Introduction to the Vulkan Graphics API

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

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.

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

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!

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.
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”

Who’s Been Specifically Working on Vulkan?

Vulkan

• Largely derived from AMD’s Mantle API
• Also heavily influenced by Apple’s Metal API and Microsoft’s DirectX 12
• Goal: much less driver complexity and overhead than OpenGL has
• Goal: much less user hand-holding
• 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.
VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = << buffer size in bytes >>
    vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = nullptr;
    VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );

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

    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.flags = 0;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = 0;
    result = vkAllocateMemory ( LogicalDevice, IN &vmai, PALLOCATOR, &MatrixBufferMemoryHandle );
    result = vkBindBufferMemory ( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );

Vulkan Code has a Distinct “Style”
Vulkan Shaders

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

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

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

The Sample2017.zip File Contains This

The Sample2017.zip File Contains This

Vulkan: Buffers

Vulkan: Creating a Data Buffer
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;
}
```

Finding the Right Type of Memory

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

```
MyBuffer MyMatrixUniformBuffer;
```

```
I find it handy to encapsulate buffer information in a struct:
```
```
typedef struct MyBuffer
{
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;
```

```
MyBuffer MyMatricesUniformBuffer;
```

Something I’ve Found Useful

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

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

Initializing a Data Buffer

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

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

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 (std40, binding = 0) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
} Matrices;
```
Filling those Uniform Variables

```cpp
glm::vec3 eye(0., 0., EYEDIST);
glm::vec3 look(0., 0., 0.);
glm::vec3 up(1., 0., 0.);
Matrices.uModelMatrix = glm::mat4(0.); // 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.

```cpp
struct matBuf Matrices;
```

The Parade of Data

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
MyBuffer MyMatrixUniformBuffer;
```

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

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.
What is a Vertex Buffer?

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

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

First, a quick review of computer graphics geometry...

**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:

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

This is like saying \( Y' = -Y \).

**Triangles in an Array of Structures**

From the file `SampleVertexBuffer.cpp`:

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

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

**A Colored Cube Example**

This object was modeled such that triangles that face the viewer still 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.
Filling the Vertex Buffer

MyBuffer MyVertexDataBuffer;

VkResult FillDataBuffer( MyVertexDataBuffer, (void *) VertexData );

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.

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

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.

// 4 = vertex, normal, color, texture coord

struct vertexInputBindingDescription
{
    uint binding;          // which binding # this is
    uint index;            // byte between successive structs
    uint stride;           // bytes between successive structs
    uint location;         // location in the layout decoration
};

struct vertexInputAttributeDescription
{
    uint binding;          // which binding # this is part of
    uint offset;           // which # vertex data struct
    uint inputRate;        // offset in struct vertex
    uint location;         // location in the layout decoration
    uint divisor;          // number of data structs
    uint locationInPacked; // in packed data
    uint locationInData;   // in data
    uint locationInStage;  // in stage
};

Pipeline Layout

vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN & GraphicsPipelineCreateInfo, OUT pGraphicsPipeline );

Pipeline Layout

vkCreateVertexInputStateCreateInfo( logicalDevice, &vvibd, OUT ppVertexInputStateCreateInfo );

Pipeline Layout

vkCreatePipelineLayout( logicalDevice, &vpvisci, OUT ppPipelineLayout );

Pipeline Layout

vkCreatePipeline( LogicalDevice, &vgpci, OUT ppPipeline );

Pipeline Layout

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.

Pipeline Layout

vkCreatePipeline( LogicalDevice, &vgpci, OUT ppPipeline );
8/1/2018

Filling the Vertex Buffer

MyBuffer MyVertexDataBuffer;

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

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

GLSL Shader:

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

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

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer }; 
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```
Telling the Command Buffer what Vertices to Draw

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

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

void bindBuffers(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.

Drawing with an Indexed Buffer

```c
void bindBuffers(CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );

vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], indexDataBuffer, indexOffset, 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 vertexOffset = 0;

# ifdef VERTEX_BUFFER
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
# endif

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

Sometimes the Same Point Needs Multiple Attributes

A color-interpolated cube like this actually has both. Point #7 above has the same color, regardless of what face it is in. However, Point #7 has 2 different normal vectors, depending on which face you are defining. Same with its texture coordinates.

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

Where values match at the corners (color)

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

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.

Shaders and SPIR-V

Vulkan Shader Stages

Vulkan: GLSL Differences from OpenGL

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

- In the compiler, there is an automatic #define VULKAN 100

Vertex and Instance indices:

- gl_VertexIndex
- gl_InstanceIndex

Both are 0-based

gl_FragColor:

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

Vulkan: GLSL Differences from OpenGL

Shader combinations of separate texture data and samplers:

uniform sampler s;
uniform texture2D t;
vec4 rgba = texture(  sampler2D( t, s ), vST );

Descriptor Sets:

layout( set=0, binding=0 ) . . .  ;

Specialization Constants:

layout( constant_id = 3 )  const int N = 5;

• Can only use basic operators, declarations, and constructors
• Only for scalars, but a vector can be constructed from specialization constants

Specialization Constants for Compute Shaders:

layout(local_size_x=8, local_size_y=16);

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

```cpp
vkCreateShaderModule(
  device,
  VkShaderModuleCreateInfo(
    code[ u_int32_t ] (codeSize in bytes),
    shaderModuleCreateFlags
  ),
  &lightBuf,
  layout( std140, set = 0, binding = 0 ) uniform matBuf
  
  layout( std140, set = 1, binding = 0 ) uniform lightBuf
  
  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

---

SPIR-V:

Standard Portable Intermediate Representation for Vulkan

```
glslangValidator shaderFile -V […] -S <stage> -o shaderBinaryFile.spv
```

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

(Can be overridden by the –S option)

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

Windows: glslangValidator.exe
Linux: `setenv LD_LIBRARY_PATH /usr/local/common/gcc-6.3.0/lib64/`

---

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

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

---

You can also run SPIR-V from a Linux Shell

```
$ glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
$ glslangValidator.exe -V sample-frag.frag -o sample-frag.spv
```
How do you know if SPIR-V compiled successfully?

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

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

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

```
0203 0723 . . .
```

How do you know if SPIR-V compiled successfully?

```c
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;
}
```

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

### Caveats on the Sample Code

- I've written everything out in appalling longhand.
- Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.
- At times, I could have hidden complexity, but I didn't. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that's a secret to you.
- I've setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
- At times, I've setup things that didn't need to be setup just to show you what could go there.
- There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
- I've typedef'ed 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:
// 'y', 'Y': 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
// 'q', 'Q': Esc: exit the program
// 'r', 'R': Toggle rotation-animation and using the mouse
// '1', '4', '9': Number of instances

```c
#include "SampleVertexData.cpp"

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

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

The Vertex Data is in a Separate File

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums
2. Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary
   information into a function. For example, where we might normally say in C++:

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

   we would actually say in C:

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

**Vk::** is a typedef, probably a struct

**vk::** is a function call

**VK::** is a constant

**My Conventions**

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

The number after “Init” gives you the ordering. In the source code, after main() comes InitGraphics(), then all of the Init*XXX() 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.

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

Where to put them

How many total

there are

Where to put them:

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

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

Your Sample2017.zip File Contains This

- Linux shader compiler
- Windows shader compiler
- GLFW

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

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 )
{
    switch( key )
    {
        case GLFW_KEY_M:
        case 'm':
        case 'M':
            Mode++;
            if( Mode >= 2 ) Mode = 0;
            break;
        default:
            fprintf( FpDebug, "Unknow key hit: 0x%04x = '%c'
", key, key );
            fflush(FpDebug);
    }
}
```
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );
    Time = glfwGetTime( );         // elapsed time, in double-precision seconds
    UpdateScene( );
    RenderScene( );
}

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

---

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 invoke GLM like this:

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

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

You can find it at: [http://glm.g-truc.net/0.9.8.5/](http://glm.g-truc.net/0.9.8.5/)

---

Why are we even talking about this?

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

```cpp
    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:

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

Exactly the same concept, but a different expression of it. Read on for details …

---

The Most Useful GLM Variables, Operations, and Functions

// constructor:

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

```cpp
    glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
    glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
    glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```
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

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

2. Telling Visual Studio about where the GLM folder is

3. 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

GLM in the Vulkan sample.cpp Program

Instancing
Instancing – What and why?

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

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

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

Making each Instance look differently – Approach #1

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

gl_InstanceIndex starts at 0

In the vertex shader:

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

In the vertex shader:

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

Still uses gl_InstanceIndex

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?

In OPENGL

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

Our example will assume the following shader uniform variables:

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

```cpp
layout( std140, set = 1, binding = 0 ) uniform lightBuf

layout( std140, set = 2, binding = 0 ) uniform lightBuf
```

```cpp
layout( std140, set = 3, binding = 0 ) uniform miscBuf
```

```cpp
// non-opaque must be in a uniform block:
layout( set = 3, binding = 0 ) uniform matBuf
```

```cpp
// non-opaque must be in a uniform block:
layout( set = 2, binding = 0 ) uniform matBuf
```

```cpp
// non-opaque must be in a uniform block:
layout( set = 1, binding = 0 ) uniform matBuf
```

```cpp
// non-opaque must be in a uniform block:
layout( set = 0, binding = 0 ) uniform matBuf
```

```cpp
layout( std140, set = 1, binding = 0 ) uniform lightBuf
```

```cpp
light:
```

```cpp
LightBuf
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```

```cpp
light:
```

```cpp
Light:
```
Step 1: Define the Descriptor Set Layouts

```c
VkDescriptorSetLayoutCreateInfo vdslc3;
vdslc3.pBindings = nullptr;
vdslc3.bindingCount = 0;
vdslc3.pNext = nullptr;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vdslc2;
vdslc1;
vdslc0;
```

Step 2: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```c
VkPipelineLayoutCreateInfo vplci;
```

Step 3: Allocating the Memory for Descriptor Sets

```c
VkResult result = vkAllocateDescriptorSets(logicalDevice, &vdsai, &DescriptorSets[0]);
```

Step 4: Allocating the Memory for Descriptor Sets

```c
VkResult result = vkAllocateDescriptorSet(pool, &dsai, &DescriptorSet);`
Step 5: Tell the Descriptor Sets where their CPU Data is

```c
// This struct identifies what buffer it points to
struct VkBufferView
{...

// This struct identifies what texture sampler and image view it owns
struct VkSampler
{...

// This struct links a Descriptor Set to the buffer it is pointing to
struct VkDescriptorBufferInfo
{...}
```

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

```c
// This struct links a Descriptor Set to the buffer it is pointing to
struct VkWriteDescriptorSet
{...
```

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

```c
vkCmdBindDescriptorSets
{...}
```
What is the Vulkan Graphics Pipeline?

The Vulkan Graphics Pipeline is similar to what OpenGL would call "The State", or "The Context".

There is also a Vulkan Compute Pipeline.

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.

Creating a Graphics Pipeline from a lot of Pieces

Creating a Typical Graphics Pipeline

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 changes.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

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

Vulkan: A Pipeline Records the Following Items:

Pipeline Layout: Descriptor Sets, Push Constants
Which Shaders are going to be used
Per-vertex input attributes: location, binding, format, offset
Per-vertex input bindings: binding, stride, inputRate
Assembly: topology
Viewport x, y, w, h, minDepth, maxDepth
Screening x, y, w, h
Rasterization: cullMode, polygonMode, frontFace, lineWidth
Depth: depthTestEnable, depthWriteEnable, depthCompareOp
Stencil: stencilTestEnable, stencilEnableFront, stencilEnableBack
Banding: blendEnable, blendFactor, blendOp, colorWriteMask
Dynamic State: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

Bold/Italics indicates that this state item can also be set with Dynamic Variables.
Link in the Shaders

- Use one vpssci array member per shader module you are using
- Use one vvibd array member per vertex input array-of-structures you are using

Link in the Per-Vertex Attributes

- Use one vviad array member per element you are using as vertex input

What is “Primitive Restart Enable”?  

“Restart Enable” is used with:  
- Indexed drawing  
- Triangle Fan and *Strip topologies

If vpssci::primitiveRestartEnable is VK_TRUE, then a special “index” indicates that the primitive should start over. This is more efficient than explicitly starting a new primitive of the same type.

One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

- Declare the viewport information
- Declare the scissor information
- Group the viewport and scissor information together

---

**Note:** The document contains various sections discussing computer graphics concepts, particularly those related to vertex input, primitive topology, and shader modules. It includes diagrams and code snippets to illustrate these concepts. The text is technical and detailed, focusing on the specifics of rendering techniques in computer graphics. The diagrams are used to visually represent the flow of data and the structure of the shaders and vertex input systems.
What is the Difference Between Changing the Viewport and Changing the Scissoring?

Viewport scaling on vertices and takes place right before the rasterizer. Changing the vertical part of the viewport causes the entire scene to get scaled (enlarged) into the viewport area.

Scissoring operates on fragments and takes place right after the rasterizer. Scissors causes the entire scene to get clipped where it falls outside the scissors area.

Which Pipeline Variables can be Set Dynamically

Declaring information about how the rasterization will take place

Stenciling/Stencil

Stencil Operations for Front and Back Faces

Uses for Stenciling

Putting it all Together (Finally…)

VkPipeline CreateCreateInfo: 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

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

Queues and Command Buffers

Vulkan: a More Typical (and Simplified) Block Diagram

- Application
- Instance
- Physical Device
- Logical Device
  - Command Buffer
  - Command Buffer
  - Command Buffer

Vulkan Queues and Command Buffers

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

CPU Thread

buffer
queue
buffer
queue
buffer
queue

Querying what Queue Families are Available

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

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

Similarly, we Can Write a Function that Finds the Proper Queue Family

```cpp
int FindQueueFamilyThatDoesGraphics( )
{
    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
    vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT vqfp);
    for( unsigned int i = 0; i < count; i++ )
    {
        if( (vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0 )
            return i;
    }
    return -1;
}
Creating a Logical Device Queue Needs to Know Queue Family Information

```c
VkDeviceQueueCreateInfo vdqci[1];
{
    queueIndex = 0;
    queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    Queue;
}

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

Creating the Command Buffers

```c
vkAllocateCommandBuffers( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue
);
```

LogicalDevice, queueFamilyIndex, queueIndex,  OUT &Queue
);
These are the Commands that could be entered into the Command Buffer, II

- `vkCmdResolveImage` for resolving images
- `vkCmdSetViewportWScalingNV` for viewport scaling
- `vkCmdReserveSpaceForCommandsNVX` for reserving space
- `vkCmdPushDescriptorSetWithTemplateKHR` for descriptor push
- `vkCmdPushDescriptorSetKHR` for descriptor push
- `vkCmdPushConstants` for constant push
- `vkCmdProcessCommandsNVX` for command processing
- `vkCmdSetBlendConstants` for blending constants
- `vkCmdWriteTimestamp` for timestamp writing
- `vkCmdSetStencilMask` for stencil mask setting
- `vkCmdSetViewport` for viewport setting
- `vkCmdSetStencilReference` for stencil reference setting
- `vkCmdSetEvent` for event setting
- `vkCmdPipelineBarrier` for pipeline barrier
- `vkCmdNextSubpass` for subpass transition

- `VkClearValue vcv[2]`
- `VkClearColorValue vccv`
- `VkClearDepthStencilValue vcdsv`

- `vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
vrpbi.pClearValues = vcv;` for setting clear values
- `vrpbi.renderArea = r2d;` for rendering area
- `vrpbi.framebuffer = Framebuffers[nextImageIndex];` for framebuffer
- `vrpbi.renderPass = RenderPass;` for render pass

- `VkSubmitInfo vsi` for submit info
- `vsi.commandBufferCount = 1;` for command buffer count
- `vsi.signalSemaphoreCount = 0;` for signal semaphore count
- `vsi.waitSemaphoreCount = 1;` for wait semaphore count

- `result = vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue);` for getting present queue
- `result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);` for creating semaphore
- `result = vkCreateFence(LogicalDevice, IN &vfci, PALLOCATOR, OUT &renderFence);` for creating fence
- `vkCreateRenderPass(LogicalDevice, IN &vrrpi, PALLOCATOR, OUT &renderPass);` for creating render pass

- `vkWaitForFences(CommandBuffers[nextImageIndex], 0, 1, IN &renderFence);` for waiting fences
- `vkQueueSubmit(LogicalDevice, CommandBuffers[nextImageIndex], 0, 1, IN &vsi, IN renderFence);` for submitting
- `vkQueueWaitIdle(LogicalDevice, OUT &nextImageIndex);` for waiting idle

Submiting a Command Buffer to a Queue for Execution

The Entire Submission / Wait / Display Process
The Swap Chain

How We Think of OpenGL Framebuffers

Video
Driver
Front
Back
Double-buffered
Color Framebuffers
Update
Refresh

Vulkan Thinks of it This Way

Front
Back
Depth-Buffer
Update
Swap Chain

What is a Swap Chain?

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

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

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

After creating the Swap Chain in the first place, the process for using the Swap Chain is:
1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to "front buffer")

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

```c
vkGetSwapchainImagesKHR
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```

Creating the Swap Chain Images and Image Views

```c
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```

Rendering into the Swap Chain, I

```c
vkGetSwapchainImagesKHR
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```

Rendering into the Swap Chain, II

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

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

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

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

We Need to Find Out What our Display Capabilities Are

```c
vkAcquireNextImageKHR
uint64_t timeout = UINT64_MAX;
result = vkCreateSemaphore( LogicalDevice, &vsci, allocator, &imageReadySemaphore );
```

Creating a Swap Chain

```c
vkGetSwapchainImagesKHR
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```

Rendering into the Swap Chain, I

```c
vkGetSwapchainImagesKHR
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```

Rendering into the Swap Chain, II

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

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

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

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

Creating the Swap Chain

```c
vkAcquireNextImageKHR
uint64_t timeout = UINT64_MAX;
result = vkCreateSemaphore( LogicalDevice, &vsci, allocator, &imageReadySemaphore );
```

Creating the Swap Chain Images and Image Views

```c
vkGetSwapchainImagesKHR
for (int i = 0; i < imageCount; i++)
    PresentImages[i] = swapchainImages[i];

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.surface = Surface;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;

result = vkCreateSwapchainKHR( logicalDevice, &vscci, allocator, &SwapChain );
```
Rendering into the Swap Chain, III

```c
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );
vkPresentInfoKHR vpi;
    vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    vpi.pNext = nullptr;
    vpi.waitSemaphoreCount = 0;
    vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
    vpi.swapchainCount = 1;
    vpi.pSwapchains = &SwapChain;
    vpi.pImageIndices = &nextImageIndex;
    vpi.pResults = (VkResult *) nullptr;
result = vkQueuePresentKHR( presentQueue, IN &vpi );
```

Rendering into the Swap Chain, IV

1. `vkBeginRenderPass()`
2. `vkCmdBindPipeline( CommandBuffer, … )`
3. `vkCmdSetxxx ( CommandBuffer, yyy )`  
    dynamic states
4. `vkCmdBindDescriptorSets( CommandBuffer, … )`, which also includes Push Constants
5. `vkCmdBindVertexBuffers( CommandBuffer, … )`
6. `vkCmdDraw( CommandBuffer, vertexCount, instanceCount, firstVertex, firstInstance )`
7. `vkEndRenderPass( )`

Vulkan: Rendering

Vulkan: Submitting to a Queue
VkClearColorValue vccv;
    vccv.float32[0] = 0.0;
    vccv.float32[1] = 0.0;
    vccv.float32[2] = 0.0;
    vccv.float32[3] = 1.0;

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

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

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

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

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

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

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

vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &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);

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, OUT &renderFence);

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);

vkDestroySemaphore(LogicalDevice, imageReadySemaphore, PALLOCATOR);


Textures

Triangles in an Array of Structures
VkDescriptorSetLayoutBinding TexSamplerSet[1];
TexSamplerSet[0].binding = 0;
TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
// uniform sampler2D uSampler
// vec4 rgba = texture(uSampler, vST);
TexSamplerSet[0].descriptorCount = 1;
TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;

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

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

Texture RGBA

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 textures and somehow include the red texel in all resolution-level textures.
Init07TextureBuffer(INOUT MyTexture * pMyTexture)

VkResult
result = vkCreateSampler(LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler);

VkSamplerCreateInfo vsci;

vsci.unnormalizedCoordinates = VK_FALSE;        // VK_TRUE means we are use raw texels as the index
vsci.minLod = 0.;                             // initial value
vsci.maxLod = 1.;                             // value during lookups
vsci.compareOp = VK_COMPARE_OP_NEVER;         // VK_FALSE means we are using the usual 0. - 1.
vsci.compareEnable = VK_FALSE;

vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.pNext = nullptr;

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

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

VkImageCreateInfo vici;

vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

vici.samples = VK_SAMPLE_COUNT_1_BIT;

vici.arrayLayers = 1;

vici.extent.width = texWidth;

vici.extent.height = texHeight;

vici.format = VK_FORMAT_R8G8B8A8_UNORM;

vici.flags = 0;

vici.pNext = nullptr;

result = vkCreateImage(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

// this second {...} is to create the actual texture image:

// ****************************************************************************
// *******************************************************************************
// we have now created the staging image -- fill it with the pixel data:

VkMemoryAllocateInfo vmai;

if (Verbose)

vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);

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

unsigned char *gpuBytes = (unsigned char *)gpuMemory;

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

vkUnmapMemory(LogicalDevice, gpuMemory);

vkBindImageMemory(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

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

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

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

VkImageSubresourceRange visr;

visr.layerCount = 1;

visr.baseArrayLayer = 0;

visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

visr.baseMipLevel = 0;

visr.levelCount = 1;

visr.sizes[0].width = texWidth;

visr.sizes[0].height = texHeight;

visr.sizes[0].depth = 1;

vkImageMemoryBarrier(LogicalDevice, /*sourceStageMask = */VK_PIPELINE_STAGE_HOST_BIT, /*sourceAccessMask = */0,

VK_PIPELINE_STAGE_TRANSFER_BIT, /*destinationStageMask = */VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,

VK_ACCESS_SHADER_WRITE_BIT, /*destinationAccessMask = */0,

/*imageLayout = */VK_IMAGE_LAYOUT_TRANSFER_SRC_BIT,

/*srcSubresource = */visr,

/*srcOffset = */0,

/*dstSubresource = */vsl,

/*dstOffset = */0,

/*transitionType = */VK_IMAGE_LAYOUT_TRANSITION_BIT,

/*transitionTarget = */VK_IMAGE_LAYOUT_TRANSFER_DST_BIT,

/*transitionSource = */VK_IMAGE_LAYOUT_TRANSFER_SRC_BIT,

/*sourceQueueFamilyIndex = */0,

/*destQueueFamilyIndex = */0,

/*sourceAccessMask = */0,

/*destinationAccessMask = */0);
result = vkEndCommandBuffer( TextureCommandBuffer );

result = VkExtent3D                              ve3;
result = VkOffset3D                              vo3;
result = VkImageSubresourceLayers visl;

// now do the final image transfer:
result = vkCmdCopyImage(TextureCommandBuffer, stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);

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

vkCmdPipelineBarrier(TextureCommandBuffer,
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;

vimb.subresourceRange = visr;
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_TRANSFER_DST_OPTIMAL;
vimb.pNext = nullptr;

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

vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;

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_TRANSFER_DST_OPTIMAL;
vimb.pNext = nullptr;

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

vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;

vimb.srcAccessMask = 0;
vimb.image = textureImage;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vimb.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vimb.pNext = nullptr;

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

vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;

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_TRANSFER_DST_OPTIMAL;
vimb.pNext = nullptr;

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

vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;
vkCmdPipelineBarrier(TextureCommandBuffer,
VkImageMemoryBarrier vimb;

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_TRANSFER_DST_OPTIMAL;

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.
Querying the Number of Physical Devices

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

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

Which Physical Device to Use, I

```c
int integratedSelect = -1;
int discreteSelect = -1;

for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    if( result != VK_SUCCESS )
        result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
    int which = -1;
    if( integratedSelect >= 0 )
        which = integratedSelect;
    if( discreteSelect >= 0 )
        which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
}
```

Asking About the Physical Device’s Features

```c
vkGetPhysicalDeviceProperties( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
```

Vulkan: Identifying the Physical Devices

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

Which Physical Device to Use, II

```c
if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 )
    result = VK_SHOULD_EXIT;
else if( integratedSelect >= 0 )
    physicalDevice = physicalDevices[integratedSelect];
else if( discreteSelect >= 0 )
    physicalDevice = physicalDevices[discreteSelect];
else
    fprintf( FpDebug, "Could not select a Physical Device
    return VK_SHOULD_EXIT;
```

Here’s What the NVIDIA 1080ti Produced

```c
Physical Device Features: geometryShader = 1
sphereIntersectionQuery = 1
```

API version: 4194360
Driver version: 4194360
Vendor ID: 10de
Device ID: 1b06
Device Type: 2 = (Discrete GPU)
API version: 4194360
Driver version: 4194360
Vendor ID: 10de
Device ID: 1b06
Device Type: 2 = (Discrete GPU)
Device Name: GeForce GTX 1080 Ti
Pipeline Cache Size: 13
Device 0 selected (GeForce GTX 1080 Ti)
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

Here's What I Got

- Memory Types:
  Memory 0:
  Memory 1:
  Memory 2:
  Memory 3:
  Memory 4:
  Memory 5:
  Memory 6:
  Memory 7: DeviceLocal
  Memory 8: DeviceLocal
  Memory 9: HostInclusive HostCoherent
  Memory 10: HostInclusive HostCoherent HostCached

- Memory Heaps:
  Heap 0: size = 0x500000 DeviceLocal
  Heap 1: size = 0x7f000000 DeviceLocal

Found 3 Queue Families:
0: queueCount = 16; Graphics Compute Transfer
1: queueCount = 1; Transfer
2: queueCount = 8; Compute
Vulkan: Overall Block Diagram

Application

Instance

Instance

Physical Device

Physical Device

Physical Device

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Command Buffer

Vulkan: Specifying a Logical Device Queue

float queuePriorities[1] =
{
  1.0
};

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

Vulkan: Creating a Logical Device

VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1; // # of device queues
vdci.pQueueCreateInfos = &vdqci; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.pEnabledLayerNames = myDeviceLayers;
vdci.enabledExtensionCount = 0;
vdci.ppEnabledExtensionNames = (const char **)nullptr; // no extensions
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = &PhysicalDeviceFeatures;

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

Vulkan: Creating the Logical Device's Queue

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

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:
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_LUNARG_validation' 'LunarG 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_get_surface_capabilities2'
0x00000001  'VK_KHR_surface'
0x00000001  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_external_hence_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'

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

Will now ask for only 3 instance extensions
VK_KHR_win32_surface
VK_KHR_xcb_i_surface
VK_EXT_debug_report

vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_KHR_display_surface_counter'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_get_surface_capabilities2'
0x00000001  'VK_KHR_surface'
0x00000001  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_external_hence_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'

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

Will now ask for only 3 instance extensions
VK_KHR_win32_surface
VK_KHR_xcb_surface
VK_EXT_debug_report

// 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 );

result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ PhysicalDeviceCount ];
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );
int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
  VkPhysicalDeviceProperties vpdp;
  vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp);
  // need some logical here to decide which physical device to select:
  if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
  {discreteSelect = i;}
  if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
  {integratedSelect = i;}
}
vkGetPhysicalDeviceProperties(PhysicalDevice, OUT &PhysicalDeviceProperties);
vkGetPhysicalDeviceFeatures(IN PhysicalDevice, OUT &PhysicalDeviceFeatures);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp);
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(PhysicalDevice, OUT &vpdmp);
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT vqfp);
delete[] vqfp;

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

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

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

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

Vulkan Highlights: Overall Block Diagram
Semaphores

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

Creating a Semaphore

```cpp
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore semaphore;
    result = vkCreateSemaphore( LogicalDevice, &vsci, PALLOCATOR, &semaphore );
```

Fences

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

Fence Example

```cpp
VkFenceCreateInfo vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT;
    VkFence fence;
    result = vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, &fence );
    // returns right away:
    result = vkGetFenceStatus( LogicalDevice, fence );
    // result = VK_SUCCESS means it has signaled
    // result = VK_NOT_READY means it has not signaled
    // blocks:
    result = vkWaitForFences( LogicalDevice, 1, &fence, VK_TRUE, UINT64_MAX );
    // timeout can be up to UINT64_MAX = 0xffffffffffffffff (= 2**64-1) nanoseconds
    // result = VK_SUCCESS means it returned because a fence signaled
    // result = VK_TIMEOUT means it returned because the timeout was exceeded
```

Semaphores Example during the Render Loop

```cpp
VkSemaphore imageReadySemaphore;
    VkSemaphoreCreateInfo vsci;
        vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
        vsci.pNext = nullptr;
        vsci.flags = 0;
        VkSemaphore imageReadySemaphore;
        result = vkCreateSemaphore( LogicalDevice, &vsci, PALLOCATOR, &imageReadySemaphore );
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, SwapChain, UINT64_MAX, imageReadySemaphore, VK_NULL_HANDLE, &nextImageIndex );
    VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
    VkSubmitInfo vsi;
        vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
        vsi.pNext = nullptr;
        vsi.waitSemaphoreCount = 1;
        vsi.pWaitSemaphores = &imageReadySemaphore;
        vsi.pWaitDstStageMask = &waitAtBottom;
        vsi.commandBufferCount = 1;
        vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
        vsi.signalSemaphoreCount = 0;
        vsi.pSignalSemaphores = (VkSemaphore) nullptr;
    result = vkQueueSubmit( presentQueue, 1, &vsi, renderFence );
```

Fence Example

```cpp
// before VK_FENCE_CREATE_UNSIGNALED_BIT
//
// VkFenceCreateInfo:
//    vkFenceCreateInfo = (VkFenceCreateInfo) { 0 };
//    VkFenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
//    VkFenceCreateInfo.pNext = nullptr;
//    VkFenceCreateInfo.flags = VK_FENCE_CREATE_UNSIGNALED_BIT;
//    VkFence fence;
//    result = vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, &fence );
//    // returns right away:
//    result = vkGetFenceStatus( LogicalDevice, fence );
//    // result = VK_SUCCESS means it has signaled
//    // result = VK_NOT_READY means it has not signaled
//    // blocks:
//    result = vkWaitForFences( LogicalDevice, 1, &fence, VK_TRUE, UINT64_MAX );
//    // timeout can be up to UINT64_MAX = 0xffffffffffffffff (= 2**64-1) nanoseconds
//    // result = VK_SUCCESS means it returned because a fence signaled
//    // result = VK_TIMEOUT means it returned because the timeout was exceeded
```
These are the Commands that could be entered into the Command Buffer, I

- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );`
- `vkCmdDebugMarkerEndEXT( commandBuffer );`
- `vkCmdCopyQueryPoolResults( commandBuffer, flags );`
- `vkCmdCopyImageToBuffer( commandBuffer, pRegions );`
- `vkCmdCopyImage( commandBuffer, pRegions );`
- `vkCmdCopyBufferToImage( commandBuffer, pRegions );`
- `vkCmdCopyBuffer( commandBuffer, pRegions );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );`
- `vkCmdBindIndexBuffer( commandBuffer, indexType );`
- `vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );`
- `vkCmdBeginRenderPass( commandBuffer, const contents );`
- `vkCmdBeginQuery( commandBuffer, flags );`
- `vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );`
- `vkCmdEndRenderPass( commandBuffer );`
- `vkCmdEndQuery( commandBuffer, query );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndexedIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdNextSubpass( commandBuffer, const contents );`
- `vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );`
- `vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );`
- `vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );`
- `vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );`
- `vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );`
- `vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );`
- `vkCmdSetStencilReference( commandBuffer, faceMask, reference );`
- `vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask ) ;`
- `vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );`
- `vkCmdSetEvent( commandBuffer, event, stageMask );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );`
- `vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );`
- `vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );`
- `vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );`
- `vkCmdResetEvent( commandBuffer, event, stageMask );`
- `vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );`
- `vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );`

From the Command Buffer Notes:

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

Events

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

Controlling Events from the Host

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

vkCreateEvent( LogicalDevice, IN &veci, PALLOCATOR, OUT &event );
result = vkCmdWaitEvents( CommandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
result = vkCmdResetEvent( CommandBuffer, IN event, stageMask );
result = vkCmdSetEvent( CommandBuffer, IN event, stageMask );
result = vkGetEventStatus( LogicalDevice, IN event, OUT &status );
result = vkResetEvent( LogicalDevice, IN event, stageMask );

result = VK_EVENT_RESET: not signaled
Note: the host cannot block waiting for an event, but it can test
```

Controlling Events from the Device

```
result = vkCmdSetEvent( CommandBuffer, event, stageMask );
result = vkCmdResetEvent( CommandBuffer, event, stageMask );
result = vkCmdSetDiscardRectangleEXT( CommandBuffer, discardRectangleCount, pDiscardRectangles, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers );

Could be an array of events
Where signaled, where wait for the signal
Memory barriers get executed after events have been signaled
Note: the device cannot test for an event, but it can block
```

Pipeline Barriers

We want all of these commands to be able to run “serial”, but, if we do that, surely there will be race conditions!
Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

Note: there is no problem with Read-then-Read (RRR) as no data has been changed

vkCmdPipelineBarrier() Function Call

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

Pipeline Stage Flags – Where in the Pipeline is this Memory being Accessed?

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

Pipeline Stage Flags – Where in the Pipeline is this Memory being Accessed?

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_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_MEMOBY_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_MEMORY_WRITE_BIT
### Example: Be sure we are done writing an output image before using it for something else

<table>
<thead>
<tr>
<th>Access Operations</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
</tbody>
</table>

### Push Constants

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

By “small”, Vulkan specifies that these must be at least 128 bytes in size, although they can be larger. For example, the maximum size is 256 bytes on the NVIDIA 1080Ti. (You can query this limit by looking at the `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:
```
layout(push_constant) uniform matrix
    mat4 modelMatrix;
```

On the application side, push constants are pushed at the shaders by binding them to the Vulkan Command Buffer:
```
vkCmdPushConstants( CommandBuffer, PipelineLayout, stageFlags,
    offset, size, pValues );
```

where:
- `stageFlags` are or'ed bits of `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_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`.

Prior to that, however, the pipeline layout needs to be told about the Push Constants:
```
result = vkCreatePipelineLayout( LogicalDevice, IN
    &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```

An Robotic Example using Push Constants

A robotic animation (i.e., a hierarchical transformation system)
```
struct arm {       // scale factor in x
    float armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```

Where each arm is represented by:
```
struct arm {       // armScale:  // scale factor in x
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale;
};
```

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

In the Reset Function

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

```
Arm1.armMatrix = glm::mat4();
Arm1.armScale = 6.f;
Arm2.armMatrix = glm::mat4();
Arm2.armScale = 4.f;
Arm3.armMatrix = glm::mat4();
Arm3.armColor = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale = 2.f;
```

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( glm::mat4 );
```

```
VkPipelineLayoutCreateInfo vplci;
```

```
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```
In the **UpdateScene Function**

```cpp
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);
glm::mat4 m21 = glm::mat4();
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));
glm::mat4 m32 = glm::mat4();
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));
Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m1g * m21; // m2g
Arm3.armMatrix = m1g * m21 * m32; // m3g
```

In the **RenderScene Function**

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

In the **Vertex Shader**

```cpp
layout(push_constant) uniform arm {
  mat4 armMatrix;
  vec3 armColor;
  float armScale; // scale factor in x
} RobotArm;
layout(location = 0) in vec3 aVertex;
...;
vec3 bVertex = aVertex; // arm coordinate system [-1., 1.] in X
bVertex.x += 1.; // now is [0., 2.]
bVertex.x /= 2.; // now is [0., 1.]
bVertex.x *= (RobotArm.armScale); // now is [0., RobotArm.armScale]
bVertex = vec3((RobotArm.armMatrix) * vec4(bVertex, 1.));
...;
gl_Position = PVM * vec4(bVertex, 1.); // Projection * Viewing * Modeling matrices
```

**Antialiasing and Multisampling**

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**

![Vulkan Distribution of Sampling Points within a Pixel](image-url)
Consider Two Triangles Whose Edges Pass Through the Same Pixel.

Supersampling

\[
\text{Final Pixel Color} = \sum \text{Color sample from subpixel,} \quad B
\]

# Fragment Shader calls = B

Multisampling

\[
\text{Final Pixel Color} = 1 \times \text{one color sample from A} + 5 \times \text{one color sample from B}
\]

# Fragment Shader calls = 2

### Setting up the Image

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

VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
.
.
vgpci.pMultisampleState = &vpmsci;

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

### Setting up the Image

```c
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;
vad[0].attachment = 0;

VkAttachmentReference depthReference;
vad[1].attachment = 1;
```

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
- 0.5 produces partial supersampling
- 1. produces complete supersampling.
Setting up the Image

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

VkRenderPassCreateInfo vrpci;
vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.attachmentCount = 2; // color and depth/stencil
vrpci.pAttachments = vad;
vrpci.subpassCount = 1;
vrpci.pSubpasses = &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkSubpassDependency *)nullptr;

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

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.

For example:

![Diagram](image)

### Back in Our Single-pass Days

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

![Diagram](image)

### Multipass Rendering

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

![Diagram](image)
**No Noise**

**Noise**

---

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

Original  | Sharpened  | Edge Detected
---|---|---

---

**Multipass, I**

```cpp
textureImage

VkAttachmentDescription
  .format = VK_FORMAT_B8G8R8A8_SRGB;
  .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  .finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
  .flags = 0;
  .attachment = 0;
  .samples = VK_SAMPLE_COUNT_1_BIT;
  .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  .storeOp = VK_ATTACHMENT_STORE_OP_STORE;
  .storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;

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

---

**Multipass, II**

```cpp
vsd[0].pPreserveAttachments = (uint32_t *)nullptr;
vsd[0].preserveAttachmentCount = 0;
vsd[0].pDepthStencilAttachment = colorReference;
vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pColorAttachments = (VkAttachmentReference *)nullptr;
vsd[0].inputAttachmentCount = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[0].flags = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

---

**Multipass, III**

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

---

**Multiplicity, IV**

```cpp
result = vkCreateRenderPass( LogicalDevice, IN &pp, OUT &RenderPass );
```

---

**Placing a Pipeline Barrier so an Image is not used before it is Ready**

```cpp
vimb.subresourceRange = visr;
```

---

8/1/2018
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 versions.

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