Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

SIGGRAPH 2020 Abridged Version

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

Course Goals

• Give a sense of how Vulkan is different from OpenGL
• Show how to do basic drawing in Vulkan
• Leave you with working, documented, understandable sample code

Sections

1. Introduction
2. Sample Code
3. Drawing
4. Shaders and SPIR-V
5. Data Buffers
6. GLFW
7. GLM
8. Instancing
9. Graphics Pipeline Data Structure
10. Descriptor Sets
11. Textures
12. Queues and Command Buffers
13. Swap Chain
14. Push Constants
15. Physical Devices
16. Logical Devices
17. Dynamic State Variables
18. Getting Information Back
19. Compute Shaders
20. Specialization Constants
21. Synchronization
22. Pipeline Barriers
23. Multisampling
24. Multiswap
25. Ray Tracing

Section titles that have been greyed-out have not been included in the ABRIDGED noteset, i.e., the one that has been made to fit in SIGGRAPH’s reduced time slot. These topics are in the FULL noteset, however, which can be found on the web page:
http://cs.oregonstate.edu/~mjb/vulkan

My Favorite Vulkan Reference

Introduction

Mike Bailey
mjb@cs.oregonstate.edu

Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

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

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

Vulkan 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-vertex-end, fixed-function, etc.
- You must manage your own transformations.
- All transformation, color and texture functionality must be done in shaders.
- Shaders are pre-half-compiled outside of your application. The compilation process is then finished during the runtime pipeline building process.

Vulkan Highlights: Pipeline State Data Structure

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

Vulkan Highlights: Pipeline State Data Structure

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

Querying the Number of Something

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

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

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

```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)null );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );
```

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

Application

Instance

Physical Device

Logical Device

Command Buffer

Steps in Creating Graphics using Vulkan

1. Create the Vulkan Instance
2. Setup the Debug Callbacks
3. Create the Surface
4. List the Physical Devices
5. Pick the right Physical Device
6. Create the Logical Device
7. Create the Uniform Variable Buffers
8. Create the Vertex Data Buffers
9. Create the texture sampler
10. Create the texture images
11. Create the Swap Chain
12. Create the Depth and Stencil Images
13. Create the RenderPass
14. Create the Framebuffer
15. Create the Command Buffer Pool
16. Create the Command Buffer Pool
17. Create the Command BufferPool
18. Read the shaders
19. Create the Descriptor Set Layout
20. Create and populate the Descriptor Set
21. Create the Graphics Pipeline
22. Update-Render-Update-Render ...

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjb@cs.oregonstate.edu

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

Sample Program Keyboard Inputs

'T', 'L': Toggle lighting off and on
'n', 'N': Toggle display mode (textures vs. colors, for now)
'y', 'p': Pause the animation
'y', 'q': quit the program
Esc: quit the program
'y', 'r': Toggle rotation-animation and using the mouse
't', 'T': Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)
'1', '4', '9' set the number of instances (in the instancing version)

Caveats on the Sample Code, I

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

Caveats on the Sample Code, II

6. There are great 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. I hope it is cleaner this way.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

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

Vulkan Conventions

My Conventions

"Init" in a function call name means that something is being setup that only needs to be setup once

The number after "Init" gives you the ordering

In the source code, after main( ) comes InitGraphics( ) and all of the InitXXX( ) functions in numerical order. After that comes the helper functions

"Find" in a function call name means that something is being looked for

"Fill" in a function call name means that some data is being supplied to Vulkan

"IN" and "OUT" ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are actually #define'd to nothing.
Double-click here to launch Visual Studio 2019 with this solution

```cpp
struct errorcode {
    VkResult resultCode;
    std::string meaning;
}

ErrorCodes[] = {
    { VK_NOT_READY, "Not Ready" },
    { VK_TIMEOUT, "Timeout" },
    { VK_EVENT_SET, "Event Set" },
    { VK_EVENT_RESET, "Event Reset" },
    { VK_INCOMPLETE, "Incomplete" },
    { VK_ERROR_OUT_OF_HOST_MEMORY, "Out of Host Memory" },
    { VK_ERROR_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" },
    { VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" },
    { VK_ERROR_DEVICE_LOST, "Device Lost" },
    { VK_ERROR_MEMORY_MAP_FAILED, "Memory Map Failed" },
    { VK_ERROR_LAYER_NOT_PRESENT, "Layer Not Present" },
    { VK_ERROR_EXTENSION_NOT_PRESENT, "Extension Not Present" },
    { VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" },
    { VK_ERROR_INCOMPATIBLE_DRIVER, "Incompatible Driver" },
    { VK_ERROR_TOO_MANY_OBJECTS, "Too Many Objects" },
    { VK_ERROR_FORMAT_NOT_SUPPORTED, "Format Not Supported" },
    { VK_ERROR_FRAGMENTED_POOL, "Fragmented Pool" },
    { VK_ERROR_SURFACE_LOST_KHR, "Surface Lost" },
    { VK_ERROR_NATIVE_WINDOW_IN_USE_KHR, "Native Window in Use" },
    { VK_SUBOPTIMAL_KHR, "Suboptimal" },
    { VK_ERROR_OUT_OF_DATE_KHR, "Error Out of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
};
```

### Reporting Error Results, II

```cpp
void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s
", prefix.c_str(), "Successful" );
        fflush(FpDebug);
        return;
    }

    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = ""
    for( int i = 0; i < numErrorCodes; i++ )
    {
        if( result == ErrorCodes[i].resultCode )
        {
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }
    fprintf( FpDebug, "%s: %s
", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```

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

### Extras in the Code

```cpp
Vulkan Topologies
```

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

Vulkan Topologies

A Colored Cube Example

Triangles Represented as an Array of Structures

Non-indexed Buffer Drawing

Stream of Vertices

Filling the Vertex Buffer
We will come to Command Buffers later, but for now, know that a Vulkan Pipelines is essentially a very large data structure that holds what OpenGL would call the state, including how to process its input.

**C/C++**:
```c
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets);
```

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

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 process its input.

**C/C++**:
```c
const uint32_t firstInstance = 0;
const uint32_t firstVertex = 0;
```

**GLSL Shader**:
```glsl
texCoord = texCoord + (vec2(0.0, 1.0) * texturePosition);
```

**Drawing with an Index Buffer**
```c
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance, 0 );
```

**Vertex Lookup**
```
       |       |
       +-----------------------------------> index
       |
       |       |       |       |       |       |       |       |
       0      1      2      3      4      5      6      7
       |       |       |       |       |       |       |       |
       ^       ^       ^       ^       ^       ^       ^       ^
```

**Stream of Vertices**
```
   0  1  2  3  4  5  6  7
   |   |   |   |   |   |   |   |
   V   V   V   V   V   V   V   V
```

**Stream of Indices**
```
   0  1  2  3  4  5  6  7
   |   |   |   |   |   |   |   |
   V   V   V   V   V   V   V   V
```

**Stream of Indices**
```
   0  1  2  3  4  5  6  7
   |   |   |   |   |   |   |   |
   V   V   V   V   V   V   V   V
```
### Drawing with an Index Buffer

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

### Sometimes the Same Point Needs Multiple Attributes

- Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is on. Sometimes it doesn’t.
- A color-interpolated cube like this actually has both. Point #7 above has the same color, regardless of what face #7 is on. However, Point #7 has 3 different normal vectors, depending on which face you are defining. Same with its texture coordinates.
- Thus, when using indexed 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.

### 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

```cpp
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
```

### How Vulkan GLSL Differs from OpenGL GLSL

- **Vulkan Vertex and Instance indices:**
  - `gl_VertexIndex`
  - `gl_InstanceIndex`
  - Both are 0-based

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

### Vulkan: Shaders’ use of Layouts for Uniform Variables

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

### Vulkan Shader Compiling

- You half-precompile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V, which stands for Standard Portable Intermediate Representation.
- SPIR-V gets turned into fully-compiled code at runtime, when the pipeline structure is finally created.
- The SPIR-V spec has been public for a few years – new shader languages are surely being developed.
- OpenGL and OpenCL have now adopted SPIR-V as well

### Advantages:

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


Shaderfile extensions:
.vert Vertex
.tesc Tessellation Control
.tese 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:        glslangValidator

Running glslangValidator.exe

Running glslangValidator.exe

Running glslangValidator.exe

Running glslangValidator.exe

How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of 0x07230203
So, if you do an od –x on the .spv file, the magic number looks like this:
0203 0723 . . .

How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of 0x07230203
So, if you do an od –x on the .spv file, the magic number looks like this:
0203 0723 . . .
The shaderc project from Google (https://github.com/google/shaderc) provides a glslangValidator wrapper program called `glslc` that has a much improved command-line interface. You use, basically, the same way:

```
glslc.exe --target-env=vulkan sample-vert.vert -o sample-vert.spv
```

There are several really nice features. The two I really like are:

1. You can #include files into your shader source
2. You can "#define" definitions on the command line like this:

```
glslc.exe --target-env=vulkan -DNUMPOINTS=4 sample-vert.vert -o sample-vert.spv
```

glslc is included in your Sample .zip file

---

**Data Buffers**

Mike Bailey

mbj@cs.oregonstate.edu

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

---

A Vulkan **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 sometimes 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;
```

This is probably a bad idea in the long run.

---

**Creating and Filling Vulkan Data Buffers**

```
VkBuffer Buffer;
VkBufferCreateInfo vbci;

vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or'ed bits of: >>
                VK_USAGE_TRANSFER_SRC_BIT
                VK_USAGE_TRANSFER_DST_BIT
                VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
                VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
                VK_USAGE_UNIFORM_BUFFER_BIT
                VK_USAGE_STORAGE_BUFFER_BIT
                VK_USAGE_INDEX_BUFFER_BIT
                VK_USAGE_VERTEX_BUFFER_BIT
                VK_USAGE_INDIRECT_BUFFER_BIT

vbci.sharingMode = << one of: >>
                   VK_SHARING_MODE_EXCLUSIVE
                   VK_SHARING_MODE_CONCURRENT

vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

---

**Creating a Vulkan Data Buffer**

```
VkBuffer Buffer;

VkBufferCreateInfo vbci;

dcType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
dcflags = 0;
dcsize = << buffer size in bytes >>
dcusage = <<or'ed buff usage in bytes >>

dcUsage = VK_USAGE_TRANSFER_SRC_BIT
dcUsage = VK_USAGE_TRANSFER_DST_BIT
dcUsage = VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
dcUsage = VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
dcUsage = VK_USAGE_UNIFORM_BUFFER_BIT
dcUsage = VK_USAGE_STORAGE_BUFFER_BIT
dcUsage = VK_USAGE_INDEX_BUFFER_BIT
dcUsage = VK_USAGE_VERTEX_BUFFER_BIT
dcUsage = VK_USAGE_INDIRECT_BUFFER_BIT

dcSharingMode = VK_SHARING_MODE_EXCLUSIVE
dcSharingMode = VK_SHARING_MODE_CONCURRENT

result = vkBufferData ( LogicalDevice, IN &Buffer, PALLOCATOR, OUT &Buffer );
```
Allocating Memory for a Vulkan Data Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

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

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

result = vkMapMemory(LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr);
```

Finding the Right Type of Memory

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

Finding the Right Type of Memory

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

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. I don’t have experience using it yet, so I’m not in a position to confidently comment on it. But, I am including its github link here and a little sample code in case you want to take a peek.

https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

This repository includes a smattering of documentation.
**typedef struct MyBuffer**

- VkDataBuffer `buffer`;
- VkDeviceMemory `vdm`;
- VkDeviceSize `size`;

**MyBuffer MyMatrixUniformBuffer;**

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

It also makes it impossible to accidentally associate the wrong VkDeviceMemory and/or VkDeviceSize with the wrong data buffer.

**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;
```

MyBuffer MyMatrixUniformBuffer;

**Initializing a Data Buffer**

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

```c
intInit05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer)
{
    ...
    vbci.size = pMyBuffer->size;
    ...
    result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer);
    ...
    pMyBuffer->vdm = vdm;
    ...
}
```

**Here's a C struct used by the Sample Code to hold some uniform variables**

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

**Here's the associated GLSL shader code to access those uniform variables**

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

**Filling those Uniform Variables**

```c
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle in radians
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat 4( 1. );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan's LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

This code assumes that this line:

```c
#define    GLM_FORCE_RADIANS
```

is listed before GLM is included!

**The Parade of Buffer Data**

The MyBuffer does not hold any actual data itself. It just information about what is in the data buffer.

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

**Filling the Data Buffer**

```c
Init05UniformBuffer( sizeof(Matrices), OUT &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer, IN (void *) &Matrices );
```
Creating and Filling the Data Buffer – the Details

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

Creating and Filling the Data Buffer – the Details

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

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

GLFW

```
#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"
...
uint32_t Width, Height;
VkSurfaceKHR Surface;
...
void InitGLFW()
{
    glfwInit();
    if( !glfwVulkanSupported() )
    {
        fprintf(stderr, "Vulkan is not supported on this system!
        exit(1);
    }
    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, OUT &Surface);
    glfwSetErrorCallback(GLFWErrorCallback);
    glfwSetKeyCallback(MainWindow, GLFWKeyboard);
    glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
    glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

You Can Also Query What Vulkan Extensions GLFW Requires

```
uint32_t count;
const char ** extensions = glfwGetRequiredInstanceExtensions(&count);
printf(FpDebug, "Found %d GLFW Required Instance Extensions:
    ", count);
for(uint32_t i = 0; i < count; i++)
{
    printf(FpDebug, 	"%s
", extensions[i]);
}
```

GLFW Keyboard Callback

```
void GLFWKeyboard(GLFWwindow * window, int key, int scancode, int action, int mods)
{
    if(action == GLFW_PRESS)
    {
        switch(key)
        {
            case GLFW_KEY_M:
            case 'm':
            case 'M':
                Mode++;
                if(Mode >= 2)
                    Mode = 0;
                break;
            default:
                printf(FpDebug, "Unknown key Hit! 0x%04x = \"%c\", key;\n                    Mod(key);"
        }
    }
}
```
### GLFW Mouse Button Callback

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

### GLFW Mouse Motion Callback

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

### Looping and Closing GLFW

```c
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
  glfwPollEvents();
  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. Even though it was written for OpenGL, it works fine with Vulkan.

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

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

You invoke GLM like this:

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

OpenGL treats all angles as given in degrees. You force GLM to treat all angles as given in radians. I recommend this so that all angles you create in all programming will be in radians.

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

---

**GLM**

Mike Bailey
mjb@cs.oregonstate.edu

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

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    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 say:

```
    glm::mat4 modelview = glm::mat4(1.); // identity
    glm::vec3 eye(0., 0., 3.);
    glm::vec3 look(0., 0., 0.);
    glm::vec3 up(0., 1., 0.);
    modelview = glm::lookAt(eye, look, up); // \{x',y',z'} = \{v\}*\{x,y,z\}
    modelview = glm::rotate(modelview, D2R*Yrot, glm::vec3(0.,1.,0.)); // \{x',y',z'} = \{v\}*\{yr\}*\{x,y,z\}
    modelview = glm::rotate(modelview, D2R*Xrot, glm::vec3(1.,0.,0.)); // \{x',y',z'} = \{v\}*\{yr\}*\{xr\}*\{x,y,z\}
    modelview = glm::scale(modelview, glm::vec3(Scale,Scale,Scale)); // \{x',y',z'} = \{v\}*\{yr\}*\{xr\}*\{s\}*\{x,y,z\}
```

This is exactly the same concept as OpenGL, but a different expression of it. Read on for details …

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.

### Constructors:

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

### Multiplications:

- `glm::mat4 * glm::mat4`
- `glm::mat4 * glm::vec4` (promote a vec3 to a vec4 via a constructor)
- `glm::mat4 * glm::vec4(glm::vec3, 1.0)` // promote a vec3 to a vec4 via a constructor

### Emulating OpenGL transformations with concatenation:

- `glm::mat4 glm::rotate(glm::mat4 const & m, float angle, glm::vec3 const & axis)`
- `glm::mat4 glm::scale(glm::mat4 const & m, glm::vec3 const & factors)`
- `glm::mat4 glm::translate(glm::mat4 const & m, glm::vec3 const & translation)`

### Viewing Volume (assign, not concatenate):

- `glm::mat4 glm::ortho(float left, float right, float bottom, float top)`
- `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

The Most Useful GLM Variables, Operations, and Functions

Installing GLM into your own space

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

GLM in the Vulkan sample.cpp Program

Here's what that GLM folder looks like

GLM in the Vulkan sample.cpp Program

GLM in the Vulkan sample.cpp Program

GLM in the Vulkan sample.cpp Program
Sidebar: Why Isn’t The Normal Matrix exactly the same as the Model Matrix?

- If the Model Matrix is all rotations and uniform scalings, then it is not. These diagrams show you why.

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

Original object and normal

Right!

Wrong!

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

Instancing

Mike Bailey
mj@cs.oregonstate.edu

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

Instancing – What and why?

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

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

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

BTW, when not using instancing, be sure the `instanceCount` is 1, not 0!

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
```

```
out vec3 vColor;
const int NUMINSTANCES = 16;
const float DELTA = 3.0;

float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
```

In the vertex shader:

---

Making each Instance look differently – Approach #2

Put the unique characteristics in a uniform buffer array and reference them

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

```
int index = gl_InstanceIndex % 1024; // or '& 1023' – gives 0 - 1023
vColor = Colors.uColors[index];
```

In the vertex shader:

---
The Graphics Pipeline Data Structure

Mike Bailey
mjb@cs.oregonstate.edu

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

A Pipeline Data Structure Contains the Following State Items:

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissoring: x, y, w, h
- Rasterization: cullMode, polygonMode, frontface, lineWidth
- Depth: depthEnable, depthWriteEnable, depthCompareOp
- Blend: blendEnable, blendFactor, blendOp, blendEquation

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

### The First Step: Create the Graphics Pipeline Layout

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

```c
result = vkCreatePipelineLayout(
    &GraphicsPipelineLayout, 
    PALLOCATOR, OUT
)
```

Why is this necessary? It is because the Descriptor Sets and Push Constants data structures have different sizes depending on how many of each you have. So, the exact structure of the Pipeline Layout depends on you telling Vulkan about the Descriptor Sets and Push Constants that you will be using.

### Creating a Graphics Pipeline from a lot of Pieces

Creating a Graphics Pipeline involves specifying different stages, such as Vertex Input, Rasterization, and Color Blending, and setting various state items for each stage.
Creating a Typical Graphics Pipeline

Options for vpiasci.topology

The Shaders to Use

What is ”Primitive Restart Enable”?
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

Triangle Strip #0:

Triangle Strip #1:

...
VkPipelineMultisampleStateCreateInfo
vpmsci
{
    .sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
    .pNext = nullptr;
    .flags = 0;
    .rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
    .sampleShadingEnable = VK_FALSE;
    .minSampleShading = 0;
    .pSampleMask = (VkSampleMask *)nullptr;
    .alphaToCoverageEnable = VK_FALSE;
    .alphaToOneEnable = VK_FALSE;
}

Declare information about how the multisampling will take place

$\text{MultiSampling State}$

We will discuss MultiSampling in a separate noteset.

VkPipelineColorBlendAttachmentState
vpcbas
{
    .blendEnable = VK_FALSE;
    .srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
    .dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    .colorBlendOp = VK_BLEND_OP_ADD;
    .srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    .dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
    .alphaBlendOp = VK_BLEND_OP_ADD;
    .colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
}

Color Blending State for each Color Attachment *

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

This controls blending between the output of each color attachment and its image memory.

$\text{Colour}_\text{new} = (1 - \alpha) \ast \text{Colour}_{\text{existing}} + \alpha \ast \text{Colour}_{\text{incoming}}$

$0 \leq \alpha \leq 1.$

* A "Color Attachment" is a framebuffer to be rendered into. You can have as many of these as you want.

VkPipelineColorBlendStateCreateInfo
vpcbsci
{
    .sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    .pNext = nullptr;
    .flags = 0;
    .logicOpEnable = VK_FALSE;
    .logicOp = VK_LOGIC_OP_COPY;
    #ifdef CHOICES
    VK_LOGIC_OP_CLEAR
    VK_LOGIC_OP_AND
    VK_LOGIC_OP_AND_REVERSE
    VK_LOGIC_OP_COPY
    VK_LOGIC_OP_AND_INVERTED
    VK_LOGIC_OP_NO_OP
    VK_LOGIC_OP_XOR
    VK_LOGIC_OP_OR
    VK_LOGIC_OP_NOR
    VK_LOGIC_OP_EQUIVALENT
    VK_LOGIC_OP_INVERT
    VK_LOGIC_OP_OR_REVERSE
    VK_LOGIC_OP_COPY_INVERTED
    VK_LOGIC_OP_OR_INVERTED
    VK_LOGIC_OP_NAND
    VK_LOGIC_OP_SET
    #endif
    .attachmentCount = 1;
    .pAttachments = &vpcbas;
    .blendConstants[0] = 0;
    .blendConstants[1] = 0;
    .blendConstants[2] = 0;
    .blendConstants[3] = 0;
}

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

Raster Operations for each Color Attachment

VkDynamicState
vds
[ ] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };

#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT -- vkCmdSetViewport( )
VK_DYNAMIC_STATE_SCISSOR -- vkCmdSetScissor( )
VK_DYNAMIC_STATE_LINE_WIDTH -- vkCmdSetLineWidth( )
VK_DYNAMIC_STATE_DEPTH_BIAS -- vkCmdSetDepthBias( )
VK_DYNAMIC_STATE_BLEND_CONSTANTS -- vkCmdSetBendConstants( )
VK_DYNAMIC_STATE_DEPTH_BOUNDS -- vkCmdSetDepthZBounds( )
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK -- vkCmdSetStencilCompareMask( )
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK -- vkCmdSetStencilWriteMask( )
VK_DYNAMIC_STATE_STENCIL_REFERENCE -- vkCmdSetStencilReference( )
#endif

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

Which Pipeline Variables can be Set Dynamically

Just used as an example in the Sample Code

The Stencil Buffer

1. While drawing into the Render Buffer, you can write values into the Stencil Buffer at the same time.
2. While drawing into the Render Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.
3. When drawing into the Render Buffer, you can write-protect certain parts of the Render Buffer based on values that are in the Stencil Buffer

Here's how the Stencil Buffer works:

Using the Stencil Buffer to Create a Magic Lens
Using the Stencil Buffer to Create a Magic Lens

1. Clear the SB = 0
2. Write-protect the color buffer
3. Fill a square, setting SB = 1
4. Write-enable the color buffer
5. Draw the solids wherever SB == 0
6. Draw the wireframes wherever SB == 1

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting

Using the Stencil Buffer to Better Outline Polygons

Clear the SB = 0

for each polygon {
    Draw the edges, setting SB = 1
    Draw the polygon wherever SB != 1
    Draw the edges, setting SB = 0
}

Using the Stencil Buffer to Perform Hidden Line Removal

```
VkStencilOpState vsosf;
// front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds
#ifdef CHOICES
    VK_STENCIL_OP_KEEP -- keep the stencil value as it is
    VK_STENCIL_OP_ZERO -- set stencil value to 0
    VK_STENCIL_OP_REPLACE -- replace stencil value with the reference value
    VK_STENCIL_OP_INCREMENT_AND_CLAMP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_CLAMP -- decrement stencil value
    VK_STENCIL_OP_INVERT -- bit-invert stencil value
    VK_STENCIL_OP_INCREMENT_AND_WRAP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_WRAP -- decrement stencil value
#endif
vsosf.compareOp = VK_COMPARE_OP_NEVER;
#ifdef CHOICES
    VK_COMPARE_OP_NEVER -- never succeeds
    VK_COMPARE_OP_LESS -- succeeds if stencil value is < the reference value
    VK_COMPARE_OP_EQUAL -- succeeds if stencil value is == the reference value
    VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if stencil value is <= the reference value
    VK_COMPARE_OP_GREATER -- succeeds if stencil value is > the reference value
    VK_COMPARE_OP_NOT_EQUAL -- succeeds if stencil value is != the reference value
    VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if stencil value is >= the reference value
    VK_COMPARE_OP_ALWAYS -- always succeeds
#endif
vsosf.compareMask = ~0;
vsosf.writeMask = ~0;
vsosf.reference = 0;
```

VkStencilOpState vsosb;
// back
vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
vsosb.failOp = VK_STENCIL_OP_KEEP;
vsosb.passOp = VK_STENCIL_OP_KEEP;
vsosb.compareOp = VK_COMPARE_OP_NEVER;
vsosb.compareMask = ~0;
vsosb.writeMask = ~0;
vsosb.reference = 0;

Stencil Operations for Front and Back Faces

- VK_STENCIL_OP_KEEP: keep the stencil value as it is
- VK_STENCIL_OP_ZERO: set stencil value to 0
- VK_STENCIL_OP_REPLACE: replace stencil value with the reference value
- VK_STENCIL_OP_INCREMENT_AND_CLAMP: increment stencil value
- VK_STENCIL_OP_DECREMENT_AND_CLAMP: decrement stencil value
- VK_STENCIL_OP_INVERT: bit-invert stencil value
- VK_STENCIL_OP_INCREMENT_AND_WRAP: increment stencil value
- VK_STENCIL_OP_DECREMENT_AND_WRAP: decrement stencil value
- VK_COMPARE_OP_NEVER: never succeeds
- VK_COMPARE_OP_LESS: succeeds if stencil value is < the reference value
- VK_COMPARE_OP_EQUAL: succeeds if stencil value is == the reference value
- VK_COMPARE_OP_LESS_OR_EQUAL: succeeds if stencil value is <= the reference value
- VK_COMPARE_OP_GREATER: succeeds if stencil value is > the reference value
- VK_COMPARE_OP_NOT_EQUAL: succeeds if stencil value is != the reference value
- VK_COMPARE_OP_GREATER_OR_EQUAL: succeeds if stencil value is >= the reference value
- VK_COMPARE_OP_ALWAYS: always succeeds
vkPipelineDepthStencilStateCreateInfo vpdssci = {0};
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;

VkPipeline GraphicsPipeline vgpci = {0};
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
vgpci.stageCount = 2;                           // number of stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;

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

later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

Sidebar: What is the Organization of the Pipeline Data Structure?

If you take a close look at the pipeline data structure creation information, you will see that almost all the pieces have a fixed size. For example, the viewport only needs 6 pieces of information – ever:

vkViewport

vv.x = 0;
vv.y = 0;
vv.width = (float)Width;
vv.height = (float)Height;
vv.minDepth = 0.0f;
vv.maxDepth = 1.0f;

There are two exceptions to this – the Descriptor Sets and the Push Constants. Each of these two can be almost any size, depending on what you allocate for them. So, I think of the Pipeline Data Structure as consisting of some fixed-layout blocks and 2 variable-layout blocks, like this:

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

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

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

Vulkan

Descriptor Sets

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
What are Descriptor Sets?

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

• Related uniform variables can be compartmentalized into a group, gaining efficiency.

• Descriptor Sets are activated when the Command Buffer is filled. Different values for the uniform buffer variables can be toggled by just swapping out the Descriptor Set that points to GPU memory, rather than re-writing the GPU memory.

• Values for the shaders’ uniform buffer variables can be compartmentalized into what quantities change often and what change seldom (scene-level, model-level, draw-level), so that uniform variables need to be re-written no more often than is necessary.

Our example will assume the following shader uniform variables:

- 1 non-opaque material must be in a uniform block:
  - layout( set = 3, binding = 0 ) uniform matBuf
    - mat uLightMat;
    - mat uShadowMat;
    - mat uReflectionMat;
    - mat uRuminationMat;
  - Matrices

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

- 1 non-opaque material must be in a uniform block:
  - layout( set = 2, binding = 0 ) uniform miscBuf
    - int uMode;
    - Misc

- All texture sampler variables must be in a uniform block:
  - layout( set = 0, binding = 0 ) uniform sampler2D uSampler;
  - sampler2D uSampler;

Uniform data created in a C++ data structure

Uniform data used in the shader

CPU:

• Knows the data’s size
• Knows where the data starts
• Knows the CPU data structure

GPU:

• Does not know the CPU or GPU data structure
• Knows the data’s size
• Knows where the data starts

GPU:

Step 1: Descriptor Set Pools

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

Step 2: Define the Descriptor Set Layouts

Think of Descriptor Set Layouts as a level of “Roseate Stone” that allows the Graphics Pipeline data structure to allocate room for the uniform variables and to access them.

Uniform data in a “blob”

uniform sampler2D usampler;

Uniform data created in a C++ data structure

uniform vec4 uLightPos;

Uniform data used in the shader

uniform mat4 uModelMatrix;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform mat3 uNormalMatrix;

uniform vec4 uLightPos;
Step 2: Define the Descriptor Set Layouts

```c
VkDescriptorSetLayoutCreateInfo vdslc0,
VkDescriptorSetLayoutCreateInfo vdslc1,
VkDescriptorSetLayoutCreateInfo vdslc2,
VkDescriptorSetLayoutCreateInfo vdslc3;

vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vdslc0.bindingCount = 0;
vdslc1.bindingCount = 0;
vdslc2.bindingCount = 0;
vdslc3.bindingCount = 0;

vdslc0.pNext = nullptr;
vdslc1.pNext = nullptr;
vdslc2.pNext = nullptr;
vdslc3.pNext = nullptr;

vdslc0.pBindings = nullptr;
vdslc1.pBindings = nullptr;
vdslc2.pBindings = nullptr;
vdslc3.pBindings = nullptr;
```

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

```c
VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.setLayoutCount = 4;
vplci.pNext = nullptr;
vplci.pLayouts = &DescriptorSetLayouts[0];
```

Step 4: Allocating the Memory for Descriptor Sets

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

---

Step 4: Allocating the Memory for Descriptor Sets

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

---

Step 2: Define the Descriptor Set Layouts

```c
VkDescriptorSetLayoutCreateInfo vdslc0,
VkDescriptorSetLayoutCreateInfo vdslc1,
VkDescriptorSetLayoutCreateInfo vdslc2,
VkDescriptorSetLayoutCreateInfo vdslc3;

vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vdslc0.bindingCount = 0;
vdslc1.bindingCount = 0;
vdslc2.bindingCount = 0;
vdslc3.bindingCount = 0;

vdslc0.pNext = nullptr;
vdslc1.pNext = nullptr;
vdslc2.pNext = nullptr;
vdslc3.pNext = nullptr;
```

---

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

```c
VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.setLayoutCount = 4;
vplci.pNext = nullptr;
vplci.pLayouts = &DescriptorSetLayouts[0];
```

---

Step 4: Allocating the Memory for Descriptor Sets

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

---

Step 2: Define the Descriptor Set Layouts

```c
VkDescriptorSetLayoutCreateInfo vdslc0,
VkDescriptorSetLayoutCreateInfo vdslc1,
VkDescriptorSetLayoutCreateInfo vdslc2,
VkDescriptorSetLayoutCreateInfo vdslc3;

vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;

vdslc0.bindingCount = 0;
vdslc1.bindingCount = 0;
vdslc2.bindingCount = 0;
vdslc3.bindingCount = 0;

vdslc0.pNext = nullptr;
vdslc1.pNext = nullptr;
vdslc2.pNext = nullptr;
vdslc3.pNext = nullptr;
```

---

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

```c
VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.setLayoutCount = 4;
vplci.pNext = nullptr;
vplci.pLayouts = &DescriptorSetLayouts[0];
```
Any collection of Descriptor Sets that match that structure can be bound into that pipeline. So, the Pipeline Layout contains the structure of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.

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

- **VkDescriptorBufferInfo**
  - `vdbi1.buffer` = MyMiscUniformBuffer, `vdbi1.offset` = 0, `vdbi1.range` = sizeof(Misc)
  - `vdbi2.buffer` = MyLightUniformBuffer, `vdbi2.offset` = 0, `vdbi2.range` = sizeof(Light)

- **VkDescriptorImageInfo**
  - `vdii.imageLayout` = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  - `vdii.sampler` = MyPuppyTexture.texSampler

- **VkDescriptorBufferInfo**
  - `vdbi3.buffer` = MyMiscUniformBuffer, `vdbi3.offset` = 0, `vdbi3.range` = sizeof(Misc)

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

`vkCreatePipelineLayout()` is used to allocate the layout, and then it is passed to `vkCreateGraphicsPipelines()` to create the pipeline. The pipeline then runs, rendering a scene using the newly created pipeline layout.

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

- **vkCmdBindDescriptorSets()**
  - Bind the Descriptor Sets to the command buffer for rendering.

**Step 8: The Entire Collection of Descriptor Set Paths**

- **AllocateDescriptorSets()**
  - Allocate memory for the Descriptor Sets

- **WriteDescriptorSets()**
  - Write CPU data into a particular Descriptor Set

- **WriteDescriptorSet()**
  - Take a particular Descriptor Set and use it in a specific Pipeline layout

- **AllocateDescriptorSets()**
  - Allocate memory for the Descriptor Sets

- **Re-write CPU data into a particular Descriptor Set

- **AllocateDescriptorSets()**
  - Allocate memory for the Descriptor Sets

- **AllocateDescriptorSets()**
  - Allocate memory for the Descriptor Sets
The pieces of the Pipeline Data Structure are fixed in size—with the exception of the Descriptor Sets and the Push Constants. Each of these two can be any size, depending on what you allocate for them. So, the Pipeline Data Structure needs to know how these two are configured before it can set its own total layout.

Think of the DS layout as being a particular-sized hole in the Pipeline Data Structure. Any data you have that matches this hole’s shape and size can be plugged in there.

The Pipeline Data Structure

Fixed Pipeline Elements

Specific Descriptor Set Layout

Any set of data that matches the Descriptor Set Layout can be plugged in there.

Textures

Mike Bailey
mjb@cs.oregonstate.edu

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

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

Intel Integrated Graphics:
2 Memory Types:
Memory 5: DeviceLocal
Memory 1: DeviceLocal HostVisible HostCoherent
Memory 2: DeviceLocal HostVisible HostCoherent HostCached

Triangles in an Array of Structures
In addition to just picking one mipmap level, the rendering system can sample from two of them, one less than the T:P ratio and one more, and then blend the two RGBA's returned. This is known as **VK_SAMPLER_MIPMAP_MODE_LINEAR**.

* Latin motto: “multa in parvo: many things in a small place”

---

**Textures’ Undersampling Artifacts**

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 left over 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.

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 never been included in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets included somehow in all resolution-level textures.
vkMapMemory
void * gpuMemory;
{
    else
    {
        VkCommandBufferBeginInfo vcbbi;
        // copy pixels from the staging image to the texture:
            // *******************************************************************************
            // *******************************************************************************
            // *******************************************************************************
        {
            // *******************************************************************************
            // *******************************************************************************
            // *******************************************************************************
        }

        VkImageCopy vic;
        vkCmdCopyImage(TextureCommandBuffer
        vic.extent = ve3;
        vic.dstOffset = vo3;
        vic.srcSubresource = visl;

        VkImageMemoryBarrier vimb;
        vimb.dstAccessMask = 0;
        vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;

        visr.layerCount = 1;
        visr.baseMipLevel = 0;
        1, IN &vimb );

        vpai.memoryTypeIndex =
        vmai.allocationSize = vmr.size;
        vmai.pNext = nullptr;
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;

        vkQueueWaitIdle
        vkQueueSubmit
        vkAllocateMemory
        vkMapMemory
        void * gpuMemory;
        {
            else
            {
                VkCommandBufferBeginInfo vcbbi;
                // copy pixels from the staging image to the texture:
                        // *******************************************************************************
                        // *******************************************************************************
                        // *******************************************************************************
                {
                        // *******************************************************************************
                        // *******************************************************************************
                        // *******************************************************************************
                }

                VkImageCopy vic;
                vkCmdCopyImage(TextureCommandBuffer
                vic.extent = ve3;
                vic.dstOffset = vo3;
                vic.srcSubresource = visl;

                VkImageMemoryBarrier vimb;
                vimb.dstAccessMask = 0;
                vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;

                visr.layerCount = 1;
                visr.baseMipLevel = 0;
                1, IN &vimb );

                vpai.memoryTypeIndex =
                vmai.allocationSize = vmr.size;
                vmai.pNext = nullptr;
                vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;

                vkQueueWaitIdle
                vkQueueSubmit
                vkAllocateMemory
                vkMapMemory
                void * gpuMemory;
                {
                    else
                    {
                        VkCommandBufferBeginInfo vcbbi;
                        // copy pixels from the staging image to the texture:
                                // *******************************************************************************
                                // *******************************************************************************
                                // *******************************************************************************
                        {
                                // *******************************************************************************
                                // *******************************************************************************
                                // *******************************************************************************
                        }

                        VkImageCopy vic;
                        vkCmdCopyImage(TextureCommandBuffer
                        vic.extent = ve3;
                        vic.dstOffset = vo3;
                        vic.srcSubresource = visl;

                        VkImageMemoryBarrier vimb;
                        vimb.dstAccessMask = 0;
                        vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;

                        visr.layerCount = 1;
                        visr.baseMipLevel = 0;
                        1, IN &vimb );

                        vpai.memoryTypeIndex =
                        vmai.allocationSize = vmr.size;
                        vmai.pNext = nullptr;
                        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;

                        vkQueueWaitIdle
                        vkQueueSubmit
                        vkAllocateMemory
                        vkMapMemory
                        void * gpuMemory;
                        {
                            else
                            {
                                VkCommandBufferBeginInfo vcbbi;
                                // copy pixels from the staging image to the texture:
                                            // *******************************************************************************
                                            // *******************************************************************************
                                            // *******************************************************************************
                            }
To create an image view for the texture image:

```c
// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageViewCreateInfo vivci;
vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
vivci.pNext = nullptr;
vivci.flags = 0;
vivci.image = textureImage;
vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
vivci.components.r = VK_COMPONENT_SWIZZLE_R;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.subresourceRange = visr;
result = vkCreateImageView(
    LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
return result;
}
```

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

---

Reading in a Texture from a BMP File

```c
Reading in a Texture from a BMP File
```

```c
result = Init06TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyTexturePuppy);
Init06TextureSampler( &MyPuppyTexture.texSampler );
```

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 Queues and Command Buffers

Mike Bailey

mjb@cs.oregonstate.edu

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

---

Vulkan Queues and Command Buffers

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

---

Querying what Queue Families are Available

```c
uint32_t  count;
vkGetPhysicalDeviceQueueFamilyProperties( 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, "	%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, " 
" );
}
```

Found 3 Queue Families:
0: Queue Family Count = 16  ;    Graphics Compute Transfer
1: Queue Family Count =   1  ;    Transfer
2: Queue Family Count =   8  ;    Compute
Similarly, we can write a function that finds the proper queue family:

```cpp
uint32_t FindQueueFamilyThatDoesGraphics()
{
    uint32_t count = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, OUT &count, OUT vqfp);
    for (unsigned int i = 0; i < count; i++)
    {
        if ((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
            return i;
    }
    return -1;
}
```

Creating a logical device needs to know queue family information:

```cpp
VkDeviceCreateInfo vdci = { 0);
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdci.pNext = nullptr;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pQueueCreateInfos = IN &vdqci[0];
vdci.queueCreateInfoCount = 1; // # of device queues wanted
```

Creating the command buffer as part of the logical device:

```cpp
vkAllocateCommandBuffer( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
```

Creating the command pool as part of the logical device:

```cpp
vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool );
```

Beginning a command buffer – one per image:

```cpp
vkBeginCommandBuffer( CommandBuffer, vcbbi );
```

Beginning a command buffer:

```cpp
vkEndCommandBuffer( CommandBuffers[0] );
```
These are the Commands that could be entered into the Command Buffer, I

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdBindPipeline( commandBuffer, pipeline )</td>
<td>Binds a pipeline to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindDescriptorSets( commandBuffer, layout, set )</td>
<td>Binds descriptor sets to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffers( commandBuffer, bind, count, offsets )</td>
<td>Binds vertex buffers to the command buffer.</td>
</tr>
<tr>
<td>vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance )</td>
<td>Draws a triangle.</td>
</tr>
<tr>
<td>vkCmdDrawIndirect( commandBuffer, stride )</td>
<td>Draws a triangle list.</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect( commandBuffer, stride )</td>
<td>Draws a triangle index list.</td>
</tr>
<tr>
<td>vkCmdDispatch( commandBuffer, x, y, z )</td>
<td>Dispatches work units to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, renderPass )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdEndRenderPass( commandBuffer )</td>
<td>Ends a render pass.</td>
</tr>
<tr>
<td>vkCmdClearAttachments( commandBuffer, attachmentCount, attach )</td>
<td>Clears attachment textures.</td>
</tr>
<tr>
<td>vkCmdCopyImageToBuffer( commandBuffer, regions )</td>
<td>Copies data from an image to a buffer.</td>
</tr>
<tr>
<td>vkCmdCopyBufferToImage( commandBuffer, regions )</td>
<td>Copies data from a buffer to an image.</td>
</tr>
<tr>
<td>vkCmdBeginQuery( commandBuffer, flags )</td>
<td>Begins a query.</td>
</tr>
<tr>
<td>vkCmdGetQueryResult( commandBuffer, query )</td>
<td>Gets the result of a query.</td>
</tr>
<tr>
<td>vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount )</td>
<td>Resets a query pool.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, contents )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdResetEvent( commandBuffer, event )</td>
<td>Resets an event.</td>
</tr>
<tr>
<td>vkCmdSignalSemaphore( commandBuffer, semaphore )</td>
<td>Signals a semaphore.</td>
</tr>
<tr>
<td>vkCmdSemaphoreWait( commandBuffer, semaphores, stageMask )</td>
<td>Waits for a semaphore.</td>
</tr>
<tr>
<td>vkCmdPipelineBarrier( commandBuffer, stageMask, dependencyFlags )</td>
<td>Causes a dependency between commands.</td>
</tr>
<tr>
<td>vkCmdBeginEvent( commandBuffer )</td>
<td>Begins an event.</td>
</tr>
<tr>
<td>vkCmdEndEvent( commandBuffer )</td>
<td>Ends an event.</td>
</tr>
<tr>
<td>vkCmdComplete( commandBuffer )</td>
<td>Completes a compute shader.</td>
</tr>
<tr>
<td>vkCmdSetEvent( commandBuffer, event )</td>
<td>Sets an event.</td>
</tr>
<tr>
<td>vkCmdResetEvent( commandBuffer, event )</td>
<td>Resets an event.</td>
</tr>
<tr>
<td>vkCmdSetEvent( commandBuffer, event )</td>
<td>Sets an event.</td>
</tr>
<tr>
<td>vkCmdSetPipeline( commandBuffer, pipeline )</td>
<td>Sets a pipeline.</td>
</tr>
<tr>
<td>vkCmdBindPipelineStageInfo( commandBuffer, stage, info )</td>
<td>Binds stage information to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindPushConstants( commandBuffer, layout, pushConstant )</td>
<td>Binds push constants to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindDescriptorSets( commandBuffer, layout, set )</td>
<td>Binds descriptor sets to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffers( commandBuffer, bind, count, offsets )</td>
<td>Binds vertex buffers to the command buffer.</td>
</tr>
<tr>
<td>vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance )</td>
<td>Draws a triangle.</td>
</tr>
<tr>
<td>vkCmdDrawIndirect( commandBuffer, stride )</td>
<td>Draws a triangle list.</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect( commandBuffer, stride )</td>
<td>Draws a triangle index list.</td>
</tr>
<tr>
<td>vkCmdDispatch( commandBuffer, x, y, z )</td>
<td>Dispatches work units to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, renderPass )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdEndRenderPass( commandBuffer )</td>
<td>Ends a render pass.</td>
</tr>
<tr>
<td>vkCmdClearAttachments( commandBuffer, attachmentCount, attach )</td>
<td>Clears attachment textures.</td>
</tr>
<tr>
<td>vkCmdCopyImageToBuffer( commandBuffer, regions )</td>
<td>Copies data from an image to a buffer.</td>
</tr>
<tr>
<td>vkCmdCopyBufferToImage( commandBuffer, regions )</td>
<td>Copies data from a buffer to an image.</td>
</tr>
<tr>
<td>vkCmdBeginQuery( commandBuffer, flags )</td>
<td>Begins a query.</td>
</tr>
<tr>
<td>vkCmdGetQueryResult( commandBuffer, query )</td>
<td>Gets the result of a query.</td>
</tr>
<tr>
<td>vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount )</td>
<td>Resets a query pool.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, contents )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdResetEvent( commandBuffer, event )</td>
<td>Resets an event.</td>
</tr>
<tr>
<td>vkCmdSignalSemaphore( commandBuffer, semaphore )</td>
<td>Signals a semaphore.</td>
</tr>
<tr>
<td>vkCmdSemaphoreWait( commandBuffer, semaphores, stageMask )</td>
<td>Waits for a semaphore.</td>
</tr>
<tr>
<td>vkCmdPipelineBarrier( commandBuffer, stageMask, dependencyFlags )</td>
<td>Causes a dependency between commands.</td>
</tr>
<tr>
<td>vkCmdBeginEvent( commandBuffer )</td>
<td>Begins an event.</td>
</tr>
<tr>
<td>vkCmdEndEvent( commandBuffer )</td>
<td>Ends an event.</td>
</tr>
<tr>
<td>vkCmdComplete( commandBuffer )</td>
<td>Completes a compute shader.</td>
</tr>
<tr>
<td>vkCmdSetEvent( commandBuffer, event )</td>
<td>Sets an event.</td>
</tr>
<tr>
<td>vkCmdResetEvent( commandBuffer, event )</td>
<td>Resets an event.</td>
</tr>
<tr>
<td>vkCmdSetPipeline( commandBuffer, pipeline )</td>
<td>Sets a pipeline.</td>
</tr>
<tr>
<td>vkCmdBindPipelineStageInfo( commandBuffer, stage, info )</td>
<td>Binds stage information to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindPushConstants( commandBuffer, layout, pushConstant )</td>
<td>Binds push constants to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindDescriptorSets( commandBuffer, layout, set )</td>
<td>Binds descriptor sets to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffers( commandBuffer, bind, count, offsets )</td>
<td>Binds vertex buffers to the command buffer.</td>
</tr>
<tr>
<td>vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance )</td>
<td>Draws a triangle.</td>
</tr>
<tr>
<td>vkCmdDrawIndirect( commandBuffer, stride )</td>
<td>Draws a triangle list.</td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect( commandBuffer, stride )</td>
<td>Draws a triangle index list.</td>
</tr>
<tr>
<td>vkCmdDispatch( commandBuffer, x, y, z )</td>
<td>Dispatches work units to the command buffer.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, renderPass )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdEndRenderPass( commandBuffer )</td>
<td>Ends a render pass.</td>
</tr>
<tr>
<td>vkCmdClearAttachments( commandBuffer, attachmentCount, attach )</td>
<td>Clears attachment textures.</td>
</tr>
<tr>
<td>vkCmdCopyImageToBuffer( commandBuffer, regions )</td>
<td>Copies data from an image to a buffer.</td>
</tr>
<tr>
<td>vkCmdCopyBufferToImage( commandBuffer, regions )</td>
<td>Copies data from a buffer to an image.</td>
</tr>
<tr>
<td>vkCmdBeginQuery( commandBuffer, flags )</td>
<td>Begins a query.</td>
</tr>
<tr>
<td>vkCmdGetQueryResult( commandBuffer, query )</td>
<td>Gets the result of a query.</td>
</tr>
<tr>
<td>vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount )</td>
<td>Resets a query pool.</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass( commandBuffer, contents )</td>
<td>Begins a render pass.</td>
</tr>
<tr>
<td>vkCmdResetEvent( commandBuffer, event )</td>
<td>Resets an event.</td>
</tr>
<tr>
<td>vkCmdSignalSemaphore( commandBuffer, semaphore )</td>
<td>Signals a semaphore.</td>
</tr>
<tr>
<td>vkCmdSemaphoreWait( commandBuffer, semaphores, stageMask )</td>
<td>Waits for a semaphore.</td>
</tr>
<tr>
<td>vkCmdPipelineBarrier( commandBuffer, stageMask, dependencyFlags )</td>
<td>Causes a dependency between commands.</td>
</tr>
<tr>
<td>vkCmdBeginEvent( commandBuffer )</td>
<td>Begins an event.</td>
</tr>
<tr>
<td>vkCmdEndEvent( commandBuffer )</td>
<td>Ends an event.</td>
</tr>
<tr>
<td>vkCmdComplete( commandBuffer )</td>
<td>Completes a compute shader.</td>
</tr>
</tbody>
</table>

Submitting a Command Buffer to a Queue for Execution

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkQueueSubmit( queue, submitInfo )</td>
<td>Submits a command buffer to a queue.</td>
</tr>
<tr>
<td>vkQueueWaitIdle( queue, flags )</td>
<td>Waits for a queue to become idle.</td>
</tr>
</tbody>
</table>

Example:

```c
VkQueueSubmit(VkQueue queue, uint32_t submitCount, const VkSubmitInfo* pSubmits, VkSemaphore waitSemaphore);  
```
The Entire Submission / Wait / Display Process

As the Vulkan 1.1 Specification says:

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

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there. But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

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

What Happens After a Queue has Been Submitted?

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")

The Swap Chain

Mike Bailey
mjb@cs.oregonstate.edu

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

How OpenGL Thinks of Framebuffers

Front

Update

Back

Present

How Vulkan Thinks of Framebuffers – the Swap Chain

The Swap Chain

Mike Bailey
mjb@cs.oregonstate.edu

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

What is a Swap Chain?

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

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

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

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

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

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

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

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there. But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.
We Need to Find Out What our Display Capabilities Are

```c
if( supported == VK_TRUE )
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics(), Surface, &supported );

VkBool32 supported;

fprintf( FpDebug, "Found %d Present Modes:
", presentModeCount);

vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );

uint32_t formatCount;

fprintf( FpDebug, "Found %d Surface Formats:
", formatCount );

VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];

VkExtent2D surfaceRes = vsc.currentExtent;

vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );

VkSurfaceCapabilitiesKHR vsc;

fprintf( FpDebug, "\** This Surface is supported by the Graphics Queue **\n" );
...
```

Creating a Swap Chain

```c
VkSwapchainCreateInfoKHR vscci;

vscci.image = PresentImages[ i ];
vscci.subresourceRange.layerCount = 1;
vscci.subresourceRange.baseArrayLayer = 0;
vscci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
vscci.components.a = VK_COMPONENT_SWIZZLE_A;
vscci.components.b = VK_COMPONENT_SWIZZLE_B;
vscci.components.g = VK_COMPONENT_SWIZZLE_G;
vscci.components.r = VK_COMPONENT_SWIZZLE_R;
vscci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vscci.pNext = nullptr;
vscci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;

vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[ i ] );
```

Creating the Swap Chain Images and Image Views

```c
VkSemaphoreCreateInfo vsci;

vsci.pNext = nullptr;
```

Rendering into the Swap Chain, 1

```c
vkBeginCommandBuffer( CommandBuffers[ nextImageIndex ], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```

VulkanDebug.txt output:

```
0: 44                0 ( VK_FORMAT_B8G8R8A8_UNORM,  VK_COLOR_SPACE_SRGB_NONLINEAR_KHR )
0:        2 ( VK_PRESENT_MODE_FIFO_KHR )
```
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.

In the shader, you define the layout of your constants. The layout must be a push constant, which means that it cannot be used inside of a uniform block. Let’s say we want to use a 4x4 matrix in our shader.

```glsl
layout(push_constant) uniform matrix4 modelMatrix;
```

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

```glsl
uint stageFlags = (VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT);
```

where:

- `stageFlags` are or’ed bits of `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`, `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`, etc.

size is in bytes

$p/variables$ is a void* pointer to the data, which, in this 4x4 matrix example, would be of type glm::mat4.
Setting up the Push Constants for the Pipeline Structure

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

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

```c
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;
```

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

An Robotic Example using Push Constants

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

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

Where each arm is represented by:

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

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

Forward Kinematics:

You Start with Separate Pieces, all Defined in their Own Local Coordinate System

1. Rotate by $\Theta_1$
2. Translate by $T_{CG}$

$$[M_{1/G}] = [T_{1/G}] \cdot [R_{\Theta_1}]$$

Positioning Part #1 With Respect to Ground

Locations?
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 $T_{1G}$

Write it

$$M_{2G} = [T_{1G}] [R_{\Theta_2}] [T_{\Theta_1}] [R_{\Theta_1}]$$

$$M_{2G} = [M_{1G}] [M_{21}]$$

Say it

Positioning Part #3 With Respect to Ground

1. Rotate by $\Theta_3$
2. Translate the length of part 2
3. Rotate by $\Theta_2$
4. Translate the length of part 1
5. Rotate by $\Theta_1$
6. Translate by $T_{1G}$

Write it

$$M_{3G} = [T_{1G}] [R_{\Theta_3}] [T_{\Theta_2}] [R_{\Theta_2}] [T_{\Theta_1}] [R_{\Theta_1}]$$

$$M_{3G} = [M_{1G}] [M_{21}] [M_{32}]$$

Say it

In the Reset Function

```c
struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
...
Arm1.armMatrix = glm::mat4(1.);
Arm1.armColor  = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale  = 6.f;
Arm2.armMatrix = glm::mat4(1.);
Arm2.armColor  = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale  = 4.f;
Arm3.armMatrix = glm::mat4(1.);
Arm3.armColor  = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale  = 2.f;
```

The constructor `glm::mat4(1.)` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`.

In the UpdateScene Function

```c
float rot1 = (float)Time;
float rot2 = 2.f * rot1;
float rot3 = 2.f * rot2;
glm::vec3 zaxis = glm::vec3(0., 0., 1.);
glm::mat4 m1g = glm::mat4(1.); // identity
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis); // [T]*[R]
glm::mat4 m21 = glm::mat4(1.); // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis); // [T]*[R]
m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm
glm::mat4 m32 = glm::mat4(1.); // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis); // [T]*[R]
m32 = glm::translate(m32, glm::vec3(0., 0., 2.)); // z-offset from previous arm
Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m1g * m21; // m2g
Arm3.armMatrix = m1g * m21 * m32; // m3g
```

In the RenderScene Function

```c
VkBuffer buffers[] = { 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);
```

The strategy is to draw each link using the same vertex buffer, but modified with a unique color, length, and matrix transformation.
In the Vertex Shader

```glsl
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
```

Physical Devices

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Vulkan: Identifying the Physical Devices

Vulkan: Which Physical Device to Use, I

Querying the Number of Physical Devices

```c
uint32_t count;
result = vkEnumeratePhysicalDevices(Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr);
if( result != VK_SUCCESS || count <= 0 ) {
fprintf(FpDebug, "Could not count the physical devices
");
return VK_SHOULD_EXIT;
}

if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 ) {
fprintf( FpDebug, "Could not count the physical devices
");
return VK_SHOULD_EXIT;
}

PhysicalDevice * physicalDevices = new PhysicalDevice[PhysicalDeviceCount];
if( result != VK_SUCCESS ) {
fprintf( FpDebug, "Could not enumerate the %d physical devices
", PhysicalDeviceCount );
return VK_SHOULD_EXIT;
}

for( unsigned int i = 0; i < PhysicalDeviceCount; i++ ) {
VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp);
if( result != VK_SUCCESS ) {
fprintf( FpDebug, "Could not get the physical device properties of device %d
", i);
return VK_SHOULD_EXIT;
}

fprintf( FpDebug, "

Device %2d:
", i );
fprintf( FpDebug, "API version: %d
", vpdp.apiVersion);
fprintf( FpDebug, "Driver version: %d
", vpdp.apiVersion);
fprintf( FpDebug, "Vendor ID: 0x%04x
", vpdp.vendorID);
fprintf( FpDebug, "Device ID: 0x%04x
", vpdp.deviceID);
fprintf( FpDebug, "Physical Device Type: %d = ", vpdp.deviceType);
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU ) fprintf( FpDebug, "(Discrete GPU)
" );
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU ) printf( FpDebug, "(Integrated GPU)
" );
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU ) printf( FpDebug, "(Virtual GPU)
" );
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU ) printf( FpDebug, "(CPU)
" );
fprintf( FpDebug, "Device Name: %s
", vpdp.deviceName);
fprintf( FpDebug, "Pipeline Cache Size: %d
", vpdp.pipelineCacheSize);
}
```

Where to put them

How many total there are

Which Physical Device to Use, I
<table>
<thead>
<tr>
<th>Physical Device</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 16 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 15 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 14 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 13 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 12 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 11 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 10 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 9 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 8 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 7 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 6 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 5 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 4 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 3 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 2 = 1</td>
</tr>
<tr>
<td>NVIDIA RTX 2080 Ti</td>
<td>Lemma 1 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 16 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 15 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 14 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 13 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 12 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 11 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 10 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 9 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 8 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 7 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 6 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 5 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 4 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 3 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 2 = 1</td>
</tr>
<tr>
<td>Intel HD Graphics 520</td>
<td>Lemma 1 = 1</td>
</tr>
</tbody>
</table>

**Which Physical Device to Use, II**

```c
void VulkanPhysicalDeviceProperties::VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    VulkanPhysicalDeviceFeatures::
    Vulcan...
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\n\t\t;   ", i, vqfp[i].queueCount);
    if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)       fprintf(FpDebug, " Graphics\n\t\t;   ");
    if((vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)       fprintf(FpDebug, " Compute \n\t\t;   ");
    if((vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)       fprintf(FpDebug, " Transfer\n\t\t;   ");
    fprintf(FpDebug, "\n");
    (void) vqfp[i].queueFlags;
}

Asking About the Physical Device's Queue Families

Here's What I Got

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

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

Looking to See What Device Layers are Available

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

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

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

Looking to See What Device Extensions are Available

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);
What Device Layers and Extensions are Available

4 physical device layers enumerated:

- `0x00401063 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
- 0 device extensions enumerated for 'VK_LAYER_NV_optimus':
  - `0x00000000 1 'VK_EXT_validation_cache'
  - `0x00000000 1 'VK_EXT_debug_marker'

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

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

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

Vulkan: Creating a Logical Device

```c
// VkDeviceCreateInfo
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vkDeviceCreateInfo(vdci);
```

```c
// VkDeviceQueueCreateInfo
VkDeviceQueueCreateInfo vdqci;
vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vkDeviceQueueCreateInfo(vdci);
```

Vulkan: Creating the Logical Device's Queue

```c
// Get the queue for this logical device:
void getQueue(VkDevice LogicalDevice, VkQueueIndex index, OUT &Queue); // QueueIndex, 0 = queueFamilyIndex, queueIndex
```

Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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