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?

Vulkan Differences from OpenGL

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

- GLSL is the same as before ... almost
- For places it's not, an implied
  \#define VULKAN 100
  is automatically supplied by the compiler
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)
- SPIR-V gets turned into fully-compiled code at runtime
- The SPIR-V spec has been public for months – new shader languages are surely being developed
- OpenCL and OpenGL 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
Simpler drivers for low-
overhead efficiency and
cross vendor portability
Layered architecture so
validation and debug
layers can be unloaded
when not needed
Run-time only has to
ingest SPIR-V
intermediate language
Unified API for mobile,
desktop, console and
embedded platforms
Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- Think OpenCL...
- E.g., vkCmdDoSomething(cmdBuffer, ...);
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed 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 Quick Reference Card – I Recommend you Get This!


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

```
Application
  Instance
  Physical Device
  Logical Device
  Command Buffer
```

**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 RenderPass
14. Create the Framebuffer(s)
15. Create the Descriptor Set Pool
16. Create the Command Buffer Pool
17. Create the Command Buffer(s)
18. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render ...

**The Vulkan Sample Code Included with These Notes**

Mike Bailey
mjb@cs.oregonstate.edu

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

- 'Y', 'y': Toggle using a vertex buffer only vs. an index buffer
- 'L', 'l': Toggle lighting off and on
- 'M', 'm': Toggle display mode (textures vs. colors, for now)
- 'P', 'p': Pause the animation
- 'Q', 'q': quit the program
- Esc: quit the program
- 'R', 'r': Toggle rotation-animation and using the mouse

Caveats on the Sample Code, I

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

Caveats on the Sample Code, II

6. There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'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

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

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

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

```
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. },  
};
```

```
static GLuint CubeTriangleIndices[ ][ 3 ] =
{  
    { 0, 2, 3 },  
    { 0, 1, 2 },  
    { 4, 1, 5 },  
    { 0, 1, 5 },  
    { 4, 5, 6 },  
    { 6, 0, 3 },  
    { 0, 6, 3 },  
    { 1, 6, 7 },  
};
```
#include “SampleVertexData.cpp”

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

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

The Vertex Data is in a Separate File

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 those typedef’d structs pass all the necessary information into a function. For example, where we might normally say in C++:
```
result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex,  OUT &Queue );
```
we would actually say in C:
```
result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex,  OUT &Queue );
```

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

```
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.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;
VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &Buffer );
```

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

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

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

Vulkan Conventions

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.

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

Where to put them

How many total there are

Where to put them

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

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

```c
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr);
result = vkEnumeratePhysicalDevices( Instance, &count, &physicalDevices[0]);
```

Extras in the Code

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

The “19” refers to the version of Visual Studio, not the year of development.
Geometry vs. Topology

Geometry: Where things are (e.g., coordinates)
Topology: How things are connected

Geometry = changed
Topology = same [1-2-3-4-1]

Geometry = same
Topology = changed [1-2-4-3-1]

Original Object

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:

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

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

X
Y
Z

0 1 2
CCW

X
Y
Z

3 2 1
CW

Vertex Orientation Issues

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

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

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

VkPipelineRasterizationStateCreateInfo vprsci;
...

vprsci.cullMode = VK_CULL_MODE_NONE
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;

Triangles Represented as an Array of Structures

From the file SampleVertexData.cpp:

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

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

    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. },

    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
};
```

Non-indexed Buffer Drawing

From the file SampleVertexData.cpp:

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

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

    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. },

    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0., 1. }
};
```
Filling the Vertex Buffer

```
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
```

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

A Preview of What Init05DataBuffer Does

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

Telling the Pipeline about its Input

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

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

C/C++:

```
vbo[0].binding = 0; // which binding # is
vbo[0].stride = sizeof(struct vertex);            // bytes between successive structs
vbo[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

GLSL Shader:

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

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

Telling the Pipeline about its Input

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

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

```
VkVertexInputAttributeDescription vviad[4];      // array per vertex input attribute
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position ); // 0
```

```
vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;      // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal ); // 12
```

```
vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;      // r, g, b
vviad[2].offset = offsetof( struct vertex, color ); // 24
```

```
vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;      // s, t
vviad[3].offset = offsetof( struct vertex, texCoord ); // 36
```
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);
```

The `vkCmdBindVertexBuffers` function is used to bind the vertex buffer to the command buffer. The vertex buffer contains the vertex data, which is used to draw the vertices.

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

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

The `VkIndexType` enumeration is used to specify the type of index data that is used in the index buffer.

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

The `Init05MyIndexDataBuffer` function initializes the index data buffer.

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

Init05MyIndexDataBuffer initializes the index data buffer with the size of the `JustIndexData` structure and fills it with the `JustIndexData` data.

```cpp
vkCmdBindVertexBuffers(CommandBuffers[1], 0, 1, vertexDataBuffers, offsets);
vkCmdBindIndexBuffer(CommandBuffers[1], indexDataBuffer, indexOffset, indexType);
vkCmdDrawIndexed(CommandBuffers[1], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);
```

The `vkCmdBindIndexBuffer` function binds the index buffer to the command buffer, and the `vkCmdDrawIndexed` function draws the indexed vertices.
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 ) . . .  ;
```

Specialization Constants:

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

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

Specialization Constants for Compute Shaders:

```
layout( local_size_x_id = 8, local_size_y_id = 16 );
```

- This sets `gl_WorkGroupSize.x` and `gl_WorkGroupSize.y`
- `gl_WorkGroupSize.z` is set as a constant

Vulkan: Shaders’ use of Layouts for Uniform Variables

All non-sampler uniform variables must be in block buffers
Vulkan Shader Compiling

- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V, 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 . . .
**Reading a SPIR-V File into a VulkanShader Module**

```c
#define SPIRV_MAGIC 0x07230203

VkResult Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
    FILE *fp = NULL;
    if (fp == NULL) {
        fprintf(stderr, "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(stderr, "Magic number for spir-v file '%s' is 0x%08x -- should be 0x%08x

    fseek(fp, 0L, SEEK_END);
    int size = ftell(fp);
    rewind(fp);
    unsigned char *code = new unsigned char[size];
    fread(code, size, 1, fp);
    fclose(fp);

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

    VkResult result = vkCreateShaderModule(LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex);
    fprintf(stderr, "Shader Module '%s' successfully loaded

    delete[] code;
    return result;
}
```

**Reading a SPIR-V File into a Shader Module**

```c
VkShaderModule ShaderModuleVertex;

VkShaderModuleCreateInfo vsmci;

vsmci.pNext = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
vsmci.flags = 0;
vsmci.codeSize = size;

VkResult result = vkCreateShaderModule(LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex);

delete[] code;
return result;
```

---

**Vulkan: Creating a Graphics Pipeline**

- **Shader Stage Info**
- **VertexInput State**
- **InputAssembly State**
- **Tesselation State**
- **Viewport State**
- **Rasterization State**
- **MultiSample State**
- **DepthStencel State**
- **ColorState State**

---

**SPIR-V: More Information**

[SPIR-V Tools](http://github.com/KhronosGroup/SPIRV-Tools)
Vertex Buffers

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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

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

What is a Vertex Buffer?

Pasting code here...

What is a Vertex Buffer?

Mike Bailey
mjb@cs.oregonstate.edu

Data Buffers

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

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

MyBuffer MyMatrixUniformBuffer;
```

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;
```

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

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

```c
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 ) );
```

The Parade of Data

```c
MyBuffer MyMatrixUniformBuffer;
```

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

```c
uniform matBuf Matrices;
```

The Descriptor Set for the Buffer

```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 = ... = 0;
vwds0.descriptorCount = 1;
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.pBufferInfo = &vdbi0;
```

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.
Filling the Data Buffer

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

Creating and Filling the Data Buffer – the Details

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

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

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

Mike Bailey
mj@cs.oregonstate.edu

http://cs.oregonstate.edu/~mj/b/vulkan
Setting Up GLFW

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

GLFW Keyboard Callback

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

GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

```c
void GLFWMouseMotion(GLFWwindow * window, double xpos, double ypos) {
    int dx = (int)xpos - Xmouse;  // change in mouse coords
    int dy = (int)ypos - Ymouse;
    if ((ActiveButton & LEFT) != 0) {
        Xrot += (ANGFACT*dy);
        Yrot += (ANGFACT*dx);
    }
    if ((ActiveButton & MIDDLE) != 0) {
        Scale += SCLFACT * (float) (dx - dy);
        if (Scale < MINSCALE)
            Scale = MINSCALE;
    }
    Xmouse = (int)xpos;  // new current position
    Ymouse = (int)ypos;
}
```
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:
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:
modelview = glm::lookAt( glm::vec3(0.,0.,3.), glm::vec3(0.,0.,0.), glm::vec3(0.,1.,0.));
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(0.5,0.5,0.5) );

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::vec3();`

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
### Telling Visual Studio about where the GLM folder is

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

### GLM in the Vulkan sample.cpp Program

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

### 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.

Wrong!

Original object and normal

Right!

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

Instancing – What and why?

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

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

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

Making each Instance look differently -- Approach #1

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

gl_InstanceIndex starts at 0

In the vertex shader:

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

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

How We Constructed the Graphics Pipeline Structure Before

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

layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
layout( location = 4 ) in vec3 aInstanceColor;
layout ( location = 0 ) out vec3 vNormal;
layout ( location = 1 ) out vec3 vColor;
layout ( location = 2 ) out vec2 vTexCoord;

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

How We Write the Vertex Shader Now
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 int uMode;
layout( std140, binding = 7 ) uniform sampler2D uSampler;
```

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.

```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?

Our example will assume the following shader uniform variables:

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

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

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

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

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

**GPU:**
- Uniform data in a "blob"*
- Doesn't know the CPU or GPU data structure
- Knows where the data starts
- Knows the data's size

**GPU:**
- Knows the shader data structure
- Doesn't know where each piece of data starts

---

* "binary large object"*

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

struct lightBuf
{
    glm::vec4 uLightPos;
} Light;

struct miscBuf
{
    float uTime;
    int uMode;
} Misc;

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;

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

---

**Step 1: Descriptor Set Pools**

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

---

**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.

---
Step 2: Define the Descriptor Set Layouts

Matrix Set DS Layout Binding: Light Set DS Layout Binding: Misc Set DS Layout Binding: Tex Sampler Set DS Layout Binding:

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

// DS #2:
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[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;

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

VkResult

Init13DescriptorSetLayouts()

VkResult result;

MatrixSet DS Layout Binding: Light Set DS Layout Binding: Misc Set DS Layout Binding: Tex Sampler Set DS Layout Binding:

// DS #0:
MatrixSet[0].binding = 0;
MatrixSet[0].descriptorCount = 1;
MatrixSet[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
MatrixSet[0].pImmutableSamplers = (VkSampler *)nullptr;

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

// DS #2:
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[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;

VkResult

Init14GraphicsPipelineLayout() {

VkResult result;

VkPipelineLayoutCreateInfo

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = &DescriptorSetLayouts[0];
result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
return result;

}
Step 4: Allocating the Memory for Descriptor Sets

```cpp
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, &vdsai, DescriptorSets[0]);
}
```

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

```cpp
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 = (VkDescriptorBufferInfo *)&vdbi0;
vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds0.pTexelBufferView = (VkBufferView *)nullptr;
vwds0.buffer = MyMatrixUniformBuffer.buffer;
vwds0.offset = 0;
vwds0.range = sizeof(Matrices);
```

This struct links a Descriptor Set to the buffer it is pointing to.

```cpp
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 = (VkDescriptorBufferInfo *)&vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
vwds1.buffer = MyLightUniformBuffer.buffer;
vwds1.offset = 0;
vwds1.range = sizeof(Light);
```

This struct links a Descriptor Set to the buffer it is pointing to.

```cpp
VkWriteDescriptorSet vwds2;
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 = (VkDescriptorBufferInfo *)&vdbi2;
vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds2.pTexelBufferView = (VkBufferView *)nullptr;
vwds2.buffer = MyMiscUniformBuffer.buffer;
vwds2.offset = 0;
vwds2.range = sizeof(Misc);
```

This struct links a Descriptor Set to the buffer it is pointing to.

```cpp
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 = (VkDescriptorImageInfo *)&vdii0;
vwds3.pTexelBufferView = (VkBufferView *)nullptr;
vwds3.buffer = MyPuppyTexture.buffer;
vwds3.offset = 0;
vwds3.range = sizeof(MyPuppyTexture);
```

This struct links a Descriptor Set to the image it is pointing to.

```cpp
uint32_t copyCount = 0;
vkUpdateDescriptorSets(LogicalDevice, 1, &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
vkUpdateDescriptorSets(LogicalDevice, 1, &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);
```
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

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

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

```c
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
Memory Types

CPU Memory

GPU Memory

Host Visible GPU Memory

Device Local GPU Memory

GPU Memory

Texture Sampling Hardware

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: DeviceLocal
Memory 8: DeviceLocal
Memory 9: HostVisible HostCoherent
Memory 10: HostVisible HostCoherent HostCached

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

Texture Sampling Parameters

```
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 );
```

OpenGL

```
VkSamplerCreateInfo vsci;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = ... = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    ...$

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

Vulkan

Texture Mip*-mapping

Average 4 pixels to make a new one

• Total texture storage is \( \approx 2 \times \) 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 RGBAs returned.

This is known as **VK_SAMPLER_MIPMAP_MODE_LINEAR**.

* Latin: multim in parvo, “many things in a small place”
VkResult InitTextureSampler( MyTexture * pMyTexture )
{
    VkResult result;
    uint32_t texWidth = pMyTexture->width;
    uint32_t texHeight = pMyTexture->height;
    unsigned char *texture = pMyTexture->pixels;
    VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each

    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;
    #ifdef CHOICES
    // ************************************************************ *******************
    // VK_SAMPLER_ADDRESS_MODE_REPEAT
    // this first {...} is to create the staging image:
    // *******************************************************************************
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
    #endif
    vsci.mipLodBias = 0.;
    vsci.anisotropyEnable = VK_FALSE;
    vsci.compareEnable = VK_FALSE;
    vsci.compareOp = VK_COMPARE_OP_NEVER;
    #ifdef CHOICES
    VK_IMAGE_TILING_OPTIMAL
    VK_IMAGE_TILING_LINEAR
    #endif
    vsci.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vsci.queueFamilyIndexCount = 0;
    vsci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler );

    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_LINEAR;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements( LogicalDevice, IN stagingImage, OUT &vmr);
    void * gpuMemory;
    vkMapMemory( LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
    if (Verbose)
    { 
        fprintf(FpDebug, "Image vmr.size = %lld\n", vmr.size);
        fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
        if (vsl.rowPitch == 4 * texWidth)
            fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    }
    else{
        unsigned char *gpuBytes = (unsigned char *)gpuMemory;
        for (unsigned int y = 0; y < texHeight; y++)
        {
            VkMemoryAllocateInfo vmai;
            vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
            vmai.pNext = nullptr;
            vmai.allocationSize = vmr.size;
            vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();  // because we want to mmap it
            VkDeviceMemory vdm;
            result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
            pMyTexture->vdm = vdm;
            vkBindImageMemory( LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
            // we have now created the staging image -- fill it with the pixel data:
            VkSubresource vis;
            vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
            vis.mipLevel = 0;
            vis.arrayLayer = 0;
            VkSubresourceLayout vsl;
            vkGetImageSubresourceLayout( LogicalDevice, IN stagingImage, vis, OUT &vsl);
            fprintf(FpDebug, "Subresource Layout:\n");
            fprintf(FpDebug, "	offset = %lld\n", vsl.offset);
            fprintf(FpDebug, "	size = %lld\n", vsl.size);
            fprintf(FpDebug, "	rowPitch = %lld\n", vsl.rowPitch);
            fprintf(FpDebug, "	arrayPitch = %lld\n", vsl.arrayPitch);
            fprintf(FpDebug, "	depthPitch = %lld\n", vsl.depthPitch);
            if (Verbose)
                fflush(FpDebug);
        }
    }
    vkUnmapMemory( LogicalDevice, vdm);
    result = vkBindImageMemory( LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
    // we have now created the staging image -- fill it with the pixel data:
    VkSubresource vis;
    vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vis.mipLevel = 0;
    vis.arrayLayer = 0;
    VkSubresourceLayout vsl;
    vkGetImageSubresourceLayout( LogicalDevice, IN stagingImage, vis, OUT &vsl);
    fprintf(FpDebug, "Subresource Layout:\n");
    fprintf(FpDebug, "	offset = %lld\n", vsl.offset);
    fprintf(FpDebug, "	size = %lld\n", vsl.size);
    fprintf(FpDebug, "	rowPitch = %lld\n", vsl.rowPitch);
    fprintf(FpDebug, "	arrayPitch = %lld\n", vsl.arrayPitch);
    fprintf(FpDebug, "	depthPitch = %lld\n", vsl.depthPitch);
    if (Verbose)
        fflush(FpDebug);
}
// *******************************************************************************
// this second {...} is to create the actual texture image:
// *******************************************************************************
{
VkImageCreateInfo vici;
vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
vici.pNext = nullptr;
vici.flags = 0;
vici.imageType = VK_IMAGE_TYPE_2D;
vici.format = VK_FORMAT_R8G8B8A8_UNORM;
vici.extent.width = texWidth;
vici.extent.height = texHeight;
vici.extent.depth = 1;
vici.mipLevels = 1;
vici.arrayLayers = 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;
// because we are transferring into it and will eventual sample from it
vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vici.queueFamilyIndexCount = 0;
vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled
VkMemoryRequirements vmr;
vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
if( Verbose ){
fprintf( FpDebug, "Texture vmr.size = %lld\n", vmr.size);
fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment);
fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
fflush( FpDebug );
}
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
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
}
// *******************************************************************************
// copy pixels from the staging image to the texture:
VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);
// ************************************************************ *******************
// transition the staging 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 = ...
newLayout = stagingImage;
vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
vimb.dstAccessMask = 0;
vimb.subresourceRange = visr;
vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0,
0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,1, IN &vimb);
// now do the final image transfer:
VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.baseArrayLayer = 0;
visl.mipLevel = 0;
visl.layerCount = 1;
VkOffset3D                              vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;
VkExtent3D                              ve3;
ve3.width = texWidth;
ve3.height = texHeight;
ve3.depth = 1;
VkImageCopy vic;
vic.srcSubresource = visl;
vic.srcOffset = vo3;
vic.dstSubresource = visl;
vic.dstOffset = vo3;
vic.extent = ve3;
vkCmdCopyImage(TextureCommandBuffer,
stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);
// ************************************************************ *******************
// transition the texture buffer layout:// *******************************************************************************
{
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = 0;
vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
vimb.subresourceRange = visr;
vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,1, IN &vimb);
}
// now do the final image transfer:
 VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.baseArrayLayer = 0;
visl.mipLevel = 0;
visl.layerCount = 1;
VkOffset3D                              vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;
VkExtent3D                              ve3;
ve3.width = texWidth;
ve3.height = texHeight;
ve3.depth = 1;
VkImageCopy vic;
vic.srcSubresource = visl;
vic.srcOffset = vo3;
vic.dstSubresource = visl;
vic.dstOffset = vo3;
vic.extent = ve3;
vkCmdCopyImage(TextureCommandBuffer,
stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);
transition the texture buffer layout a second time:

```c
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;
vlm.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vlm.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vlm.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vlm.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vlm.srcAccessMask = 0;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vlm.subresourceRange = visr;

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

result = vkEndCommandBuffer(TextureCommandBuffer);

```c
VkSubmitInfo vsi;
vsis.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsis.pNext = nullptr;
vsis.commandBufferCount = 1;
vsis.pCommandBuffers = &TextureCommandBuffer;
vsis.waitSemaphoreCount = 0;
vsis.pWaitSemaphores = ... = 0;
vsis.pSignalSemaphores = (VkSemaphore *)nullptr;
vsis.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
```

result = vkQueueSubmit(Queue, 1, IN &vsi, VK_NULL_HANDLE);

result = vkQueueWaitIdle(Queue);

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.

```c
// 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;
vlm.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;
vlm.subresourceRange = visr;
```

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

Reading in a Texture from a BMP File

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

```c
MyTexture MyPuppyTexture;
```

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

http://cs.oregonstate.edu/~mjb/vulkan
What is the Vulkan Graphics Pipeline?

Here's what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context". It is a data structure.
2. The Vulkan Graphics Pipeline is not the processes that OpenGL would call "the graphics pipeline".
3. For the most part, the Vulkan Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipelines.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

The First Step: Create the Graphics Pipeline Layout

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

```c
VkResult Init14GraphicsPipelineLayout() {
    VkResult result;

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

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

    return result;
}
```

Vulkan: A Pipeline Records the Following Items:

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

**Bold/Italic** indicates that this state item can also be set with Dynamic Variables

Creating a Graphics Pipeline from a lot of Pieces

- Array naming the states that can be set dynamically
Creating a Typical Graphics Pipeline

```
link in the Shaders

VkPipelineShaderStageCreateInfo

VkPipelineVertexInputStateCreateInfo

VkVertexInputBindingDescription

VkVertexInputAttributeDescription

VkPipelineRasterizationStateCreateInfo

VkPipelineMultiDrawStateCreateInfo

VkPipelineGeometryShaderStageCreateInfo

 VkPipelineShaderStageCreateInfo  
  stage = VK_SHADER_STAGE_FRAGMENT_BIT;  
  pName = "main";

VkPipeline TessellationStateCreateInfo

VkPipelineColorBlendStateCreateInfo

VkPipeline OcclusionQueryStateCreateInfo

VkPipelineMultisampleStateCreateInfo

VkPipelineDepthStencilStateCreateInfo

VkPipelineDynamicStateCreateInfo

VkPipelineDescriptorSetLayoutCreateInfo

VkPipelineLayoutCreateInfo

VkPipelineCreateInfo

VkPipeline

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

vpiasci.primitiveRestartEnable = VK_FALSE;

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

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

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:

...
Declaring information about how the rasterization will take place

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

**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.

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

**Color Blending State for each Color Attachment**

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

```
VkPipelineColorBlendAttachmentState vpcbas;
vpcbas.blendEnable = VK_FALSE;
vpcbas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vpcbas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
vpcbas.colorBlendOp = VK_BLEND_OP_ADD;
vpcbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpcbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpcbas.alphaBlendOp = VK_BLEND_OP_ADD;
vpcbas.colorWriteMask =
  VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
```
Color Blending State for each Color Attachment

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

Stencil Operations for Front and Back Faces

Uses for Stencil Operations
Operations for Depth Values

VkPipelineDepthStencilStateCreateInfo
vpdssci = createInfo;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;  // never succeeds
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;

VkGraphicsPipelineCreateInfo
vgpci = createInfo;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#ifdef CHOICESVK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BITVK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BITVK_PIPELINE_CREATE_DERIVATIVE_BIT
vgpci.stageCount = 2;                           // number of stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;

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

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…)

Queue all of the individual state information and create the pipeline

Queues and Command Buffers

Mike Bailey
mj@cs.oregonstate.edu

http://cs.oregonstate.edu/~mj/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, 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++ )
{ fprintf( FpDebug, "	%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 *)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. // 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

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

Creating a Logical Device Needs to Know Queue Family Information

Creating the Command Pool as part of the Logical Device

VkResult Init06CommandPool()
{
    VkResult result;
    VkCommandPoolCreateInfo vc pci;
    vc pci .sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vc pci .pNext = nullptr;
    vc pci .flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
    #ifdef CHOICES
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
    #endif
    vc pci .queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(LogicalDevice, IN & vc pci , PALLOCATOR, OUT & CommandPool);
    return result;
}

Creating the Command Buffers

VkResult Init06CommandBuffers()
{
    VkResult result;
    // allocate 2 command buffers for the double-buffered rendering:
    {
        VkCommandBufferAllocateInfo vc bai;
        vc bai .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vc bai .pNext = nullptr;
        vc bai .commandPool = CommandPool;
        vc bai .level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vc bai .commandBufferCount = 2; // 2, because of double-buffering
        result = vkAllocateCommandBuffers(LogicalDevice, IN & vc bai , OUT & CommandBuffers[0]);
    }
    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    {
        VkCommandBufferAllocateInfo vc bai;
        vc bai .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vc bai .pNext = nullptr;
        vc bai .commandPool = CommandPool;
        vc bai .level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vc bai .commandBufferCount = 1;
        result = vkAllocateCommandBuffers(LogicalDevice, IN & vc bai , OUT & TextureCommandBuffer);
    }
    return result;
}

Beginning a Command Buffer

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);

VkCommandBufferBeginInfo vc bb i;
vc bb i .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vc bb i .pNext = nullptr;
vc bb i .flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN & vc bb i );

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

These are the Commands that could be entered into the Command Buffer, II
The Entire Submission / Wait / Display Process

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

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

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

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

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

- Front
- Back
- Double-buffered Color Framebuffers
- Update
- Refresh
- Video Driver

Vulkan Thinks of it This Way

- Front
- Back
- Depth-Buffer

What is a Swap Chain?

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

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

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

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

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

What is a Swap Chain?

Because it has the word “chain” in it, let’s try to visualize the Swap Chain as a physical chain.

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

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

In the same way that bicycle chain links are “re-used”, Swap Chain images get re-used too.
What is a Swap Chain?

- **Being Drawn Into**
- **Being Presented**

We Need to Find Out What our Display Capabilities Are

```c
VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
prevExtent = vsc.currentExtent;
fprintf( FpDebug, "vkGetPhysicalDeviceSurfaceCapabilitiesKHR:
" );

VkBool32 supported;
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics( ), Surface, &supported );
if( supported == VK_TRUE )
fprintf( FpDebug, "** This Surface is supported by the Graphics Queue **
" );

uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
vSurfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );
fprintf( FpDebug, "Found %d Surface Formats:
" );

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "Found %d Present Modes:
" );
```

We Need to Find Out What our Display Capabilities Are

- **minImageCount = 2**
- **maxImageCount = 8**
- **currentExtent = 1024 x 1024**
- **maxImageExtent = 1024 x 1024**
- **maxImageArrayLayers = 1**
- **supportedTransforms = 0x0001**
- **supportedCompositeAlpha = 0x0001**
- **supportedUsageFlags = 0x009f**

**This Surface is supported by the Graphics Queue**

- Found 2 Surface Formats:
  - 0: BGR8UNORM
  - 1: BGR8SRGB

- Found 3 Present Modes:
  - 0: FIFO
  - 1: FIFO RELAXED
  - 2: MAILBOX

Creating a Swap Chain

```c
vkCreateSwapchain( );
VkSwapchainCreateInfo
surface
imageFormat
imageColorSpace
imageExtent
imageArrayLayers
imageUsage
imageSharingMode
preTransform
compositeAlpha
presentMode
clipped
```
Creating a Swap Chain

VkSurfaceCapabilitiesKHR
vkGetPhysicalDeviceSurfaceCapabilitiesKHR
VkSwapchainCreateInfoKHR
vkCreateSwapchainKHR

Creating the Swap Chain Images and Image Views

VkSemaphoreCreateInfo
vkCreateSemaphore

Rendering into the Swap Chain, I

VkFenceCreateInfo
vkCreateFence

Rendering into the Swap Chain, II
result = vkWaitForFences( LogicalDevice, 1, &renderFence, VK_TRUE, UINT64_MAX );

VkPresentInfoKHR
  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 );

Application

Instance

Physical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Vulkan: a More Typical (and Simplified) Block Diagram

uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );
Vulkan: Identifying the Physical Devices

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

Which Physical Device to Use, I

Which Physical Device to Use, II

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; ++i )
{
    VkPhysicalDeviceProperties vdp;
    vkGetPhysicalDeviceProperties( physicalDevices[i], OUT &vdp );
    if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        discreteSelect = i;
    if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        integratedSelect = i;
    if( vdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
        fprintf( FpDebug, "Could not select a Physical Device
" );
    return VK_SHOULD_EXIT;
}

Asking About the Physical Device’s Features

VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( PhysicalDevice, OUT &PhysicalDeviceFeatures );
if( PhysicalDeviceFeatures.geometryShader == VK_TRUE )
    fprintf( FpDebug, "Device Supports Geometry Shader
" );
else if( PhysicalDeviceFeatures.tessellationShader == VK_TRUE )
    fprintf( FpDebug, "Device Supports Tessellation Shader
" );
else if( PhysicalDeviceFeatures.multiDrawIndirect == VK_TRUE )
    fprintf( FpDebug, "Device Supports Multi Draw Indirect
" );
else if( PhysicalDeviceFeatures.wideLines == VK_TRUE )
    fprintf( FpDebug, "Device Supports Wide Lines
" );
else if( PhysicalDeviceFeatures.largePoints == VK_TRUE )
    fprintf( FpDebug, "Device Supports Large Points
" );
else if( PhysicalDeviceFeatures.multiViewport == VK_TRUE )
    fprintf( FpDebug, "Device Supports MultiViewport
" );
else if( PhysicalDeviceFeatures.occlusionQueryPrecise == VK_TRUE )
    fprintf( FpDebug, "Device Supports Occlusion Query Precise
" );
else if( PhysicalDeviceFeatures.pipelineStatisticsQuery == VK_TRUE )
    fprintf( FpDebug, "Device Supports Pipeline Statistics Query
" );
else if( PhysicalDeviceFeatures.shaderFloat64 == VK_TRUE )
    fprintf( FpDebug, "Device Supports Shader Float 64
" );
else if( PhysicalDeviceFeatures.shaderInt64 == VK_TRUE )
    fprintf( FpDebug, "Device Supports Shader Int 64
" );
else if( PhysicalDeviceFeatures.shaderInt16 == VK_TRUE )
    fprintf( FpDebug, "Device Supports Shader Int 16
" );
Here's what the NVIDIA RTX 2080 Ti produced

```
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:
gameShader = 1
tessellationShader = 1
multiDrawIndirect = 1
wideLines = 1
largePoints = 1
multiViewport = 1
occlusionQueryPrecise = 1
pipelineStatisticsQuery = 1
shaderFloat64 = 1
shaderInt64 = 1
shaderInt16 = 1
```

Here's what the Intel HD Graphics 520 produced

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

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

Asking about the physical device's different memories

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

Here's what I got

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

2 Memory Heaps:
  Heap 0: size = 0x00000000 DeviceLocal
  Heap 1: size = 0x00000000
```
Asking About the Physical Device’s Queue Families

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

Here’s What I Got

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

Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu

Vulkan. Logical Devices

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

Vulkan: a More Typical (and Simplified) Block Diagram
Looking to See What Device Layers are Available

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

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

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

Looking to See What Device Extensions are Available

```c
// 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);
```

What Device Layers and Extensions are Available

```c
4 physical device layers enumerated:
0x00401063  1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
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'
```

Vulkan: Creating a Logical Device

```c
float queuePriorities[1] = { 1.);

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

```c
VkDeviceCreateInfo vdci;
vci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdi.pNext = nullptr;
vdi.flags = 0;
vdi.queueCreateInfoCount = 1;
vdi.pQueueCreateInfos = &vdicq;
vdi.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdi.ppEnabledLayerNames = myDeviceLayers;
vdi.enabledExtensionCount = 0;
vdi.ppEnabledExtensionNames = (const char **)nullptr;
vdci.pEnabledFeatures = &PhysicalDeviceFeatures;
result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice);
```
Creating the Logical Device’s Queue

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

Dynamic State Variables

Mike Bailey
mj@cs.oregonstate.edu

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

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
{
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH,
};

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

VkGraphicsPipelineCreateInfo
vgpci
{
    pDynamicState = &vpdsci;
};

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

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

Filling the Dynamic State Variables in the Command Buffer

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

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

Push Constants

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

By “small”, Vulkan specifies that these must be at least 128 bytes in size, although they can be larger. For example, the maximum size is 256 bytes on the NVIDIA 1080ti. (You can query this limit by looking at the maxPushConstantSize parameter in the VkPhysicalDeviceLimits structure.) Unlike uniform buffers and vertex buffers, these are not backed by memory. They are actually part of the Vulkan pipeline.
On the shader side, if, for example, you are sending a 4x4 matrix, the use of push constants in the shader looks like this:

```
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:

```
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:

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

VkPipelineLayoutCreateInfo vplci;
vec13.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, PALLOCATOR, OUT &GraphicsPipelineLayout );
```

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

Where each arm is represented by:

```c
struct arm
{
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale; // scale factor in x
};
struct armArm1;
struct armArm2;
struct armArm3;
```
In the Reset Function

```c
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;
```

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

Setup the Push Constant for the Pipeline Structure

```c
VkPushConstantRange vpcr[1];

vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

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

In the UpdateScene Function

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

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

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

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

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

Arm1.armMatrix = m1g; // m1g

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

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

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

Arm2.armMatrix = m1g * m21; // m2g

m21 = glm::translate(m21, glm::vec3(2.*Arm2.armScale, 0., 0.));

m21 = glm::rotate(m21, rot3, zaxis);

m21 = glm::translate(m21, glm::vec3(0., 0., 2.));

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

In the RenderScene Function
In the Vertex Shader

```cpp
layout( push_constant ) uniform arm {
    mat4  armMatrix;  // scale factor in x
    vec3  armColor;   
    float armScale;   
} 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 *= (RobotArm.armMatrix * vec4(bVertex, 1.)));  

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

**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.

```cpp
VkQueryPoolCreateInfo vqpci;
vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
vqpci.pNext = nullptr;
vqpci.flags = 0;
vqpci.queryType = << one of: >>
VK_QUERY_TYPE_OCCLUSION
VK_QUERY_TYPE_PIPELINE_STATISTICS
VK_QUERY_TYPE_TIMESTAMP
vqpci.queryCount = 1;
vqpci.pipelineStatistics = 0;  // bitmask of what stats you are querying for if you are doing a pipeline statistics query
vqpci.flags = 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 );
```
Resetting, Filling, and Examining a Query Pool

```
vkCmdResetQueryPool( CommandBuffer, occlusionQueryPool, 0, 1 );
vkCmdBeginQuery( CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT );
...  
vkCmdEndQuery( CommandBuffer, occlusionQueryPool, 0 );
```

defines

```
#define DATASIZE 128
uint32_t data[DATASIZE];
result = vkGetQueryPoolResults( LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags );
```

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, NUM_STATS*sizeof(uint32_t), counts, 0, VK_QUERY_RESULT_WAIT_BIT );
```

```
#define vqpci.pipelineStatistics 0x1L |
| VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT |
| VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT |
| VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_PATCHES_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_TESSELATION_CONTROL_SHADER_PATCHES_BIT |
| VK_QUERY_PIPELINE_STATISTIC_TESSELATION_EVALUATION_SHADER_INVOCATIONS_BIT |
| VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT |
```

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

```
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

### Compute Pipeline

Here is how you create a Compute Pipeline

- Start by Creating the Data Buffers
  - 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.

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

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

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

--

Mike Bailey

mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Creating a Shader Storage Buffer

VkBuffer Buffer;

VkBufferCreateInfo vbc;
  vbc.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  vbc.pNext = nullptr;
  vbc.flags = 0;
  vbc.size = buffer size in bytes;
  vbc.usage = VK_USAGE_STORAGE_BUFFER_BIT;
  vbc.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
  vbc.queueFamilyIndexCount = 0;
  vbc.pQueueFamilyIndices = (const int32_t *) nullptr;

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

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

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

Create the Compute Pipeline Layout

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

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

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

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

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

Create the Compute Pipeline

This is the number of work-items per work-group, set in the compute shader.
The number of work-groups is set in the vkCmdDispatch() function call in the C/C++ program.

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. );

uint gid = gl_GlobalInvocationID.x;  // the .y and .z are both 1 in this case
POINT p = Positions[ gid ].xyz;
VELOCITY v = Velocities[ gid ].xyz;

POINT pp = p + v * DT + 0.5 * DT * DT * G;
VELOCITY vp = v + G * DT;

Positions[ gid ].xyz = pp;
Velocities[ gid ].xyz = vp;
```

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

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

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?

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

Graphics Trick Alert: Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough (and it is), nobody will ever know...

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

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

Specialization Constants

Mike Bailey
mjb@cs.oregonstate.edu

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

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 shader stage
vsme[0].constantID = 7; // Specialization Constant
vsme[0].offset = 0; // # bytes into the Specialization Constant
vsme[0].size = sizeof(asize); // size of this one item is

VkSpecializationInfo vsi;
vsixpMapEntryCount = 1;
vsixpDataSize = sizeof(asize); // size of all the Specialization Constants together
vsixpData = &asize; // array of all the Specialization Constants

VkPipelineShaderStageCreateInfo vpssci; // Vulkan shader stage create info
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]; // Vulkan compute pipeline create info
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 );
```

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.

Linking the Specialization Constants into the Compute Pipeline
Specialization Constant Example – Setting Multiple Constants

In the compute shader:

```cpp
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:

```cpp
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;
vsi.pMapEntries = &vsme[0];
```

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

In the compute shader:

```cpp
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:

```cpp
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];
```

Where Synchronization Fits in the Overall Block Diagram

Synchronization

Mike Bailey
mj@cs.oregonstate.edu

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

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

Creating a Semaphore

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

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

Semaphores Example during the Render Loop

```cpp
VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

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

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

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

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

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
Fences

```c
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT; // = 0
// VK_FENCE_CREATE_SIGNALED_BIT is only other option
VkFence fence;
result = vkCreateFence( LogicalDevice, IN &vfci, PALLOCATOR, OUT &fence);
// returns right away:
result = vkGetFenceStatus( LogicalDevice, IN fence);
// result = VK_SUCCESS means it has signaled
// result = VK_NOT_READY means it has not signaled

// blocks:
result = vkWaitForFences( LogicalDevice, 1, IN &fence, waitForAll, timeout);
// waitForAll = VK_TRUE:   wait for all fences in the list
// waitForAll = VK_FALSE: wait for any one fence in the list
// timeout is a uint64_t timeout in nanoseconds  (could be 0, which means to return immediately)
// timeout can be up to UINT64_MAX = 0x ffffffffffffffff (= 580+ years)
// result = VK_SUCCESS means it returned because a fence (or all fences) signaled
// result = VK_TIMEOUT means it returned because the timeout was exceeded
```

Events

- Events provide even finer-grained synchronization
- Events are a primitive that can be signaled by the host or the device
- Can even signal at one place in the pipeline and wait for it at another place in the pipeline
- Signaling in the pipeline means "signal 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

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

Note: the 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, C1, _event);

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

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

Pipeline Barriers

Mike Bailey
mjb@cs.oregonstate.edu

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

vkCmdPipelineBarrier( ) Function Call

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

vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);

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?

<table>
<thead>
<tr>
<th>Stage Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
</tr>
</tbody>
</table>

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

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

### Pipeline Stages

- **Vertex Shader**
- **Primitives Assembly**
- **Tessellation Control Shader**
- **Tessellation Primitive Generator**
- **Tessellation Evaluation Shader**
- **Geometry Shader**
- **Primitives Assembly**
- **Rasterizer**
- **Fragment Shader**

### Pipeline Stages and what Access Operations can Happen There

<table>
<thead>
<tr>
<th>Stage Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
</tr>
</tbody>
</table>
Access Operations and what Pipeline Stages they can be used in:

<table>
<thead>
<tr>
<th>Access Operations</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
</tr>
</tbody>
</table>

The Scenario:

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

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

- **src** stages:
  - 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_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

- **dst** stages:
  - VK_PIPELINE_STAGE_HOST_BIT
  - VK_PIPELINE_STAGE_TRANSFER_BIT
  - VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
  - VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
  - VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
  - VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
  - VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
  - VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
  - VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
  - VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
  - VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
  - VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
  - VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
  - VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT

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

- **src** stages:
  - 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_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

- **dst** stages:
  - VK_PIPELINE_STAGE_HOST_BIT
  - VK_PIPELINE_STAGE_TRANSFER_BIT
  - VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
  - VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
  - VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
  - VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
  - VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
  - VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
  - VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
  - VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
  - VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
  - VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
  - VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
  - VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT

Access types:

- **src** access types (no access setting needed)
- **dst** access types (no access setting needed)
The Scenario

src cars
dst cars

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

<table>
<thead>
<tr>
<th>VkImageLayout</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
<td>Used as a color attachment</td>
</tr>
<tr>
<td>VK_IMAGE_LAYOUT_GENERAL</td>
<td>Read into a shader as a texture</td>
</tr>
<tr>
<td>VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL</td>
<td>Copy from</td>
</tr>
<tr>
<td>VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL</td>
<td>Copy to</td>
</tr>
<tr>
<td>VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL</td>
<td>Show image to viewer</td>
</tr>
<tr>
<td>VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL</td>
<td></td>
</tr>
</tbody>
</table>

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

Aliasing

The Display We Want

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

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

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

What the signal really is: what we want

**Sampling Interval**

**Sampled Points**

---

**MultiSampling**

Multisampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there.

There are two approaches to this:

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

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

---

**Vulkan Distribution of Sampling Points within a Pixel**
Consider Two Triangles Whose Edges Pass Through the Same Pixel

Vulkan Distribution of Sampling Points within a Pixel

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.125, 0.125)</td>
<td>(0.375, 0.375)</td>
<td>(0.625, 0.625)</td>
<td>(0.875, 0.875)</td>
</tr>
<tr>
<td>(0.25, 0.25)</td>
<td>(0.4375, 0.4375)</td>
<td>(0.6875, 0.6875)</td>
<td>(0.9375, 0.9375)</td>
</tr>
<tr>
<td>(0.125, 0.25)</td>
<td>(0.375, 0.625)</td>
<td>(0.625, 0.875)</td>
<td>(0.875, 1.0)</td>
</tr>
<tr>
<td>(0.25, 0.75)</td>
<td>(0.4375, 0.875)</td>
<td>(0.6875, 1.0)</td>
<td></td>
</tr>
</tbody>
</table>

Supersampling

Final Pixel Color = \[
\frac{\sum \text{Color sample from subpixel}}{8}
\]

# Fragment Shader calls = 8

Multisampling

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

# Fragment Shader calls = 2
VkPipelineMultisampleStateCreateInfo vpmsci = 
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmisc.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vpmisc.sampleShadingEnable = VK_TRUE;
vpmisc.minSampleShading = 0.5f;
vpmisc.pSampleMask = (VkSampleMask *)nullptr;
vpmisc.alphaToCoverageEnable = VK_FALSE;
vpmisc.alphaToOneEnable = VK_FALSE;

VkGraphicsPipelineCreateInfo vpci = 
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vpci.pNext = nullptr;

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

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

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

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

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

VkSubpassDescription vsd;
vsd.flags = 0;
vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd.inputAttachmentCount = 0;
vsd.pInputAttachments = ...
vsd.preserveAttachmentCount = 0;
vsd.pPreserveAttachments = (uint32_t *)nullptr;

VkRenderPassCreateInfo vrpci = 
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.attachmentCount = 2; // color and depth/stencil
vrpci.pAttachments = vad;
vrpci.pSubpasses = IN &vsd;
vrpci.dependencyCount = 0;
vrpci.pDependencies = (...)nullptr;

result = 
vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
Resolving the Image:
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

```cpp
VOffset3D
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;

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

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

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

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

Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Computer Graphics API

Thanks for coming today!

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