of structures
};

// Define particle constants
const vectoR  G  = vectoR( 0., -9.8, 0. );
const float  DT  = 0.1;
const SPHERE  Sphere = vec4( -100., -800., 0., 600. );

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POIINT  p  = Positions[ gid ].xyz;
VELOCITY  v  = Velocities[ gid ].xyz;
POIINT  pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY  vp = v + G*DT;
Positions[ gid ].xyz  = pp;
Velocities[ gid ].xyz = vp;
```

The Particle System Compute Shader – The Physics

```cpp
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4

const VECTOR  G        =  VECTOR( 0., -9.8, 0. );
const float  DT        =  0.1;
const SPHERE Sphere = vec4( -100., -800., 0.,  600. );

uint gid = gi_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POIINT  p  = Positions[ gid ].xyz;
VELOCITY  v  = Velocities[ gid ].xyz;
POIINT  pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY  vp = v + G*DT;
Positions[ gid ].xyz  = pp;
Velocities[ gid ].xyz = vp;
```

The Particle System Compute Shader – How About Introducing a Bounce?

```cpp
VELOCITY  Bounce( VELOCITY vin, VECTOR n ) {
  VELOCITY  vout = reflect( vin, n );
  return vout;
}
```

```cpp
bool  IsInsideSphere( POINT p, SPHERE s )
{
  float r = length( p - s.xyz );
  return (  r < s.w );
}
```

```cpp
VELOCITY  BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
  VECTOR  n = normalize( p - s.xyz );
  return Bounce( v, n );
}
```
uint gid = gl_GlobalInvocationID.x; // the y and z are both 1 in this case
POINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;
POINT pp = p + v*DT + .5*DT*DT*G;
VELOCITY vp = v + G*DT;
if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + .5*DT*DT*G;
}
Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;

The Particle System Compute Shader –
How About Introducing a Bounce?

Dispatching the Compute Shader from the Command Buffer

const int NUM_PARTICLES = 100000;
const int NUM_WORK_ITEMS = 64;
const int NUM_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatch( CommandBuffer, NUM_WORK_GROUPS, 1, 1 );

Or,
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 ); // offset

Remember the Compute Pipeline?
A Specialization Constant is a way of injecting an integer or Boolean constant into an .spv-compiled version of a shader right before the final compilation.

That final compilation happens when you call `vkCreateComputePipelines()`.

Without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago.

In the compute shader:
```
layout( constant_id = 0 ) const int numXworkItems = 32;
layout( local_size_x = numXworkItems, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:
```
VkSpecializationMapEntry vsme[1]; // one array element for each Specialization Constant
vsme.constantID = 0; vsme.offset = 0; // # bytes into the Specialization Constant array this one item is
vsme.size = sizeof(int); // size of just this Specialization Constant
int numXworkItems = 64;
```

```
VkSpecializationInfo vsi;
vs.i.mapEntryCount = 1;
vs.i.pMapEntries = &vsme[0];
vs.i.dataSize = sizeof(int); // size of all the Specialization Constants together
vs.i.pData = &numXworkItems; // array of all the Specialization Constants
```

### Linking the Specialization Constants into the Compute Pipeline

```
VkSpecializationMapEntry vsme[1];
vsme.constantID = 0;
vsme.offset = 0;
vsme.size = sizeof(int);
int numXworkItems = 64;
```

```
VkSpecializationInfo vsi;
vs.i.mapEntryCount = 1;
vs.i.pMapEntries = &vsme[0];
vs.i.dataSize = sizeof(int);
vs.i.pData = &numXworkItems;
```

```
VkPipelineShaderStageCreateInfo vpssci;
vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0; vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = &vsi;
```

```
VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;
result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline );
```