Mike Bailey
mjb@cs.oregonstate.edu

Introduction to the
Vulkan
Computer Graphics API

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

Mike Bailey

• Professor of Computer Science, Oregon State University
• Has been in computer graphics for over 30 years
• Has had over 7,000 students in his university classes
• mjb@cs.oregonstate.edu

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

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.
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.
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.

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
- OpenGL and OpenGL have adopted SPIR-V as well

The Basic Computer Graphics Pipeline, Shader-style

- Model Transform
- View Transform
- Project Transform
- Fragment Processing, Texturing, Per-fragment Lighting

The Basic Computer Graphics Pipeline, Vulkan-style

- MC = Model Vertex Coordinates
- WC = World Vertex Coordinates
- EC = Eye Vertex Coordinates
- Per-vertex in variables:
  - gl_Position
  - gl_ModelViewMatrix
  - gl_ProjectionMatrix
  - gl_ModelViewProjectionMatrix
- Per-vertex out variables:
  - gl_Vertex, gl_Normal, gl_Color
- Uniform Variables:
  - Vertex Shader
  - Fragment Shader
- Framebuffer
Moving part of the driver into the application

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

Vulkan 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. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render-…

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjba@cs.oregonstate.edu

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

Sample Program Keyboard Inputs

'i', 'I': Toggle using a vertex buffer only vs. an index buffer
'T', 'L': Toggle lighting off and on
'm', 'M': Toggle display mode (textures vs. colors, for now)
'p', 'P': Pause the animation
'q', 'Q': Quit the program
Esc: Quit the program
'f', 'F': Toggle rotation-animation and using the mouse
1, 4, 9 Number of instances

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 setup just to show you what could go there.

Caveats on the Sample Code, II

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'ed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.
**Main Program**

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

**InitGraphics()**

```c
void InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    InstaInstance( );
    InitGLFW( );
    Init02CreateDebugCallbacks( );
    Init03PhysicalDeviceAndQueueFamilyProperties( );
    InstLogicalDeviceAndQueue( );
    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 );
    Init07TextureBuffersAndBFordFile( "puppy.bmp", &MyPuppyTexture );
    Init08Swapchain( );
    Init09DepthStencilImage( );
    Init10RenderPasses( );
    Init11Framebuffers( );
    Init12SwapShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SwapShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool( );
    Init13DescriptorLayouts();Init13DescriptorSets( );
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
                                         VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
    A Colored Cube
```
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 );
   ```

The Vertex Data is in a Separate File

```cpp
#include "SampleVertexData.cpp"
struct vertex
{
    glm::vec3 position;  // move position from struct vertex to SampleVertexData
    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. },
    ...
};
```
Vulkan Code has a Distinct "Style" of Setting Information in structs and then Passing that Information as a pointer-to-struct.

```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 = VK_USAGE_UNIFORM_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = nullptr;

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

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

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

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, &MatrixBufferMemoryHandle );

result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```

Vulkan Conventions

- `VkXXX` is a typedef, probably a struct
- `vkXXX()` is a function call
- `VK_XXX` is a constant

My Conventions

- "Init" in a function call name means that something is being setup that only needs to be setup once
- The number after "Init" gives you the ordering

In the source code, after `main()` comes `InitGraphics()`, then all of the `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

```c
unit32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT &physicalDevices[0] );
```

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

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

result = vkEnumeratePhysicalDevices( Instance, OUT &count, NULL );
result = vkEnumeratePhysicalDevices( Instance, OUT &count, &physicalDevices[0] );
```

Your Sample2019.zip File Contains This

- Linux shader compiler
- Windows shader compiler
- Double-click here to launch Visual Studio 2019 with this solution

The "19" refers to the version of Visual Studio, not the year of development.
```c
#define REPORT(s)               { PrintVkError( result, s );  fflush(FpDebug); }
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****
”, s );  fflush(FpDebug); }

bool Paused;bool Verbose;

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

---

**Geometry vs. Topology**

- **Original Object**
  - Geometry: changed
  - Topology: same (1-2-3-4-1)
  - Change topology

- **Geometry**
  - Where things are (e.g., coordinates)

- **Topology**
  - How things are connected

---

**Vulkan Topologies**

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

Vertex Orientation Issues

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system. Internally, however, the Vulkan pipeline uses a left-handed system:

The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the projection matrix, like this: ProjectionMatrix[ 1 ][ 1 ] *= -1.; This is like saying "Y' = -Y".

VkPipelineRasterizationStateCreateInfo vprsci;

vprsci.cullMode = VK_CULL_MODE_NONE;

Vulkan's change in coordinate systems can mess up the backface culling. So I recommend, at least at first, that you do no culling.

VertexPipelineRasterizationStateCreateInfo vprsci;

vprsci.cullMode = VK_CULL_MODE_NONE;

Triangles Represented as an Array of Structures

From the file SampleVertexData.cpp:

Modeled in right-handed coordinates
Non-indexed Buffer Drawing

- From the file `SampleVertexData.cpp`:

```cpp
def 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. }`.

- From the file `SampleVertexData.cpp`:

```cpp
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
```

- `VkResult Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )`
  - `VkResult result; result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;`

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

A Preview of What Init05DataBuffer Does

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

Telling the Pipeline about its Input

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

C/C++:

```cpp
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;
```

GLSL Shader:

```glsl
layout( location = 0 ) in vec3 aVertex; // one of these per buffer data buffer
layout( location = 1 ) in vec3 aNormal; // which binding if this is
layout( location = 2 ) in vec3 aColor;  // bytes between successive structs
layout( location = 3 ) in vec2 aTexCoord; // bytes between successive structs
```

MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer( IN MyBufferData.size, OUT MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );

- `VkResult Init05MyVertexDataBuffer( IN MyBufferData.size, OUT MyVertexDataBuffer )`
  - `VkResult result; result = Init05DataBuffer( MyBufferData.size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, MyVertexDataBuffer );
    return result;`
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;
};

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

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

struct JustIndexData
{
    0, 2, 3,
    0, 3, 1,
    4, 5, 7,
    4, 7, 6,
    1, 3, 2,
    1, 2, 0,
    0, 1, 5,
    0, 5, 4,
};

int JustIndexData[] =
{ 0, 2, 3,
  0, 3, 1,
  4, 5, 7,
  4, 7, 6,
  1, 3, 2,
  1, 2, 0,
  0, 1, 5,
  0, 5, 4,
};
VkResult
Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    return result;
}

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

Drawing with an Indexed Buffer

VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
// 0, 1 = firstBinding, bindingCount
vkCmdBindIndexBuffer(CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32);
const uint32_t vertexCount = sizeof( JustVertexData ) / sizeof( JustVertexData[0] );
const uint32_t indexCount = sizeof( JustIndexData ) / sizeof( JustIndexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
const uint32_t vertexOffset = 0;
vkCmdDrawIndexed(CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,
vertexOffset, firstInstance);

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

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

Thus, when using index-ed buffer drawing, you need to create a new vertex struct if any of {position, normal, color, texCoords} changes from what was previously-stored at those coordinates.
Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

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

From the Quick Reference Card

Vulkan: Creating a Data Buffer

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

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

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

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

Finding the Right Type of Memory

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

MyBuffer MyMatrixUniformBuffer;

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

Something I've Found Useful

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

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

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

## Here's a C struct to hold some uniform variables

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

Here's the shader code to access those uniform variables

```c
layout( std140, set = 0, binding = 0 ) uniform matBuf 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.);
glm::vec3 up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4(); // identity
Matrices.uViewMatrix = glm::lookAt(eye, look, up);
Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan’s LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
```

## The Parade of Data

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

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

```c
struct matBuf Matrices;
```

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

uniform matBuf Matrices;

There is one more step in here: Descriptor Sets.
The Descriptor Set for the Buffer

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
VkDescriptorBufferInfo vdbi0;
vdbi0.buffer = MyMatrixUniformBuffer.buffer;
vdbi0.offset = 0;       // bytes
vdbi0.range = sizeof(Matrices);

VkWriteDescriptorSet vwds0;       // ds 0:
vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds0.pNext = nullptr;
vwds0.dstSet = DescriptorSets[0];
vwds0.dstBinding = 0;
vwds0.dstArrayElement = 0;
vwds0.descriptorCount = 1;
vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds0.pBufferInfo = &vdbi0;

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

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;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );

    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );

    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;

    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );
               // 0 is the offset
    return result;
}
```

Filling the Data Buffer

```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;
}
```

Creating and Filling the Data Buffer – the Details

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

- 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 #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 ) . . .

Push Constants:
layout( push_constant ) . . .

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

Specialization Constants for Compute Shaders:
layout( local_size_x_id = 3, 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:
  ```c
  layout(std140, set = 0, binding = 0) uniform matBuf {
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
  } Matrices;
  ```

- 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 OpenCL 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


  Shaderfile extensions:
  - `vert` Vertex
  - `tesc` Tessellation Control
  - `tese` Tessellation Evaluation
  - `geom` Geometry
  - `frag` Fragment
  - `comp` Compute

  (Can be overridden by the `-S` option)

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

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

Running glslangValidator.exe

- `glslangValidator.exe -V sample.vert -o sample vert.spir-vert`
- `glslangValidator.exe -V sample.frag -o sample frag.spir-frag`
- `glslangValidator.exe -V sample.geom -o sample geom.spir-geom`
- `glslangValidator.exe -V sample.comp -o sample comp.spir-comp`
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'
        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 );
    VkShaderModule ShaderModuleVertex;
    . . .
    VkShaderModuleCreateInfo vsmci;
    vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    vsmci.pNext = nullptr;
    vsmci.flags = 0;
    vsmci.codeSize = size;
    vsmci.pCode = (uint32_t *)code;
    VkResult result = vkCreateShaderModule( LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex );
    fprintf( FpDebug, "Shader Module '%s' successfully loaded
    delete [  ] code;
    return result;
}
```

Reading a SPIR-V File into a Vulkan Shader Module

---

Reading a SPIR-V File into a Shader Module

```
VkPipelineShaderStageCreateInfo
VkPipelineVertexInputStateCreateInfo
VkViewportStateCreateInfo
VkPipelineRasterizationStateCreateInfo
VkPipelineColorBlendStateCreateInfo
VkPipelineDynamicStateCreateInfo
```

Vulkan: Creating a Graphics Pipeline

---

Vulkan: Creating a Graphics Pipeline
SPIR-V: More Information

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

GLFW

Mike Bailey
mjb@cs.oregonstate.edu

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

Setting Up GLFW

```c
void InitGLFW() {
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API); // don't want a context
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
    MainWindow = glfwCreateWindow(Width, Height, "Vulkan Sample", NULL, NULL);
    VkResult result = glfwCreateWindowSurface(Instance, MainWindow, NULL, &Surface);
    glfwSetErrorCallback(GLFWErrorCallback);
    glfwSetKeyCallback(MainWindow, GLFWKeyboard);
    glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
    glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

GLFW 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, "Unknow key hit: 0x%04x = '%c'
", key, key);
        }
    }
}
```
GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

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

Looping and Closing GLFW

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

GLM

```
```

Vulkan.

Mike Bailey
mj@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
What is GLM?

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

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

You can find it at:

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

You invoke GLM like this:

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

If GLM is not installed in a system place, put it somewhere you can get access to. Later on, these notes will show you how to use it from there.

Why are we even talking about this?

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

```cpp
gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );
glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glRotatef( (GLfloat)Xrot, 1., 0., 0. );
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

you would now have to say:

```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

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.

### Constructor:

```cpp
glm::mat4(); // identity matrix
glm::vec4();
```

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

### Multiplications:

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

### Emulating OpenGL transformations with concatenation:

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

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

```cpp
glm::mat4 glm::ortho( 0., 0., 3., 0., 0., 0., 0., 1., 0., 0., 0. );
glRotatef( glm::float4(0., 0., 1., 0.));
glRotatef( glm::float4(1., 0., 0., 0.));
glScalef( glm::float4(0., 0., 0., 1.));
```

### Viewing (assign, not concatenate):

```cpp
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

```cpp
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!

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

Right!

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

Mike Bailey
mj@cs.oregonstate.edu

http://cs.oregonstate.edu/~mj/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.

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

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

Making each Instance look differently -- Approach #1

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

`gl_InstanceIndex` starts at 0.

In the vertex shader:

```
int NUMINSTANCES = 16;
float DELTA = 3.0;
float xdelta = DELTA * float(gl_InstanceIndex % 4);
float ydelta = DELTA * float(gl_InstanceIndex / 4);
vColor = vec3(1., float((1.+gl_InstanceIndex) / float(NUMINSTANCES)), 0.);
xdelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.;
ydelta -= DELTA * sqrt(float(NUMINSTANCES)) / 2.;
vec4 vertex = vec4(aVertex.xyz + vec3(xdelta, ydelta, 0.), 1.);
gl_Position = PVM * vertex;
```

Making each Instance look differently -- Approach #2

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

Still uses `gl_InstanceIndex`.

In the vertex shader:

```
layout(std140, set = 3, binding = 0) uniform colorBuf{
vec3 uColors[1024];
} Colors;
out vec3 vColor;
...
int index = gl_InstanceIndex % 1024;  // 0 - 1023
vColor = Colors.uColors[index];
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```
How We Constructed the Graphics Pipeline Structure Before

VkVertexInputBindingDescription
vvibd[0]]
// 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 Write the Vertex Shader Now

```glsl
#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. );
}
```

In OpenGL

OpenGL puts all uniform data in the same "set", but with different binding numbers, so you can get at each one. Each uniform variable gets updated one-at-a-time.

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

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

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

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

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

### What are Descriptor Sets?

<table>
<thead>
<tr>
<th>CPU: Uniform data created in a C++ data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Known the CPU data structure</td>
</tr>
<tr>
<td>- Known where the data starts</td>
</tr>
<tr>
<td>- Known the data size</td>
</tr>
<tr>
<td>- Doesnt know the GPUs or GPU data structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU: Uniform data in a &quot;blob&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Known the shader data structure</td>
</tr>
<tr>
<td>- Doesnt know where each piece of data starts</td>
</tr>
</tbody>
</table>

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

struct lightBuf
{
    glm::vec4 uLightPos;
};

struct miscBuf
{
    float uTime;
    int uMode;
};

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

layout( std140, set = 1, binding = 0 ) uniform lightBuf
{
    vec4 uLightPos;
} Light;

layout( std140, set = 2, binding = 0 ) uniform miscBuf
{
    float uTime;
    int uMode;
} Misc;

layout( set = 3, binding = 0 ) uniform sampler2D uSampler;
```

### Step 1: Descriptor Set Pools

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

```cpp
device

vkCreateDescriptorPool() |

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

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

Step 4: Allocating the GPU Memory for Descriptor Sets

```cpp
VkResult Init13DescriptorSets() {
    VkResult result;
    VkDescriptorSetAllocateInfo vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;
    result = vkAllocateDescriptorSets(LogicalDevice, IN &vdsai, OUT &DescriptorSets[0]);
    return result;
}
```

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

```cpp
VkDescriptorBufferInfo vdbi0;
    vdbi0.buffer = MyMatrixUniformBuffer.buffer;
    vdbi0.offset = 0;
    vdbi0.range = sizeof(Matrices);

VkDescriptorBufferInfo vdbi1;
    vdbi1.buffer = MyLightUniformBuffer.buffer;
    vdbi1.offset = 0;
    vdbi1.range = sizeof(Light);

VkDescriptorBufferInfo vdbi2;
    vdbi2.buffer = MyMiscUniformBuffer.buffer;
    vdbi2.offset = 0;
    vdbi2.range = sizeof(Misc);

VkDescriptorImageInfo vdii0;
    vdii0.sampler = MyPuppyTexture.texSampler;
    vdii0.imageView = MyPuppyTexture.texImageView;
    vdii0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```

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

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

This struct identifies what texture sampler and image view it owns
Step 5a: Tell the Descriptor Sets where their CPU data is

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

This struct links a Descriptor Set to the buffer it is pointing to.

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

This struct links a Descriptor Set to the buffer it is pointing to.

```
// ds 2:
VkWriteDescriptorSet vwds2;
vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds2.pNext = nullptr;
vwds2.dstSet = DescriptorSets[2];
vwds2.dstBinding = 0;
vwds2.dstArrayElement = 0;
vwds2.descriptorCount = 1;
vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
vwds2.pBufferInfo = &vdbi2;
vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
vwds2.pTexelBufferView = (VkBufferView *)nullptr;
```

This struct links a Descriptor Set to the image it is pointing to.

```
// ds 3:
VkWriteDescriptorSet vwds3;
vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds3.pNext = nullptr;
vwds3.dstSet = DescriptorSets[3];
vwds3.dstBinding = 0;
vwds3.dstArrayElement = 0;
vwds3.descriptorCount = 1;
vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
vwds3.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
vwds3.pImageInfo = &vdii0;
vwds3.pTexelBufferView = (VkBufferView *)nullptr;
```

This struct links a Descriptor Set to the image it is pointing to.

Step 5b: Upload CPU data into the GPU Descriptor Set

```
uint32_t copyCount = 0;
if (this could also have been done with one call and an array of VkWriteDescriptorSets:

vkUpdateDescriptorSets( LogicalDevice, IN &vwds0, IN copyCount, (VkCopyDescriptorSets *)nullptr );
```

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

```
// ds 0:
VkWriteDescriptorSet vwds0;
```

This struct links a Descriptor Set to the buffer it is pointing to.

```
// ds 1:
VkWriteDescriptorSet vwds1;
```

This struct links a Descriptor Set to the buffer it is pointing to.

```
// ds 2:
VkWriteDescriptorSet vwds2;
```

This struct links a Descriptor Set to the buffer it is pointing to.

```
// ds 3:
VkWriteDescriptorSet vwds3;
```

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

```c
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.

---

**Textures**

Mike Bailey

mjb@cs.oregonstate.edu

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

---

**Triangles in an Array of Structures**

```c
struct vertex
{
    glm::vec3       position; // vertex #0:
    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. }}
};
```

---

**Memory Types**

- **CPU Memory**
- **GPU Memory**
  - Host Visible
  - Device Local

Texture Sampling Hardware
- RGBA to the Shader
**Memory Types**

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

**Intel Integrated Graphics:**
- Memory 0: DeviceLocal
- Memory 1: DeviceLocal HostVisible HostCoherent
- Memory 2: DeviceLocal HostVisible HostCoherent HostCached

**Texture Sampling Parameters**

### OpenGL

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

### Vulkan

```c
VkSamplerCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
vsci.pNext = nullptr;vsci.flags = 0;
vsci.magFilter = VK_FILTER_LINEAR;
vsci.minFilter = VK_FILTER_LINEAR;
vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

result = vkCreateSampler(LogicalDevice, IN &vsci, PALLOCATOR, pTextureSampler);

### Texture Mip-mapping

- Average 4 pixels to make a new one

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

*Latin: multiformarum, "many things in a small place"*
```c
#ifdef CHOICES
int InitTextureBuffer(INOUT MyTexture * pMyTexture)
{
    VkResult result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage); // allocated, but not filled

    uint32_t texWidth = pMyTexture->width;
    uint32_t texHeight = pMyTexture->height;
    unsigned char *texture = pMyTexture->pixels;
    VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each

    VkImageCreateInfo vici;
    VkMemoryAllocateInfo vmai;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.tiling = VK_IMAGE_TILING_LINEAR;
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &vdm);
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.usage |= VK_IMAGE_USAGE_SAMPLED_BIT;
    vici.usage |= VK_IMAGE_USAGE_TRANSFER_DST_BIT;
    vici.usage |= VK_IMAGE_USAGE_STORAGE_BIT;
    vici.usage |= VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
    vici.usage |= VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
    vici.usage |= VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;
    vici.usage |= VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT;
    if (Verbose)
    {
        fprintf(FpDebug, "Subresource Layout:
");
        fprintf(FpDebug, "	offset = %lld
", vsl.offset);
        fprintf(FpDebug, "	size = %lld
", vsl.size);
        fprintf(FpDebug, "	rowPitch = %lld
", vsl.rowPitch);
        fprintf(FpDebug, "	arrayPitch = %lld
", vsl.arrayPitch);
        fprintf(FpDebug, "	depthPitch = %lld
", vsl.depthPitch);
        fflush(FpDebug);
    }

    void * gpuMemory;
    vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
    vkUnmapMemory(LogicalDevice, vdm);
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled

    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
    if (Verbose)
    {
        fprintf(FpDebug, "Texture vmr.size = %lld
", vmr.size);
        fprintf(FpDebug, "Texture vmr.alignment = %lld
", vmr.alignment);
        fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
        fflush(FpDebug);
    }

    VkMemoryAllocateInfo vmai;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal();
    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);
}
```
copy pixels from the staging image to the texture:

```cpp
VkCommandBufferBeginInfo vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

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

// *******************************************************************************
// transition the texture buffer layout:// *******************************************************************************
{
    VkImageSubresourceRange visr;
    visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visr.baseMipLevel = 0;
    visr.levelCount = 1;
    visr.baseArrayLayer = 0;
    visr.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = ... = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = textureImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    vimb.subresourceRange = visr;
    vkCmdPipelineBarrier(TextureCommandBuffer,
                          VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
                          0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr,
                          1, IN &vimb);
    // now do the final image transfer:
    VkImageSubresourceLayers visl;
    visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visl.baseArrayLayer = 0;
    visl.mipLevel = 0;
    visl.layerCount = 1;
    VkOffset3D                              vo3;
    vo3.x = 0;
    vo3.y = 0;
    vo3.z = 0;
    VkExtent3D                              ve3;
    ve3.width = texWidth;
    ve3.height = texHeight;
    ve3.depth = 1;
    VkImageCopy vic;
    vic.srcSubresource = visl;
    vic.srcOffset = vo3;
    vic.dstSubresource = visl;
    vic.dstOffset = vo3;
    vic.extent = ve3;
    vkCmdCopyImage(TextureCommandBuffer,
                    stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
                    textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
                    IN &vic);
}

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

result = vkEndCommandBuffer(TextureCommandBuffer);
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &TextureCommandBuffer;
vsi.waitSemaphoreCount = 0;
vsi.pWaitSemaphores = (VkSemaphore *)nullptr;
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
result = vkQueueSubmit(Queue, 1, IN &vsi, VK_NULL_HANDLE);
result = vkQueueWaitIdle(Queue);
```
To 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, &vivci, PALLOCATOR, &pMyTexture->texImageView);
    return result;
```

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

---

### Reading in a Texture from a BMP File

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

Reading in a Texture from a BMP File

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

---

### What is the Vulkan Graphics Pipeline?

Here’s what you need to know:

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

The Graphics Pipeline Layout is 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, &vplci, nullptr, &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.

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

Vulkan: A 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)

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

Creating a Typical Graphics Pipeline

These settings seem pretty typical to me. Let's write a simplified Pipeline-creator that accepts Vertex and Fragment shader modules and the topology, and always uses the settings in red above.
Link in the Per-Vertex Attributes

```cpp
struct vertex
{
  float3 position;  // vertex position
  float3 normal;   // vertex normal
  float4 color;    // vertex color
  float2 texCoord; // texture coordinate
};
```

Use one `vviad` array member per shader module you are using.

Use one `vvibd` array member per vertex input array-of-structures you are using.

### Primitive Restart Enable

The `primitiveRestartEnable` flag is used with:
- Indexed drawing.
- Triangle Fan and "Strip" topologies.

If `primitiveRestartEnable` is `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.

- **Indexed drawing**: If `true`, the index before the special index is restarted.
- **Triangle Fan/Strip**: If `true`, the special index is `0xffffffff`.

```cpp
typedef enum VkIndexType
{
  VK_INDEX_TYPE_UINT = 0,
  VK_INDEX_TYPE_UINT16 = 2,
  VK_INDEX_TYPE_UINT32 = 3,
};
```

### Link in the Shaders

```cpp
// vviad[0] describes vertex attributes:
vkVertexInputAttributeDescription vviad[4] =
{
  .sType = VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION,
  .location = 0,                  // location in the layout
  .format = VK_FORMAT_VEC3;       // x, y, z
  .offset = offsetof( struct vertex, position );                  // 0
  .binding = 0;                  // 0
  .stride = sizeof( struct vertex );
  .inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
};
```

Use one `vpssci` array member per shader module you are using as vertex input.

These are defined at the top of the sample code so that you don't need to use confusing image-looking formats for positions, normals, and tex coordinates.

### What is "Primitive Restart Enable"?

- `VK_FALSE`
- `VK_TRUE`

```cpp
vkPipelineShaderStageCreateInfo vpssci[2];
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pName = "main";
vpssci[0].flags = 0;
vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
vpssci[0].module = vertexShader;
vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
vpssci[0].pNext = nullptr;
vpssci[0].pName = "main";
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
vpssci[1].pNext = nullptr;
```

Declare the vertex topology.

The `Topology` type is used to describe the input vertex attributes.

```cpp
typedef enum VkTopology
{
  VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
  VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
  VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
};
```

```
typedef enum VkIndexType
{
  VK_INDEX_TYPE_UINT = 0,
  VK_INDEX_TYPE_UINT16 = 2,
  VK_INDEX_TYPE_UINT32 = 3,
};
```

Declare the binding descriptions and attribute descriptions.

### Geometry Shader info

```
VkPipelineGeometryStateCreateInfo vpgsci =
{
  .sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO,
  .pNext = nullptr,
  .flags = 0,
  .primitiveRestartEnable = VK_FALSE;
};
```

```cpp
typedef enum VkIndexType
{
  VK_INDEX_TYPE_UINT = 0,
  VK_INDEX_TYPE_UINT16 = 2,
  VK_INDEX_TYPE_UINT32 = 3,
};
```

### Tessellation Shader info

```
struct vertex
{
  float3 position;  // vertex position
  float3 normal;   // vertex normal
  float4 color;    // vertex color
  float2 texCoord; // texture coordinate
};
```

```
typedef enum VkIndexType
{
  VK_INDEX_TYPE_UINT = 0,
  VK_INDEX_TYPE_UINT16 = 2,
  VK_INDEX_TYPE_UINT32 = 3,
};
```
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

Triangle Strip #0:
Triangle Strip #1:
Triangle Strip #2:
...

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

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

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

VkPipelineRasterizationStateCreateInfo vprsci;
vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
vprsci.pNext = nullptr;
vprsci.flags = 0;
vprsci.depthClampEnable = VK_FALSE;
vprsci.rasterizerDiscardEnable = VK_FALSE;
vprsci.polygonMode = VK_POLYGON_MODE_FILL; // choose from ...
#ifdef CHOICES
VK_POLYGON_MODE_LINE
VK_POLYGON_MODE_POINT#endif
vprsci.cullMode = VK_CULL_MODE_NONE; // recommend this because of the projMatrix[1,1] *= -1.;
#ifdef CHOICES
VK_CULL_MODE_NONE
VK_CULL_MODE_FRONT_BIT
VK_CULL_MODE_BACK_BIT
VK_CULL_MODE_FRONT_AND_BACK_BIT#endif
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE; // choose from ...
#ifdef CHOICES
VK_FRONT_FACE_COUNTER_CLOCKWISE
VK_FRONT_FACE_CLOCKWISE#endif
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;
vprsci.lineWidth = 1.f;
**What is “Depth Clamp Enable”?**

```cpp
vprsci.depthClampEnable = VK_FALSE;
```

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

A good use for this is Polygon Capping:

![Polygon Capping Example](image)

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

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

**What is “Depth Bias Enable”?**

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

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

**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.

```cpp
VkPipelineColorBlendAttachmentState vpcbas;
```

![Color Blending Diagram](image)

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

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

---

**Color Blending State for each Color Attachment**

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

```cpp
VkPipelineColorBlendStateCreateInfo vpcbsci;
```

![Color Blending Diagram](image)

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

```cpp
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
vpcbsci.pNext = nullptr;
vpcbsci.flags = 0;
vpcbsci.logicOpEnable = VK_FALSE;
vpcbsci.logicOp = VK_LOGIC_OP_COPY;
```

```cpp
#ifdef CHOICES
VK_LOGIC_OP_CLEAR
VK_LOGIC_OP_AND
VK_LOGIC_OP_AND_INVERTED
VK_LOGIC_OP_NO_OP
VK_LOGIC_OP_XOR
VK_LOGIC_OP_OR
VK_LOGIC_OP_NOR
VK_LOGIC_OP_EQUIVALENT
VK_LOGIC_OP_INVERT
VK_LOGIC_OP_OR_INVERTED
VK_LOGIC_OP_NAND
VK_LOGIC_OP_SET
#endif
```

```cpp
vpcbsci.attachmentCount = 1;
vpcbsci.pAttachments = &vpcbas;
vpcbsci.blendConstants[0] = 0;
vpcbsci.blendConstants[1] = 0;
vpcbsci.blendConstants[2] = 0;
vpcbsci.blendConstants[3] = 0;
```
Which Pipeline Variables can be Set Dynamically

```cpp
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport( )
VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor( )
VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth( )
VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias( )
VK_DYNAMIC_STATE_BLEND_CONSTANTS        -- vkCmdSetBendConstants( )
VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthZBounds( )
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompare Mask( )
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK     -- vkCmdSetStencilWriteM ask( )
VK_DYNAMIC_STATE_STENCIL_REFERENCE     -- vkCmdSetStencilReferen ces( )
#endif

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

Stencil Operations for Front and Back Faces

```cpp
VkStencilOpState vsosf;  // front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds

#ifdef CHOICES
VK_STENCIL_OP_KEEP -- keep the stencil value as it is
VK_STENCIL_OP_ZERO -- set stencil value to 0
VK_STENCIL_OP_REPLACE -- replace stencil value with the reference value
VK_STENCIL_OP_INCREMENT_AND_CLAMP -- increment stencil value
VK_STENCIL_OP_DECREMENT_AND_CLAMP -- decrement stencil value
VK_STENCIL_OP_INVERT -- bit-invert stencil value
VK_STENCIL_OP_INCREMENT_AND_WRAP -- increment stencil value
VK_STENCIL_OP_DECREMENT_AND_WRAP -- decrement stencil value
#endif
vsosf.compareOp = VK_COMPARE_OP_NEVER;       // never succeeds

#ifdef CHOICES
VK_COMPARE_OP_NEVER -- never succeeds
VK_COMPARE_OP_LESS -- succeeds if stencil value is < the reference value
VK_COMPARE_OP_EQUAL -- succeeds if stencil value is == the reference value
VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if stencil value is <= the reference value
VK_COMPARE_OP_GREATER -- succeeds if stencil value is > the reference value
VK_COMPARE_OP_NOT_EQUAL -- succeeds if stencil value is != the reference value
VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if stencil value is >= the reference value
VK_COMPARE_OP_ALWAYS -- always succeeds
#endif
vsosf.compareMask = ~0;
vsosf.writeMask = ~0;
vsosf.reference = 0;

VkStencilOpState vsosb;  // back
vsosb.depthFailOp = VK_STENCIL_OP_KEEP;       // never succeeds
vsosb.failOp = VK_STENCIL_OP_KEEP;
vsosb.passOp = VK_STENCIL_OP_KEEP;
vsosb.compareOp = VK_COMPARE_OP_NEVER;       // never succeeds
vsosb.compareMask = ~0;
vsosb.writeMask = ~0;
vsosb.reference = 0;
```

Uses for Stencil Operations

- Magic Lenses
- Polygon edges without Z-fighting

Operations for Depth Values

```cpp
VkPipelineDepthStencilStateCreateInfo vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;

#ifdef CHOICES
VK_COMPARE_OP_NEVER -- never succeeds
VK_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
VK_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
VK_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
VK_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
VK_COMPARE_OP_ALWAYS -- always succeeds
#endif
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.frontCompareOp = VK_COMPARE_OP_LESS;
vpdssci.backCompareOp = VK_COMPARE_OP_LESS;
vpdssci.frontDepthToBackOrder = VK_TRUE;
vpdssci.backDepthToBackOrder = VK_TRUE;
```
Putting it all Together! (finally…)

```cpp
vkGraphicsPipelineCreateInfo
vkpci

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

#ifdef CHOICESVK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BITVK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BITVK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

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

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vkpci, PALLOCATOR, OUT pGraphicsPipeline );
return result;
```

Group all of the individual state information and create the pipeline

Putting it all Together! (finally…)

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

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

Queues and Command Buffers

Mike Bailey
mjb@cs.oregonstate.edu

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

Simplified Block Diagram
Vulkan Queues and Command Buffers

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

Querying what Queue Families are Available

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

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

Similarly, we Can Write a Function that Finds the Proper Queue Family

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

Creating a Logical Device Needs to Know Queue Family Information

```cpp
VkDeviceCreateInfo vdci;
vkGetDeviceQueueCreateInfo(IN VkDeviceCreateInfo, OUT &vdci, O NULL, OUT &LogicalDevice);
```

Creating the Command Pool as part of the Logical Device

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

Creating the Command Buffers

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

Beginning a Command Buffer

```c
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore(LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore);
    uint32_t nextImageIndex;
    vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN UINT64_MAX, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

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

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

```c
vkCmdBeginQuery(commandBuffer, flags);
vkCmdBeginRenderPass(commandBuffer, const contents);
vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);
vkCmdBindIndexBuffer(commandBuffer, indexType);
vkCmdBindPipeline(commandBuffer, pipeline);
vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
vkCmdBlitImage(commandBuffer, filter);
vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
vkCmdClearColorImage(commandBuffer, pRanges);
vkCmdClearDepthStencilImage(commandBuffer, pRanges);
vkCmdCopyBuffer(commandBuffer, pRegions);
vkCmdCopyBufferToImage(commandBuffer, pRegions);
vkCmdCopyImage(commandBuffer, pRegions);
vkCmdCopyImageToBuffer(commandBuffer, pRegions);
vkCmdCopyQueryPoolResults(commandBuffer, flags);
vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
vkCmdDebugMarkerEndEXT(commandBuffer);
vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);
vkCmdDispatchIndirect(commandBuffer, offset);
vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
vkCmdDrawIndexedIndirect(commandBuffer, stride);
vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);
vkCmdDrawIndirect(commandBuffer, stride);
vkCmdDrawIndirectCountAMD(commandBuffer, stride);
vkCmdEndQuery(commandBuffer, query);
vkCmdEndRenderPass(commandBuffer);
vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);
```
These are the Commands that could be entered into the Command Buffer, II

```c
vkCmdBindBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
vkCmdBindImageBuffer(commandBuffer, srcImage, srcImageLayout, dependencyFlags, imageMemoryBarriers);
VkInheritanceInfo inInfo = { 0 }
vkCmdBeginCommandBuffer(commandBuffer, &inInfo);

VkProcessCommandsInfo cmdInfo = { 0 };
vkCmdProcessCommandsNVX(commandBuffer, &cmdInfo);

VkSemaphoreCreateInfo vsci = { 0 };
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

VkPipelineLayout layout = GraphicsPipelineLayout;

VkCommandBuffer commandBuffer = CommandBuffers[nextImageIndex];

vkCmdClearColorValue(colorValue, 0.0, 0.0, 0.0, 0.0);
vkCmdClearDepthStencilValue(depthValue, 0.0, 0.0);

VkClearValue vcv[2] = { colorValue, depthValue };
vkCmdClearRect(commandBuffer, 0, 1, &r2d);

VkRenderPassBeginInfo vrpbi = { 0 };
vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
rpbi.renderPass = RenderPass;
rpbi.framebuffer = Framebuffers[nextImageIndex];
rpbi.renderArea = r2d;

vkCmdBindVertexBuffers(commandBuffer, 0, 1, buffers, offsets);
const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdBeginRenderPass(commandBuffer, &vrpbi, VK_SUBPASS_CONTENTS_INLINE);
vkCmdEndRenderPass(commandBuffer);
vkEndCommandBuffer(commandBuffer);
```
VkFenceCreateInfo vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

vfci.pNext = nullptr;

vfci.flags = 0;

VkFence renderFence;

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

result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkQueue presentQueue;

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

// 0 = queueIndex

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 = &SemaphoreRenderFinished;

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

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

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);

VkPresentInfoKHR vpi;

vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

vpi.pNext = nullptr;

vpi.waitSemaphoreCount = 0;

vpi.pWaitSemaphores = (VkSemaphore *)nullptr;

vpi.swapchainCount = 1;

vpi.pSwapchains = &SwapChain;

vpi.pImageIndices = &nextImageIndex;

vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR(presentQueue, IN &vpi);

The Entire Submission / Wait / Display Process

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

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

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

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

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

The Swap Chain

Mike Bailey

mjb@cs.oregonstate.edu

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

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

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

VkExtent2D surfaceRes = vsc.currentExtent;

fprintf( FpDebug, "
vkGetPhysicalDeviceSurfaceCapabilitiesKHR:
" );

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

if( supported == VK_TRUE )
    fprintf( FpDebug, "** This Surface is supported by the Graphics Queue **
" );

uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceFormats );

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

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );

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

We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:
vkGetPhysicalDeviceSurfaceCapabilitiesKHR:

minImageCount = 2 ; maxImageCount = 8

minImageExtent = 1024 x 1024

maxImageExtent = 1024 x 1024

maxImageArrayLayers = 1

supportedTransforms = 0x0001

currentTransform = 0x0001

supportedCompositeAlpha = 0x0001

supportedUsageFlags = 0x009f

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

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

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

Creating a Swap Chain

VkSwapchainCreateInfoKHR vscci;
vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
vscci.pNext = nullptr; vscci.flags = 0;
vscci.surface = Surface; vscci.minImageCount = 2; // double buffering
vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
vscci.imageExtent.width = surfaceRes.width;
vscci.imageExtent.height = surfaceRes.height;
vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
vscci.imageArrayLayers = 1;
vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE; vscci.queueFamilyIndexCount = 0;
vscci.pQueueFamilyIndices = nullptr;
vscci.oldSwapchain = VK_NULL_HANDLE; vscci.clipped = VK_TRUE;
result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );

Creating a Swap Chain
Creating the Swap Chain Images and Image Views

uint32_t imageCount; // # of display buffers – 2? 3?
result = vkGetSwapchainImagesKHR(LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr);

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

PresentImageViews = new VkImageView[imageCount];
for( unsigned int i = 0; i < imageCount; i++ ){
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.subresourceRange.baseMipLevel = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;
    vivci.image = PresentImages[i];
    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
}

Rendering into the Swap Chain, I

VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
result = vkCreateSemaphore(LogicalDevice, &vsci, PALLOCATOR, OUT &imageReadySemaphore);

uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
result = vkAcquireNextImageKHR(LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex);

result = vkBeginCommandBuffer(CommandBuffers[nextImageIndex], IN &vcbbi);
result = vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
result = vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
result = vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
result = vkEndCommandBuffer(CommandBuffers[nextImageIndex]);

Rendering into the Swap Chain, II

VkFenceCreateInfo vfci;
result = vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

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

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

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

Rendering into the Swap Chain, III

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

VkPresentKHRKHR;
result = vkQueuePresentKHR(presentQueue, IN &vpi);
Physical Devices

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: a More Typical (and Simplified) Block Diagram

Vulkan: Identifying the Physical Devices

Querying the Number of Physical Devices

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

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

```
VkResult result = VK_SUCCESS;
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)&nullptr);
if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 ){
    fprintf( FpDebug, "Could not count the physical devices
" );
    return VK_SHOULD_EXIT;
}
fprintf(FpDebug, "
%d physical devices found.
", PhysicalDeviceCount);
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[PhysicalDeviceCount];
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices);
if( result != VK_SUCCESS ){
    fprintf( FpDebug, "Could not enumerate the %d physical devices
", PhysicalDeviceCount );
    return VK_SHOULD_EXIT;
}
int discreteSelect = -1;
int integratedSelect = -1;
for(unsigned int i = 0; i < PhysicalDeviceCount; i++)
{
VkPhysicalDeviceProperties vpdp;
vkGetPhysicalDeviceProperties(IN physicalDevices[i], OUT &vpdp);
if(result != VK_SUCCESS)
{
fprintf(FpDebug, "Could not get the physical device properties of device %d\n", i);
return VK_SHOULD_EXIT;
}
fprintf(FpDebug, "\nDevice %2d:\n", i);
fprintf(FpDebug, "API version: %d\n", vpdp.apiVersion);
fprintf(FpDebug, "Driver version: %d\n", vpdp.apiVersion);
fprintf(FpDebug, "Vendor ID: 0x%04x\n", vpdp.vendorID);
fprintf(FpDebug, "Device ID: 0x%04x\n", vpdp.deviceID);
fprintf(FpDebug, "Physical Device Type: %d = " , vpdp.deviceType);
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU)
fprintf(FpDebug, "(Discrete GPU)\n");
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU)
fprintf(FpDebug, "(Integrated GPU)\n");
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU)
fprintf(FpDebug, "(Virtual GPU)\n");
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU)
fprintf(FpDebug, "(CPU)\n");
fprintf(FpDebug, "Device Name: %s\n", vpdp.deviceName);
fprintf(FpDebug, "Pipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0]);
}
// need some logic here to decide which physical device to select:
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU)
discreteSelect = i;
if(vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU)
integratedSelect = i;
int which = -1;
if(discreteSelect >= 0)
{
which = discreteSelect;
PhysicalDevice = physicalDevices[which];
}
else if(integratedSelect >= 0)
{
which = integratedSelect;
PhysicalDevice = physicalDevices[which];
}
else
{
fprintf(FpDebug, "Could not select a Physical Device\n");
return VK_SHOULD_EXIT;
}

VkPhysicalDeviceProperties PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures(IN PhysicalDevice, OUT &PhysicalDeviceFeatures);
fprintf(FpDebug, \nPhysical Device Features:\n" geometryShader = %d\n" , PhysicalDeviceFeatures.geometryShader);
fprintf(FpDebug, \n"tessellationShader = %d\n" , PhysicalDeviceFeatures.tessellationShader);
fprintf(FpDebug, \n"multiDrawIndirect = %d\n" , PhysicalDeviceFeatures.multiDrawIndirect);
fprintf(FpDebug, \n"wideLines = %d\n" , PhysicalDeviceFeatures.wideLines);
fprintf(FpDebug, \n"largePoints = %d\n" , PhysicalDeviceFeatures.largePoints);
fprintf(FpDebug, \n"multiViewport = %d\n" , PhysicalDeviceFeatures.multiViewport);
fprintf(FpDebug, \n"occlusionQueryPrecise = %d\n" , PhysicalDeviceFeatures.occlusionQueryPrecise);
fprintf(FpDebug, \n"pipelineStatisticsQuery = %d\n" , PhysicalDeviceFeatures.pipelineStatisticsQuery);
fprintf(FpDebug, \n"shaderFloat64 = %d\n" , PhysicalDeviceFeatures.shaderFloat64);
fprintf(FpDebug, \n"shaderInt64 = %d\n" , PhysicalDeviceFeatures.shaderInt64);
fprintf(FpDebug, \n"shaderInt16 = %d\n" , PhysicalDeviceFeatures.shaderInt16);

vkEnumeratePhysicalDevices:
Device: 0:
API version: 4196499
Driver version: 4196499
Vendor ID: 0xe0de
Device ID: 0x1e04
Physical Device Type: 2 = (Discrete GPU)
Device Name: RTX 2080 Ti
Pipeline Cache Size: 206
Device #0 selected (RTX 2080 Ti)

Physical Device Features:
geometryShader = 1
tessellationShader = 1
multiDrawIndirect = 1
wideLines = 1
largePoints = 1
multiViewport = 1
occlusionQueryPrecise = 1
pipelineStatisticsQuery = 1
shaderFloat64 = 1
shaderInt64 = 1
shaderInt16 = 1
Here's What the Intel HD Graphics 520 Produced

vkEnumeratePhysicalDevices:
Device 0:
  API version: 4194360
  Driver version: 4194360
  Vendor ID: 0x8086
  Device ID: 0x1916
  Physical Device Type: 1 = (Integrated GPU)
  Device Name: Intel(R) HD Graphics 520
Pipeline Cache Size: 213
Device #0 selected ('Intel(R) HD Graphics 520')

Physical Device Features:
  geometryShader = 1
  tessellationShader = 1
  multiDrawIndirect = 1
  wideLines = 1
  multiViewport = 1
  occlusionQueryPrecise = 1
  pipelineStatisticsQuery = 1
  shaderFloat64 = 1
  shaderInt64 = 1
  shaderInt16 = 1

Here's What I Got

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

2 Memory Heaps:
Heap 0: size = 0x67c00000 DeviceLocal
Heap 1: size = 0x6ca00000

Asking About the Physical Device's Different Memories

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

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

Asking About the Physical Device's Queue Families

uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr);
if( count > 0 ) {
    VkQueueFamilyProperties *vqf = new VkQueueFamilyProperties[count];
    vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT vqf);
    for( unsigned int i = 0; i < count; i++ ){
        fprintf(FpDebug, "Queue Family %d:\n", i);
        fprintf(FpDebug, "queueCount = %d\n", vqf[i].queueCount );
        if ( ( vqf[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
            fprintf(FpDebug, "Graphics\n" );
        if ( ( vqf[i].queueFlags & VK_QUEUE_COMPUTE_BIT ) != 0 )
            fprintf(FpDebug, "Compute\n" );
        if ( ( vqf[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )
            fprintf(FpDebug, "Transfer\n" );
    }
}
Here’s What I Got

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

Logical Devices

Here’s What I Got

Here’s What I Got

Here’s What I Got

Here’s What I Got

Looking to See What Device Layers are Available

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

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

// see what device layers are available:
uint32_t layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);
VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
result = vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, deviceLayers);
Looking to See What Device Extensions are Available

```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);
```

Looking to See What Device Extensions are Available

What Device Layers and Extensions are Available

```c
4 physical device layers enumerated:

0x00401063   1  'VK_LAYER_NV_optimus'  'NVIDIA Optimus layer'
0 device extensions enumerated for 'VK_LAYER_NV_optimus':

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

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

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

Vulkan: Creating a Logical Device

```c
float   queuePriorities[1] = {1.0f};
VkDeviceQueueCreateInfo vdqci;
vdqci.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vkQueueFlags = 0;
vkQueueFamilyIndex = 0;
vkQueueCount = 1;
vkQueueProperties = queuePriorities;
result = vkCreateDeviceQueue( LogicalDevice, 0, 0, OUT &Queue );
```

Vulkan: Creating a Logical Device

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.

Vulkan: Creating a Graphics Pipeline

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

- VK_DYNAMIC_STATE_VIEWPORT
- VK_DYNAMIC_STATE_SCISSOR
- VK_DYNAMIC_STATE_LINE_WIDTH
- VK_DYNAMIC_STATE_DEPTH_BIAS
- VK_DYNAMIC_STATE_BLEND_CONSTANTS
- VK_DYNAMIC_STATE_DEPTH_BOUNDS
- VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK
- VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
- VK_DYNAMIC_STATE_STENCIL_REFERENCE
Creating a Pipeline

VkDynamicState
{
VK_DYNAMIC_STATE_VIEWPORT,
VK_DYNAMIC_STATE_LINE_WIDTH,
};

VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState);// i.e., 2
vpdsci.pDynamicStates = &vds;

VkGraphicsPipelineCreateInfo vgpci;
vgpci.pDynamicState = &vpdsci;

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

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

Filling the Dynamic State Variables in the Command Buffer

The command buffer-bound function calls to set these dynamic states are:
vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);
vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);
vkCmdSetLineWidth(commandBuffer, linewidth);
vkCmdSetDepthBias(commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor);
vkCmdSetBlendConstants(commandBuffer, blendConstants[4]);
vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);
vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
vkCmdSetStencilReference(commandBuffer, faceMask, reference);

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.

http://cs.oregonstate.edu/~mjb/vulkan
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 matrix{
    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 or’ed bits of `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`, `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`, etc.
- `size` is in bytes
- `pValues` is a void * pointer to the data, which in this 4x4 matrix example, would be of type `glm::mat4`.

Setting up the Push Constants for the Pipeline Structure

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

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

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

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

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;
    glm::vec3 armColor;
    float armScale;  // scale factor in x
};
```

struct arm Arm1;
struct arm Arm2;
struct arm Arm3;
In the Reset Function

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

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

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

Setup the Push Constant for the Pipeline Structure

VkPushConstantRange vpcr[1];

vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

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

VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;vplci.flags = 0;

vplci.setLayoutCount = 4;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.pushConstantRangeCount = 1;
vplci.pPushConstantRanges = &vpcr[0];

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

In the UpdateScene Function

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

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

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

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

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

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

In the RenderScene Function

VkBuffer buffers[1] = { MyVertexBuffer.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

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

layout( location = 0 ) in vec3 aVertex;

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

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

Getting Information Back from the Graphics System

Mike Bailey
mjb@cs.oregonstate.edu

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

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

Setting up Query Pools

VkQueryPoolCreateInfo vqpci;
vqpci.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
vqpci.pNext = nullptr;
vqpci.flags = 0;
vqpci.queryType = << one of: >>
    VK_QUERY_TYPE_OCCLUSION
    VK_QUERY_TYPE_PIPELINE_STATISTICS
    VK_QUERY_TYPE_TIMESTAMP
vqpci.queryCount = 1;
vqpci.pipelineStatistics = 0; // bitmask of what stats you are querying for if you are doing a pipeline statistics query

VkQueryPool occlusionQueryPool;
result = vkCreateQueryPool(LogicalDevice, IN &vqpci, PALLOCATOR, OUT &occlusionQueryPool);

VkQueryPool statisticsQueryPool;
result = vkCreateQueryPool(LogicalDevice, IN &vqpci, PALLOCATOR, OUT &statisticsQueryPool);

VkQueryPool timestampQueryPool;
result = vkCreateQueryPool(LogicalDevice, IN &vqpci, PALLOCATOR, OUT &timestampQueryPool);
vkCmdResetQueryPool(CommandBuffer, occlusionQueryPool, 0, 1);
vkCmdBeginQuery(CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT);
...  
vkCmdEndQuery(CommandBuffer, occlusionQueryPool, 0);
#define DATASIZE 128  
uint32_t data[DATASIZE];
result = vkGetQueryPoolResults(LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags);

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).

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

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

Occlusion Query

Pipeline Statistics Query

Occlusion Query

Timestamp Query
Timestamp Query

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

```cpp
void vkCmdWriteTimeStamp(CommandBuffer, pipelineStages, timestampQueryPool, 0);
```

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT`
- `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`
- `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`
- `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT`
- `VK_PIPELINE_STAGE_TRANSFER_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_HOST_BIT`

**Compute Shaders**

Mike Bailey
mjb@cs.oregonstate.edu

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

Here is how you create a Compute Pipeline

Start by Creating the Data Buffers

This is a Particle System application, so we need Positions, Velocities, and (possibly) Colours.

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

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

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

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

VkBuffer Buffer;

VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = << buffer size in bytes >>;
vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const int32_t []) nullptr;

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

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
<< 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 );
    // 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

VkDescriptorSetComputeSetLayoutCreateInfo vdsclci;
vdsclci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdsclci.pNext = nullptr;
vdsclci.flags = 0;
vdsclci.bindingCount = 1;
vdsclci.pBindings = ComputeSet[0];

result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdsclci, PALLOCATOR, OUT &ComputeSetLayout );

VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 1;
vplci.pSetLayouts = ComputeSetLayout;

vplci.pushConstantRangeCount = 0;
vplci.pPushConstantRanges = (VkPushConstantRange *) nullptr;

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

The number of work-items per work-group is set in the compute shader.
The number of work-groups is set in the vkCmdDispatch( ) function call in the C/C++ program.

The Particle System Compute Shader -- The Physics

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

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

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

VELOCITY
BounceSphere( POINT p, VELOCITY v, SPHERE s )
{ VECTOR n = normalize( p - s.xyz );
  return Bounce( v, n );
}

bool
IsInsideSphere( POINT p, SPHERE s )
{ float r = length( p - s.xyz );
  return ( r < s.w );
}
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;
if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + .5*DT*DT*G;
}
Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;

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

Dispatching the Compute Shader from the Command Buffer

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 );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 ); // Buffer holds the 3 sizes, offset=0

Specialization Constants

http://cs.oregonstate.edu/~mjb/vulkan
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 a halfway-compiled version of a shader right before the rest-of-the-way compilation.

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

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

Why Do We Need Specialization Constants?

Specialization Constants could be used for:

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

Specialization Constant Example -- Setting an Array Size

In the compute shader

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

In the Vulkan C/C++ program:

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

Linking the Specialization Constants into the Compute Pipeline

```plaintext
result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, OUT ComputePipeline);
```
Specialization Constant Example – Setting Multiple Constants

In the compute shader:
```cpp
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:
```cpp
struct abc { int a, int b, float c; } abc;
VkSpecializationMapEntry vsme[3];
vsme[0].constantID = 9;
sme[0].offset = offsetof( abc, a );
vsme[0].size = sizeof(abc.a);
vsme[1].constantID = 10;
vsme[1].offset = offsetof( abc, b );
vsme[1].size = sizeof(abc.b);
vsme[2].constantID = 11;
vsme[2].offset = offsetof( abc, c );
vsme[2].size = sizeof(abc.c);
VkSpecializationInfo vsi;
vsii.mapEntryCount = 3;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(abc); // size of all the Specialization Constants together
vsii.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:
```cpp
layout( local_size_x = 32, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:
```cpp
int numXworkitems = 64;
VkSpecializationMapEntry vsme[1];
vsme[0].constantID = 12;
vsme[0].offset = 0;
vsme[0].size = sizeof(int);
VkSpecializationInfo vsi;
vsi.mapEntryCount = 1;
vsi.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(int);
vsii.pData = &numXworkitems;
```

Where Synchronization Fits in the Overall Block Diagram

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
• 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
VkSemaphore imageReadySemaphore;
VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;

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

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];

vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore) nullptr;

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

Fences

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

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

Could be an array of events
Where signaled, where wait for the signal

mjc – November 19, 2019

Pipeline Barriers

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
**Potential Memory Race Conditions that Pipeline Barriers can Prevent**

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

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

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

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

---

**vkCmdPipelineBarrier() Function Call**

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

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

**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 its intersection, the proper dst traffic light will be turned red.
5. When the last car in the src group completely makes it through its intersection, the proper dst traffic light can be turned back to green.
6. The Vulkan command pipeline ordering is this: (1) the src cars get released, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released (which end up being stopped by a red light somewhere).
Pipeline Stage Masks –
Where in the Pipeline is this Memory Data being Generated or Consumed?

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

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

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

Pipeline Stages and what Access Operations can Happen There
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.

**Access Operations and what Pipeline Stages they can be used In**

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

**Access types**

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

**Stages**

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

**The Scenario**

- src cars are generating the image
- dist cars are doing something with that image
The Scenario

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

- VK_IMAGE_LAYOUT_UNDEFINED
- VK_IMAGE_LAYOUT_GENERAL
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
- VK_IMAGE_LAYOUT_PREINITIALIZED
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
- VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR

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

Antialiasing and Multisampling

Mike Bailey
mjb@cs.oregonstate.edu

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

The Display We Want

Too often, the Display We Get
**Aliasing**

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

What the signal really is: what we want

Sampling Interval

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

Sampled Points

---

**MultiSampling**

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

There are two approaches 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]

   One pixel

   One pixel

   8x8 samples

   4x4 samples

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]

   Multisampling Count 1-bit
   Multisampling Count 2-bit
   Multisampling Count 4-bit

---

**Vulkan Distribution of Sampling Points within a Pixel**

![Vulkan Sampling Diagram]
Vulkan Distribution of Sampling Points within a Pixel

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.171, 0.125)</td>
<td>(0.3125, 0.3125)</td>
<td>(0.5625, 0.5625)</td>
<td>(0.875, 0.875)</td>
</tr>
<tr>
<td>(0.125, 0.25)</td>
<td>(0.25, 0.625)</td>
<td>(0.5, 0.9375)</td>
<td>(0.875, 0.9375)</td>
</tr>
<tr>
<td>(0.625, 0.625)</td>
<td>(0.9375, 0.9375)</td>
<td>(0.3125, 0.625)</td>
<td>(0.875, 0.625)</td>
</tr>
<tr>
<td>(0.75, 0.75)</td>
<td>(0.875, 0.875)</td>
<td>(0.0625, 0.3125)</td>
<td>(0.875, 0.4375)</td>
</tr>
</tbody>
</table>

Consider Two Triangles Whose Edges Pass Through the Same Pixel

Supersampling

Final Pixel Color = \[ \frac{7}{8} \text{ Color sample from subpixel} \]

# Fragment Shader calls = 8

Multisampling

Final Pixel Color = \[ \frac{3 \times \text{ One color sample from A} + 5 \times \text{ One color sample from B}}{8} \]

# Fragment Shader calls = 2
vkPipelineMultisampleStateCreateInfo
vpmsci;
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vmsci.flags = 0;
vmsci.rasterizationSamples = VK_SAMPLE_COUNT_8_BIT;
vmsci.minSampleShading = 1.0f;
vmsci.sampleShadingEnable = VK_TRUE;
vmsci.maxSampleShading = 1.0f;
vmsci.pSampleMask = VK_SAMPLE_MASK_8_BIT;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;

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

At least this fraction of samples will get their own fragment shader calls (as long as they pass the depth and stencil tests).
0. produces simple multisampling
(0.1...) produces partial supersampling
1. produces complete supersampling
Resolving the Image:
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

```cpp
VOffset3D
vo3 = {
  x = 0,
  y = 0,
  z = 0,
};

VkExtent3D
ve3 = {
  width = Width;
  height = Height;
  depth = 1;
};

VkImageSubresourceLayers
visl = {
  aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
  mipLevel = 0;
  baseArrayLayer = 0;
  layerCount = 1;
};

VkImageResolve
vir = {
  srcSubresource = visl;
  srcOffset = vo3;
  dstSubresource = visl;
  dstOffset = vo3;
  extent = ve3;
};

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