An Introduction to the Vulkan Computer Graphics API

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

Acknowledgements

First of all, thanks to the inaugural class of 19 students who braved new, unrefined, and just-in-time course materials to take the first Vulkan class at Oregon State University – Winter Quarter, 2018. Thanks for your courage and patience!

Second, thanks to NVIDIA! The GeForce 1080ti cards are what made this course possible.

Third, thanks to Kathleen Mattson and the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding onto them.)

As an aside, the Vulkan development effort was originally called glNext, which created the false impression that this was a replacement for OpenGL. It’s not.

Why is it so important to keep the GPU Busy?

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.
Who is the Khronos Group?

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

Who’s Been Specifically Working on Vulkan?

Vulkan

- Somewhat derived from AMD’s Mantle API
- Also heavily influenced by Apple’s Metal API and Microsoft’s DirectX 12
- Goal: much less driver complexity and overhead than OpenGL has
- Goal: much less user hand-holding – Vulkan can crash
- Goal: higher single-threaded performance than OpenGL can deliver
- Goal: able to do multithreaded graphics
- Goal: able to handle tiled rendering

Vulkan Differences from OpenGL

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

Moving part of the driver into the application
Vulkan Topologies

VK_PRIMITIVE_TOPOLOGY_POINT_LIST
V0 V3 V1 V2

VK_PRIMITIVE_TOPOLOGY_LINE_LIST
V0 V3 V1 V2

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
V0 V3 V1 V2

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
V4

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

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;

A Colored Cube Example

Triangles Represented as an Array of Structures

Vertex 7
Vertex 5
Vertex 4
Vertex 3
Vertex 2
Vertex 0

Triangles Draw
8 7 6 5 4 3 2 1 0

Non-indexed Buffer Drawing

Filling the Vertex Buffer
What Int05DataBuffer Does

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

C/C++

```c
struct vertex
{
  glm::vec3 position; // 0
  glm::vec3 color;   // 24
  glm::vec3 normal; // 36
  glm::vec2 texCoord; // 48
};
```

GLSL Shader:

```glsl
layout( location = 1 ) in vec3 aColor; // 24
layout( location = 0 ) in vec3 aVertex; // 0
layout( location = 3 ) in vec2 aTexCoord; // 48
```

Telling the Pipeline about its Input

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

C/C++

```c
// Tell the pipeline about its input

uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);

VkVertexInputBindingDescription vvibd[1]; // one of these per buffer data buffer
vvibd[0].binding = 0; // which binding this is part of
vvibd[0].location = 0; // location in the layout decoration
vvibd[0].size = sizeof(VertexData[0]); // bytes between successive structs

VkVertexInputAttributeDescription vviad[4]; // array per vertex input attribute
vviad[0].binding = 0; // which binding this is part of
vviad[0].location = 0; // location in the layout decoration
vviad[0].offset = offsetof(struct vertex, position); // 0
vviad[0].format = VK_FORMAT_R32G32B32_SFLOAT; // float vec3

vviad[1].binding = 0; // which binding this is part of
vviad[1].location = 1; // location in the layout decoration
vviad[1].offset = offsetof(struct vertex, normal); // 12
vviad[1].format = VK_FORMAT_R32G32B32_SFLOAT; // float vec3

vviad[2].binding = 0; // which binding this is part of
vviad[2].location = 2; // location in the layout decoration
vviad[2].offset = offsetof(struct vertex, color); // 24
vviad[2].format = VK_FORMAT_R32G32B32_SFLOAT; // float vec3

vviad[3].binding = 0; // which binding this is part of
vviad[3].location = 3; // location in the layout decoration
vviad[3].offset = offsetof(struct vertex, texCoord); // 36
vviad[3].format = VK_FORMAT_R32G32_SFLOAT; // float vec2
```

GLSL Shader:

```glsl
layout( location = 0 ) in vec3 aVertex; // 0
layout( location = 1 ) in float aVertexCount; // size
layout( location = 2 ) in float aFirstInstance; // size
layout( location = 3 ) in vec2 aTexCoord; // 48
```

Telling the Command Buffer what to Draw

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

```c
VkDeviceSize vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
```

Better to do sizeof than to hard-code a number.
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.
A Data Buffer is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a “Binary Large Object”, or “BLOB”.)

It is up to you to be sure that the writer and the reader of the Data Buffer are interpreting the bytes in the same way!

Vulkan calls these things “Buffers”. But, Vulkan calls other things “Buffers”, too, such as Texture Buffers and Command Buffers. So, I have taken to calling these things “Data Buffers” and have even gone to far as to override some of Vulkan’s own terminology:

typedef VkBuffer VkDataBuffer;

Vulkan: Buffers

vkCreateBuffer( )

VkBufferCreateInfo

bufferUsage

queueFamilyIndices

size (bytes)

LogicalDevice

vkGetBufferMemoryRequirements( )

Buffer

VkMemoryAllocateInfo

size = memoryType

vkAllocateMemory( )

LogicalDevice

vkBindBufferMemory( )

bufferMemoryHandle

vkMapMemory( )

gpuAddress

<< do the memory copy >>
Finding the Right Type of Memory

```cpp
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_PROPERTYDEVICE_LOCAL_BIT ) != 0 )
        {
            return i;
        }
    }
    return -1;
}
```

Finding the Right Type of Memory

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

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

Finding the Right Type of Memory

Something I've Found Useful

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

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

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

Finding the Right Type of Memory

Here's the C struct to hold some uniform variables

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

Here's the shader code to access those uniform variables

```glsl
layout( std140, set = 0, binding = 0 ) uniform matBuf;
```

Finding the Right Type of Memory

```glsl
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.;
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 a container 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.

There is one more step in here—Descriptor Sets. Here’s a quick preview…

CPU:

Filling the Data 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, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT pGpuMemory);
    memcpy(pGpuMemory, data, (size_t)myBuffer.size);
    VkUnmapMemory(LogicalDevice, IN myBuffer.vdm);
    return VK_SUCCESS;
}

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

Copy to GPU Memory via Memory Mapping

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

[Shaders and SPIR-V]

Mike Bailey
mjb@cs.oregonstate.edu
http://cs.oregonstate.edu/~mjb/vulkan
The Shaders’ View of the Basic Computer Graphics Pipeline

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

Vulkan Shader Stages

- **Shader stages**
  - Vertex Shader
  - Geometry Shader
  - Tesselation Control Shader
  - Tesselation Evaluation Shader
  - Fragment Shader

Vulkan: GLSL Differences from OpenGL

- **Vertex and Instance indices:**
  - In OpenGL, there is an automatic `gl_VertexIndex`, `gl_InstanceIndex`
  - Both are 0-based

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

Vulkan Shader Compiling

- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V
- SPIR-V gets turned into fully-compiled code at runtime
- SPIR-V spec has been public for a couple of years — new shader languages are surely being developed
- OpenGL and OpenCL will be moving to SPIR-V as well

Vulkan: Shaders’ use of Layouts for Uniform Variables

- All opaque (non-sampler) uniform variables must be in block buffers
  - `layout (set=10, binding = 0 ) uniform mattBuffer {
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
  } Matrices;

  - Non-opaque must be in a uniform block
    - `layout (set=40, binding = 1, binding=0 ) uniform lightBuffer {
      vec4 uLightPos;
    } Light;

  - Non-opaque must be in a uniform block
    - `layout (set=40, binding = 1, binding=0 ) uniform lightBuffer {
      vec4 uLightPos;
    } Light;

  - Non-opaque must be in a uniform block
    - `layout (set=40, binding = 1, binding=0 ) uniform lightBuffer {
      vec4 uLightPos;
    } Light;`
SPIR-V: Standard Portable Intermediate Representation for Vulkan


Shader file extensions:
- .vert Vertex
- .tess Tessellation Control
- .tevs Tessellation Evaluation
- .geom Geometry
- .frag Fragment
- .comp Compute

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

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

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

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

You Can run SPIR-V from a Linux Shell

$ glslangValidator.exe -V sample-vert.vert -o sample-vert.spv
$ glslangValidator.exe -V sample-frag.frag -o sample-frag.spv

---

Compile for Vulkan (--V is compile for OpenGL)

Specify the output file

The input file. The compiler determines the shader type by the file extension:
- .vert Vertex shader
- .tccs Tessellation Control Shader
- .tecs Tessellation Evaluation Shader
- .geom Geometry shader
- .frag Fragment shader
- .comp Compute shader
Same as C/C++: the compiler gives you no nasty messages. Also, if you care, legal .spv files have a magic number of 0x07230203. So, if you do an `od -x` on the .spv file, the magic number looks like this:

```
0203 0723 . . .
```

### How do you know if SPIR-V compiled successfully?

You can also take a look at SPIR-V Assembly

```bash
glsangValidator.exe -V -H sample.vert -o sample.spv
```

This prints out the SPIR-V "assembly" to standard output. Other than nerd interest, there is no graphics-programming reason to look at this.

```glsl
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat3 uNormalMatrix;
} Matrices;

// non-opaque must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
  vec4 uLightPos;
} Light;

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

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

void main( )
{
  mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
  gl_Position = PVM * vec4( aVertex, 1.);
  vNormal = Matrices.uNormalMatrix * aNormal;
  vColor = aColor;
  vTexCoord = aTexCoord;
}
```
This is the SPIR-V Assembly, Part I

This is the SPIR-V Assembly, Part II

This is the SPIR-V Assembly, Part III

SPIR-V: Printing the Configuration

glsan/validator – c

Installing bash on Windows

1. Open Settings
2. Click on Update & security
3. Click on Developer
4. Under “Use developer features”, select the Developer mode option to setup the environment to install Bash.
5. On the message box, click Yes to turn on developer mode.
6. After the necessary components install, you need to restart your computer.
7. Once your computer reboots, open Control Panel.
8. Click on Programs.
9. Click on Turn Windows features on or off.
10. Check the Windows Subsystem for Linux (beta) option.
11. Click OK.
12. After the components installed on your computer, click the Restart now button to complete the task.

Managing the environment variables, you will notice that Bash will not appear in the “Recently added” list of apps, this is because Bash isn’t actually installed yet. Now that you have setup the necessary components, use the following steps to complete the installation of Bash:

1. Open Start, do a search for Bash.exe, and press Enter.
2. On the command prompt, type y and press Enter to download and install Bash from the Windows Store.
3. Then you need to create a default user account. This account doesn’t have to be the same as your Windows account. You need to replace the value in the required field and press Enter (you can’t use the username “admin”).
4. Close the “bash.exe” command prompt.

Now that you completed the installation and setup, you can open the Bash tool from the Start menu like you would with any other app.

SPIR-V: More Information

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

mjb – November 24, 2018
Caveats on the Sample Code

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.
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 a secret to you.
4. I've setup Vulkan structs every time they are used, even though, in most cases, they could have been partially or completely setup once and then re-used.
5. At times, I've setup things that didn't need to be setup just to show you what could go there.
6. There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'd a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you how certain options could be set.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

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

InitGraphics(), I

```c
void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices),   &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer,     (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light),      &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer(  sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer,                   (void *) VertexData );
    Init06CommandPool( );
    Init06CommandBuffers( );
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init08Swapchain( );
    Init09DepthStencilImage( );
    Init10RenderPasses( );
    Init11Framebuffers( );
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorSetLayouts();
    Init13DescriptorSets( );
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```
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 penalty for leaving in vertex attributes you aren’t going to use is memory space, but not 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.
Your Sample2017.zip File Contains This

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

Extras in the Code

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
GLFW Keyboard Callback

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

GLFW Mouse Button Callback

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

What is GLM?

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

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

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

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

If GLM is not installed in a system place, put it somewhere you can get access to.
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) );
```

The Most Useful GLM Variables, Operations, and Functions

---

GLM in the Vulkan sample.cpp Program

```c
    // viewing volume (assign, not concatenate):
    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 );
```

The Most Useful GLM Variables, Operations, and Functions

---

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

### Multiplications:
- `glm::mat4 * glm::mat4`
- `glm::mat4 * glm::vec4`
- `glm::mat4 * glm::vec3 (glm::vec3, 1.)`
- `glm::mat4 * glm::vec2 (glm::vec2, 1.)`

### 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 );`

---

Your Sample2017.zip File Contains GLM Already

---

Matrix Multiplication is not Commutative

```
Y

Rotate, then translate

X

Translate, then rotate
```
Matrix Multiplication is Associative

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} = \begin{bmatrix}
  T_{x', y', z'} & R_z & T_{x', y', z'} \\
  0 & 0 & 0 \\
  0 & 0 & 0 \\
  1
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

One matrix – the Current Transformation Matrix, or CTM

Why isn’t The Normal Matrix just the same as the Model Matrix?

Wrong!

Right!

Making each instance look differently -- Approach #1

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

```
gl_InstanceIndex starts at 0

To the vertex shader:

```
Making each Instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer and reference them
Still uses gl_InstanceIndex

In the vertex shader:

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

out vec3 vColor;

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

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

Making each Instance look differently -- Approach #3

Put a series of unique characteristics in a data buffer, one element per instance.
Read a new characteristic for each instance
Internally uses gl_InstanceIndex, but you don’t

How We Constructed the Graphics Pipeline Structure Before

```
VkVertexInputBindingDescription vvibd[1];
// an array containing one of these per buffer being used
vvibd[0].binding = 0;           // which binding # this is
vvibd[0].stride = sizeof( struct vertex );              // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX; // vertex
This definition says that we should
advance through the input buffer
by this much every time we hit a
new vertex
```

```
VkVertexInputAttributeDescription vviad[4];
// an array containing one of these per vertex attribute in all bindings
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position );                  // 0
vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;       // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal );                    // 12
vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;       // r, g, b
vviad[2].offset = offsetof( struct vertex, color );                     // 24
vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;       // s, t
vviad[3].offset = offsetof( struct vertex, texCoord );                  // 36
```

This is just the Vertex Input State Portion of the Graphics Pipeline Structure

How We Constructed the Graphics Pipeline Structure Before
How We Constructed the Graphics Pipeline Structure Before

```plaintext```

```glsl```
```
vkVertexInputAttributeDescription vviad[5];

result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, IN &
VkGraphicsPipelineCreateInfo vgpci

vgpci.pVertexInputState = &vpvisci;

vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.vertexBindingDescriptionCount = 2;
vpvisci.flags = 0;
vpvisci.pNext = nullptr;
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = vviad;
```
```

How We Construct the Graphics Pipeline Structure Now

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

Create a data buffer with one glm::vec3 (to hold r, g, b) for each Instance.
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Wouldn't it be nice if we could update a bunch of related uniform variables all at once?

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

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

In OpenGL, these are all in one set. They all get bound, whether you need them here or not.

In OpenGL, we have "layout( std140, binding = 0 ) uniform mat4 uModelMatrix;"

If we had

```
layout( std140, set = 0, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, set = 1, binding = 0 ) uniform mat4 uProjMatrix;
layout( std140, set = 2, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, set = 3, binding = 0 ) uniform mat4 uNormalMatrix;
layout( std140, set = 4, binding = 0 ) uniform vec4 uTime;
layout( std140, set = 5, binding = 0 ) uniform int uMode;
layout( std140, set = 6, binding = 0 ) uniform int uLighting;
layout( std140, set = 7, binding = 0 ) uniform sampler2D uSampler;
```

We would need to do these for each scene.

We can update uniforms "on the fly" by swapping Descriptor Sets.

For each scene, we need

```c
Bind Descriptor Set #0
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''}
Step 2: Define the Descriptor Set Layouts

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

2. Define the Descriptor Set Layouts:
   - Create an `VkDescriptorSetLayoutCreateInfo` structure for each layout.
   - Specify the binding count, binding types, and descriptor types for each layout.

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

- Create a `VkPipelineLayoutCreateInfo` structure.
- Set the `layoutCount` to the number of descriptors set layouts.
- Set the `pSetLayouts` to point to an array of descriptor set layout handles.

Step 4: Allocating the Memory for Descriptor Sets

- Create a `VkDescriptorSetAllocateInfo` structure.
- Set the `descriptorPool` to the descriptor pool.
- Set the `descriptorSetCount` to the number of descriptor sets.
- Set the `pSetLayout` to the pipeline layout.

Function Call Examples:

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

result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
```

Memory Layout:

- Uniform Buffer: `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`
- Combined Image Sampler: `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`
- Texture Sampler: `VK_DESCRIPTOR_TYPE_SAMPLER`
- Sampler: `VK_DESCRIPTOR_TYPE_SAMPLER`
Step 5: Tell the Descriptor Sets where their CPU Data is

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

// ds 1:
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.descriptorCount = 1;
vwds1.dstArrayElement = 0;
vwds1.dstBinding = 0;
vwds1.dstSet = DescriptorSets[1];
vwds1.pNext = nullptr;
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;

// ds 2:
vwds2.pBufferInfo = nullptr;
vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds2.descriptorCount = 1;
vwds2.dstArrayElement = 0;
vwds2.dstBinding = 0;
vwds2.dstSet = DescriptorSets[2];
vwds2.pNext = nullptr;
vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;

// ds 3:
vwds3.dstSet = DescriptorSets[3];
vwds3.pNext = nullptr;
vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
```

Good to use `sizeof`.

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

```c
vgpci = (VkGraphicsPipelineCreateInfo *)malloc(sizeof(VkGraphicsPipelineCreateInfo));
if (vgpci)
{
    memset(vgpci, 0, sizeof(VkGraphicsPipelineCreateInfo));
    vgpci->sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    vgpci->pNext = nullptr;
    vgpci->layout = DescriptorSets[0];
    vgpci->basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
    vgpci->basePipelineIndex = 0;
    vgpci->pStages = vpssci;
    vgpci->stageCount = 2;
    vgpci->pVertexInputState = &vpvisci;
    vgpci->pInputAssemblyState = &vpiasci;
    vgpci->pRasterizationState = &vprsci;
    vgpci->pColorBlendState = &vpcbsci;
    vgpci->pDepthStencilState = &vpdssci;
    vgpci->pDynamicState = &vpdsci;
    vgpci->pShaderStages = &vpssci;
    vgpci->graphicsPipelineCreateFlags = 0;
}
```

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

```c
vkCmdBindDescriptorSets( LogicalDevice, VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline, 4, DescriptorSets, 0, (uint32_t *)nullptr );
```
The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”. There’s a lot that goes into it. For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.

1. The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”.
2. There’s a lot that goes into it.
3. For the most part, the Graphics Pipeline is meant to be immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

The First Step: Create the Graphics Pipeline Layout

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

```c
void Init14GraphicsVertexFragmentPipeline(VkDevice device, VkPipelineLayout createInfo, VkPipelineLayout outInfo)
{
    VkResult result = Init14GraphicsVertexFragmentPipeline(device, createInfo, outInfo);
    return result;
}
```

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

Vulkan: A Pipeline Records the Following Items:

- **Pipeline Layout**: DescriptorSets, PushConstants
- **Vertex Shader**
- **Tessellation Shaders**
- **Geometry Shader**
- **Fragment Shader**
- **Vertex Input attributes**: location, binding, format, offset
- **Specialization info**: which stage (VERTEX, etc.)
- **Input Assembly**: topology
- **Viewport**: x, y, w, h
- **Dynamic State**: which states can be set dynamically (bound to the command buffer, outside the Pipeline)
- **Blending**: blendEnable, blendFactor, blendOp
- **Stencil**: stencilTestEnable, stencilOp
- **Depth**: depthTestEnable, depthWriteEnable, depthCompareOp
- **Depth Clamping**: depthClampingEnable
- **Scissoring**: scissorTestEnable, scissor
- **Line Width**: lineWidth
- **Rasterization**: cullMode, polygonMode, frontFace, lineWidth
- **Cull Mode**: Front, Back, None
- **Discard Enable**: discard

Creating a Typical Graphics Pipeline

```c
Vulkan::Init14GraphicsVertexFragmentPipeline(VkDevice device, VkPipelineLayout createInfo, VkPipelineLayout outInfo)
{
    VkResult result = Init14GraphicsVertexFragmentPipeline(device, createInfo, outInfo);
    return result;
}
```

These settings seem pretty typical to me. Let’s write a simplified example using the bare minimum - the pipeline layout and the topology, and always uses the settings in red above.
or,

When using the primitive restart code, the easy way to do it is like this:

```cpp
if (your VkIndexType is VK_INDEX_TYPE_UINT32, then the special index is
```
What is the Difference Between Changing the Viewport and Changing the Scissoring?

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 (squeezed) into the viewport area.

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”?

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

A good use for this is Polygon Capping:

The front of the polygon is clipped, exposing to the viewer that this is really a shell, not a solid.

The gray area shows what would happen with depthClampEnable (except it would have been red).

What is “Depth Bias Enable”?  

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

MultiSampling State
**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.

- VK_COLOR_COMPONENT_R_BIT
- VK_COLOR_COMPONENT_G_BIT
- VK_COLOR_COMPONENT_B_BIT
- VK_COLOR_COMPONENT_A_BIT

**Which Pipeline Variables can be Set Dynamically?**

- VK_DYNAMIC_STATE_VIEWPORT
- VK_DYNAMIC_STATE_SCISSOR

**Uses for Stencil Operations**

Polygon edges without Z-fighting

Magic Lens

**Stencil Operations for Front and Back Faces**

- VK_STENCIL_OP_DECREMENT_AND_WRAP
- VK_STENCIL_OP_INCREMENT_AND_CLAMP
- VK_STENCIL_OP_REPLACE
- VK_STENCIL_OP_ZERO
- VK_STENCIL_OP_KEEP

**Operations for Depth Values**

- VK_COMPARE_OP_LESS
- VK_COMPARE_OP_EQUAL
- VK_COMPARE_OP_NEVER

- VK_COMPARE_OP_ALWAYS
- VK_COMPARE_OP_GREATER_OR_EQUAL
- VK_COMPARE_OP_GREATER
- VK_COMPARE_OP_LESS_OR_EQUAL
- VK_COMPARE_OP_EQUAL

- VK_COMPARE_OP_NEVER

- VK_COMPARE_OP_ALWAYS
- VK_COMPARE_OP_GREATER_OR_EQUAL
- VK_COMPARE_OP_GREATER
- VK_COMPARE_OP_LESS_OR_EQUAL
- VK_COMPARE_OP_EQUAL

This controls blending between the output of the fragment shader and the input to the color attachments.
**Group all of the individual state information and create the pipeline**

```
// VkGraphicsPipelineCreateInfo
VkGraphicsPipelineCreateInfo vgpci = {};  // Need to initialize

vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.pipelineCreateFlags = 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;
```

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

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

**Putting it all Together! (finally…)**

**Vulkan: a More Typical (and Simplified) Block Diagram**

**Queues and Command Buffers**

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

**Querying what Queue Families are Available**

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

11/24/2018
Similarly, we can write a function that finds the proper queue family:

```c
int FindQueueFamilyThatDoesGraphics(VkPhysicalDevice PhysicalDevice)
{
    uint32_t count = 0;
    VkQueueFamilyProperties *vqfp = nullptr;
    vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, &count, (VkQueueFamilyProperties **)&vqfp);
    for (uint32_t i = 0; i < count; i++)
    {
        if ((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
            return i;
    }
    return -1;
}
```

Creating a Logical Device Queue Needs to Know Queue Family Information:

```c
VkQueueCreateInfo queueCreateInfo = {
    .sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO,
    .queueFamilyIndex = FindQueueFamilyThatDoesGraphics(),
    .queueCount = 1,
    .pQueuePriorities = &queuePriorities[0],
};
```

Creating the Command Pool as part of the Logical Device:

```c
vkCreateCommandPool(LogicalDevice, &queueCreateInfo, nullptr, &commandPool);
```

Creating the Command Buffers:

```c
vkAllocateCommandBuffers(LogicalDevice, &commandPool, CommandBuffers, count);
```

Beginning a Command Buffer:

```c
vkBeginCommandBuffer(CommandBuffers[nextImageIndex], &commandBeginInfo);
```

Similarly, we can write a function that finds the proper queue family:

```c
int FindQueueFamilyThatDoesGraphics(VkPhysicalDevice PhysicalDevice)
{
    uint32_t count = 0;
    VkQueueFamilyProperties *vqfp = nullptr;
    vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, &count, (VkQueueFamilyProperties **)&vqfp);
    for (uint32_t i = 0; i < count; i++)
    {
        if ((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
            return i;
    }
    return -1;
}
```

Creating a Logical Device Queue Needs to Know Queue Family Information:

```c
VkQueueCreateInfo queueCreateInfo = {
    .sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO,
    .queueFamilyIndex = FindQueueFamilyThatDoesGraphics(),
    .queueCount = 1,
    .pQueuePriorities = &queuePriorities[0],
};
```

Creating the Command Pool as part of the Logical Device:

```c
vkCreateCommandPool(LogicalDevice, &queueCreateInfo, nullptr, &commandPool);
```

Creating the Command Buffers:

```c
vkAllocateCommandBuffers(LogicalDevice, &commandPool, CommandBuffers, count);
```

Beginning a Command Buffer:

```c
vkBeginCommandBuffer(CommandBuffers[nextImageIndex], &commandBeginInfo);
```
These are the Commands that could be entered into the Command Buffer, I

1. `vkCmdBindPipeline` (commandBuffer, pipeline);
2. `vkCmdBeginQuery` (commandBuffer, flags);
3. `vkCmdEndQuery` (commandBuffer, query);
4. `vkCmdEndRenderPass` (commandBuffer);
5. `vkCmdDraw` (commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
6. `vkCmdDispatch` (commandBuffer, groupCountX, groupCountY, groupCountZ);
7. `vkCmdDispatchIndirect` (commandBuffer, offset);
8. `vkCmdDebugMarkerBeginEXT` (commandBuffer, pMarkerInfo);
9. `vkCmdCopyBufferToImage` (commandBuffer, pRegions);
10. `vkCmdCopyBuffer` (commandBuffer, pRegions);
11. `vkCmdClearDepthStencilImage` (commandBuffer, pRanges);
12. `vkCmdClearColorImage` (commandBuffer, pRanges);
13. `vkCmdBlitImage` (commandBuffer, filter);
14. `vkCmdBindVertexBuffers`
15. `vkCmdDrawIndirectCountAMD` (commandBuffer, stride);
16. `vkCmdDrawIndirect` (commandBuffer, stride);
17. `vkCmdDrawIndexedIndirect` (commandBuffer, stride);
18. `vkCmdDrawIndexed` (commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);

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

1. `vkCmdWriteTimestamp` (commandBuffer, pipelineStage, queryPool, query);
2. `vkCmdSetBlending` (commandBuffer, blendConstants[4]);
3. `vkCmdSetBlendConstants` (commandBuffer, blendConstants[4]);
4. `vkCmdResetQueryPool` (commandBuffer, queryPool, firstQuery, queryCount);
5. `vkCmdReserveSpaceForCommandsNVX` (commandBuffer, pReserveSpaceInfo);
6. `vkCmdPushDescriptorSetWithTemplate` (commandBuffer, descriptorUpdateTemplate, layout, set, pData);
7. `vkCmdPushDescriptorSets` (commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);
8. `vkCmdPushConstants` (commandBuffer, layout, stageFlags, offset, size, pValues);
9. `vkCmdPipelineBarrier` (commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
10. `vkCmdSetEvent` (commandBuffer, event, stageMask);
11. `vkCmdSetDeviceMaskKHX` (commandBuffer, deviceMask);
12. `vkCmdSetDepthBounds` (commandBuffer, minDepthBounds, maxDepthBounds);

Submitting a Command Buffer to a Queue for Execution

1. `vkBeginCommandBuffer` (CommandBuffers[nextImageIndex], 0, 1, IN &vcbbi);
2. `vkCmdBindPipeline` (commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, CommandBuffers[nextImageIndex], 0, 1, IN &vpbi);
3. `vkCmdBindDescriptorSets` (commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, CommandBuffers[nextImageIndex], 0, 1, IN &vpbi, buffers, offsets);
4. `vkCmdSetViewport` (commandBuffer, firstViewport, viewportCount, pViewports);
5. `vkCmdSetScissor` (commandBuffer, firstScissor, scissorCount, pScissors);
6. `vkCmdSetViewport` (commandBuffer, IN &viewport);
7. `vkCmdSetScissor` (commandBuffer, IN &scissor);
8. `vkCmdSetViewport` (commandBuffers[nextImageIndex], 0, 1, IN &vpbi);
9. `vkCmdBeginQuery` (commandBuffers[nextImageIndex], flags);
10. `vkCmdEndQuery` (commandBuffers[nextImageIndex], query);
11. `vkCmdBeginRenderPass` (commandBuffers[nextImageIndex], 0, 1, IN &vrbpi);
12. `vkCmdEndRenderPass` (commandBuffers[nextImageIndex], 0, 1, IN &vrbpi);
13. `vkCmdBeginQuery` (commandBuffers[nextImageIndex], flags);
14. `vkCmdEndQuery` (commandBuffers[nextImageIndex], query);
15. `vkCmdWriteTimestamp` (commandBuffers[nextImageIndex], pipelineStage, queryPool, query);
16. `vkCmdSetBlending` (commandBuffers[nextImageIndex], blendConstants[4]);
17. `vkCmdSetBlending` (commandBuffers[nextImageIndex], blendConstants[4]);
18. `vkCmdResetQueryPool` (commandBuffers[nextImageIndex], queryPool, firstQuery, queryCount);
19. `vkCmdReserveSpaceForCommandsNVX` (commandBuffers[nextImageIndex], pReserveSpaceInfo);
20. `vkCmdPushDescriptorSetWithTemplate` (commandBuffers[nextImageIndex], descriptorUpdateTemplate, layout, set, pData);
21. `vkCmdPushDescriptorSets` (commandBuffers[nextImageIndex], pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);
22. `vkCmdPushConstants` (commandBuffers[nextImageIndex], layout, stageFlags, offset, size, pValues);
23. `vkCmdPipelineBarrier` (commandBuffers[nextImageIndex], srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
24. `vkCmdSetEvent` (commandBuffers[nextImageIndex], event, stageMask);
25. `vkCmdSetDeviceMaskKHX` (commandBuffers[nextImageIndex], deviceMask);
26. `vkCmdSetDepthBounds` (commandBuffers[nextImageIndex], minDepthBounds, maxDepthBounds);

vkAcquireNextImage

1. `uint32_t nextImageIndex;`
The Entire Submission / Wait / Display Process

1. Create fence
2. Get the queue
3. Fill in the queue information
4. Submit the queue
5. Wait for the fence

How We Think of OpenGL Framebuffers

Video

Driver

Depth-Buffer

Swap Chain

What is a Swap Chain?

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

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

Swap Chains are arranged as a ring buffer

Swap Chains are tightly coupled to the window system.

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

1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to "front buffer")
We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

```c
VulkanDebug.txt output:

Found 3 Present Modes:

- **This Surface is supported by the Graphics Queue**
- `vkGetPhysicalDeviceSurfaceCapabilities`

- `2:        1 ( VK_PRESENT_MODE_MAILBOX )`
- `1:        3 ( VK_PRESENT_MODE_FIFO_RELAXED )`
- `0:        2 ( VK_PRESENT_MODE_FIFO )`

- `0:       44                0 ( VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR )`

- `result = supportedUsageFlags = 0x009f`
- `supportedCompositeAlpha = 0x0001`
- `currentTransform = 0x0001`
- `supportedTransforms = 0x0001`
- `maxImageArrayLayers = 1`
- `maxImageExtent = 1024 x 1024`
- `minImageExtent = 1024 x 1024`
- `currentExtent = 1024 x 1024`
- `minImageCount = 2 ; maxImageCount = 8`

```

Creating a Swap Chain

```c
Creating a Swap Chain Images and Image Views

```c
WnSurfaceCapabilities
```
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    ...  
    VkAcquireNextImage(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);
    ...
    VkSubmitInfo vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.waitSemaphoreCount = 1;
    vsi.pWaitSemaphores = &imageReadySemaphore;
    vsi.pWaitDstStageMask = &waitAtBottom;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
    result = vkQueueSubmit(presentQueue, 1, &vsi, IN renderFence);  // 1 = submitCount
    ...
    VkPresentInfo vpi;
    vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO;
    vpi.pNext = nullptr;
    vpi.waitSemaphoreCount = 0;
    vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
    vpi.swapchainCount = 1;
    vpi.pSwapchains = &SwapChain;
    vpi.pImageIndices = &nextImageIndex;
    vpi.pResults = (VkResult *) nullptr;
    result = vkQueuePresent(presentQueue, IN &vpi);  

Enable texture mapping:

```gl
    glEnable(GL_TEXTURE_2D);
```

Draw your polygons, specifying a (s,t) at each vertex:

```gl
    glBegin(GL_POLYGON);
    glTexCoord2f(s0, t0);
    glNormal3f(nx0, ny0, nz0);
    glVertex3f(x0, y0, z0);
    glTexCoord2f(s1, t1);
    glNormal3f(nx1, ny1, nz1);
    glVertex3f(x1, y1, z1);
    // . . .
    glEnd();
```

Disable texture mapping:

```gl
    glDisable(GL_TEXTURE_2D);
```
The easiest way to figure out what s and t are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane,

\[
s = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}},
\]

\[
t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}},
\]

Or, for a sphere,

\[
s = \frac{\theta - (-\pi)}{2\pi},
\]

\[
t = \frac{\phi - \frac{\pi}{2}}{\pi}.
\]

From the Sphere code:

\[
s = \frac{(\text{lng} + \pi)}{2\pi},
\]

\[
t = \frac{(\text{lat} + \pi/2)}{\pi}.
\]
As an object gets farther away and covers a smaller and smaller part of the screen, the texels : pixels ratio used in the coverage becomes larger and larger. This means that there are pieces of the texture leftover in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

Textures’ Undersampling Artifacts

Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all white, with the red texel having no effect on the colors in the final image. The solution is to create lower-resolution versions of the same texture so that the red texel gets filtered into in resolution-level textures.

Texture Mip*-mapping

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

"Latin: multae in panvo, "many things in a small place"
# endif

VK_IMAGE_LAYOUT_UNDEFINED

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

result =

if (Verbose)
    vkGetImageMemoryRequirements
    VkMemoryRequirements vmr;
    |
    {
        if (Verbose)
            VkImageSubresource vis;
            result =
            result =
    }
    if (Verbose)
        VkMemoryRequirements vmr;

VkExtent3D                              ve3;
VkOffset3D                              vo3;
VkImageSubresourceLayers visl;

// now do the final image transfer:
VkImageMemoryBarrier vimb;
VkImageSubresourceRange visr;

vkBindImageMemory

vkAllocateMemory

vkCreateImage(LogicalDevice

ve3.depth = 1;
ve3.width = texWidth;
vo3.z = 0;
vo3.y = 0;
vo3.x = 0;
visl.baseArrayLayer = 0;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
1, IN &vimb);

vimb.subresourceRange = visr;
vimb.srcAccessMask = 0;

vimb.image = textureImage;

vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;

vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

vimb.pNext = nullptr;

vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

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

vkCmdPipelineBarrier

vkCmdCopyImage(TextureCommandBuffer

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

vkCmdCopyImage

vkUnmapMemory

void * gpuMemory;

memcpy

vkMapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()

FindMemoryThatIsDeviceLocal(

vkMapMemory

void * gpuMemory;

memcpy

vkUnmapMemory

FindMemoryThatIsHostVisible()
result = Init06TextureSampler( &MyPuppyTexture.texSampler );

Note that, at this point, the CPU buffer and the GPU Staging Buffer are no longer needed, and can be destroyed.

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.

Vulkan: a More Typical (and Simplified) Block Diagram

Querying the Number of Physical Devices

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

Note that, at this point, the CPU buffer and the GPU Staging Buffer are no longer needed, and can be destroyed.
Here's What the Intel HD Graphics 520 Produced

<table>
<thead>
<tr>
<th>Vulkan: Identifying the Physical Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkEnumeratePhysicalDevices:</td>
</tr>
<tr>
<td>Device:</td>
</tr>
<tr>
<td>Device Name: Intel(R) HD Graphics 520</td>
</tr>
<tr>
<td>Device ID: 0x1916</td>
</tr>
<tr>
<td>Vendor ID: 0x8086</td>
</tr>
<tr>
<td>Driver version: 4194360</td>
</tr>
<tr>
<td>API version: 4194360</td>
</tr>
<tr>
<td>Physical Device Type: VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU</td>
</tr>
<tr>
<td>Physical Device Features:</td>
</tr>
<tr>
<td>geometryShader = 1</td>
</tr>
<tr>
<td>multiDrawIndirect = 1</td>
</tr>
<tr>
<td>largePoints = 1</td>
</tr>
<tr>
<td>inclusionQueryPrecise = 1</td>
</tr>
<tr>
<td>shaderFloat64 = 1</td>
</tr>
<tr>
<td>shaderInt16 = 1</td>
</tr>
<tr>
<td>shaderInt64 = 1</td>
</tr>
<tr>
<td>shaderFloat64 = 1</td>
</tr>
<tr>
<td>shaderInt16 = 1</td>
</tr>
<tr>
<td>shaderInt64 = 1</td>
</tr>
</tbody>
</table>

Here's What the NVIDIA 1080ti Produced

<table>
<thead>
<tr>
<th>Which Physical Device to Use, I</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkEnumeratePhysicalDevices:</td>
</tr>
<tr>
<td>Device:</td>
</tr>
<tr>
<td>Device Name: GeForce GTX 1080 Ti</td>
</tr>
<tr>
<td>Device ID: 0x1b06</td>
</tr>
<tr>
<td>Vendor ID: 0x10de</td>
</tr>
<tr>
<td>Driver version: 4194360</td>
</tr>
<tr>
<td>API version: 4194360</td>
</tr>
<tr>
<td>Physical Device Type: VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU</td>
</tr>
<tr>
<td>Physical Device Features:</td>
</tr>
<tr>
<td>geometryShader = 1</td>
</tr>
<tr>
<td>multiDrawIndirect = 1</td>
</tr>
<tr>
<td>largePoints = 1</td>
</tr>
<tr>
<td>inclusionQueryPrecise = 1</td>
</tr>
<tr>
<td>shaderFloat64 = 1</td>
</tr>
<tr>
<td>shaderInt16 = 1</td>
</tr>
<tr>
<td>shaderInt64 = 1</td>
</tr>
</tbody>
</table>

Which Physical Device to Use, II

| VkEnumeratePhysicalDevices:      |
| Device:                         |
| Device Name: 'Intel(R) HD Graphics 520' |
| Device ID: 0x1916                |
| Vendor ID: 0x8086                |
| Driver version: 4194360          |
| API version: 4194360             |
| Physical Device Type: VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU |
| Physical Device Features:       |
| geometryShader = 1              |
| multiDrawIndirect = 1           |
| largePoints = 1                 |
| inclusionQueryPrecise = 1       |
| shaderFloat64 = 1               |
| shaderInt16 = 1                 |
| shaderInt64 = 1                 |

Here's What the NVIDIA 1080ti Produced

<table>
<thead>
<tr>
<th>Which Physical Device to Use, II</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkEnumeratePhysicalDevices:</td>
</tr>
<tr>
<td>Device:</td>
</tr>
<tr>
<td>Device Name: GeForce GTX 1080 Ti</td>
</tr>
<tr>
<td>Device ID: 0x1b06</td>
</tr>
<tr>
<td>Vendor ID: 0x10de</td>
</tr>
<tr>
<td>Driver version: 4194360</td>
</tr>
<tr>
<td>API version: 4194360</td>
</tr>
<tr>
<td>Physical Device Type: VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU</td>
</tr>
<tr>
<td>Physical Device Features:</td>
</tr>
<tr>
<td>geometryShader = 1</td>
</tr>
<tr>
<td>multiDrawIndirect = 1</td>
</tr>
<tr>
<td>largePoints = 1</td>
</tr>
<tr>
<td>inclusionQueryPrecise = 1</td>
</tr>
<tr>
<td>shaderFloat64 = 1</td>
</tr>
<tr>
<td>shaderInt16 = 1</td>
</tr>
<tr>
<td>shaderInt64 = 1</td>
</tr>
</tbody>
</table>
Asking About the Physical Device’s Different Memories

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

Here’s What I Got

11 Memory Types:
Memory  0:
Memory  1:
Memory  2:
Memory  3:
Memory  4:
Memory  5:
Memory  6:
Memory  7:  DeviceLocal
Memory  8:  DeviceLocal
Memory  9:  HostVisible HostCoherent
Memory 10:  HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0:  size = 0x75000000 DeviceLocal
Heap 1:  size = 0x89000000

Asking About the Physical Device’s Queue Families

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

Here’s What I Got

Found 3 Queue Families:
0: queueCount = 16  ;    Graphics Compute  Transfer
1: queueCount =  1  ;    Transfer
2: queueCount =  8  ;    Compute
Looking to See What Device Layers are Available

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

Looking to See What Device Extensions are Available

```c
const char *
myDeviceExtensions
[ ] =
{ 
    VK_surface,
    VK_win32_surface,
    VK_EXT_debug_report,
};
```

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

Looking to See What Device Extensions are Available

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

What Device Layers and Extensions are Available

3 physical device layers enumerated:
```c

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

Vulkan: Specifying a Logical Device Queue

```c
float queuePriorities[1] = { 1.0f };
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.pQueueProperties = queuePriorities;
```

Vulkan: Creating a Logical Device

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

Vulkan: Creating the Logical Device's Queue

```c
uint32_t queueIndex = 0;
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.pQueueProperties = queueProperties;
```c

Vulkan: Creating a Logical Device

```c
uint32_t queueIndex = 0;
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.pQueueProperties = queueProperties;
```c

Vulkan: Creating the Logical Device's Queue

```c
uint32_t queueIndex = 0;
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.pQueueProperties = queueProperties;
```c

Vulkan: Creating a Logical Device
Vulkan Layers

![Vulkan Layers Diagram]

Layers are code that can be installed between the Application and Vulkan. Normally, Vulkan is meant to run “flat out”. Layers can take the extra time to perform useful functions like printing debugging messages, printing function calls, etc.

They are not always necessary, but when you need them, you will be really glad they are there!

- Vulkan
- Layer C
- Layer B
- Application

Looking to See What Extensions are Both Wanted and Available

```c
vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
 0x00000008  'VK_EXT_debug_report'
 0x00000001  'VK_KHR_get_surface_capabilities2'
 0x00000001  'VK_KHR_surface'
 0x00000001  'VK_NVX_device_group_creation'
 0x00000001  'VK_external_memory_capabilities'
 0x00000001  'VK_KHR_external_memory_capabilities'
 0x00000001  'VK_KHR_external_semaphore_capabilities'
 0x00000001  'VK_KHR_external_fence_capabilities'
 0x00000001  'VK_KHR_external_memory_capabilities'
 0x00000001  'VK_KHR_surface_capabilities2'

VkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
 0x00000008  'VK_EXT_debug_report'
 0x00000001  'VK_KHR_get_surface_capabilities2'
 0x00000001  'VK_KHR_surface'
 0x00000001  'VK_NVX_device_group_creation'
 0x00000001  'VK_external_memory_capabilities'
 0x00000001  'VK_KHR_external_memory_capabilities'
 0x00000001  'VK_KHR_external_semaphore_capabilities'
 0x00000001  'VK_KHR_external_fence_capabilities'
 0x00000001  'VK_KHR_surface_capabilities2'

// look for extensions both on the wanted list and the available list:
std::vector<char *> extensionsWantedAndAvailable;
extensionsWantedAndAvailable.clear();
for (uint32_t wanted = 0; wanted < numExtensionsWanted; wanted++)
{
    for (uint32_t available = 0; available < numExtensionsAvailable; available++)
    {
        if (strcmp(instanceExtensions[wanted], InstanceExtensions[available].extensionName) == 0)
        {
            extensionsWantedAndAvailable.push_back(InstanceExtensions[available].extensionName);
            break;
        }
    }
}
```

Looking to See What Instance Layers and Instance Extensions are Available

```
const char * instanceLayers[] = {
  "VK_LAYER_LUNARG_api_dump", // turn this on if want to see each function call and its arguments (very slow!)
  "VK_LAYER_LUNARG_core_validation",
  "VK_LAYER_LUNARG_object_tracker",
  "VK_LAYER_LUNARG_parameter_validation",
  "VK_LAYER_NV_optimus",
};

const char * instanceExtensions[] = {
  "VK_surface",
  #ifdef _WIN32
  "VK_win32_surface",
  #endif
  "VK_EXT_debug_report",
};
```

```
// look for layers that are available:
void *libraryHandle = loadLibrary("sofile.so");

// look for layers that are available:
VkEnumerateInstanceLayerProperties(instanceLayers, &numLayersAvailable, InstanceLayers);
```

```
uint32_t numExtensionsWanted = sizeof(instanceExtensions) / sizeof(char *);
// see what extensions are available:
vkEnumerateInstanceExtensionProperties((char *)nullptr, &numExtensionsAvailable, (VkExtensionProperties *)nullptr);
InstanceExtensions = new VkExtensionProperties[numExtensionsAvailable];
result = vkEnumerateInstanceExtensionProperties((char *)nullptr, &numExtensionsAvailable, InstanceExtensions);
```

```
for (uint32_t wanted = 0; wanted < numExtensionsWanted; wanted++)
{
    for (uint32_t available = 0; available < numExtensionsAvailable; available++)
    {
        if (strcmp(instanceExtensions[wanted], InstanceExtensions[available].extensionName) == 0)
        {
            extensionsWantedAndAvailable.push_back(InstanceExtensions[available].extensionName);
            break;
        }
    }
}
```
Will now ask for 3 instance extensions
  VK_surface
  VK_win32_surface
  VK_EXT_debug_report

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

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ ){
  VkPhysicalDeviceProperties vpdp;
  vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp);
  // need some logical here to decide which physical device to select:
  if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
    discreteSelect = i;
  if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
    integratedSelect = i;
}

int which = -1;
if( discreteSelect >= 0 )
  which = discreteSelect;
PhysicalDevice = physicalDevices[which];
else if( integratedSelect >= 0 )
  which = integratedSelect;
PhysicalDevice = physicalDevices[which];
else
  fprintf( FpDebug, "Could not select a Physical Device\n" );
  return VK_SHOULD_EXIT;
delete[ ] physicalDevices;

vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp );
vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp );
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp);
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr );
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
delete[ ] vqfp;

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

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

uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, deviceLayers);
for (unsigned int i = 0; i < layerCount; i++)
{
  // see what device extensions are available:
  uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
(VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount,
deviceExtensions);
//
}
delete[ ] deviceLayers;
vkEnumerateDeviceLayerProperties:
3 physical device layers enumerated:
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarGValidation Layer'
0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':

Synchronization

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

Semaphores

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

Ask for Something
Try to Use the Something

Creating a Semaphore

VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
uint32_t nextImageIndex;
vkAcquireNextImage( LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkSubmitInfo vsi;
vsii.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore) nullptr;
result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence );
Fences

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

```
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT; // = 0
// VK_FENCE_CREATE_SIGNALED_BIT is only other option

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

// returns right away:
result = vkGetFenceStatus(LogicalDevice, IN fence);
// result = VK_SUCCESS means it has signaled
// result = VK_NOT_READY means it has not signaled

// blocks:
result = vkWaitForFences(LogicalDevice, 1, IN &fence, waitForAll, timeout);
// waitForAll = VK_TRUE:   wait for all fences in the list
// waitForAll = VK_FALSE: wait for any one fence in the list
// timeout is a uint64_t timeout in nanoseconds  (could be 0, which means to return immediately)
// timeout can be up to UINT64_MAX = 0xffffffff (384 years) or 0xffffffffffffffff = 580+ years
// result = VK_SUCCESS means it returned because a fence (or all fences) signaled
// result = VK_TIMEOUT means it returned because the timeout was exceeded
```

Events

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

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

VkEvent event;
result = vkCreateEvent(LogicalDevice, IN &veci, PALLOCATOR, OUT &event);
result = vkSetEvent(LogicalDevice, IN event);
result = vkResetEvent(LogicalDevice, IN event);
result = vkGetEventStatus(LogicalDevice, IN event);
// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled

Note: the CPU cannot block waiting for an event, but it can test for one
```

Controlling Events from the Host

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

VkEvent event;
result = vkCreateEvent(LogicalDevice, IN &veci, PALLOCATOR, OUT &event);
result = vkSetEvent(LogicalDevice, IN event);
result = vkResetEvent(LogicalDevice, IN event);
result = vkGetEventStatus(LogicalDevice, IN event);
// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled
```

Note: the CPU cannot block waiting for an event, but it can test for one

Controlling Events from the Device

```
result = vkCmdSetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdResetEvent(CommandBuffer, IN event, pipelineStageBits);
result = vkCmdWaitEvents(CommandBuffer, IN event);
```

Could be an array of events

Where signaled: where what for the signal

Memory barriers get executed after events have been signaled

Note: the GPU cannot test for an event, but it can block waiting for one
We don't any one of these commands to have to wait on a previous command unless you say so. In general, we want all of these commands to be able to run "flat-out".

But, if we do that, surely there will be nasty race conditions! 

Potential Memory Race Conditions that Pipeline Barriers can Prevent

1. Write-then-Read (WR) – the memory write in one operation starts overwriting the memory that another operation’s read needs to use
2. Read-then-Write (RW) – the memory read in one operation hasn’t yet finished before another operation starts overwriting that memory
3. Write-then-Write (WW) – 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

vkCmdPipelineBarrier() Function Call

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

1. The cross-streets are named after pipeline stages
2. All traffic lights start out green ("we want all of these commands to be able to run flat-out")
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 dst traffic light will be turned red
5. When the last car in the src group completely makes it through its intersection, the 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), (4) the src cars clear their intersection, (5) the dst cars get released

Pipeline Stages

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

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

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

- VK_ACCESS_MEMORY_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT

Access Operations and what Pipeline Stages they can be used In

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
Example: Be sure we are done writing an output image before using it for something else.

- 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_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

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

- VK_ACCESS_MEMORY_WRITE_BIT
- VK_ACCESS_MEMORY_READ_BIT
- VK_ACCESS_HOST_WRITE_BIT
- VK_ACCESS_HOST_READ_BIT
- VK_ACCESS_TRANSFER_WRITE_BIT
- VK_ACCESS_TRANSFER_READ_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_UNIFORM_READ_BIT
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS_INDEX_READ_BIT
- VK_ACCESS_INDIRECT_COMMAND_READ_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT

Push Constants

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

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

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

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

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

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

where:

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

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

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

VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = vpcr;

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

Where each arm is represented by:

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

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

### An Robotic Example using Push Constants

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

Where each arm is represented by:

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

Hook the Pieces Together, Change Parameters, and Things Move
(All Young Children Understand This)

1. Rotate by $\theta_1$
2. Translate by $T_{1/G}$

$$[M_{1/G}] = [T_{1/G}] \cdot [R_{\theta_1}]$$

Soy it

Positioning Part #2 With Respect to Ground

1. Rotate by $\theta_2$
2. Translate the length of part 1
3. Rotate by $\theta_1$
4. Translate by $T_{1/G}$

$$[M_{3/G}] = [T_{1/G}] \cdot [R_{\theta_1}] \cdot [T_{2/1}] \cdot [R_{\theta_2}]$$

$$[M_{2/G}] = [M_{3/G}] \cdot [M_{2/1}]$$

Soy it

In the Reset Function

```c
struct arm
struct arm
struct arm
struct arm
Aarm1;
Aarm2;
Aarm3;
...
Aarm1.armMatrix = glm::mat4();
Aarm1.armColor  = glm::vec3(0.0f, 1.0f, 0.0f);
Aarm1.armScale  = 6.0f;
Aarm2.armMatrix = glm::mat4();
Aarm2.armColor  = glm::vec3(1.0f, 0.0f, 0.0f);
Aarm2.armScale  = 4.0f;
Aarm3.armMatrix = glm::mat4();
Aarm3.armColor  = glm::vec3(0.0f, 0.0f, 1.0f);
Aarm3.armScale  = 2.0f;
```

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

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

VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = vpcr;

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

In the UpdateScene Function

```cpp
// Time, arm scale
float rot1 = (float)Time;
float rot2 = 2.f * rot1;
float rot3 = 2.f * rot2;

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

glm::mat4 m1g = glm::mat4();
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);

glm::mat4 m21 = glm::mat4();
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));

glm::mat4 m32 = glm::mat4();
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));

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

In the RenderScene Function Without Pipeline Barriers

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

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

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

But, the problem is that
1. The vkCmdDraws must not start until the vkCmdPushConstants are done, and
2. The vkCmdPushConstants must not start until the vkCmdDraws are done

This is the type of problem that Pipeline Barriers were meant to solve.

Setting Up Global Memory Pipeline Barriers

```cpp
VkMemoryBarrier vmb;
vmb.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
vmb.pNext = nullptr;
vmb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
vmb.dstAccessMask = VK_ACCESS_HOST_READ_BIT;

vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, 1, &vmb, 0, nullptr);
```

Setting Up Buffer Memory Pipeline Barriers

```cpp
VkBufferMemoryBarrier vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
vbmb.dstAccessMask = VK_ACCESS_HOST_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer = NULL;
vbmb.offset = 0;
vbmb.size = 0;

vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, 1, &vbmb, 0, nullptr);
```

Setting Up Image Memory Pipeline Barriers

```cpp
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vimb.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_INPUT_ATTACHMENT_KHR;
vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_INPUT_ATTACHMENT_KHR;
vimb.srcQueueFamilyIndex = 0;
vimb.dstQueueFamilyIndex = 0;
vimb.image = NULL;
vimb.subresourceRange = VK_IMAGE_ASPECT_COLOR_BIT;

vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, 1, &vimb, 0, nullptr);
```
In the RenderScene function

\[
\text{VkBuffer buffers[1] = \{ MyVertexDataBuffer.buffer \};}
\]
\[
\text{vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);}
\]
\[
\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void*) &Arm1);}
\]
\[
\text{vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr);}
\]
\[
\text{vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);}
\]
\[
\text{vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr);}
\]
\[
\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void*) &Arm2);}
\]
\[
\text{vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr);}
\]
\[
\text{vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);}
\]
\[
\text{vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr);}
\]
\[
\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void*) &Arm3);}
\]
\[
\text{vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr);}
\]

In the Vertex Shader

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

layout (location = 0) in vec3 aVertex;

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

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

Aliasing

The Display We Want
Too often, the Display We Get

MultiSampling.pptx

Antialiasing and Multisampling

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

Anti-aliasing by Multisampling

4x 16x
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:

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

2. Multisampling: Perform a single color render for the one entire 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.

Note: per-sample depth and stencil tests are performed first to decide which color renders actually should be done.
Converting the multisampled image to a VK_SAMPLE_COUNT_1_BIT image

Resolving the Image:

Multipass Rendering

Mike Bailey
mb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Multipass Rendering uses Attachments —

What is a Vulkan Attachment Anyway?

"An attachment is an image associated with a renderpass that can be used as the input or output of one or more of its subpasses."

— Vulkan Programming Guide

An attachment can be written to, read from, or both.

For example:

```
Attachment
  
Subpass
   
Attachment
   
Subpass
   
Attachment
  
Subpass
  
Framebuffer
```

Back in Our Single-pass Days

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

```
Render
  
Framebuffer
```

Here comes a quick reminder of how we did that.

Afterwards, we will extend that.

```
Subpass #0
  
Attachment #0
   
Color Attachment
  
Attachment #1
   
Depth Attachment
  
Subpass #1
```

Multipass Rendering

So far, we’ve only performed single-pass rendering, but within a single Vulkan RenderPass, we can also have several subpasses, each of which is feeding information to the next subpass or subpasses.

In this case, we will look at following up a 3D rendering with some image processing on the outcome.

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

```
Original
  
Sharpened
  
Edge Detected
```

Back in Our Single-pass Days, II

```
Subpass #0
  
Attachment #0
   
ColorAttachment
  
Attachment #1
   
DepthAttachment
  
Attachment #2
```

```
Original Sharpened Edge Detected
  
No Noise
  
Noise
```

Notice how close this resembles a Directed Acyclic Graph (DAG) data structure: nodes connected by arrows that point in one direction.
Placing a Pipeline Barrier so an Image is not used before it is Ready

```cpp
VkAttachmentDescription:
  .format = VK_FORMAT_D32_SFLOAT_S8_UINT;
  .flags = 0;
  .finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
  .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  .stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .flags = 0;
  .finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
  .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  .stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  .samples = VK_SAMPLE_COUNT_1_BIT;
  .flags = 0;
  .finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC;
  .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  .stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  .storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  .loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  .samples = VK_SAMPLE_COUNT_1_BIT;
  .flags = 0;
```
Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

Which Pipeline State Variables can be Changed Dynamically

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

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

Creating a Pipeline

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;

Creating a Pipeline

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

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

Filling 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 );
null
Timestamp Query

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

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

```cpp
vkCmdCopyQueryPoolResults(CommandBuffer, timestampQueryPool, 0, 1,
buffer, 0, 0,
VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT);
```

An Introduction to the
Vulkan Computer Graphics API

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

This work is licensed under a Creative Commons
Attribution-NonCommercial-NoDerivatives 4.0
International License