Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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 sample code

Mike Bailey

• Professor of Computer Science, Oregon State University
• Has been in computer graphics for over 30 years
• Has had over 8,000 students in his university classes
• mjb@cs.oregonstate.edu

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. Multipass
25. Ray Tracing

Welcome! I'm happy to be here. I hope you are too!

http://cs.oregonstate.edu/~mjb/vulkan
My Favorite Vulkan Reference


Introduction

Mike Bailey
mjb@cs.oregonstate.edu

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

Acknowledgements

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

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

Second, thanks to NVIDIA for all of their support!

2004: OpenGL 2.0 / GLSL 1.10 includes Vertex and Fragment Shaders

2008: OpenGL 3.0 / GLSL 1.30 adds features left out before

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

### History of Shaders

- **2014**: Khronos starts Vulkan effort
- **2016**: Vulkan 1.0
- **2016**: Vulkan 1.1
- **2020**: Vulkan 1.2

There is lots more detail at: [https://en.wikipedia.org/wiki/Vulkan_(API)](https://en.wikipedia.org/wiki/Vulkan_(API))

### 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.
**Why is it so important to keep the GPU Busy?**

<table>
<thead>
<tr>
<th>Title</th>
<th>Speed</th>
<th>Turbo</th>
<th>Turbo+PBO</th>
<th>ECC 16K B</th>
<th>ECC 128K B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan X</td>
<td>23.10</td>
<td>25.16</td>
<td>15.99</td>
<td>12.8</td>
<td>7.2</td>
</tr>
<tr>
<td>Titan Z</td>
<td>23.10</td>
<td>25.16</td>
<td>15.99</td>
<td>12.8</td>
<td>7.2</td>
</tr>
</tbody>
</table>

---

**From WikiPedia:**

“Vulcan is the god of fire including the fire of volcanoes, metalworking, and the forge in ancient Roman religion and myth. Vulcan is often depicted with a blacksmith’s hammer. The Vulcanalia was the annual festival held August 23 in his honor. His Greek counterpart is Hephaestus, the god of fire and smithy. In Etruscan religion, he is identified with Sethians. Vulcan belongs to the most ancient stage of Roman religion: Varro, the ancient Roman scholar and writer, citing the Annales Maximi, records that king Titus Tatius dedicated altars to a series of deities among which Vulcan is mentioned.”


---

**Who was the original Vulcan?**

**Who is the Khronos Group?**

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

Who’s Been Specifically Working on Vulkan?

Vulkan

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

Vulkan Differences from OpenGL

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

- Model Transform
- View Transform
- Per-vertex Lighting
- Projection Transform
- Rasterization
- Fragment Processing, Texturing, Per-fragment Lighting
- Framebuffer

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

The Basic Computer Graphics Pipeline, Shader-style

- Model Transform
- View Transform
- Per-vertex in variables
- Projection Transform
- Rasterization
- Fragment Processing, Texturing, Per-fragment Lighting
- Framebuffer

Per-vertex in variables
Uniform Variables

The Basic Computer Graphics Pipeline, Vulkan-style

- Vertex Shader
- Fragment Shader
- Rasterization
- Framebuffer

Per-vertex in variables
Uniform Variables

Moving part of the driver into the application

- Complex drivers lead to driver overhead and cross vendor unpredictability
- Error management is always active
- Driver processes full shading language source

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

Khronos Group
Vulkan Highlights: Command Buffers

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

![Command Buffers Diagram]

Vulkan: Creating a Pipeline

- `VkGraphicsPipelineCreateInfo` includes:
  - Shader stages
  - Binding descriptions
  - Input attachment descriptions
  - Viewports
  - Rasterization state
  - Multi-sample state
  - Depth-stencil state
  - Color blend state
  - Dynamic state

- `VkPipelineShaderStageCreateInfo` includes:
  - Stage type
  - Shader module

- ` VkPipelineVertexInputStateCreateInfo` includes:
  - Binding descriptions
  - Stride descriptions

- ` VkPipelineInputAssemblyStateCreateInfo` includes:
  - Topology

- ` VkViewportStateCreateInfo` includes:
  - Dimensions
  - Scissors

- ` VkPipelineRasterizationStateCreateInfo` includes:
  - Cull mode
  - Polygon mode
  - Front face

- ` VkPipelineDepthStencilStateCreateInfo` includes:
  - Depth comparison operator
  - Stencil test enable

- ` VkPipelineColorBlendStateCreateInfo` includes:
  - Blending enable
  - Color blend factors

Vulkan: Pipeline State Objects

- 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

- `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 OpenCL and Vulkan pattern (get used to it):
Vulkan Code has a Distinct “Style” of Setting Information in structs and then Passing that Information as a pointer-to-the-struct:

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = buffer size in bytes;  // << buffer size in bytes >>
vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;

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

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

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

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &MatrixBufferMemoryHandle );
result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```

Vulkan Quick Reference Card – I Recommend you Print This!

Vulkan Highlights: Overall Block Diagram
**Steps in Creating Graphics using Vulkan**

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

**Vulkan GPU Memory**

- Your application allocates GPU memory for the objects it needs  
- To write and read that GPU memory, you map that memory to the CPU address space  
- Your application is responsible for making sure that what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.

**Vulkan Render Passes**

- Drawing is done inside a render pass  
- Each render pass contains what framebuffer attachments to use  
- Each render pass is told what to do when it begins and ends
Vulkan Compute Shaders

- Compute pipelines are allowed, but they are treated as something special (just like OpenGL treats them)
- Compute passes are launched through dispatches
- Compute command buffers can be run asynchronously

Vulkan Synchronization

- Synchronization is the responsibility of the application
- Events can be set, polled, and waited for (much like OpenCL)
- Vulkan itself does not ever lock – that’s your application’s job
- Threads can concurrently read from the same object
- Threads can concurrently write to different objects

Vulkan Shaders

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

Advantages:

1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends

The “19” refers to the version of Visual Studio, not the year of development.
The Vulkan Sample Code Included with These Notes

Mike Bailey
mjba@cs.oregonstate.edu

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

Sample Program Output

Sample Program Keyboard Inputs

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

Y, 'Y': 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 setup once and then re-used each time.
5. At times, I've setup things that didn't need to be setup just to show you what could go there.
Caveats on the Sample Code, II

6. There are great uses for C++ classes and methods here to hide some complexity, but I’ve not done that.

7. I’ve typedef’ed a couple things to make the Vulkan phraseology more consistent.

8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure

9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.

10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

```c
int main( int argc, char * argv[ ] ) {
    Width = 800; Height = 600;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 ) {
        fprintf( stderr, "Cannot open debug print file '%s'
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
    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;
    }
}
```

InitGraphics(), I

```c
void InitGraphics( ) {
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW();
    Init02CreateDebugCallbacks();
    Init03PhysicalDeviceAndGetQueueFamilyProperties();
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer);
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool();
    Init06CommandBuffers( );
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyPuppyTexture );
    Init08Swapchain();
    Init09DepthStencilImage( );
    Init10RenderPasses( );
    Init11Framebuffers( );
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorSetLayouts();
    Init13DescriptorSets();
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

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

The Vertex Data is in a Separate File

#include "SampleVertexData.cpp"

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

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

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

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

What if you don’t need all of this information?

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

For example, what if you are not doing texturing in this application? Should you re-do this struct and leave the texCoord element out?

As best as I can tell, the only costs for retaining vertex attributes that you aren’t going to use are some GPU memory space and possibly some inefficient uses of the cache, but not gross performance. So, I recommend keeping this struct intact, and, if you don’t need texturing, simply don’t use the texCoord values in your vertex shader.
Vulkan Software Philosophy

Vulkan has lots of typedefs that define C/C++ structs and enums.

Vulkan takes a non-C++ object-oriented approach in that those typedef'd structs pass all the necessary information into a function. For example, where we might normally say in C++:

```cpp
result = LogicalDevice->vkGetDeviceQueue( queueFamilyIndex, queueIndex, OUT &Queue );
```

we would actually say in C:

```c
result = vkGetDeviceQueue( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
```

Vulkan Conventions

My Conventions

- VkXxx is a typedef, probably a struct
- vkYyy( ) is a function call
- VK_ZZZ is a constant

My Conventions

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

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

```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr);
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT &physicalDevices[0]);
```

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

<table>
<thead>
<tr>
<th>How many total</th>
<th>Where to put them</th>
</tr>
</thead>
<tbody>
<tr>
<td>there are</td>
<td></td>
</tr>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, OUT &amp;count, NULLptr );</td>
<td></td>
</tr>
<tr>
<td>result = vkEnumeratePhysicalDevices( Instance, OUT &amp;count, &amp;physicalDevices[0] );</td>
<td></td>
</tr>
</tbody>
</table>

Your Sample2019.zip File Contains This

The “19” refers to the version of Visual Studio, not the year of development.
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, "Feature Not Present" },
    { VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" },
    { VK_ERROR_TOO_MANY_OBJECTS, "Too Many Objects" },
    { 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, "Out of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
    { VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" },
    { VK_SUBOPTIMAL_KHR, "Suboptimal" },
    { 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, "Out of Date" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
};

void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s\n", 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\n", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}

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

bool Paused;
bool Verbose;

#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );

const int32_t OFFSET_ZERO = 0;

Extras in the Code

Drawing

Mike Bailey

mjb@cs.oregonstate.edu

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

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

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

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

A Colored Cube Example

Vertices and triangle indices from SampleVertexData.cpp:

Triangles Represented as an Array of Structures

From the file SampleVertexData.cpp:

```cpp
From the file SampleVertexData.cpp:

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

static GLuint CubeTriangleIndices[12] = {
    // triangle 0-2-3:// vertex #0:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. },
    // vertex #2:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. },
    // vertex #3:
    {  1.,  1., -1. },
    {  0.,  0., -1. },
    {  1.,  1.,  0. },
    {  0.,1. }
};
```
From the file SampleVertexData.cpp:

Non-indexed Buffer Drawing

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

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

A Preview of What init05DataBuffer Does

VkResult init05DataBuffer(VkDeviceSize size, 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 = size;
    vbci.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer(LogicalDevice, &vbci, PALLOCATOR, &pMyBuffer->buffer);
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements(LogicalDevice, pMyBuffer->buffer, &vmr);
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, &vmai, PALLOCATOR, &vdm);
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory(LogicalDevice, pMyBuffer->buffer, vdm, 0);
    return result;
}

C/C++:

GLSL Shader:

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

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

Telling the Pipeline about its Input

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

#### Vertices structure

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

#### Attribute Descriptions

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

#### Input Assembly State Create Info

```c
VkPipelineInputAssemblyStateCreateInfo vpiasci;  // used to describe the input vertex attributes
    vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
    vpiasci.pNext = nullptr;
    vpiasci.flags = 0;
    vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
```

### Telling the Command Buffer what Vertices to Draw

```c
VkBuffer buffers[1] = {MyVertexDataBuffer.buffer};
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets);
const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```
Drawing with an Index Buffer

```
struct vertex JustVertexData[] = {
  // vertex #0:
  { -1., -1., -1. },{  0.,  0., -1. },
  {  0.,  0.,  0. },
  {  1., 0. },
  // vertex #1:
  {  1., -1., -1. },
  {  0.,  0., -1. },
  {  1.,  0.,  0. },

  . . .
};

int JustIndexData[] = {
  0, 2, 3,
  0, 3, 1,
  4, 5, 7,
  4, 7, 6,
  1, 3, 7,
  1, 7, 5,
  0, 4, 6,
  0, 6, 2,
  2, 7, 3,
  0, 1, 5,
  0, 5, 4,
};
```

int vertex Lookup[] = {
  0, 2, 3, 0, 3, 2, 0, 1, 5, 0, 1, 5,
};

```
vnDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);
```
Indirect Drawing (not to be confused with Indexed)

```c
typedef struct
{
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;
```

Compare this with:

```c
vkCmdDrawIndirect(commandBuffers[nextImageIndex], buffer, offset, drawCount, stride);
```

Indexed Indirect Drawing (i.e., both Indexed and Indirect)

```c
vkCmdDrawIndexedIndirect(commandBuffer, buffer, offset, drawCount, stride);
```

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 in. Sometimes it doesn’t.

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

Thus, when using 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.
### The OBJ File Format – a triple-indexed way of Drawing

The OBJ file format uses 1-based indexing for faces!

### 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

```c
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: OpenGl uses:
- 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

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

Shader combinations of separate texture data and samplers:
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );

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

Push Constants:
layout( push_constant ) . . .

Specialization Constants:
layout( constant_id = 3 )  const int N = 5;
- Only for scalars, but a vector’s components can be constructed from specialization constants

Specialization Constants for Compute Shaders:
layout( local_size_x_id = 8, local_size_y_id = 16 );
- This sets gl_WorkGroupSize.x and gl_WorkGroupSize.y
- gl_WorkGroupSize.z is set as a constant

Vulkan-Shaders’ use of Layouts for Uniform Variables

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

glslangValidator shaderFile -V [H] [-I<dir>] [-S <stage>] -o shaderBinaryFile.spv

Shaderfile extensions:
- .vert Vertex
- .tesc Tessellation Control
- .tess 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

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

1. Click on the Microsoft Start icon

2. Type the word bash

This is only available within 64-bit Windows 10.

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

Pick one:
- Can get to your personal folders
- Does not have make

- Can get to your personal folders
- Does have make

Running glslangValidator.exe
The input file. The compiler determines the
shader type by the file extension:
- .vert Vertex shader
- .tccs Tessellation Control Shader
- .tecs Tessellation Evaluation Shader
- .geom Geometry shader
- .frag Fragment shader
- .comp Compute shader

Same as C/C++ — the compiler gives you no nasty messages.
Also, if you care, legal .spv files have a magic number of 0x07230203
So, if you do an od -x on the .spv file, the magic number looks like this:
0203 0723 . . .
VkGraphicsPipelineCreateInfo

Shader stages
VertexInput State
InputAssembly State
Tesselation State
Viewport State
Rasterization State
MultiSample State
DepthStencil State
ColorBlend State
Dynamic State
Pipeline layout
RenderPass

basePipelineHandle
basePipelineIndex

VkPipelineShaderStageCreateInfo

VkPipelineVertexInputStateCreateInfo

VkVertexInputBindingDescription

Viewport
x, y, w, h,
minDepth,
maxDepth
offset
extent

VkPipelineRasterizationStateCreateInfo

cullMode
polygonMode
frontFace
lineWidth

VkSpecializationInfo

which stage (VERTEX, etc.)

VkShaderModule

VkPipelineInputAssemblyStateCreateInfo

Topology

VkVertexInputAttributeDescription

binding
stride
inputRate
location

VkPipelineDepthStencilStateCreateInfo

depthTestEnable
depthWriteEnable
depthCompareOp
stencilTestEnable
stencilOpStateFront
stencilOpStateBack

blendEnable
srcColorBlendFactor
dstColorBlendFactor
colorBlendOp
srcAlphaBlendFactor
dstAlphaBlendFactor
alphaBlendOp
colorWriteMask

VkPipelineColorBlendAttachmentState

This is the SPIR-V Assembly, Part I

You can also take a look at SPIR-V Assembly

glslangValidator.exe -V -H sample.vert -o sample.vert.spv

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

For example, if this is your Shader Source

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

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

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

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

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

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

This is the SPIR-V Assembly, Part I
A Google-Wrapped Version of glslangValidator

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 -DNUMPONTS=4 sample-vert.vert -o sample-vert.spv
```

glslc is included in your Sample.zip file.

Data Buffers

Mike Bailey
mjb@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:

```c
typedef VkBuffer VkDataBuffer;
```

This is probably a bad idea in the long run.
Creating and Filling Vulkan Data Buffers

Creating a Vulkan Data Buffer

Creating a Vulkan Data Buffer

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

Finding the Right Type of Memory
Finding the Right Type of Memory

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

Finding the Right Type of Memory

- 11 Memory Types:
  - Memory 0:
  - Memory 1:
  - Memory 2:
  - Memory 3:
  - Memory 4:
  - Memory 5:
  - Memory 6:
  - Memory 7: DeviceLocal
  - Memory 8: DeviceLocal
  - Memory 9: HostVisible HostCoherent
  - Memory 10: HostVisible HostCoherent HostCached
- 2 Memory Heaps:
  - Heap 0: size = 0xb7c00000 DeviceLocal
  - Heap 1: size = 0xfac00000

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;

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.

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

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

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

This code assumes that this line:
#define GLM_FORCE_RADIANS
is listed before GLM is included!
This C struct is holding the original data, written by the application.

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

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

```
MyBuffer MyMatrixUniformBuffer;
```

```
uniform matBuf Matrices;
```

---

**Filling the Data Buffer**

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

---

```cpp
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &pMyBuffer->buffer );

    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );         // 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, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```

---

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

    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.
Creating and Filling the Data Buffer – the Details

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

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

---

GLFW

Mike Bailey
mjb@cs.oregonstate.edu

```c
#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"
...
```

```c
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

```c
uint32_t count;
const char ** extensions = glfwGetRequiredInstanceExtensions(&count);
for( uint32_t i = 0; i < count; ++i )
{
    fprintf(FpDebug, "%s
", extensions[i]);
}
```

Found 2 GLFW Required Instance Extensions:

- VK_KHR_surface
- VK_KHR_win32_surface

GLFW Keyboard Callback

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

GLFW Mouse Button Callback

```c
GLFWMouseButton(GLFWwindow * window, int button, int action, int mods )
{
    int b = 0; // LEFT, MIDDLE, or RIGHT
    switch( button )
    {
        case GLFW_MOUSE_BUTTON_LEFT:
            b = LEFT;
            break;
        case GLFW_MOUSE_BUTTON_MIDDLE:
            b = MIDDLE;
            break;
        case GLFW_MOUSE_BUTTON_RIGHT:
            b = RIGHT;
            break;
        default:
```

GLFW Mouse Motion Callback

```c
GLFWMouseMotion(GLFWwindow * window, double xpos, double ypos )
{
    int dx = (int)xpos - Xmouse;
    int dy = (int)ypos - Ymouse;
    if( ActiveButton & LEFT )
    {
        Xrot += (ANGFACT*dy);
        Yrot += (ANGFACT*dx);
    }
    if( ActiveButton & MIDDLE )
    {
        Scale += SCLFACT * (float) ( dx - dy );
        if( Scale < MINSCALE )
            Scale = MINSCALE;
    }
    Xmouse = (int)xpos;
    Ymouse = (int)ypos;
```
Looping and Closing GLFW

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

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

Does not block – processes any waiting events, then returns.

If you would like to block waiting for events, use:

glfwWaitEvents();

You can have the blocking wake up after a timeout period with:

glfwWaitEventsTimeout(double secs);

You can wake up one of these blocks from another thread with:

glfwPostEmptyEvent();

What is GLM?

GLM is a set of C++ classes and functions to fill in the programming gaps in writing the basic vector and matrix mathematics for OpenGL applications. However, even though it was written for OpenGL, it works fine with Vulkan.

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

You can find it at:

http://glm.g-truc.net/0.9.8.5/

You invoke GLM like this:

#include <glm/glm.h>
#include <glm/gtc/matrix_transform.h>
#include <glm/gtc/matrix_inverse.h>

OpenGL treats all angles as given in degrees. This line forces 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. Later on, these notes will show you how to use it from there.
Why are we even talking about this?

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

```c
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:

```c
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\}
```

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

### Constructor:

- `glm::mat4( 1. );` // identity matrix
- `glm::vec4( );`
- `glm::vec3( );`

### Multiplications:

- `glm::mat4 * glm::mat4`
- `glm::mat4 * glm::vec4`
- `glm::mat4 * glm::vec3( glm::vec3, 1. )` // promote a vec3 to a vec4 via a constructor

### Emulating OpenGL transformations with concatenation:

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

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

- `glm::mat4 glm::ortho( float left, float right, float bottom, float top, float near, float far );`
- `glm::mat4 glm::frustum( float left, float right, float bottom, float top, float near, float far );`
- `glm::mat4 glm::perspective( float fovy, float aspect, float near, float far );`

### Viewing (assign, not concatenate):

- `glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );`

Installing GLM into your own space

I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.
Here’s what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

GLM in the Vulkan sample.cpp Program
Or, in matrix form:

This is called a “Linear Transformation” because all of the coordinates are raised to the 1st power, that is, there are no $x^2$, $x^3$, etc. terms.

$$\begin{bmatrix}
    x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
    A & B & C & D \\
    E & F & G & H \\
    I & J & K & L
\end{bmatrix}
\begin{bmatrix}
    x \\
y \\
z
\end{bmatrix}$$

Translation

$$\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}$$

Rotation about X

$$\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}$$

Rotation about Y

$$\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}$$

Rotation about Z

$$\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} =
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}$$

The Rotation Matrix for an Angle ($\theta$) about an Arbitrary Axis ($Ax$, $Ay$, $Az$)

$$\begin{bmatrix}
A_xA_z + \cos \theta(1 - A_yA_z) & A_xA_y - \cos \theta A_yA_z & A_zA_x - \cos \theta A_yA_z + \sin \theta A_x \\
A_xA_y + \cos \theta A_yA_z & A_yA_z - \cos \theta(1 - A_yA_z) & A_yA_x - \cos \theta A_yA_z - \sin \theta A_x \\
A_zA_x + \cos \theta A_yA_z & A_yA_x - \cos \theta A_yA_z + \sin \theta A_x & A_zA_y - \cos \theta(1 - A_yA_z)
\end{bmatrix}$$

For this to be correct, $A$ must be a unit vector.
Our rotation matrices only work around the origin. What if we want to rotate about an arbitrary point (A, B)?

A: We create more than one matrix.

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

One matrix – the Current Transformation Matrix, or CTM

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} = \left(T_{x,A,B}\right)\left[R_y\right]\left(T_{z,A,B}\right)\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix} = \left[T_{z,A,B}\right]\left[R_y\right]\left[T_{x,A,B}\right]\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

Matrix Multiplication is not Commutative

Matrix Multiplication is Associative

One Matrix to Rule Them All

```cpp
glm::mat4 Model = glm::mat4(1.);
Model = glm::translate(Model, glm::vec3(A, B, 0.));
Model = glm::rotate(Model, thetaRadians, glm::vec3(Ax, Ay, Az));
Model = glm::translate(Model, glm::vec3(-A, -B, 0.));

glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.); glm::vec3 up(0.,1.,0.);

glm::mat4 View = glm::lookAt(eye, look, up);

glm::mat4 Projection = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Projection[1][1] *= -1.;

glm::mat3 Matrix = Projection * View * Model;

glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(Model));
```
Why Isn’t The Normal Matrix exactly the same as the Model Matrix?

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

Wrong!

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

Right!

Original object and normal

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

Instancing

Mike Bailey
mjba@cs.oregonstate.edu

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

Instancing – What and why?

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

Making each Instance look differently -- Approach #1

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

gl_InstanceIndex starts at 0

In the vertex shader:

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);
vColor = vec3(1, 1, float(1 + gl_InstanceIndex)) / float(NUMINSTANCES), 0.);
xdelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2;
ydelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2;
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0), 1.);
gl_Position = PVM * vertex; // [p]*[v]*[m]

BTW, when not using instancing, be sure the instanceCount is 1, not 0!
Put the unique characteristics in a uniform buffer array and reference them

Still uses gl_InstanceIndex

In the vertex shader:

```glsl
layout( std140, set = 3, binding = 0 ) uniform colorBuf {
    vec3 uColors[1024];
} Colors;
out vec3 vColor;
.
.
int index = gl_InstanceIndex % 1024; // or "& 1023" – gives 0 - 1023
vColor = Colors.uColors[ index ];
vec4 vertex = ...
.
.
gl_Position = PVM * vertex; // [p]*[v]*[m]
```

What is the Vulkan Graphics Pipeline?

Here's what you need to know:

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

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

There is also a Vulkan Compute Pipeline Data Structure – we will get to that later.
Graphics Pipeline Stages and what goes into Them

The GPU and Driver specify the Pipeline Stages – the Vulkan Graphics Pipeline declares what goes in them.

- **Vertex Shader module**
  - Specialization info
  - Vertex input binding
  - Vertex input attributes
- **Topology**
- **Viewport**
- **Tessellation, Geometry Shader**
- **Viewport Scissoring**
- **Depth Clamping**
  - DiscardEnable
  - PolyclampMode
- **Rasterization**
  - Which states are dynamic
  - DepthTestEnable
  - DepthWriteEnable
- **Fragment Shader module**
- **Color Blending parameters**

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: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

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

The First Step: Create the Graphics Pipeline Layout

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

```cpp
VkResult init14GraphicsPipelineLayout() {
    VkResult result;
    VkPipelineLayoutCreateInfo vplci = {
        .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,
        .pNext = nullptr,
        .flags = 0,
        .setLayoutCount = 4,
        .pSetLayouts = &DescriptorSetLayouts[0],
        .pushConstantRangeCount = 0,
        .pPushConstantRanges = (VkPushConstantRange *)nullptr,
    };
    result = vkCreatePipelineLayout(
        LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}
```

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

```cpp
vkCreateGraphicsPipeline() {
    VkPipelineLayoutCreateInfo vplci = {
        .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,
        .pNext = null,
        .flags = 0,
        .setLayoutCount = 4,
        .pSetLayouts = &DescriptorSetLayouts[0],
        .pushConstantRangeCount = 0,
        .pPushConstantRanges = (VkPushConstantRange *)null,
    };
    result = vkCreatePipelineLayout(
        LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}
```
Creating a Typical Graphics Pipeline

The Shaders to Use

```c
// Pipeline creation code

VkPipelineShaderStageCreateInfo vpssci[2];

// Fragment shader

void Init14GraphicsVertexFragmentPipeline(
    VkShaderModule vertexShader,
    VkShaderModule fragmentShader,
    OUT VkPipeline *pGraphicsPipeline)
{
    vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[0].pNext = nullptr;
    vpssci[0].flags = 0;
    vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[1].pNext = nullptr;
    vpssci[1].flags = 0;
    vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    vpssci[1].module = fragmentShader;
    vpssci[1].pName = "main";
    vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    // Pipeline vertex input state create info
    VkPipelineVertexInputStateCreateInfo vpvisci;

    // Vertex binding description
    VkPipelineInputAssemblyStateCreateInfo vpiasci;

    // Tessellation state create info
    VkPipelineTessellationStateCreateInfo vptsci;

    // Geometry state create info
    VkPipelineGeometryStateCreateInfo vpgsci;

    // Fragment state create info
    VkPipelineColorBlendStateCreateInfo vpccsi;"
What is “Primitive Restart Enable”?

vpiasci.primitiveRestartEnable = VK_FALSE;

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

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

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

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 0xffff.
If your VkIndexType is VK_INDEX_TYPE_UINT32, it is 0xffffffff.

One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

Triangle Strip #0
Triangle Strip #1
Triangle Strip #2
...
What is the Difference Between Changing the Viewport and Changing the Scissoring?

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

Original Image

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

What is “Depth Clamp Enable”?

`vprsci.depthClampEnable = VK_FALSE;`

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

A good use for this is Polygon Capping:

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

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

What is “Depth Bias Enable”?

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

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

Setting the Rasterizer State
MultiSampling State

```cpp
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

MultiSampling in a separate noteseet.

Color Blending State for each Color Attachment

```cpp
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 = (VkColorComponentFlags)VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT; }
```

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

Colour new = (1 - \alpha) \times Colour existing + \alpha \times Colour 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.

Which Pipeline Variables can be Set Dynamically

```cpp
VkPipelineDynamicStateCreateInfo
vpdsci = { .sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO,
    .pNext = nullptr,
    .dynamicStateCount = 0,
    .pDynamicStates = vds; }
```

Just used as an example in the Sample Code
The Stencil Buffer

Update

Depth
Stencil
Render

Here's how the Stencil Buffer works:
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.

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

Using the Stencil Buffer to Perform Polygon Capping
Using the Stencil Buffer to Perform Polygon Capping

1. Clear the SB = 0
2. Draw the polygons, setting SB = ~ SB
3. Draw a large gray polygon across the entire scene wherever SB ≠ 0

Using the Stencil Buffer to Better Outline Polygons

```
for( each polygon )
{
    Draw the edges, setting SB = 1
    Draw the polygon wherever SB ≠ 1
    Draw the edges, setting SB = 0
}
```

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

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

Stencil Operations for Front and Back Faces

Operations for Depth Values

Putting it all Together! (finally…)

Group all of the individual state information and create the pipeline
Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

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

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:

```cpp
VkViewport
    v.x = 0;
    v.y = 0;
    v.width = (float)Width;
    v.height = (float)Height;
    v.minDepth = 0.0f;
    v.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

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?

```cpp
layout( std140, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, binding = 1 ) uniform mat4 uViewMatrix;
layout( std140, binding = 2 ) uniform mat4 uProjectionMatrix;
layout( std140, binding = 3 ) uniform mat3 uNormalMatrix;
layout( std140, binding = 4 ) uniform vec4 uLightPos;
layout( std140, binding = 5 ) uniform float uTime;
layout( std140, binding = 6 ) uniform int uMode;
layout( binding = 7 ) uniform sampler2D uSampler;
```
Descriptor Sets are an intermediate data structure that tells shaders how to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders. There are three advantages in doing things this way:

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

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

**What are Descriptor Sets?**

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

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

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

### Step 1: Descriptor Set Pools

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

```cpp
struct matBuf {
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
    glm::mat3 uNormalMatrix;
};
struct lightBuf {
    glm::vec4 uLightPos;
};
struct miscBuf {
    float uTime;
    int uMode;
};
struct sampler2D uSampler;
```

```cpp
layout( std140, set = 0, binding = 0 ) uniform matBuf
    {    
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat3 uNormalMatrix;
    } Matrices;
layout( std140, set = 1, binding = 0 ) uniform lightBuf
    {    
        vec4 uLightPos;
    } Light;
layout( std140, set = 2, binding = 0 ) uniform miscBuf
    {    
        float uTime;
        int uMode;
    } Misc;
layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```
Step 2: Define the Descriptor Set Layouts

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

```
matrixSet DS Layout Binding: LightSet DS Layout Binding: MiscSet DS Layout Binding:TexSamplerSet DS Layout Binding:

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

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

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

// DS #3:
TexSamplerSet[0].binding = 0;
TexSamplerSet[0].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
TexSamplerSet[0].descriptorCount = 1;
TexSamplerSet[0].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;
```
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```cpp
VkResult Init14GraphicsPipelineLayout() {
    VkResult result;
    VkPipelineLayoutCreateInfo vplci = {};
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}
```

Step 4: Allocating the Memory for Descriptor Sets

```cpp
VkResult Init13DescriptorSets() {
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai = {};
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;
    result = vkAllocateDescriptorSets(LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
    return result;
}
```
Step 5: Tell the Descriptor Sets where their CPU Data is

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

This struct identifies what texture sampler and image view it owns

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

This struct links a Descriptor Set to the image it is pointing to
Step 7: Bind Descriptor Sets into the Command Buffer when Drawing

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

So, the Pipeline Layout contains the structure of the Descriptor Sets. Any collection of Descriptor Sets that match that structure can be bound into that pipeline.

Sidebar: The Entire Collection of Descriptor Set Paths

- `vkCreateDescriptorPool()`: Create the pool of Descriptor Sets for future use
- `vkCreateDescriptorSetLayout()`: Describe a particular Descriptor Set layout and use it in a specific Pipeline layout
- `vkAllocateDescriptorSets()`: Allocate memory for particular Descriptor Sets
- `vkCmdBindDescriptorSets()`: Tell a particular Descriptor Set where its CPU data is
- `vkUpdateDescriptorSets()`: Re-write CPU data into a particular Descriptor Set
- `vkCmdBindDescriptorSets()`: Make a particular Descriptor Set "current" for rendering

Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

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.
Textures

Mike Bailey
mjb@cs.oregonstate.edu

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

The Basic Idea

Texture mapping is a computer graphics operation in which a separate image, referred to as the texture, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a texture map.

Also, to prevent confusion, the texture pixels are not called pixels. A pixel is a dot in the final screen image. A dot in the texture image is called a texture element, or texel.

Similarly, to avoid terminology confusion, a texture’s width and height dimensions are not called X and Y. They are called S and T. A texture map is not generally indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always S=0, the right side is S=1, the bottom is T=0, and the top is T=1. Thus, you do not need to be aware of the texture’s resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.

Enable texture mapping:

```glEnable( GL_TEXTURE_2D );```

Draw your polygons, specifying s and t at each vertex:

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

Disable texture mapping:

```glDisable( GL_TEXTURE_2D );```
Triangles in an Array of Structures

```cpp
struct vertex {
    glm::vec3 position;   // Position of the vertex in 3D space.
    glm::vec3 normal;     // Normal vector indicating the orientation.
    glm::vec3 color;      // Color of the vertex.
    glm::vec2 texCoord;   // Texture coordinates.
};

struct vertex VertexData[] = {
    // Triangle 0-2-3:
    // Vertex #0:
    { -1.0f, -1.0f, -1.0f }, // Position
    {  0.0f,  0.0f, -1.0f }, // Position
    {  0.0f,  0.0f,  0.0f }, // Position
    {  1.0f,  0.0f }, // Texture

    // Vertex #2:
    { -1.0f,  1.0f, -1.0f }, // Position
    {  0.0f,  0.0f, -1.0f }, // Position
    {  0.0f,  1.0f,  0.0f }, // Position
    {  1.0f,  1.0f }, // Texture

    // Vertex #3:
    {  1.0f,  1.0f, -1.0f }, // Position
    {  0.0f,  0.0f, -1.0f }, // Position
    {  1.0f,  1.0f,  0.0f }, // Position
    {  0.0f,  1.0f }, // Texture
};
```

Using a Texture: How do you know what (s,t) to assign to each vertex?

The easiest way to figure out what s and t are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane,

\[
s = \frac{x - X_{min}}{X_{max} - X_{min}} \quad t = \frac{y - Y_{min}}{Y_{max} - Y_{min}}
\]

Or, for a sphere,

\[
s = \frac{\Theta - (-\pi)}{2\pi} \quad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}
\]

Using a Texture: How do you know what (s,t) to assign to each vertex?

Uh-oh. Now what? Here's where it gets tougher…,
You really are at the mercy of whoever did the modeling...

Be careful where s abruptly transitions from 1. back to 0.

Memory Types

NVIDIA Discrete Graphics:
11 Memory Types:
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal HostVisible
- Memory 2: DeviceLocal HostVisible HostCoherent
- Memory 3: HostVisible HostCoherent
- Memory 4: HostVisible HostCoherent HostCached

Intel Integrated Graphics:
3 Memory Types:
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal HostVisible HostCoherent
- Memory 2: DeviceLocal HostVisible HostCoherent HostCached
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 leftover in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

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.

Texture Mip*-mapping

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

* Latin: multum in parvo, "many things in a small place"
vici.queueFamilyIndexCount = 0;
Init07TextureBuffer( INOUT MyTexture * pMyTexture)
{
    VkResult result;
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled
    uint32_t texWidth = pMyTexture->width;
    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, IN stagingImage, OUT &vmr);
    unsigned char *texture = pMyTexture->pixels; VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each
    if (Verbose)
    {
        VkImage stagingImage;
        VkImage textureImage;
        fprintf(FpDebug, "Image vmr.alignment = %lld
", vmr.alignment);
        fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
        // *******************************************************************************
        fflush(FpDebug);
        // this second {...} is to create the actual texture image:
        // *******************************************************************************
        
        VkImageCreateInfo vici;
        vici.pNext = nullptr;
        vici.flags = 0;
        void * gpuMemory;
        vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
        vici.imageType = VK_IMAGE_TYPE_2D;
        vici.format = VK_FORMAT_R8G8B8A8_UNORM;
        if (vsl.rowPitch == 4 * texWidth)
            vici.arrayLayers = 1;
        vici.samples = VK_SAMPLE_COUNT_1_BIT;
        vici.tiling = VK_IMAGE_TILING_OPTIMAL;
        memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
        vici.queueFamilyIndexCount = 0;
    }

    VkDeviceMemory vdm;
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled
    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
    if (Verbose)
    {
        fprintf(FpDebug, "Texture vmr.size = %lld
", vmr.size);
        fprintf(FpDebug, "Texture vmr.alignment = %lld
", vmr.alignment);
        fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
        fflush(FpDebug);
    }
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
    result = vkBindImageMemory(LogicalDevice, IN textureImage, IN vdm, 0); // 0 = offset
    // *******************************************************************************
};
// copy pixels from the staging image to the texture:
VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

// *******************************************************************************
// transition the staging buffer layout:
// *******************************************************************************
{
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
//vkQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = stagingImage;
vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
vimb.dstAccessMask = 0;
vimb.subresourceRange = visr;
result = vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0,
0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,
1, IN &vimb);
}

// *******************************************************************************
// transition the texture buffer layout:
// *******************************************************************************
{
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
//vkQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = 0;
vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
vimb.subresourceRange = visr;
result = vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,
1, IN &vimb);

// now do the final image transfer:
VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.baseArrayLayer = 0;
visl.mipLevel = 0;
visl.layerCount = 1;
VkOffset3D vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;
VkExtent3D ve3;
ve3.width = texWidth;
ve3.height = texHeight;
ve3.depth = 1;
VkImageCopy vic;
vic.srcSubresource = visl;
vic.srcOffset = vo3;
vic.dstSubresource = visl;
vic.dstOffset = vo3;
vic.extent = ve3;
vkCmdCopyImage(TextureCommandBuffer,
stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);
}

// *******************************************************************************
// transition the texture buffer layout a second time:
// *******************************************************************************
{
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
//vkQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.image = textureImage;
vimb.srcAccessMask = 0;
vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
vimb.subresourceRange = visr;
result = vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,
1, IN &vimb);
}

result = vkEndCommandBuffer(TextureCommandBuffer);
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &TextureCommandBuffer;
vsi.waitSemaphoreCount = 0;
vsi.pWaitSemaphores = (VkSemaphore *)nullptr;
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
result = vkQueueSubmit(Queue, 1, IN &vsi, VK_NULL_HANDLE);
result = vkQueueWaitIdle(Queue);
// 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;

textureImage

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

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: Overall Block Diagram

Application

Instance

Instance

Physical Device

Physical Device

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Command Buffer
Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

```c
int FindQueueFamilyThatDoesGraphics()
{
uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, IN &count, OUT vqfp);
for (unsigned int i = 0; i < count; i++)
{
if ((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
    return i;
}
return -1;
}
Creating a Logical Device Needs to Know Queue Family Information

```c
float queuePriorities[] = {
    1. // one entry per queueCount
};
VkDeviceQueueCreateInfo vdqci[1];
vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1;
vdqci[0].queuePriorities = (float *) queuePriorities;
```

Creating the Command Pool as part of the Logical Device

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

Creating the Command Buffers

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

Beginning a Command Buffer – One per Image

```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
uint32_t nextImageIndex;
vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );
```
These are the Commands that could be entered into the Command Buffer, I

- vkBeginCommandBuffer
- vkCmdBeginQuery
- vkCmdBeginRenderPass
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindPipeline
- vkCmdBindVertexBuffers
- vkCmdBlitImage
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdCopyBuffer
- vkCmdCopyBufferToImage
- vkCmdCopyImage
- vkCmdCopyImageToBuffer
- vkCmdCopyQueryPoolResults
- vkCmdDebugMarkerBeginEXT
- vkCmdDebugMarkerEndEXT
- vkCmdDebugMarkerInsertEXT
- vkCmdDispatch
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndirect
- vkCmdEndQuery
- vkCmdEndRenderPass
- vkCmdExecuteCommands
- vkCmdFillBuffer
- vkCmdNextSubpass
- vkCmdPipelineBarrier
- vkCmdProcessCommandsNVX
- vkCmdPushConstants
- vkCmdPushDescriptorSet
- vkCmdReserveSpaceForCommandsNVX
- vkCmdResetEvent
- vkCmdResetQueryPool
- vkCmdResolveImage
- vkCmdSetBlendConstants
- vkCmdSetDepthBias
- vkCmdSetDepthBounds
- vkCmdSetDeviceMaskKHX
- vkCmdSetDiscardRectangleEXT
- vkCmdSetEvent
- vkCmdSetLineWidth
- vkCmdSetScissor
- vkCmdSetStencilCompareMask
- vkCmdSetStencilReference
- vkCmdSetStencilWriteMask
- vkCmdSetViewport
- vkCmdSetViewportWScalingNV
- vkCmdUpdateBuffer
- vkCmdWaitEvents
- vkCmdWriteTimestamp
- vkCmdCopyQueryPoolResults
- vkCmdDebugMarkerBeginEXT
- vkCmdDebugMarkerEndEXT
- vkCmdDebugMarkerInsertEXT

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

- vkBeginCommandBuffer
- vkCmdBeginQuery
- vkCmdBeginRenderPass
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindPipeline
- vkCmdBindVertexBuffers
- vkCmdBlitImage
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdCopyBuffer
- vkCmdCopyBufferToImage
- vkCmdCopyImage
- vkCmdCopyImageToBuffer
- vkCmdCopyQueryPoolResults
- vkCmdDebugMarkerBeginEXT
- vkCmdDebugMarkerEndEXT
- vkCmdDebugMarkerInsertEXT
- vkCmdDispatch
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndirect
- vkCmdEndQuery
- vkCmdEndRenderPass
- vkCmdExecuteCommands
- vkCmdFillBuffer
- vkCmdNextSubpass
- vkCmdPipelineBarrier
- vkCmdProcessCommandsNVX
- vkCmdPushConstants
- vkCmdPushDescriptorSet
- vkCmdReserveSpaceForCommandsNVX
- vkCmdResetEvent
- vkCmdResetQueryPool
- vkCmdResolveImage
- vkCmdSetBlendConstants
- vkCmdSetDepthBias
- vkCmdSetDepthBounds
- vkCmdSetDeviceMaskKHX
- vkCmdSetDiscardRectangleEXT
- vkCmdSetEvent
- vkCmdSetLineWidth
- vkCmdSetScissor
- vkCmdSetStencilCompareMask
- vkCmdSetStencilReference
- vkCmdSetStencilWriteMask
- vkCmdSetViewport
- vkCmdSetViewportWScalingNV
- vkCmdUpdateBuffer
- vkCmdWaitEvents
- vkCmdWriteTimestamp
vkClearColorValue vccv;
  vccv.float32[0] = 0.0;
vccv.float32[1] = 0.0;
vccv.float32[2] = 0.0;
vccv.float32[3] = 1.0;

vkClearDepthStencilValue vcdsv;
  vcdsv.depth = 1.f;
vcdsv.stencil = 0;

vkClearValue vcv[2];
  vcv[0].color = vccv;
  vcv[1].depthStencil = vcdsv;

VkOffset2D o2d = { 0, 0 };
VkExtent2D e2d = { Width, Height };
VkRect2D r2d = { o2d, e2d };

VkRenderPassBeginInfo vrpbi;
  vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
  vrpbi.pNext = nullptr;
  vrpbi.renderPass = RenderPass;
  vrpbi.framebuffer = Framebuffers[nextImageIndex];
  vrpbi.renderArea = r2d;
  vrpbi.clearValueCount = 2;
  vrpbi.pClearValues = vcv; // used for VK_ATTACHMENT_LOAD_OP_CLEAR

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

VkViewport viewport = {
  0.,                     // x
  (float)Width,                        // y
  (float)Height,                     // minDepth
  1.                      // maxDepth
};

vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &viewport);         // 0=firstViewport, 1=viewportCount

VkRect2D scissor = {
  0,0,
  Width,
  Height
};

vkCmdSetScissor(CommandBuffers[nextImageIndex], 0, 1, IN &scissor);

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

// dynamic offset count, dynamic offsets
vkCmdBindPushConstants(CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values);

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);               // 0, 1 = firstBinding, bindingCount

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);

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

result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence);     // 1 = submitCount
result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);     // waitAll, timeout
vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

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

result = vkQueuePresentKHR(presentQueue, IN &vpi);
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.
What is a Swap Chain?

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

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

Swap Chains are arranged as a ring buffer.

Swap Chains are tightly coupled to the window system. After creating the Swap Chain in the first place, the process for using the Swap Chain is:

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

We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

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

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

Found 2 Surface Formats:
0: 44 0 (VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR )
1: 50 0 (VK_FORMAT_B8G8R8A8_SRGB, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR )

Found 3 Present Modes:
0: 2 (VK_PRESENT_MODE_FIFO_KHR )
1: 3 (VK_PRESENT_MODE_FIFO_RELAXED_KHR )
2: 1 (VK_PRESENT_MODE_MAILBOX_KHR )
```

Creating a Swap Chain

```text
vkCreateSwapchainKHR( )
VkSwapchainCreateInfo surface
imageFormat
imageColorSpace
imageExtent
imageArrayLayers
imageUsage
imageSharingMode
preTransform
compositeAlpha
presentMode
clipped
```

We Need to Find Out What our Display Capabilities Are

```text
vkGetDevicePhysicalSurfaceCapabilitiesKHR( )
VkSurfaceCapabilities
minImageCount
maxImageCount
currentExtent
minImageExtent
maxImageExtent
maxImageArrayLayers
supportedTransforms
currentTransform
supportedCompositeAlpha
supportedUsageFlags
```
Creating a Swap Chain

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

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr; vscci.flags = 0; vscci.surface = Surface;
vscci.minImageCount = 2;
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.colorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
vscci.imageArrayLayers = 1;
vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );

Creating the Swap Chain Images and Image Views

uint32_t imageCount;
result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr );
PresentImages = new VkImage[imageCount];
result = vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );

Rendering into the Swap Chain, I

VkSemaphoreCreateInfo vsci;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
nextImageIndex = UINT64_MAX;
result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
result = vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, VK_SUBPASS_CONTENTS_INLINE );
result = vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
result = vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
result = vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );

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

Push Constants

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

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

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

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

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

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

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

Setting up the Push Constants for the Pipeline Structure

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

```glsl
struct arm
{
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale; // scale factor in x
};
struct armArm1;
struct armArm2;
struct armArm3;
```

An Robotic Example using Push Constants

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

Where each arm is represented by:

```glsl
struct arm
{
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale; // scale factor in x
};
struct armArm1;
struct armArm2;
struct armArm3;
```

Forward Kinematics:

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

1
2
3
Forward Kinematics:
Hook the Pieces Together, Change Parameters, and Things Move
(All Young Children Understand This)

Forward Kinematics:
Given the Lengths and Angles, Where do the Pieces Move To?

Positioning Part #1 With Respect to Ground

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

Write it

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

Say it

Why Do We Say it Right-to-Left?

We adopt the convention that the coordinates are multiplied on the right side of the matrix:

$$
\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = [M_{1/G}] \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = [T_{1/G}] \times [R_{\Theta_1}] \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
$$

So the right-most transformation in the sequence multiplies the $(x,y,z,1)$ first and the left-most transformation multiplies it last.
Positioning Part #2 With Respect to Ground

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

Write it

$$
\begin{bmatrix}
M_{2/G} \\
M_{2/G}
\end{bmatrix} =
\begin{bmatrix}
T_{1/G} & R_{\Theta_1} & T_{2/1} & R_{\Theta_2}
\end{bmatrix}
\begin{bmatrix}
M_{1/G} & M_{2/1}
\end{bmatrix}
$$

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 $T_{1/G}$

Write it

$$
\begin{bmatrix}
M_{3/G} \\
M_{3/G}
\end{bmatrix} =
\begin{bmatrix}
T_{1/G} & R_{\Theta_1} & T_{2/1} & R_{\Theta_2} & T_{3/2} & R_{\Theta_3}
\end{bmatrix}
\begin{bmatrix}
M_{1/G} & M_{2/1} & M_{3/2}
\end{bmatrix}
$$

Say it

In the Reset Function

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

Setup the Push Constant for the Pipeline Structure

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

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

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```
In the **UpdateScene Function**

```cpp
float rot1 = (float)Time;  // 1
float rot2 = 2.f * rot1;  // 2
float rot3 = 2.f * rot2;  // 3

glm::vec3 zaxis = glm::vec3(0., 0., 1.);  // 4
glm::mat4 m1g = glm::mat4(1.f);  // identity  // 5
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));  // 6
m1g = glm::rotate(m1g, rot1, zaxis);  // [T]*[R]  // 7
m21 = glm::mat4(1.f);  // identity  // 8
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));  // 9
m21 = glm::rotate(m21, rot2, zaxis);  // [T]*[R]  // 10
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));  // z-offset from previous arm  // 11
m32 = glm::mat4(1.f);  // identity  // 12
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));  // 13
m32 = glm::rotate(m32, rot3, zaxis);  // [T]*[R]  // 14
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));  // z-offset from previous arm  // 15

Arm1.armMatrix = m1g;  // m1g  // 16
Arm2.armMatrix = m1g * m21;  // m2g  // 17
Arm3.armMatrix = m1g * m21 * m32;  // m3g  // 18
```

In the **RenderScene Function**

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };  // 19
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);  // 20
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1);  // 21
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);  // 22
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2);  // 23
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);  // 24
vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3);  // 25
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);  // 26
```

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**

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

layout( location = 0 ) in vec3 aVertex;

...  // 27

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.));  // 28

...  // 29

gl_Position = PVM * vec4(bVertex, 1.);  // Projection * Viewing * Modeling matrices  // 30
```
Physical Devices

Mike Bailey
mjb@cs.oregonstate.edu

Vulkan: Overall Block Diagram

Vulkan: a More Typical (and Simplified) Block Diagram

Querying the Number of Physical Devices

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

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

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

Which Physical Device to Use, I

int discreteSelect = -1;
int integratedSelect = -1;
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    if( physicalDevices[i].deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )  
        discreteSelect = i;
    if( physicalDevices[i].deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU ) 
        integratedSelect = i;
}
int which = -1;
if( discreteSelect >= 0 )
{
    which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
}
else if( integratedSelect >= 0 )
{
    which = integratedSelect;
    PhysicalDevice = physicalDevices[which];
}
else
{
    fprintf( FpDebug, "Could not select a Physical Device
" );
    return VK_SHOULD_EXIT;
}

Asking About the Physical Device’s Features

VkPhysicalDeviceProperties PhysicalDeviceProperties;
vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );
pprintf( FpDebug, "PhysicalDevice Properties:
" );
pprintf( FpDebug, "PhysicalDeviceProperties.deviceName = %s
", PhysicalDeviceProperties.deviceName );
pprintf( FpDebug, "PhysicalDeviceProperties.apiVersion = %d
", PhysicalDeviceProperties.apiVersion );
pprintf( FpDebug, "PhysicalDeviceProperties.vendorID = 0x%04x
", PhysicalDeviceProperties.vendorID );
pprintf( FpDebug, "PhysicalDeviceProperties.deviceID = 0x%04x
", PhysicalDeviceProperties.deviceID );

Which Physical Device to Use, II

VkPhysicalDeviceProperties *_physicalDeviceProperties = nullptr;
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT _physicalDevices );
if( result != VK_SUCCESS )
{
    fprintf( FpDebug, "Could not enumerate the %d physical devices
", PhysicalDeviceCount );
    return VK_SHOULD_EXIT;
}

Vulkan: Identifying the Physical Devices

VkResult result = VK_SUCCESS;
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
if( result != VK_SUCCESS )
{
    fprintf( FpDebug, "Could not count the physical devices
" );
    return VK_SHOULD_EXIT;
}
for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
{
    if( physicalDevices[i].deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
        discreteSelect = i;
    if( physicalDevices[i].deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
        integratedSelect = i;
    else
    {
        fprintf( FpDebug, "Could not select a Physical Device
" );
        return VK_SHOULD_EXIT;
    }

    VkPhysicalDeviceProperties *vpdp = nullptr;
    vkGetPhysicalDeviceProperties( 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 );

    vkPhysicalDeviceProperties( physicalDeviceProperties = 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 Name: %s
", vpdp->deviceName );
    fprintf( FpDebug, "Pipeline Cache Size: %d
", vpdp->pipelineCacheSize );
    fprintf( FpDebug, "Instance Count: %d
", vpdp->instanceCount );
    fprintf( FpDebug, "Queue Flags: %d
", vpdp->queueFlags );
    fprintf( FpDebug, "Queue Indices: %d
", vpdp->queueIndices );
    fprintf( FpDebug, "Queue Types: %d
", vpdp->queueTypes );
    fprintf( FpDebug, "Queue Priorities: %d
", vpdp->queuePriorities );

    VkPhysicalDeviceProperties( physicalDeviceFeatures = nullptr );
    vkGetPhysicalDeviceFeatures( physicalDevice, OUT &physicalDeviceFeatures );
    if( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not enumerate the physical device features
" );
        return VK_SHOULD_EXIT;
    }

    fprintf( FpDebug, "Physical Device Features:
" );
    fprintf( FpDebug, "geometryShader = %2d
", physicalDeviceFeatures.geometryShader);
    fprintf( FpDebug, "tessellationShader = %2d
", physicalDeviceFeatures.tessellationShader );
    fprintf( FpDebug, "multiDrawIndirect = %2d
", physicalDeviceFeatures.multiDrawIndirect );
    fprintf( FpDebug, "wideLines = %2d
", physicalDeviceFeatures.wideLines );
    fprintf( FpDebug, "largePoints = %2d
", physicalDeviceFeatures.largePoints );
    fprintf( FpDebug, "multiViewport = %2d
", physicalDeviceFeatures.multiViewport );
    fprintf( FpDebug, "occlusionQueryPrecise = %2d
", physicalDeviceFeatures.occlusionQueryPrecise );
    fprintf( FpDebug, "pipelineStatisticsQuery = %2d
", physicalDeviceFeatures.pipelineStatisticsQuery );
    fprintf( FpDebug, "shaderFloat4x4 = %2d
", physicalDeviceFeatures.shaderFloat4x4 );
    fprintf( FpDebug, "shaderInt64 = %2d
", physicalDeviceFeatures.shaderInt64 );
    fprintf( FpDebug, "shaderInt8x8 = %2d
", physicalDeviceFeatures.shaderInt8x8 );
    fprintf( FpDebug, "shaderInt16 = %2d
", physicalDeviceFeatures.shaderInt16 );
}
Here's What the NVIDIA RTX 2080 Ti Produced

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

Here's What the Intel HD Graphics 520 Produced

vkEnumeratePhysicalDevices:
Device 0:
API version: 4194360
Driver version: 4194360
Vendor ID: 0x8086
Device ID: 0x1916
Physical Device Type: 1 = (Integrated GPU)
Device Name: Intel(R) HD Graphics 520
Pipeline Cache Size: 213
Device #0 selected (Intel(R) HD Graphics 520)
Physical Device Features:
- geometryShader = 1
- tessellationShader = 1
- multiDrawIndirect = 1
- wideLines = 1
- largePoints = 1
- multiViewport = 1
- occlusionQueryPrecise = 1
- pipelineStatisticsQuery = 1
- shaderFloat64 = 1
- shaderInt64 = 1
- shaderInt16 = 1

Asking About the Physical Device's Different Memories

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

Asking About the Physical Device's Different Memories

VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
pWrite(FpDebug, "%d Memory Heaps:\n", vpdmp.memoryHeapCount );
for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ){
  VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
  fprintf(FpDebug, "Heap %d: size = 0x%08lx ", i, vmh.size);
  if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT  ) != 0 )     fprintf(FpDebug, "Device Local" );
  fprintf(FpDebug, "\n" );
}

Here's What I Got

11 Memory Types:
- Memory 0:
- Memory 1:
- Memory 2:
- Memory 3:
- Memory 4:
- Memory 5:
- Memory 6:
- Memory 7: DeviceLocal
- Memory 8: DeviceLocal
- Memory 9: HostVisible HostCoherent
- Memory 10: HostVisible HostCoherent HostCached
2 Memory Heaps:
Heap 0: size = 0x67000000 DeviceLocal
Heap 1: size = 0x7a00000
Asking About the Physical Device’s Queue Families

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

Here’s What I Got

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

Vulkan: Overall Block Diagram

Logical Devices

Mike Bailey
mjba@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjba/vulkan
const char * myDeviceLayers[] = {
    // "VK_LAYER_LUNARG_api_dump",
    // "VK_LAYER_LUNARG_core_validation",
    // "VK_LAYER_LUNARG_image","VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    // "VK_LAYER_NV_optimus"
};
const char * myDeviceExtensions[] = {
    "VK_KHR_surface",
    "VK_KHR_win32_surface",
    "VK_EXT_debug_report"
    // "VK_KHR_swapchains"
};

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

Looking to See What Device Layers are Available

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

// 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);

Looking to See What Device Extensions are Available

What Device Layers and Extensions are Available

4 physical device layers enumerated:

0x00401063 1 "VK_LAYER_NV_optimus" "NVIDIA Optimus layer"
0x00401072 1 "VK_LAYER_LUNARG_core_validation" "LunarG Validation Layer"
2 device extensions enumerated for "VK_LAYER_LUNARG_core_validation":
  0x00000001  "VK_EXT_validation_cache"
  0x00000004  "VK_EXT_debug_marker"

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

0x00401072 1 "VK_LAYER_LUNARG_parameter_validation" "LunarG Validation Layer"
2 device extensions enumerated for "VK_LAYER_LUNARG_parameter_validation":
  0x00000001  "VK_EXT_validation_cache"
  0x00000004  "VK_EXT_debug Marker"
Vulkan: Creating a Logical Device

```cpp
float queuePriorities[1] = { 1.0f; }
VkDeviceQueueCreateInfo vdqci;
vqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = 0;
vqci.queueCount = 1;
vqci.pQueueProperties = queuePriorities;
result = vkCreateDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice; )
```

Vulkan: Creating the Logical Device’s Queue

```cpp
vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue; )
```

Dynamic State Variables

Mike Bailey
mjb@cs.oregonstate.edu

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

Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

**VkGraphicsPipelineCreateInfo**

- Shader stages
- VertexInput State
- InputAssembly State
- Tesselation State
- Viewport State
- Rasterization State
- MultiSample State
- DepthStencil State
- ColorBlend State
- Dynamic State
- Pipeline layout
- RenderPass
- basePipelineHandle
- basePipelineIndex

**VkPipelineShaderStageCreateInfo**

**VkPipelineVertexInputStateCreateInfo**

**VkVertexInputBindingDescription**

**VkViewportStateCreateInfo**

**Viewport**

- x, y, w, h,
- minDepth,
- maxDepth
- offset
- extent

**Scissor**

**VkPipelineRasterizationStateCreateInfo**

- cullMode
- polygonMode
- frontFace
- lineWidth

**VkSpecializationInfo**

- which stage (VERTEX, etc.)

**VkShaderModule**

**VkPipelineInputAssemblyStateCreateInfo**

- Topology

**VkVertexInputAttributeDescription**

- binding
- stride
- inputRate
- location

**VkPipelineDepthStencilStateCreateInfo**

**VkPipelineColorBlendStateCreateInfo**

- depthTestEnable
- depthWriteEnable
- depthCompareOp
- stencilTestEnable
- stencilOpStateFront
- stencilOpStateBack
- blendEnable
- srcColorBlendFactor
- dstColorBlendFactor
- colorBlendOp
- srcAlphaBlendFactor
- dstAlphaBlendFactor
- alphaBlendOp
- colorWriteMask

**VkPipelineColorBlendAttachmentState**

Creating a Pipeline

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

Which Pipeline State Variables can be Changed Dynamically

The possible dynamic variables are shown in the *VkDynamicState* enum:

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

Filling the Dynamic State Variables in the Command Buffer

First call:

```
vkCmdBindPipeline( … );
```

Then, the command buffer-bound function calls to set these dynamic states are:

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

Mike Bailey
mjb@cs.oregonstate.edu

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

There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp

- Vulkan requires you to first setup "Query Pools", one for each specific type
- This indicates that Vulkan thinks that Queries are time-consuming (relatively) to setup, and thus better to set them up in program-setup than in program-runtime

Setting up Query Pools

- VkQueryPoolCreateInfo
  - sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO
  - pNext = nullptr
  - flags = 0
  - queryType = one of: VK_QUERY_TYPE OCCLUSION, VK_QUERY_TYPE_PIPELINE_STATISTICS, VK_QUERY_TYPE_TIMESTAMP
  - queryCount = 1
  - pipelineStatistics = bitmask of what stats you are querying for if you are doing a pipeline statistics query

- VkQueryPool
  - occlusionQueryPool
  - statisticsQueryPool
  - timestampQueryPool

Resetting, Filling, and Examining a Query Pool

- vkCmdResetQueryPool (CommandBuffer, occlusionQueryPool, 0, 1);
- vkCmdBeginQuery (CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT);
- ...
- vkCmdEndQuery (CommandBuffer, occlusionQueryPool, 0);
- #define DATASIZE 128
- uint32_t data[DATASIZE];
- result = vkGetQueryPoolResults (LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags);
- possible combinations of flags:
  - VK_QUERY_RESULT_64_BIT
  - VK_QUERY_RESULT_WAIT_BIT
  - VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
  - VK_QUERY_RESULT_PARTIAL_BIT
  - stride is # of bytes in between each result
Occlusion Queries count the number of fragments drawn between the `vkCmdBeginQuery` and the `vkCmdEndQuery` that pass both the Depth and Stencil tests. This is commonly used to see what level-of-detail should be used when drawing a complicated object.

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

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

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

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

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

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

The `vkCmdWriteTimeStamp()` function produces the time between when this function is called and when the first thing reaches the specified pipeline stage. Even though the stages are “bits”, you are supposed to only specify one of them, not “or” multiple ones together.

```c
vkCmdWriteTimeStamp( CommandBuffer, pipelineStages, timestampQueryPool, 0 );
```
The Rendering Draws the Particles by Reading the Position and Color Buffers

The Example We Are Going to Use Here is a Particle System

The Compute Shader Moves the Particles by Recomputing the Position and Velocity Buffers

The Rendering Draws the Particles by Reading the Position and Color Buffers

The Data in your C/C++ Program will look like This

```
#define NUM_PARTICLES (1024*1024) // total number of particles to move
#define NUM_WORK_ITEMS_PER_GROUP                    64 // # work-items per work-group
#define  NUM_X_WORK_GROUPS                     (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)

struct pos
{
  glm::vec4; // positions
};

struct vel
{
  glm::vec4; // velocities
};

struct col
{
  glm::vec4; // colors
};
```

Note that .w and .vw are not actually needed. But, by making these structure sizes a multiple of 4 floats, it doesn't matter if they are declared with the std140 or the std430 qualifier. I think this is a good thing.

The Data in your Compute Shader will look like This

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

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

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

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

Here is how you create a Compute Pipeline Data Structure

A Reminder about Data Buffers

Creating a Shader Storage Buffer
Allocating Memory for a Buffer, Binding a Buffer to Memory, and Filling the Buffer

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

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

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

Create the Compute Pipeline Layout

```cpp
vkDestroyPipelineLayout(LogicalDevice, ComputePipelineLayout, PALLOCATOR);
```

Create the Compute Pipeline

```cpp
VkPipeline ComputePipeline;
result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1,
&vcpci[0], PALLOCATOR, &ComputePipeline);
```
Creating a Vulkan Data Buffer

```
VkBuffer Buffer;
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = NUM_PARTICLES * sizeof(glm::vec4);
vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const int32_t)nullptr;
result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &posBuffer);
```

Allocating Memory and Binding the Buffer

```
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements(LogicalDevice, posBuffer, OUT &vmr);
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory(LogicalDevice, posBuffer, IN vdm, 0); // 0 is the offset
```

Fill the Buffers

```
struct pos * positions;
vkMapMemory(LogicalDevice, IN myPosBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &positions);
for( int i = 0; i < NUM_PARTICLES; i++ )
    positions[i].x = Ranf(XMIN, XMAX);
    positions[i].y = Ranf(YMIN, YMAX);
    positions[i].z = Ranf(ZMIN, ZMAX);
    positions[i].w = 1.;
vkUnmapMemory(LogicalDevice, IN myPosBuffer.vdm);
```

```
struct vel * velocities;
vkMapMemory(LogicalDevice, IN myVelBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &velocities);
for( int i = 0; i < NUM_PARTICLES; i++ )
    velocities[i].x = Ranf(VXMIN, VXMAX);
    velocities[i].y = Ranf(VYMIN, VYMAX);
    velocities[i].z = Ranf(VZMIN, VZMAX);
    velocities[i].w = 0.;
vkUnmapMemory(LogicalDevice, IN myVelBuffer.vdm);
```

```
struct col * colors;
vkMapMemory(LogicalDevice, IN myColBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &colors);
for( int i = 0; i < NUM_PARTICLES; i++ )
    colors[i].r = Ranf(.3f, 1.);
    colors[i].g = Ranf(.3f, 1.);
    colors[i].b = Ranf(.3f, 1.);
    colors[i].a = 1.;
vkUnmapMemory(LogicalDevice, IN myColBuffer.vdm);
```

# Include `<stdlib.h>`

```
#define TOP 2147483647. // 2^31 - 1
float Ranf(float low, float high)
{
    long random(); // returns integer 0 - TOP
    float r = (float)rand();
    return low + r * (high - low) / (float)RAND_MAX;
}
```
The Particle System Compute Shader

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

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

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

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

The Particle System Compute Shader

The Data gets Divided into Large Quantities call **Work-Groups**, each of which is further Divided into Smaller Units Called **Work-Items**

The number of **work-items** per **work-group** set in the compute shader. The number of **work-groups** is set in the `vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY, workGroupCountZ );` function call in the application program.

The Data Needs to be Divided into Large Quantities call **Work-Groups**, each of which is further Divided into Smaller Units Called **Work-Items**

The Invocation Space can be 1D, 2D, or 3D. This one is 1D.

```
#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}
```

20 total items to compute:

```
5 \times 4 = \frac{20}{4}
```

A Mechanical Equivalent...

http://news.cision.com
The Particle System Compute Shader – The Physics

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

const VECTOR G = VECTOR( 0., -9.8, 0. );
const float DT = 0.1;
const SPHERE Sphere = vec4( -100., -800., 0., 600. ); // x, y, z, r

uint gid = gl_GlobalInvocationID.x; // where I am in the global dataset (6 in this example)

POINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;
POINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;

if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + 0.5*DT*DT*G;
}

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

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

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

// plane equation:  Ax + By + Cz + D = 0
// (it turns out that (A,B,C) is the normal)
VELOCITY BouncePlane( POINT p, VELOCITY v, PLANE pl)
{
    VECTOR n = normalize( VECTOR( pl.xyz ) );
    return Bounce( v, n );
}

bool IsUnderPlane( POINT p, PLANE pl )
{
    float r = pl.x*p.x + pl.y*p.y + pl.z*p.z + pl.w;
    return ( r < 0. );
}
```

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

```cpp
uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;
POINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;

if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + 0.5*DT*DT*G;
}

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

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

```c
#define NUM_PARTICLES (1024*1024)
#define NUM_WORK_ITEMS_PER_GROUP 64
#define NUM_X_WORK_GROUPS (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)

vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);

vkCmdDispatch(CommandBuffer, NUM_X_WORK_GROUPS, 1, 1);
```

This is the number of work-groups, set in the application program. The number of work-items per work-group is set in the layout in the compute shader:

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

Displaying the Particles

```c
VkVertexInputBindingDescription vvibd[3]; // one of these per buffer data buffer
vvibd[0].binding = 0; // which binding # this is
vvibd[0].stride = sizeof(struct pos); // bytes between successive structs
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

vvibd[1].binding = 1;
vibd[1].stride = sizeof(struct vel);
vibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

vvibd[2].binding = 2;
vibd[2].stride = sizeof(struct col);
vibd[2].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

layout( location = 0 ) in vec4 aPosition;
layout( location = 1 ) in vec4 aVelocity;
layout( location = 2 ) in vec4 aColor;
```

VkVertexInputAttributeDescription vviad[3]; // array per vertex input attribute
vviad[0].location = 0; // location in the layout decoration
vviad[0].binding = 0; // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC4; // x, y, z, w
vviad[0].offset = offsetof(struct pos, pos); // 0

vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC4; // nx, ny, nz
vviad[1].offset = offsetof(struct vel, vel); // 0

vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC4; // r, g, b, a
vviad[2].offset = offsetof(struct col, col); // 0

VkPipelineVertexInputStateCreateInfo vpvisci; // used to describe the input vertex attributes
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.vertexBindingDescriptionCount = 3;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexAttributeDescriptionCount = 3;
vpvisci.pVertexAttributeDescriptions = vviad;

VkPipelineInputAssemblyStateCreateInfo vpiasci; // used to describe the input vertex attributes
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
vpiasci.flags = 0;
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
Telling the Pipeline about its Input

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

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

Setting a Pipeline Barrier so the Drawing Waits for the Compute

```
VkBuffer buffers[] = MyPosBuffer.buffer, MyVelBuffer.buffer, MyColBuffer.buffer };
size_t offsets[] = { 0, 0, 0 };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 3, buffers, offsets );
const uint32_t vertexCount = NUM_PARTICLES;
const uint32_t  instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], NUM_PARTICLES, 1, 0, 0 ); // vertexCount, instanceCount, firstVertex, firstInstance
```

Drawing

```
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier
{
 commandBuffer;
 VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,
 VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount
 IN &vbmb, 0, nullptr
};
```

Setting a Pipeline Barrier so the Compute Waits for the Drawing

```
VkBufferMemoryBarrier vmbm;
 vmbm.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vmbm.pNext = nullptr;
vmbm.srcAccessFlags = VK_ACCESS_UNIFORM_READ_BIT;
vmbm.dstAccessFlags = VK_ACCESS_SHADER_WRITE_BIT;
vmbm.srcQueueFamilyIndex = 0;
vmbm.dstQueueFamilyIndex = 0;
vmbm.buffer = 0;
vmbm.offset = 0;
vmbm.size = NUM_PARTICLES * sizeof( glm::vec4 );
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier
{
 commandBuffer;
 VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
 VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount
 IN &vmbm, 0, nullptr
};
```
**Specialization Constants**

Mike Bailey  
mjb@cs.oregonstate.edu

---

**What Are Specialization Constants?**

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

Normally, the half-way compile finalizes all constant values and compiles the code that uses them.

But, it would be nice every so often to have your Vulkan program sneak into the halfway-compiled binary and manipulate some constants at runtime. This is what Specialization Constants are for. A Specialization Constant is a way of injecting an integer, Boolean, uint, float, or double constant into a halfway-compiled version of a shader right before the rest-of-the-way compilation.

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

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

---

**Why Do We Need Specialization Constants?**

Specialization Constants could be used for:

- Setting the work-items per work-group in a compute shader
- Setting a Boolean flag and then eliminating the if-test that used it
- Setting an integer constant and then eliminating the switch-statement that looked for it
- Making a decision to unroll a for-loop because the number of passes through it are small enough
- Collapsing arithmetic expressions into a single value
- Collapsing trivial simplifications, such as adding zero or multiplying by 1
Specialization Constants are Described in the Compute Pipeline

In the compute shader:

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

In the Vulkan C/C++ program:

```
int asize = 64;
VkSpecializationMapEntry vsme[1];
vsmes[0].constantID = 7;
// one array element for each
// Specialization Constant
vsmes[0].offset = 0; // # bytes into the Specialization Constant
// array this one item is
vsmes[0].size = sizeof(asize); // size of just this Specialization Constant
VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsmes[0];
vsi.dataSize = sizeof(asize); // size of all the Specialization Constants together
vsi.pData = &asize; // array of all the Specialization Constants
```

In the C/C++ program:

```
struct abc { int a, int b, float c; } abc;
VkSpecializationMapEntry vsme[3];
vsmes[0].constantID = 9;
vsmes[0].offset = offsetof(abc, a);
vsmes[0].size = sizeof(abc.a);
vsmes[1].constantID = 10;
vsmes[1].offset = offsetof(abc, b);
vsmes[1].size = sizeof(abc.b);
vsmes[2].constantID = 11;
vsmes[2].offset = offsetof(abc, c);
vsmes[2].size = sizeof(abc.c);
VkSpecializationInfo vsi;
vsi.mapEntryCount = 3;
vsi.pMapEntries = &vsmes[0];
vsi.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsi.pData = &abc; // array of all the Specialization Constants
```

It's important to use `sizeof( )` and `offsetof( )` instead of hardcoding numbers!
In the compute shader:

```c
layout( local_size_x_id=12 ) in;
layout( local_size_x = 32, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:

```c
int numXworkItems = 64;
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 12;
vsme[0].offset = 0;
vsme[0].size = sizeof(int);
VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(int);
vsii.pData = &numXworkItems;
```

Remember the Overall Block Diagram?

Where Synchronization Fits in the Overall Block Diagram
Semaphores

- Used to synchronize work executing on difference queues within the same logical device
- You create them, and give them to a Vulkan function which sets them. Later on, you tell a Vulkan function to wait on this particular semaphore
- You don’t end up setting, resetting, or checking the semaphore yourself
- Semaphores must be initialized ("created") before they can be used

![Semaphores Diagram](image)

Creating a Semaphore

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

This doesn’t actually do anything with the semaphore – it just sets it up

Semaphores Example during the Render Loop

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

    ;
    ; acquire the next image
    ;
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX,
        IN &imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );

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

Fences

- Used when the host needs to wait for the device to complete something big
- Used to synchronize the application with commands submitted to a queue
- Announces that queue-submitted work is finished
- Much finer control than semaphores
- You can un-signal, signal, test or block-while-waiting
# Fences

```c
#define VK_FENCE_CREATE_UNSIGNALED_BIT 0

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

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

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

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

## Fence Example

```c
VkFence renderFence;
result = vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

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

// Wait on the fence(s)
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```

# Events

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

## Controlling Events from the Host

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

VkEvent event;
result = vkCreateEvent( LogicalDevice, IN &veci, PALLOCATOR, OUT &event );

result = vkSetEvent( LogicalDevice, IN event );
result = vkResetEvent( LogicalDevice, IN event );
result = vkGetEventStatus( LogicalDevice, IN event );
// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled
```

Note: the host cannot block waiting for an event, but it can test for it
Controlling Events from the Device

```c
result = vkCmdSetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdResetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdWaitEvents( CommandBuffer, 1, &event, array of events, srcPipelineStageBits, dstPipelineStageBits, Where signaled, where wait for the signal
memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
```

Memory barriers get executed after events have been signaled.

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

Pipeline Barriers

Mike Bailey
mj@cs.oregonstate.edu

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

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

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

- `vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );`
- `vkCmdNextSubpass( commandBuffer, contents );`
- `vkCmdBeginQuery( commandBuffer, flags );`
- `vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );`
- `vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );`
- `vkCmdBindIndexBuffer( commandBuffer, indexType );`
- `vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );`
- `vkCmdBindPipeline( commandBuffer, pipeline );`
- `vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );`
- `vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );`
- `vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );`
- `vkCmdBindDescriptorSetsWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );`
- `vkCmdBeginRenderPass( commandBuffer, const contents );`
- `vkCmdBlitImage( commandBuffer, filter );`
- `vkCmdResetEvent( commandBuffer, event, stageMask );`
- `vkCmdCopyBuffer( commandBuffer, pRegions );`
- `vkCmdCopyBufferToImage( commandBuffer, pRegions );`
- `vkCmdCopyImage( commandBuffer, pRegions );`
- `vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );`
- `vkCmdCopyImageToBuffer( commandBuffer, pRegions );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDispatchIndirect( commandBuffer, offset );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdDrawIndexedIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDispatchIndirect( commandBuffer, offset );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdDrawIndexedIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDispatchIndirect( commandBuffer, offset );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdDrawIndexedIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDispatchIndirect( commandBuffer, offset );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdDrawIndexedIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
- `vkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );`
- `vkCmdDispatchIndirect( commandBuffer, offset );`
- `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
- `vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );`
- `vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );`
- `vkCmdDrawIndexedIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirect( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdDrawIndirectCountAMD( commandBuffer, stride );`
- `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
- `vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );`
- `vkCmdClearColorImage( commandBuffer, pRanges );`
- `vkCmdClearDepthStencilImage( commandBuffer, pRanges );`
Potential Memory Race Conditions that Pipeline Barriers can Prevent

1. Write-then-Read (WtR) – the memory write in one operation starts overwriting the memory that another operation's read needs to use

2. Read-then-Write (RtW) – the memory read in one operation hasn't yet finished before another operation starts overwriting that memory

3. Write-then-Write (WtW) – two operations start overwriting the same memory and the end result is non-deterministic

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

vkCmdPipelineBarrier() Function Call

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

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

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

The Scenario

1. The cross-streets are named after pipeline stages
2. All traffic lights start out green
3. There are special sensors at all intersections that will know when any car in the src group is in that intersection
4. There are connections from those sensors to the traffic lights so that when any car in the src group is in the intersection, the proper dst traffic light will be turned red
5. When the last car in the src group completely makes it through its intersection, the proper dst traffic light is turned back to green
6. The Vulkan command pipeline ordering is this: (1) the src cars get released, (2) the pipeline barrier is invoked (which turns some light red), (3) the dst cars stop at the red light, (4) the src intersection clears, (5) all lights are now green, (6) the dst cars continue.
Pipeline Stage Masks –
Where in the Pipeline is this Memory Data being Generated or Consumed?

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

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

VK_ACCESS_INDIRECT_COMMAND_READ_BIT
VK_ACCESS_INDEX_READ_BIT
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
VK_ACCESS_UNIFORM_READ_BIT
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
VK_ACCESS_SHADER_READ_BIT
VK_ACCESS_SHADER_WRITE_BIT
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
VK_ACCESS_TRANSFER_READ_BIT
VK_ACCESS_TRANSFER_WRITE_BIT
VK_ACCESS_HOST_READ_BIT
VK_ACCESS_HOST_WRITE_BIT
VK_ACCESS_MEMORY_READ_BIT
VK_ACCESS_MEMORY_WRITE_BIT
Access Operations and what Pipeline Stages they can be used In

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

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

src cars

TOP_OF_PIPE Street
VERTEX_INPUT Street
VERTEX_SHADER Street
BOTTOM_OF_PIPE Street
TRANSFER_BIT Street
COLOR_ATTACHMENT_OUTPUT Street
FRAGMENT_SHADER Street

dst cars

Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan.

Aliasing

The Display We Want
Too often, the Display We Get

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

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

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

VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
VK_IMAGE_LAYOUT_PREINITIALIZED
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR

Used as a color attachment.
Read into a shader as a texture.
Copy from.
Copy to.
Show image to viewer.
Aliasing

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

What the signal really is: what we want

Sampling Interval

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

Sampled Points

The Nyquist Criterion

"The Nyquist [sampling] rate is twice the maximum component frequency of the function [i.e., signal] being sampled." -- WikiPedia

MultiSampling

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

There are two approaches to this:

1. **Supersampling**: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. Render the image at each of these sub-pixels.

2. **Multisampling**: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. If any of them pass, then perform a single color render for the one pixel and assign that single color to all the sub-pixels that passed the depth and stencil tests.

The final step will be to average those sub-pixels' colors to produce one final color for this whole pixel. This is called **resolving** the pixel.
Consider Two Triangles Who Pass Through the Same Pixel

Let’s assume (for now) that the two triangles don’t overlap — that is, they look this way because they butt up against each other.

Supersampling

\[
\text{Final Pixel Color} = \frac{1}{8} \sum \text{Color sample from subpixel}
\]

# Fragment Shader calls = 8
Consider Two Triangles Who Pass Through the Same Pixel

Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?

A: The ideas are all still the same, but the blue one had to deal with 8 sub-pixels (instead of 5 like before). But, the red triangle came along and obsoleted 3 of those blue sub-pixels. Note that the “resolved” image will still turn out the same as before.
Setting up the Image

```cpp
VkPipelineMultisampleStateCreateInfo vpmsci;
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vpmsci.sampleShadingEnable = VK_TRUE;
vpmsci.minSampleShading = 0.5;
vpmsci.pSampleMask = (VkSampleMask *)nullptr;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;
```

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

- 0. produces simple multisampling
- 1. produces complete supersampling

---

### Comments

- **VkAttachmentDescription**
  - `vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;`  
  - `vad[0].samples = VK_SAMPLE_COUNT_8_BIT;`  
  - `vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;`  
  - `vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;`  
  - `vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;`  
  - `vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;`  
  - `vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;`  
  - `vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;`  
  - `vad[0].flags = 0;`

- **VkAttachmentReference**
  - `colorReference.attachment = 0;`  
  - `colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;`
  
- **VkAttachmentReference**
  - `depthReference.attachment = 1;`  
  - `depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;`

---

### Diagrams

1. **Diagram 1**: Shows the setup of the `VkPipelineMultisampleStateCreateInfo` and `VkGraphicsPipelineCreateInfo`.
2. **Diagram 2**: Illustrates the setting up of the image, including multisampling configurations.
3. **Diagram 3**: Demonstrates the attachment descriptions and references.
4. **Diagram 4**: Highlights the creation of a render pass.
Resolving the Image:
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT Image

![Diagram]

Multipass Rendering
Mike Bailey
mjb@cs.oregonstate.edu

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

What is an Example of Wanting to do This?

There is a process in computer graphics called **Deferred Rendering**. The idea is that a game-quality fragment shader takes a long time (relatively) to execute, but, with all the 3D scene detail, a lot of the rendered fragments are going to get z-buffered away anyhow. So, why did we invoke the fragment shaders so many times when we didn't need to?

Here's the trick:

Let's create a grossly simple fragment shader that writes out (into multiple framebuffers) each fragment's:

- position (x,y,z)
- normal (n_x,n_y,n_z)
- material color (r,g,b)
- texture coordinates (s,t)
- the current light source positions and colors
- the current eye position

When we write these out, the final framebuffers will contain just information for the pixels that can be seen. We then make a second pass running the expensive lighting model just for those pixels. This known as the **G-buffer Algorithm**.
So far, we’ve only performed single-pass rendering, within a single Vulkan RenderPass.

Here comes a quick reminder of how we did that. Afterwards, we will extend it.

```
VkAttachmentDescription vad[2];

vad[0].flags = 0;
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

vad[1].flags = 0;
vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

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

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

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

```c
vad[0].flags = 0;
vad[0].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
vsd[0].flags = 0;
vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
vsd[0].inputAttachmentCount = 0;
vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
vsd[0].pDepthStencilAttachment = (VkAttachmentReference *)nullptr;
vsd[0].preserveAttachmentCount = 0;
vsd[0].pPreserveAttachments = (uint32_t *) nullptr;
```

Multipass, II

```c
depthOutput.attachment = 0; // depth
depthOutput.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
gBufferInput.attachment = 0; // gbuffer
gBufferInput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
lightingInput[0].attachment = 0; // depth
lightingInput[0].layout = VK_IMAGE_LAYOUT_SHADER_READ_OPTIMAL;
lightingInput[1].attachment = 1; // gbuffer
lightingInput[1].layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Multipass, III

```c
lightingOutput.attachment = 2; // color rendering
lightingOutput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Multipass, IV

```c
// Notice how similar this is to creating a Directed Acyclic Graph (DAG).```

Notice how similar this is to creating a Directed Acyclic Graph (DAG).
VkRenderPassCreateInfo

vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
vrpci.pNext = nullptr;
vrpci.flags = 0;
vrpci.attachmentCount = 3; // depth, gbuffer, output
vrpci.pAttachments = vad;
vrpci.subpassCount = 3;
vrpci.pSubpasses = vsd;
vrpci.dependencyCount = 2;
vrpci.pDependencies = vsdp;

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

vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// subpass #0 is automatically started here
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #1 is started here
vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #2 is started here
vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
The Ray-trace Pipeline Involves Five New Shader Types

- **Ray Generation Shader** (rgen) runs on a 2D grid of threads. It begins the entire ray-tracing operation.
- **Intersection Shader** (rint) implements ray-primitive intersections.
- **Any Hit Shader** (rahit) is called when the Intersection Shader finds a hit.
- The **Closest Hit Shader** (rchit) is called with the information about the hit that happened closest to the viewer. Typically lighting is done here, or firing off new rays to handle reflection and refraction.
- **Miss Shader** (rmiss) is called when no intersections are found for a given ray. Typically it just sets its pixel color to the background color.

### Ray Intersection Process for a Sphere

1. Sphere equation: \((x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2\)
2. Ray equation: \((x,y,z) = (x_0,y_0,z_0) + t(dx,dy,dz)\)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying-through and simplifying gives:

\[A t^2 + B t + C = 0\]

Solve for \(t_1, t_2\)

- If both \(t_1\) and \(t_2\) are complex, then the ray missed the sphere.
- If \(t_1 = t_2\), then the ray brushed the sphere at a tangent point.
- If both \(t_1\) and \(t_2\) are real and different, then the ray entered and exited the sphere.

In Vulkan terms:

\[
\begin{align*}
&\text{gl}_\text{WorldRayOriginKHR} = (x_0,y_0,z_0) \\
&\text{gl}_\text{WorldRayDirectionKHR} = (dx,dy,dz)
\end{align*}
\]
The Ray Intersection Process for a Cube

1. Plane equation: \(Ax + By + Cz + D = 0\)
2. Ray equation: \((x,y,z) = (x_0,y_0,z_0) + t(dx,dy,dz)\)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying-through and simplifying gives:

\[At + B = 0\]

Solve for \(t\)

A cube is actually the intersection of 6 half-space planes (just 4 are shown here). Each of these will produce its own \(t\) intersection value. Treat them as pairs: \((t_{x1},t_{x2})\), \((t_{y1},t_{y2})\), \((t_{z1},t_{z2})\)

The ultimate entry and exit values are:

\[
t_{\text{min}} = \max\{\min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2})\}
\]

\[
t_{\text{max}} = \min\{\max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2})\}
\]

Creating Bottom Level Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
  - The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
  - Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.
  - The BLAS is used as a Model Coordinate bounding box.
  - The TLAS is used as a World Coordinate bounding box.
  - A TLAS can instance multiple BLAS’s.

In a Raytracing, each ray typically hits a lot of Things

Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
  - The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
  - Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.
    - The BLAS is used as a Model Coordinate bounding box.
    - The TLAS is used as a World Coordinate bounding box.
    - A TLAS can instance multiple BLAS’s.

Creating Bottom Level Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
  - The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
  - Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.
    - The BLAS is used as a Model Coordinate bounding box.
    - The TLAS is used as a World Coordinate bounding box.
    - A TLAS can instance multiple BLAS’s.

Creating Bottom Level Acceleration Structures

- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
  - The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
  - Top-level Acceleration Structure (TLAS) holds a pointer to elements of the BLAS and a transformation.
    - The BLAS is used as a Model Coordinate bounding box.
    - The TLAS is used as a World Coordinate bounding box.
    - A TLAS can instance multiple BLAS’s.
Creating Top Level Acceleration Structures

```cpp
vkCreateAccelerationStructureKHR TopLevelAccelerationStructure;

VkAccelerationStructureInfoKHR vasi;
vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
vasi.flags = 0;
vasi.pNext = nullptr;
vasi.instanceCount = << number of bottom level acceleration structure instances >>;
vasi.geometryCount = 0;
vasi.pGeometries = VK_NULL_HANDLE;

VkAccelerationStructureCreateInfoKHR vasci;
vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
vasci.pNext = nullptr;
vasci.info = &vasi;
vasci.compactedSize = 0;

result = vkCreateAccelerationStructureKHR( LogicalDevice, &vasci, PALLOCATOR, &TopLevelAccelerationrStructure
```

Ray Generation Shader

```cpp
layout( location = 1 ) rayPayloadKHR myPayLoad
{
vec4 color;
};

void main()
{
traceKHR( topLevel, …, 1 );
imageStore( framebuffer, gl_GlobalInvocationIDKHR.xy, color );
}

A "payload" is information that keeps getting passed through the process. Different stages can add to it. It is finally consumed at the very end, in this case by writing color into the pixel being worked on.
```

A New Built-in Function

```cpp
void traceKHR
(
 accelerationStructureKHR topLevel,
 uint rayFlags,
 uint cullMask,
 uint sbtRecordOffset,
 uint sbtRecordStride,
 uint missIndex,
 vec3 origin,
 float tmin, vec3 direction,
 float tmax,
 int payload
);
```

Intersection Shader

```cpp
Intersect a ray with an arbitrary 3D object. Passes data to the Any Hit shader. There is a built-in ray-triangle Intersection Shader.
```
Miss Shader
Handle a ray that doesn't hit any objects

```glsl
rayPayloadKHR myPayLoad {
    vec4 color;
};
void main() {
    color = vec4( 0.0, 0.0, 0.0, 1.0 );
}
```

Any Hit Shader
Handle a ray that hits anything. Store information on each hit. Can reject a hit.

```glsl
layout( binding = 4, set = 0) buffer outputProperties {
    float outputValues[ ];
} outputData;
layout(location = 0) rayPayloadInKHR uint outputId;
layout(location = 1) rayPayloadInKHR uint hitCounter;
hitAttributeKHR vec 3 attribs;
void main() {
    outputData.outputValues[ outputId + hitCounter ] = gl_PrimitiveID;
    hitCounter = hitCounter + 1;
}
```

Closest Hit Shader
Handle the intersection closest to the viewer. Collects data from the Any Hit shader. Can spawn more rays.

```glsl
rayPayloadKHR myPayLoad {
    vec4 color;
};
void main() {
    vec3 stp = gl_WorldRayOriginKHR + gl_HitKHR * gl_WorldRayDirectionKHR;
    color = texture( MaterialUnit, stp );  // material properties lookup
}
```

Other New Built-in Functions
Loosely equivalent to “discard”

```glsl
void terminateRayKHR( );
void ignoreIntersectionKHR( float hit, uint hitKind );
void reportIntersectionKHR( float hit, uint hitKind );
```
Ray Trace Pipeline Data Structure

```
Ray Trace Pipeline Data Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPipeline</td>
<td>RaytracePipeline</td>
</tr>
<tr>
<td>VkPipelineLayout</td>
<td>PipelineLayout</td>
</tr>
<tr>
<td>VkPipelineLayoutCreateInfo</td>
<td>VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.sType</td>
<td>= VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.pNext</td>
<td>= nullptr;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.flags</td>
<td>= 0;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.setLayoutCount</td>
<td>= 1;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.pSetLayouts</td>
<td>= &amp;descriptorSetLayout; microwave PipelineLayoutCreateInfo;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.pushConstantRangeCount</td>
<td>= 0;</td>
</tr>
<tr>
<td>vkpPipelineLayoutCreateInfo.pPushConstantRanges</td>
<td>= nullptr;</td>
</tr>
</tbody>
</table>
```

```
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, nullptr, OUT &PipelineLayout );
```

```
VkRayTracingPipelineCreateInfoKHR vrtpci;
vrtpci.sType = VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
vrtpci.pNext = nullptr;
vrtpci.flags = 0;
vrtpci.stageCount = << # of shader stages in the ray-trace pipeline >>;
vrtpci.pStages = << what those shader stages are >>;
vrtpci.groupCount = << # of shader groups >>;
vrtpci.pGroups = << pointer to the groups (a group is a combination of shader programs) >>;
vrtpci.maxRecursionDepth = << how many recursion layers deep the ray tracing is allowed to go >>;
vrtpci.layout = PipelineLayout; vrtpci.basePipelineHandle = VK_NULL_HANDLE;
vrtpci.basePipelineIndex = 0;
result = vkCreateRayTracingPipelinesKHR( LogicalDevice, PALLOCATOR, 1, IN &rvrtpci, nullptr, OUT &RaytracePipeline );
```

The Trigger comes from the Command Buffer: `vlCmdBindPipeline( ) and vkCmdTraceRaysKHR( )`

```
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_RAYTRACING_KHR, RaytracePipeline );
```

```
vkCmdTraceRaysKHR( CommandBuffer, raygenShaderBindingTableBuffer, raygenShaderBindingOffset, missShaderBindingTableBuffer, missShaderBindingOffset, missShaderBindingStride, hitShaderBindingTableBuffer, hitShaderBindingOffset, hitShaderBindingStride, callableShaderBindingTableBuffer, callableShaderBindingOffset, callableShaderBindingStride, width, height, depth );
```

https://www.youtube.com/watch?v=QL7sXc2iNJ8

Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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

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