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 is the Khronos Group?

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

Who’s Been Specifically Working on Vulkan?

Playing “Where’s Waldo” with Khronos Membership

Vulkan Differences from OpenGL

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

1. Model Transform
2. View Transform
3. Projection Transform
4. Rasterization
5. Per-vertex in variables
6. Uniform Variables
7. MC = Model Vertex Coordinates
8. WC = World Vertex Coordinates
9. EC = Eye Vertex Coordinates
10. Per-vertex out variables

Vertex Shader

- gl_Vertex, gl_Normal, gl_Color
- g_ModelViewMatrix, g_ProjectionMatrix, g_ModelViewProjectionMatrix

Fragment Shader

- Framebuffer
- gl_FragColor
- Fragment Processing, Texturing, Per-fragment Lighting

The Basic Computer Graphics Pipeline, Vulkan-style

1. Model Transform
2. View Transform
3. Projection Transform
4. Rasterization
5. Per-vertex in variables
6. Uniform Variables
7. g_Position, Per-vertex out variables

Vertex Shader

- gl_Position, gl_Normal, gl_Color
- gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix

Fragment Shader

- Framebuffer
- Output color(s)
- Uniform Variables

Vulkan Shaders

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

- Application
  - Traditional graphics drivers include significant context, memory and error management
  - Direct GPU Control

- Application
  - Responsible for memory allocation and thread management to generate command buffers
  - GPU

- Khronos Group

- OpenGL
  - Khronos Group

- Vulkan
  - Khronos Group

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

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

Vulkan Highlights: Pipelines

- In OpenGL, your "pipeline state" is whatever your current graphics attributes are: color, transformations, textures, shaders, etc.
- Changing the state on-the-fly one item at-a-time is very expensive
- Vulkan forces you to set all your state variables 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: 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 RenderPasses
14. Create the Framebuffer(s)
15. Create the Descriptor Set Pool
16. Create the Command Buffer Pool
17. Create the Command Buffer(s)
18. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render- ...

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Sample Program Keyboard Inputs

<table>
<thead>
<tr>
<th>Keys</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y, Y:</td>
<td>Toggle using a vertex buffer only vs. an index buffer</td>
</tr>
<tr>
<td>Y, L:</td>
<td>Toggle lighting off and on</td>
</tr>
<tr>
<td>m, M:</td>
<td>Toggle display mode (textures vs. colors, for now)</td>
</tr>
<tr>
<td>p, P:</td>
<td>Pause the animation</td>
</tr>
<tr>
<td>q, Q:</td>
<td>quit the program</td>
</tr>
<tr>
<td>Esc:</td>
<td>quit the program</td>
</tr>
<tr>
<td>Y, R:</td>
<td>Toggle rotation-animation and using the mouse</td>
</tr>
</tbody>
</table>

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’ed 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

```cpp
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;
}
```
void InitGraphics( )
{
    HERE_I_AmA "InitGraphics";
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallback( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    InitLogicalDeviceAndQueue( );
    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( );
    Init12Shader( "sample-vert.spv", &ShaderModuleVertex );
    Init12Shader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorsetLayouts();
    Init13DescriptorSets( );
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
        VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}

static GLuint CubeTriangleIndices[3] =
{ 
    { 0, 2, 3 },{ 0, 3, 1 },
    { 4, 7, 6 },{ 1, 7, 5 },{ 0, 4, 6 },
    { 0, 6, 2 },
    { 2, 6, 7 },{ 2, 7, 3 },{ 0, 1, 5 }{ 0, 5, 4 };

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

struct vertex VertexData[] =
{ 
    // triangle 0-2-3:
    // vertex #0:
    { -1., -1., -1. },{  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. } },
    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. }
    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
    // vertex #4:
    {  1., -1., -1. },
    {  0.,  0., -1. },
    {  1., -1.,  0. },
    {  0.,-1. }
    // vertex #5:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    { -1., -1.,  0. },
    {  0.,-1. }
    // vertex #6:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    { -1.,  1.,  0. },
    {  0., 1. }
    // vertex #7:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
};

struct CubeVertexIndices[3] =
{ 
    { 0, 4, 3 },
    { 0, 3, 2 },
    { 0, 2, 1 },
    { 1, 2, 3 },
    { 1, 3, 7 },
    { 1, 7, 6 },
    { 7, 6, 2 },
    { 7, 2, 1 };

struct CubeVertexCoordinates[8] =
{ 
    { 0.2, 0.2, 0.2 },
    { 0.5, 0.5, 0.5 },
    { 0.8, 0.8, 0.8 },
    { 0.1, 0.1, 0.1 },
    { 0.2, 0.2, 0.2 },
    { 0.5, 0.5, 0.5 },
    { 0.8, 0.8, 0.8 },
    { 0.1, 0.1, 0.1 };
};

A Colored Cube

A Colored Cube
The Vertex Data is in a Separate File

```
#include "SampleVertexData.cpp"

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

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

The Vertex Data is in a Separate File

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 in 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 these 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

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = <buffer size in bytes>;
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;
VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &Buffer );
```

```c
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );      // fills vmr
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = 0;
result = vkAllocateMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```
Vulkan Conventions

VkXxx is a typedef, probably a struct
vkXxx() is a function call
VK_XXX is a constant

My Conventions

"Init" in a function call name means that something is being setup that only needs to be setup once
The number after "Init" gives you the ordering
In the source code, after main() comes InitGraphics(), then all of the InitxxYYY() functions in numerical order. After that comes the helper functions
"Find" in a function call name means that something is being looked for
"Fill" in a function call name means that some data is being supplied to Vulkan
"IN" and "OUT" ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are each actually #define'd to nothing.

Your Sample2019.zip File Contains This

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

Extras in the Code

Double-click here to launch Visual Studio 2019 with this solution

The "19" refers to the version of Visual Studio, not the year of development.
Mike Bailey  
mjb@cs.oregonstate.edu

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

Geometry vs. Topology

Where things are (e.g., coordinates)

Topology:
How things are connected

Original Object

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

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

Vulkan Topologies

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

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

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

$$\text{ProjectionMatrix}[:-1,2:] = -1.$$  
This is like saying "Y' = -Y".

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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

In a right-handed coordinate system, 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**.

---

**Triangles Represented as an Array of Structures**

From the file `SampleVertexData.cpp`:

```cpp
def vertex:
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;

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

Modeled in right-handed coordinates.
Filling the Vertex Buffer

MyBuffer MyVertexBuffer;
InitMyVertexBuffer( sizeof(VertexData), &MyVertexBuffer );
FillDataBuffer( MyVertexBuffer, (void *) VertexData );

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

A Preview of What Init05DataBuffer Does

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 );
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );
    return result;
}

Telling the Pipeline about its Input

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

C/C++:

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

GLSL Shader:

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

Vulkan

Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    MyVertexBufferCreateInfo( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, &MyVertexBuffer );
    FillDataBuffer( MyVertexBuffer, (void *) VertexData );
}

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;

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

GLSL Shader:

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 Command Buffer what Vertices to Draw

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

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

Drawing with an Indexed Buffer

```cpp
vkCmdBindVertexBuffer(commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets);
vkCmdBindIndexBuffer(commandBuffer, indexDataBuffer, indexOffset, indexType);
typedef enum VkIndexType
{
    VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
    VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
} VkIndexType;
```

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

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

Drawing with an Indexed Buffer

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

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

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

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

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

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

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

- Specialization Constants:
  ```
  layout( constant_id = 3 ) const int N = 5;
  ```

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

Vulkan: Shaders’ use of Layouts for Uniform Variables

- All non-sampler uniform variables must be in block buffers
  ```
  layout( local_size_x_id = 8, local_size_y_id = 16 ) uniform matBuf{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
  } Matrices;
  ```

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

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 . . .
```c
#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 );

    VkShaderModule ShaderModuleVertex;
    . . .
    VkPipelineShaderStageCreateInfo
    VertexInput State
    InputAssembly State
    Tesselation State
    Viewport State
    Rasterization State
    MultiSample State
    DepthStencil State
    ColorBlend State
    Dynamic State
    Pipeline layout
    RenderPass
    basePipelineHandle
    basePipelineIndex
    VkPipelineShaderStageCreateInfo
    
    VkPipelineInputAssemblyStateCreateInfo
    VkViewportStateCreateInfo
    x, y, w, h,
    minDepth,
    maxDepth
    offset
    extent
    VkPipelineRasterizationStateCreateInfo
    cullMode
    polygonMode
    frontFace
    lineWidth
    VkPipelineColorBlendAttachmentState
    depthTestEnable
    depthWriteEnabledepthCompareOp
    stencilTestEnable
    stencilOpStateFront
    stencilOpStateBack
    blendEnable
    srcColorBlendFactor
    dstColorBlendFactor
    colorBlendOp
    srcAlphaBlendFactor
    dstAlphaBlendFactor
    alphaBlendOp
    colorWriteMask
    VkPipelineDynamicStateCreateInfo
    vkCreateGraphicsPipeline( )

SPIR-V: More Information
SPIR-V Tools:
http://github.com/KhronosGroup/SPIRV-Tools
```

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 MyVertexBuffer;
InitMyVertexDataBuffer( sizeof(VertexData), &MyVertexBuffer );
FillMyDataBuffer( MyVertexBuffer, (void *) VertexData );

VkResult
InitMyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result = InitDataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
```
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 int32_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);

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

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

### Finding the Right Type of Memory

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

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

### Something I’ve Found Useful

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

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

```c
MyBuffer MyMatrixUniformBuffer;
```

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

```c
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer ) {
    vbci.size = pMyBuffer->size = size;
    VkResult result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer);
    pMyBuffer->vdm = vdm;
    return result;
}
```
Here's a C struct to hold some uniform variables

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

Here's the shader code to access those uniform variables

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

Filling those Uniform Variables

```c
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

```
MyBuffer MyMatrixUniformBuffer;
```

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

```
uniform matBuf Matrices;
```

The Descriptor Set for the Buffer

We will come to **Descriptor Sets** later, but for 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.

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

VkWriteDescriptorSet vwds0;
    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;

    vkUpdateDescriptorSets( LogicalDevice, 1, &vwds0, 0, (VkCopyDescriptorSet *)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;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, pMyBuffer->buffer, &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, &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, vdm, 0 );
    // 0 is the offset
    return result;
}
```

```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, 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, 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

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
```c
void
InitGLFW()
{
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
    MainWindow = glfwCreateWindow(Width, Height, "Vulkan Sample", NULL, NULL);
    VkResult result = glfwCreateWindowSurface(Instance, MainWindow, NULL, &Surface);
    glfwSetErrorCallback(GLFWErrorCallback);
    glfwSetKeyCallback(MainWindow, GLFWKeyboard);
    glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
    glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

```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);
        }
    }
}
```

```c
void
GLFWMouseButton(GLFWwindow *window, int button, int action, int mods)
{
    int b = 0; // LEFT, MIDDLE, or RIGHT
    switch(button)
    {
    case GLFW_MOUSE_BUTTON_LEFT:
        b = LEFT;
        break;
    case GLFW_MOUSE_BUTTON_MIDDLE:
        b = MIDDLE;
        break;
    case GLFW_MOUSE_BUTTON_RIGHT:
        b = RIGHT;
        break;
    default:
        b = 0;
        fprintf(FpDebug, "Unknown mouse button: %d
", button);
        break;
    }
    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
    }
}
```

```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);
        if (Scale < MINSCALE)
            Scale = MINSCALE;
    }
    Xmouse = (int)xpos;
    Ymouse = (int)ypos;
}
```
Looping and Closing GLFW

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

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

What is GLM?

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

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

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

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

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

Why are we even talking about this?

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

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

you would now have to say:

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

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

// constructor:
glm::mat4();  // identity matrix
glm::vec4();  

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

// multiplications:
glm::mat4 * glm::mat4
glm::mat4 * glm::vec4
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. Telling Visual Studio about where the GLM folder is

2. 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.

3. GLM in the Vulkan sample.cpp Program

4. Your Sample2019.zip File Contains GLM Already
Why Isn't The Normal Matrix just the Same as the Model Matrix?

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

Original object and normal

Wrong!

Right!

\[
\text{glm::mat3 NormalMatrix = glm::inverseTranspose( glm::mat3(Model) );}
\]

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

Instancing – What and why?

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

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

Making each Instance look differently – Approach #1

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

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

Making each Instance look differently – Approach #1

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

\[
\text{int NUMINSTANCES = 16;}
\text{float DELTA = 3.0;}
\text{float xdelta = DELTA * float( gl_InstanceIndex % 4 );}
\text{float ydelta = DELTA * float( gl_InstanceIndex / 4 );}
\text{vColor = vec3( 1., float( (1.+gl_InstanceIndex) ) / float( NUMINSTANCES ), 0. );}
\text{xdelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;}
\text{ydelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;}
\text{vec4 vertex = vec4( aVertex.xyz + vec3( xdelta, ydelta, 0. ), 1. );}
\text{gl_Position = PVM * vertex;}
\]
Put the unique characteristics in a uniform buffer and reference them
Still uses gl_InstanceIndex

In the vertex shader:
```
int index = gl_InstanceIndex % 1024; // 0 - 1023
vColor = Colors.uColors[ index ];
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

How We Constructed the Graphics Pipeline Structure Before

#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

```
layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
layout ( location = 4 ) in vec3 aInstanceColor;

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

void main( ){
  mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
  vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );
  vColor = aInstanceColor;
  vTexCoord = aTexCoord;
  gl_Position = PVM * vec4( aVertex, 1. );
}
```

How We Write the Vertex Shader Now

```
layout( binding = 0 ) uniform colorBuf
{
  vec3 uColors[1024];
} Colors;

out vec3 vColor;

int index = gl_InstanceIndex % 1024; // 0 - 1023
vColor = Colors.uColors[ index ];
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```
In OpenGL

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

Each uniform variable gets updated one-at-a-time.

Wouldn’t it be nice if we could update a collection of related uniform variables all at once, without having to update the uniform variables that are not related to this collection?

```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 float uMode;
layout( binding = 7 ) uniform sampler2D uSampler;
```

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

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

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

What are Descriptor Sets?

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

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

**CPU:**
- Uniform data created in a C++ data structure
- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size

**GPU:**
- Uniform data used in the shader
- Knows the shader data structure
- Knows where the data starts
- Knows the data's size
- Doesn't know where each piece of data starts

**GPU:**
- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size
- Doesn't know the CPU or GPU data structure

### C++ Data Structure

```cpp
struct matBuf {
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
    glm::mat3 uNormalMatrix;
};
struct lightBuf {
    glm::vec4 uLightPos;
};
struct miscBuf {
    float uTime;
    int uMode;
};
```

```cpp
layout( std140, set = 0, binding = 0 ) uniform matBuf{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
}
Matrices;
layout( std140, set = 1, binding = 0 ) uniform lightBuf{
    vec4 uLightPos;
}
Light;
layout( std140, set = 2, binding = 0 ) uniform miscBuf{
    float uTime;
    int uMode;
}
Misc;
```

### Step 1: Descriptor Set Pools

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

```cpp
VkResult Init13DescriptorSetPool() {
    VkResult result;
    VkDescriptorPoolSize vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;
    #ifdef CHOICES
        VK_DESCRIPTOR_TYPE_SAMPLER
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
        VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
        VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
        VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
        VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
        VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
        VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
    #endif
    VkDescriptorPoolCreateInfo vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;
    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = &vdps[0];
    result = vkCreateDescriptorPool
        ( LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
    return result;
}
```

### Step 2: Define the Descriptor Set Layouts

I think of Descriptor Set Layouts as a kind of "Rosetta Stone" that allows the Graphics Pipeline data structure to allocate room for the uniform variables and to access them.

**MatrixSet DS Layout**
- Binding: set = 0, binding = 0
- Uniform: mat4

**LightSet DS Layout**
- Binding: set = 1, binding = 0
- Uniform: vec4

**MiscSet DS Layout**
- Binding: set = 2, binding = 0
- Uniform: float, int

**TexSamplerSet DS Layout**
- Binding: set = 3, binding = 0
- Uniform: sampler2D

---

- [Step 1: Descriptor Set Pools](#)
- [Step 2: Define the Descriptor Set Layouts](#)
Step 2: Define the Descriptor Set Layouts

```cpp
VkResult Init13DescriptorSetLayouts()
{
    VkResult result;
    // DS #0:
    MatrixSet[1] VkDescriptorSetLayoutBinding
    MatrixSet[0].binding            = 0;
    MatrixSet[0].descriptorType     = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MatrixSet[0].descriptorCount    = 1;
    MatrixSet[0].stageFlags         = VK_SHADER_STAGE_VERTEX_BIT;
    MatrixSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #1:
    LightSet[1] VkDescriptorSetLayoutBinding
    LightSet[0].binding            = 0;
    LightSet[0].descriptorType     = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    LightSet[0].descriptorCount    = 1;
    LightSet[0].stageFlags         = VK_SHADER_STAGE_FRAGMENT_BIT;
    LightSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #2:
    MiscSet[1] VkDescriptorSetLayoutBinding
    MiscSet[0].binding            = 0;
    MiscSet[0].descriptorType     = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    MiscSet[0].descriptorCount    = 1;
    MiscSet[0].stageFlags         = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    MiscSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #3:
    TexSamplerSet[1] VkDescriptorSetLayoutBinding
    TexSamplerSet[0].binding            = 0;
    TexSamplerSet[0].descriptorType     = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    TexSamplerSet[0].descriptorCount    = 1;
    TexSamplerSet[0].stageFlags         = VK_SHADER_STAGE_FRAGMENT_BIT;
    TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;

    return result;
}
```

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

```cpp
VkResult Init14GraphicsPipelineLayout()
{
    VkResult result;
    VkDescriptorSetLayoutCreateInfo vdslc0
    vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc0.pNext = nullptr;
    vdslc0.flags = 0;
    vdslc0.bindingCount = 1;
    vdslc0.pBindings = &MatrixSet[0];

    VkDescriptorSetLayoutCreateInfo vdslc1
    vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc1.pNext = nullptr;
    vdslc1.flags = 0;
    vdslc1.bindingCount = 1;
    vdslc1.pBindings = &LightSet[0];

    VkDescriptorSetLayoutCreateInfo vdslc2
    vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc2.pNext = nullptr;
    vdslc2.flags = 0;
    vdslc2.bindingCount = 1;
    vdslc2.pBindings = &MiscSet[0];

    VkDescriptorSetLayoutCreateInfo vdslc3
    vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    vdslc3.pNext = nullptr;
    vdslc3.flags = 0;
    vdslc3.bindingCount = 1;
    vdslc3.pBindings = &TexSamplerSet[0];

    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}
```
Step 4: Allocating the Memory for Descriptor Sets

```c
VkResult
Init13DescriptorSets()
{
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;
    result = vkAllocateDescriptorSets(LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
}
```

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

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

This struct identifies what texture sampler and image view it owns

```c
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 = IN &vdbi0;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pTexelBufferView = (VkBufferView *)nullptr;
```

```c
// ds 1:
VkWriteDescriptorSet
vwds1;
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds1.pNext = nullptr;
vwds1.dstSet = DescriptorSets[1];
vwds1.dstBinding = 0;
vwds1.dstArrayElement = 0;
vwds1.descriptorCount = 1;
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.pBufferInfo = IN &vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
```

```c
// ds 2:
VkWriteDescriptorSet
vwds2;
// ds 2:
vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds2.pNext = nullptr;
vwds2.dstSet = DescriptorSets[2];
vwds2.dstBinding = 0;
vwds2.dstArrayElement = 0;
vwds2.descriptorCount = 1;
vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds2.pBufferInfo = IN &vdbi2;
vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds2.pTexelBufferView = (VkBufferView *)nullptr;
```

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

```c
uint32_t copyCount = 0;// this could have been done with one call and an array of VkWriteDescriptorSets:
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);
```
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

```
VkGraphicsPipelineCreateInfo
  
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;

#ifdef CHOICESVK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
  VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
#endif

gpci.stageCount = 2; // number of stages in this pipeline

gpci.pStages = ...
  ...
  ...
  &vpiasci;

gpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;

gpci.pViewportState = &vpvsci;

gpci.pRasterizationState = &vprsci;

gpci.pMultisampleState = &vpmsci;

gpci.pDepthStencilState = &vpdssci;

gpci.pColorBlendState = &vpcbsci;

gpci.pDynamicState = &vpdsci;

gpci.layout = IN GraphicsPipelineLayout;

gpci.renderPass = IN RenderPass;

gpci.subpass = 0; // subpass numbervgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;

gpci.basePipelineIndex = 0;

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

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

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

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

---

Triangles in an Array of Structures
GPU Memory

Host
Visible
CPU Memory

Device
Local
Texture
Sampling Hardware

Host
Coherent

Device
Local

GPU Memory

RGBA to the Shader

Memory Types

NVIDIA Discrete Graphics:
11 Memory Types:
Memory 0: Memory 1:
Memory 2: Memory 3:
Memory 4: Memory 5:
Memory 6: Memory 7: Device.Local
Memory 8: Device.Local
Memory 9: Host.Visible Host.Coherent

Intel Integrated Graphics:
3 Memory Types:
Memory 0: Device.Local
Memory 1: Device.Local Host.Visible Host.Coherent
Memory 2: Device.Local Host.Visible Host.Coherent Host.Cached

Texture Sampling Parameters

OpenGL

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );

Vulkan

VkSamplerCreateInfo vsci;
vsci.magFilter = VK_FILTER_LINEAR;
vsci.minFilter = VK_FILTER_LINEAR;
vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, pTextureSampler );

Texture Mip*-mapping

Average 4 pixels to make a new one

Average 4 pixels to make a new one

Average 4 pixels to make a new one

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

Graphics hardware determines which level to use based on the texels : pixels ratio.

In addition to just picking one mip-map level, the rendering system can sample from two of them, one less that the T:P ratio and one more, and then blend the two RBGAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin, multum in parvo, "many things in a small place"
VkResult Init07TextureSampler(MyTexture * pMyTexture)
{
    VkSamplerCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.mipLodBias = 0.;
    vsci.anisotropyEnable = VK_FALSE;
    vsci.maxAnisotropy = 1.;
    vsci.compareEnable = VK_FALSE;
    vsci.compareOp = VK_COMPARE_OP_NEVER;
    vsci.imageType = VK_IMAGE_TYPE_2D;
    #ifdef CHOICES
    vsci.format = VK_FORMAT_R8G8B8A8_UNORM;
    // VK_IMAGE_TILING_LINEAR
    // VK_IMAGE_USAGE_TRANSFER_SRC_BIT
    // VK_IMAGE_USAGE_TRANSFER_DST_BIT
    // VK_IMAGE_USAGE_SAMPLED_BIT
    // VK_IMAGE_USAGE_STORAGE_BIT
    // VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
    // VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
    // VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
    // VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
    // VK_BORDER_COLOR_INT_OPAQUE_BLACK
    #endif
    vsci.mipLevels = 1;
    vsci.arrayLayers = 1;
    vsci.samples = VK_SAMPLE_COUNT_1_BIT;
    vsci.tiling = VK_IMAGE_TILING_LINEAR;
    vsci.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vsci.unnormalizedCoordinates = VK_FALSE;

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

    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.compareEnable = VK_FALSE;
    vici.compareOp = VK_COMPARE_OP_NEVER;
    vici.imageType = VK_IMAGE_TYPE_2D;
    #ifdef CHOICES
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    // VK_IMAGE_TILING_LINEAR
    // VK_IMAGE_USAGE_TRANSFER_SRC_BIT
    // VK_IMAGE_USAGE_TRANSFER_DST_BIT
    // VK_IMAGE_USAGE_SAMPLED_BIT
    // VK_IMAGE_USAGE_STORAGE_BIT
    // VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
    // VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
    // VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
    // VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
    // VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
    // VK_BORDER_COLOR_INT_OPAQUE_BLACK
    #endif
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

    result = vkCreateImage(LogicalDevice, &vici, PALLOCATOR, OUT &stagingImage);
    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);

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

    if (vsl.rowPitch == 4 * texWidth)
    {
        VkMemoryAllocateInfo vmai;
        memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        vmai.pNext = nullptr;
        vmai.allocationSize = vmr.size;
        vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
        VkDeviceMemory vdm;
        result = vkAllocateMemory(LogicalDevice, &vmai, PALLOCATOR, OUT &vdm);
        unsigned char *gpuBytes = (unsigned char *)gpuMemory;
        for (unsigned int y = 0; y < texHeight; y++)
        {
            pMyTexture->vdm = vdm;
            memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
        }
    }
    else
    {
        VkDeviceMemory vdm;
        result = vkAllocateMemory(LogicalDevice, &vmai, PALLOCATOR, OUT &vdm);
        unsigned char *gpuBytes = (unsigned char *)gpuMemory;
        for (unsigned int y = 0; y < texHeight; y++)
        {
            pMyTexture->vdm = vdm;
            memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
        }
    }
    // we have now created the staging image -- fill it with the pixel data:
    vkUnmapMemory(LogicalDevice, vdm);
// *******************************************************************************
// copy pixels from the staging image to the texture:
// this second {...} is to create the actual texture image:
// *******************************************************************************

VkCommandBufferBeginInfo vcbbi;
{
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_OPTIMAL;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = stagingImage;
    vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = visr;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
}

result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

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

VkMemoryRequirements vmr;

vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
{
    fprintf(FpDebug, "Texture vmr.size = %lld
", vmr.size);
    fprintf(FpDebug, "Texture vmr.alignment = %lld
", vmr.alignment);
    fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
    fflush(FpDebug);
}

VkMemoryAllocateInfo vmai;

vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal();  // because we want to sample from it

VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindImageMemory(LogicalDevice, IN textureImage, IN vdm, 0);  // 0 = offset

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

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

VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = 0;
vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
vimb.subresourceRange = visr;

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

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

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

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

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

VkExtent3D ve3;
ve3.width = texWidth;
ve3.height = texHeight;
ve3.depth = 1;
Reading in a Texture from a BMP File

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

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

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

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

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.
What is the Vulkan Graphics Pipeline?

The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context". It is a data structure.

The Vulkan Graphics Pipeline is not the processes that OpenGL would call "the graphics pipeline".

For the most part, the Vulkan Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.

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

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

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

Here's what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context". 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 meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

Let the Pipeline Layout know about the Descriptor Set and Push Constant layouts.

Vulkan: A Pipeline Records the Following Items:

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

**Bold/Italics** indicates that this state item can also be set with Dynamic Variables.
Creating a Typical Graphics Pipeline

```
Creating a Typical Graphics Pipeline

```

```
Link in the Shaders

```

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

```
Link in the Shaders

```

```
Use one vspsci[] array member per shader module you are using

```

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

```
Link in the Shaders

```

```
Declare the binding descriptions and attribute descriptions

```

```
Declare the vertex topology

```

```
Tessellation Shader info

```

```
Geometry Shader info

```

```
Use one vvibd[] array member per vertex input array-of-structures you are using

```

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

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

```c
vkpiasci.primitiveRestartEnable = VK_FALSE;
```

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

If `vkpiasci.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.

### VkIndexType

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

If your `VkIndexType` is `VK_INDEX_TYPE_UINT16`, then the special index is `0xffff`. If your `VkIndexType` is `VK_INDEX_TYPE_UINT32`, 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 (scrunched) into the viewport area.

**Scissoring:**

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

Declare information about how the rasterization will take place

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, not a solid.

The gray area shows what would happen with depthClampEnable (except 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.

Z-fighting

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.

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

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

vtkPipelineColorBlendStateCreateInfo

vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;

vpcbsci.pNext = nullptr;

vpcbsci.flags = 0;

vpcbsci.logicOpEnable = VK_FALSE;

vpcbsci.logicOp = VK_LOGIC_OP_COPY;

#ifdef CHOICES

VK_LOGIC_OP_CLEAR

VK_LOGIC_OP_AND

VK_LOGIC_OP_AND_REVERSE

VK_LOGIC_OP_COPY

VK_LOGIC_OP_AND_INVERTED

VK_LOGIC_OP_XOR

VK_LOGIC_OP_OR

VK_LOGIC_OP_NOR

VK_LOGIC_OP_EQUIVALENT

VK_LOGIC_OP_INVERT

VK_LOGIC_OP_OR_REVERSE

VK_LOGIC_OP_COPY_INVERTED

VK_LOGIC_OP_OR_INVERTED

VK_LOGIC_OP_NAND

VK_LOGIC_OP_SET
#endif

vpcbsci.attachmentCount = 1;

vpcbsci.pAttachments = &vpcbas;

vpcbsci.blendConstants[0] = 0;

vpcbsci.blendConstants[1] = 0;

vpcbsci.blendConstants[2] = 0;

vpcbsci.blendConstants[3] = 0;

Which Pipeline Variables can be Set Dynamically

VktDynamicState

vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };

#ifdef CHOICES

VK_DYNAMIC_STATE_VIEWPORT

-- vkCmdSetViewport( )

VK_DYNAMIC_STATE_SCISSOR

-- vkCmdSetScissor( )

VK_DYNAMIC_STATE_LINE_WIDTH

-- vkCmdSetLineWidth( )

VK_DYNAMIC_STATE_DEPTH_BIAS

-- vkCmdSetDepthBias( )

VK_DYNAMIC_STATE_BLEND_CONSTANTS

-- vkCmdSetBendConstants( )

VK_DYNAMIC_STATE_DEPTH_BOUNDS

-- vkCmdSetDepthZBounds( )

VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK

-- vkCmdSetStencilCompareMask( )

VK_DYNAMIC_STATE_STENCIL_WRITE_MASK

-- vkCmdSetStencilWriteMask( )

VK_DYNAMIC_STATE_STENCIL_REFERENCE

-- vkCmdSetStencilReference( )
#endif

VkPipelineDynamicStateCreateInfo

vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;

vpdsci.pNext = nullptr;

vpdsci.flags = 0;

vpdsci.dynamicStateCount = 0;

vpdsci.pDynamicStates = vds;

Stencil Operations for Front and Back Faces

VkStencilOpState

vsosf;

vsosb;

vsosf.depthFailOp = VK_STENCIL_OP_KEEP;

vsosf.failOp = VK_STENCIL_OP_KEEP;

vsosf.passOp = VK_STENCIL_OP_KEEP;

vsosf.compareOp = VK_COMPARE_OP_NEVER;

vsosf.compareMask = ~0;

vsosf.writeMask = ~0;

vsosf.reference = 0;

VkStencilOpState

vsosb.depthFailOp = VK_STENCIL_OP_KEEP;

vsosb.failOp = VK_STENCIL_OP_KEEP;

vsosb.passOp = VK_STENCIL_OP_KEEP;

vsosb.compareOp = VK_COMPARE_OP_NEVER;

vsosb.compareMask = ~0;

vsosb.writeMask = ~0;

vsosb.reference = 0;

Uses for Stencil Operations

Polygon edges without Z-fighting

Magic Lenses
Operations for Depth Values

VkPipelineDepthStencilStateCreateInfo

- `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;``

``
VK_COMPARE_OP_NEVER -- never succeedsVK_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
VK_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
VK_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
VK_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
VK_COMPARE_OP_ALWAYS -- always succeeds``

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

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

Putting it all Together! (finally…)

``
VkPipelineCreateCreateInfo

- `sType = VK_STRUCTURE_TYPE_PIPELINE_CREATE_INFO``
- `pNext = nullptr``
- `flags = 0;``

``#ifdef CHOICESVK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BITVK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BITVK_PIPELINE_CREATE_DERIVATIVE_BIT``
``#endif``
``
- `stageCount = 2;``
- `pStages = vpssci;``
- `pVertexInputState = &vpvisci;``
- `pInputAssemblyState = &vpiasci;``
- `pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr``
- `pViewportState = &vpvsci;``
- `pRasterizationState = &vprsci;``
- `pMultisampleState = &vpmsci;``
- `pDepthStencilState = &vpdssci;``
- `pColorBlendState = &vpcbsci;``
- `pDynamicState = &vpdsci;``
- `layout = IN GraphicsPipelineLayout;``
- `renderPass = IN RenderPass;``
- `subpass = 0;``
- `basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;``
- `basePipelineIndex = 0;``

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

Queues and Command Buffers

Mike Bailey

mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

Similarly, we can write a function that finds the proper Queue Family

```c
int FindQueueFamilyThatDoesGraphics() {
  uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT VkQueueFamilyProperties nullptr);
VkQueueFamilyProperties* vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, IN &count, OUT vqfp);
for(unsigned int i = 0; i < count; i++) {
  if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
    return i;
}
return -1;
}```
float queuePriorities[] = {
    1.0f, // one entry per queueCount
};

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

VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1; // # of device queues wanted
vdci.pQueueCreateInfos = IN &vdqci[0]; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.ppEnabledLayerNames = myDeviceLayers;
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures; // already created

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

VkQueue Queue;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
uint32_t queueIndex = 0;
result = vkGetDeviceQueue(LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue);

The code above demonstrates how to create a logical device in Vulkan. It involves creating a queue family index and then using it to create a queue. The logical device is also created here.

Creating the Command Pool as part of the Logical Device

VkResult Init06CommandPool() {
    VkResult result;
    VkCommandPoolCreateInfo vcpci;
    vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcpci.pNext = nullptr;
    vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();

    result = vkCreateCommandPool(LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
    return result;
}

Creating the Command Buffers

VkResult Init06CommandBuffers() {
    VkResult result;
    VkCommandBufferAllocateInfo vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = CommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 2; // 2, because of double-buffering

    result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &CommandBuffers[nextImageIndex]);

    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    vcbai.commandBufferCount = 1;
    result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer);

    return result;
}

Finally, we can see the code for creating a semaphore to signal when the next image is ready:

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

Beginning a Command Buffer

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

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

VkCommandBufferInheritanceInfo cbii;
result = vkGetCommandBufferInheritanceInfo(CommandBuffers[nextImageIndex], OUT &cbii);

result = vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
These are the Commands that could be entered into the Command Buffer, I

```c
vkCmdBeginQuery(vkDevice, queue, queryPool, queryID);
vkCmdEndQuery(vkDevice, queue, queryPool, queryID);
vkCmdBeginRenderPass(vkDevice, renderPass, framebuffers[0], renderArea);
```

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

```c
vkCmdBindPipeline(vkDevice, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
vkCmdBindPipeline(vkDevice, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
vkCmdDispatch(vkDevice, groupCountX, groupCountY, groupCountZ);
```

---

```c
VkClearColorValue clearColor = { 0.0f, 0.0f, 0.0f, 1.0f };
VkClearDepthStencilValue clearDepth = { 1.0f, 0 };
```

```c
VkSemaphoreCreateInfo semaphoreCreateInfo = {
    .sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
};
```

```c
VkCommandBufferBeginInfo commandBufferBeginInfo = {
    .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    .pNext = nullptr,
    .flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,
};
```
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, 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.
The Swap Chain

Vulkan

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

Mike Bailey
mjb@cs.oregonstate.edu

How We Think of OpenGL Framebuffers

Video Driver
Update
Front
Back
Back
Swap Chain

Refresh
Depth-Buffer
Update

Vulkan Thinks of it This Way

Swap Chain
Depth-Buffer
Update
Front

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.

We Need to Find Out What our Display Capabilities Are

```c
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "\n\nvkGetPhysicalDeviceSurfaceCapabilitiesKHR: \n" );
VkBool32 supported;
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics( ), Surface, &supported );
if( supported == VK_TRUE )
    fprintf( FpDebug, "** This Surface is supported by the Graphics Queue ** \n" );
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
printf( FpDebug, "\nFound %d Surface Formats:\n", formatCount )
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
printf( FpDebug, "\nFound %d Present Modes:\n", presentModeCount );
```
Creating a Swap Chain

vkCreateSwapchain( )

VkSwapchainCreateInfo

surface
imageFormat
imageColorSpace
imageExtent
imageArrayLayers
imageUsage
imageSharingMode
preTransform
compositeAlpha
presentMode
clipped

vkGetDevicePhysicalSurfaceCapabilities( )

VkSurfaceCapabilities

minImageCount
maxImageCount
currentExtent
minImageExtent
maxImageExtent
maxImageArrayLayers
supportedTransforms
currentTransform
supportedCompositeAlpha

vkGetSwapChainImages( )

vkCreateImageView( )

Creating a Swap Chain

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

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr;
vscci.flags = 0;
surface = Surface;
vscci.minImageCount = 2;
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;
vkCreateSwapchainKHR
( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );

PresentImages = new VkImage[imageCount];
result = vkGetSwapchainImagesKHR
( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );

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] );
}
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;

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

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

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;

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

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

VkPresentInfoKHR vpi;
vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;vpi.pNext = nullptr;
vpi.waitSemaphoreCount = 0;
vpi.pWaitSemaphores = (VkSemaphore *)nullptr;vpi.swapchainCount = 1;vpi.pSwapchains = &SwapChain;
vpi.pImageIndices = &nextImageIndex;
vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR(presentQueue, IN &vpi);
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );

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

Where to put them
How many total there are

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

Which Physical Device to Use, I

Which Physical Device to Use, II

Which Physical Device to Use
Asking About the Physical Device’s Features

```c
VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
fprintf( FpDebug, "Physical Device Features:
" );
fprintf( FpDebug, "geometryShader = %2d
", PhysicalDeviceFeatures.geometryShader);
fprintf( FpDebug, "tessellationShader = %2d
", PhysicalDeviceFeatures.tessellationShader );
fprintf( FpDebug, "multDrawIndirect = %2d
", PhysicalDeviceFeatures.multDrawIndirect );
fprintf( FpDebug, "largePoints = %2d
", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "multiViewport = %2d
", PhysicalDeviceFeatures.multiViewport);
fprintf( FpDebug, "occlusionQueryPrecise = %2d
", PhysicalDeviceFeatures.occlusionQueryPrecise );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d
", PhysicalDeviceFeatures.pipelineStatisticsQuery );
fprintf( FpDebug, "shaderFloat64 = %2d
", PhysicalDeviceFeatures.shaderFloat64 );
fprintf( FpDebug, "shaderInt64 = %2d
", PhysicalDeviceFeatures.shaderInt64 );
fprintf( FpDebug, "shaderInt16 = %2d
", PhysicalDeviceFeatures.shaderInt16 );
```

Here’s What the NVIDIA RTX 2080 Ti Produced

```c
vkEnumeratePhysicalDevices:
Device  0:
API version: 4198499
Driver version: 4198499
Vendor ID: 0x10de
Device ID: 0x1e04
Physical Device Type: 2 = (Discrete GPU)
Device Name: RTX 2080 Ti
Pipeline Cache Size: 206
Device #0 selected (‘RTX 2080 Ti’)
Physical Device Features:
geometryShader = 1
tessellationShader = 1
multDrawIndirect = 1
wideLines = 1
largePoints = 1
multiViewport = 1
occlusionQueryPrecise = 1
pipelineStatisticsQuery = 1
shaderFloat64 = 1
shaderInt64 = 1
shaderInt16 = 1
```

Asking About the Physical Device’s Different Memories

```c
VkPhysicalDeviceMemoryProperties vdpmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vdpmp );
for( unsigned int i = 0; i < vdpmp.memoryTypeCount; i++ )
{
    VkMemoryType vmt = vdpmp.memoryTypes[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, "
" );
}
```
Here's What I Got

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

2 Memory Heaps:
- Heap 0: size = 0xa7c00000 DeviceLocal
- Heap 1: size = 0xfac00000

Here's What I Got

uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
fprintf( FpDebug, "Found %d Queue Families:\n", count );

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

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"
};

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

// see what device extensions are available:
uint32_t  extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
&extensionCount, (VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
&extensionCount, deviceExtensions[]);

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

0x00401072   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'

0x00401072   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'

0x00401072   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = 1;  // # of device queues
vdci.pQueueCreateInfos = (const VkDeviceQueueCreateInfo *)&vdqci;  // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.enabledLayerCount = 0;
vdci.ppEnabledLayerNames = myDeviceLayers;
vkGetDeviceQueue(LogicalDevice, 0, 0, &Queue);  // 0, 0 = queueFamilyIndex, queueIndex

Vulkan: Creating a Logical Device

float queuePriorities[1] = {
    1
};
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.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

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

VkGraphicsPipelineCreateInfo vgpci;  
// ...

if (you declare certain state variables to be dynamic like this, then you must fill them in the command buffer! Otherwise, they are undefined.)
```

Filling the Dynamic State Variables in the Command Buffer

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

```c
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, writeMask);
vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
vkCmdSetStencilReference(commandBuffer, faceMask, reference);
```

Push Constants

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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

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

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

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

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

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

```cpp
VkPushConstantRange vpcr[1];
vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vpcr[0].offset = 0;
vpcr[0].size = sizeof( glm::mat4 );

VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = &vpcr[0];

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```
An Robotic Example using Push Constants

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

Where each arm is represented by:

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

In the Reset Function

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

Arm1.armMatrix = glm::mat4();
Arm1.armColor = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale = 6.f;

Arm2.armMatrix = glm::mat4();
Arm2.armColor = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale = 4.f;

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

Setup the Push Constant for the Pipeline Structure

```
VkPushConstantRange vpcr[1];
    vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
                      VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
    vpcr[0].offset = 0;
    vpcr[0].size = sizeof( 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, IN &vplci,
                                 OUT &GraphicsPipelineLayout );
```

In the UpdateScene Function

```
float rot1 = (float)Time;
float rot2 = 2.f * rot1;
float rot3 = 2.f * rot2;

Arm1.armMatrix = m1g;
Arm1.armColor = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale = 6.f;

Arm2.armMatrix = glm::mat4();
Arm2.armColor = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale = 4.f;

Arm3.armMatrix = glm::mat4();
Arm3.armColor = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale = 2.f;
```
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.x *= RobotArm.armScale;  // now is [0., RobotArm.armScale]
bVertex = vec3( RobotArm.armMatrix * vec4( bVertex, 1. ) );

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

---

**Getting Information Back from the Graphics System**

Mike Bailey
mjb@cs.oregonstate.edu

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

---

**Setting up Query Pools**

- 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
```
VkQueryPoolCreateInfo vqpci;
    vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
    vqpci.pNext = nullptr;
    vqpci.flags = 0;
    vqpci.queryType = VK_QUERY_TYPE_OCCLUSION | VK_QUERY_TYPE_PIPELINE_STATISTICS | VK_QUERY_TYPE_TIMESTAMP;
    vqpci.queryCount = 1;
    vqpci.pipelineStatistics = 0;

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

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

VkQueryPool timestampQueryPool;
result = vkCreateQueryPool( LogicalDevice, IN &vqpci, PALLOCATOR, OUT &timestampQueryPool);
```

### Occlusion Query

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

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

**Some hints:**

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

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

### Pipeline Statistics Query

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

```
uint32_t counts[NUM_STATS];
result = vkGetQueryPoolResults( LogicalDevice, statisticsQueryPool, 0, 1, sizeof(uint32_t), counts, 0, VK.QueryResultWaitBit);
```
Timestamp Query

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

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

---

**Timestamp Query**

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

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

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

---

**Compute Shaders**

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Start by Creating the Data Buffers

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

```cpp
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
};
```

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

Creating a Shader Storage Buffer

```cpp
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 int32_t) nullptr;

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

Creating a Shader Storage Buffer

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

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

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

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

```cpp
VkResult Fill05DataBuffer ( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &GpuMemory );
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, myBuffer.vdm );
    return VK_SUCCESS;
}
```
Create the Compute Pipeline Layout

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

Create the Compute Pipeline

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

The Particle System Compute Shader -- Setup

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

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

The Particle System Compute Shader -- The Physics

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

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POIINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;
POIINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;
Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;
```
VELOCITY Bounce( VELOCITY vin, VECTOR n )
{
    VELOCITY vout = reflect( vin, n );
    return vout;
}

VELOCITY BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
    VECTOR n = normalize( p - s.xyz );
    return Bounce( v, n );
}

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

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

Dispatching the Compute Shader from the Command Buffer

const int NUM_PARTICLES = 1024*1024;
const int NUM_WORK_ITEMS = 64;
const int 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.

Or,

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 );  // Buffer holds the 3 sizes, offset=0
What Are Specialization Constants?

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

Normally, the half-way 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 sneak 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 rest-of-the-way compilation.

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

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

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

Specialization Constant Example -- Setting an Array Size

In the compute shader:

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

In the Vulkan C/C++ program:

```c
int asize = 64;
VkSpecializationMapEntry vsme[1];
// one array element for each
// Specialization Constant
vsme[0].constantID = 7;
// bytes into the Specialization Constant
vsme[0].offset = 0; // # bytes into the Specialization Constant
vsme[0].size = sizeof(asize); // size of just this Specialization Constant

VkSpecializationInfo vsi;
// array this one item is
vsi.mapEntryCount = 1;
// # Specialization Constants
vsi.pMapEntries = vsme;
// size of all the Specialization Constants
tsizeof(vsi.mapEntryCount);
// array of all the Specialization Constants
vsipDataSize = sizeof(asize);
```

In the Vulkan C/C++ program:

```c
vkCreateComputePipelines();
```
Linking the Specialization Constants into the Compute Pipeline

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

VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(asize);
vsii.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 Constants – Setting Multiple Constants

In the compute shader:
```c
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:
```c
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;
vsii.mapEntryCount = 3;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsii.pData = &abc; // array of all the Specialization Constants
```

Specialization Constants – Setting the Number of Work-items Per Work-Group in the Compute Shader

In the compute shader:
```c
layout( local_size_x_id = 12 ) in;
layout(local_size_x = 32, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:
```c
int numXworkItems = 64;
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 12;
vsme[0].offset = 0;
vsme[0].size = sizeof(int);

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

Synchronization

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Where Synchronization Fits in the Overall Block Diagram

- **Application**
- **Instance**
- **Physical Device**
- **Logical Device**
- **Queue**
- **Command Buffer**
- **Event**
- **Semaphore**
- **Host**

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

#### Semaphores Example during the Render Loop

```c
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 );
```

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

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

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

Creating a Semaphore

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

```c
 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 );
```

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

```c
VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;vsi.commandBufferCount = 1;vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore) nullptr;
```

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

```
// Could be an array of fences
```

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

```
```
Controlling Events from the Host

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

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

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

Note: the host cannot block waiting for an event, but it can test for it.

Controlling Events from the Device

result = vkCmdSetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdResetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdWaitEvents(CommandBuffer, 1, &event, srcPipelineStageBits, dstPipelineStageBits, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);

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

Could be an array of events
Where signaled, where wait for the signal
Memory barriers get executed after events have been signaled

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

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

http://cs.oregonstate.edu/~mjb/vulkan
From the Command Buffer Notes:
These are the Commands that can be entered into the Command Buffer, II

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

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

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

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

- `vkCmdPipelineBarrier` (commandBuffer, srcStageMask, dstStageMask)

- Defines what data we will be blocking/un-blocking on

**The Scenario**

- TOP_OF_PIPE
- VERTEX_INPUT
- VERTEX_SHADER
- TRANSFER_BIT
- COLOR_ATTACHMENT_OUTPUT
- FRAGMENT_SHADER
- BOTTOM_OF_PIPE
- pipelineStage, queryPost, query

src cars

dst cars
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 last 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

The Scenario

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

```
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_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
```

```
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
```

```
src cars
dst cars
```

**The Scenario**

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

```
vink = VkImageMemoryBarrier
vink.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vink.pNext = nullptr;
vink.srcAccessMask = ??;
vink.dstAccessMask = ??;
vink.oldLayout = ??;
vink.newLayout = ??;
vink.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vink.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vink.image = ??;
vink.subresourceRange = visr;
```
Aliasing

The Display We Want

Too often, the Display We Get

"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.
Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

\[ \text{Final Pixel Color} = \frac{\sum \text{Color sample from subpixel}_i}{\# \text{Fragment Shader calls}} \]

# Fragment Shader calls = 8
Multisampling

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

0. produces simple multisampling
(0.1) produces partial supersampling
1. produces complete supersampling

VkPipelineMultisampleStateCreateInfo

vkmsci.pNext = nullptr;
vkmsci.flags = 0;
vkmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vkmsci.sampleShadingEnable = VK_TRUE;
vkmsci.minSampleShading = 0.5;
vkmsci.pSampleMask = (VkSampleMask *)nullptr;
vkmsci.alphaToCoverageEnable = VK_FALSE;
vkmsci.alphaToOneEnable = VK_FALSE;

VkGraphicsPipelineCreateInfo

vkpci.pMultisampleState = &vkmsci;

VkAttachmentDescription

vkad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vkad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vkad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vkad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
vkad[0].flags = 0;
vkad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vkad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vkad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vkad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
vkad[1].flags = 0;

VkAttachmentReference

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

VkAttachmentReference

vkcref.depthReference.attachment = 0;
vkcref.depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
Slide 309

Setting up the Image

```c
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 = vsd;
vrpci.subpassCount = 1;
vrpci.pSubpasses = &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (VkSubpassDependency *)nullptr;

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

Slide 310

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

```c
void x = 0;
void y = 0;
void z = 0;

VkImageResolve

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

Slide 311

Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Computer Graphics API

Thanks for coming today!

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