Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Computer Graphics API

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

Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

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

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

Who is the Khronos Group?

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

Who’s Been Specifically Working on Vulkan?

Vulkan Differences from OpenGL

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

The Basic Computer Graphics Pipeline, Shader-style

Vulkan Shaders

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

Vulkan Shaders

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

Vulkan Shaders

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

Vulkan Shaders

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

Traditional
Application

Application

Application

Application

Application

Application

Application

Application

Application

Complex drivers lead to
driver overhead and
unpredictability
Error management is
design difficult
Driver processes full
shading language source
Separate APIs for
desktop and mobile
markets

Simpler drivers for
low-overhead efficiency
and cross vendor
portability
Moving part of the driver into the application

Moving part of the driver into the application

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

Vulkan Highlights: Command Buffers

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

Vulkan Highlights: Pipelines

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

Vulkan Quick Reference Card – I Recommend you Get This!

Steps in Creating Graphics using Vulkan

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

Caveats on the Sample Code, I

1. I've written everything out in appalling longhand.
2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything easily.
3. At times, I could have hidden complexity, but I didn't. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that's kept a secret from you.
4. I've setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
5. At times, I've setup things that didn't need to be set up for you what could go there.
6. There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'd a couple things to make the Vulkan phrasing more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Caveats on the Sample Code, II

Sample Program Keyboard Inputs

'Y', 'y': Toggle using a vertex buffer only vs. an index buffer
'Y', 'L': Toggle lighting off and on
'I', 'i': Toggle display mode (textures vs. colors, for now)
'I', 'P': Pause the animation
'I', 'Q': Quit the program
Esc: Quit the program
'R', 'r': Toggle rotation animation and using the mouse
1, 4, 9: Number of instances
Main Program

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

InitGraphics( ), I

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

A Colored Cube

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

```c
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. },
  // vertex #4:
  { -1., -1.,  1. },
  {  0.,  0.,  1. },
  {  0.,  0.,  1. },
  {  1., 0. },
  // vertex #5:
  { -1.,  1.,  1. },
  {  0.,  0.,  1. },
  {  0.,  1.,  0. },
  {  1., 1. },
  // vertex #6:
  {  1., -1., -1. },
  {  0.,  0., -1. },
  {  1., -1.,  0. },
  {  0., 1. },
  // vertex #7:
  { -1., -1.,  1. },
  {  0.,  0.,  1. },
  {  0.,  0.,  1. },
  {  1., 0. },
};
```

The Vertex Data is in a Separate File

```c
#include "SampleVertexData.cpp"
```
What if you don’t need all of this information?

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

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

Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums
2. Vulkan takes a non-C++ object-oriented approach in that those typedef’d structs pass all the necessary information into a function. For example, where we might normally say in C++:
   ```cpp
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex, OUT &Queue );
   ```
   we would actually say in C:
   ```c
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```

Vulkan Code has a Distinct “Style” of Setting Information in structs

and then Passing that Information as a pointer-to-the-struct

Vulkan Conventions

My Conventions

- **Vk**Xxx is a typedef, probably a struct
- **vk**Xxx( ) is a function call
- **VK**_XXX is a constant
- “Init” in a function call name means that something is being setup that only needs to be setup once
- The number after “Init” gives you the ordering
- In the source code, after main( ) comes InitGraphics( ), then all of the InitxxYYY( ) functions in numerical order. After that comes the helper functions
- “Find” in a function call name means that something is being looked for
- “Fill” in a function call name means that some data is being supplied to Vulkan
- “IN” and “OUT” ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are each actually #define’d to nothing.

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

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

WHERE TO PUT THEM
HOW MANY TOTAL THERE ARE

This way of querying information is a recurring OpenCL and Vulkan pattern (get used to it):
```c
#define REPORT(s)               { PrintVkError( result, s );  fflush(FpDebug); }
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****
```

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

**Geometry vs. Topology**

Where things are (e.g., coordinates)

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

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

**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

**Vertex Orientation Issues**

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system:

- X
- Y
- Z

- Clockwise (CCW)

Internally, however, the Vulkan pipeline uses a left-handed system:

- Z
- Y
- X

- Counterclockwise (CW)

This best way to handle this is to continue to draw in a left-handed coordinate system and then flip it in the projection matrix, like this:

- ProjectionMatrix = \[ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

This is like saying "Y' = -Y".
Vertex Orientation Issues

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

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

Vulkan's change in coordinate systems can mess up the backface culling. So I recommend, at least at first, that you do not culling.

```c
from the file
Init05DataBuffer
VkResult
{
{
struct vertex VertexData[] =
};
{
struct vertex
result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory(LogicalDevice, pMyBuffer->buffer, vdm);
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
vkGetBufferMemoryRequirements(LogicalDevice, IN pMyBuffer->buffer, OUT &vmr); // fills vmr
VkDeviceMemory vdm;
VkMemoryAllocateInfo vmai;
vkGetBufferMemoryRequirements(LogicalDevice, IN pMyBuffer->buffer, OUT &vmr); // fills vmr
VkMemoryRequirements vmr;
return result;
}
{
// vertex #3:
}
{
// vertex #2:
}
{
// vertex #0:
// triangle 0-2-3:
}

SampleVertexData.cpp

C/C++:

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

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

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

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );
const uint32_t vertexCount = sizeof( VertexData ) / sizeof( VertexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

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

Triangles
Draw

Drawing with an Indexed Buffer

VkResult Init05MyVertexDataBuffer( sizeof(JustVertexData), IN &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer,               (void *) JustVertexData );

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 );

Drawing with an Indexed Buffer

Triangles
Draw

Drawing with an Indexed Buffer

Drawing with an Indexed Buffer
Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

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

Thus, when using index-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.

---

**Vulkan: Creating a Data Buffer**

```c
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>
vbci.usage = <<or'ed bits of: >>
  VK_USAGE_TRANSFER_SRC_BIT
  VK_USAGE_TRANSFER_DST_BIT
  VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
  VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
  VK_USAGE_UNIFORM_BUFFER_BIT
  VK_USAGE_STORAGE_BUFFER_BIT
  VK_USAGE_INDEX_BUFFER_BIT
  VK_USAGE_VERTEX_BUFFER_BIT
  VK_USAGE_INDIRECT_BUFFER_BIT
vbci.sharingMode = << one of: >>
  VK_SHARING_MODE_EXCLUSIVE
  VK_SHARING_MODE_CONCURRENT
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

---

**Vulkan: Allocating Memory for a 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;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();

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

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

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

Finding the Right Type of Memory

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;

Initialization of a Data Buffer

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

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

Here's a C struct to hold some uniform variables

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

Here's the shader code to access those uniform variables

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

```c
uint32_t Height, Width;
const double FOV = glm::radians(60.); // field-of-view angle
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
Matrices.uModelMatrix = glm::mat4( ); // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

The Parade of Data

```c
uint32_t Height, Width;
const double FOV = glm::radians(60.); // field-of-view angle
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
Matrices.uModelMatrix = glm::mat4( ); // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

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

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

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

```c
MyBuffer MyMatrixUniformBuffer;
uniform matBuf Matrices;
```

The Descriptor Set for the Buffer

```c
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN 0, (VkCopyDescriptorSet *)nullptr );
```

We will come to Descriptor Sets later, but for now think of them as the link between the BLOB of uniform variables in GPU memory and the block of variable names in your shader programs.

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

Creating and Filling the Data Buffer – the Details

```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 );         // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );             // 0 is the offset
    return result;
}
```

Creating and Filling the Data Buffer – the Details

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

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

---

**Vulkan: GLSL Differences from OpenGL, I**

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:

- In the compiler there is an automatic pre-define VULKAN 100

Vulkan Vertex and Instance indices:

- gl_VertexIndex
- gl_InstanceIndex

- Both are 0-based

**gl_FragColor:**

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

---

**Vulkan: GLSL Differences from OpenGL, II**

Shader combinations of separate texture data and samplers:

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

Descriptor Sets:

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

Specialization Constants:

```
layout( constant_id = 3 )  const int N = 5;
```

Specialization Constants for Compute Shaders:

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

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

---

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

```
// non-sampler variables must be in a uniform block
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
mat4 uModelMatrix;
mat4 uViewMatrix;
mat4 uProjectionMatrix;
mat3 uNormalMatrix;
}
```

All non-sampler uniform variables must be in block buffers

---

**Vulkan Shader Compiling**

- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V, which stands for Standard Portable Intermediate Representation.
- SPIR-V gets turned into fully-compiled code at runtime
- SPIR-V spec has been public for a couple of years – new shader languages are surely being developed
- OpenGL and OpenGL have adopted SPIR-V as well

---

**Advantages:**

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

Standard Portable Intermediate Representation for Vulkan

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

Shaderfile extensions:
.vert Vertex
.tesl Tessellation Control
.tese Tessellation Evaluation
.geom Geometry
.frag Fragment
.comp Compute

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

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

Running glslangValidator.exe

How do you know if SPIR-V compiled successfully?

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

How do you know if SPIR-V compiled successfully?

#define SPIRV_MAGIC 0x07230203

VkResult Init12SpirvShader (std::string filename, VkShaderModule * pShaderModule)
{
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb");
    if( fp == NULL )
    {
        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 );
    Reading a SPIR-V File into a Vulkan Shader Module

VkGraphicsPipelineCreateInfo

Shader stages
- VertexInput State
- InputAssembly State
- Tessellation State
- Viewport State
- Rasterization State
- MultiSample State
- DepthStencil State
- ColorBlend State
- Dynamic State
- Pipeline layout

Pipeline basePipelineHandle
basePipelineIndex

VkPipelineShaderStageCreateInfo

VkVertexInputStateCreateInfo

VkViewportStateCreateInfo

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

VkPipelineRasterizationStateCreateInfo

cullMode
polygonMode
frontFace
lineWidth

VkPipelineDepthStencilStateCreateInfo

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

VkPipelineColorBlendAttachmentState

VkPipelineDynamicStateCreateInfo
Setting Up GLFW

```c
void InitGLFW()
{
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);  // don't want a context
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);        // fixed size window
    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);
}
```

GLFW Keyboard Callback

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

GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

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

```cpp
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
    glfwPollEvents( );
    Time = glfwGetTime( );          // elapsed time, in double-precision seconds
    UpdateScene( );
    RenderScene( );
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
}
```

What is GLM?

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

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

You can find it at:

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

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.

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, you would now have to say:

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

Exactly the same concept, but a different expression of it. Read on for details …

The Most Useful GLM Variables, Operations, and Functions

A constructor:

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

A multiplication:

- glm::mat4 glm::mat4 * glm::mat4; // promote a vec3 to a vec4 via a constructor

A viewing volume (assign, not concatenate):

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

A viewing (assign, not concatenate):

- glm::mat4 glm::lookAt(glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up);
Installing GLM into your own space

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

Here's what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

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

GLM in the Vulkan sample.cpp Program

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

glm::vec3( eye(0., -0.5f, EYEDIST);
glm::vec3( look(0., -0.5f, 0.);
glm::vec3( up(0., 1., 0.);
Matrices.uViewMatrix = glm::lookAt( eye, look, up);
Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uProjectionMatrix[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix );
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
Fill05DataBuffer( MyMatrixUniformBuffer, (void*) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Fill05DataBuffer( MyMiscUniformBuffer, (void*) &Misc );

Your Sample2019.zip File Contains GLM Already
Why Isn't The Normal Matrix just the Same as the Model Matrix?

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

Wrong!

Right!

Wrong!

Right!

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

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

Original object and normal

Wrong!

Right!

Wrong!

Right!

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

These diagrams show you why.

Instancing

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

Instancing – What and why?

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

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

Making each instance look differently – Approach #1

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

Making each instance look differently – Approach #2

Put the unique characteristics in a uniform buffer and reference them.
VkVertexInputBindingDescription vvibd[1]; // an array containing one of these per buffer being used
vvibd[0].binding = 0;           // which binding # this is
vvibd[0].stride = sizeof( struct vertex );              // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

This definition says that we should advance through the input buffer by this much every time we hit a new vertex.

How We Constructed the Graphics Pipeline Structure Before

```cpp
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
.
.
layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
layout( location = 4 ) in vec3 aInstanceColor;
.
layout ( location = 0 ) out vec3 vNormal;
layout ( location = 1 ) out vec3 vColor;
layout ( location = 2 ) out vec2 vTexCoord;

void main( )
{
    mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
    vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );
    //vColor = aColor;
    vColor = aInstanceColor;
    vTexCoord = aTexCoord;
    gl_Position = PVM * vec4( aVertex, 1. );
}
```

How We Write the Vertex Shader Now

In OpenGL

OpenGL puts all uniform data in the same "set", but with different binding numbers, so you can get at each one. Each uniform variable gets updated one-at-a-time. Wouldn't it be nice if we could update a collection of related uniform variables all at once, without having to update the uniform variables that are not related to this collection?

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

As you would have done it in OpenGL

Descriptor Sets

Our example will assume the following shader uniform variables:

```cpp
// non-opaque must be in a uniform block:
layout( std140,
set = 0, binding = 0 ) uniform matBuf {
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;
layout( std140,
set = 1, binding = 0 ) uniform lightBuf {
    vec4 uLightPos;
} Light;
layout( std140,
set = 2, binding = 0 ) uniform miscBuf {
    float uTime;
    int uMode;
} Misc;
layout(              binding = 0 ) uniform sampler2D uSampler;
```
### Step 1: Descriptor Set Pools

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

### Step 2: Define the Descriptor Set Layouts

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

---

**CPU:**
- Uniform data created in a C++ data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts
  - Knows the CPU data structure

**GPU:**
- Uniform data created in a “blob”*
  - Knows the shader data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts

---

### Uniform data in a “blob”*

Uniform data created in a “blob”* – instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

---

### Uniform data created in a C++ data structure

Uniform data created in a C++ data structure

---

### C++ data structure

C++ data structure

---

### Uniform data used in the shader

Uniform data used in the shader

---

### Uniform data in a “blob”*

Uniform data in a “blob”* – instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

---

### Uniform data created in a C++ data structure

Uniform data created in a C++ data structure

---

### Uniform data used in the shader

Uniform data used in the shader

---

**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 know how to access them.

---

**CPU:**
- Uniform data created in a C++ data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts
  - Knows the CPU data structure

**GPU:**
- Uniform data created in a “blob”*
  - Knows the shader data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts

---

### Uniform data in a “blob”*

Uniform data created in a “blob”* – instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

---

### Uniform data created in a C++ data structure

Uniform data created in a C++ data structure

---

### Uniform data used in the shader

Uniform data used in the shader

---

**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 know how to access them.

---

**CPU:**
- Uniform data created in a C++ data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts
  - Knows the CPU data structure

**GPU:**
- Uniform data created in a “blob”*
  - Knows the shader data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts

---

### Uniform data in a “blob”*

Uniform data created in a “blob”* – instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

---

### Uniform data created in a C++ data structure

Uniform data created in a C++ data structure

---

### Uniform data used in the shader

Uniform data used in the shader

---

**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 know how to access them.

---

**CPU:**
- Uniform data created in a C++ data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts
  - Knows the CPU data structure

**GPU:**
- Uniform data created in a “blob”*
  - Knows the shader data structure
  - Doesn’t know the CPU or GPU data structure
  - Knows the data’s size
  - Knows where the data starts

---

### Uniform data in a “blob”*

Uniform data created in a “blob”* – instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.

---

### Uniform data created in a C++ data structure

Uniform data created in a C++ data structure

---

### Uniform data used in the shader

Uniform data used in the shader

---

**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 know how to access them.
Step 3: Include the Descriptor Set Layouts in a Graphics Pipeline Layout

```c
Step 4: Allocating the GPU Memory for Descriptor Sets

```
Step 5b: Upload CPU Data into the GPU Descriptor Set

uint32_t copyCount = N;
if (this could also have been done with one call and an array of VkWriteDescriptorSets):
  vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr);
  vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr);
  vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr);
  vkUpdateDescriptorSets(LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr);

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

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

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

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

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

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. }
  }
};
• 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 TP ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

"Latino muddles it, Venus, many things in a small place!"
create an image view for the texture image:

```cpp
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

VkImageViewCreateInfo vivci;

vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;

vivci.pNext = nullptr;

vivci.flags = 0;

vivci.image = textureImage;

vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;

vivci.format = VK_FORMAT_R8G8B8A8_UNORM;

vivci.components.r = VK_COMPONENT_SWIZZLE_R;

vivci.components.g = VK_COMPONENT_SWIZZLE_G;

vivci.components.b = VK_COMPONENT_SWIZZLE_B;

vivci.components.a = VK_COMPONENT_SWIZZLE_A;

vivci.subresourceRange = visr;

result = vkCreateImageView(
LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
```

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 BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick's `convert`, Adobe Photoshop, or GNU's GIMP.

Reading in a Texture from a BMP File

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

This function can be found in the `sample.cpp` file. The BMP file needs to be created by something that writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick's `convert`, Adobe Photoshop, or GNU's GIMP.

The Graphics Pipeline Data Structure

Mike Bailey
mjb@cs.oregonstate.edu

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

The First Step: Create the Graphics Pipeline Layout

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

```cpp
VkResult Init14GraphicsPipelineLayout( ) {
VkResult result;
VkPipelineLayoutCreateInfo vplci;

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;
}
```

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

The Graphics Pipeline Records the Following Items:

- Pipeline Layout: DescriptorSets, PushConstants
- Which Shaders are going to be used
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissoring: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

```cpp
VkResult Init14GraphicsPipelineLayout( ) {
VkResult result;
VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = &DescriptorSetLayouts[0];

vplci.pushConstantRangeCount = 0;

vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

return result;
}
```

• Pipeline Layout: DescriptorSets, PushConstants
• Which Shaders are going to be used
• Per-vertex input attributes: location, binding, format, offset
• Per-vertex input bindings: binding, stride, inputRate
• Assembly: topology
• Viewport: x, y, w, h, minDepth, maxDepth
• Scissoring: x, y, w, h
• Rasterization: cullMode, polygonMode, frontFace, lineWidth
• Depth: depthTestEnable, depthWriteEnable, depthCompareOp
• Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
• Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
• DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

This state item can also be set with Dynamic Variables
Creating a Graphics Pipeline from a lot of Pieces

Creating a Typical Graphics Pipeline

What is "Primitive Restart Enable"?

Restart Enable is used with:

- Indexed drawing:
- Triangle Fan and "Strip" topologies.

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

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 0xFFFF.
If your VkIndexType is VK_INDEX_TYPE_UINT32, it is 0xFFFFFFFF.
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

Triangle Strip #0:

Triangle Strip #1:

Triangle Strip #2:

....

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

Original Image

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

Setting the Rasterizer State

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

What is “Depth Clamp Enable”?

What is “Depth Bias Enable”?

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

Z-fighting
Color Blending State for each Color Attachment

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

Color Blending State for each Color Attachment

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

Which Pipeline Variables can be Set Dynamically

Which Pipeline Variables can be Set Dynamically

Which Pipeline Variables can be Set Dynamically

Operations for Depth Values

Uses for Stencil Operations

Polygon edges without Z-fighting

Magic Lenses
Group all of the individual state information and create the pipeline:

```
VkGraphicsPipelineCreateInfo vgpci;

vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;

// Define stages if needed
#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 = &vdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;

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

return result;
```

Putting it all Together! (finally…)

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

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

Vulkan Queues and Command Buffers

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

```
Application

Instance

Physical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

CPU Thread

CPU Thread

CPU Thread

CPU Thread

Cmdbuffer

Cmdbuffer

Cmdbuffer

Cmdbuffer

queue

queue

queue

queue

```

Querying what Queue Families are Available

```
uint32_t  count;

vkGetPhysicalDeviceQueueFamilyProperties
        PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr);

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];

vkGetPhysicalDeviceQueueFamilyProperties
        PhysicalDevice, &count, OUT &vqfp,);

for( unsigned int i = 0; i < count; i++ )
{

fprintf( FpDebug, "	%d: Queue Family Count = %2d  ;   ", i, vqfp[i].queueCount );

if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
    fprintf( FpDebug, " Graphics" );

if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT  ) != 0 )
    fprintf( FpDebug, " Compute ");

if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )
    fprintf( FpDebug, " Transfer" );

fprintf(FpDebug, "\n");

Found 3 Queue Families:

0: Queue Family Count = 16  ;    Graphics Compute Transfer
1: Queue Family Count =   1  ;    Transfer
2: Queue Family Count =   8  ;    Compute
```

Simplified Block Diagram

```
```

Vulkan Queues and Command Buffers

http://cs.oregonstate.edu/~mjb/vulkan
Similarly, we can write a function that finds the proper queue family:

```c
int FindQueueFamilyThatDoesGraphics(
    VkPhysicalDevice logicalDevice)
{
    VkQueueFamilyProperties* vqfp = new VkQueueFamilyProperties[count];
    vkGetPhysicalDeviceQueueFamilyProperties(logicalDevice, &count, vqfp);

    for (unsigned int i = 0; i < count; i++)
    {
        if ((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
        {
            result = vkCreateSemaphore(logicalDevice, &vsci, pAllocator, &imageReadySemaphore);
            vkCmdBeginRenderPass(commandBuffer, contents);
            vkCmdBeginQuery(commandBuffer, flags);
        }
    }

    vkCmdEndRenderPass(commandBuffer);
    vkCmdEndQuery(commandBuffer, query);
    vkCmdDrawIndexedIndirect(logicalDevice, indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);
}
```

Creating a Logical Device Needs to Know Queue Family Information:

```c
VkDeviceCreateInfo vdci;
vdci.pEnabledFeatures = PhysicalDeviceFeatures; // already created
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char*);
vdci.ppEnabledLayerNames = myDeviceLayers;
vdci.pNext = nullptr;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

Creating the Command Pool as part of the Logical Device:

```c
VkCommandPoolCreateInfo vc pci;
vc pci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
```

Creating the Command Buffers:

```c
vkAllocateCommandBuffers(logicalDevice, &vcbai, &TextureCommandBuffer);
```

Beginning a Command Buffer:

```c
vkCmdBeginRenderPass(commandBuffer, contents);
```

These are the Commands that could be entered into the Command Buffer, I:
These are the Commands that could be entered into the Command Buffer, in any order:

```c
vkCmdBeginRenderPass
VkRenderPassBeginInfo vrpbi;

vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier*

vkCmdNextSubpass( commandBuffer, contents );

vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );

vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );

vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );

vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );

vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );

vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );

vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );

vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );

vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );

vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );

vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );

vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );

vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );

vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );

vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );

vkCmdResetEvent( commandBuffer, event, stageMask );

vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```

These are the Constraints that can be expressed via semaphores and fences.

When one command depends on a previous command reaching a certain state first.

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there.

But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

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

Mike Bailey
mjb@cs.oregonstate.edu

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

How We Think of OpenGL Framebuffers

Video Driver
Front
Back
Double-buffered
Color Framebuffers
Update
Refresh

Vulkan Thinks of it This Way

Depth-Buffer
Update
Swap Chain
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 really arranged as a ring buffer.

Swap Chains are tightly coupled to the window system.

After creating the Swap Chain in the first place, the process for using the Swap Chain is:
1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to "front buffer")

What is a Swap Chain?

Because it has the word "chain" in it, let’s try to visualize the Swap Chain as a physical chain.

A bicycle chain isn’t far off. A bicycle chain goes around and around, each section of the chain taking its turn on the gear teeth, off the gear teeth, on, off, on, off, etc.

Because the Swap Chain is actually a ring buffer, the images in a Swap Chain go around and around too, each image taking its turn being drawn into, being presented, drawn into, being presented etc.

In the same way that bicycle chain links are “re-used”, Swap Chain images get re-used too.
We Need to Find Out What our Display Capabilities Are

```cpp
result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGraphics(), Surface, VkBool32 supported; 

fprintf( FpDebug, "Found %d Present Modes:
", presentModeCount ) ; 
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];

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

uint32_t formatCount;
uint32_t presentModeCount;
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];

vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkSurfaceCapabilitiesKHR vsc;

for( unsigned int i = 0; i < presentModeCount; i++, presentModeCount )

PresentImageViews = new VkImageView[ imageCount ]; 
PresentImages = new VkImage[ imageCount ];

VkImageViewCreateInfo vivci;

vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );

vkCreateImageView( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

VkSemaphoreCreateInfo vsci;

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

result = vkEndCommandBuffer( CommandBuffers[ nextImageIndex ]);

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

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

vkCmdEndRenderPass( CommandBuffers[ nextImageIndex ]);
```

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

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

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

```
Creating the Swap Chain Images and Image Views

```
Creating a Swap Chain

```
Rendering into the Swap Chain, I

```
```
vkCreateFence
LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

vkGetDeviceQueue
LogicalDevice, FindQueueFamilyThatDoesGraphics, 0,
OUT &presentQueue);

vkQueueSubmit
presentQueue, 1, IN &vsi, IN renderFence);

vkWaitForFences
LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);

vkQueuePresentKHR
presentQueue, IN &vpi);
int integratedSelect = -1;
int discreteSelect = -1;

fprintf( FpDebug, "occlusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.occlusionQueryPrecise );
fprintf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport );
fprintf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShader );
fprintf( FpDebug, "geometryShader = %2d\n", PhysicalDeviceFeatures.geometryShader );

fprintf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64 );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStatisticsQuery );
fprintf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines );
fprintf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect );
fprintf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16 );
fprintf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64 );

vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );

if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )   fprintf( FpDebug, " (Discrete GPU)\n" );
fprintf( FpDebug, "Physical Device Type: %d =\n", vpdp.deviceType) ;
fprintf( FpDebug, "Device ID: 0x%04x\n", vpdp.deviceID );
fprintf( FpDebug, "Driver version: %d\n", vpdp.apiVersion );
fprintf( FpDebug, "API version: %d\n", vpdp.apiVersion );

fprintf( FpDebug, "Device %2d:\n", i );

if( result != VK_SUCCESS )
  vkGetPhysicalDeviceProperties( IN physicalDevices[ i ], OUT &vpdp );
VkPhysicalDeviceProperties vpdp;

fprintf( FpDebug, "Pipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0] );
fprintf( FpDebug, "Device Name: %s\n", vpdp.deviceName );

if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )    fprintf( FpDebug, " (Virtual GPU)\n" );
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU ) fprintf( FpDebug, " (Integrated GPU)\n" );
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )            fprintf( FpDebug, " (CPU)\n" );

return VK_SHOULD_EXIT;

fprintf( FpDebug, "Could not get the physical device properties of device %d\n", i );

for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ )
  fprintf( FpDebug, "Memory Heaps:\n", vpdmp.memoryHeapCount );
  fprintf(FpDebug, "Heap %d: \n", i);
  VkMemoryHeap vmh = vpdmp.memoryHeaps[ i ];
  if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT  ) != 0 )     fprintf( FpDebug, " DeviceLocal" );
  else     fprintf( FpDebug, " size = 0x%08lx", (unsigned long int)vmh.size);

  VkMemoryType vmt = vpdmp.memoryTypes[ i ];
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT   ) != 0 )    fprintf( FpDebug, " LazilyAllocated" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT       ) != 0 )   fprintf( FpDebug, " HostCached" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT       ) != 0 )   fprintf( FpDebug, " HostVisible" );
  if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT      ) != 0 )    fprintf( FpDebug, " DeviceLocal" );
  fprintf( FpDebug, "Memory %2d: \n", i);
11 Memory Types:

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

2 Memory Heaps:

- Heap 0: size = 0x7c00000 DeviceLocal
- Heap 1: size = 0x1c0000000

{code}
Here's What I Got
{code}

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

Asking About the Physical Device's Queue Families

{code}
Here's What I Got
{code}

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

Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: a More Typical (and Simplified) Block Diagram

Vulkan: a More Typical (and Simplified) Block Diagram

Looking to See What Device Layers are Available

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

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

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

{code}
Looking to See What Device Extensions are Available

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

What Device Layers and Extensions are Available

```c
4 physical device layers enumerated:
0x00401063   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0x00401072   1  'VK_LAYER_LUNARG_core_validation'  'LunarG Validation Layer'
0x00401072   1  'VK_LAYER_LUNARG_object_tracker'  'LunarG Validation Layer'
0x00401072   1  'VK_LAYER_LUNARG_parameter_validation'  'LunarG Validation Layer'
```

```c
4 device extensions enumerated for 'VK_LAYER_NV_optimus':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
```

```c
4 device extensions enumerated for 'VK_LAYER_LUNARG_core_validation':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
```

```c
4 device extensions enumerated for 'VK_LAYER_LUNARG_object_tracker':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
```

```c
4 device extensions enumerated for 'VK_LAYER_LUNARG_parameter_validation':
0x00000001  'VK_EXT_validation_cache'
0x00000004  'VK_EXT_debug_marker'
```

Vulkan: Creating a Logical Device

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

Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

http://cs.oregonstate.edu/~mjb/vulkan
Creating a Pipeline

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

Creating a Graphics Pipeline

Which Pipeline State Variables can be Changed Dynamically

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

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

Push Constants

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

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

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

```cpp
layout(push_constant) uniform mat4 modelMatrix;
// Matrix;
```

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

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

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

An Robotic Example using Push Constants

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

Where each arm is represented by:

```cpp
struct arm
{
    glm::mat4 armMatrix; // scale factor in x
    glm::vec3 armColor;
    glm::mat4 armMatrix = glm::mat4();
    Arm1.armColor = glm::vec3( 0.0, 0.74, 0.13 );
    Arm1.armMatrix = glm::mat4();
    Arm2.armColor = glm::vec3( 1.0, 0.5, 0.4 );
    Arm2.armMatrix = glm::mat4();
    Arm3.armColor = glm::vec3( 0.0, 0.11 );
    Arm3.armMatrix = glm::mat4();
    Arm3.armMatrix = 2.0;
};
```

In the Reset Function

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

Setting up the Push Constants for the Pipeline Structure

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

```cpp
VkPushConstantRange vpcr[1] = { 
    .sType = VK_STRUCTURE_TYPE_PUSH_CONSTANT_RANGE,
    .stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
    .offset = 0,
    .size = sizeof(glm::mat4),
};
```

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

```cpp
vplci = VkPipelineLayoutCreateInfo{
    .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,
    .pushConstantRangeCount = 1,
    .pSetLayouts = DescriptorSetLayouts,
    .setLayoutCount = 4,
    .pSetLayouts = NULL,
    .flags = 0,
    .pNext = NULL,
    .pPushConstantRanges = &vpcr[0],
};
```

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

```cpp
vkBindPipeline( CommandBuffer, PipelineLayout, pipelineHandle );
```

```cpp
vkCmdPushConstants( CommandBuffer, PipelineLayout, stageFlags, offset, size, pValues );
```
In the **UpdateScene Function**

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

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

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

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

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 );

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

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

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

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

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

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

In the **Vertex Shader**

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

layout( location = 0 ) in vec3 aVertex;

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

bVertex = vec3( RobotArm.armMatrix * vec4(bVertex, 1.));

// Projection * Viewing * Modeling matrices
```

### Getting Information Back from the Graphics System

Mike Bailey
mjb@cs.oregonstate.edu

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

- There are 3 types of Queries: Occlusion, Pipeline Statistics, and Timestamp
- Vulkan requires you to first setup “Query Pools”, one for each specific type
- This indicates that Vulkan thinks that Queries are time-consuming (relatively) to setup, and thus better to set them up in program-setup than in program-runtime
vkCmdResetQueryPool(CommandBuffer, occlusionQueryPool, 0, 1);
vkCmdBeginQuery(CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT);
vkCmdEndQuery(CommandBuffer, occlusionQueryPool, 0);

```c
#define DATASIZE 128
uint32_t data[DATASIZE];
result = vkGetQueryPoolResults(LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags);
```

Occlusion Queries count the number of fragments drawn between the `vkCmdBeginQuery` and the `vkCmdEndQuery` that pass both the Depth and Stencil tests. This is commonly used to see what level-of-detail should be used when drawing a complicated object.

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

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

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

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

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

Compute Shaders

To learn more about compute shaders, visit [http://cs.oregonstate.edu/~mjb/vulkan](http://cs.oregonstate.edu/~mjb/vulkan).
Here is how you create a Compute Pipeline

```
layout( std140, set = 0, binding = 0 )
buffer Pos
{
vec4  Positions[   ]; // array of structures
};
layout( std140, set = 0, binding = 1 )
buffer Vel
{
vec4  Velocities[   ]; // array of structures
};
layout( std140, set = 0, binding = 2 )
buffer Col
{
vec4  Colours[   ]; // array of structures
};
```

Start by Creating the Data Buffers

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

```
void * pGpuMemory;
vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
// 0 and 0 are offset and flags
memcpy( pGpuMemory, data, (size_t)myBuffer.size );
vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
return VK_SUCCESS;
```

Fill the Data Buffer

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

Creating a Shader Storage Buffer

```
vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );
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 );
```

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

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

Create the Compute Pipeline Layout

```
result = vkCreateBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vma );
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 );
```
Create the Compute Pipeline

VkPipeline

ComputePipeline

VkPipelineShaderStageCreateInfo vpssci;

vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0;
vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkComputePipelineCreateInfo vcpci[1];

vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;

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

The Particle System Compute Shader – Setup

```cpp
#version 430
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 )  buffer  Pos {
  vec4  Positions[ ]; // array of structures
};
layout( std140, set = 0, binding = 1 )  buffer  Vel {
  vec4  Velocities[ ]; // array of structures
};
layout( std140, set = 0, binding = 2 )  buffer  Col {
  vec4  Colours[ ]; // array of structures
};
layout( local_size_x = 64,  local_size_y = 1, local_size_z = 1 )   in;
```

The Particle System Compute Shader – The Physics

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

const VECTOR  G        =  VECTOR( 0., -9.8, 0. );
const float        DT        =  0.1;
const SPHERE Sphere = vec4( -100., -800., 0.,  600. ); // x, y, z, r
... 
uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case
POINT        p  = Positions[ gid ].xyz;
VELOCITY v  = Velocities[ gid ].xyz;
POINT         pp = p + v*DT + .5*DT*DT*G;
VELOCITY  vp = v + G*DT;
Positions[ gid ].xyz  = pp;
Velocities[ gid ].xyz = vp;
```

Dispatching the Compute Shader from the Command Buffer

```cpp
const int NUM_PARTICLES         = 1024*1024;
const int NUM_WORK_ITEMS     =              64;
const int NUM_X_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;...

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatch( CommandBuffer, NUM_X_WORK_GROUPS, 1,  1 );
```

Graphics Trick Alert: Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough (and it is), nobody will ever know.
What Are Specialization Constants?

In Vulkan, all shaders get halfway-compiled by SPIR-V and then the rest-of-the-way compiled by the Vulkan driver. Normally, the half-way compile fixes all constant values and compiles the code that uses them.

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

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

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

Why Do We Need Specialization Constants?

Specialization Constants could be used for:

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

Specialization Constant Example — Setting an Array Size

In the compute shader

```cpp
layout( constant_id = 7 ) const int ASIZE = 32;

int array[ASIZE];
```

In the Vulkan C/C++ program:

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

`VkPipelineShaderStageCreateInfo` `vpsscii`:

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

`VkComputePipelineCreateInfo` `vcpci`:

```cpp
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcki[0].pNext = nullptr;
vcki[0].flags = 0;
vcki[0].stage = vpsscii;
vcki[0].layout = ComputePipelineLayout;
vcki[0].basePipelineHandle = VK_NULL_HANDLE;
vcki[0].basePipelineIndex = 0;
```

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

Linking the Specialization Constants into the Compute Pipeline
### Specialization Constant Example – Setting Multiple Constants

**In the compute shader:**

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

**In the C/C++ program:**

```c
struct abc { int a, int b, float c; } abc;
VkSpecializationMapEntry vsme[3];
vsme[0].constantID = 9;
vsm[0].offset = offsetof( abc, a );
vsm[0].size = sizeof(abc.a);
vsme[1].constantID = 10;
vsm[1].offset = offsetof( abc, b );
vsm[1].size = sizeof(abc.b);
vsme[2].constantID = 11;
vsm[2].offset = offsetof( abc, c );
vsm[2].size = sizeof(abc.c);
```

```c
VkSpecializationInfo vsi;
vsi.mapEntryCount = 3;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsi.pData = &abc; // array of all the Specialization Constants
```

### Specialization Constants – Setting the Number of Work-items Per Work-Group in the Compute Shader

**In the compute shader:**

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

**In the C/C++ program:**

```c
int numXworkItems = 64;
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 12;
vsm[0].offset = 0;
vsm[0].size = sizeof(int);
```

```c
VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsme[0];
vsi.dataSize = sizeof(int);
vsi.pData = &numXworkItems;
```

### Synchronization

**Where Synchronization Fits in the Overall Block Diagram**

- **Application**
  - Instance
  - Physical Device
  - Logical Device
  - Logical Device
  - Queue
  - Queue
  - Command Buffer
  - Command Buffer
  - Event

**Semaphores**

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

**Creating a Semaphore**

```c
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;
VkSemaphore semaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &semaphore );
```
Semaphores Example during the Render Loop

```c
// Semaphores Example during the Render Loop

VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;

vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

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

uint32_t nextImageIndex;

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkSubmitInfo vsi;

vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

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, &vsi, renderFence);
```

Fences

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

Fence Example

```c
// Fence Example

VkFenceCreateInfo vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

vfci.pNext = nullptr;

vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT; // = 0

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, &vsi, renderFence);
```

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
// Controlling Events from the Host

VkEventCreateInfo veci;

veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;

veci.pNext = nullptr;

veci.flags = 0;

VkEvent event;

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

result = vkSetEvent(LogicalDevice, event);

result = vkResetEvent(LogicalDevice, event);

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

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

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

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

1. The cross-streets are named after pipeline stages.
2. All traffic lights start out green.
3. There are special sensors at all intersections that will know when the first car in the src group enters that intersection.
4. There are connections from those sensors to the traffic lights so that when the first car in the src group enters the intersection, the proper dst traffic light will be turned red.
5. When the last car in the dst group completely makes it through its intersection, the proper dst traffic light can be turned back to green.
6. The Vulkan command pipeline ordering is this: (1) the src cars get released, (2) the dst cars get released, (3) the pipeline barrier is invoked (which turns some lights red), (4) the proper dst traffic light is turned red, (5) the first car in the dst group enters its intersection, the proper dst traffic light can be turned back to green.

**Pipeline Stages**

- Top of Pipe
- Bottom of Pipe
- Transfer
- Compute
- Fragment
- Vertex
- Host
- Shader
- Input
- Uniform
- Index
- Indirect

**Access Masks**

- Access Indirect Command Read Bit
- Access Index Read Bit
- Access Attribute Read Bit
- Access Input Attachment Read Bit
- Access Shader Read Bit
- Access Color Attachment Read Bit
- Access Depth Stencil Attachment Read Bit
- Access Transfer Read Bit
- Access Host Read Bit
- Access Memory Read Bit
- Access Indirect Command Write Bit
- Access Index Write Bit
- Access Attribute Write Bit
- Access Input Attachment Write Bit
- Access Shader Write Bit
- Access Color Attachment Write Bit
- Access Depth Stencil Attachment Write Bit
- Access Transfer Write Bit
- Access Host Write Bit
- Access Memory Write Bit

**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_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
Access Operations and what Pipeline Stages they can be used in

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

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

VkImageLayout – How an Image gets Laid Out in Memory
depends on how it will be Used
Multisampling is a computer graphics technique to improve the quality of your output image by looking inside every pixel to see what the rendering is doing there. There are two approaches to this:

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

   ![Supersampling Diagram]

2. **Multisampling**: Perform a single color render for the one pixel. Then, pick some number of unique sub-pixels within that pixel and perform depth and stencil tests there. Assign the single color to all the sub-pixels that made it through the depth and stencil tests.

   ![Multisampling Diagram]
Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

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

# Fragment Shader calls = 8

Multisampling

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

# Fragment Shader calls = 2

VkPipelineMultisampleStateCreateInfo

\[ \text{vpmsci} \]

\[ \text{vpmsci.sType} = \text{VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO} \]

\[ \text{vpmsci.pNext} = \text{nullptr} \]

\[ \text{vpmsci.flags} = 0 \]

\[ \text{vpmsci.rasterizationSamples} = \text{VK_SAMPLE_COUNT_8_BIT} \]

\[ \text{vpmsci.sampleShadingEnable} = \text{VK_TRUE} \]

\[ \text{vpmsci.minSampleShading} = 0.5f \]

\[ \text{vpmsci.pSampleMask} = (\text{VkSampleMask} \*) \text{nullptr} \]

\[ \text{vpmsci.alphaToCoverageEnable} = \text{VK_FALSE} \]

\[ \text{vpmsci.alphaToOneEnable} = \text{VK_FALSE} \]

VkGraphicsPipelineCreateInfo

\[ \text{vgpci} \]

\[ \text{vgpci.sType} = \text{VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO} \]

\[ \text{vgpci.pNext} = \text{nullptr} \]

\[ \text{vgpci.pMultisampleState} = \& \text{vpmsci} \]

\[ \text{result} = \text{vkCreateGraphicsPipelines} ( \text{LogicalDevice}, \text{VK_NULL_HANDLE}, \text{1}, \& \text{vgpci}, \text{PALLOCATOR}, \text{OUT pGraphicsPipeline} ) \]

Setting up the Image

 VkPyramid:MultisampleDataCreateInfo

\[ \text{vpmsci} \]

\[ \text{vpmsci.sType} = \text{VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO} \]

\[ \text{vpmsci.pNext} = \text{nullptr} \]

\[ \text{vpmsci.flags} = 0 \]

\[ \text{vpmsci.rasterizationSamples} = \text{VK_SAMPLE_COUNT_8_BIT} \]

\[ \text{vpmsci.sampleShadingEnable} = \text{VK_TRUE} \]

\[ \text{vpmsci.minSampleShading} = 0.5f \]

\[ \text{vpmsci.pSampleMask} = (\text{VkSampleMask} \*) \text{nullptr} \]

\[ \text{vpmsci.alphaToCoverageEnable} = \text{VK_FALSE} \]

\[ \text{vpmsci.alphaToOneEnable} = \text{VK_FALSE} \]

VirtualPipelineCreateCreateInfo

\[ \text{vpmsci} \]

\[ \text{vpmsci.sType} = \text{VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO} \]

\[ \text{vpmsci.pNext} = \text{nullptr} \]

\[ \text{vpmsci.pMultisampleState} = \& \text{vpmsci} \]

\[ \text{result} = \text{vkCreateGraphicsPipelines} ( \text{LogicalDevice}, \text{VK_NULL_HANDLE}, \text{1}, \& \text{vgpci}, \text{PALLOCATOR}, \text{OUT pGraphicsPipeline} ) \]
Setting up the Image

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

Setting up the Image

```c
VkAttachmentReference colorReference;
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

```c
VkAttachmentReference depthReference;
depthReference.attachment = 1;
depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```c
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 = (uint3 introducing the Vulkan Computer Graphics API

Thanks for coming today!

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