The Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

SIGGRAPH '23 Courses
August 06-10, 2023
Los Angeles, CA, USA
ACM 978-8-4007-0145-0/23/08.
10.1145/3587423.3595529

Mike Bailey
• Professor of Computer Science, Oregon State University
• Has been in computer graphics for over 30 years
• Has had over 11,000 students in his university classes
• Has taught over 100 conference and workshop short courses
• mjb@cs.oregonstate.edu

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

Course Goals
• Better understand what Vulkan is and when it would be a good application solution (and when it wouldn't)
• Leave you with working, understandable Vulkan code
• Understand the inner workings of Vulkan, especially the parts that are different from OpenGL and Direct X 11

Introduction

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 always for your courage and patience!

Second, thanks to NVIDIA for all of your support! These courses could not have ever happened without you!

Third, thanks to the Khronos Group for the great Vulkan teaching materials and other swag! (Look at those happy faces in the photo holding the reference cards.)

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?

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

Who's Been Specifically Working on Vulkan?
Vulkan

- 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

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.

Moving part of the driver into the application

Complex drivers lead to driver overhead and cross vendor unpredictability
Error management is always active
Driver processes full shading language source
Separate APIs for desktop and mobile markets

Vulkan Reference Card – I Recommend you Get and Print This!

Even though we are up to Vulkan 1.3, the Reference Card is 1.1

Even though we are up to Vulkan 1.3, the Reference Card is 1.1


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

Your Sample2019-COLOREDCUBE.zip File Contains This

Sample Program Output

Sample Program Keyboard Inputs

'l' (ell), 'L': Toggle lighting off and on
'm', 'M': Toggle display mode (textures vs. colors, for now)
'p', 'P': Pause the animation
'q', 'Q': quit the program
Esc: quit the program
'y', 'Y': Toggle rotation-animation and using the mouse
'I', 'I': Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)
'I', 'i': 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 typedefed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan_core.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 = 1024;
    Height = 1024;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE );
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
    Reset();
    InitGraphics();
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents();
        Time = glfwGetTime();          // elapsed time, in double-precision seconds
        UpdateScene();
        RenderScene();
    }
    fprintf(FpDebug, "Closing the GLFW window
");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan();
    glfwDestroyWindow( MainWindow );
    glfwTerminate();
    return 0;
}
```

InitGraphics( ), I

```c
void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndGetQueueFamilyProperties( );
    Init04LogicalDeviceAndQueue( );
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool( );
    Init06CommandBuffers( );
```
InitGraphics( ), II

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 );

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 typedefed structs pass all the necessary information into a function. For example, where we might normally say, using C++ class methods:

```cpp
result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex,  OUT &Queue );
```
Vulkan has chosen to do it like this:

```cpp
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 InitBoxYYY( ) 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

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

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

```cpp
result = vkEnumeratePhysicalDevices( Instance, OUT &count, NULL );
result = vkEnumeratePhysicalDevices( Instance, OUT &count, &physicalDevices[0] );
```
Your Sample2019-COLOREDCUBE.zip File Contains This

The "19" refers to the version of Visual Studio, not the year of development.

Vulkan Program Flow – the Setup

- Create a GLFW Vulkan Window
- Query the Physical Devices and Choose (1 in our case)
- Decide on the Extensions and Layers You Want
- Create the Logical Device
- Create the Queue(s) (1 in our case)
- Allocate and Fill memory for the Vertices and Indices
- Allocate and Fill memory for the Uniform Buffers
- Create the Command Buffers (3 in our case)
  - If using Textures, create the Sampler, Read the Texture, and move it to Device Local Memory
- Create the Swap Chain (2 images in our case)
  - Be sure you have Compiled the Shaders into .spv files
- Create the Descriptor Set Data Structures
- Fill the Graphics Pipeline Data Structure(s)

Vulkan Program Flow – the Rendering Loop

while( the GLFW Window should not close )
{
    UpdateScene();
    RenderScene();
}

Create the Transformations
Fill the Uniform Buffers

Acquire the next Swap Chain Image

Begin its Command Buffer
Create the RenderPass with the Framebuffer information

for( all the different Graphics Pipeline Data Structures being used )
{
    Bind that Graphics Pipeline Data Structure
    Set any Dynamic State Variables
    Bind the Proper Descriptor Set Values
    Do the Drawing
}

End the RenderPass
End the Command Buffer
Submit the Command Buffer to a Queue
Wait for the Queue to Finish Submitting

Present the Image to the Viewer

**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

The same as OpenGL topologies, with a few left out.

**typedef enum VkPrimitiveTopology**

```c
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

This data is contained in the file `SampleVertexData.cpp`

```c
static GLuint CubeIndices[3][3] = {
    { 0, 2, 3 },
    { 0, 3, 1 },
    { 4, 5, 7 },
    { 4, 7, 6 },
    { 1, 0, 1 },
    { 0, 1, 2 },
    { 1, 5, 2 },
    { 0, 4, 6 },
    { 0, 6, 2 },
    { 2, 6, 7 },
    { 2, 7, 3 },
    { 6, 1, 9 },
    { 6, 9, 4 }
};
```

Modeled in right-handed coordinates.

From the `SampleVertexData.cpp`

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

```c
static GLuint CubeIndices[3][3] = {
    { 0, 2, 3 },
    { 0, 3, 1 },
    { 4, 5, 7 },
    { 4, 7, 6 },
    { 1, 0, 1 },
    { 0, 1, 2 },
    { 1, 5, 2 },
    { 0, 4, 6 },
    { 0, 6, 2 },
    { 2, 6, 7 },
    { 2, 7, 3 },
    { 6, 1, 9 },
    { 6, 9, 4 }
};
```

This data is contained in the file `SampleVertexData.cpp`
Non-indexed Buffer Drawing

Stream of Vertices

A Preview of What `Init05DataBuffer` Does

Telling the Pipeline about its Input

Always use the C/C++ `sizeof()` construct rather than hardcoding the byte count!
Telling the Pipeline about its Input

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

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

Always use the C/C++ construct `offsetof`, rather than hardcoding the byte offset!

Telling the Pipeline Data Structure about its Input

We will come to the Pipeline Data Structure 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.

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

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

Always use the C/C++ construct `offsetof`, rather than hardcoding the byte offset!

Telling the Command Buffer what Vertices to Draw

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

```cpp
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
```

Always use the C/C++ construct `sizeof`, rather than hardcoding a byte count!
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. },
  {  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, 6, 7,
  2, 7, 3,
  0, 1, 5,
  0, 5, 4,
};

Drawing with an Index Buffer

vkCmdBindVertexBuffer( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );

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

Drawing with an Index Buffer

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

Init05MyVertexDataBuffer( sizeof(JustVertexData), IN &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
Init05MyIndexDataBuffer( sizeof(JustIndexData), IN &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );

Drawing with an Index Buffer

vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );

Drawing with an Index Buffer

VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer }
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer }
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32 );
const uint32_t vertexCount = sizeof( JustVertexData ) / sizeof( JustVertexData[0] );
const uint32_t indexCount = sizeof( JustIndexData )  / sizeof( JustIndexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
const uint32_t vertexOffset = 0;
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
Sometimes a vertex 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. Vertex #7 above has the same color, regardless of what face it is in. However, Vertex #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.

Terrain Surfaces are a Great Application of Indexed Drawing

But, to Draw that Terrain Surface, You Need “Primitive Restart”

“Primitive Restart” is used with:
- Indexed drawing
- TRIANGLE_FAN and TRIANGLE_STRIP topologies

A special “index” is used to indicate that the triangle strip should start over. This is more efficient than explicitly ending the current triangle strip and explicitly starting a new one.

```c
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 restart index is 0xffff.
If your VkIndexType is VK_INDEX_TYPE_UINT32, then the restart index is 0xffffffff.

That is, a one in all available bits.
The OBJ File Format – a triple-indexed way of Drawing

V / T / N

We have a `vkLoadObjFile()` function to load an OBJ file into your Vulkan program!

Data Buffers

From the Reference Card

Even though Vulkan is up to 1.3, the most current Vulkan Reference card is version 1.1

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 so far as to extend some of Vulkan’s own terminology:

typedef VkBuffer VkDataBuffer;

This is probably a bad idea in the long run.

### Terminology Issues

Creating a Vulkan Data Buffer

```c
VkBuffer Buffer;  // or “VkDataBuffer Buffer”
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or’ed bits of: >>
    VK_USAGE_TRANSFER_SRC_BIT
    VK_USAGE_TRANSFER_DST_BIT
    VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
    VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
    VK_USAGE_UNIFORM_BUFFER_BIT
    VK_USAGE_STORAGE_BUFFER_BIT
    VK_USAGE_INDEX_BUFFER_BIT
    VK_USAGE_VERTEX_BUFFER_BIT
vbci.sharingMode = << one of: >>
    VK_SHARING_MODE_EXCLUSIVE
    VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &Buffer );
```

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

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

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
// FindMemoryThatIsHostVisible() ... ;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible() ;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR,  OUT &vdm );

result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset...

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );
<< do the memory copy >>
result = vkUnmapMemory( LogicalDevice, IN vdm );
```
int FindMemoryThatIsHostVisible()
{
    VkPhysicalDeviceMemoryProperties vpdmp;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
    for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
    {
        VkMemoryType vmt = vpdmp.memoryTypes[i];
        if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )
        {
            return i;
        }
    }
    return -1;
}

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

Finding the Right Type of Memory

Memory Types:
- Memory 0: DeviceLocal
- Memory 1: HostVisible HostCoherent
- Memory 2: HostVisible HostCoherent HostCached
- Memory 3: DeviceLocal HostVisible HostCoherent
- Memory 4: DeviceLocal HostVisible HostCoherent
- Memory 5: DeviceLocal

Memory Heaps:
- Heap 0: size = 0xdbb00000 DeviceLocal
- Heap 1: size = 0xd504000
- Heap 2: size = 0x0d600000 DeviceLocal
- Heap 3: size = 0x02000000 DeviceLocal

These are the numbers for the Nvidia A6000 cards.

Memory-Mapped Copying to GPU Memory, Example I

void *mappedDataAddr;
vkMapMemory( LogicalDevice, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&mappedDataAddr );
memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );
vkUnmapMemory( LogicalDevice, myBuffer.vdm );
struct vertex *vp;

vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&vp );

for( int i = 0; i < numTrianglesInObjFile; i++ ) // number of triangles
    for( int j = 0; j < 3; j++ ) // 3 vertices per triangle
        {  
            vp->position = glm::vec3( . . . );
            vp->normal = glm::vec3( . . . );
            vp->color = glm::vec3( . . . );
            vp->texCoord = glm::vec2( . . . );
            vp++;
        }

vkUnmapMemory( LogicalDevice, myBuffer.vdm );

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. 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 also includes a smattering of documentation.

See our class VMA noteset for more VMA details

#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"

VkBufferCreateInfo vbci;

VmaAllocationCreateInfo vaci;

vaci.physicalDevice = PhysicalDevice;

VmaAllocator var;

vmaCreateMemory( IN &var, IN &vbci, IN &vaci, OUT &buffer, OUT &van, nullptr );

void *mappedDataAddr;

vmaMapMemory( var, van, OUT &mappedDataAddr );

memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );

vmaUnmapMemory( var, van );

typedef struct MyBuffer
{
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size; // in bytes
} MyBuffer;

#pragma once

// example:
MyBuffer MyObjectUniformBuffer;

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

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

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

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

```c
VkResult Init05DataBuffer(VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer *pMyBuffer)
{
    . . .
    vbci.size = pMyBuffer->size = size;
    . . .
    result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer);
    . . .
    pMyBuffer->vdm = vdm;
    . . .
}
```

Here are C/C++ structs used by the Sample Code to hold some uniform variables

Here’s the associated GLSL shader code to access those uniform variables:

```glsl
The uNormal is set to:
glm::inverseTranspose( uView * uSceneOrient * uModel )
```

In the vertex shader, each object vertex gets transformed by:

```glsl
uProjection* uView * uSceneOrient * uModel
```

In the vertex shader, each surface normal vector gets transformed by the `uNormal`

Filling those Uniform Variables

```c
const float EYEDIST = 3.0f;
const double FOV = glm::radians(60.);      // field-of-view angle in radians
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Scene.uProjection = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Scene.uProjection[1][1] *= -1.; // account for Vulkan’s LH screen coordinate system
Scene.uView = glm::lookAt( eye, look, up );
Scene.uSceneOrient = glm::mat4( 1. );
Object.uModelOrient = glm::mat4( 1. );              // identity
Object.uNormal = glm::inverseTranspose( Scene.uView * Scene.uSceneOrient * Object.uModel )
```

This code assumes that this line:

```c
#define GLM_FORCE_RADIANS
```

is listed before GLM is #included!

The Parade of Buffer Data

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

This C struct is holding the original data, written by the application.

```c
struct objectBuf Object;
```

This struct is holding the original data, written by the application.

```c
Object.uModelOrient = glm::mat4( 1. );              // identity
Object.uNormal = glm::inverseTranspose( Scene.uView * Scene.uSceneOrient * Object.uModel )
```

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

```c
define GLM_FORCE_RADIANS
```
Creating and Filling the Data Buffer – the Details

```c
VkResult
Init05UniformBuffer( sizeof(Object), OUT &MyObjectUniformBuffer );
Fill05DataBuffer( MyObjectUniformBuffer, IN (void *)&Object );
```

```c
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);
    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;
}

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

Shaders and SPIR-V

Remember – to Vulkan and GPU memory, these are just bits. It is up to you to handle their meaning correctly.
### The Shaders’ View of the Basic Computer Graphics Pipeline

- Fixed Function
- Programmable

- You need 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

#### Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:
- In the compiler, there is an automatic `#define VULKAN 130` or whatever the current version number is. Typically, you use this like:
  ```c
  #ifdef VULKAN
  ...
  #endif
  ```
- OpenGL uses:
  ```c
  gl_VertexID
  gl_InstanceID
  ```

#### Vulkan Vertex and Instance indices:
- Both are 0-based

#### How Vulkan GLSL Differences from OpenGL GLSL

- **Shader combinations of separate texture data and samplers as an option:**
  ```c
  uniform sampler s;
  uniform texture2D t;
  vec4 rgba = texture( sampler2D( t, s ), vST );
  ```

- **Descriptor Sets:**
  ```c
  layout( set=0, binding=0 ) . . . ;
  ```

- **Push Constants:**
  ```c
  layout( push_constant ) . . . ;
  ```

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

- **For example, Specialization Constants can be used with Compute Shaders:**
  ```c
  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

- **Note:** Our sample code doesn’t use this.
Vulkan: Shaders’ use of Layouts for Uniform Variables

```c
vkCreateShaderModule(
    device,    
    &VkShaderModuleCreateInfo(
        code, codeSize, 
        shaderModuleCreateFlags, 
        layout( std140, set = 0, binding = 0 ) uniform sceneMatBuf, 
        layout( std140, set = 1, binding = 0 ) uniform objectMatBuf, 
        layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;
    )
);
```

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

### Vulkan Shader Compiling

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

---

#### 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

- **Options:**
  - -V: Compile for Vulkan
  - -G: Compile for OpenGL
  - -I: Directory to look in for #includes
  - -S: Specify stage rather than get it from shaderfile extension
  - -c: Print out the maximum sizes of various properties

---

### SPIR-V: Vendor-Specific Code

GLS Source → **External GLSL Compiler** → SPIR-V → **Compiler in driver** → Vendor-specific code

**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

---

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

You can run the glslangValidator program from the Windows Command Prompt, but I have found it easier to run the SPIR-V compiler from Windows-Bash.

To install the bash shell on your own Windows machine, go to this URL:


Or, follow these instructions:

1. Head to the Start menu search bar, type in ‘terminal,’ and launch the Windows Terminal as administrator. (On some systems, this is called the Command Prompt.)
2. Type in the following command in the administrator: `wsl --install`
3. Restart your PC once the installation is complete.

As soon as your PC boots up, the installation will begin again. Your PC will start downloading and installing the Ubuntu software. You’ll soon get asked to set up a username and password. This can be the same as your system’s username and password, but doesn’t have to be. The installation will automatically start off from where you left it.
#ifndef _WIN32
typedef int errno_t;
int     fopen_s( FILE**, const char *, const char * );
#endif

#define SPIRV_MAGIC             0x07230203

VkResult Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule ) {
    FILE *fp;
    #ifdef WIN32
        errno_t err = fopen_s( &fp, filename.c_str(), "rb" );
        if( err != 0 )
    #else
        fp = fopen( filename.c_str(), "rb" );
        if( fp == NULL )
    #endif
    {
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str() );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
    {
        fprintf( FpDebug, "Magic number for spir-v file '%s is 0x%08x -- should be 0x%08x
", filename.c_str(), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char * code = new unsigned char [size];
    fread( code, size, 1, fp );
    fclose( fp );
    // ...
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:

```bash
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:

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

This causes a:

```c
#define NUMPOINTS 4
```

to magically be inserted into the top of your source code.

Instancing – What and why?

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

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

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

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

Making each Instance look differently -- Approach #1

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

```
gl_InstanceIndex
```

```
gl_InstanceIndex starts at 0
```

In the vertex shader:

```gl
float DELTA = 3.0;
float s = sqrt( float(Sporadic.uNumInstances));
float c = ceil( float(s));
int cols = int(c);
int fullRows = gl_InstanceIndex/cols;
int remainder = gl_InstanceIndex%cols;
float xdelta = DELTA*float(remainder);
float ydelta = DELTA*float(fullRows);
vColor = vec3(1., float(1.+gl_InstanceIndex), 0.);
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
```

In the vertex shader:
Put the unique characteristics in a uniform buffer array and reference them

Still uses `gl_InstanceIndex`

In the vertex shader:

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

Referencing that Table as a Uniform Buffer

```glsl
struct atom {
    vec3 position;
    int atomicNumber;
};
```

The Transformation Setup

```glsl
void main() {
    mat4 P = Scene.uProjection;
    mat4 V = Scene.uView;
    mat4 SO = Scene.uSceneOrient;
    mat4 M = Object.uModel;
    mat4 VM = V * SO * M;
    mat4 PVM = P * VM;
    // surface normal vector
    vec3 vN = normalize(mat3(Object.uNormal) * aNormal);
    vec3 lightPos = vec4(Scene.uLightPos.xyz, 1.0); // light source in fixed location because not transformed
    vec3 vL = normalize(lightPos.xyz - ECposition.xyz); // vector from the point to the light
    vec3 eyePos = vec4(0., 0., 0., 1.0); // eye position after applying the viewing matrix
    vec3 vE = normalize(eyePos.xyz - ECposition.xyz); // vector from the point to the eye
    // other code...
}
```
Using the gl_InstanceIndex Variable

```c
int atomicNumber = atoms[gl_InstanceIndex].atomicNumber;
vec3 position = atoms[gl_InstanceIndex].position;
float radius;

if (atomicNumber == 1) {
    radius = 0.37;
    vColor = vec3(1.,1.,1.);
} else if (atomicNumber == 6) {
    radius = 0.77;
    vColor = vec3(0.,1.,0.);
} else if (atomicNumber == 7) {
    radius = 0.70;
    vColor = vec3(0.,0.,1.);
} else if (atomicNumber == 8) {
    radius = 0.66;
    vColor = vec3(1.,0.,0.);
} else {
    radius = 0.75;
    vColor = vec3(1.,0.,1.);    // big magenta ball to tell us something is wrong
}
vec3 bVertex = aVertex;
bVertex.xyz *= radius;
bVertex.xyz += position;
gl_Position = PVM * vec4(bVertex, 1.);
```

GLFW

```
#include <glfw3.h>

uint32_t Width, Height;
VkSurfaceKHR Surface;

void InitGLFW() {
    glfwInit;
    if (!glfwVulkanSupported) {
        fprintf(stderr, "Vulkan is not supported on this system!
");
        exit(1);
    }
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
    MainWindow = glfwCreateWindow(Width, Height, "Vulkan Sample", NULL, NULL);
    VkResult result = glfwCreateWindowSurface(Instance, MainWindow, NULL, OUT &Surface);
    glfwSetErrorCallback(GLFWErrorCallback);
    glfwSetKeyCallback(MainWindow, GLFWKeyboard);
    glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
    glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

Setting Up GLFW
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, "\nFound %d GLFW Required Instance Extensions:
", count);
for( uint32_t i = 0; i < count; i++ )
    fprintf( FpDebug, "\t%s\n", extensions[i] );
```

Found 2 GLFW Required Instance Extensions:

- VK_KHR_surface
- VK_KHR_win32_surface

GLFW Keyboard Callback

```c
void GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
    if( action == GLFW_PRESS )
    {
        switch( key )
        {
            //case GLFW_KEY_M:
            case 'm':
                Module++; // keep object from turning inside-out or disappearing:
                break;
            default:
                fprintf( FpDebug, "Unknown key hit: 0x%04x = '%c'
", key, key);
                fflush(FpDebug);
                break;
        }
    }
}
```

GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

```c
void GLFWMouseMotion( GLFWwindow * window, double xpos, double ypos )
{
    int dx = (int)xpos - Xmouse; // change in mouse coords
    int dy = (int)ypos - Ymouse;
    if( (ActiveButton & LEFT) != 0 )
    {
        Xrot += (ANGFACT * dy);
        Yrot += (ANGFACT * dx);
    }
    if( (ActiveButton & MIDDLE) != 0 )
    {
        Scale += SCLFACT * (float)(dx - dy);
        if( Scale < MINSCALE ) Scale = MINSCALE;
    }
    Xmouse = (int)xpos; // new current position
    Ymouse = (int)ypos;
}
```
While ( \text{glfwWindowShouldClose} \ ( \text{MainWindow} ) == 0 )
{
    \text{glfwPollEvents} ( );
    \text{Time} = \text{glfwGetTime} ( );  // elapsed time, in double-precision seconds
    \text{UpdateScene} ( );
    \text{RenderScene} ( );
}
\text{vkQueueWaitIdle} ( \text{Queue} );
\text{vkDeviceWaitIdle} ( \text{LogicalDevice} );
\text{DestroyAllVulkan} ( );
\text{glfwDestroyWindow} ( \text{MainWindow} );
\text{glfwTerminate} ( );

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

If you would like to block waiting for events, use:
\text{glfwWaitEvents} ( );

You can have the blocking wake up after a timeout period with:
\text{glfwWaitEventsTimeout} ( \text{double secs} );

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

\text{#define} \quad \text{GLM_FORCE_RADIANS}
\text{#include} \quad \text{glm/glm.hpp}
\text{#include} \quad \text{glm/gtc/matrix_transform.hpp}
\text{#include} \quad \text{glm/gtc/matrix_inverse.hpp}

OpenGL treats all angles as given in degrees. This line forces GLM to
treat all angles as given in radians.

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. );
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}
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.)); // {x',y',z'} = [v]*[yr]*{x,y,z}
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.)); // {x',y',z'} = [v]*[yr]*[xr]*{x,y,z}
modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) ); // {x',y',z'} = [v]*[yr]*[xr]*[s]*{x,y,z}
```

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

---

### The Most Useful GLM Variables, Operations, and Functions

#### GLM in the Vulkan sample.cpp Program

```c
if( UseMouse )
{
    if( Scale < MINSCALE )
        Scale = MINSCALE;
    Matrices.uModelMatrix = glm::mat4( 1. );           // identity
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0. ) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0. ) );
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
    // done this way, the Scale is applied first, then the Xrot, then the Yrot
}
else
{
    if( ! Paused )
    {
        const glm::vec3 axis = glm::vec3( 0., 1., 0. );
        Matrices.uModelMatrix = glm::rotate( glm::mat4( 1. ), (float)glm::radians( 360.f*Time/SECONDS_PER_CYCLE ), axis );
    }
}
```

```c
glm::vec3 eye(0.,0.,EYEDIST );
glm::vec3 look(0.,0.,0.);
glm::vec3   up(0.,1.,0.);
Matrices.uVewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1f, 1000.f );
Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix );

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

#### GLM in the Vulkan sample.cpp Program

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

glm::mat4 glm::ortho( float left, float right, float bottom, float top, float near, float far );   // viewing (assign, not concatenate):

glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );
```
Here's the vertex shader shader code to use the matrices:

```glsl
vNormal = uNormalMatrix * aNormal;

gl_Position = uProjectMatrix * uViewMatrix * uSceneMatrix * uModelMatrix * aVertex;
```

Descriptor Sets

In OpenGL

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

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

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

```glsl
layout( std140, binding = 0 ) uniform mat4 uModelMatrix;
layout( std140, binding = 1 ) uniform mat4 uViewMatrix;
layout( std140, binding = 2 ) uniform mat4 uProjectionMatrix;
layout( std140, binding = 3 ) uniform mat3 uNormalMatrix;
layout( std140, binding = 4 ) uniform vec4 uLightPos;
layout( std140, binding = 5 ) uniform float uTime;
layout( std140, binding = 6 ) uniform int uMode;
layout( std140, binding = 7 ) uniform sampler2D uSampler;
```

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

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

```glsl
for ( sporadically )
{
    Bind Descriptor Set #0
    for( the entire scene )
    {
        Bind Descriptor Set #1
        for( each object in the scene )
        {
            Bind Descriptor Set #2
            Do the drawing
        }
    }
}
```

What are Descriptor Sets?

`std140` has to do with the alignment of the different data types. It is the simplest, and so we use it in class to give everyone the highest probability that their system will be compatible with the alignment.
Descriptor Sets

Our example will assume the following shader uniform variables:

```c++
struct sporadicBuf
{
  int uMode;
  int uUseLighting;
  int uNumInstances;
} Sporadic;

struct sceneBuf
{
  glm::mat4 uProjection;
  glm::mat4 uView;
  glm::mat4 uSceneOrient;
  glm::vec4 uLightPos;
  glm::vec4 uLightColor;
  glm::vec4 uLightKaKdKs;
  float uTime;
} Scene;

struct objectBuf
{
  glm::mat4 uModel;
  glm::mat4 uNormal;
  glm::vec4 uColor;
  float uShininess;
} Object;
```

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 during initialization and then pull from that pool later.
Step 2: Define the Descriptor Set Layouts

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

### Sporadic Set DS Layout Binding:

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Object Set DS Layout Binding:

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Scene Set DS Layout Binding:

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TexSamplerSet DS Layout Binding:

<table>
<thead>
<tr>
<th>binding</th>
<th>descriptorType</th>
<th>descriptorCount</th>
<th>pipeline stage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>set = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Array of Descriptor Set Layouts

![Image of array of descriptor set layouts]
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

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

---

Step 4: Allocating the Memory for Descriptor Sets

```c
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] );
}
```

---

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

```c
texture_sampler = MyPuppyTexture.texSampler; 
vdbi0.imageView = MyPuppyTexture.texImageView; 
vdbi0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```
**Step 5: Tell the Descriptor Sets where their CPU Data is**

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

// ds 1:
VkWriteDescriptorSet vwds1;
vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds1.pNext = nullptr;
vwds1.dstSet = DescriptorSets[1];
vwds1.dstBinding = 0;
vwds1.dstArrayElement = 0;
vwds1.descriptorCount = 1;
vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds1.pBufferInfo = &vdbi1;
vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds1.pTexelBufferView = (VkBufferView *)nullptr;
```

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

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

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

```cpp
vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
```
Sidebar: The Entire Descriptor Set Journey

- `VkDescriptorPoolCreateInfo`:
  - `vkCreateDescriptorPool()`:
    - Create the pool of Descriptor Sets for future use.

- `VkDescriptorSetLayoutBinding`:
  - `vkCreateDescriptorSetLayout()`:
    - Describe a particular Descriptor Set layout and use it in a specific Pipeline layout.

- `VkDescriptorSetAllocateInfo`:
  - `vkAllocateDescriptorSets()`:
    - Allocate memory for particular Descriptor Sets.

- `VkDescriptorBufferInfo`, `VkDescriptorImageInfo`:
  - `vkWriteDescriptorSet()`:
    - Re-write CPU data into a particular Descriptor Set.

- `VkDescriptorSetAllocateInfo`:
  - `vkBindDescriptorSets()`:
    - 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.
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. 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_TRIANGLES );
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

```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. }
  },
  ...n
};
```

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,
Or, for a sphere,

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

\[ s = \frac{\text{lng} + M\_\pi}{2 \times M\_\pi} \quad \quad t = \frac{\text{lat} + M\_\pi/2}{M\_\pi} \]

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….

\[ s = ? \quad t = ? \]
Something I’ve Found Useful

I find it handy to encapsulate texture information in a struct, just like I do with buffer information:

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

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

I find it handy to encapsulate texture information in a struct, just like I do with buffer information:

```c
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
```

```c
MyTexture MyPuppyTexture;
VkSamplerCreateInfo vsci;
vsci.magFilter = VK_FILTER_LINEAR;
vsci.minFilter = VK_FILTER_LINEAR;
vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
...
result = vkCreateSampler(LogicalDevice, &vsci, PALLOCATOR, &MyPuppyTexture->texSampler);
```

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-resolution textures of the same texture so that the red texel gets included somehow in all resolution-level textures.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
<td>RGBA</td>
</tr>
</tbody>
</table>

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 Texture:Pixel ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

* Latin: multiplicum in parvo, “many things in a small place”
Init07TextureSampler (MyTexture * pMyTexture)

Init07TextureBuffer (INOUT MyTexture * pMyTexture)

vkCreateSampler (LogicalDevice, IN &vsci, PALLOCATOR, OUT &MyPuppyTexture->texSampler);

vkGetImageSubresourceLayout (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);

vkCreateImage (LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

vkMapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkUnmapMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkBindImageMemory (LogicalDevice, stagingImage, IN &vis, OUT &vsl);

vkAllocateMemory (LogicalDevice, stagingImage, IN &vmai, PALLOCATOR, OUT &stagingImage); // allocated, but not filled
// this second {...} is to create the actual texture image:

VkCommandBufferBeginInfo vcbbi;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

// transition the texture buffer layout:

VkImageMemoryBarrier vimb;
visr.layerCount = 1;
visr.levelCount = 1;
visr.baseMipLevel = 0;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.stageMask = VK_PIPELINE_STAGE_TRANSFER_BIT;
visr.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
visr.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
visr.pNext = nullptr;
visr.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

vkCmdPipelineBarrier(TextureCommandBuffer,
0, (VkBufferMemoryBarrier *)nullptr,
0, (VkMemoryBarrier *)nullptr,
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
vimb.pNext = nullptr;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

vkCmdCopyImage(TextureCommandBuffer,
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);

// because we are transferring into it and will eventual sample from it

// transition the staging buffer layout:

VkCommandBufferBeginInfo vcbbi;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

// copy pixels from the staging image to the texture:

VkImageCopy vic;

vkCmdCopyImage(TextureCommandBuffer,
textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);
Reading in a Texture from a BMP File

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

MyTexture MyPuppyTexture;
```

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

This function can be found in the sample.cpp file. The BMP file needs to be created by something that writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick's convert, Adobe Photoshop, or GNU's GIMP.
What is the Vulkan Graphics Pipeline Data Structure (GPDS)?

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. Since you know the OpenGL state, a lot of the Vulkan GPDS will seem familiar to you.
3. The current shader program is part of the state. (It was in OpenGL too, we just didn’t make a big deal of it.)
4. The Vulkan Graphics Pipeline is not the processes that OpenGL would call “the graphics pipeline”.
5. 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 GPDS.
6. The shaders get compiled the rest of the way when their Graphics Pipeline Data Structure gets created.

The First Step: Create the Graphics Pipeline Layout

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

```c
VkPipelineLayout GraphicsPipelineLayout; // global

VkResult Init14GraphicsPipelineLayout()
{
    VkResult result;
    VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

    return result;
}
```

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.

There are also a Vulkan Compute Pipeline Data Structure and a Raytrace Pipeline Data Structure – we will get to those later.

Vulkan 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.

A Graphics Pipeline Data Structure Contains the Following State Items:

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use (half-compiled SPIR-V modules)
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology (e.g., VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST)
- 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, blendColorBlendOp, 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 be changed with Dynamic State Variables
Creating a Graphics Pipeline from a lot of Pieces

Creating a Typical Graphics Pipeline

The Shaders to Use

Link in the Per-Vertex Attributes
VkPipelineVertexInputStateCreateInfo vpvisci;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.vertexBindingDescriptionCount = 1;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = vviad;

VkPipelineInputAssemblyStateCreateInfo vpiasci;
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
vpiasci.flags = 0;
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
#ifdef CHOICES
VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
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
#endif
vpiasci.primitiveRestartEnable = VK_FALSE;

VkPipelineTessellationStateCreateInfo vptsci;
vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
vptsci.pNext = nullptr;
vptsci.flags = 0;
vptsci.patchControlPoints = 0; // number of patch control points

VkPipelineGeometryStateCreateInfo vpgsci;
vpgsci.sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO;
vpgsci.pNext = nullptr;
vpgsci.flags = 0;

What is “Primitive Restart Enable”? 

vpiasci.primitiveRestartEnable = VK_FALSE;

“Restart Enable” is used with:
- Indexed drawing.
- TRIANGLE_FAN and TRIANGLE_STRIP topologies

If vpiasci.primitiveRestartEnable is VK_TRUE, then a special “index” can be used to indicate that the primitive should start over. This is more efficient than explicitly ending the current triangle strip and explicitly starting a new one.

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, then the special index is 0xffffffff.

That is, a one in all available bits

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

Triangle Strip #0
Triangle Strip #1
Triangle Strip #2
...
Declare the viewport information


Declare the scissoring information


Group the viewport and scissoring information together


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

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

Scissoring:
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.

You Can Think of the Stencil Buffer as a Separate Framebuffer, or, You Can Think of it as being Per-Pixel

Both are correct, but I like thinking of it "per-pixel" better.

Using the Stencil Buffer to Create a Magic Lens


I Once Used the Stencil Buffer to Create a Magic Lens for Volume Data

In this case, the scene inside the lens was created by drawing the same object, but drawing it with its near clipping plane being farther away from the eye position.

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Using the Stencil Buffer to Better Outline Polygons

Stencil Operations for Front and Back Faces

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

VkStencilOpState vsosb;
// back
vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
vsosb.failOp = VK_STENCIL_OP_KEEP;
vsosb.passOp = VK_STENCIL_OP_KEEP;
vsosb.compareOp = VK_COMPARE_OP_NEVER;
vsosb.compareMask = ~0;
vsosb.writeMask = ~0;
vsosb.reference = 0;
```
Operations for Depth Values

VkPipelineDepthStencilStateCreateInfo

- sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
- pNext = nullptr;
- flags = 0;
- depthTestEnable = VK_TRUE;
- depthWriteEnable = VK_TRUE;
- depthCompareOp = VK_COMPARE_OP_LESS;
  - VK_COMPARE_OP_NEVER -- never succeeds
  - VK_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
  - VK_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
  - VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
  - VK_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
  - VK_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
  - VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
  - VK_COMPARE_OP_ALWAYS -- always succeeds
- depthBoundsTestEnable = VK_FALSE;
- front = vsosf;
- back = vsosb;
- minDepthBounds = 0.;
- maxDepthBounds = 1.;
- stencilTestEnable = VK_FALSE;

Putting it all Together! (finally…)

When Drawing, We will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer

Vulkan

Queues and Command Buffers
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, but doesn't have to be
- 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 already has them
- Each Queue belongs to a Queue Family
- We don't create Queue Families – the Physical Device already has them

```c
// Querying what Queue Families are Available
uint32_t count;
if (vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr);

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

For the Nvidia A6000 cards:

```
Found 3 Queue Families:
0: Queue Family Count = 16  ;    Graphics Compute Transfer
1: Queue Family Count =   2  ;    Transfer
2: Queue Family Count =   8  ;    Compute Transfer
```
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(IN 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
VkDeviceCreateInfo vdci;
vdc.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdc.pNext = nullptr;
vdc.flags = 0;
vdc.queueCreateInfoCount = 1; // # of device queues wanted
vdci.pQueueCreateInfos = IN &vdqci[0]; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdc.ppEnabledLayerNames = myDeviceLayers;
vdc.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdc.ppEnabledExtensionNames = myDeviceExtensions;
vdc.pEnabledFeatures = IN &PhysicalDeviceFeatures; // already created
result = vkCreateLogicalDevice(PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice);
```

Creating the command pool as part of the logical device:

```c
VkResult Init06CommandPool()
{
    VkResult result;
    VkCommandPoolCreateInfo vcpci;
    vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcpci.pNext = nullptr;
    vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
    return result;
}
```

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[0]);
    }
    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    { 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 = 1;
      result = vkAllocateCommandBuffers(LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer);
    }
    return result;
}
```
vkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

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

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

VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);

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

vkCmdPreprocessGeneratedCommands
vkCmdPushConstants
vkCmdPushDescriptorSet
vkCmdPushDescriptorSetWithTemplate
vkCmdResetEvent
vkCmdResolveImage
vkCmdResolveImage2
vkCmdSetBlendConstants
vkCmdSetCheckpoint
vkCmdSetCoarseSampleOrder
vkCmdSetCullMode
vkCmdSetDepthBias
vkCmdSetDepthBiasEnable
vkCmdSetDepthBounds
vkCmdSetDepthBoundsTestEnable
vkCmdSetDepthCompareOp
vkCmdSetDepthTestEnable
vkCmdSetDepthWriteEnable
vkCmdSetDeviceMask
vkCmdSetDiscardRectangle
vkCmdSetEvent
vkCmdSetExclusiveScissor
vkCmdSetFragmentShadingRate
vkCmdSetFragmentShadingRate
vkCmdSetFrontFace
vkCmdSetLineStipple
vkCmdSetLineWidth
vkCmdSetLogicOp
vkCmdSetMask
vkCmdSetScissor
vkCmdSetScissorWithCount
vkCmdSetStencilCompareMask
vkCmdSetStencilOp
vkCmdSetStencilReference
vkCmdSetStencilTestEnable
vkCmdSetStencilWriteMask
vkCmdSetVertexInput
vkCmdSetViewport
vkCmdSetViewportShadingRatePalette
vkCmdSetViewportWithCount
vkCmdSetViewportWScaling
vkCmdSubpassShading
vkCmdTraceRaysIndirect2
vkCmdTraceRaysIndirect
vkCmdTraceRays
vkCmdUpdateBuffer
vkCmdWriteAnnotations
vkCmdWriteAccelerationStructuresProperties
vkCmdWriteBufferMarker
vkCmdWriteBufferMarker
vkCmdWriteTimestamp
vkCmdWriteTimestamp2

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

How the RenderScene() Function Works

VkResult RenderScene( )
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX, IN VK_NULL_HANDLE, IN VK_NULL_HANDLE, OUT &nextImageIndex );
    VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
    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;
    VkClearColorValue 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
    0.,                     // y
    (float)Width,           // width
    (float)Height,          // height
    0.,                     // minDepth
    1.                      // maxDepth
};

vkCmdSetViewport(CommandBuffers[nextImageIndex], 0, 1, IN &viewport);

VkRect2D scissor =
{
    0,                     // x
    0,                     // y
    Width,                // width
    Height               // height
};

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

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

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);

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]);
How OpenGL Thinks of Framebuffers

Update

Refresh

Back

Depth

How Vulkan Thinks of Framebuffers – the Swap Chain

Update

Front

Back

Depth

Back

Present

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

VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;
fprintf( FpDebug, "vkGetPhysicalDeviceSurfaceCapabilitiesKHR: 
" );

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

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

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "Found %d Present Modes: 
" );
Here’s What the Vulkan Spec Has to Say About Present Modes, I

VK_PRESENT_MODE_FIFO_KHR specifies that the presentation engine does not wait for a vertical blanking period to update the current image, meaning this mode may result in visible tearing. No internal queuing of presentation requests is needed, so the requests are applied immediately.

VK_PRESENT_MODE_MAILBOX_KHR specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for reuse by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.

VK_PRESENT_MODE_IMMEDIATE_KHR specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty. This is the only value of presentMode that is required to be supported.

VK_PRESENT_MODE_FIFO_RELAXED_KHR specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image, then the presentation engine does not wait for another vertical blanking period for the update, meaning this mode may result in visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.
Creating a Swap Chain

VkSurfaceCapabilitiesKHR vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(PhysicalDevice, Surface, OUT &vsc);
vsc.currentExtent.width = surfaceRes.width;
vsc.currentExtent.height = surfaceRes.height;
vsc.currentFormat = VK_FORMAT_B8G8R8A8_UNORM;
vsc.colorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vsc.imageExtent.width = surfaceRes.width;
vsc.imageExtent.height = surfaceRes.height;
vsc.format = VK_FORMAT_B8G8R8A8_UNORM;
vsc.colorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
result = vkCreateSwapchainKHR(LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain);

Creating the Swap Chain Images and Image Views

VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
vkAcquireNextImageKHR(LogicalDevice, SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

Rendering into the Swap Chain, I

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

Rendering into the Swap Chain, II

VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
result = vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);
VkQueue presentQueue;
vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue);

VkSubmitInfo vsi;
result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence);
result = vkWaitForFences( LogicalDevice, 1, &renderFence, VK_TRUE, UINT64_MAX );

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

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

```
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr);
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices);
```

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

```
int discreteSelect = -1;
int integratedSelect = -1;
for(unsigned int i = 0; i < PhysicalDeviceCount; i++) {
    VkPhysicalDeviceProperties vpdp;
    vkGetPhysicalDeviceProperties(IN physicalDevices[i], OUT &vpdp);
    if(result != VK_SUCCESS) {
        fprintf(FpDebug, "Could not get the physical device properties of device %d", i);
        return VK_SHOULD_EXIT;
    }
    fprintf(FpDebug, "

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

```
Which Physical Device to Use, I
```
Asking About the Physical Device’s Features

```c
VkPhysicalDeviceProperties propDevice; 
vkGetPhysicalDeviceProperties( PhysicalDevice, &propDevice );

propDev->FpDebug( "Physical Device Properties:\n" );
propDev->FpDebug( "geometryShader = %2d\", propDevice->geometryShader );
propDev->FpDebug( "tessellationShader = %2d\", propDevice->tessellationShader );
propDev->FpDebug( "multidrawIndirect = %2d\", propDevice->multidrawIndirect );
```

Asking About the Physical Device’s Different Memories

```c
VkPhysicalDeviceMemoryProperties propMem; 
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, &propMem );

propDev->FpDebug( "Memory Properties:\n" );
propDev->FpDebug( "Memory %\;:\n" );
```

Here’s What the Nvidia A6000 Produced

```c
Init03PhysicalDeviceAndGetQueueFamilyProperties
```

Here’s What the Intel HD Graphics 520 Produced

```c
Init03PhysicalDeviceAndGetQueueFamilyProperties
```
Here's What I Got on the Nvidia A6000

6 Memory Types:
Memory 0: DeviceLocal
Memory 1: HostVisible HostCoherent
Memory 2: HostVisible HostCoherent HostCached
Memory 4: DeviceLocal HostVisible HostCoherent
Memory 5: DeviceLocal

4 Memory Heaps:
Heap 0: size = 0xdbb00000 DeviceLocal
Heap 1: size = 0xfd504000
Heap 2: size = 0x0d600000 DeviceLocal
Heap 3: size = 0x02000000 DeviceLocal

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

Asking About the Physical Device's Queue Families

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

Logical Devices
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;
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);

// see what device extensions are available:
uint32_t  extensionCount;
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr);
VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCount];
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
What Device Layers and Extensions are Available

4 physical device layers enumerated:

- 0x004030cd 1 'VK_LAYER_NV_optimus' "NVIDIA Optimus layer"
- 0x00400033 1 'VK_LAYER_LUNARG_core_validation' "Lunarg Validation Layer"
- 0x00400033 1 'VK_LAYER_LUNARG_object_tracker' "Lunarg Validation Layer"
- 0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' "Lunarg Validation Layer"

160 device extensions enumerated for "VK_LAYER_NV_optimus":

- 0x00400033 1 'VK_LAYER_LUNARG_core_validation' "Lunarg Validation Layer"
- 0x00400033 1 'VK_LAYER_LUNARG_object_tracker' "Lunarg Validation Layer"
- 0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' "Lunarg Validation Layer"

Vulkan: Creating a Logical Device

```c
float queuePriorities[1] = { 1.0f};

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

result = vkCreateDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
```

Vulkan: Creating the Logical Device’s Queue

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

Layers and Extensions

```c
// 13 instance layers enumerated:
```

- 0x00403023 2 'VK_LAYER_LUNARG_platform_info'
- 0x00403023 1 'VK_LAYER_LUNARG_standard_validation'
- 0x00403023 1 'VK_LAYER_LUNARG_secondary_cube'
- 0x00403023 1 'VK_LAYER_LUNARG_golden_pixel'
- 0x00403023 1 'VK_LAYER_LUNARG_sharded_device'
- 0x00403023 1 'VK_LAYER_LUNARG_golden_pixel'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy_pixel'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy Vulkan'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy device'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy Vulkan'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy device'
- 0x00403023 1 'VK_LAYER_LUNARG_dummy Vulkan'

Lunarg validation layer
Lunarg standard validation
Lunarg gold pixel
Lunarg secondary cube
Lunarg platform info
vkEnumerateInstanceLayerProperties:
13 instance layers enumerated:
0x00400033 2 'VK_LAYER_LUNARG_api_dump' 'LunarG debug layer'
0x00400033 1 'VK_LAYER_LUNARG_core_validation' 'LunarG Validation Layer'
0x00400033 1 'VK_LAYER_LUNARG_monitor' 'Execution Monitoring Layer'
0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'
0x00400033 1 'VK_LAYER_LUNARG_screenshot' 'LunarG image capture layer'
0x00400033 1 'VK_LAYER_LUNARG_standard_validation' 'LunarG Standard Validation'
0x00400033 1 'VK_LAYER_GOOGLE_threading' 'Google Validation Layer'
0x00400033 1 'VK_LAYER_GOOGLE_unique_objects' 'Google Validation Layer'
0x00400033 1 'VK_LAYER_LUNARG_vktrace' 'Vktrace tracing library'
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x0040000d 34 'VK_LAYER_RENDERDOC_Capture' 'Debugging capture layer for RenderDoc'

vkEnumerateInstanceExtensionProperties:
11 extensions enumerated:
0x00000008  'VK_KHR(surface)' 'NVIDIA Optimus layer'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_get_surface_capabilities2'
0x00000019  'VK_KHR_surface'
0x00000008  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_validation_layer'
0x00000001  'VK_KHR_external_fence_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'

vkEnumerateDeviceLayerProperties:
3 physical device layers enumerated:
0x00400038 1 'VK_LAYER_NV_optimus' 'NVIDIA Optimus layer'
0x00400033 1 'VK_LAYER_LUNARG_object_tracker' 'LunarG Validation Layer'
0x00400033 1 'VK_LAYER_LUNARG_parameter_validation' 'LunarG Validation Layer'

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

const char * instanceExtensions[] =
{ //VK_KHR_surface'
    "VK_KHR_win32_surface'',
    "VK_KHR_win32_surface''
};
13 instance layers available:
0x00400033   2  'VK_LAYER_LUNARG_api_dump'  'LunarG debug layer'
0x00400033   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
0x00400033   1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
0x00400033   1  'VK_LAYER_LUNARG_monitor'  'Execution Monitoring Layer'
0x00400033   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_screenshot'  'LunarG image capture layer'
0x00400033   1  'VK_LAYER_LUNARG_standard_validation'  'LunarG Standard Validation'
0x00400033   1  'VK_LAYER_GOOGLE_threading'  'Google Validation Layer'
0x00400033   1  'VK_LAYER_GOOGLE_unique_objects'  'Google Validation Layer'
0x00400033   1  'VK_LAYER_LUNARG_vktrace'  'Vktrace tracing library'
0x00400038   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x0040000d   1  'VK_LAYER_NV_nsight'  'NVIDIA Nsight interception layer'
0x00400000  34  'VK_LAYER_RENDERDOC_Capture'  'Debugging capture layer for RenderDoc'

11 instance extensions available:
0x00000008  'VK_EXT_debug_report'
0x00000001  'VK_EXT_display_surface_counter'
0x00000001  'VK_KHR_get_physical_device_properties2'
0x00000001  'VK_KHR_get_surface_capabilities2'
0x00000019  'VK_KHR_surface'
0x00000006  'VK_KHR_win32_surface'
0x00000001  'VK_KHR_device_group_creation'
0x00000001  'VK_KHR_external_fence_capabilities'
0x00000001  'VK_KHR_external_semaphore_capabilities'
0x00000001  'VK_KHR_external_memory_capabilities'

Will now ask for 3 instance extensions
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_report
result = vkEnumeratePhysicalDevices(Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr);

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

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

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

vkGetPhysicalDeviceProperties(PhysicalDevice, OUT &PhysicalDeviceProperties);
vkGetPhysicalDeviceFeatures(IN PhysicalDevice, OUT &PhysicalDeviceFeatures);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp);
vkGetPhysicalDeviceFormatProperties(PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp);
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(PhysicalDevice, OUT &vpdmp);

uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, deviceLayers);
for (unsigned int i = 0; i < layerCount; i++)
{
    // see what device extensions are available:
    uint32_t extensionCount;
vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, (VkExtensionProperties *)nullptr);
result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName, &extensionCount, deviceExtensions);
}
delete[] deviceLayers;

VkResult result;
float queuePriorities[NUM_QUEUES_WANTED] =
{
    1.
};
VkDeviceQueueCreateInfo vdqci[NUM_QUEUES_WANTED];
vdqci[0].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1; // how many queues to create
vdqci[0].pQueuePriorities = queuePriorities; // array of queue priorities [0.,1.]
const char * myDeviceLayers[] =
{
    //VK_LAYER_LUNARG_api_dump,
    //VK_LAYER_LUNARG_core_validation,
    //VK_LAYER_LUNARG_image,
    "VK_LAYER_LUNARG_object_tracker",
    "VK_LAYER_LUNARG_parameter_validation",
    "VK_LAYER_LUNARG_m3d_tracker",
};
const char * myDeviceExtensions[] =
{
    "VK_KHR_swapchain",
};
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, such as mat4 transformation matrices. This is a good feature, since Vulkan, at times, makes it cumbersome to send changes to the graphics.

By “small”, Vulkan specifies that there will be at least 128 bytes that can be used, 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 do not live in their own GPU memory. They are actually included inside the Vulkan graphics pipeline data structure.

---

<table>
<thead>
<tr>
<th>4 physical device layers enumerated:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkEnumerateDeviceExtensionProperties: Successful</td>
</tr>
<tr>
<td>0 device extensions enumerated for 'VK_LAYER_NV_optimus':</td>
</tr>
<tr>
<td>vkEnumerateDeviceExtensionProperties: Successful</td>
</tr>
<tr>
<td>0 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':</td>
</tr>
<tr>
<td>vkEnumerateDeviceExtensionProperties: Successful</td>
</tr>
<tr>
<td>0 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':</td>
</tr>
<tr>
<td>vkEnumerateDeviceExtensionProperties: Successful</td>
</tr>
<tr>
<td>0 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':</td>
</tr>
</tbody>
</table>
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;
} Model;
```

On the application side, push constants are pushed at the shaders by giving 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_TESSELLATION_CONTROL_SHADER_BIT`
  - `VK_PIPELINE_STAGE_TESSELATION_EVALUATION_SHADER_BIT`
  - `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
  - `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `size` is in bytes
- `pValues` is a `void` * pointer to the data, which, in this 4x4 matrix example, would be of type `glm::mat4`.

A Robotic Example using Push Constants

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

Where each arm is represented by:

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

Setting up the Push Constants for the Graphics Pipeline Data Structure

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

```c
VkPushConstantRange
vpcr[1];

vpcr[0].stageFlags =
  VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
  VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

vpcr[0].size = sizeof(glm::mat4);
```

```c
VkPipelineLayoutCreateInfo
vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

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

Forward Kinematics:

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

```cpp
struct arm      Arm1;
struct arm      Arm2;
struct arm      Arm3;
...
Arm1.armMatrix = glm::mat4(1.1); // identity
Arm1.armColor  = glm::vec3(0.f, 1.f, 0.f); // green
Arm1.armScale  = 6.f;
Arm2.armMatrix = glm::mat4(1.1); // identity
Arm2.armColor  = glm::vec3(1.f, 0.f, 0.f); // red
Arm2.armScale  = 4.f;
Arm3.armMatrix = glm::mat4(1.1); // identity
Arm3.armColor  = glm::vec3(0.f, 0.f, 1.f); // blue
Arm3.armScale  = 2.f;
```

The constructor `glm::mat4(1.1)` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`.

Set the Push Constant for the Graphics Pipeline Data Structure

```cpp
VkPushConstantRange vpcr[1];

vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

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

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

In the `UpdateScene()` Function

```cpp
float rot1 = (float)(2.*M_PI*Time); // rotation for arm1, in radians
float rot2 = 2.f * rot1; // rotation for arm2, in radians
float rot3 = 2.f * rot2; // rotation for arm3, in radians

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

m1g = glm::mat4(1.); // identity
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));

m1g = glm::rotate(m1g, rot1, zaxis); // [T][R]

m21 = glm::mat4(1.); // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));

m21 = glm::rotate(m21, rot2, zaxis); // [T][R]

m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm

m32 = glm::mat4(1.); // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));

m32 = glm::rotate(m32, rot3, zaxis); // [T][R]

m32 = glm::translate(m32, glm::vec3(0., 0., 2.)); // z-offset from previous arm

Arm1.armMatrix = m1g; // m1g

Arm2.armMatrix = m1g * m21; // m2g

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

In the `RenderScene()` Function

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

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);

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

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1);

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

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2);

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

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3);

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

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

```glsl
layout( push_constant ) uniform arm
{
    mat4 armMatrix;
    vec3 armColor;
    float armScale; // scale factor in x
} RobotArm;
layout( location = 0 ) in vec3 aVertex;

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

// Projection * Viewing * Modeling matrices

gl_Position = PVMM * vec4( bVertex, 1. );
```

Synchronization

Remember the Overall Block Diagram?

Where Synchronization Fits in the Overall Block Diagram
• Indicates that a batch of commend has been processed from a queue. Basically
  announces “I am finished”!
• You create one and give it to a Vulkan function which sets it. Later on, you tell another
  Vulkan function to wait for this semaphore to be signaled.
• You don’t end up setting, resetting, or checking the semaphore yourself.
• Semaphores must be initialized (“created”) before they can be used.

Semaphores

Ask for Something → Try to Use that Something

Semaphore

Creating a Semaphore

VkSemaphoreCreateInfo
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 );

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

Fcences

• Used to synchronize CPU-GPU tasks.
• Used when the host needs to wait for the device to complete something big.
• Announces that queue-submitted work is finished.
• You can un-signal, signal, test or block-while-waiting.
Events

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

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, &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 = 0xffffffffffffffff (= 580+ years)
// result = VK_SUCCESS means it returned because a fence (or all fences) signaled
// result = VK_TIMEOUT means it returned because the timeout was exceeded
```

Fence Example

```c
// Fence Example

VkFence renderFence;
vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkQueue presentQueue;
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);

result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);

result = vkQueuePresentKHR(presentQueue, IN &vp); // don't present the image until done rendering
```

Events

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

Controlling Events from the Host

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

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

// result = VK_EVENT_SET: signaled
// result = VK_EVENT_RESET: not signaled

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

result = vkCmdSetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdResetEvent( CommandBuffer, IN event, pipelineStageBits );
result = vkCmdWaitEvents( CommandBuffer, 1, &event, srcPipelineStageBits, dstPipelineStageBits, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );

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

Pipeline Barriers

A series of vkCmdxxx( ) calls are meant to run “flat-out”, that is, as fast as the Vulkan runtime can get them executing. But, many times, that is not desirable because the output of one command might be needed as the input to a subsequent command.

Pipeline Barriers solve this problem by declaring which stages of the hardware pipeline in subsequent vkCmdyyy( ) calls need to wait until which stages in previous vkCmdxxx( ) calls are completed.

Why Do We Need Pipeline Barriers?

Potential Memory Race Conditions that Pipeline Barriers can Prevent

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

Note: there is no problem with Read-after-Read (R-a-R) as no data gets changed.
A Pipeline Barrier is a way to establish a dependency between commands that were submitted before the barrier and commands that are submitted after the barrier.

The hope is to maximize the number of unblocked stages: produce data early and consume data late.

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 lights will be turned red
5. When the last car in the src group completely makes it through its intersection, the proper dst traffic lights are turned back to green
6. The Vulkan command pipeline ordering is this: (1) the src cars get released by the previous vkCmdxxx, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released by the next vkCmdyyy, (4) the dst cars stop at the red light, (5) the src cars clear the intersection, (6) the dst lights turn 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
Pipeline Stages

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_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

- 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
Example: Be sure we are done writing an Output image before using it as a Fragment Shader Texture

Example: The Scenario

src cars are generating the image
dst cars are waiting to use that image as a texture

Antialiasing and Multisampling

The Display We Want
Too often, the Display We Get
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. **Results in the best image, but the most rendering time.**

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. **Results in a good image, with less rendering time.**

The final step is to average those sub-pixels’ colors to produce one final color for this whole pixel. This is called **resolving** the pixel.

### Consider Two Triangles That 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.

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.125, 0.125)</td>
<td>(0.3125, 0.3125)</td>
<td>(0.625, 0.625)</td>
<td>(0.9375, 0.9375)</td>
</tr>
<tr>
<td>(0.25, 0.25)</td>
<td>(0.625, 0.625)</td>
<td>(0.9375, 0.9375)</td>
<td></td>
</tr>
<tr>
<td>(0.375, 0.375)</td>
<td>(0.8125, 0.8125)</td>
<td>(0.9375, 0.9375)</td>
<td></td>
</tr>
<tr>
<td>(0.5, 0.5)</td>
<td>(0.8125, 0.8125)</td>
<td>(0.9375, 0.9375)</td>
<td></td>
</tr>
</tbody>
</table>

Vulkan Specification Distribution of Sampling Points within a 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.

### 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.
Consider Two Triangles Who Pass Through the Same Pixel

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?

<table>
<thead>
<tr>
<th>Number of Fragment Shader Calls</th>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Setting up the Image

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

VkGraphicsPipelineCreateInfo
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;

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

Setting up the Image

VkAttachmentDescription
vad[0].format = VK_FORMAT_B8G8R8A8_SRGB; // 24-bit color
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_COLOR_ATTACHMENT_OPTIMAL;

vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT; // 32-bit floating-point depth
vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference
vad[0].attachment = 0;
vad[0].layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

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

0. produces simple multisampling
(0. - 1.) produces partial supersampling
1. Produces complete supersampling
Setting up the Image

```cpp
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;
```

```cpp
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;
```

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

Resolving the Image:

```cpp
VlOffset3D vo3;
vo3.x = 0;
vo3.y = 0;
vo3.z = 0;
```

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

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

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

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

Resolving the Image: Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

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

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

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

```cpp
VkImageResolve vir;
vir.srcSubresource = visl;
vir.srcOffset = vo3;
```
Vulkan Program Flow – the Rendering Loop

while( the GLFW Window should not close ) {
    UpdateScene();
    RenderScene();
}

Create the Transformations
Fill the Uniform Buffers

Acquire the Next Swap Chain Image

Begin its Command Buffer

Create the RenderPass with the Framebuffer information

for( all the different Graphics Pipeline Data Structures being used )
    
    Bind that Graphics Pipeline Data Structure
    
    Set any Dynamic State Variables
    
    Bind the Proper Descriptor Set Values
    
    Do the Drawing

End the RenderPass
End the Command Buffer
Submit the Command Buffer to a Queue
Wait for the Queue to Finish Submitting
Present the Image to the Viewer

So What Do We All Do Now?

• Performance-critical
  • Performance-uncritical
  • Need ray-tracing

The Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
SIGGRAPH '23 Courses, August 06-10, 2023, Los Angeles, CA, USA
ACM 978-8-4007-0145-0/23/08.
10.1145/3587423.3595529
http://cs.oregonstate.edu/~mj/vulkan