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.
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 deprecate in OpenGL are really deprecate in Vulkan: built-in pipeline transformations, begin-end, fixed-function, etc.
- You must manage your own transformations.
- All transformation, color, texture functionality must be done in shaders.
- Shaders are pre-“half-compiled” outside of your application. The compilation process is then finished during the pipeline-building process.

The Basic Computer Graphics Pipeline, Shader-style

- **Model transform**
- **View transform**
- **Projection transform**
- **Fragment processing**
- **Per-fragment lighting**
- **Per-vertex lighting**
- **Rasterization**

- **Uniform Variables**
  - gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix

- **Per-vertex in variables**
  - gl_Vertex, gl_Normal, gl_Color

- **Per-vertex out variables**
  - gl_Position

Vulkan Shaders

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

Moving part of the driver into the application

- Complex drivers lead to driver overhead and cross vendor unpredictability.
- Error management is always active.
- Driver processes full shading language source.
- Separate APIs for desktop and mobile markets.
- External GLSL Compiler.
- SPIR-V.
- Vendor-specific code.
- Suitable drivers for low-overhead efficiency and cross vendor portability.
- Layered architecture as 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!


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 Output

Sample Program Keyboard Inputs

- Y, T: Toggle using a vertex buffer only vs. an index buffer
- T, 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
- Y, 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 typedefed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

```cpp
int main( int argc, char * argv[] )
{
   Width  = 800;
   Height = 600;
   errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
   if( err != 0 )
   {
      fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE );
      FpDebug = stderr;
   }
   fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
   Reset( );
   InitGraphics( );
   // loop until the user closes the window:
   while( glfwWindowShouldClose( MainWindow ) == 0 )
   {
      glfwPollEvents( );
      Time = glfwGetTime( );          // elapsed time, in double-precision seconds
      UpdateScene( );
      RenderScene( );
   }
   fprintf(FpDebug, "Closing the GLFW window
");
   vkQueueWaitIdle( Queue );
   vkDeviceWaitIdle( LogicalDevice );
   DestroyAllVulkan( );
   glfwDestroyWindow( MainWindow );
   glfwTerminate( );
   return 0;
}
```
#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. },  // position
        {  0.,  0., -1. },  // normal
        {  0.,  0.,  0. },  // color
        {  1., 0. }         // texCoord
    },
    // vertex #2:
    {
        { -1.,  1., -1. },  // position
        {  0.,  0., -1. },  // normal
        {  0.,  1.,  0. },  // color
        {  1., 1. }         // texCoord
    },
    // vertex #3:
    {
        {  1.,  1., -1. },  // position
        {  0.,  0., -1. },  // normal
        {  1.,  1.,  0. },  // color
        {  0., 1. }         // texCoord
    },
    // ...  
};

A Colored Cube

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++:
   ```cpp
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex, OUT &Queue );
   ```
   we would actually say in C:
   ```c
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```

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

Vulkan Conventions

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

My Conventions

“Init” in a function call name means that something is being setup that only needs to be setup once
The number after “Init” gives you the ordering
In the source code, after main( ) comes InitGraphics(), then all of the InitxxYYY( ) functions in numerical order. After that comes the helper functions

“Find” in a function call name means that something is being looked for
“Fill” in a function call name means that some data is being supplied to Vulkan

“IN” and “OUT” ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are each actually #define’d to nothing.

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):

```
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *) nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT &physicalDevices[0] );
```
Mike Bailey  
mjb@cs.oregonstate.edu

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

Topology:  
How things are connected

Original Object

Geometry vs. Topology

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

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

Vulkan Topologies

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

Vertex Orientation Issues

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

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

ProjectionMatrix\[1\][1] *= -1.;
This is like saying "Y' = -Y".

X
Y
Z

0 1
2
3
CCW

X
Y
Z

3 2
1
0
CW

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

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

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

VkPipelineRasterizationStateCreateInfo vprsci;

vkPipelineRasterizationStateCreateInfo

vertexOrientation = VK_CULL_MODE_NONE;

frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
A data structure that holds (what OpenGL would call) the input data for our application.

The Vulkan Pipeline is essentially a very large shader program that tells the GPU what to do.

Telling the Pipeline about its Input

The Vulkan Pipeline has a few components that we need to understand:

1. **Input Data**: The data that is passed to the pipeline.
2. **Vertex Buffer**: The data that is processed by the pipeline.
3. **Shader Program**: The code that processes the vertex data.
4. **Render Pass**: The process of rendering the vertex data.

### Filling the Vertex Buffer

Here is an example of how to fill the vertex buffer:

```c
struct vertex

void Init05MyVertexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)

VkResult Fill05DataBuffer(MyVertexDataBuffer, (void *) VertexData);

Init05MyVertexDataBuffer(sizeof(VertexData), &MyVertexDataBuffer);

MyBuffer MyVertexDataBuffer;

result = Fill05DataBuffer(MyVertexDataBuffer, (void *) VertexData);
```

### GLSL Shader

Here is an example of a GLSL shader:

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

### Non-indexed Buffer Drawing

Here is an example of how to draw triangles from a non-indexed buffer:

```c
void Draw(VkBuffer v buffer, IN vdm, 0); // 0 is the offset
```

### A Preview of What Init05DataBuffer Does

Here is an example of how to allocate memory for the vertex buffer:

```c
VkDeviceSize size = 0;
void result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
```

### Triangles Represented as an Array of Structures

Here is an example of how to represent triangles as an array of structures:

```c
struct vertex

void Init05DataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)

VkResult Fill05DataBuffer(MyVertexDataBuffer, (void *) VertexData);

Init05MyVertexDataBuffer(sizeof(VertexData), &MyVertexDataBuffer);

MyBuffer MyVertexDataBuffer;

result = Fill05DataBuffer(MyVertexDataBuffer, (void *) VertexData);
```
We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

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

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

```
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 );
```

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` and `gl_InstanceIndex`
    - Both are 0-based

- WebGL Color:
  - 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:

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

Descriptor Sets:

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

Specialization Constants:

```glsl
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:

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

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

Vulkan: Shaders’ use of Layouts for Uniform Variables

- All non-sampler uniform variables must be in block buffers

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

- Non-sampler 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

- 

How do you know if SPIR-V compiled successfully?

Same as C/C++ – the compiler gives you no nasty messages.
Also, if you care, legal .spv files have a magic number of 0x07230203
So, if you do an `od -x` on the .spv file, the magic number looks like this:

```
0203 0723 . . .
```

Reading a SPIR-V File into a Vulkan Shader Module

```
VkShaderModule ShaderModuleVertex;

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);
```

Reading a SPIR-V File into a Shader Module

```
VkShaderModuleCreateInfo vsmc;
   vsmc.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
   vsmc.pNext = nullptr;
   vsmc.flags = 0;
   vsmc.codeSize = size;
   vsmc.pCode = (uint32_t*)"code";
   VkShaderModule ShaderModule = vkCreateShaderModule(LogicalDevice, &vsmc, PALLOCATOR);
   return result;
```
Vulkan: Creating a Graphics Pipeline

Vertex Buffers

Mike Bailey
mjb@cs.oregonstate.edu

Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

Filling the Vertex Buffer

MyBuffer MyVertexBuffer;

VkResult Init05MyVertexBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}

SPIR-V: More Information

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

What is a Vertex Buffer?

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

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

First, a quick review of computer graphics geometry...
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or'ed bits of: >>
VK_USAGE_TRANSFER_SRC_BIT
VK_USAGE_TRANSFER_DST_BIT
VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
VK_USAGE_UNIFORM_BUFFER_BIT
VK_USAGE_STORAGE_BUFFER_BIT
VK_USAGE_INDEX_BUFFER_BIT
VK_USAGE_VERTEX_BUFFER_BIT
VK_USAGE_INDIRECT_BUFFER_BIT
vbci.sharingMode = << one of: >>
VK_SHARING_MODE_EXCLUSIVE
VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const int32_t) nullptr;

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

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

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

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

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

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

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;
}

11 Memory Types:
Memory  0:  DeviceLocal
Memory  1:  DeviceLocal
Memory  2:  HostVisible HostCoherent
Memory  3:  HostVisible HostCoherent HostCached
Memory  4:  
Memory  5:  
Memory  6:  
Memory  7:  DeviceLocal
Memory  8:  
Memory  9:  HostVisible HostCoherent
Memory 10:  HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0:  size = 0x67f000000 DeviceLocal
Heap 1:  size = 0x52a000000
typedef struct MyBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;

MyBuffer MyMatrixUniformBuffer;

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

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

Here’s a C struct to hold some uniform variables

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

Here’s the shader code to access those uniform variables

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

Filling those Uniform Variables

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

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

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

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

The Descriptor Set for the Buffer

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

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 = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, &vbci, PALLOCATOR, &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, pMyBuffer->buffer, &vmr );
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, &vmai, PALLOCATOR, &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );
    // 0 is the offset
    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, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &pGpuMemory );
    // 0 and 0 are offset and flags
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, myBuffer.vdm );
    return VK_SUCCESS;
}

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

Setting Up GLFW

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

GLFW Keyboard Callback

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 );
                break;
        }
    }
}
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 )
    {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;
        Ymouse = (int)ypos;
        ActiveButton |= b;              // set the proper bit
    }
    else
    {
        ActiveButton &= ~b;             // clear the proper bit
    }
}
```

GLFW Mouse Motion Callback

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

Looping and Closing GLFW

```c
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );
    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:

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

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

Why are we even talking about this?

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

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

you would now have to say:

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

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

### Constructors
- `glm::mat4();` // identity matrix
- `glm::vec4();`
- `glm::vec3();`

### 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.

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

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

2. A period, indicating that the project folder should also be searched when a `#include` is encountered. If you put it somewhere else, enter that full or relative path instead.
GLM in the Vulkan sample.cpp Program

if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4();  // identity
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale, Scale, Scale) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0.) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0.) );
// done this way, the Yrot is applied first, then the Xrot, then the Scale
}
else
{
    if( ! Paused )
    {
        const glm::vec3 axis = glm::vec3( 0., 1., 0. );
        Matrices.uModelMatrix = glm::rotate( glm::mat4(), (float)glm::.radians( 360.f*Time/SECONDS_PER_CYCLE ),   axis );
    }
}
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1. // Vulkan's projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix );
Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );

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.

Instancing.pptx

Mike Bailey
mjb@cs.oregonstate.edu

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

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, CommandBuffer[activeImageIndex], 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_Instance Index to define a unique display property, such as position or color.

gl_InstanceIndex starts at 0

In the vertex shader:

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

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

In the vertex shader:

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

In the vertex shader:

```mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );```

In OpenGL:

```uniform sampler2D uSampler;```
**What are Descriptor Sets?**

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

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

**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

MatrixSet DS Layout Binding: LightSet DS Layout Binding: MiscSet DS Layout Binding: TexSamplerSet DS Layout Binding:

set = 0 set = 1 set = 2 set = 3

vdslc0 DS Layout CI:

pipeline stage(s) = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT:

descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER:

vdslc1 DS Layout CI:

pipeline stage(s) = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT:

descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER:

vdslc2 DS Layout CI:

pipeline stage(s) = VK_SHADER_STAGE_FRAGMENT_BIT:

descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER:

vdslc3.pBindings = vdslc3.flags = 0;

vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

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

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

vplci.flags = 0;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.setLayoutCount = 4;

vplci.pLayouts = &LightSet[0];

vplci.pSetLayouts = &MatrixSet[0];

vplci.pNext = nullptr;

vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

vplci.pushConstantRangeCount = 0;

vplci.pDescriptorSets = (VkDescriptorSet *)nullptr;

vplci.pImmutableSamplers = (VkSampler *)nullptr;

Step 4: Allocating the Memory for Descriptor Sets

VulkanResult ret = VkDeviceAllocateMemory( LogicalDevice, IN &vdii, OUT &descriptorSetMemory );

VulkanResult ret = VkDeviceAllocateMemory( LogicalDevice, IN &vdbi0, OUT &descriptorSetMemory[0] );

VulkanResult ret = VkDeviceAllocateMemory( LogicalDevice, IN &vdbi1, OUT &descriptorSetMemory[1] );

VulkanResult ret = VkDeviceAllocateMemory( LogicalDevice, IN &vdbi2, OUT &descriptorSetMemory[2] );

VulkanResult ret = VkDeviceAllocateMemory( LogicalDevice, IN &vdbi3, OUT &descriptorSetMemory[3] );

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

VulkanResult ret = VkDeviceSetDescriptorTable( LogicalDevice, IN descriptorSet, OUT &descriptorSetMemory[0], OUT arrayBuffer );

VulkanResult ret = VkDeviceSetDescriptorTable( LogicalDevice, IN descriptorSet, OUT &descriptorSetMemory[1], OUT arrayBuffer );

VulkanResult ret = VkDeviceSetDescriptorTable( LogicalDevice, IN descriptorSet, OUT &descriptorSetMemory[2], OUT arrayBuffer );

VulkanResult ret = VkDeviceSetDescriptorTable( LogicalDevice, IN descriptorSet, OUT &descriptorSetMemory[3], OUT arrayBuffer );
Step 6: Include the Descriptor Set Layout when Creating a Graphics Pipeline

```c
vkGraphicsPipelineCreateInfo vgpci = { 0};
vgpci.pStages = vpssci;
vgpci.flags = 0;
vgpci.pNext = nullptr;
vgpci.basePipelineIndex = 0;
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.subpass = 0;                              // subpass number
vgpci.renderPass = IN RenderPass;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pVertexInputState = &vpvisci;
vgpci.layout = IN
vgpci.pDynamicState = &vpdsci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pViewportState = &vpvsci;
```

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

```c
vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], IN &GraphicsPipeline vgpci, mjb – September 11, 2019
```

Step 5: Tell the Descriptor Sets where their data is

```c
DriverDescriptorSet vdii0 = { 0};
vdii0.pNext = nullptr;
```

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

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

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

```c
VkSamplerCreateInfo vsci;

vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
vsci.minLod = 0.;

vsci.compareOp = VK_COMPARE_OP_NEVER;

vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;

vsci.minFilter = VK_FILTER_LINEAR;

vsci.magFilter = VK_FILTER_LINEAR;

vsci.pNext = nullptr;

vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
```

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

```
// VK_FALSE means we are using the usual 0. - 1.
```
This function can be found in the result = Init06TextureSampler( &MyPuppyTexture.texSampler );
tool such as ImageMagick's writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a

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

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.

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.

The Graphics Pipeline Layout is fairly static. Only the layout of the Descriptor Sets and information on the Push Constants need to be supplied.
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, h, w, minDepth, maxDepth
- Scissors: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Blending: blendEnable, blendEnable, srcAlphaBlendFactor, dstAlphaBlendFactor, 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 Typical Graphics Pipeline

Creating a Graphics Pipeline from a lot of Pieces

Creating a Graphics Pipeline from a lot of Pieces

Link in the Shaders

Link in the Per-Vertex Attributes

Declares the binding and attribute descriptions

Tessellation Shader info

Geometry Shader info

Use core upload array member per shader module you are using

Use core upload array member per vertex input array-of-structures you are using

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

```
Triangle Strip #0:

Triangle Strip #1:

Triangle Strip #2:
```

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

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

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

What is "Depth Clamp Enable"?

```
vprsci.depthClampEnable = VK_FALSE;
```

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:

```
Original Image

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

```
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

```
Triangle Strip #0:

Triangle Strip #1:

Triangle Strip #2:
```

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

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

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

What is "Depth Clamp Enable"?

```
vprsci.depthClampEnable = VK_FALSE;
```

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:

```
Original Image

Scissoring:
Scissoring operates on fragments
and takes place right after the
rasterizer. Changing the vertical
part of the scissor causes the
entire scene to get clipped where it
falls outside the scissor area.
```
Depth Bias Enable allows scaling and translation of the Z-depth values as they come through the rasterizer to avoid Z-fighting.

**Color Blending State for each Color Attachment**

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

**Stencil Operations for Front and Back Faces**

Polygon edges without Z-fighting

**Uses for Stencil Operations**

Magic Lenses
Operations for Depth Values

```
VkPipelineDepthStencilStateCreateInfo vpdssci;
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;
// VK_COMPARE_OP_NEVER -- never succeeds
// VK_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
// VK_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
// VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
// VK_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
// VK_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
// VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
// VK_COMPARE_OP_ALWAYS -- always succeeds
#endif
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;
```

Putting it all Together! (finally…)

```
VkPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif
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;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline );
return result;
```
Querying what Queue Families are Available

```c
for( unsigned int i = 0; i < count; i++ )
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, IN &count, OUT vqfp );

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

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

```c
FindQueueFamilyThatDoesGraphics()
```

Creating a Logical Device Needs to Know Queue Family Information

```c
result = vkGetDeviceQueue( PhysicalDevice, IN &queueFamilyIndex, OUT &Queue );
```

Creating the Command Pool as part of the Logical Device

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

Creating the Command Buffers

```c
vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT CommandBuffers[0] );
```

Beginning a Command Buffer

```c
vkBeginCommandBuffer( CommandBuffers[0], IN &vcbbi );
```
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.

The Swap Chain

Mike Bailey
mjbailey@cs.oregonstate.edu

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

How We Think of OpenGL Framebuffers

Vulkan Thinks of it This Way

What is a Swap Chain?

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

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

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")
VulkanDebug.txt output:

Found 3 Present Modes:
- VK_PRESENT_MODE_MAILBOX_KHR
- VK_PRESENT_MODE_FIFO_RELAXED_KHR
- VK_PRESENT_MODE_FIFO_KHR

Found 2 Surface Formats:
1: VK_FORMAT_B8G8R8A8_SRGB, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
0: VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR

We Need to Find Out What our Display Capabilities Are
- vkGetPhysicalDeviceSurfaceSupportKHR
  - vkGetPhysicalDeviceSurfaceCapabilitiesKHR
  - VkSurfaceCapabilitiesKHR

Creating a Swap Chain
- VkSwapchainCreateInfoKHR
- vkCreateSwapchainKHR

Creating the Swap Chain Images and Image Views
- VkSwapchainKHR
- VkImageView

We Need to Find Out What our Display Capabilities Are
- VulkanDebug.txt output:
- VkSurfaceFormatKHR
- VkPresentModeKHR
- VkPhysicalDeviceSurfaceCapabilitiesKHR
- VkExtent2D
vkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

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

uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

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

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
VK_RESULT_vkEnumeratePhysicalDevices = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices );

int which = -1;

vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures);

fprintf( FpDebug, "Could not get the physical device properties of device %d\n", i );
return VK_SHOULD_EXIT;
Asking About the Physical Device’s Different Memories

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

Here’s What I Got

11 Memory Types:
Memory 0: DeviceLocal
Memory 1: DeviceLocal
Memory 2: HostVisible HostCoherent
Memory 3: HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0: size = 0x30000000 DeviceLocal
Heap 1: size = 0x60000000

Asking About the Physical Device’s Queue Families

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

Here’s What I Got

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

Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu

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

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

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

Looking to See What Device Layers are Available

// 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

4 physical device layers enumerated:
0x00401063   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00401072   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
0x00401072   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
2 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
0x00401072   1  'VK_LAYER_LUNARG_image'  'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_image':
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

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

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

Vulkan: Creating the Logical Device's Queue

float queuePriorities[1] =
{
1.0f
};

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

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

Dynamic State Variables

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

Which Pipeline State Variables can be Changed Dynamically

Creating a Pipeline

VkDynamicState vds[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo vpdsci;

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

VkGraphicsPipelineCreateInfo vgpci;

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

Push Constants

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

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

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

```glsl
layout( push_constant ) uniform matrix
mat4 modelMatrix;
```

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

```glsl```
```
vkCmdPushConstants( CommandBuffer, PipelineLayout, stageFlags, offset, size, p/values );
```

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.

An Robotic Example using Push Constants

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

Where each arm is represented by:

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

```c```
```
struct armArm1;
struct armArm2;
struct armArm3;
```

In the Reset Function

```c```
```
struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
```
```
Arm1.armMatrix = glm::mat4(1.0f, 0.0f, 0.0f);
Arm1.armColor = glm::vec3(1.0f, 0.0f, 0.0f);
Arm1.armScale = 2.0f;
Arm2.armMatrix = glm::mat4(1.0f, 0.0f, 0.0f);
Arm2.armColor = glm::vec3(0.0f, 0.0f, 0.0f);
Arm2.armScale = 4.0f;
Arm3.armMatrix = glm::mat4(1.0f, 0.0f, 0.0f);
Arm3.armColor = glm::vec3(0.0f, 0.0f, 0.0f);
Arm3.armScale = 6.0f;
```

The constructor glm::mat4(1.0f, 0.0f, 0.0f) produces an identity matrix. The actual transformation matrices will be set in UpdateScene().

Setting up the Push Constants for the Pipeline Structure

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

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

```c```
```
VkPipelineLayoutCreateInfo vplci;```
```
vplci.pPushConstantRanges = &vpcr[1];
```

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

Setup the Push Constant for the Pipeline Structure

```c```
```
VkPushConstantRange vpcr[1];
vpcr[1].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
```
```
vpcr[1].offset = 0;
```
```
result = vkCreatePipelineLayout( LogicalDevice, IN &vpcr[1], PALLOCATOR, OUT &GraphicsPipelineLayout );```

Creating a Pipeline

A robotic animation (i.e., a hierarchical transformation system)
In the UpdateScene Function

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

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

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);

m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));

m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));

Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m1g * m21; // m2g
Arm3.armMatrix = m1g * m21 * m32; // m3g
```

In the RenderScene Function

```cpp
VkBuffer buffers[1] = { MyVertexBuffer.buffer };  
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

In the Vertex Shader

```cpp
layout(push_constant) uniform arm {
    mat4  armMatrix;
    vec3  armColor;
    float armScale;         // scale factor in x
} RobotArm;

layout(location = 0) in vec3 aVertex;

vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
bVertex.x += 1.; // now is [0., 2.]
bVertex.x /= 2.; // now is [0., 1.]
bVertex.x *= (RobotArm.armScale); // now is [0., RobotArm.armScale]
bVertex = vec3(RobotArm.armMatrix * vec4(bVertex, 1.));
```

Getting Information Back from the Graphics System

```
• 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
```

Setting up Query Pools

```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;

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

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

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);`

```c
#define DATASIZE 128
uint32_t data[DATASIZE];
```

```c
result = vkGetQueryPoolResults(LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags);
```

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)

### Occlusion Query

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

```c
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`.

```c
uint32_t counts[NUM_STATS];
result = vkGetQueryPoolResults(LogicalDevice, statisticsQueryPool, 0, 1, sizeof(uint32_t)*NUM_STATS, counts, 0, VK_QUERY_RESULT_WAIT_BIT);
```

### Timestamp Query

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

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

Even though the stages are "bits", you are supposed to only specify one of them, not "or" multiple ones together.
Here is how you create a Compute Pipeline

Start by Creating the Data Buffers

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

1. layout (std140, set = 0, binding = 0) buffer Pos
   vec4 Positions[ ]; // array of structures
2. layout (std140, set = 0, binding = 1) buffer Vel
   vec4 Velocities[ ]; // array of structures
3. layout (std140, set = 0, binding = 2) buffer Col
   vec4 Colors[ ]; // array of structures

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

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, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &pGpuMemory);
  // 0 and 0 are offset and flags
  memcpy(pGpuMemory, data, (size_t)myBuffer.size);
  vkUnmapMemory(LogicalDevice, myBuffer.vdm);
  return VK_SUCCESS;
}

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

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

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

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

```cpp
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 = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;

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

The Particle System Compute Shader -- Setup

```cpp
#version 430
#extension GL_ARB_compute_shader : enable

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
};

layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```

The Particle System Compute Shader -- The Physics

```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. ); // x, y, z, r

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, Sphere );
pp = p + vp*DT + .5*DT*DT*G;
}

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 );
}
```

Dispatching the Compute Shader from the Command Buffer

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

In Vulkan, all shaders get halfway-compiled by SPIR-V and then the rest-of-the-way compiled by the Vulkan driver. Normally, the half-way compile fixes all constant values and compiles the code that uses them. But, it would be nice every so often to have your Vulkan program sneak into the halfway-compiled binary and manipulate some constants at runtime. This is what Specialization Constants are for. A Specialization Constant is a way of injecting an integer, Boolean, uint, float, or double constant into 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.

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

In the compute shader:

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

In the Vulkan C/C++ program:

```cpp
int asize = 64;
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 7;
vsme[0].offset = 0;
vsme[0].size = sizeof(asize);
VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(asize);
vsi.pData = &asize;
```

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

```cpp
VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;
```
Specialization Constant Example – Setting Multiple Constants

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

In the C/C++ program:
```c
struct abc { int a, int b, float c; }
abc 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);
```  
```c
VkSpecializationInfo vsi;
vsi.mapEntryCount = 3;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsi.pData = &abc; // array of all the Specialization Constants
```

In the compute shader:
```glsl
int numXworkItems = 64;
```  
```glsl
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 12;
vsme[0].offset = 0;
vsme[0].size = sizeof(int);
```  
```glsl
VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(int);
vsi.pData = &numXworkItems;
```

Where Synchronization Fits in the Overall Block Diagram

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

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

**Creating a Semaphore**
```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
VkSemaphore semaphore;
result = vkCreateSemaphore( LogicalDevice, &vsci, PALLOCATOR, &semaphore );
```
Siemens Example during the Render Loop

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

Fences

```
#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
result = vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, &renderFence);
```

Events

```
VkEventCreateInfo veci;
veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;
veci.pNext = nullptr;
veci.flags = 0;
result = vkCreateEvent(LogicalDevice, &veci, PALLOCATOR, &event);
```

Controlling Events from the Host

```
VkEvent event;
result = vkCreateEvent(LogicalDevice, &event, PALLOCATOR, &event);
result = vkSetEvent(LogicalDevice, event);
result = vkResetEvent(LogicalDevice, event);
result = vkGetEventStatus(LogicalDevice, event);
```

- 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

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
from the Command Buffer Notes:

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

- vkCmdDrawIndexedIndirectCountAMD
- vkCmdDrawIndirectCountAMD
- vkCmdDrawIndexed
- vkCmdDraw
- vkCmdResolveImage
- vkCmdResetQueryPool
- vkCmdResetEvent
- vkCmdSetEvent
- vkCmdSetDiscardRectangleEXT
- vkCmdSetDeviceMaskKHX
- vkCmdSetDepthBounds
- vkCmdSetDepthBias
- vkCmdSetBlendConstants
- vkCmdResolveImage
- vkCmdResetQueryPool
- vkCmdResetEvent
- vkCmdSetDiscardRectangleEXT
- vkCmdSetDeviceMaskKHX
- vkCmdSetDepthBounds
- vkCmdSetDepthBias
- vkCmdSetBlendConstants

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.

- srcStageMask
- dstStageMask
- srcAccessMask
- dstAccessMask
- imageMemoryBarriers
- bufferMemoryBarriers
- memoryBarrierCount
- pBufferMemoryBarriers
- pImageMemoryBarriers
- memoryBarrierCount

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

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

http://cs.oregonstate.edu/~mjb/vulkan
Pipeline Stage Masks –

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

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_FRAMEBUFFER_ATTACHMENT_OBJECT_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_PHASES_BIT

Access Masks –

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

- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDIRECT_COMMAND_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_MEMORY_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_WRITE_BIT
- VK_ACCESS_UNIFORM_WRITE_BIT
- VK_ACCESS_INPUT_ATTACHMENT_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESSINDIRECT_COMMAND_WRITE_BIT

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

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

Example: Show image to viewer.
Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

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

Aliasing

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

Sampling Interval

Sampled Points

Aliasing

The Display We Want

Too often, the Display We Get

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

Supersampling

Multisampling

Setting up the Image

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

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

  - result = vkCreateGraphicsPipelines
    - LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline

Setting up the Image

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

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

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

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

# Fragment Shader calls = 2

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

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

```c
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;
```

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

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

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

Resolving the Image:

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

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

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

VkImageResolve vir;
vir.srcSubresource = visl;
vir.srcOffset = vo3;
visl.layerCount = 1;
visl.baseArrayLayer = 0;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

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

Thanks for coming today!

---

Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Computer Graphics API

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