Introduction to the Vulkan Graphics API

Mike Bailey
Oregon State University
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 Vulkan class at OSU – Winter Quarter, 2018. Thanks for your courage and patience!

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

Third, thanks to Kathleen Mattson and the Khronos Group for the great laminated Vulkan Quick Reference Cards!

Ongoing Notes and Code
The notes and code presented here are constantly being updated.
Go to:
http://cs.oregonstate.edu/~mjb/vulkan
for all the latest versions.

What Prompted the Community’s 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author.
SIGGRAPH ’18 Courses, August 12-16, 2018, Vancouver, BC, Canada
ACM 978-1-4503-5809-5/18/08.
10.1145/3214834.3214848

OpenGL 4.2 Pipeline Flowchart

OpenGL’s complexity has been hard on driver-writers.
Why is it so important to keep the GPU Busy?

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

Who is the Khronos Group?

Playing “Where’s Waldo”

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
• 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.
Vulkan Code has a Distinct “Style”

```
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, &vbci, PALLOCATOR, &Buffer);
```

```
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements(LogicalDevice, Buffer, &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, &vmai, PALLOCATOR, &MatrixBufferMemoryHandle);
```

```
result = vkBindBufferMemory(LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0);
```

Vulkan Highlights: Overall Block Diagram

Vulkan Highlights: a More Typical Block Diagram

Vulkan Quick Reference Card

Vulkan Quick Reference Card

Vulkan Synchronization

- 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
- OpenGL and OpenCL will be moving to SPIR-V as well

GLSL Source

External

GLSL Compiler

Vendor-specific code

GLSL Source

Compiler to

Driver

Advantages:

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

The Sample2017.zip File Contains This

Data Buffers

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:

def VkDataBuffer:

typedef VkBuffer

Vulkan: Buffers

Vulkan: Creating a Data Buffer

typedef VkBuffer VkDataBuffer;
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();

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

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

Finding the Right Type of Memory

```cpp
int FindMemoryThatIsHostVisible( uint32_t memoryTypeBits )
{
    VkPhysicalDeviceMemoryProperties vpdmp;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp);
    for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
    {
        VkMemoryType vmt = vpdmp.memoryTypes[i];
        VkMemoryPropertyFlags vmpf = vmt.propertyFlags;
        if( ( memoryTypeBits & (1<<i) ) != 0 )
        {
            if( ( vmpf & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )
            {
                return i;
            }
        }
    }
    return -1; // Not found
}
```

Finding the Right Type of Memory

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

Something I've Found Useful

```
It find it handy to encapsulate buffer information in a struct:

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

MyBuffer MyMatrixUniformBuffer;
```

Initializing a Data Buffer

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

Here's the C struct to hold some uniform variables

```
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

```
layout (std140, set = 0, binding = 0) uniform matBuf;
```
Filling those Uniform Variables

```cpp
glm::vec3 eye(0., 0., EYEDIST);
glm::vec3 look(0., 0., 0.);
glm::vec3 up(0., 1., 0.);
Matrices.uModelMatrix = glm::mat4( );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uNormalMatrix = glm::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 the collection of data buffer information that will be used by Vulkan.

```cpp
struct matBuf Matrices;
```

The Parade of Data

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

Creating and Filling the Data Buffer – the Details

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

Creating and Filling the Data Buffer – the Details

```cpp
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.

Filling the Data Buffer

```cpp
Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer, [void *] &Matrices );
```

Vertex Buffers
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...

What is a Vertex Buffer?

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:

\[ \text{ProjectionMatrix}[1][1] = -1. \]

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

Vulkan Topologies

VK_PRIMITIVE_TOPOLOGY_POINT_LIST

VK_PRIMITIVE_TOPOLOGY_LINE_LIST

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

A Colored Cube Example

Triangles in an Array of Structures

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

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

Vertex Orientation Issues

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".

Modelled in right-handed coordinates

Vertex Orientation Issues

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

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

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

So I recommend, at least at first, that you do no culling.
The Vulkan Pipeline

- **Vertex Input Stage**
  - Input Assembly
  - Transformation, Geometry
  - Vertex Input
  - Model View Projection

- **Dynamic State**
  - Depth/stencil
  - Vertex shader
  - Fragment Shader

- **Pipeline Layout**
  - Fragment Shader Stage
  - Color Blending Stage

- **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.

```cpp
vkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
return result;
```

```cpp
{ glm::vec2 texCoord; glm::vec3 color; glm::vec3 normal; glm::vec3 position; vvib[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX; vvib[0].stride = sizeof( struct vertex ); vvib[0].binding = 0; vvib[1].location = 1; vvib[1].layout = VK_FORMAT_VEC3; vvib[1].offset = offsetof( struct vertex, normal ); vvib[2].location = 2; vvib[2].layout = VK_FORMAT_VEC3; vvib[2].offset = offsetof( struct vertex, color ); vvib[3].location = 3; vvib[3].layout = VK_FORMAT_VEC2; vvib[3].offset = offsetof( struct vertex, texCoord ); };
```

```cpp
vkVertexInputBindingDescription vvib[1]; // one array per vertex data buffer
vkVertexInputAttributeDescription vviad[4]; // one array per vertex attribute
```

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

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

Telling the Command Buffer what Vertices to Draw

8/17/2018

Non-indexed Buffer Drawing

Triangles Represented as an Array of Structures

Filling the Vertex Buffer

C/C++:

GLSL Shader:

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

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

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

```
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,
```

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

```
Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
// fills pMyBuffer
return result;
}
Init05MyVertexDataBuffer(  sizeof(JustVertexData), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer,               (void *) JustVertexData );
Init05MyIndexDataBuffer(  sizeof(JustIndexData), &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer,                (void *) JustIndexData );
```
Sometimes the Same Point Needs Multiple Attributes
Where values match at the corners (color)
Where values do not match at the corners (texture coordinates)

Shaders and SPIR-V

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

Vulkan Shader Stages

Vulkan: GLSL Differences from OpenGL
- Verbose and Instance indices:
  - 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

Vulkan: GLSL Differences from OpenGL
- Shader combinations of separate texture data and samplers:

Describing Concatenated Brands of Separate Texture Data and Samplers:

Descriptor Set:
- layout( set=0, binding=0 )
- . . .

Push Constants:
- layout( push_constant )
- . . .

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
- gl_WorkGroupSize.z is still as it was
Vulkan: Shaders’ use of Layouts for Uniform Variables

```c
vkCreateShaderModule( )
VkShaderModuleCreateInfo( )
device
code[ ] (u_int32_t)
codeSize (in bytes)
shaderModuleCreateFlags
```

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

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

```
void (non- opaque in a uniform block)

layout( std140, 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 OpenCL will be moving to SPIR-V as well

**GLSL Compiler**

- GLSL Source
- SPIR-V Vendor-specific code
- Compiler in driver
- External

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
6. SPIR-V:

Standard Portable Intermediate Representation for Vulkan

```c
glslangValidator shaderFile -V [ -G | -S stage ] -o shaderBinaryFile.spv
```

Shaderfile extensions:

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

1. Specify stage rather than get it from shaderfile extension
2. Can be overridden by the -S option

**Complier in driver**

Vendor-specific code

**You Can Run the SPIR-V Compiler on Windows 10 from a Bash Shell**

1. Click on the Microsoft Start icon
2. Type `word` bash

**Running glslangValidator.exe**

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

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
```
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 . . .

How do you know if SPIR-V compiled successfully?

```cpp
vkResult Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule ) {
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb" );
    if( fp == NULL ) {
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str() );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC ) {
        fprintf( FpDebug, "Magic number for spir-v file '%s' is 0x%08x -- should be 0x%08x
", filename.c_str(), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char *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, &vsmci, PALLOCATOR, pShaderModule );
    fprintf( FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );
    delete [] code;
    return result;
}
```

### Caveats on the Sample Code

- I've written everything out in appalling longhand.
- Everything is in one file (except the geometry data). It really should be broken up, but this way you can read 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.
// Keyboard commands:
// 'i', 'I': Toggle using a vertex buffer only vs. a vertex/index buffer
// 'l', 'L': Toggle lighting off and on
// 'm', 'M': Toggle display mode (textures vs. colors)
// 'p', 'P': Pause the animation
// 'r', 'R': Toggle rotation-animation and using the mouse
// '1', '4', '9' Number of instances

#include "SampleVertexData.cpp"

struct vertex
{  
grm::vec3 position;  
grm::vec3 normal;  
grm::vec3 color;  
grm::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.},
  }  
  . . .
};

int main( int argc, char * argv[] )
{
  Width  = 800;
  Height = 600;
  // sample code, from: "Sample Code Keyboard Commands"
  errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
  if( err != 0 )
  {
    fprintf( stderr, "Cannot open debug print file '%s'
    FpDebug = stderr;
  }
  fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
    Reset( );
    InitGraphics( );
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
      glfwPollEvents( );
      Time = glfwGetTime( );  // elapsed time, in double-precision seconds
      UpdateScene( );
      RenderScene( );
    }
    fprintf(FpDebug, "Closing the GLFW window
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
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 typedef'd structs pass all the necessary information into a function. For example, where we might normally say in C++:

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

   we would actually say in C:

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

Vulkan Conventions

My Conventions

- **Vk**: a typedef, probably a struct
- **vk**: a function call
- **VK**: a constant

- "Init" in a function call name means that something is being setup that only needs to be setup once
- The number after "Init" gives you the ordering
- In the source code, after main() comes InitGraphics(), then all of the InitXXYYY() functions in numerical order. After that comes the helper functions
- "Find" in a function call name means that something is being looked for
- "Fill" in a function call name means that some data is being supplied to Vulkan
- "IN" and "OUT" ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, they have no significance.

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):

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

Your Sample2017.zip File Contains This

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

GLFW

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:
        Mode++;
        if( Mode >= 2 )
            Mode = 0;
        break;
    default:
        fprintf( FpDebug, "Unknow key hit: 0x%04x = '%c'
", key, key );
        fflush(FpDebug);
    }
}
}
```

Looping and Closing GLFW

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

Why are we even talking about this?

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

```c
    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:

```c
    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 …

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/0.9.8.0

You invoke GLM like this:

```c
#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.

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.

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

```c
glm::mat4 * glm::mat4
glm::mat4 * glm::vec4
glm::mat4 * glm::vec3
```

If multiplications:

```c
glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
```

If emulating OpenGL transformations with concatenation:

```c
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```
The Most Useful GLM Variables, Operations, and Functions

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

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

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

Telling Visual Studio about where the GLM folder is
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.

GLM in the Vulkan sample.cpp Program

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 Xrot is applied first, then the Yrot, 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 ) );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );

Your Sample2017.zip File Contains GLM Already
Instancing

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

Instancing – What and why?

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 gl_InstanceIndex to define a unique display property, such as position or color.

gl_InstanceIndex starts at 0

int NUM_INSTANCES = 16;
float DELTA = 3.0;
float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vColor = vec3(1., float(1.+gl_InstanceIndex) / float(NUM_INSTANCES), 0.);
xdelta -= DELTA * sqrt(float(NUM_INSTANCES)) / 2.;
ydelta -= DELTA * sqrt(float(NUM_INSTANCES)) / 2.;
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);

Making each Instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer and reference them

Still uses gl_InstanceIndex

In the vertex shader:

layout(std140, set = 3, binding = 0) uniform colorBuf {
  vec3 uColors[1024];
} Colors;

out vec3 vColor;

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

Using Descriptor Sets
Wouldn’t it be nice if we could update a bunch of related uniform variables all at once?

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?

What are Descriptor Sets?

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

- Related uniform variables can be updated as a group, gaining efficiency.
- Different values for the shaders’ uniform buffer variables can be compartmentalized into what Descriptor Sets are activated when the Command Buffer is filled. Different values for related uniform variables can be updated as a group, gaining efficiency.
- Related uniform variables can be compartmentalized into what Descriptor Sets are activated when the Command Buffer is filled. Different values for related uniform variables can be updated as a group, gaining efficiency.

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*

Uniform data in a “blob”*
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.

### Step 2: Define the Descriptor Set Layouts

- **MatrixSet DS Layout Binding**:
  - pipeline stage(s):...
  - descriptorCount:...
  - descriptorType:...
  - bindingCount:...

- **LightSet DS Layout Binding**:
  - pipeline stage(s):...
  - descriptorCount:...
  - descriptorType:...
  - bindingCount:...

- **MiscSet DS Layout Binding**:
  - pipeline stage(s):...
  - descriptorCount:...
  - descriptorType:...
  - bindingCount:...

- **TexSamplerSet DS Layout Binding**:
  - pipeline stage(s):...
  - descriptorCount:...
  - descriptorType:...
  - bindingCount:...

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

```cpp
VkResult result = vkCreatePipelineLayout( LogicalDevice, &vplci, PALLOCATOR, OUT *GraphicsPipelineLayout);
return result;
```

### Step 4: Allocating the Memory for Descriptor Sets

```cpp
vkAllocateDescriptorSets( LogicalDevice, &vdslc0, PALLOCATOR, OUT *DescriptorSet)
```
Step 4: Allocating the Memory for Descriptor Sets

```c
void* descriptorSetAllocate(void* descriptorSet, VkDescriptorSetAllocateInfo* info);
```

This struct identifies what buffer it

```c
vkCmdBindDescriptorSets
```

Step 5: Tell the Descriptor Sets where their data is

```c
void vkUpdateDescriptorSets
```

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

```c
void vkCreateGraphicsPipelineCreateInfo
```

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

```c
void vkCmdBindDescriptorSets
```
The Vulkan Graphics Pipeline

The Graphics Pipeline is like what OpenGL would call "The State", or "The Context".

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.

Here’s what you need to know:
- Create the Descriptor Set layout: on the command buffer, set the layout of the Descriptor Sets.
- Create the Push Constants layout: on the command buffer, set the layout of the Push Constants.

Vulkan: A Pipeline Records the Following Items:
- Pipeline Layout: DescriptorSet, PushConstant
- Which Shaders are going to be used?
- Per-vertex input attributes: location, binding, stride, inputRate
- Per-vertex input binding: binding, stride, inputRate
- Specialization info
- Vertex Input Stage
- Vertex Shader stage
- Fragment ShaderStage
- Color Blending Stage
- ... (other stages)
- Dynamic State
- Which states are dynamic?
- Depth/Stencil State
- Rasterization State
- MultiSample State
- Tesselation State
- Vertex Input State
- Input Assembly State
- RenderPass
- ... (other stages)

Here’s what you need to know:
- srcAlphaBlendFactor, dstAlphaBlendFactor, srcColorBlendFactor, dstColorBlendFactor, colorWriteMask, blendEnable, alphaBlendOp
- frontFace, cullMode, polygonMode, lineWidth
- stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- depthTestEnable, depthWriteEnable, depthCompareOp
- viewport: x, y, w, h, minDepth, maxDepth
- scissor: x, y, w, h

What is the Vulkan Graphics Pipeline?

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

There’s also a Vulkan Compute Pipeline.

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

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

Creating a Graphics Pipeline from a lot of Pieces

The Vulkan Graphics Pipeline is divided into several stages, each handled by different shader stages:
- Vertex Shader Stage
- Fragment Shader Stage
- Color Blending Stage
- MultiSample State
- Rasterization State
- Depth/Stencil State
- Dynamic State
- Push Constants
- Input Assembly State
- Vertex Input State
- Specialization info
- Vertex Shader module
- Fragment Shader module
- Descriptor Set
- Push Constants
- Pipeline Layout
- Vertex Binding
- Attribute Location
- Glyph
- ... (other stages)

The Graphics Pipeline Stages and what goes into them

The GPU and Driver specify the Pipeline Stages – the Vulkan Graphics Pipeline declares what goes in them:
- Vertex Input Stage
- Vertex Shader Stage
- Fragment Shader Stage
- Color Blending Stage
- Depth/Stencil State
- MultiSample State
- Rasterization State
- Dynamic State
- Push Constants
- Input Assembly Stage
- Vertex Binding
- Attribute Location
- Glyph
- ... (other stages)
Creating a Typical Graphics Pipeline

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

What is "Primitive Restart Enable"?

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

One Really Good use of Restart Enable is in Drawing Terrain

Surfaces with Triangle Strips

These are defined at the top of the sample code so that you don't need to specify them again when drawing the position, normal, and tex coords.
What is the Difference Between Changing the Viewport and Changing the Scissoring?

Viewporting operates on the entire frame buffer before the rasterizer. Changing the scissor causes the entire scene to get scaled (proportionally) into the viewport area.

Scissoring operates on fragments and takes place right after rasterization. Changing the scissor information together with the viewport and scissor information allows for fragment-level clipping and takes place right after the rasterization.

Viewports affect the entire scene, while scissors affect individual fragments. Viewports are used for scaling, while scissors are used for clipping.

Which Pipeline Variables can be Set Dynamically

Setting the Rasterizer State

Stencil Operations for Front and Back Faces

Uses for Stencil Operations
Putting It all Together! (finally…)

Group all of the individual state information and create the pipeline

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

Queues and Command Buffers

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

Vulkan: a More Typical (and Simplified) Block Diagram
Similarly, we can write a function that finds the proper queue family:

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

Creating a logical device queue needs to know queue family information:

```c
VkDeviceQueueCreateInfo vdqci[1];
float queuePriorities[] = {1.0f};
vdqci.queuePriorities = (float *)queuePriorities;
vdqci.queueCount = 1;
vdqci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci.flags = 0;
vdqci.pNext = nullptr;
vdqci.sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
result = vkCreateDeviceQueue(LogicalDevice, IN &vdqci, PALLOCATOR, OUT &Queue);
```

Creating the command pool as part of the logical device:

```c
VkCommandPoolCreateInfo vcpci;
vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
vcpci.pNext = nullptr;
vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
result = vkCreateCommandPool(LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
```

Beginning a command buffer:

```c
VkCommandBufferBeginInfo vcbbi;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.pNext = nullptr;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
result = vkBeginCommandBuffer(CommandBuffers[0], IN &vcbbi);
```
The Entire Submission / Wait / Display Process

```cpp
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 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')

What is a Swap Chain?

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.
Creating a Swap Chain

We Need to Find Out What our Display Capabilities Are

Creating the Swap Chain Images and Image Views

Rendering into the Swap Chain, I
vkCreateFenceCreateInfo 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 = 1;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;
result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence);  // 1 = submitCount

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

VkPresentInfoKHR vpi;
vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
vpi.pNext = nullptr;
vpi.waitSemaphoreCount = 0;
vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
vpi.swapchainCount = 1;
vpi.pSwapchains = &SwapChain;
vpi.pImageIndices = &nextImageIndex;
vpi.pResults = (VkResult *)nullptr;
result = vkQueuePresentKHR(presentQueue, IN &vpi);
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, imageReadySemaphore, 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;
    vcbbi.pInheritanceInfo = nullptr;
    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 = 1.0f;
    vcdsv.stencil = 0;

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

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

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

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

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

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

    vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
                             GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
    //vkCmdBindPushConstants( CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, 
    //                       offset, size, void *values );

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

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

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

VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;
VkFence renderFence;
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, &renderFence );
Triangles in an Array of Structures

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

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

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

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

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-resolution versions of the same texture so that the red texel gets included somehow in all resolution-level textures.
In addition to just picking one mip-map level, the rendering system can sample from two of them, one less that the TIP ratio and one more, and then blend the two RGBA returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin: multum in parvo, "many things in a small place"
result = VkCommandBufferBeginInfo vcbbi;

// copy pixels from the staging image to the texture:
{
    // *******************************************************************************
    // transition the staging buffer layout:
    // *******************************************************************************
    
    VkImageSubresourceLayers visl;
    VkImageMemoryBarrier vimb;
    VkImageSubresourceRange visr;
    VkImageCopy vic;

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

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

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

    visl.layerCount = 1;
    visl.mipLevel = 0;

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

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

    textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vimb);

    vic.dstOffset = vo3;
    vic.dstSubresource = visl;
    vic.srcSubresource = visl;

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

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

    textureImage, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, 1, IN &vimb);

    result = vkEndCommandBuffer( TextureCommandBuffer );
}

// create an image view for the texture image:
VkImageViewCreateInfo vivci;

vivci.subresourceRange = visr;
vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;
vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vivci.image = textureImage;
vivci.flags = 0;

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

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.
Vulkan: Identifying the Physical Devices

Vulkan: Overall Block Diagram

Vulkan: Which Physical Device to Use, I

Vulkan: Asking About the Physical Device's Features

Vulkan: Which Physical Device to Use, II

Vulkan: Querying the Number of Physical Devices

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

There are

```
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```

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

```c
uint32_t count;

result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
```
Here's What the NVIDIA 1080Ti Produced

```
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

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

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

Asking About the Physical Device's Different Memories

```
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
fprintf( FpDebug, "\n%d Memory Types:\n", vpdmp.memoryTypeCount);
for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ ){
  VkMemoryType vmt = vpdmp.memoryTypes[i];
  fprintf( FpDebug, "Memory %2d: ", i );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT       ) != 0 )    fprintf( FpDebug, " DeviceLocal" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT       ) != 0 )    fprintf( FpDebug, " HostVisible" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT      ) != 0 )    fprintf( FpDebug, " HostCoherent" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT        ) != 0 )    fprintf( FpDebug, " HostCached" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT   ) != 0 )    fprintf( FpDebug, " LazilyAllocated" );
  fprintf(FpDebug, "\n" );
}fprintf( FpDebug, "\n%d Memory Heaps:\n", vpdmp.memoryHeapCount );
for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ){
  fprintf(FpDebug, "Heap %d: ", i);
  VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
  fprintf( FpDebug, " size = 0x%08lx " , (unsigned long int)vmh.size );
  if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT  ) != 0 )     fprintf( FpDebug, " DeviceLocal" );
  fprintf(FpDebug, "\n" );
}```

Here's What I Got

```
11 Memory Types:
  Memory 0:
  Memory 1:
  Memory 3:
  Memory 4:
  Memory 5:
  Memory 6:
  Memory 7: DeviceLocal
  Memory 8: DeviceLocal
  Memory 9: HostVisible HostCoherent
  Memory 10: HostVisible HostCoherent HostCached
2 Memory Heaps:
  Heap 0: size = 0x00000000 DeviceLocal
  Heap 1: size = 0x00000000
```

Asking About the Physical Device's Queue Families

```
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
fprintf( FpDebug, "\nFound %d Queue Families:\n", count );
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
for( unsigned int i = 0; i < count; i++ ){
  fprintf( FpDebug, "\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
```

Asking About the Physical Device's Queue Families
Logical Devices

Vulkan: Overall Block Diagram

Vulkan: Creating a Logical Device

Vulkan: Creating the Logical Device's Queue

Vulkan: Specifying a Logical Device Queue

Layers and Extensions
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 );

vkEnumerateInstanceLayerProperties:
13 instance layers enumerated:
0x00400033   2  'VK_LAYER_LUNARG_api_dump' 'LunarG debug layer'
0x00400033   1  'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_monitor' 'Execution Monitoring Layer'
0x00400033   1  'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_screenshot' 'LunarG image capture layer'
0x00400033   1  'VK_LAYER_LUNARG_standard_validation' 'LunarG Standard Validation'
0x00400033   1  'VK_LAYER_GOOGLE_threading' 'Google Validation Layer'
0x00400033   1  'VK_LAYER_GOOGLE_unique_objects' 'Google Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_vktrace' 'Vktrace tracing library'
0x00400038   1  'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x0040000d   1  'VK_LAYER_NV_nsight'  'NVIDIA Nsight interception layer'
0x00400000  34  'VK_LAYER_RENDERDOC_Capture' 'Debugging capture layer for RenderDoc'

vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_EXT_display_surface_counter'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_surface_capabilites2'
0x00000005  'VK_KHR_surface_capabilites'
0x00000001  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_win32_surface_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'

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

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

vkEnumerateDeviceLayerProperties:
3 physical device layers enumerated:
0x00400038  1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033  1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00400003  1  'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'

vkEnumerateDeviceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_KHR_surface'
0x00000001  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_surface_capabilities'

Will now ask for only 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_KHR_debug_report
result = vkEnumeratePhysicalDevices(Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice*)nullptr);

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

int discreteSelect = -1;
int integratedSelect = -1;

for (unsigned int i = 0; i < PhysicalDeviceCount; i++) {
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties(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;
else if (integratedSelect >= 0)
    which = integratedSelect;
PhysicalDevice = physicalDevices[which];
delete[] physicalDevices;

vkGetPhysicalDeviceProperties(PhysicalDevice, OUT &PhysicalDeviceProperties);
vkGetPhysicalDeviceFeatures(PhysicalDevice, OUT &PhysicalDeviceFeatures);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, VK_FORMAT_R32G32B32A32_SFLOAT, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, VK_FORMAT_R8G8B8A8_UNORM, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, VK_FORMAT_B8G8R8A8_UNORM, &vfp);

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

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

float queuePriorities[NUM_QUEUES_WANTED] = {
    1.0f
};
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];

for (int i = 0; i < NUM_QUEUES_WANTED; i++) {
    vdqci[i].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    vdqci[i].pNext = nullptr;
    vdqci[i].flags = 0;
    vdqci[i].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    vdqci[i].queueCount = 1;                // how many queues to create
    vdqci[i].pQueuePriorities = queuePriorities;    // array of queue priorities [0.,1.]
}

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

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

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

Synchronization
Vulkan Highlights: Overall Block Diagram

Instance

Physical Device

Application

Logical Device

Logical Device

Command Buffer

Event

Semaphore

Fence

Host

• 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 semaphore
• You don’t end up setting, resetting, or checking the semaphore yourself
• Semaphores must be initialized (“created”) before they can be used

Ask for Something

Try to Use the Something

Semaphore

Creating a Semaphore

VkSemaphoreCreateInfo

vkSemaphoreCreateInfo

vkCreateSemaphore

VkSemaphore

Creating a Semaphore Example during the Render Loop

Semaphore

Fences

Fences

Fence

Could be an array of fences
Controlling Events from the Device

- 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

Events

- Events provide even finer-grained synchronization
- 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

From the Command Buffer Notes:

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

vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, vkCmdNextSubpass( commandBuffer, contents );

vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask) 

vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );

vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, 

vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );

vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );

vkCmdSetStencilReference( commandBuffer, faceMask, reference );

vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );

vkCmdSetLineWidth( commandBuffer, lineWidth );

vkCmdSetEvent( commandBuffer, event, stageMask );

vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );

vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );

vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );

vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );

vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );

vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );

vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );

vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );

vkCmdResetEvent( commandBuffer, event, stageMask );

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 (RR) as no data has been changed

Where in the Pipeline is this Memory being Accessed?

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_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Pipeline Stage Flags –

Note: the concept of an in-order pipeline is accurate, but really the src and dst triggering action only depends on the name of the street where you are right now

A Pipeline Barrier is a way to establish a memory dependency between commands that were submitted before the barrier and commands that are submitted after the barrier

vkCmdPipelineBarrier( commandBuffer, 

memoryBarrierCount, pBufferMemoryBarriers, 

bufferMemoryBarrierCount, pBufferMemoryBarriers, 

imageMemoryBarrierCount, pimageMemoryBarriers );

Defines what data we will be blocking on blocking on

 Guarantees that the pipeline stage has completely finished with one set of data before

VK_DEPENDENCY_BY_REGION_BIT

allowing this pipeline stage to proceed with the next set of data
Access Masks – What are you interested in Generating or Consuming this Memory for?

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_MEMORY_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESSTRANSFER_WRITE_BIT
- VK_ACCESSTRANSFER_READ_BIT
- VK_ACCESSDEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESSDEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESSINDIRECT_COMMAND_READ_BIT

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

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

The Scenario

dst cars are generating the image
src cars are doing something with that image

Access Operations and what Pipeline Stages they can be used in

Push Constants

VK_ACCESS_MEMORY_WRITE_BIT
VK_ACCESS_MEMORY_READ_BIT
VK_ACCESS_HOST_WRITE_BIT
VK_ACCESS_HOST_READ_BIT
VK_ACCESS_TRANSFER_WRITE_BIT
VK_ACCESS_TRANSFER_READ_BIT
VK_ACCESSDEPTH_STENCIL_ATTACHMENT_WRITE_BIT
VK_ACCESSDEPTH_STENCIL_ATTACHMENT_READ_BIT
VK_ACCESS_SHADER_WRITE_BIT
VK_ACCESS_SHADER_READ_BIT
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
VK_ACCESS_UNIFORM_READ_BIT
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
VK_ACCESS_INDEX_READ_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

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 at least 4 GB in the NVIDIA 1080ti. (You can query this limit by looking at the maxPushConstantSize parameter in the VkPhysicalDeviceLimits structure.) Unlike uniform buffers and vertex buffers, these are not backed by memory. They are actually part of the Vulkan pipeline.

Push Constants

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

```glsl
layout (push_constant) uniform matrix modelMatrix;
```

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

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

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

Setting up the Push Constants for the Pipeline Structure

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

```cpp
VkPushConstantRange vpcr[1] = {
    .binding = 0,
    .stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT,
    .offset = 0,
    .size = sizeof(glm::mat4),
};

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

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

An Robotic Example using Push Constants

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

Where each arm is represented by:

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

struct armArm1;
struct armArm2;
struct armArm3;

In the Reset Function

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

VkPushConstantRange
vpcr[1];

vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

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

VkPipelineLayoutCreateInfo
vplci;

type = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

pNext = nullptr;

flags = 0;

setLayoutCount = 4;

pSetLayouts = DescriptorSetLayouts;

pushConstantRangeCount = 1;

pPushConstantRanges = vpcr;

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


In the UpdateScene Function

float rot1 = (float)Time;

float rot2 = 2.f * rot1;

float rot3 = 2.f * rot2;

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

glm::mat4 m1g = glm::mat4();

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

m1g = glm::rotate(m1g, rot1, zaxis);

m1g = glm::translate(m1g, glm::vec3(2.*Arm1.armScale, 0., 0.));

m1g = glm::rotate(m1g, rot2, zaxis);

m1g = glm::translate(m1g, glm::vec3(0., 0., 2.));

m1g = glm::translate(m1g, glm::vec3(2.*Arm2.armScale, 0., 0.));

m1g = glm::rotate(m1g, rot3, zaxis);

m1g = glm::translate(m1g, glm::vec3(0., 0., 2.));

Arm1.armMatrix = m1g; // m1g

Arm2.armMatrix = m1g * m21; // m2g

Arm3.armMatrix = m1g * m21 * m32; // m3g


In the RenderScene Function

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

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

vkCmdBindConstantBufferBase(imageIndex, 0, MyVertexDataBuffer.buffer);

vkCmdBindPipeline(imageIndex, PipelineLayout);

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

vkCmdBindPipeline(imageIndex, GraphicsPipelineLayout);

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

vkCmdBindPipeline(imageIndex, GraphicsPipelineLayout);

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

Antialiasing and Multisampling

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 to this:

1. **Supersampling**: Pick some number of unique points within a pixel, render the image into each of these sub-pixels (including depth and stencil tests), then average them together.

2. **Multisampling**: Pick some number of unique points within each pixel and perform a depth and stencil render there. Then, perform a single color render for that pixel. Assign that RGBA to all the sub-pixels that made it through the depth and stencil tests.
Vulkan Distribution of Sampling Points within a Pixel

Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

Multisampling

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 = 0.5;
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

Final Pixel Color = \sum \text{Color sample from subpixel, } i

# Fragment Shader calls = 8
Setting up the Image

```cpp
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:

```cpp
VlOffset3D 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.
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.

```c
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[2].format = VK_FORMAT_B8G8R8A8_SRGB;

vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
vad[1].flags = 0;

vad[0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;

vad[2].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vad[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[2].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[2].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[2].samples = VK_SAMPLE_COUNT_1_BIT;
vad[2].flags = 0;
```

Subpass #0:
- Color Attachment
- Depth Attachment
- Output

Subpass #1:
- Image Processing Pass

Subpass #2:
- 3D Rendering Pass

**Multipass Algorithm to Render and then Image Process**

**Multipass, I**

```c
vrpci.pAttachments = vrpci.attachmentCount = 3; // color, depth/stencil, output
vrpci.flags = 0;
vrpci.pNext = nullptr;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pSubpasses = vrpci.subpassCount = 2;

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

**Multipass, II**

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

**Multipass, III**

```c
outputReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
outputReference.attachment = 2;
```

**Multipass, IV**

```c
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
deepthReference.attachment = 1;
```

**Multipass, V**

```c
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
colorReference.attachment = 0;
```
Placing a Pipeline Barrier so an Image is not used before it is Ready

```cpp
vkImageMemoryBarrier
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);
```

Multipass, V

```cpp
vkCmdBeginRenderPass
(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// first subpass is automatically started here
vkCmdBindPipeline
(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
vkCmdBindDescriptorSets
(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *) nullptr);
vkCmdBindVertexBuffers
(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
vkCmdDraw
(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
...

vkCmdNextSubpass
(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// second subpass is started here – doesn’t need any new drawing vkCmd’s
...

vkCmdEndRenderPass
(CommandBuffers[nextImageIndex]);
```

A Wrap-up: Here are some good Vulkan References (there are probably more by now – check the SIGGRAPH Bookstore)


The notes and code presented here are constantly being updated. Go to:

```
http://cs.oregonstate.edu/~mjb/vulkan
```
for all the latest version.

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