Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

Vulkan is better at keeping the GPU busy than OpenGL is. OpenGL drivers need to do a lot of CPU work before handing work off to the GPU. Vulkan lets you get more power from the GPU card you already have. This is especially important if you can hide the complexity of Vulkan from your customer base and just let them see the improved performance. Thus, Vulkan has had a lot of support and interest from game engine developers, 3rd party software vendors, etc.

As an aside, the Vulkan development effort was originally called "glNext", which created the false impression that this was a replacement for OpenGL. It’s not.
Who's Been Specifically Working on Vulkan?

Vulkan from OpenGL

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

The Basic Computer Graphics Pipeline, Shader-style

The Basic Computer Graphics Pipeline, Vulkan-style

Vulkan Shaders

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

Moving part of the driver into the application

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

Simple drivers for hardware efficiency and cross-vendor predictability
Layered architecture so validation and debug layers can be unloaded when not needed
Next time only has to ingest SPIR-V intermediate language
Unified API for mobile, desktop, console and embedded platforms
**Vulkan Highlights: Command Buffers**

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

**Vulkan Highlights: Pipelines**

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

**Vulkan Highlights: Overall Block Diagram**

**Vulkan Highlights: a More Typical Block Diagram**

**Steps in Creating Graphics using Vulkan**

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

Mike Bailey
mjb@cs.oregonstate.edu

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

The Vulkan Sample Code Included with These Notes

Sample Program Output

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

Sample Program Keyboard Inputs

'i', 'I': Toggle using a vertex buffer only vs. an index buffer
'i', 'I': Toggle lighting off and on
'i', 'I': Toggle display mode (textures vs. colors, for now)
'p', 'P': Pause the animation
'q', 'Q': quit the program
'Esc': quit the program
'k', 'K': Toggle rotation-animation and using the mouse

Caveats on the Sample Code, I

1. I've written everything out in appalling longhand.
2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything easily.
3. At times, I could have hidden complexity, but I didn't. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that's kept a secret from you.
4. I've setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
5. At times, I've setup things that didn't need to be setup just to show you what could go there.

Caveats on the Sample Code, II

6. There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'd a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

```c
int main( int argc, char * argv[ ] )
{
    Width  = 800;
    Height = 600;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE );
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
    Reset( );
    InitGraphics( );
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents( );
        Time = glfwGetTime( );          // elapsed time, in double-precision seconds
        UpdateScene( );
        RenderScene( );
    }
    fprintf(FpDebug, "Closing the GLFW window
");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```
```c
void InitGraphics()
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance();
    InitGLFW();
    Init02CreateDebugCallbacks();
    Init03PhysicalDeviceAndGetQueueFamilyProperties();
    Init04LogicalDeviceAndQueue();
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool();
    Init06CommandBuffers();
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init08Swapchain();
    Init09DepthStencilImage();
    Init10RenderPasses();
    Init11Framebuffers();
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool();
    Init13DescriptorSetLayouts();
    Init13DescriptorSets();
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
                                        VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

### A Colored Cube

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

### The Vertex Data is in a Separate File

```c
#include "SampleVertexData.cpp"
```

### What if you don't need all of this information?

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

As best as I can tell, the only penalties for leaving out vertex attributes that you aren’t going to use is memory space and possibly some inefficient uses of the cache, but not gross performance. So, I recommend keeping this struct intact, and, if you don’t need texturing, simply don’t use the texCoord values in your vertex shader.
Vulkan Software Philosophy

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

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

   we would actually say in C:

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

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

Vulkan Conventions

<table>
<thead>
<tr>
<th>vkXxx</th>
<th>is a typedef, probably a struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkXxx()</td>
<td>is a function call</td>
</tr>
<tr>
<td>VK_XXX</td>
<td>is a constant</td>
</tr>
</tbody>
</table>

My Conventions

- “Init” in a function 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 InitXXX( ) functions in numerical order. After that comes the helper functions.
- “Find” in a function name means that something is being looked for.
- “Fill” in a function name means that some data is being supplied to Vulkan.
- “IN” and “OUT” ahead of function argument names are simply to let you know how an argument is going to be used by the function. Otherwise, “IN” and “OUT” have no significance. They are each actually #define’d to nothing.

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

```c
uint32_t  count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, &count, &physicalDevices[0]);
```
Geometry: Where things are (e.g., coordinates)

Topology: How things are connected

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

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

Original Object

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 do all vertex drawing in a right-handed coordinate system and then fix it up in the projection matrix, like this:

ProjectionMatrix*[1 1] = -1.;

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

Vulkan Topologies

VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_PATCH_LIST

Vulkan Topology Issues

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

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

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

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

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

Vulkan Topologies

#include <Vulkan/….>
We will come to the Pipeline state, but for now, know that a Vulkan pipeline is essentially a very large data structure that holds all of the state, including how to process its input.

**C/C++:**
```c
result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer);
```

**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;
```

**Telling the Pipeline about its Input:**
```c
vkGetBufferMemoryRequirements(LogicalDevice, IN pMyBuffer->buffer, OUT &vmr);         // fills vmr
vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory(LogicalDevice, IN &vbci, PALLOCATOR, 0, OUT &vdm);
result = Fill05DataBuffer(MyVertexDataBuffer, (void *) VertexData);
```

**Triangles Represented as an Array of Structures**
```
struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

**Filling the Vertex Buffer**
```
SampleVertexData.cpp
```

**Non-indexed Buffer Drawing**
```
Triangles

```

**A Preview of What Init05DataBuffer Does**
```
```c
void Init05DataBuffer(size_t size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer)
```

**Telling the Pipeline about its Input:**
```
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.

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

```cpp
// In vertex buffer
const uint32_t vertexCount = sizeof(JustVertexData) / sizeof(JustVertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t vertexOffset = 0;

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

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 a point that is common to multiple faces has the same attributes, no matter what face it is in.

Sometimes it doesn't.

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

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

Where values match at the corners (color)

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

Shaders and SPIR-V

Mike Bailey
mjb@cs.oregonstate.edu

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

The Shaders’ View of the Basic Computer Graphics Pipeline

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

Vulkan: GLSL Differences from OpenGL, I

Vulkan Vertex and Instance indices:
• gl_VertexIndex
• gl_InstanceIndex

• Both are 0-based

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

Vulkan: GLSL Differences from OpenGL, II

Shader combinations of separate texture data and samplers:
uniform sampler2D 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;

• Only for scalars, but a vector’s components can be constructed from specialization constants

Specialization Constants for Compute Shaders:
layout( local_size_x_id = 8, local_size_y_id = 16 );

• This sets gl_WorkGroupSize.x and gl_WorkGroupSize.y

Vulkan: Shaders’ use of Layouts for Uniform Variables

All non-sampler uniform variables must be in block buffers

// non-sampler variables must be in a uniform block
layout( block_size = 16, binding = 0 ) uniform mat4 uModel;
Vulkan Shader Compiling

You pre-compile your shaders with an external compiler.

Your shaders get turned into an intermediate form known as SPIR-V, which stands for Standard Portable Intermediate Representation.

SPIR-V gets turned into fully-compiled code at runtime.

SPIR-V spec has been public for a couple of years — new shader languages are surely being developed.

OpenGL and OpenCL have adopted SPIR-V as well.

Advantages:

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

External GLSL Compiler

GLSL Source

GLSL Sampler

SPIR-V

Compiler in driver

Vendor-specific code

Running glslangValidator.exe

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?

#define SPIRV_MAGIC 0x07230203

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, OUT & ShaderModuleVertex );
    fprintf( FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );
    delete [] code;
    return result;
}
Vulkan: Creating a Graphics Pipeline

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...

Filling the Vertex Buffer

```c
MyBuffer MyVertexDataBuffer;

int64_t MyVertexDataBuffer.size(VertexData, &MyVertexDataBuffer);
F803DataBuffer(MyVertexDataBuffer, (void*) VertexData);

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

Data Buffers

What is a Vertex Buffer?

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

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

First, a quick review of computer graphics geometry...
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or'ed bits of: >>
VK_USAGE_TRANSFER_SRC_BIT
VK_USAGE_TRANSFER_DST_BIT
VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
VK_USAGE_UNIFORM_BUFFER_BIT
VK_USAGE_STORAGE_BUFFER_BIT
VK_USAGE_INDEX_BUFFER_BIT
VK_USAGE_VERTEX_BUFFER_BIT
VK_USAGE_INDIRECT_BUFFER_BIT
vbci.sharingMode = << one of: >>
VK_SHARING_MODE_EXCLUSIVE
VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

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

MyBuffer MyMatrixUniformBuffer;

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

something I've found useful

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)
{
    ...    // size = pMyBuffer->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 a C struct to hold some uniform variables

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

Here's the shader code to access those uniform variables

```
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan's LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

Filling those Uniform Variables

```
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan's LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

This C struct is holding the actual data. It is writeable by the application.

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

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

The Descriptor Set for the Buffer

We will come to Descriptor Sets later, but now think of them as the link between the BLOB of uniform variables in GPU memory and the block of variable names in your shader programs.

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.

The C struct is holding the actual data. It is writeable by the application.

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

Here's a quick preview…

```
VkDescriptorBufferInfo vdbi0;
vdbi0.buffer = MyMatrixUniformBuffer.buffer;
```

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

```c
Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
```

Creating and Filling the Data Buffer – the Details

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

```c
VkResult Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to Init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
    // 0 and 0 are offset and flags
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}
```

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

GLFW Keyboard Callback

```c
void GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
    if( action == GLFW_PRESS )
    {
        switch( key )
        {
        //case GLFW_KEY_M:
        //case 'm':
        //case 'M':
          Mode++;
          if( Mode >= 2 )
          Mode = 0;
          break;
        default:
          fprintf( FpDebug, "Unknow key hit: 0x%04x = '%c'
", key, key );
          fflush(FpDebug);
        }
    }
}
```
GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

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

Looping and Closing GLFW

```c
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );          // elapsed time, in double-precision seconds
    Time = glfwGetTime( );
    UpdateScene( );
    RenderScene( );
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
}
```

GLM.pptx

**What is GLM?**

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

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

You can find it at:

http://glm.g-truc.net/0.9.8.5/

You invoke GLM like this:

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

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

**Why are we even talking about this?**

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

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

you would now have to say:

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

Exactly the same concept, but a different expression of it. Read on for details …
The Most Useful GLM Variables, Operations, and Functions

**Constructor:**
- `glm::mat4();`  // identity matrix
- `glm::vec4();`  // vector

**Multiplications:**
- `glm::mat4 * glm::mat4`
- `glm::mat4 * glm::vec4; glm::vec3, 1;`  // promote a vec3 to a vec4 via a constructor

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

**Viewing volume (assign, not concatenate):**
- `glm::mat4 glm::ortho(float left, float right, float bottom, float top, float near, float far);`
- `glm::mat4 glm::ortho(float left, float right, float bottom, float top);`
- `glm::mat4 glm::frustum(float left, float right, float bottom, float top, float near, float far);`
- `glm::mat4 glm::perspective(float fovy, float aspect, float near, float far);`

**Viewing (assign, not concatenate):**
- `glm::mat4 glm::lookAt(glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up);`

---

Installing GLM into your own space

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

Here's what that GLM folder looks like

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.

---

Telling Visual Studio about where the GLM folder is

1. Click on **Project** > **Properties** > **Configuration Properties** > **_Platform_** > **_Target_** > **_Folder Path_**.
2. Enter the path to the GLM folder or select the folder from the drop-down list.
3. Click **OK** to save the changes and close the Properties window.
4. Save your project.
5. Build your project to ensure the changes are applied.

---

Telling Visual Studio about where the GLM folder is

1. Click on **Project** > **Properties** > **Configuration Properties** > **_Platform_** > **_Target_** > **_Folder Path_**.
2. Enter the path to the GLM folder or select the folder from the drop-down list.
3. Click **OK** to save the changes and close the Properties window.
4. Save your project.
5. Build your project to ensure the changes are applied.
GLM in the Vulkan sample.cpp Program

```cpp
if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4();           // identity
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale, Scale, Scale) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3(0.,1.,0.) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3(1.,0.,0.) );
    // done this way, the 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 );
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
```

Why Isn't The Normal Matrix just the Same as the Model Matrix?

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

These diagrams show you why:

Wrong!

Right!

- `glm::mat3 NormalMatrix = glm::mat3(Model);`
- `glm::mat3 NormalMatrix = glm::mat3(Model);`

- `glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(Model));`
- `glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(Model));`

-- Approach #1 --

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

These diagrams show you why.

Original object and normal

Wrong!

Right!

Why Isn't The Normal Matrix just the Same as the Model Matrix?

Instancing

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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.

```cpp
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.);
```

In the vertex shader:

```cpp
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.); add = DELTA * float(NUMINSTANCES); ydelta = DELTA * float(NUMINSTANCES); z(delta = DELTA * float(NUMINSTANCES)); x;
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
```

- `gl_Position = PVM * vertex;`
Put the unique characteristics in a uniform buffer and reference them

Still uses `gl_InstanceIndex` in the vertex shader:

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

In the vertex shader:

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

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

In OpenGL, puts all uniform data in the same "set", but with different binding numbers, so you can get at each one.

Each uniform variable gets updated one-at-a-time. Wouldn't it be nice if we could update a collection of related uniform variables all at once, without having to update the uniform variables that are not related to this collection?

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

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

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

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

1. Define the Descriptor Set Layouts
2. Define the Descriptor Set Pools
3. Draw your objects

1. Define the Descriptor Set Layouts

- Descriptor Set Layouts are the shader uniforms that allow the programmer to link C++ objects to the GPU. The uniform block will always be the same size. Choose what you want to link to the GPU and how you want to store the data: float, int, vec4, mat4, sampler2D, sampler3D.

2. Define the Descriptor Set Pools

- If you’re not using a “pool” of Descriptor Sets, you’ll write the Descriptor Set each time you want to change it. If you do this, you’ll be ultra-slow. Instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

3. Draw your objects

- Draw your objects.
Step 2: Define the Descriptor Set Layouts

- **Matrix Set DS Layout Binding:**
  - Pipeline stage(s): VK_SHADER_STAGE_VERTEX_BIT
  - DescriptorCount: 1
  - DescriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - Binding: 0

- **TexSampler Set DS Layout Binding:**
  - Pipeline stage(s): VK_SHADER_STAGE_FRAGMENT_BIT
  - DescriptorCount: 1
  - DescriptorType: VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
  - Binding: 0

- **Light Set DS Layout Binding:**
  - Pipeline stage(s): VK_SHADER_STAGE_FRAGMENT_BIT
  - DescriptorCount: 1
  - DescriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - Binding: 0

- **Misc Set DS Layout Binding:**
  - Pipeline stage(s): VK_SHADER_STAGE_VERTEX_BIT
  - DescriptorCount: 1
  - DescriptorType: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - Binding: 0

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

1. **vplci** = VkPipelineLayoutCreateInfo
2. Set up the pipeline layout creation information.
3. Include the descriptor set layouts.

Step 4: Allocating the Memory for Descriptor Sets

1. **vdsai** = VkDescriptorSetAllocateInfo
2. Allocate the descriptor sets.

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

1. Set the descriptor buffer information.
2. Set the descriptor image view information.
3. Set the descriptor sampler information.

In summary, the process involves defining the descriptor set layouts, including them in the pipeline layout, allocating memory for descriptor sets, and specifying where their data resides in CPU memory.
Step 5: Tell the Descriptor Sets where their CPU Data is

```c
// ds 0:
vwds0.pBufferInfo = IN
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.descriptorCount = 1;
vwds0.dstArrayElement = 0;
vwds0.dstBinding = 0;
vwds0.dstSet = DescriptorSets[0];
vwds0.pNext = nullptr;
vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;

// ds 1:
vwds1.pBufferInfo = IN
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.descriptorCount = 1;
vwds1.dstArrayElement = 0;
vwds1.dstBinding = 0;
vwds1.dstSet = DescriptorSets[1];
vwds1.pNext = nullptr;
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
```

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

```c
// this could have been done with one call and an array of VkWriteDescriptorSets:
uint32_t copyCount = 0;
VkWriteDescriptorSet vwds0 = IN
vwds0.pTexelBufferView = (VkBufferView *)nullptr;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pBufferInfo = IN
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.descriptorCount = 1;
vwds0.dstArrayElement = 0;
vwds0.dstBinding = 0;
vwds0.dstSet = DescriptorSets[0];
vwds0.pNext = nullptr;
vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
```

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

```
So, the Pipeline Layout contains the structure of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.
```

---

**Textures**

Mike Bailey

mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Memory Types

- CPU Memory
- GPU Memory
- Host Visible
- Device Local
- GPU Memory

Texture Sampling Parameters

- OpenGL
- Vulkan

Texture Mip-mapping

- NVIDIA Discrete Graphics:
  - Memory Types:
    - Memory 1: Device Local
    - Memory 2: Host Visible
    - Memory 3: Host Coherent
    - Memory 4: Host Cached
    - Memory 5: Device Local
    - Memory 6: Host Visible
    - Memory 7: Host Coherent
    - Memory 8: Host Cached
    - Memory 9: Host Visible
    - Memory 10: Host Coherent
    - Memory 11: Host Cached

- Intel Integrated Graphics:
  - Memory Types:
    - Memory 1: Device Local
    - Memory 2: Host Visible
    - Memory 3: Host Coherent
    - Memory 4: Host Cached

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

"Many things in a small place."
#ifdef CHOICES
// *******************************************************************************
// this second {...} is to create the actual texture image:
// *******************************************************************************

result = VkDeviceMemory vdm;
result = vkGetImageMemoryRequirements
VkMemoryRequirements vmr;

vkGetImageSubresourceLayout
VkSubresourceLayout vsl;

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

VkMemoryAllocateInfo vmai;

vkBindImageMemory

// because we are transferring into it and will eventual sample from it
ve3.depth = 1;
vo3.z = 0;
vo3.y = 0;
visl.mipLevel = 0;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

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

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

// because we want to mmap it
FindMemoryThatIsHostVisible()
( LogicalDevice, stagingImage, IN &vis, OUT &vsl);

( LogicalDevice, stagingImage, OUT &vmr);

( LogicalDevice, IN textureImage, OUT &vmr);

FindMemoryThatIsHostVisible()  // because we want to sample from it
vkGetImageSubresourceLayout
VkSubresourceRange visr;

vis.arrayLayer = 0;
vis.mipLevel = 0;
vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

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

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

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

vkBindImageMemory

// transition the texture buffer layout:
// *******************************************************************************

VkCommandBufferBeginInfo vcbbi;

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

vkCmdCopyImage(TextureCommandBuffer
VkImageCopy vic;

for (unsigned int y = 0; y < texHeight; y++)
unsigned char *gpuBytes = (unsigned char *)gpuMemory;
memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

vkCmdPipelineBarrier
vimb.srcAccessMask = 0;
vimb.image = textureImage;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.pNext = nullptr;

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

vkCmdCopyImage(TextureCommandBuffer

vkBeginCommandBuffer
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
vcbbi.pNext = nullptr;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

vkCmdPipelineBarrier
vimb.subresourceRange = visr;
vimb.dstAccessMask = 0;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.pNext = nullptr;

vkCmdCopyImage(TextureCommandBuffer

vkCmdPipelineBarrier
vimb.srcAccessMask = 0;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.srcAccessMask = 0;
vimb.pNext = nullptr;
What is the Vulkan 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”. It is a data structure.
2. The Vulkan Graphics Pipeline is not the processes that OpenGL would call the graphics pipeline.
3. For the most part, the Vulkan Graphics Pipeline is read-only—that is, once its combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.
4. The shaders get compiled the last of the way when their Graphics Pipeline gets created.

The First Step: Create the Graphics Pipeline Layout

The Graphics Pipeline Data Structure

Mike Bailey
mjb@cs.oregonstate.edu

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

Reading in a Texture from a BMP File

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

```
result = Init06TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyTexturePuppy);
```

```
typedef struct MyTexture
{
    uint32_t width;
    uint32_t height;
    VkImage texImage;
    VkSampler texSampler;
    VkFormat texFormat;
    VkDeviceMemory vdm;
} MyTexture;
```

```
// *******************************************************************************
// transition the texture buffer layout a second time:
// *******************************************************************************

result = vkCmdPipelineBarrier(TextureCommandBuffer,
                              VK_PIPELINE_STAGE_TRANSFER_BIT,
                              VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
                              0,
                              (VkBufferMemoryBarrier *)nullptr,
                              0,
                              (VkMemoryBarrier *)nullptr,
                              0,
                              (VkPipelineStageFlags *)nullptr);

vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;

vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;

vivci.format = VK_FORMAT_R8G8B8A8_UNORM;

vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;

vivci.image = textureImage;
```

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

```
result = Init14GraphicsPipelineLayout( );
```

```
result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipeline.pPipelineLayout);
```

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

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

There is also a Vulkan Compute Pipeline—we will get to that later.
Vulkan: A Pipeline Records the Following Items:

- Pipeline layout: DescriptorSet, PushConstants
- Which Shaders are going to be used
- Per-vertex input bindings: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissoring: x, y, w, h
- Rasterization: polygonMode, lineWidth, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilFront, stencilBack
- Blending: blendEnable, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the pipeline)

Solid black indicates that this state item can also be set with Dynamic Variables

Creating a Typical Graphics Pipeline

Link in the Shaders

Link in the Per-Vertex Attributes

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

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

These are offsets to the top of the sample code.
What is “Primitive Restart Enable”? 

`vpiasci.primitiveRestartEnable = VK_FALSE;`

“Restart Enable” is used with:  
• Indexed drawing.  
• Triangle Fan and “Strip topologies

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

- `VK_INDEX_TYPE_UINT16 = 0` (0 – 65,535)  
- `VK_INDEX_TYPE_UINT32 = 1` (0 – 4,294,967,295)

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

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

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

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

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

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

Setting the Rasterizer State

`vprsci.depthClampEnable = VK_FALSE;`

What is “Depth Clamp Enable”? 

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

A good use for this is Polygon Capping.

The front of the polygon is clipped, revealing to the viewer that this is really a shell rather than a solid. The gray area shows what would happen if depthClampEnable were true (it would have been red).
**What is "Depth Bias Enable"?**

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

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

---

**Color Blending State for each Color Attachment**

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

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**VkPipelineColorBlendStateCreateInfo**

```
vpcbsci.pAttachments = &vpcbas;
vpcbsci.logicOp = VK_LOGIC_OP_COPY;
vpcbsci.logicOpEnable = VK_FALSE;
vpcbsci.flags = 0;
vpcbsci.pNext = nullptr;
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
```

**Which Pipeline Variables can be Set Dynamically**

```
vkdState = VK_DYNAMIC_STATE_VIEWPORT;
vkdDynamicState = VK_DYNAMIC_STATE_STENCIL_REFERENCE;
```

**VkPipelineDynamicStateCreateInfo**

```
vpdsci.pDynamicStates = vds;
vpdsci.dynamicStateCount = 0;                   // leave turned off for now
vpdsci.flags = 0;
vpdsci.pNext = nullptr;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
```

**Magic Lenses**

---

**Stencil Operations for Front and Back Faces**

`VDS`:

<table>
<thead>
<tr>
<th><code>VDS</code></th>
<th><code>VDS</code></th>
<th><code>VDS</code></th>
<th><code>VDS</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
</tr>
<tr>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
</tr>
<tr>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
</tr>
<tr>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
<td>VK_DONT_CARE</td>
</tr>
</tbody>
</table>

**Uses for Stencil Operations**

Polygon edges without Z-fighting
vkPipelineDepthStencilStateCreateInfo vpdssci =
{
    .sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .depthTestEnable = VK_TRUE,
    .depthWriteEnable = VK_TRUE,
    .depthCompareOp = VK_COMPARE_OP_LESS,
    .depthBoundsTestEnable = VK_FALSE,
    .front = vsosf,
    .back = vsosb,
    .minDepthBounds = 0.,
    .maxDepthBounds = 1.,
    .stencilTestEnable = VK_FALSE,
};

vkGraphicsPipelineCreateInfo vgpci =
{
    .sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .stageCount = 2,
    .pStages = vpssci,
    .pVertexInputState = &vpvisci,
    .pInputAssemblyState = &vpiasci,
    .pTessellationState = nullptr,
    .pViewportState = &vpvsci,
    .pRasterizationState = &vprsci,
    .pMultisampleState = &vpmsci,
    .pDepthStencilState = &vpdssci,
    .pColorBlendState = &vpcbsci,
    .pDynamicState = &vpdsci,
    .layout = IN GraphicsPipelineLayout,
    .renderPass = IN RenderPass,
    .subpass = 0,
    .basePipelineHandle = VK_NULL_HANDLE,
    .basePipelineIndex = 0,
};

result = vkCreateGraphicsPipelines(logicalDevice, VK_NULL_HANDLE, 1, &vgpci, PALLOCATOR, OUT pGraphicsPipeline);
return result;
for(unsigned int i = 0; i < count; i++)
{
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
    vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr);
    uint32_t count = -1;
    return -1;
    int

    if((vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
        fprintf(FpDebug, " Transfer");
    if((vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
        fprintf(FpDebug, " Compute");
    if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
        fprintf(FpDebug, " Graphics");
    fprintf(FpDebug, "	%d: Queue Family Count = %2d ; ", i, vqfp[i].queueCount);

    uint32_t queueIndex = 0;
    uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateLogicalDevice(IN PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice);

    VkDeviceCreateInfo vdci;
    float queuePriorities[] = {1.0f};
    Creating a Logical Device Needs to Know Queue Family Information
    VkDeviceQueueCreateInfo vdqci[1];
    Init06CommandBuffers();
    VkResult result = vkCreateLogicalDevice(IN PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice);
    VkResult result = vkCreateCommandPool(IN PhysicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
    VkSemaphoreCreateInfo vsci;
    VkCommandBufferAllocateInfo vcbai;
    VkCommandBufferBeginInfo vcbbi;
    vkEndCommandBuffer(IN CommandBuffers[0]);
    vkEndCommandBuffer(IN CommandBuffers[1]);
    VkResult result = vkCreateSemaphore(IN LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    vkCreateSemaphore(IN LogicalDevice, IN &vsci, PALLOCATOR, OUT &nextImageIndex);
    vkQueueSubmit(IN LogicalDevice, 1, IN CommandBuffers, IN &flushSemaphore);
    VkSemaphore imageReadySemaphore = 0;
    VkSemaphore nextImageIndex = 0;
    CommandBuffers[0], IN &flushSemaphore,

    Finding what Queue Families are Available
    Creating the Command Pool as part of the Logical Device
    Creating the Command Buffers
    Beginning a Command Buffer
These are the Commands that could be entered into the Command Buffer, I

敬

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

敬

The Entire Submission | Wait / Display Process

敬
What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

"Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g., sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences."

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there.

But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.

Swap Chain

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan thinks of it this way

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 really 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.

```c
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "\n\nvkGetPhysicalDeviceSurfaceCapabilitiesKHR:\n" );
...
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
fprintf( FpDebug, "\nFound %d Surface Formats:\n", formatCount )
...
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "\nFound %d Present Modes:\n", presentModeCount );
...
VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr;
vscci.flags = 0;
vscci.surface = Surface;
vscci.minImageCount = 2; // double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.imageArrayLayers = 1;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.queueFamilyIndexCount = 0;
vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
vscci.oldSwapchain = VK_NULL_HANDLE;
vscci.clipped = VK_TRUE;
result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
```
Creating the Swap Chain Images and Image Views

uint32_t imageCount; // # of display buffers – 2? 3?

result = vkGetSwapchainImagesKHR(LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr);

PresentImages = new VkImage[imageCount];

result = vkGetSwapchainImagesKHR(LogicalDevice, SwapChain, OUT &imageCount, PresentImages);

// present views for the double-buffering:
PresentImageViews = new VkImageView[imageCount];

for (unsigned int i = 0; i < imageCount; i++) {
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vivci.subresourceRange.baseMipLevel = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;
    vivci.image = PresentImages[i];
    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
}

Rendering into the Swap Chain, I

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

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

uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;

vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

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

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

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);

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

result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);
### Asking About the Physical Device's Features

```c
// PhysicalDeviceProperties
vkGetPhysicalDeviceProperties(PhysicalDevice, PhysicalDeviceProperties); // GetPhysicalDeviceProperties

fprintf(stderr, "PhysicalDeviceProperties:
"); // Print device properties
```

Here's What the NVIDIA RTX 2080 Ti Produced

```c
// PhysicalDeviceFeatures
physicalDeviceFeatures.shaderInt16 = 1; // Shader Int16
physicalDeviceFeatures.pipelineStatisticsQuery = 1; // Pipeline Statistics Query
```

### Here's What the NVIDIA RTX 2080 Ti Produced

```c
Vulkan: Identifying the Physical Devices

uint32_t PhysicalDeviceCount = 0; // PhysicalDeviceCount
VkPhysicalDevice * physicalDevices; // PhysicalDevices

result = vkEnumeratePhysicalDevices(Instance, &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr); // Enumerate Physical Devices
if (result != VK_SUCCESS) // Check for success
    fprintf(stderr, "Could not enumerate the %d physical devices
", PhysicalDeviceCount);

PhysicalDeviceCount = 0;
```

### Querying the Number of Physical Devices

```c
uint32_t PhysicalDeviceCount = 0; // PhysicalDeviceCount
VkPhysicalDevice * physicalDevices; // PhysicalDevices

result = vkEnumeratePhysicalDevices(Instance, &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr); // Enumerate Physical Devices
if (result != VK_SUCCESS) // Check for success
    fprintf(stderr, "Could not enumerate the %d physical devices
", PhysicalDeviceCount);

PhysicalDeviceCount = 0;
```

### Which Physical Device to Use, I

```c
// Querying the Number of Physical Devices
result = vkEnumeratePhysicalDevices(Instance, &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr); // Enumerate Physical Devices
if (result != VK_SUCCESS) // Check for success
    fprintf(stderr, "Could not enumerate the %d physical devices
", PhysicalDeviceCount);

PhysicalDeviceCount = 0;
```

### Which Physical Device to Use, II

```c
// Which Physical Device to Use, II
result = vkEnumeratePhysicalDevices(Instance, &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr); // Enumerate Physical Devices
if (result != VK_SUCCESS) // Check for success
    fprintf(stderr, "Could not enumerate the %d physical devices
", PhysicalDeviceCount);

PhysicalDeviceCount = 0;
```

### Asking About the Physical Device's Features

```c
// PhysicalDeviceFeatures
vkGetPhysicalDeviceProperties(PhysicalDevice, PhysicalDeviceFeatures); // GetPhysicalDeviceProperties

fprintf(stderr, "PhysicalDeviceFeatures:
"); // Print device features
```
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 the Intel HD Graphics 520 Produced

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

Asking About the Physical Device's Different Memories

Here's What I Got

15 Memory Types:
- Memory 0:
- Memory 1:
- Memory 2:
- Memory 3:
- Memory 4:
- Memory 5:
- Memory 6:
- Memory 7: DeviceLocal
- Memory 8: DeviceLocal
- Memory 9: HostVisible HostCoherent
- Memory 10: HostVisible HostCoherent HostCached
- Memory 11:
- Memory 12:
- Memory 13:
- Memory 14:
- Memory 15:

2 Memory Heaps:
- Heap 0: size = 0xb7c00000 DeviceLocal
- Heap 1: size = 0xfac00000

Asking About the Physical Device's Queue Families

Here's What I Got

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

Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
const char * myDeviceLayers[] = {
    // "VK_LAYER_LUNARG_api_dump",
    // "VK_LAYER_LUNARG_core_validation",
    // "VK_LAYER_LUNARG_image",
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    // "VK_LAYER_NV_optimus"
};

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

// see what device layers are available:
uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);

Looking to See What Device Layers are Available

4 physical device layers enumerated:
0x00401063   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':

Looking to See What Device Extensions are Available

What Device Layers and Extensions are Available

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

Vulkan: Creating a Logical Device

float   queuePriorities[1] = {
    1.0f};

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

Vulkan: Creating the Logical Device's Queue
Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

Which Pipeline State Variables can be Changed Dynamically

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

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

Creating a Pipeline

```cpp
VkDynamicState vds[] = {
  VK_DYNAMIC_STATE_VIEWPORT,
  VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState);
vpdsci.pDynamicStates = &vds;

VkGraphicsPipelineCreateInfo vgpci;

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

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

Push Constants

The command buffer-bound function calls to set these dynamic states are:

- `vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports );`
- `vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors );`
- `vkCmdSetLineWidth(commandBuffer, linewidth );`
- `vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );`
- `vkCmdSetBlendConstants(commandBuffer, blendConstants[4] );`
- `vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds );`
- `vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask );`
- `vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask );`
- `vkCmdSetStencilReference(commandBuffer, faceMask, reference );`

Filling the Dynamic State Variables in the Command Buffer

The command buffer-bound function calls to set these dynamic states are:
Push Constants

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

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

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
\begin{bmatrix}
\end{bmatrix};
```

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

### Setting up the Push Constants for the Pipeline Structure

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

```cpp
VkPushConstantRange
{
    stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT;
    offset = 0;
    size = sizeof(glm::mat4);
}
```

### Creating a Pipeline

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

Where each arm is represented by:

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

### In the Reset Function

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

```c
vkPushConstantRange vpcr[1];
vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vpcr[0].offset = 0;
vpcr[0].size = sizeof(struct arm);
VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = &vpcr[0];
result = vkCreatePipelineLayout(LogicalDevice, &vplci, PALLOCATOR, &GraphicsPipelineLayout);
```

In the UpdateScene Function

```c
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

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

In the Vertex Shader

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

Getting Information Back from the Graphics System

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan

Vulkan

• There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp
• Vulkan requires you to first setup “Query Pools”, one for each specific type
• This indicates that Vulkan thinks that Queries are time-consuming (relatively) to setup, and thus better to set them up in program-setup than in program-runtime
Don't draw the colors – just draw the depths (especially if the fragment shader is time-consuming).

Don't draw the whole bounding volume – cull away the back faces (two reasons: time and correctness).

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

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

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

Pipeline Statistics Query

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

Resetting, Filling, and Examining a Query Pool

The vkCmdResetQueryPool() function is called and when the first thing reaches the specified pipeline stage.

Timestamp Query

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

Even though the stages are "bits", you are supposed to only specify one of them, not "or" multiple ones together.
Start by Creating the Data Buffers

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

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

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

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

Creating a Shader Storage Buffer

VkBuffer Buffer;

VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = << buffer size in bytes >>;
    vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

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

Fill the Data Buffer

VkResult FillDataBuffer ( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory);
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}
The Particle System Compute Shader -- Setup

```c
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 )  buffer  Pos;
layout( std140, set = 0, binding = 1 )  buffer  Vel;
layout( std140, set = 0, binding = 2 )  buffer  Col;

int width = ...;
int height = ...

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
vecc4 Colours[ width*height ]; // array of structures
vecc4 Velocities[ width*height ]; // array of structures
vecc4 Positions[ width*height ]; // array of structures
vec4 dt = vec4(1.0/width, 1.0/height, 0, 0); // this is the size of one pixel
```

This is the number of work-items per work-group, set in the compute shader. The number of work-groups is set in the `vkCmdDispatch()` function call in the C/C++ program.
Dispatching the Compute Shader from the Command Buffer

```cpp
costIn NUM_PARTICLES = 1024*1024;
costIn NUM_WORK_ITEMS = 64;
costIn NUM_X_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;
...
vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);
vkCmdDispatch(CommandBuffer, NUM_X_WORK_GROUPS, 1, 1);
```

This is the number of work-groups, set in the C/C++ program. The number of work-items per work-group is set in a layout in the compute shader.

Dr.

```cpp
vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);
vkCmdDispatchIndirect(CommandBuffer, Buffer, 0); // Buffer holds the 3 sizes, offset=0
```

Or,

This is the number of work-groups, set in the C/C++ program. The number of work-items per work-group is set in a layout in the compute shader.

The Bouncing Particle System Compute Shader – What Does It Look Like?

Specialization Constants

Mike Bailey
mjb@cs.oregonstate.edu

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

Why Do We Need Specialization Constants?

Specialization Constants could be used for:
- Setting the work-items per work-group in a compute shader
- Setting a Boolean flag and then eliminating the if-test that used it
- Setting an integer constant and then eliminating the switch-statement that looked for it
- Making a decision to unroll a for-loop because the number of passes through it are small enough
- Collapsing arithmetic expressions into a single value
- Collapsing trivial simplifications, such as adding by zero or multiplying by 1

In Vulkan, all shaders get halfway-compiled by SPIR-V and then the real-of-the-way compiled by the Vulkan driver.

Normally, the halfway compile fixes all constant values and compiles the code that uses them.

But it would be nice every so often to have your Vulkan program sneek into the halfway-compiled binary and manipulate some constants at runtime. This is what Specialization Constants are for.

A Specialization Constant is a way of injecting an integer, boolean, uint, float, or double constant into a halfway-compiled version of a shader right before the real-of-the-way compilation.

That final compilation happens when you call

```cpp
vkCreateComputePipelines()
```

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

Specialization Constant Example -- Setting an Array Size

In the compute shader

```cpp
layout( constant_id = 7 ) const int ASIZE = 32;
int array[ASIZE];
```

In the Vulkan C/C++ program

```cpp
int asize = 32;
VkSpecializationMapEntry vsme[1];
vkSpecializationMapEntry_t
vsme[0].constantID = 7;
vsme[0].offset = 0; // # bytes into the Specialization Constant
vsme[0].size = sizeof(asize); // size of just the Specialization Constant

VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(asize); // size of all the Specialization Constants together
vsi.pData = &asize; // array of all the Specialization Constants
```
Linking the Specialization Constants into the Compute Pipeline

VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 7;
vsme[0].offset = 0;
vsme[0].size = sizeof(asize);

VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = vsme[0];
vsi.dataSize = sizeof(asize);
vsi.pData = &asize;

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

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

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

Specialization Constant Example – Setting Multiple Constants

In the compute shader
layout(constant_id = 9) const int a = 1;
layout(constant_id = 10) const int b = 2;
layout(constant_id = 11) const float c = 3.14;

In the C/C++ program:
struct abc { int a, int b, float c; }
abc;

VkSpecializationMapEntry vsme[3];
vsme[0].constantID = 9;
vsme[0].offset = offsetof(abc, a);
vsme[0].size = sizeof(abc.a);
vsme[1].constantID = 10;
vsme[1].offset = offsetof(abc, b);
vsme[1].size = sizeof(abc.b);
vsme[2].constantID = 11;
vsme[2].offset = offsetof(abc, c);
vsme[2].size = sizeof(abc.c);

VkSpecializationInfo vsi;
vsi.mapEntryCount = 3;
vsi.pMapEntries = vsme[0];
vsi.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsi.pData = &abc; // array of all the Specialization Constants

Synchronization

Mike Bailey
mjb@cs.oregonstate.edu

Where Synchronization Fits in the Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Semaphore

Ask for Something

Try to Use the Something

Semaphore

• 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, IN &vsci, PALLOCATOR, OUT &semaphore);
```

Semaphores Example during the Render Loop

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

uint32_t nextImageIndex;
vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);
```

### 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

```cpp
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

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

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

Events

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

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

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

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

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

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

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

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

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

Example: Don't read a buffer back to the host until a shader is done writing it
VkImageLayout – How an Image gets Laid Out in Memory
depends on how it will be Used

Read into a shader as a texture
Show image to viewer
Used as a color attachment

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

* Aliasing
* The Display We Want
* Too often, the Display We Get

Aliasing

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

What the signal really is: what we want
Sampling Interval
Sampled Points

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

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 sub-pixels within a pixel, render the image at each of these sub-pixels (including depth and stencil tests), then average them together.

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

Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

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

Multisampling

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

Setting up the Image

```cpp
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);
```

How dense is the sampling

VK_TRUE means to allow some sort of multisampling to take place
Setting up the Image

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

- 0: produce simple multisampling
- (0, 1): produces partial supersampling
- 1: produces complete supersampling

```
VkPipelineMultisampleStateCreateInfo vpmsci;
vpmsci.minSampleShading = 0.5;
```

```
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;
```

```
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, vrpci, PALLOCATOR, &RenderPass);
```

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

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

```
VkImageSubresourceLayers vsl;
vsl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
vsl.mipLevel = 0;
vsl.baseArrayLayer = 0;
vsl.layerCount = 1;
```

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

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

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

Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Computer Graphics API

Thanks for coming today!

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