Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

Course Goals

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

Sections

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

My Favorite Vulkan Reference


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

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

Second, thanks to NVIDIA for all of their support!

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

Ali Alsalehy
Natasha Anisimova
Jianchang Bi
Christopher Cooper
Richard Connell
Braden Cusens
Bergamini-Fastie
Trevor Hamrick
Zach Leiva
Victor Li

Second, thanks to NVIDIA for all of their support!

Ali Alsalehy
Natasha Anisimova
Jianchang Bi
Christopher Cooper
Richard Connell
Braden Cusens
Bergamini-Fastie
Trevor Hamrick
Zach Leiva
Victor Li

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

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

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

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

2016: Vulkan 1.1

2020: Vulkan 1.2

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

Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

Vulkan is better at keeping the GPU busy than OpenGL is. OpenGL drivers need to do a lot of CPU work before handing work off to the GPU. Vulkan lets you get more power from the GPU card you already have.

This is especially important if you can hide the complexity of Vulkan from your customer base and just let them see the improved performance. Thus, Vulkan has had a lot of support and interest from game engine developers, 3rd party software vendors, etc.

As an aside, the Vulkan development effort was originally called “glNext”, which created the false impression that this was a replacement for OpenGL. It’s not...
Why is it so important to keep the GPU Busy?

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


Who was the original Vulcan?

Who is the Khronos Group?

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

Who’s Been Specifically Working on Vulkan?
Vulkan

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

Vulkan Differences from OpenGL

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

The Basic Computer Graphics Pipeline, OpenGL-style

Model Transform

View Transform

Projection Transform

Rasterization

Per-vertex in variables

Uniform Variables

Output color(s)

Per-fragment in variables

Uniform Variables

Per-vertex out variables

Uniform Variables

Vertex, Normal, Color

MC = Model Vertex Coordinates

WC = World Vertex Coordinates

EC = Eye Vertex Coordinates

The Basic Computer Graphics Pipeline, Shader-style

Vertex Shader

Fragment Shader

Uniform Variables

Per-vertex in variables

Uniform Variables

Per-vertex out variables

Uniform Variables

Per-fragment in variables

Uniform Variables

Per-fragment out variables

Uniform Variables

The Basic Computer Graphics Pipeline, Vulkan-style

Per-vertex in variables

Uniform Variables

Per-vertex out variables

Uniform Variables

Per-fragment in variables

Uniform Variables

Per-fragment out variables

Uniform Variables

Moving part of the driver into the application

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

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

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

Vulkan Highlights: Pipeline State Objects

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

Vulkan Quick Reference Card – I Recommend you Print This!

Vulkan Quick Reference Card


Vulkan Highlights: Overall Block Diagram

- Application
  - Instance
    - Physical Device
      - Logical Device
        - Command Buffer

Vulkan Highlights: a More Typical Block Diagram

- Application
  - Instance
    - Physical Device
      - Logical Device
        - Command Buffer

Steps in Creating Graphics using Vulkan

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

Vulkan GPU Memory

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

Vulkan Render Passes

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

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

Vulkan Synchronization

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

Vulkan Shaders

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

---

The Vulkan Sample Code Included with These Notes

Mike Bailey
mjb@cs.oregonstate.edu

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

Sample Program Output

The "19" refers to the version of Visual Studio, not the year of development.
Sample Program Keyboard Inputs

T, L: Toggle lighting off and on
V, M: Toggle display mode (textures vs. colors, for now)
Y, P: Pause the animation
Y, Q: quit the program
Esc: quit the program
Y, R: Toggle rotation-animation and using the mouse
Y, Y: Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)
Y, 4, 9: Set the number of instances (in the instancing version)

Caveats on the Sample Code, I

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

Caveats on the Sample Code, II

6. There are great uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
7. I’ve typedef’ed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I’ve divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invert 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( );
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents( );
        Time = glfwGetTime( );          // elapsed time, in double-precision seconds
        UpdateScene( );
        RenderScene( );
    }
    fprintf(FpDebug, "Closing the GLFW window
");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```

InitGraphics( ), I

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

InitGraphics( ), II

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

A Colored Cube

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. }
    }
    ...
};

#include "SampleVertexData.cpp"

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

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

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

Vulkan Software Philosophy

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

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

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

we would actually say in C:

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

Vulkan Conventions

VKXXX is a typedef, probably a struct
vkYYY() is a function call
VK_ZZZ is a constant

My Conventions

“Init” in a function call name means that something is being setup that only needs to be setup once
The number after “Init” gives you the ordering
In the source code, after main() comes InitGraphics(), then all of the 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 actually #define'd to nothing.
Querying the Number of Something and Allocating Enough Structures to Hold Them All

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

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

Where to put them
How many total there are

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

Reporting Error Results, I

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

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

Extras in the Code

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

bool Paused;
bool Verbose;

#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
const int32_t OFFSET_ZERO = 0;

Vulkan

Drawing

Mike Bailey
mjb@cs.oregonstate.edu
Vulkan Topologies

VK_PRIMITIVE_TOPOLOGY_POINT_LIST
V0 V1 V2 V3

VK_PRIMITIVE_TOPOLOGY_LINE_LIST
V0 V1 V2 V3

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
V0 V1 V2 V3

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

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

A Colored Cube Example

Triangles Represented as an Array of Structures

Non-indexed Buffer Drawing

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

**A Preview of What init05DataBuffer Does**

**C/C++:**

always use the C/C++ sizeof construct if/else rather than hardcoding the value.

```c
void vkCreatePipeline( LogicalDevice, VK_NULL_HANDLE, 1, IN &vpssci, out &vpvsci, out MyBuffer *) {
    result = vkCreatePipeline( LogicalDevice, IN &vpssci, out &vpvsci, out MyBuffer *); // 0 is the offset
}
```

**GLSL Shader:**

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

**Telling the Pipeline about its Input**

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

**Telling the Command Buffer what Vertices to Draw**

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

```c
result = createGraphicsPipeline( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, out &vgpci, out MyBuffer *) {
    result = createGraphicsPipeline( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, out &vgpci, out MyBuffer *) {
        result = vkCreatePipeline( LogicalDevice, IN &vpssci, out &vpvsci, out MyBuffer *); // 0 is the offset
    }
```
Drawing with an Index Buffer

Stream of Vertices

Stream of Indices

Verteck Lookup

Triangles

Draw

void vkCmdDrawIndexedIndirect(VkCommandBuffer commandBuffer, uint32_t offset, uint32_t count, VkDeviceSize stride);

typedef struct
{
    uint32_t indexCount;
    uint32_t instanceCount;
    uint32_t firstIndex;
    int32_t vertexOffset;
    uint32_t firstInstance;
} VkDrawIndexedIndirectCommand;

Compare this with:

void vkCmdDrawIndexed(VkCommandBuffer commandBuffer, uint32_t count, uint32_t instanceCount, uint32_t firstIndex, uint32_t vertexOffset, uint32_t firstInstance);

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

void vkCmdDrawIndexedIndirect(VkCommandBuffer commandBuffer, uint32_t offset, uint32_t drawCount, VkDeviceSize stride);

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

Compare this with:

void vkCmdDrawIndirect(VkCommandBuffer commandBuffer, uint32_t drawCount, VkDeviceSize stride);

Indirect Drawing (not to be confused with Indexed)

void vkCmdDrawIndirect(VkCommandBuffer commandBuffer, uint32_t offset, uint32_t drawCount, VkDeviceSize stride);

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

Compare this with:

void vkCmdDraw(VkCommandBuffer commandBuffer, uint32_t drawCount, VkDeviceSize stride);

Triangles Draw

Drawing with an Index Buffer

mjb – July 24, 2020

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

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

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

vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets);

vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType);

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

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

VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
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 );

vkCmdDrawIndirect( CommandBuffers[nextImageIndex], buffer, offset, drawCount, stride);
Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

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

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

### The OBJ File Format – a triple-indexed way of Drawing

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

V / T / N

### Shaders and SPIR-V

Mike Bailey
mjb@cs.oregonstate.edu

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

### The Shaders’ View of the Basic Computer Graphics Pipeline

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

### Vulkan Shader Stages

Shader stages

```cpp
//Shader stages
Vulkan Shader Stages

Isolated spine's VkPipelineStageFlags in:
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
```
How Vulkan GLSL Differs from OpenGL GLSL

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

Vulkan Vertex and Instance indices:

<table>
<thead>
<tr>
<th>gl_VertexIndex</th>
<th>gl_InstanceIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both are 0-based</td>
<td></td>
</tr>
</tbody>
</table>

• Both are 0-based

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

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

OpenGL uses:

| gl_VertexID | gl_InstanceID |
|
|-----------|-------------|
| Both are 0-based |

Shader combinations of separate texture data and samplers:

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

Descriptor Sets:

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

Specialization Constants:

```glsl
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:

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

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

Push Constants:

```glsl
layout( push_constant ) ...
```

Note: our sample code doesn’t use this.

Vulkan: Shaders’ use of Layouts for Uniform Variables

```glsl
vkCreateShaderModule( )
VkShaderModuleCreateInfo( )
device

code[ ] (u_int32_t) codeSize (in bytes)
shaderModuleCreateFlags
```

// non-sampler variables must be in a uniform block:

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

// non-sampler variables must be in a uniform block:

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

layout( set = 2, binding = 0 ) uniform sampler2D uTexUnit;

All non-sampler uniform variables must be in block buffers

Vulkan Shader Compiling

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

External GLSL Compiler in driver Vendor specific code

GLSL Source -> SPIR-V -> Compiler in driver Vendor specific code

Advantages:

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

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

<table>
<thead>
<tr>
<th>Shadder file extensions:</th>
<th>-V Compile for Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>-G Compile for OpenGL</td>
<td></td>
</tr>
<tr>
<td>-I Directory(e)s to look in for Includes</td>
<td></td>
</tr>
<tr>
<td>-S Specify stage rather than get it from shaderFile extension</td>
<td></td>
</tr>
<tr>
<td>-c Print out the maximum sizes of various properties</td>
<td></td>
</tr>
</tbody>
</table>

Windows: glslangValidator.exe
Linux: glslangValidator

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

1. Click on the Microsoft Start icon
2. Type the word bash

This is only available within 64-bit Windows 10.
You Can Run the SPIR-V Compiler on Windows from a Bash Shell

This is only available within 64-bit Windows 10.

Pick one:

- Can get to your personal folders
- Does not have make
- Can get to your personal folders
- Does have make

This is only available within 64-bit Windows 10.

Running glslangValidator.exe

Running glslangValidator.exe

Running glslangValidator.exe

Reading a SPIR-V File into a Vulkan Shader Module

```
#define SPIRV_MAGIC             0x07230203
```
For example, if this is your Shader Source

```glsl
void main() {
    layout ( location = 2 ) out vec2 vTexCoord;
    layout ( location = 0 ) out vec3 vNormal;
    layout( location = 3 ) in vec2 aTexCoord;
    layout( location = 1 ) in vec3 aNormal;
    layout( std140, set = 1, binding = 0 ) uniform lightBuf Matrices;
    layout( std140, set = 0, binding = 0 ) uniform matBuf
}
```

You can also take a look at SPIR-V Assembly

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

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

```
This is the SPIR-V Assembly, Part I
```

```
This is the SPIR-V Assembly, Part II
```

```
This is the SPIR-V Assembly, Part III
```
1. You can `#include` files into your shader source. There are several really nice features. The two I really like are:

   a. `glslc` is included in your Sample .zip file.
   b. Improved command-line interface. You use, basically, the same way:

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

   *Note:* The shaderc project from Google provides a `glslangValidator` wrapper program called `glslc` that has a much improved command-line interface. You use, basically, the same way:

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

   General Constant Matrix Vector Indexing
   General Variable Indexing
   General Sampler Indexing
   General Varying Indexing
   General Attribute Matrix Vector Indexing
   General Uniform Indexing
   Do-While Loops
   While Loops
   Non-Inductive For Loops
   Max Samples 4
   Max Combined Clip And Cull Distances 8
   Max Transform Feedback Interleaved Components 64
   Max Transform Feedback Buffers 4
   Max Atomic Counter Buffer Size 16384
   Max Combined Atomic Counter Buffers 1
   Max Fragment Atomic Counter Buffers 1
   Max Geometry Atomic Counter Buffers 0
   Max Tess Evaluation Atomic Counter Buffers 0
   Max Tess Control Atomic Counter Buffers 0
   Max Vertex Atomic Counter Buffers 0
   Max Atomic Counter Bindings 1
   Max Combined Atomic Counters 8
   Max Fragment Atomic Counters 8
   Max Geometry Atomic Counters 0
   Max Tess Evaluation Atomic Counters 0
   Max Tess Control Atomic Counters 0
   Max Viewports 16
   Max Tess Gen Level 64
   Max Patch Vertices 32
   Max Tess Patch Components 120
   Max Tess Evaluation Uniform Components 1024
   Max Tess Evaluation Texture Image Units 16
   Max Tess Evaluation Output Components 128
   Max Tess Evaluation Input Components 128
   Max Tess Control Total Output Components 4096
   Max Tess Control Uniform Components 1024
   Max Tess Control Texture Image Units 16
   Max Tess Control Output Components 128
   Max Tess Control Input Components 128
   Max Geometry Varying Components 64
   Max Geometry Uniform Components 1024
   Max Geometry Total Output Components 1024
   Max Geometry Output Vertices 256
   Max Geometry Texture Image Units 16
   Max Combined Image Uniforms 8
   Max Combined Image Units And Fragment Outputs 8
   Max Image Units 8
   Max Fragment Input Components 128
   Max Geometry Output Components 128
   Max Geometry Input Components 64
   Max Vertex Output Components 64
   Max Varying Components 60
   Max Compute Atomic Counter Buffers 1
   Max Compute Atomic Counters 8
   Max Compute Image Uniforms 8
   Max Compute Texture Image Units 16
   Max Compute Uniform Components 1024
   Max Compute Work Group Size Z 64
   Max Compute Work Group Size Y 1024
   Max Compute Work Group Size X 1024
   Max Compute Work Group Count Z 65535
   Max Compute Work Group Count Y 65535
   Max Compute Work Group Count X 65535
   Max Clip Distances 8
   Min Program Texel Offset -8
   Max Program Texel Offset 7
   Max Fragment Input Vectors 15
   Max Vertex Output Vectors 16
   Max Fragment Uniform Vectors 16
   Max Varying Vectors 8
   Max Vertex Uniform Vectors 128
   Max Draw Buffers 32
   Max Fragment Uniform Components 4096
   Max Texture Image Units 32
   Max Combined Texture Image Units 80
   Max Vertex Texture Image Units 32
   Max Varying Floats 64
   Max Vertex Uniform Components 4096
   Max Vertexattribs 64
   Max Texture Coords 32
   Max Texture Units 32
   Max Clip Planes 6
   Max Lights 32

   A Vulkan Data Buffer is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a "Binary Large Object," or "BLOB").

   It is up to you to be sure that the writer and the reader of the Data Buffer are interpreting the bytes in the same way!

   Vulkan calls these things "Buffers," but, Vulkan calls other things "Buffers," too, such as Texture Buffers and Command Buffers. So, I sometimes have taken to calling these things "Data Buffers" and have even gone so far as to override some of Vulkan's own terminology:

   ```
   typedef VkBuffer VkDataBuffer;
   ```

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

Creating a Vulkan Data Buffer

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

Finding the Right Type of Memory

Finding the Right Type of Memory
Sidebar: The Vulkan Memory Allocator (VMA)

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

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

This repository includes a smattering of documentation.

```c
#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"

VkBufferCreateInfo vbci;

VmaAllocationCreateInfo vaci;

vaci.physicalDevice = PhysicalDevice;

ewaci.device = LogicalDevice;

vaci.usage = VMA_MEMORY_USAGE_GPU_ONLY;

VmaAllocator var;

vmaCreateAllocator(IN &vaci, OUT &var);

VkBuffer Buffer;

VmaAllocation van;

vmaCreateBuffer(IN var, IN &vbci, IN &vaci, OUT &Buffer, OUT &van, nullptr);

void *mappedDataAddr;

vmaMapMemory(IN var, IN van, OUT &mappedDataAddr);

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

vmaUnmapMemory(IN var, IN van);
```

Something I’ve Found Useful

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

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

```c
MyBuffer MyMatrixUniformBuffer;
```

It’s the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need. It also makes it impossible to accidentally associate the wrong VkDeviceMemory and/or VkDeviceSize with the wrong data buffer.

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;
    return result;
}
```

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

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

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

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

Filling those Uniform Variables

```c
uint32_t                        Height, Width;
const double FOV =              glm::radians(60.);      // field-of-view angle in radians
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4( 1. );              // identity
Matrices.uViewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.; // account for Vulkan’s LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

This code assumes that this line:

```
#define GLM_FORCE_RADIANS
```

is listed before GLM is included!
This C struct is holding the original data, written by the application.

MyBuffer MyMatrixUniformBuffer;

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

MyBuffer MyMatrixUniformBuffer;

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

MyBuffer MyMatrixUniformBuffer;

\begin{verbatim}
  uniform matBuf Matrices;
\end{verbatim}

Filling the Data Buffer

\begin{verbatim}
Init05UniformBuffer( sizeof(Matrices), OUT &MyMatrixUniformBuffer);
Fill05DataBuffer( MyMatrixUniformBuffer, IN (void *) &Matrices);
\end{verbatim}

Creating and Filling the Data Buffer – the Details

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

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
Remember – to Vulkan and GPU memory, these are just bits. It is up to you to handle their meaning correctly.
\end{verbatim}

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

GLFW

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Setting Up GLFW

```
#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"
...
uint32_t Width, Height;
VkSurfaceKHR Surface;
...
void InitGLFW(void){
  glfwInit();
  if( !glfwVulkanSupported() ){
    fprintf(stderr, "Vulkan is not supported on this system!
");
    exit(1);
  }
  glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
  glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
  MainWindow = glfwCreateWindow(Width, Height, "Vulkan Sample", NULL, NULL);
  VkResult result = glfwCreateWindowSurface(Instance, MainWindow, NULL, OUT &Surface);
  glfwSetErrorCallback(GLFWErrorCallback);
  glfwSetKeyCallback(MainWindow, GLFWKeyboard);
  glfwSetCursorPosCallback(MainWindow, GLFWMouseMotion);
  glfwSetMouseButtonCallback(MainWindow, GLFWMouseButton);
}
```

You Can Also Query What Vulkan Extensions GLFW Requires

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

GLFW Keyboard Callback

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

GLFW Mouse Button Callback

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

GLFW Mouse Motion Callback

```
void GLFWMouseMotion(GLFWwindow *window, double xpos, double ypos ){
  int dx = (int)xpos - Xmouse;            // change in mouse coords
  int dy = (int)ypos - Ymouse;            // change in mouse coords
  if( dx > 0 )
    Xrot += ( ANGFACT*dy );
  if( dx < 0 )
    Yrot += ( ANGFACT*dx );
  if( dy > 0 )
    Scale -= SCLFACT * (float) ( dx - dy );
  if( dy < 0 )
    Scale += SCLFACT * (float) ( dx - dy );
  if( Scale < MINSCALE )
    Scale = MINSCALE;
  Xmouse = (int)xpos;                     // new current position
  Ymouse = (int)ypos;
}
```
while( glfwWindowShouldClose( MainWindow ) == 0 )
{
glfwPollEvents();

Time = glfwGetTime();          // elapsed time, in double-precision seconds

UpdateScene();

RenderScene();
}

vkQueueWaitIdle( Queue );
vkDeviceWaitIdle( LogicalDevice );

DestroyAllVulkan();

glfwDestroyWindow( MainWindow );

glfwTerminate();

Does not block – processes any waiting events, then returns

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

glfwWaitEvents();

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

glfwWaitEventsTimeout( double secs );

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

glfwPostEmptyEvent();

GLM

Mike Bailey
mjb@cs.oregonstate.edu

What is GLM?

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

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

You can find it at:

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

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

I recommend this so that all angles you create in all programming will be in radians.

Why are we even talking about this?

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

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );
glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glRotatef( (GLfloat)Xrot, 1., 0., 0. );
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );

you would now say:

glm::mat4 modelview = glm::mat4( 1. ); // identity
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
modelview = glm::lookAt( eye, look, up );

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

The Most Useful GLM Variables, Operations, and Functions

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

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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

// constructor:

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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:

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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

// constructor:

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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

// constructor:

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

// multiplications:

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

// emulating OpenGL transformations with concatenation:

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

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
The Most Useful GLM Variables, Operations, and Functions

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

// viewing (assign, not concatenate):
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );

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.

Installing GLM into your own space

Here's what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

1.
2.

Telling Visual Studio about where the GLM folder is

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

GLM in the Vulkan sample.cpp Program

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

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

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

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

\[
\begin{align*}
    x' &= Ax + By + Cz + D \\
y' &= Ex + Fy + Gz + H \\
z' &= Ix + Jy + Kz + L
\end{align*}
\]

Let's use this matrix to answer a few questions:

### How Does this Matrix Stuff Really Work?

**Question:** Our rotation matrices only work around the origin? What if we want to rotate about an arbitrary point \((A, B)\)?

**Answer:** We create more than one matrix.

**Write it**

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

**Say it**

### How It Really Works :-)

\[
\begin{bmatrix}
    \cos 90^\circ & \sin 90^\circ \\
    -\sin 90^\circ & \cos 90^\circ
\end{bmatrix}
\]

http://xkcd.com

### Transformation Matrices

**Translation**

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

**Rotation about X**

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

**Rotation about Y**

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    0 & 1 & 0 & 1 \\
    -1 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0
\end{bmatrix}
\]

**Rotation about Z**

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

For this to be correct, \(A\) must be a unit vector.

### Compound Transformations

**Q:** Our rotation matrices only work around the origin? What if we want to rotate about an arbitrary point \((A, B)\)?

**A:** We create more than one matrix.

**Write it**

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

**Say it**

### Matrix Multiplication is not Commutative

- **Rotate, then translate**
- **Translate, then rotate**
Matrix Multiplication is Associative

\[
\begin{pmatrix}
  x' \\
  y' \\
  z'
\end{pmatrix} =
\begin{bmatrix}
  T_{x,a,b} & R_a & T_{x,a,b} \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]

One matrix -- the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

\[
\begin{pmatrix}
  x' \\
  y' \\
  z'
\end{pmatrix} =
\begin{bmatrix}
  T_{x,a,b} & R_a & T_{x,a,b} \\
  0 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]

Why Isn't The Normal Matrix exactly the same as the Model Matrix?

Wrong!

Right!

Instancing – What and why?

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

Making each Instance look differently -- Approach #1

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

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

Instancing

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Put the unique characteristics in a uniform buffer array 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; // or "& 1023" – gives 0 - 1023
vColor = Colors.uColors[ index ];
```

```
vec4 vertex = ...;
```

```
gl_Position = PVM * vertex; // [p]*[v]*[m]
```

The Graphics Pipeline Data Structure

Mike Bailey
mjb@cs.oregonstate.edu

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

The First Step: Create the Graphics Pipeline Layout

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

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

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

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex output attributes: stencil, depth
- Assembly topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissors: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Blending: blendEnable, stencilFront, stencilBack, stencilOp
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

`#define VK_FORMAT_VEC4          VK_FORMAT_R32G32B32A32_SFLOAT`

// these are here for convenience and readability:
`#endif`

`#define VK_FORMAT_X             VK_FORMAT_R32_SFLOAT`

`#define VK_FORMAT_FLOAT         VK_FORMAT_R32_SFLOAT`

`#define VK_FORMAT_XY            VK_FORMAT_R32G32_SFLOAT`

`#define VK_FORMAT_VEC2          VK_FORMAT_R32G32_SFLOAT`

`#define VK_FORMAT_STP           VK_FORMAT_R32G32B32_SFLOAT`

`#define VK_FORMAT_VEC3          VK_FORMAT_R32G32B32_SFLOAT`

`vviad[0].format = VK_FORMAT_VEC3;       // x, y, z`
`vviad[0].binding = 0;                   // which binding this is part of`
`vviad[0].location = 0;                  // location in the layout`

4 = vertex, normal, color, texture coord

`vviad[3].offset = offsetof(struct vertex, texCoord);               // 36`
`vviad[3].format = VK_FORMAT_VEC2;       // s, t`
`vviad[3].location = 3;`

`vviad[2].format = VK_FORMAT_VEC3;       // r, g, b`
`vviad[2].binding = 0;`

`vviad[1].binding = 0;`

`vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;`

Creating a Typical Graphics Pipeline

Link in the Per-Vertex Attributes

Creating a Graphics Pipeline from a lot of Pieces

The Shaders to Use

A Pipeline Data Structure Contains the Following State Items:

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex output attributes: stencil, depth
- Assembly topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissors: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Blending: blendEnable, stencilFront, stencilBack, stencilOp
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

`#define VK_FORMAT_VEC4          VK_FORMAT_R32G32B32A32_SFLOAT`

// these are here for convenience and readability:
`#endif`

`#define VK_FORMAT_X             VK_FORMAT_R32_SFLOAT`

`#define VK_FORMAT_FLOAT         VK_FORMAT_R32_SFLOAT`

`#define VK_FORMAT_XY            VK_FORMAT_R32G32_SFLOAT`

`#define VK_FORMAT_VEC2          VK_FORMAT_R32G32_SFLOAT`

`#define VK_FORMAT_STP           VK_FORMAT_R32G2B32_SFLOAT`

`#define VK_FORMAT_VEC3          VK_FORMAT_R32G3B32_SFLOAT`

`vviad[0].format = VK_FORMAT_VEC3;       // x, y, z`
`vviad[0].binding = 0;                   // which binding this is part of`
`vviad[0].location = 0;                  // location in the layout`

4 = vertex, normal, color, texture coord

`vviad[3].offset = offsetof(struct vertex, texCoord);               // 36`
`vviad[3].format = VK_FORMAT_VEC2;       // s, t`
`vviad[3].location = 3;`

`vviad[2].format = VK_FORMAT_VEC3;       // r, g, b`
`vviad[2].binding = 0;`

`vviad[1].binding = 0;`

`vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;`
Options for vpiasci.topology

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

What is “Primitive Restart Enable”?

vpiasci.primitiveRestartEnable = VK_FALSE;

“What is “Primitive Restart Enable”?

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

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

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

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

If your VkIndexType is VK_INDEX_TYPE_UINT32, it is
0xffffffff.

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

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

Original Image

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

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.

Setting the Rasterizer State

What is the Difference Between Changing the Viewport and Changing the Scissoring?
What is “Depth Clamp Enable”?

vprsci.depthClampEnable = VK_FALSE;

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

A good use for this is Polygon Capping:

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

What is “Depth Bias Enable”?

vprsci.depthBiasEnable = VK_FALSE;

vprsci.depthBiasConstantFactor = 0.f;

vprsci.depthBiasSlopeFactor = 0.f;

vprsci.depthBiasEnable = VK_FALSE;

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

MultiSampling State

VkPipelineMultisampleStateCreateInfo

vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;

vpmsci.pNext = nullptr;

vpmsci.flags = 0;

vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;

vpmsci.sampleShadingEnable = VK_FALSE;

vpmsci.minSampleShading = 0;

vpmsci.pSampleMask = (VkSampleMask *)nullptr;

vpmsci.alphaToCoverageEnable = VK_FALSE;

vpmsci.alphaToOneEnable = VK_FALSE;

Declare information about how the multisampling will take place.

We will discuss MultiSampling in a separate noteset.

Color Blending State for each Color Attachment

VkPipelineColorBlendAttachmentState

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;

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

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

Which Pipeline Variables can be Set Dynamically

// Just used as an example in the Sample Code

VkDynamicState

vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };

#ifdef CHOICES

VK_DYNAMIC_STATE_VIEWPORT -- vkCmdSetViewport( )

VK_DYNAMIC_STATE_SCISSOR -- vkCmdSetScissor( )

VK_DYNAMIC_STATE_LINE_WIDTH -- vkCmdSetLineWidth( )

VK_DYNAMIC_STATE_DEPTH_BIAS -- vkCmdSetDepthBias( )

VK_DYNAMIC_STATE_BLEND_CONSTANTS -- vkCmdSetBendConstants( )

VK_DYNAMIC_STATE_DEPTH_BOUNDS -- vkCmdSetDepthZBounds( )

VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK -- vkCmdSetStencilCompareMask( )

VK_DYNAMIC_STATE_STENCIL_WRITE_MASK -- vkCmdSetStencilWriteMask( )

VK_DYNAMIC_STATE_STENCIL_REFERENCE -- vkCmdSetStencilReferences( )

#endif

VkPipelineDynamicStateCreateInfo

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;

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

© 2023 SIGGRAPH
The Stencil Buffer

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

Using the Stencil Buffer to Create a Magic Lens

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

Using the Stencil Buffer to Perform Polygon Capping

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

Outlining Polygons the Naive Way

1. Draw the polygons
2. Draw the edges

Z-fighting
Using the Stencil Buffer to Better Outline Polygons

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

Using the Stencil Buffer to Perform Hidden Line Removal

Stencil Operations for Front and Back Faces

Operations for Depth Values

Putting it all Together! (finally…)

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Fixed-layout Pipeline Blocks</th>
<th>Variable-layout Pipeline Blocks</th>
</tr>
</thead>
</table>

In OpenGL

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

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

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

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

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

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

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

What are Descriptor Sets?

Descriptor Sets are an intermediate data structure that tells shaders how to connect information held in GPU memory to groups of related uniform variables and texture sampler declarations in shaders.
I think of Descriptor Set Layouts as a kind of “Rosetta Stone” that allows the GPU to understand what the shader expects. Once you define the layout, you can also specify a set of immutables that will be shared between all shaders that use that layout. Then you have what we call a “pool” of descriptors that you allocate once and use with each shader that needs it. The descriptors can then be bound to the shader on the fly, which is very fast.

Step 1: Descriptor Set Pools

You don't allocate Descriptor Sets on the fly – that is too slow. Instead, you allocate a “pool” of Descriptor Sets and then pull from that pool later.
Step 4: Allocating the Memory for Descriptor Sets

- `VkDescriptorSetAllocateInfo`:
  - `vdsai.descriptorSetCount`: Number of descriptor sets to allocate.
  - `vdsai.pNext = nullptr`: No additional data.
  - `vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`: Type of descriptor set allocate info.

- For each descriptor set layout:
  - `vdslc0/vdslc1/vdslc2/vdslc3`:
    - `vdslc.x.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO`: Type of descriptor set layout create info.
    - `vdslc.x.pBindings`: Array of descriptor bindings.
    - `vdslc.x.bindingCount`: Number of bindings.
    - `vdslc.x.flags = 0`: Flags set to 0.
    - `vdslc.x.pNext = nullptr`: No additional data.

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

- `VkWriteDescriptorSet`:
  - `vwds0`:
    - `vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`: Type of write descriptor set.
    - `vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`: Type of descriptor.
    - `vwds0.dstArrayElement = 0`: Array element.
    - `vwds0.dstBinding = 0`: Binding.
    - `vwds0.dstSet = DescriptorSets[0]`: Descriptor set.
    - `vwds0.pBufferInfo = (VkBufferInfo *)nullptr`: Buffer info.
    - `vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr`: Image info.
    - `vwds0.pTexelBufferView = (VkBufferView *)nullptr`: Texel buffer view.
    - `vwds0.pNext = nullptr`: No additional data.
  - `vwds1`:
    - Same as `vwds0`.
    - `vwds2`:
    - `vwds3`:

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

- `VkPipelineLayoutCreateInfo`:
  - `vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO`: Type of pipeline layout create info.
  - `vplci.setLayoutCount = 4`: Number of layouts.
  - `vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr`: No push constant ranges.

- `vkCreatePipelineLayout`:
  - `vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);`: Create the pipeline layout.
  - `result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, OUT &GraphicsPipelineLayout);`: Return the result.

Step 5: Tell the Descriptor Sets where their data is

- vkCmdBindDescriptorSets

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

- vkCreateDescriptorPooling
- vkCreateDescriptorSetLayout
- vkCreatePipelineLayout
- vkAllocateDescriptorSets
- vkCmdBindDescriptorSets

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

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

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

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

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

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

The mapping between the geometry of the 3D object and the S and T of the texture image works like this:

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that the rasterizer is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when it goes to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.

Triangles in an Array of Structures

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

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

\[
s = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \quad t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}}
\]
Or, for a sphere,

\[
s = \frac{\theta - (-\pi)}{2\pi}, \quad t = \frac{\phi - (-\pi/2)}{\pi}.
\]

 Uh-oh. Now what? Here's where it gets tougher....

\[
s = \frac{\theta - (-\pi)}{2\pi}, \quad t = \frac{\phi - (-\pi/2)}{\pi}.
\]

You really are at the mercy of whoever did the modeling...

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

**GPU Memory**

- Host Visible
- GPU Memory
- Device Local
- GPU Memory

**Memory Types**

**NVIDIA Discrete Graphics:**

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

```gl
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

Vulkan:

```vk
vkSamplerCreateInfo vsci;
```

- Graphics hardware determines which level to use based on the texels : pixels ratio.
- Total texture storage is ~ 2x what it was without mip-mapping.
- In addition to just picking one mip-map level, the rendering system can sample from two of them, one less than the T.P ratio and one more, and then blend the two RGBA’s returned. This is known as vk_sampler_remap_mode.linear.

Textures’ Undersampling Artifacts

As an object gets farther away and covers a smaller and smaller part of the screen, the texels : pixels ratio used in the coverage becomes larger and larger. This means that there are pieces of the texture leftover in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all white, with the red texel having never been included in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets included somehow in all resolution-level textures.

- 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 than the T.P ratio and one more, and then blend the two RGBA’s returned. This is known as VK_SAMPLER_REMAP_MODE_LINEAR.

* Latin: multum in parvo, “many things in a small place”
vkUnmapMemory

if (vsI.rowPitch == 4 * texWidth)

// transition the staging buffer layout:
result =

VkCommandBufferBeginInfo vcbbi;

// copy pixels from the staging image to the texture:
VkImageCopy vic;

for (unsigned int y = 0; y < texHeight; y++)

unsigned char *gpuBytes = (unsigned char *)gpuMemory;

memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

vkCmdPipelineBarrier

stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,

vic.extent = ve3;

vic.dstSubresource = visl;

vic.srcOffset = vo3;

vkBeginCommandBuffer

TextureCommandBuffer,

IN &vcbbi);

ve3.width = texWidth;

vo3.z = 0;

vo3.x = 0;

visl.layerCount = 1;

visl.mipLevel = 0;

visl.baseArrayLayer = 0;

visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vkCmdPipelineBarrier

TextureCommandBuffer,

IN &vcbbi);

visr.layerCount = 1;

visr.baseArrayLayer = 0;

visr.levelCount = 1;

visr.baseMipLevel = 0;

visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vkCmdPipelineBarrier

TextureCommandBuffer,

IN &vcbbi);

vkBindImageMemory

vkAllocateMemory

vkCreateImage

vkImageMemoryBarrier

vkImageCopy

ffi

fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment) ;

fprintf( FpDebug, "Texture vmr.size = %lld\n", vmr.size);

vkBindImageMemory

vkAllocateMemory

vkCreateImage

vkImageMemoryBarrier

vkImageCopy

ffi

fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment) ;

fprintf( FpDebug, "Texture vmr.size = %lld\n", vmr.size);
// create an image view for the texture image:
// (an "image view" is used to indirectly access an image)
VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

VkImageViewCreateInfo vivci;
vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
vivci.pNext = nullptr;
vivci.flags = 0;
vivci.image = textureImage;
vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
vivci.components.r = VK_COMPONENT_SWIZZLE_R;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.subresourceRange = visr;

result = vkCreateImageView(LogicalDevice, &vivci, PALLOCATOR, &pMyTexture->texImageView);
return result;

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

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

Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Command Buffer

Vulkan Queues and Command Buffers

- Graphics commands are recorded in command buffers, e.g., vkCmdDispatch(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

CPU Thread

CPU Thread

CPU Thread

CPU Thread
# Querying what Queue Families are Available

```cpp
for( unsigned int i = 0; i < count; i++ )
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];

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

uint32_t count = -1;

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

# Similarly, We Can Write a Function That Finds the Proper Queue Family

```cpp
int FindQueueFamilyThatDoesGraphics( )
{
    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
// allocating command pool and logical device

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

vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );
```

# Creating the Command Pool as part of the Logical Device

```cpp
// creating command pool

VkCommandBufferAllocateInfo vcbbi;
vcbbi.commandBufferCount = 2; // 2, because of double-buffering
vcbbi.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
vcbbi.pNext = nullptr;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
```

# Creating the Command Buffers

```cpp
// allocating command buffers for the double-buffered rendering:

VkCommandBufferAllocateInfo vcbbai;
vcbbai.commandBufferCount = 2; // 2, because of double-buffering
vcbbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
vcbbai.pNext = nullptr;
vcbbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );
```

# Beginning a Command Buffer – One per Image

```cpp
// creating command buffer

VkCommandBufferAllocateInfo vcbbci;
vcbbci.commandBufferCount = 1;
vcbbci.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
result = vkAllocateCommandBuffers( CommandBuffers[nextImageIndex], IN &vcbai, OUT &CommandBuffer[nextImageIndex] );
```
These are the Commands that could be entered into the Command Buffer, I

- vkAllocateCommandBuffer
- vkCmdPushConstants
- vkCmdProcessCommandsNVX
- vkCmdFillBuffer
- vkCmdPushDescriptorSetKHR
- vkCmdPushDescriptorSetWithTemplateKHR
- vkCmdWriteTimestamp
- vkCmdWaitEvents
- vkCmdUpdateBuffer
- vkCmdSetViewport
- vkCmdSetStencilWriteMask
- vkCmdSetStencilReference
- vkCmdSetScissor
- vkCmdSetLineWidth
- vkCmdSetDiscardRectangleEXT
- vkCmdSetDeviceMaskKHX
- vkCmdResetEvent
- vkCmdReserveSpaceForCommandsNVX
- vkCmdSetBlendConstants
- vkCmdResolveImage

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

- VkRenderPassBeginInfo
- VkRect2D
- VkOffset2D
- VkClearDepthStencilValue
- VkClearColorValue

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

- VkCommandBufferBeginInfo
- VkCommandBufferAllocateInfo
- VkCommandBufferBeginInfo

Beginning a Command Buffer

- Begin a Command Buffer
- End a Command Buffer
Submitting a Command Buffer to a Queue for Execution

VkSubmitInfo vsi;
  vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  vsi.pNext = nullptr;
  vsi.commandBufferCount = 1;
  vsi.pCommandBuffers = &CommandBuffer;
  vsi.waitSemaphoreCount = 1;
  vsi.pWaitSemaphores = imageReadySemaphore;
  vsi.signalSemaphoreCount = 0;
  vsi.pSignalSemaphores = (VkSemaphore *)nullptr;

Submitting a Command Buffer to a Queue for Execution

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);

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.

How OpenGL Thinks of Framebuffers

How Vulkan Thinks of Framebuffers – the Swap Chain

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. The is called the Swap Chain.

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

Swap Chains are arranged as a ring buffer.

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

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

We Need to Find Out What our Display Capabilities Are

what is a swap chain?

VulkanDebug.txt output:

- Found 2 Surface Formats:
  - This Surface is supported by the Graphics Queue

vkGetPhysicalDeviceSurfaceCapabilitiesKHR:

- Found 3 Present Modes:
  1: 50 (VK_FORMAT_B8G8R8A8_SRGB, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR)
  1: 3 (VK_PRESENT_MODE_FIFO_RELAXED_KHR)
  0: 2 (VK_PRESENT_MODE_FIFO_KHR)

Present Images = new VkImage[imageCount] =

for( unsigned int i = 0; i < imageCount; i++ )
{
    
    // present views for the double-buffering:
    Vivci.image = PresentImages[i];
    Vivci.subresourceRange.layerCount = 1;
    Vivci.subresourceRange.baseArrayLayer = 0;
    Vivci.subresourceRange.levelCount = 1;
    Vivci.subresourceRange.baseMipLevel = 0;
    Vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    Vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    Vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    Vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    Vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    Vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    Vivci.pNext = nullptr;
    Vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
}

PresentViews = new VkImageView[imageCount] =

for( unsigned int i = 0; i < imageCount; i++ )
{
    
    VkImageViewCreateInfo vivci;
    vsc.currentExtent;
    vkCreateSwapchainKHR(LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain);
    vkGetPhysicalDeviceSurfaceCapabilitiesKHR(LogicalDevice, Surface, OUT &vsc);
    VkSurfaceCapabilitiesKHR vsc;
    vkGetPhysicalDeviceSurfacePresentModesKHR(LogicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *)nullptr);
    uint32_t presentModeCount;
    vkGetPhysicalDeviceSurfaceFormatsKHR(LogicalDevice, Surface, &formatCount, (VkSurfaceFormatKHR *)nullptr);
    uint32_t formatCount;
    vkGetDevicePhysicalSurfaceCapabilitiesKHR(LogicalDevice, Surface, OUT &vsc);
    VkSurfaceCapabilitiesKHR vsc;
    result = vkGetPhysicalDeviceSurfaceSupportKHR(LogicalDevice, FindQueueFamilyThatDoesGraphics(), Surface, &supported);
    VkBool32 supported;
    fprintf(FpDebug, "** This Surface is supported by the Graphics Queue **\n");
    PresentImages = new VkImage[imageCount];
    PresentImageViews = new VkImageView[imageCount];
    PresentViews = new VkImageView[imageCount];
    for( unsigned int i = 0; i < imageCount; i++ )
    {
        
        Vivci.image = PresentImages[i];
        Vivci.subresourceRange.layerCount = 1;
        Vivci.subresourceRange.baseArrayLayer = 0;
        Vivci.subresourceRange.levelCount = 1;
        Vivci.subresourceRange.baseMipLevel = 0;
        Vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        Vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        Vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        Vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        Vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
        Vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        Vivci.pNext = nullptr;
        Vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
    }
    vkGetSwapchainImagesKHR(LogicalDevice, SwapChain, OUT &images);
    imageCount = images.imageCount;
    for( unsigned int i = 0; i < imageCount; i++ )
    {
        
        Vivci.image = images.images[i];
        Vivci.subresourceRange.layerCount = 1;
        Vivci.subresourceRange.baseArrayLayer = 0;
        Vivci.subresourceRange.levelCount = 1;
        Vivci.subresourceRange.baseMipLevel = 0;
        Vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        Vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        Vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        Vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        Vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
        Vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        Vivci.pNext = nullptr;
        Vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
    }
    vkGetSwapchainImagesKHR(LogicalDevice, SwapChain, OUT &images);
    imageCount = images.imageCount;
    for( unsigned int i = 0; i < imageCount; i++ )
    {
        
        Vivci.image = images.images[i];
        Vivci.subresourceRange.layerCount = 1;
        Vivci.subresourceRange.baseArrayLayer = 0;
        Vivci.subresourceRange.levelCount = 1;
        Vivci.subresourceRange.baseMipLevel = 0;
        Vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        Vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        Vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        Vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        Vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
        Vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        Vivci.pNext = nullptr;
        Vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[i]);
    }
}
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.
On the shader side, if, for example, you are sending a 4x4 matrix, the use of push constants in the shader looks like this:

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

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

```c
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`.

### An Robotic Example using Push Constants

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

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

Where each arm is represented by:

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

### Forward Kinematics:

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

Hook the Pieces Together, Change Parameters, and Things Move (All Young Children Understand This)
Positioning Part #1 With Respect to Ground

1. Rotate by $\Theta_1$
2. Translate by $T_{G1}$

Write it

$$M_{1/G} = [T_{G1}] \cdot [R_{\Theta_1}]$$

Say it

Why Do We Say it Right-to-Left?

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

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [M_{1G}] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = [T_{G1}] \cdot [R_{\Theta_1}][M_{1/G}] \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

So the right-most transformation in the sequence multiplies the (x,y,z,1) first and the left-most transformation multiplies it last.

Positioning Part #2 With Respect to Ground

1. Rotate by $\Theta_2$
2. Translate the length of part 1
3. Rotate by $\Theta_1$
4. Translate by $T_{G1}$

Write it

$$M_{2/G} = [T_{G1}][R_{\Theta_1}][T_{21}][R_{\Theta_2}]$$

Say it

Positioning Part #3 With Respect to Ground

1. Rotate by $\Theta_3$
2. Translate the length of part 2
3. Rotate by $\Theta_2$
4. Translate the length of part 1
5. Rotate by $\Theta_1$
6. Translate by $T_{G1}$

Write it

$$M_{3/G} = [T_{G1}][R_{\Theta_1}][T_{21}][R_{\Theta_2}][T_{S2}][R_{\Theta_3}]$$

Say it

In the Reset Function

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

Setup the Push Constant for the Pipeline Structure

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

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

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

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

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

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

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

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

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

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

In the RenderScene Function

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

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

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

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

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

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

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

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

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

In the Vertex Shader

```cpp
layout(push_constant) uniform arm
{

mat4 armMatrix;
vec3 armColor;
float armScale; // scale factor in x
} RobotArm;

layout(location = 0) in vec3 aVertex;

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

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

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

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

Physical Devices

Mike Bailey
mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
Vulkan: a More Typical (and Simplified) Block Diagram

Application

Instance

Physical Device

Logical Device

Command Buffer

Command Buffer

Command Buffer

Vulkan: Identifying the Physical Devices

Vulkan: Querying the Number of Physical Devices

Vulkan: Which Physical Device to Use, I

Vulkan: Which Physical Device to Use, II

Vulkan: Asking About the Physical Device’s Features

// need some logical here to decide which physical device to select:
if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
    discreteSelect = i;
else if( integratedSelect >= 0 )
    discreteSelect = integratedSelect;
else if( integratedSelect >= 0 )
	if( discreteSelect >= 0 )
		int which = -1;
	else
		PhysicalDevice = physicalDevices[which];
else
		PhysicalDevice = physicalDevices[which];

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

VkResult result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices);

VkPhysicalDeviceProperties PhysicalDeviceFeatures;

vkGetPhysicalDeviceFeatures( IN physicalDevices[i], OUT &PhysicalDeviceFeatures );
Here’s What the NVIDIA RTX 2080 Ti Produced

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

Here’s What the Intel HD Graphics 520 Produced

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

Asking About the Physical Device’s Different Memories

```
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp);
fprintf( FpDebug, "\n%d Memory Types:\n", vpdmp.memoryTypeCount );
for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
{
    VkMemoryType vmt = vpdmp.memoryTypes[i];
    fprintf( FpDebug, "Memory %2d: ", i );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT ) != 0 )    fprintf( FpDebug, " DeviceLocal" );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )    fprintf( FpDebug, " HostVisible" );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT ) != 0 )    fprintf( FpDebug, " HostCoherent" );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT ) != 0 )        fprintf( FpDebug, " HostCached" );
    if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT ) != 0 )    fprintf( FpDebug, " LazilyAllocated" );
    fprintf(FpDebug, "\n");
}
fprintf( FpDebug, "\n%d Memory Heaps:\n", vpdmp.memoryHeapCount );
for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ )
{
    fprintf(FpDebug, "Heap %d: ", i);
    VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
    fprintf( FpDebug, " size = 0x%08lx", (unsigned long int)vmh.size ) ;
    if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT ) != 0 )     fprintf( FpDebug, " DeviceLocal" );     // only one in use
    fprintf(FpDebug, "\n");
}
```

Asking About the Physical Device’s Queue Families

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

Asking About the Physical Device’s Queue Families

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

Here’s What I Got

```
11 Memory Types:
Memory 0: HostVisible
Memory 1: HostVisible
Memory 2: HostVisible
Memory 3: HostVisible
Memory 4: HostVisible
Memory 5: HostVisible
Memory 6: HostVisible
Memory 7: HostVisible
Memory 8: HostVisible
Memory 9: HostVisible
Memory 10: HostVisible
2 Memory Heaps:
Heap 0: size = 0x00000000 DeviceLocal
Heap 1: size = 0x00000000
```

Here’s What I Got

```
Found 3 Queue Families:
  queueCount = 10 : Graphics Compute Transfer
  queueCount = 2 : Transfer
  queueCount = 1 : Compute
```
Logical Devices

Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: Overall Block Diagram

Application
Instance
Physical Device
Logical Device
Command Buffer

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",

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

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

What Device Layers and Extensions are Available

4 physical device layers enumerated:

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

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

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

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

```cpp
VkDeviceCreateInfo vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdcis.queueCreateInfoCount = 1;  // # of device queues
vdci.pQueueCreateInfos = &queueCreateInfo;
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdcis.enabledLayerCount = 0;
vdcis.ppEnabledLayerNames = myDeviceLayers;
vdcis.enabledExtensionCount = 0;
vdcis.ppEnabledExtensionNames = (const char **)nullptr;  // no extensions
vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdcis.ppEnabledExtensionNames = myDeviceExtensions;
vdcis.pEnabledFeatures = &PhysicalDeviceFeatures;

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

Vulkan: Creating the Logical Device's Queue

```cpp
float queuePriorities[1] = {1.0};
VkDeviceQueueCreateInfo vdqci;
vdcis.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
vdcis.pNext = nullptr;
vdcis.flags = 0;
vdcis.queueFamilyIndex = 0;
vdcis.queueCount = 1;
vdcis.pQueueProperties = queuePriorities;
```

Creating a Pipeline with Dynamically Changeable State Variables

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

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

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

Creating a Pipeline

```cpp
vkCreateGraphicsPipeline();
```

Which Pipeline State Variables can be Changed Dynamically

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

- `VK_DYNAMIC_STATE_VIEWPORT`
- `VK_DYNAMIC_STATE_SCISSOR`
- `VK_DYNAMIC_STATE_LINE_WIDTH`
- `VK_DYNAMIC_STATE_DEPTH_BIAS`
- `VK_DYNAMIC_STATE_BLEND_CONSTANTS`
- `VK_DYNAMIC_STATE_DEPTH_BOUNDS`
- `VK_DYNAMIC_STATE_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
- sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO
- pNext = nullptr
- flags = 0
- dynamicStateCount = sizeof(vds) / sizeof(VkDynamicState)
- pDynamicStates = &vds

VkGraphicsPipelineCreateInfo
- pDynamicState = &vpdsci

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

First call:
```
vkCmdBindPipeline( … );
```

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

Getting Information Back from the Graphics System

Mike Bailey
mjb@cs.oregonstate.edu

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

Setting up Query Pools

- 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

Resetting, Filling, and Examining a Query Pool

- Define DATASIZE
- Define data array
- Call vkGetQueryPoolResults
- OR’ed combinations of
  - VK_QUERY_RESULT_64_BIT
  - VK_QUERY_RESULT_WAIT_BIT
  - VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
  - VK_QUERY_RESULT_PARTIAL_BIT
  - stride is # of bytes in between each result
Occlusion Queries count the number of fragments drawn between the `vkCmdBeginQuery` and the `vkCmdEndQuery` that pass both the Depth and Stencil tests. This is commonly used to see what level of detail should be used when drawing a complicated object.

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

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

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

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

```c
// vqpci.pipelineStatistics = or'd bits of:
// VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT
// VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT
// VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT
// VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT
// VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT
```

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

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

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

```c
vkCmdWriteTimeStamp( CommandBuffer, pipelineStages, timestampQueryPool, 0 );
```

The Rendering Draws the Particles by Reading the Position and Color Buffers. The Example We Are Going to Use Here is a Particle System.

**Compute Shaders**

Mike Bailey

mjb@cs.oregonstate.edu

http://cs.oregonstate.edu/~mjb/vulkan
#define NUM_PARTICLES (1024*1024) // total number of particles to move
#define NUM_WORK_ITEMS_PER_GROUP 64 // # work-items per work-group
#define NUM_X_WORK_GROUPS (NUM_PARTICLES/NUM_WORK_ITEMS_PER_GROUP)

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

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

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

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

The Data in your C/C++ Program will look like This
This is a Particle System application, so we need Positions, Velocities, and (possibly) Colors

The Data in your Compute Shader will look like This

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.

The Data in your Graphics Pipeline will look like This

Here is how you create a Compute Pipeline Data Structure

A Reminder about Data Buffers

Creating a Shader Storage Buffer
result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout);

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

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

vkGetBufferMemoryRequirements(LogicalDevice, MyBuffer, OUT &vmr);

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

result = vkBindBufferMemory(LogicalDevice, MyBuffer, IN vdm, 0);

result = vkCreatePipelineShaderStageCreateInfo(LogicalDevice, IN &vpssci, VK_SHADER_STAGE_COMPUTE_BIT);

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

ComputePipelineLayout ComputeSetLayout &ComputePipeline

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

ComputeSetLayout ComputeSet[3]

ComputeSet[0].binding = 0;
ComputeSet[0].stage = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[0].flags = 0;
ComputeSet[0].pNext = nullptr;
ComputeSet[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
ComputeSet[0].basePipelineIndex = 0;
ComputeSet[0].basePipelineHandle = VK_NULL_HANDLE;
ComputeSet[0].layout = VK_NULL_HANDLE;

ComputeSet[1].binding = 1;
ComputeSet[1].stage = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[1].flags = 0;
ComputeSet[1].pNext = nullptr;
ComputeSet[1].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
ComputeSet[1].basePipelineIndex = 1;
ComputeSet[1].basePipelineHandle = VK_NULL_HANDLE;
ComputeSet[1].layout = VK_NULL_HANDLE;

ComputeSet[2].binding = 2;
ComputeSet[2].stage = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[2].flags = 0;
ComputeSet[2].pNext = nullptr;
ComputeSet[2].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
ComputeSet[2].basePipelineIndex = 2;
ComputeSet[2].basePipelineHandle = VK_NULL_HANDLE;
ComputeSet[2].layout = VK_NULL_HANDLE;

vkBindDescriptorSets(LogicalDevice, ComputePipeline, ComputeSetLayout, 1, &ComputeSet[0], 0, nullptr);

vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &PosBuffer);

vkGetBufferMemoryRequirements(LogicalDevice, PosBuffer, OUT &vmr);

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

result = vkBindBufferMemory(LogicalDevice, PosBuffer, IN vdm, 0);

MyBuffer Buffer:

vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &PosBuffer);

vkGetBufferMemoryRequirements(LogicalDevice, PosBuffer, OUT &vmr);

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

result = vkBindBufferMemory(LogicalDevice, PosBuffer, IN vdm, 0);
struct pos * positions;

vkMapMemory(LogicalDevice, IN myPosBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &positions);

for (int i = 0; i < NUM_PARTICLES; i++) {
    positions[i].x = Ranf(XMIN, XMAX);
    positions[i].y = Ranf(YMIN, YMAX);
    positions[i].z = Ranf(ZMIN, ZMAX);
    positions[i].w = 1.;
}

vkUnmapMemory(LogicalDevice, IN myPosBuffer.vdm);

struct vel * velocities;

vkMapMemory(LogicalDevice, IN myVelBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &velocities);

for (int i = 0; i < NUM_PARTICLES; i++) {
    velocities[i].x = Ranf(VXMIN, VXMAX);
    velocities[i].y = Ranf(VYMIN, VYMAX);
    velocities[i].z = Ranf(VZMIN, VZMAX);
    velocities[i].w = 0.;
}

vkUnmapMemory(LogicalDevice, IN myVelBuffer.vdm);

struct col * colors;

vkMapMemory(LogicalDevice, IN myColBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &colors);

for (int i = 0; i < NUM_PARTICLES; i++) {
    colors[i].r = Ranf(.3f, 1.);
    colors[i].g = Ranf(.3f, 1.);
    colors[i].b = Ranf(.3f, 1.);
    colors[i].a = 1.;
}

vkUnmapMemory(LogicalDevice, IN myColBuffer.vdm);

#include <stdlib.h>
#define TOP 2147483647. // 2^31 - 1

float Ranf(float low, float high) {
    long random(); // returns integer 0 - TOP
    float r = (float)rand();
    return low + r * (high - low) / (float)RAND_MAX;
}

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

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

A Mechanical Equivalent...
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4  // xc, yc, zc, r
#define PLANE vec4  // a, b, c, d

const VECTOR G = VECTOR(0., -9.8, 0.);
const float DT = 0.1;

const SPHERE Sphere = vec4(-100., -800., 0., 600.); // x, y, z, r

uint gid = gl_GlobalInvocationID.x; // where I am in the global dataset (6 in this example)
// (as a 1d problem, the .y and .z are both 1)

POINT p = Positions[ gid ].xyz;
VELOCITY v = Velocities[ gid ].xyz;

POINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;

Positions[ gid ].xyz = pp;
Velocities[ gid ].xyz = vp;
Displaying the Particles

Telling the Pipeline about its Input

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


VkBufferMemoryBarrier vbmb;

vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;

vbmb.pNext = nullptr;

vbmb.srcAccessFlags = VK_ACCESS_SHADER_WRITE_BIT;

vbmb.dstAccessFlags = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT;

vbmb.srcQueueFamilyIndex = 0;

vbmb.dstQueueFamilyIndex = 0;

vbmb.buffer = inetBuffer;

vbmb.offset = 0;

vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);

const uint32 bufferMemoryBarrierCount = 1;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount, &vbmb, 0, nullptr);

Drawing

Setting a Pipeline Barrier so the Compute Waits for the Drawing

VkBuffer buffers[  ] = MyPosBuffer.buffer, MyVelBuffer.buffer, MyColBuffer.buffer

size_t offsets[  ] = { 0, 0, 0 };

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 3, buffers, offsets);

const uint32_t vertexCount = NUM_PARTICLES;

const uint32_t  instanceCount = 1;

const uint32_t firstVertex = 0;

const uint32_t firstInstance = 0;

vkCmdDraw(CommandBuffers[nextImageIndex], NUM_PARTICLES, 1, 0, 0);

// vertexCount, instanceCount, firstVertex, firstInstance

Setting a Pipeline Barrier so the Drawing Waits for the Compute

VkBufferMemoryBarrier

vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;

vbmb.pNext = nullptr;

vbmb.srcAccessFlags = 0;

vbmb.dstAccessFlags = VK_ACCESS_UNIFORM_READ_BIT;

vbmb.srcQueueFamilyIndex = 0;

vbmb.dstQueueFamilyIndex = 0;

vbmb.buffer = inetBuffer;

vbmb.offset = 0;

vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);

const uint32 bufferMemoryBarrierCount = 1;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount, &vbmb, 0, nullptr);
Specialization Constants

Mike Bailey
mjb@cs.oregonstate.edu

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

What Are Specialization Constants?

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

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

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

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

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

Why Do We Need Specialization Constants?

Specialization Constants could be used for:

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

Specialization Constants are Described in the Compute Pipeline

In the compute shader

In the Vulkan C/C++ program:

Specialization Constant Example -- Setting an Array Size

```c
int asize = 64;

VkSpecializationInfo vsi;

// one array element for each // Specialization Constant
vsi.mapEntryCount = 1;

// size of just this Specialization Constant
vsi.mapEntry[0].constantID = 0;
vsme[0].offset = 0;
vsme[0].size = sizeof(asize);

vsi.dataSize = sizeof(asize);

// array of all the Specialization Constants
vsi.pData = &asize;
```
Linking the Specialization Constants into the Compute Pipeline

```
result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1,
                                    VkSpecializationMapEntry vsme[1];
                                    int asize = 64;
                                    VkSpecializationInfo vsi;
                                    VkPipelineShaderStageCreateInfo vpssci
                                    VkComputePipelineCreateInfo vcpci[1]

In the compute shader

struct abc { int a, int b, float c; }

layout( constant_id = 1 ) in;
layout( local_size_x = 32, local_size_y = 1, local_size_z = 1 ) in;

In the C/C++ program:

int numXworkItems = 64;

VkSpecializationMapEntry vsme[1];
vsme[0].offset = offsetof( abc, a );
vsme[0].constantID = 9;
vsme[0].size = sizeof(abc.a);

vkCreateComputePipelines(logicalDevice, VK_NULL_HANDLE, 1,
                        &vsme[0], &asize, &vsi.pMapEntries, &vsi.mapEntryCount,
                        &vsi.pData, &vsi.dataSize, &vpssci.sType, &vpssci.pSpecializationInfo,
                        &vpssci.pName, computeShader, &vpssci.stage, &vpssci.flags,
                        &vpssci.pNext, &vcpci[0].basePipelineIndex, &vcpci[0].basePipelineHandle,
                        &vcpci[0].layout, &vcpci[0].stage, &vcpci[0].flags, &vcpci[0].pNext)
```

Specialization Constants – Setting Multiple Constants

```
In the compute shader

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

In the C/C++ program:

```c
int a = 1;
int b = 2;
float c = 3.14;
```

Specialization Constant Example – Setting Multiple Constants

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

In the compute shader

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

In the C/C++ program:

```c
int numXworkItems = 64;
```

Remember the Overall Block Diagram?

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

Vulkan.

Synchronization

Mike Bailey
mjb@cs.oregonstate.edu

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

International License
Attribution-NonCommercial-NoDerivatives 4.0
This work is licensed under a Creative Commons
• Used to synchronize work executing on different queues within the same logical device
• You create them, and give them to a Vulkan function which sets them. Later on, you tell a Vulkan function to wait on this particular semaphore
• You don’t end up setting, resetting, or checking the semaphore yourself
• Semaphores must be initialized (“created”) before they can be used

Semaphores

Ask for Something → Semaphore → Try to Use that Something

Creating a Semaphore

VkSemaphoreCreateInfo vsci;

vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

VkSemaphore semaphore;

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

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

Semaphores Example during the Render Loop

VkSemaphore imageReadySemaphore;

VkSemaphoreCreateInfo vsci;

vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

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

uint32_t nextImageIndex;

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkSubmitInfo vsi;

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

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

Fences

Fence Example

VkFenceCreateInfo vfci;

vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = VK_FENCE_CREATE_UNSIGNALED_BIT;

VkFence fence;

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

Set the fence

Wait on the fence(s)

result = vkGetFenceStatus(LogicalDevice, IN fence);

// result = VK_SUCCESS means it has signaled
// result = VK_NOT_READY means it has not signaled

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

// if the fence is still signaled, it will return immediately
// if there is a wait for a fence in the list
// if there is a wait for any one fence in the list
// if the fence is an unsignaled fence
// if the fence is a signaled fence
// if the fence is a signaled fence
// if the fence is a signaled fence

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

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

Controlling Events from the Host

```
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers,
                 bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
```

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

Controlling Events from the Device

```
result = vkCmdSetEvent( CommandBuffer, VkPipelineStageFlagBits::FinishedEvent, signal );
result = vkCmdResetEvent( CommandBuffer, VkPipelineStageFlagBits::FinishedEvent, reset );
```

Pipeline Barriers

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

From the Command Buffer Notes:

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

```
vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
vkCmdDispatchIndirect( commandBuffer, offset );
vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );
vkCmdDebugMarkerEndEXT( commandBuffer );
vkCmdDebugMarkerBeginEXT( commandBuffer, pMarkerInfo );
vkCmdCopyQueryPoolResults( commandBuffer, flags );
vkCmdCopyImageToBuffer( commandBuffer, pRegions );
vkCmdCopyImage( commandBuffer, pRegions );
vkCmdCopyBuffer( commandBuffer, pRegions );
vkCmdClearDepthStencilImage( commandBuffer, pRanges );
vkCmdClearColorImage( commandBuffer, pRanges );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );
vkCmdBlitImage( commandBuffer, filter );
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
vkCmdBindPipeline( commandBuffer, pipeline );
vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdBeginQuery( commandBuffer, flags );
vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );
vkCmdEndRenderPass( commandBuffer );
vkCmdEndQuery( commandBuffer, query );
vkCmdDrawIndirectCountAMD( commandBuffer, stride );
vkCmdDrawIndirect( commandBuffer, stride );
vkCmdDrawIndexedIndirect( commandBuffer, stride );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
```

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

From the Command Buffer Notes:

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

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

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 (RR) as no data has been changed.

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

vkCmdPipelineBarrier() Function Call

defines what data we will be blocking on or un-blocking on.

The Scenario

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

Pipeline Stage Masks – Where in the Pipeline is this Memory Data being Generated or Consumed?

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
VK_PIPELINE_STAGE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
Access Masks – What are you interested in generating or consuming this memory for?

VK_ACCESS_INDIRECT_COMMAND_READ_BIT
VK_ACCESS_INDEX_READ_BIT
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
VK_ACCESS_UNIFORM_READ_BIT
VK_ACCESS_SHADER_READ_BIT
VK_ACCESS_SHADER_WRITE_BIT
VK_ACCESS_COLOR_ATTACHMENT_READ_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_HOSTREAD_BIT
VK_ACCESS_HOST_WRITE_BIT
VK_ACCESS_MEMORY_READ_BIT
VK_ACCESS_MEMORY_WRITE_BIT

Access Operations and what Pipeline Stages they can be used in

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

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

Pipeline Stages and what Access Operations are Allowed

The Scenario

src.cars are generating the image
dst cars are doing something with that image

mjb – July 24, 2020
The Scenario

src cars

dst cars

 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

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

Aliasing

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

http://cs.oregonstate.edu/~mjb/vulkan
The Nyquist Criterion

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

MultiSampling

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

There are two approaches to this:

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

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

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

Consider Two Triangles Who Pass Through the Same Pixel

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

Supersampling

Final Pixel Color = \( \frac{\sum \text{Color sample from subpixel}}{\# \text{Fragment Shader calls}} \)
**Multisampling**

Consider Two Triangles Who Pass Through the Same Pixel

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

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

**Setting up the Image**

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

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

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

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

<table>
<thead>
<tr>
<th>Multisampling</th>
<th>Supersampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue fragment shader calls</td>
<td>1</td>
</tr>
<tr>
<td>Red fragment shader calls</td>
<td>1</td>
</tr>
</tbody>
</table>

**VkPipelineMultisampleStateCreateInfo**

```
vpmsci.mLeastFractionOfSamplesToSample = 0.5f;
vpmsci.mFractionalSupersampling = VK_TRUE;
```

0. produces simple multisampling
1. produces partial supersampling
1. Produces complete supersampling
Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

Subpass:

```cpp
void ComputeSubpassAttachments()
{
    // Color attachment
    vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
    vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    vad[0].flags = 0;
    vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
    vad[0].flags = 0;
    vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    vad[0].flags = 0;
    vad[0].samples = VK_SAMPLE_COUNT_8_BIT;
}
```

For the *ImageLayout, use VK_IMAGE_LAYOUT_GENERAL.

Resolving the Image:

Converting the Multisampled Image to a VK_SAMPLE_COUNT_1_BIT image

```cpp
void ResolveImage()
{
    VkImageCreateInfo imageCreateInfo;
    imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageCreateInfo.imageType = VK_IMAGE_TYPE_2D;
    imageCreateInfo.width = Width;
    imageCreateInfo.height = Height;
    imageCreateInfo.depth = 1;
    imageCreateInfo.arraySize = 1;
    imageCreateInfo.format = VK_FORMAT_D32_SFLOAT_S8_UINT;
    imageCreateInfo.mipLevels = 1;
    imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
    imageCreateInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT;
    imageCreateInfo.flags = 0;
    imageCreateInfoنش_ = &imageCreateInfo;
    image = result;
}
```

For the *ImageLayout, use VK_IMAGE_LAYOUT_GENERAL.

Multi-pass Rendering uses Attachments --

What is a Vulkan Attachment Anyway?

"An attachment is an image associated with a renderpass that can be used as the input or output of one or more of its subpasses."

— Vulkan Programming Guide

An attachment can be written to, read from, or both.

For example:

- Subpass
- Attachment
- Subpass
- Framebuffer

What is an Example of Wanting to do This?

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

Here’s the trick:

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

- position (x,y,z)
- normal (nx,ny,nz)
- material color (r,g,b)
- texture coordinates (s,t)

As well as:

- the current light source positions and colors
- the current eye position

When we write these out, the final framebuffers will contain just information for the pixels that can be seen. We then make a second pass running the expensive lighting model just for those pixels. This is known as the *G-buffers Algorithm*.
Back in Our Single-pass Days

So far, we've only performed single-pass rendering, within a single Vulkan RenderPass.

Here comes a quick reminder of how we did that.

Afterwards, we will extend it.

Attachment  #1

Depth Attachment

Attachment  #0

3D Rendering Pass

Output

Subpass #0

Back in Our Single-pass Days, I

Subpass #1

Lighting Pass

Attachment  #1

Gluefer Attachments

Attachment  #0

3D Rendering Pass

Subpass #0

Subpass #1

Output

Multipass Rendering

So far, we've only performed single-pass rendering, but within a single Vulkan RenderPass, we can also have several subpasses, each of which is feeding information to the next subpasses or subpasses.

In this case, we will look at following up a 3D rendering with Gbuffer operations.

Attachment  #2

Output
Multipass, III

```c
VkSubpassDescription
  vsd[0].flags = 0;
  vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
  vsd[0].inputAttachmentCount = 0;
  vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
  vsd[0].colorAttachmentCount = 0;
  vsd[0].pColorAttachments = (VkAttachmentReference *)nullptr;
  vsd[0].pResolveAttachments = (VkAttachmentReference *)nullptr;
  vsd[0].pDepthStencilAttachment = &depthOutput;
  vsd[0].preserveAttachmentCount = 0;
  vsd[0].pPreserveAttachments = (uint32_t *)nullptr;
  vsd[1].flags = 0;
  vsd[1].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
  vsd[1].inputAttachmentCount = 0;
  vsd[1].pInputAttachments = (VkAttachmentReference *)nullptr;
  vsd[1].colorAttachmentCount = 1;
  vsd[1].pColorAttachments = &gBufferOutput;
  vsd[1].pResolveAttachments = (VkAttachmentReference *)nullptr;
  vsd[1].pDepthStencilAttachment = (VkAttachmentReference *)nullptr;
  vsd[1].preserveAttachmentCount = 0;
  vsd[1].pPreserveAttachments = (uint32_t *)nullptr;
  vsd[2].flags = 0;
  vsd[2].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
  vsd[2].inputAttachmentCount = 2;
  vsd[2].pInputAttachments = &lightingInput[0];
  vsd[2].colorAttachmentCount = 1;
  vsd[2].pColorAttachments = &lightingOutput;
  vsd[2].pResolveAttachments = (VkAttachmentReference *)nullptr;
  vsd[2].pDepthStencilAttachment = (VkAttachmentReference *)nullptr;
  vsd[2].preserveAttachmentCount = 0;
  vsd[2].pPreserveAttachments = (uint32_t *)nullptr;

VkSubpassDependency
  vsdp[0].srcSubpass = 0; // depth rendering
  vsdp[0].dstSubpass = 1; // gbuffer
  vsdp[0].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
  vsdp[0].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
  vsdp[0].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
  vsdp[0].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
  vsdp[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
  vsdp[1].srcSubpass = 1; // gbuffer
  vsdp[1].dstSubpass = 2; // color output
  vsdp[1].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
  vsdp[1].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
  vsdp[1].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
  vsdp[1].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
  vsdp[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
```

Multipass, IV

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

Multipass, V

```c
void vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// subpass #0 is automatically started here
void vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
void vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
void vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
void vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
// ...
void vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #1 is started here
// ...
void vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #2 is started here
// ...
void vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
```

Multipass, VI

```c
void vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
// subpass #0 is automatically started here
void vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);
void vkCmdBindDescriptorSets(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr);
void vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
void vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
// ...
void vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #1 is started here
// ...
void vkCmdNextSubpass(CommandBuffers[nextImageIndex], VK_SUBPASS_CONTENTS_INLINE);
// subpass #2 is started here
// ...
void vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);
```
The Ray-trace Pipeline Involves Five New Shader Types

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

The Ray Intersection Process for a Sphere

1. Sphere equation: \((x-xc)^2 + (y-yc)^2 + (z-zc)^2 = R^2\)
2. Ray equation: \((x,y,z) = (x0,y0,z0) + t*(dx,dy,dz)\)

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

\[At^2 + Bt + C = 0\]

Solve for \(t_1, t_2\)

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

In Vulkan terms:

- \(gl_{WorldRayOriginKHR}\) = \((x0,y0,z0)\)
- \(gl_{HitKHR}\) = \(t\)
- \(gl_{WorldRayDirectionKHR}\) = \((dx,dy,dz)\)

The Ray Intersection Process for a Cube

1. Plane equation: \(Ax + By + Cz + D = 0\)
2. Ray equation: \((x,y,z) = (x0,y0,z0) + t*(dx,dy,dz)\)

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

\[At + B = 0\]

Solve for \(t\)

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

The ultimate entry and exit values are:

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

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

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

Creating Bottom Level Acceleration Structures

```cpp
vkCreateAccelerationStructureKHR BottomLevelAccelerationStructure;
VkAccelerationStructureInfoKHR vasi;
vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
vasi.flags = 0;
vasi.pNext = nullptr;
vasi.instanceCount = 0;
vasi.geometryCount = << number of vertex buffers >>;
vasi.pGeometries = << vertex buffer pointers >>;
VkAccelerationStructureCreateInfoKHR vasci;
vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
vasci.pNext = nullptr;
vasci.info = &vasi;
vasci.compactedSize = 0;
result = vkCreateAccelerationStructureKHR( LogicalDevice, &vasci, PALLOCATOR, OUT &BottomLevelAccelerator);
```

Creating Top Level Acceleration Structures

```cpp
vkCreateAccelerationStructureKHR TopLevelAccelerationStructure;
VkAccelerationStructureInfoKHR vasi;
vasi.sType = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
vasi.flags = 0;
vasi.pNext = nullptr;
vasi.instanceCount = << number of bottom level acceleration structure instances >>;
vasi.geometryCount = 0;
vasi.pGeometries = VK_NULL_HANDLE;
VkAccelerationStructureCreateInfoKHR vasci;
vasci.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR;
vasci.pNext = nullptr;
vasci.info = &vasi;
vasci.compactedSize = 0;
result = vkCreateAccelerationStructureKHR( LogicalDevice, &vasci, PALLOCATOR, &TopLevelAccelerator);
```

Ray Generation Shader

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

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

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

Intersection Shader

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

In Vulkan terms:

```cpp
gl_WorldRayOriginKHR = (x0,y0,z0);
gl_HitKHR = t;
gl_WorldRayDirectionKHR = (dx,dy,dz);
```

A New Built-in Function

```cpp
void traceKHR( accelerationStructureKHR topLevel, 
               uint rayFlags, 
               uint cullMask, 
               uint sbtRecordOffset, 
               uint sbtRecordStride, 
               uint missIndex, 
               vec3 origin, 
               float tmin, 
               vec3 direction, 
               float tmax, 
               int payload );
```
Handle a ray that doesn't hit any objects.

```plaintext
rayPayloadKHR myPayload
{
    vec4 color;
}
void main()
{
    color = vec4(0., 0., 0., 1.);
}
```

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

```plaintext
layout(binding = 4, set = 0) buffer outputProperties
{
    float outputValues[];
} outputData;
layout(location = 0) rayPayloadInKHR uint outputId;
layout(location = 1) rayPayloadInKHR uint hitCounter;
hitAttributeKHR vec3 attribs;
void main()
{
    outputData.outputValues[outputId + hitCounter] = gl_PrimitiveID;
    hitCounter = hitCounter + 1;
}
```

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

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

Loosely equivalent to “discard”

```plaintext
void terminateRayKHR();
void ignoreIntersectionKHR(float hit, uint hitKind);
```

The Trigger comes from the Command Buffer:

```plaintext
vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_RAYTRACING_KHR, RaytracePipeline);
vkCmdTraceRaysKHR(CommandBuffer, raygenShaderBindingTableBuffer, raygenShaderBindingOffset, missShaderBindingTableBuffer, missShaderBindingOffset, missShaderBindingStride, hitShaderBindingTableBuffer, hitShaderBindingOffset, hitShaderBindingStride, callableShaderBindingTableBuffer, callableShaderBindingOffset, callableShaderBindingStride, width, height, depth);
```
Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

https://www.youtube.com/watch?v=QL7xXi2INJ8

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