1

Introduction to the Vulkan Computer Graphics API

Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

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

FULL.pptx

mjb – July 24, 2020

2

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

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

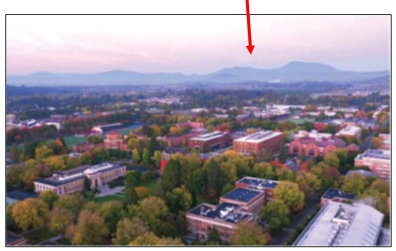
mjb – July 24, 2020

3

Mike Bailey

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

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



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

mjb – July 24, 2020

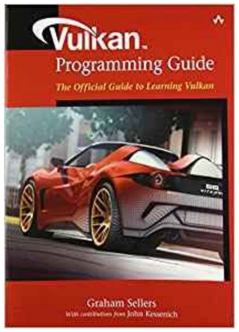
4

Sections

mjb – July 24, 2020

5

My Favorite Vulkan Reference



Graham Sellers, *Vulkan Programming Guide*, Addison-Wesley, 2017.

6

Vulkan.

Introduction

Mike Bailey

mjb@cs.oregonstate.edu

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

7

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!

Ali Alsalehy	Alan Neads
Natasha Anisimova	Raja Petroff
Jianchang Bi	Bei Rong
Christopher Cooper	Lawrence Roy
Richard Cunard	Lily Shellhammer
Braxton Cuneo	Hannah Solorzano
Benjamin Fields	Jian Tang
Trevor Hammock	Glenn Upthagrove
Zach Lerew	Logan Wingard
Victor Li	

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.)

8

History of Shaders

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:

https://www.khronos.org/opengl/wiki/History_of_OpenGL

History of Shaders

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

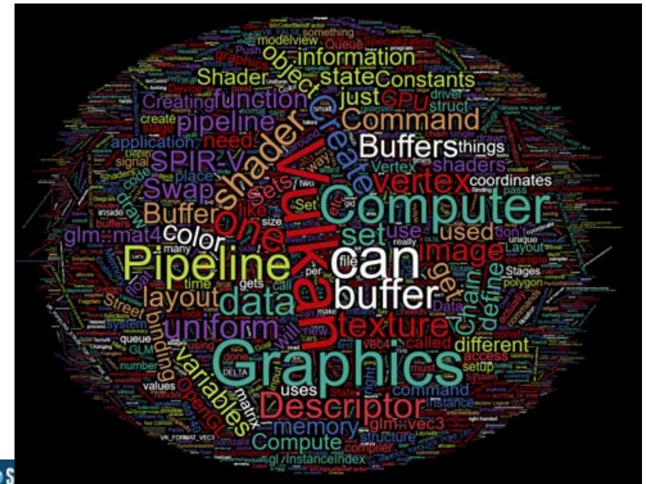
2016: Vulkan 1.1

2020: Vulkan 1.2

There is lots more detail at:

https://en.wikipedia.org/wiki/Vulkan_(API)

Everything You Need to Know is Right Here … Somewhere ☺



Top Three Reasons that Prompted the Development of Vulkan

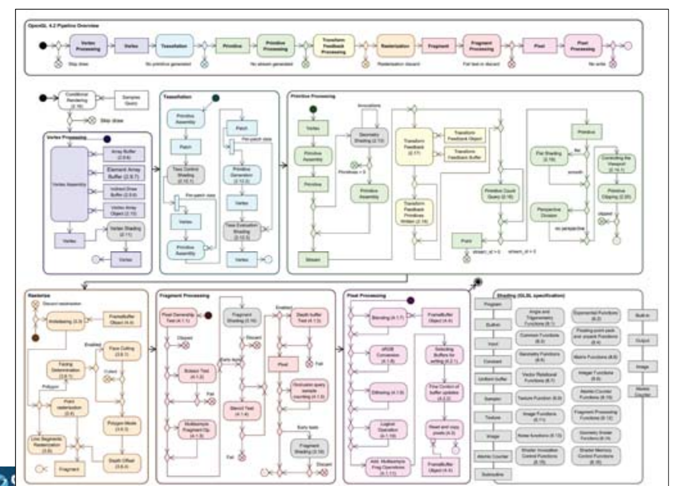
1. Performance
2. Performance
3. Performance

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.

OpenGL 4.2 Pipeline Flowchart



Why is it so important to keep the GPU Busy?

NVidia Titan V Specs vs. Titan Xp, 1080 Ti

	Titan V	Tesla V100	Tesla P100	GTX 1080 Ti	GTX 1080
GPU	GV100	GV100	GP100 Cut-Down Pascal	GP102 Pascal	GP104-400 Pascal
Transistor Count	21.1B	21.1B	15.3B	12B	7.2B
Fab Process	12nm FFN	12nm FFN	16nm FinFET	16nm FinFET	16nm FinFET
CUDA Cores / Tensor Cores	5120 / 640	5120 / 640	3584 / 0	3584 / 0	2560 / 0
TMUs	320		224	224	160
ROPs	?		96 (?)	88	64
Core Clock	1200MHz		1328MHz	-	1607MHz
Boost Clock	1455MHz	1370MHz	1480MHz	1600MHz	1733MHz
FP32 TFLOPs	15TFLOPs	14TFLOPs	10.6TFLOPs	~11.4TFLOPs	9TFLOPs
Memory Type	HBM2	HBM2	HBM2	GDDR5X	GDDR5X
Memory Capacity	12GB	16GB	16GB	11GB	8GB
Memory Clock	1.7Gbps HBM2	1.75Gbps HBM2	?	11Gbps	10Gbps GDDR5X
Memory Interface	3072-bit	4096-bit	4096-bit	352-bit	256-bit
Memory Bandwidth	653GB/s	900GB/s	?	~484GB/s	320.32GB/s
Total Power Budget ("TDP")	250W	250W	300W	250W	180W
Power Connectors	1x 8-pin 1x 6-pin		?	1x 8-pin 1x 6-pin	1x 8-pin
Release Date	12/07/2017		4Q16-1Q17	TBD	5/27/2016
Release Price	$3000	$10000	-	$700	Reference: $700 MSRP: $600 Now: $500

The nVidia Titan V graphics card is not targeted at gamers, but rather at scientific and machine/deep learning applications. That does not, however, mean that the card is incapable of gaming, nor does it mean that we can't extrapolate future key performance metrics for Volta. The Titan V is a derivative of the earlier released GV100 GPU, part of the Tesla accelerator card series. The key differentiator is that the Titan V ships at $3000, whereas the Tesla V100 was available as part of a $10,000 developer kit. The Tesla V100 still offers greater memory capacity by 4GB – 16GB HBM2 versus 12GB HBM2 – and has a wider memory interface, but other core features remain matched or nearly matched. Core count, for one, is 5120 CUDA cores on each GPU, with 640 Tensor cores (used for Tensorflow deep/machine learning workloads) on each GPU.

Who was the original Vulcan?

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 smithery. 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."

https://en.wikipedia.org/wiki/Vulcan_(mythology)

Why Name it after the God of the Forge?

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.

COLLADA. DataFormat EGL. glTF NNEF
OpenCL OpenGL ES OpenGL OpenGL SC OpenVG.
OpenXR OpenVX. SPIR. SYCL. Vulkan.
WebGL

Playing "Where's Waldo" with Khronos Membership

Who's Been Specifically Working on Vulkan?

Vulkan

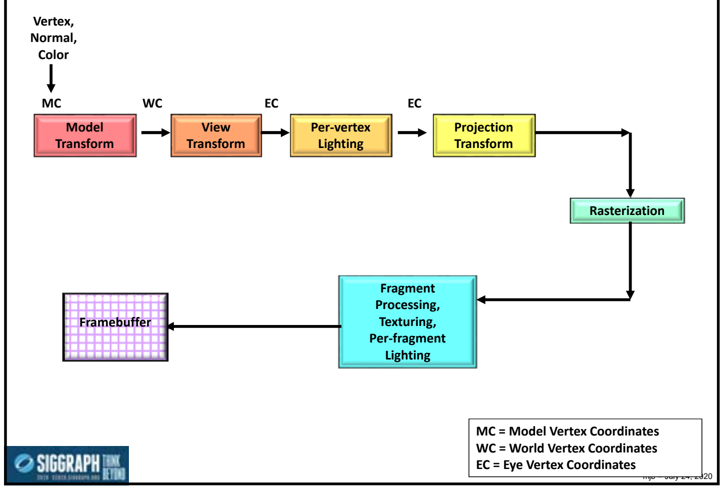
- Originally derived from AMD's *Mantle* API
- Also heavily influenced by Apple's *Metal* API and Microsoft's *DirectX 12*
- Goal: much less driver complexity and overhead than OpenGL has
- Goal: much less user hand-holding
- Goal: higher single-threaded performance than OpenGL can deliver
- Goal: able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

Vulkan Differences from OpenGL

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

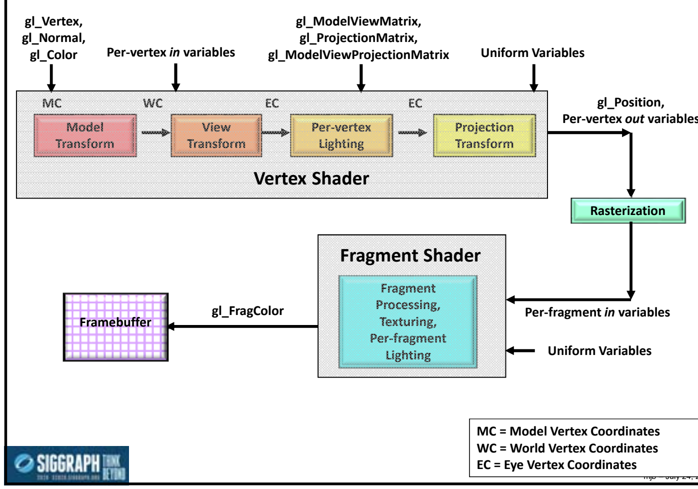
The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

21



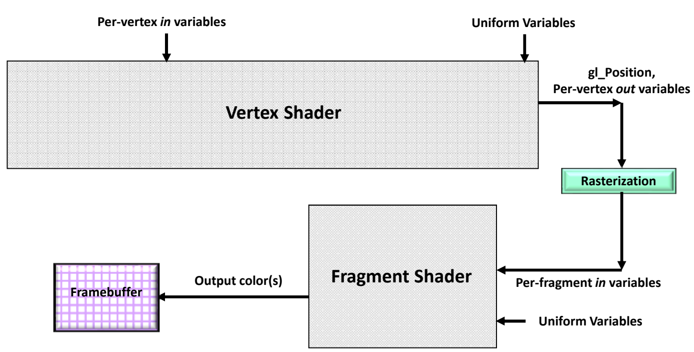
The Basic Computer Graphics Pipeline, Shader-style

22



The Basic Computer Graphics Pipeline, Vulkan-style

23



mjb – July 24, 2020

Moving part of the driver into the application

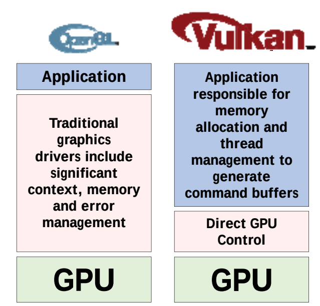
24

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets



Khronos Group

Simpler drivers for low-overhead efficiency and cross vendor portability

Layered architecture so validation and debug layers can be unloaded when not needed

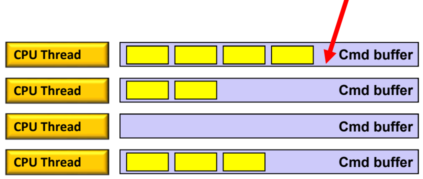
Run-time only has to ingest SPIR-V intermediate language

Unified API for mobile, desktop, console and embedded platforms

mjb – July 24, 2020

Vulkan Highlights: Command Buffers

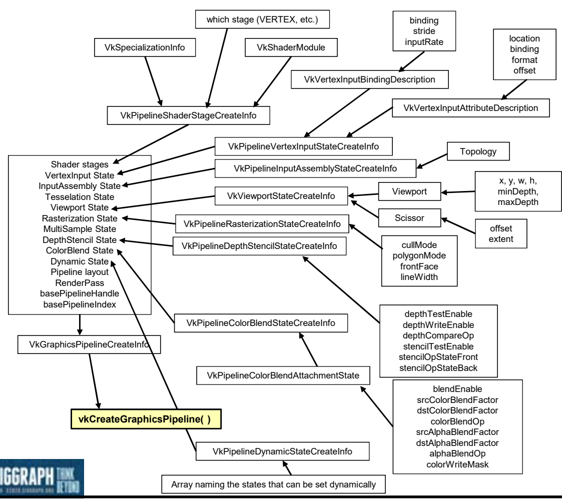
- Graphics commands are sent to command buffers
- E.g., *vkCmdDoSomething(cmdBuffer, …);*
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed to Queues when the application wants them to be flushed
- Each command buffer can be filled from a different thread



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: Creating a Pipeline



Querying the Number of Something


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

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

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

How many total there are — **Where to put them**

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

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

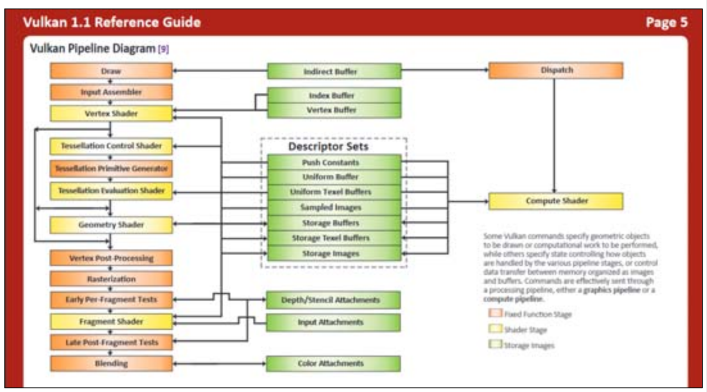
Vulkan Code has a Distinct "Style" of Setting Information in *structs* and then Passing that Information as a pointer-to-the-struct


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

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

VkMemoryRequirements        vmr;

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

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

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

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


Vulkan Quick Reference Card – I Recommend you Print This!

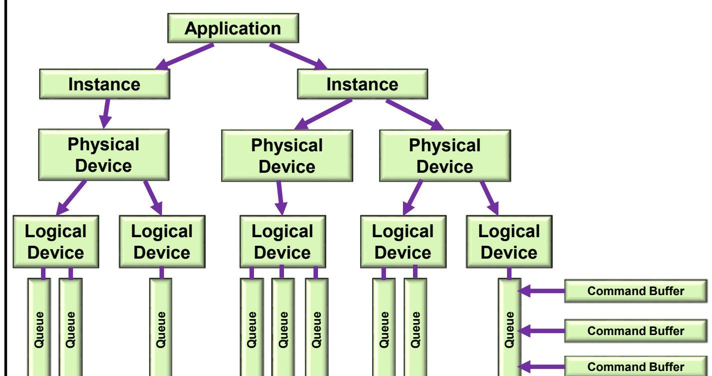
https://www.khronos.org/files/vulkan11-reference-guide.pdf

Vulkan Quick Reference Card



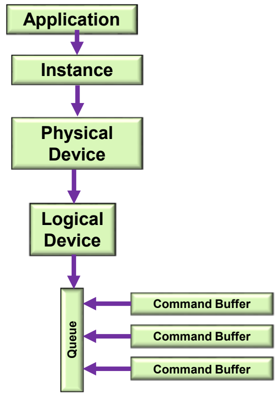
https://www.khronos.org/files/vulkan11-reference-guide.pdf

Vulkan Highlights: Overall Block Diagram



Vulkan Highlights: a More Typical Block Diagram

33



Steps in Creating Graphics using Vulkan

34

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

Vulkan GPU Memory

35

- 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

36

- 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

37

- 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

38

- 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

39

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

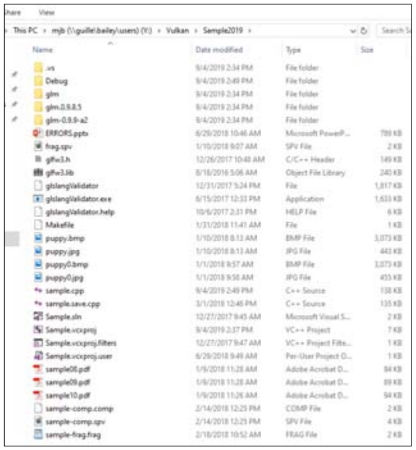
GLSL Source → **External GLSL Compiler** (Develop Time) → SPIR-V → **Compiler in driver** (Run Time) → Vendor-specific code

Advantages:

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

Your Sample2019.zip File Contains This

40



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

The Vulkan Sample Code Included with These Notes

Mike Bailey

mjb@cs.oregonstate.edu

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

Sample Program Output

Sample Program Keyboard Inputs

'l', 'L':	Toggle **l**ighting off and on
'm', 'M':	Toggle display **m**ode (textures vs. colors, for now)
'p', 'P':	**P**ause the animation
'q', 'Q':	**q**uit the program
Esc:	quit the program
'r', 'R':	**T**oggle rotation-animation and using the mouse
'i', 'I':	Toggle using a vertex buffer only vs. an **i**ndex buffer (in the index buffer version)
'1', '4', '9'	Set the number of instances (in the instancing version)

Caveats on the Sample Code, I

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

Caveats on the Sample Code, II

6. There are great uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'ed a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program


```
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'\n", DEBUGFILE );
		FpDebug = stderr;
	}
	fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d\n", 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\n");

	vkQueueWaitIdle( Queue );
	vkDeviceWaitIdle( LogicalDevice );
	DestroyAllVulkan( );
	glfwDestroyWindow( MainWindow );
	glfwTerminate( );
	return 0;
}
```

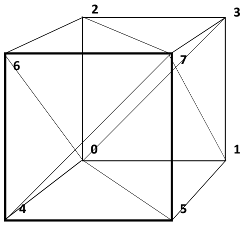

InitGraphics( ), I


```
void
InitGraphics( )
{
	HERE_I_AM( "InitGraphics" );

	VkResult result = VK_SUCCESS;

	Init01Instance( );

	InitGLFW( );

	Init02CreateDebugCallbacks( );

	Init03PhysicalDeviceAndGetQueueFamilyProperties( );

	Init04LogicalDeviceAndQueue( );

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

	Init05UniformBuffer( sizeof(Light),	&MyLightUniformBuffer );
	Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );

	Init05MyVertexDataBuffer(  sizeof(VertexData), &MyVertexDataBuffer );
	Fill05DataBuffer( MyVertexDataBuffer,			(void *) VertexData );

	Init06CommandPool( );
	Init06CommandBuffers( );
```


InitGraphics( ), II


```
	Init07TextureSampler( &MyPuppyTexture.texSampler );
	Init07TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);

	Init08Swapchain( );

	Init09DepthStencilImage( );

	Init10RenderPasses( );

	Init11Framebuffers( );

	Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
	Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );

	Init13DescriptorSetPool( );
	Init13DescriptorSetLayouts();
	Init13DescriptorSets( );

	Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
				VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline  );
}
```

A Colored Cube



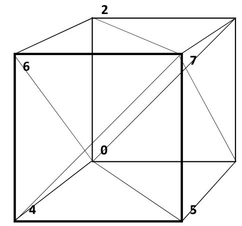
```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

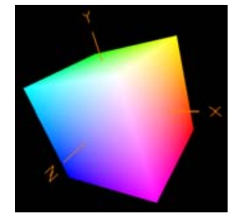
```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        { 1., -1., -1. },
        { -1., 1., -1. },
        { 1., 1., -1. },
        { -1., -1., 1. },
        { 1., -1., 1. },
        { -1., 1., 1. },
        { 1., 1., 1. }
};
```

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


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. }
        },
```





The Vertex Data is in a Separate File

#include "SampleVertexData.cpp"

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

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

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


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


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

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

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

Vulkan Software Philosophy

53

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

Vulkan takes a non-C++ object-oriented approach in that those typedef'ed 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 );
```

mjb – July 24, 2020

Vulkan Conventions

54

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.

mjb – July 24, 2020

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

55

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

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

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

	How many total there are	Where to put them
result = vkEnumeratePhysicalDevices(Instance,	&count,	nullptr);
result = vkEnumeratePhysicalDevices(Instance,	&count,	&physicalDevices[0]);

mjb – July 24, 2020

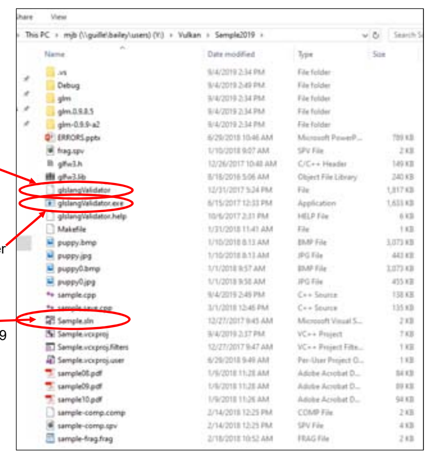
Your Sample2019.zip File Contains This

56

Linux shader compiler

Windows shader compiler

Double-click here to launch Visual Studio 2019 with this solution



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

mjb – July 24, 2020

Reporting Error Results, I

57

```
struct errorcode
{
        VkResult       resultCode;
        std::stringmeaning;
}
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_INCOMPATIBLE_DRIVER,              "Incompatible Driver"    },
        { 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,                  "Error Out of Date"      },
        { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR,         "Incompatible Display"   },
        { VK_ERROR_VALIDATION_FAILED_EXT,            "Valuidation 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"  },
};
```


Reporting Error Results, II

58

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

        const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
        std::string meaning = "";
        for( int i = 0; i < numErrorCodes; i++ )
        {
                if( result == ErrorCodes[i].resultCode )
                {
                        meaning = ErrorCodes[i].meaning;
                        break;
                }
        }

        fprintf( FpDebug, "\n%s: %s\n", prefix.c_str(), meaning.c_str() );
        fflush(FpDebug);
}
```


Extras in the Code

59

```
#define REPORT(s)          { PrintVkError( result, s );  fflush(FpDebug); }


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


bool              Paused;

bool              Verbose;



#define DEBUGFILE          "VulkanDebug.txt"

errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );


const int32_t   OFFSET_ZERO =   0;
```


60

Vulkan

Drawing

Mike Bailey

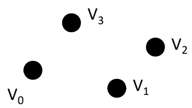
mjb@cs.oregonstate.edu

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

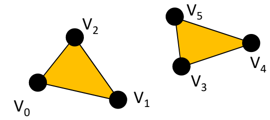
Vulkan Topologies

61

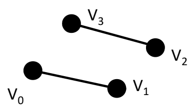
VK_PRIMITIVE_TOPOLOGY_POINT_LIST



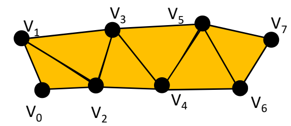
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST



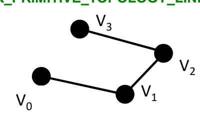
VK_PRIMITIVE_TOPOLOGY_LINE_LIST



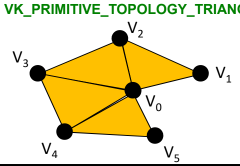
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP



VK_PRIMITIVE_TOPOLOGY_LINE_STRIP



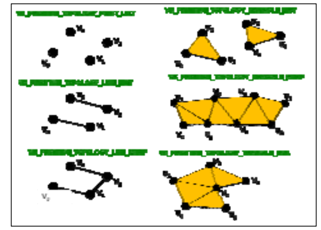
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN



mjb – July 24, 2020

Vulkan Topologies

62

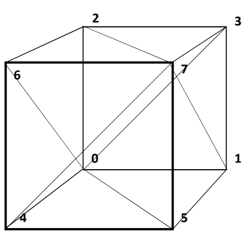


```
typedef enum VkPrimitiveTopology
{
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
    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;
```

mjb – July 24, 2020

A Colored Cube Example

63



```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        { 1., -1., -1. },
        { -1., 1., -1. },
        { 1., 1., -1. },
        { -1., -1., 1. },
        { 1., -1., 1. },
        { -1., 1., 1. },
        { 1., 1., 1. }
};
```

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

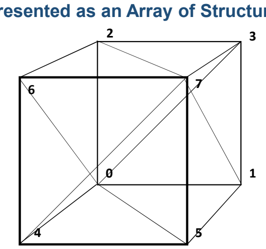
mjb – July 24, 2020

Triangles Represented as an Array of Structures

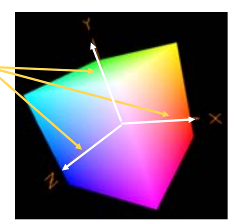
64

From the file **SampleVertexData.cpp**:

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

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

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

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



Modeled in right-handed coordinates



mjb – July 24, 2020

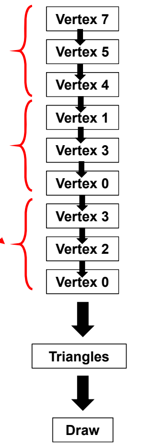
Non-indexed Buffer Drawing

65

From the file **SampleVertexData.cpp**:

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

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

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

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

Stream of Vertices



mjb – July 24, 2020

Filling the Vertex Buffer

66

```
struct vertex  VertexData[ ] =
{
        . . .
};

MyBuffer          MyVertexDataBuffer;


Init05MyVertexDataBuffer(  sizeof(VertexData), OUT &MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer,              (void *) VertexData );


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

mjb – July 24, 2020

A Preview of What *Init05DataBuffer* Does

67

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

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

        VkMemoryAllocateInfo                 vmai;
                vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
                vmai.pNext = nullptr;
                vmai.allocationSize = vmr.size;
                vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );

        VkDeviceMemory                       vdm;
        result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
        pMyBuffer->vdm = vdm;

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

mjb – July 24, 2020

Telling the Pipeline about its Input

68

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

C/C++:

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

GLSL Shader:

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

```
VkVertexInputBindingDescription              vvibd[1];      // one of these per buffer data buffer
        vvibd[0].binding = 0;                               // which binding # this is
        vvibd[0].stride = sizeof( struct vertex );          // bytes between successive structs
        vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

mjb – July 24, 2020

Telling the Pipeline about its Input

69

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

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

```
VkVertexInputAttributeDescription      vviad[4];        // array per vertex input attribute
        // 4 = vertex, normal, color, texture coord
        vviad[0].location = 0;              // location in the layout decoration
        vviad[0].binding = 0;               // which binding description this is part of
        vviad[0].format = VK_FORMAT_VEC3;      // x, y, z
        vviad[0].offset = offsetof( struct vertex, position );         // 0

        vviad[1].location = 1;
        vviad[1].binding = 0;
        vviad[1].format = VK_FORMAT_VEC3;      // nx, ny, nz
        vviad[1].offset = offsetof( struct vertex, normal );           // 12

        vviad[2].location = 2;
        vviad[2].binding = 0;
        vviad[2].format = VK_FORMAT_VEC3;      // r, g, b
        vviad[2].offset = offsetof( struct vertex, color );            // 24

        vviad[3].location = 3;
        vviad[3].binding = 0;
        vviad[3].format = VK_FORMAT_VEC2;      // s, t
        vviad[3].offset = offsetof( struct vertex, texCoord );         // 36
```

Always use the C/C++ construct ***offsetof***, rather than hardcoding the value!

SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Telling the Pipeline about its Input

70

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.

```
VkPipelineVertexInputStateCreateInfo       vpvisci;       // used to describe the input vertex attributes
        vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
        vpvisci.pNext = nullptr;
        vpvisci.flags = 0;
        vpvisci.vertexBindingDescriptionCount = 1;
        vpvisci.pVertexBindingDescriptions = vvibd;
        vpvisci.vertexAttributeDescriptionCount = 4;
        vpvisci.pVertexAttributeDescriptions = vviad;

VkPipelineInputAssemblyStateCreateInfo      vpiasci;
        vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
        vpiasci.pNext = nullptr;
        vpiasci.flags = 0;
        vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;;
```

SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Telling the Pipeline about its Input

71

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

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

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

SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Telling the Command Buffer what Vertices to Draw

72

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

```
VkBuffer buffers[1] = MyVertexDataBuffer.buffer;

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

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

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

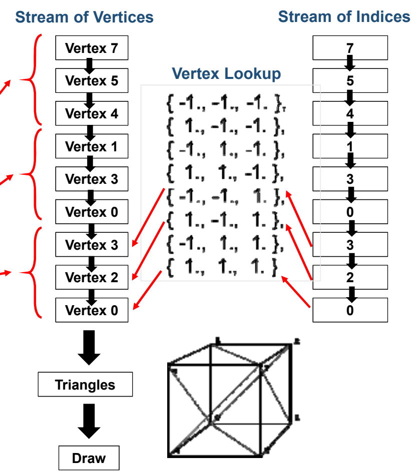
Always use the C/C++ construct ***sizeof***, rather than hardcoding a count!

SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Drawing with an Index Buffer


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

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

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



Drawing with an Index Buffer


```
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
} VkIndexType;
```

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


Drawing with an Index Buffer


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

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

Init05MyIndexDataBuffer(  sizeof(JustIndexData),   IN &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer,             (void *) JustIndexData );
```


Drawing with an Index Buffer


```
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer       = { MyJustIndexDataBuffer.buffer  };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
                                        // 0, 1 = firstBinding, bindingCount
vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32 );

const uint32_t vertexCount = sizeof( JustVertexData ) / sizeof( JustVertexData[0] );
const uint32_t indexCount  = sizeof( JustIndexData )  / sizeof( JustIndexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
const uint32_t vertexOffset  = 0;

vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,
                        vertexOffset, firstInstance );
```

Indirect Drawing (not to be confused with Indexed)


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

vkCmdDrawIndirect(CommandBuffers[nextImageIndex], buffer, offset, drawCount, stride);

Compare this with:

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

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

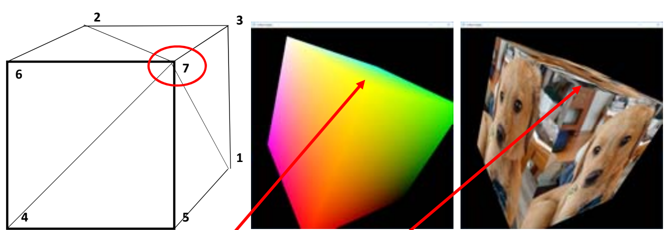
vkCmdDrawIndexedIndirect(commandBuffer, buffer, offset, drawCount, stride);

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

Compare this with:

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

Sometimes the Same Point Needs Multiple Attributes



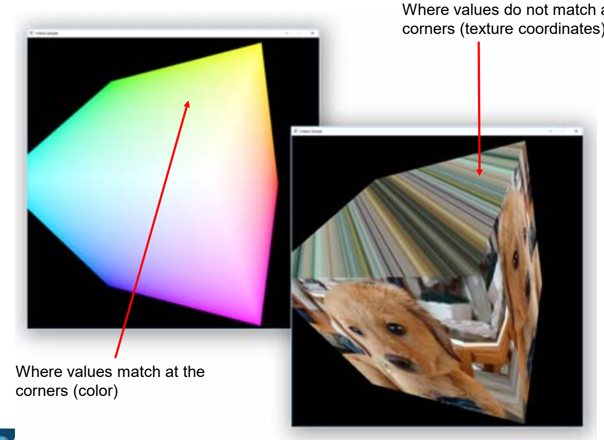
Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn't.

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

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

Sometimes the Same Point Needs Multiple Attributes

Where values do not match at the corners (texture coordinates)



Where values match at the corners (color)

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

```
v 1.710541 1.283360 -0.040860
v 1.714593 1.273043 -0.041268
v 1.706114 1.279109 -0.040795
v 1.719083 1.277235 -0.041195
v 1.722786 1.267216 -0.041939
v 1.727196 1.271285 -0.041795
v 1.730680 1.261384 -0.042630
v 1.723121 1.280378 -0.037323
v 1.714513 1.286599 -0.037101
v 1.706156 1.293797 -0.037073
v 1.702207 1.290297 -0.040704
v 1.697843 1.285852 -0.040489
v 1.709169 1.295845 -0.029862
v 1.717523 1.288344 -0.029807

. . .

vn  0.1725  0.2557 -0.9512
vn -0.1979 -0.1899 -0.9616
vn -0.2050 -0.2127 -0.9554
vn  0.1664  0.3020 -0.9387
vn -0.2040 -0.1718 -0.9638
vn  0.1645  0.3203 -0.9329
vn -0.2055 -0.1698 -0.9638
vn  0.4419  0.6436 -0.6249
vn  0.4573  0.5682 -0.6841
vn  0.5160  0.5538 -0.6535
vn  0.1791  0.2082 -0.9616
vn -0.2167 -0.2250 -0.9499
vn  0.6624  0.6871 -0.2987
```

```
vt 0.816406 0.955536
vt 0.822754 0.959168
vt 0.815918 0.959442
vt 0.823242 0.955292
vt 0.829102 0.958862
vt 0.829590 0.955109
vt 0.835449 0.958618
vt 0.824219 0.951263
vt 0.817383 0.951538
vt 0.810059 0.951385
vt 0.809570 0.955383
vt 0.809082 0.959320
vt 0.811035 0.946381

. . .

f 73/73/75 65/65/67 66/66/68
f 66/66/68 74/74/76 73/73/75
f 74/74/76 66/66/68 67/67/69
f 67/67/69 75/75/77 74/74/76
f 75/75/77 67/67/69 69/69/71
f 69/69/71 76/76/78 75/75/77
f 71/71/73 72/72/74 77/77/79
f 72/72/74 78/78/80 77/77/79
f 78/78/80 72/72/74 73/73/75
f 73/73/75 79/79/81 78/78/80
f 79/79/81 73/73/75 74/74/76
f 74/74/76 80/80/82 79/79/81
f 80/80/82 74/74/76 75/75/77
f 75/75/77 81/81/83 80/80/82
```

V / T / N

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

Shaders and SPIR-V

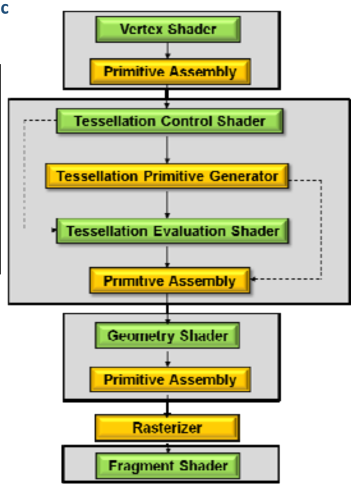
Mike Bailey

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



= Fixed Function

= Programmable

Vulkan Shader Stages

Shader stages

```
typedef enum VkPipelineStageFlagBits {
   VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
   VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
   VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
   VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
   VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
   VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
   VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
   VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
   VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
   VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
   VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
   VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
   VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
   VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
   VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
   VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
   VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
```

How Vulkan GLSL Differs from OpenGL GLSL

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

- In the compiler, there is an automatic
  #define VULKAN 100

Vulkan Vertex and Instance indices:	OpenGL uses:
gl_VertexIndex	gl_VertexID
gl_InstanceIndex	gl_InstanceID

- 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

How Vulkan GLSL Differs from OpenGL GLSL

Shader combinations of separate texture data and samplers:

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

Note: our sample code doesn't use this.

Descriptor Sets:

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

Push Constants:

```
layout( push_constant ) . . . ;
```

Specialization Constants:

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

- Only for scalars, but a vector's components can be constructed from specialization constants

Specialization Constants for Compute Shaders:

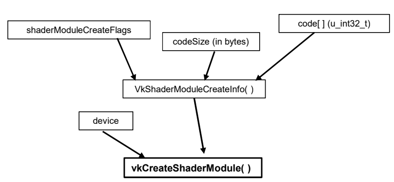
```
layout( local_size_x_id = 8, local_size_y_id = 16 );
```

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

Vulkan: Shaders' use of Layouts for Uniform Variables

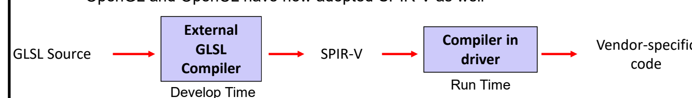

```
// non-sampler variables must be in a uniform block:
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat3 uNormalMatrix;
} Matrices;

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



Advantages:

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

SPIR-V: Standard Portable Intermediate Representation for Vulkan

89

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

Shaderfile extensions:

.vert	Vertex
.tesc	Tessellation Control
.tese	Tessellation Evaluation
.geom	Geometry
.frag	Fragment
.comp	Compute

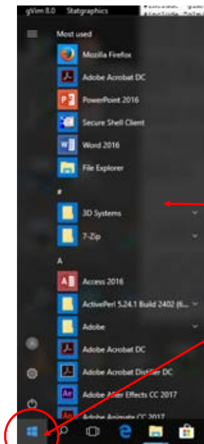
(Can be overridden by the –S option)

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

Windows: glslangValidator.exe
Linux: glslangValidator

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

90



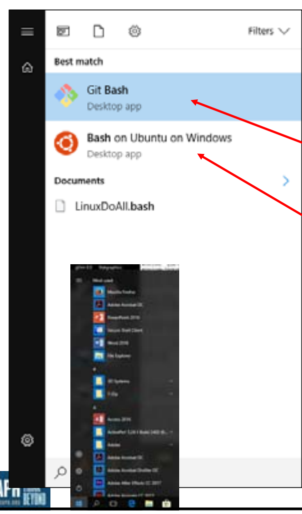
This is only available within 64-bit Windows 10.

2. Type the word *bash*

1. Click on the Microsoft Start icon

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

91



This is only available within 64-bit Windows 10.

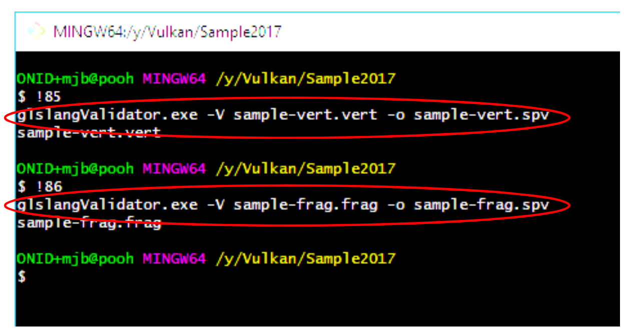
Pick one:

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

- Can get to your personal folders
- Does have make

Running glslangValidator.exe

92



Running glslangValidator.exe

93

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

Compile for Vulkan ("-G" is compile for OpenGL)

The input file. The compiler determines the shader type by the file extension:

.vert	Vertex shader
.tccs	Tessellation Control Shader
.tecs	Tessellation Evaluation Shader
.geom	Geometry shader
.frag	Fragment shader
.comp	Compute shader

Specify the output file

How do you know if SPIR-V compiled successfully?

94

Same as C/C++ -- the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of **0x07230203**

So, if you do an **od –x** on the .spv file, the magic number looks like this:

0203 0723 . . .

Reading a SPIR-V File into a Vulkan Shader Module

95

```
#define SPIRV_MAGIC          0x07230203
. . .
VkResult
Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule )
{
        FILE *fp;
        (void) fopen_s( &fp, filename.c_str(), "rb");
        if( fp == NULL )
        {
                fprintf( FpDebug, "Cannot open shader file '%s'\n", filename.c_str( ) );
                return VK_SHOULD_EXIT;
        }
        uint32_t magic;
        fread( &magic, 4, 1, fp );
        if( magic != SPIRV_MAGIC )
        {
                fprintf( FpDebug, "Magic number for spir-v file '%s is 0x%08x -- should be 0x%08x\n",
                                        filename.c_str( ), magic, SPIRV_MAGIC );
                return VK_SHOULD_EXIT;
        }

        fseek( fp, 0L, SEEK_END );
        int size = ftell( fp );
        rewind( fp );
        unsigned char *code = new unsigned char [size];
        fread( code, size, 1, fp );
        fclose( fp );
```

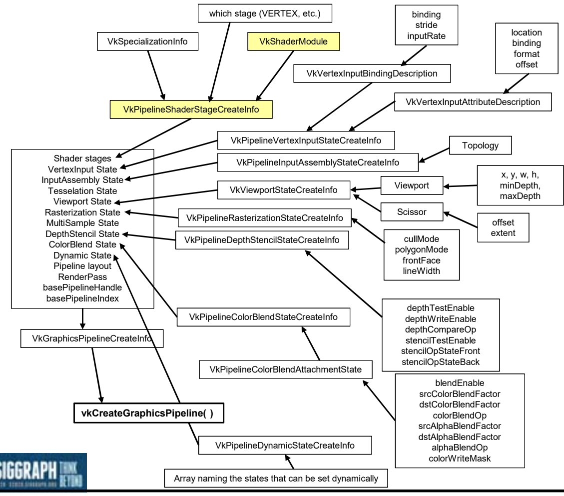

Reading a SPIR-V File into a Shader Module

96

```
VkShaderModule          ShaderModuleVertex;
. . .

        VkShaderModuleCreateInfo        vsmci;
                vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
                vsmci.pNext = nullptr;
                vsmci.flags = 0;
                vsmci.codeSize = size;
                vsmci.pCode = (uint32_t *)code;

        VkResult result = vkCreateShaderModule( LogicalDevice, &vsmci, PALLOCATOR, OUT & ShaderModuleVertex );
        fprintf( FpDebug, "Shader Module '%s' successfully loaded\n", filename.c_str() );
        delete [ ] code;
        return result;
}
```

Vulkan: Creating a Pipeline

97



mjb – July 24, 2020

You can also take a look at SPIR-V Assembly

98

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. ☺

mjb – July 24, 2020

For example, if this is your Shader Source

99

```
#version 400
#extension GL_ARB_separate_shader_objects  : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
      mat4 uModelMatrix;
      mat4 uViewMatrix;
      mat4 uProjectionMatrix;
      mat3 uNormalMatrix;
} Matrices;

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

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


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

void
main( )
{

      mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
      gl_Position = PVM * vec4( aVertex, 1. );

      vNormal   = Matrices.uNormalMatrix * aNormal;
      vColor    = aColor;
      vTexCoord = aTexCoord;
}
```

mjb – July 24, 2020

This is the SPIR-V Assembly, Part I

100

```
                    Capability Shader
          1:        ExtInstImport  "GLSL.std.450"
                    MemoryModel Logical GLSL450
                    EntryPoint Vertex 4  "main" 34 37 48 53 56 57 61 63
                    Source GLSL 400
                    SourceExtension  "GL_ARB_separate_shader_objects"
                    SourceExtension  "GL_ARB_shading_language_420pack"
                    Name 4  "main"
                    Name 10  "PVM"
                    Name 13  "matBuf"
                    MemberName 13(matBuf) 0  "uModelMatrix"
                    MemberName 13(matBuf) 1  "uViewMatrix"
                    MemberName 13(matBuf) 2  "uProjectionMatrix"
                    MemberName 13(matBuf) 3  "uNormalMatrix"
                    Name 15  "Matrices"
                    Name 32  "gl_PerVertex"
                    MemberName 32(gl_PerVertex) 0  "gl_Position"
                    MemberName 32(gl_PerVertex) 1  "gl_PointSize"
                    MemberName 32(gl_PerVertex) 2  "gl_ClipDistance"
                    Name 34  ""
                    Name 37  "aVertex"
                    Name 48  "vNormal"
                    Name 53  "aNormal"
                    Name 56  "vColor"
                    Name 57  "aColor"
                    Name 61  "vTexCoord"
                    Name 63  "aTexCoord"
                    Name 65  "lightBuf"
                    MemberName 65(lightBuf) 0  "uLightPos"
                    Name 67  "Light"
                    MemberDecorate 13(matBuf) 0 ColMajor
                    MemberDecorate 13(matBuf) 0 Offset 0
                    MemberDecorate 13(matBuf) 0 MatrixStride 16
                    MemberDecorate 13(matBuf) 1 ColMajor
                    MemberDecorate 13(matBuf) 1 Offset 64
                    MemberDecorate 13(matBuf) 1 MatrixStride 16
                    MemberDecorate 13(matBuf) 2 ColMajor
                    MemberDecorate 13(matBuf) 2 Offset 128
                    MemberDecorate 13(matBuf) 2 MatrixStride 16
                    MemberDecorate 13(matBuf) 3 ColMajor
                    MemberDecorate 13(matBuf) 3 Offset 192
                    MemberDecorate 13(matBuf) 3 MatrixStride 16
                    Decorate 13(matBuf) Block
                    Decorate 15(Matrices) DescriptorSet 0
```

mjb – July 24, 2020

This is the SPIR-V Assembly, Part II

101

```
#version 400
#extension GL_ARB_separate_shader_objects : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat3 uNormalMatrix;
} Matrices;

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

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

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

void
{

        gl_Position = PVM * vec4( aVertex, 1. );

        vNormal = Matrices.uNormalMatrix * aNormal;
        vColor = aColor;
        vTexCoord = aTexCoord;
}
```

```
                Decorate 15(Matrices) Binding 0
                MemberDecorate 32(gl_PerVertex) 0 BuiltIn Position
                MemberDecorate 32(gl_PerVertex) 1 BuiltIn PointSize
                MemberDecorate 32(gl_PerVertex) 2 BuiltIn ClipDistance
                Decorate 32(gl_PerVertex) Block
                Decorate 37(aVertex) Location 0
                Decorate 48(vNormal) Location 0
                Decorate 53(aNormal) Location 1
                Decorate 56(vColor) Location 1
                Decorate 57(aColor) Location 2
                Decorate 61(vTexCoord) Location 2
                Decorate 63(aTexCoord) Location 3
                MemberDecorate 65(lightBuf) 0 Offset 0
                Decorate 65(lightBuf) Block
                Decorate 67(Light) DescriptorSet 1
                Decorate 67(Light) Binding 0
        2:      TypeVoid
        3:      TypeFunction 2
        6:      TypeFloat 32
        7:      TypeVector 6(float) 4
        8:      TypeMatrix 7(fvec4) 4
        9:      TypePointer Function 8
       11:      TypeVector 6(float) 3
       12:      TypeMatrix 11(fvec3) 3
13(matBuf):        TypeStruct 8 8 8 12
       14:      TypePointer Uniform 13(matBuf)
15(Matrices):   14(ptr) Variable Uniform
       16:      TypeInt 32 1
       17:   16(int) Constant 2
       18:      TypePointer Uniform 8
       21:   16(int) Constant 1
       25:   16(int) Constant 0
       29:      TypeInt 32 0
       30:   29(int) Constant 1
       31:      TypeArray 6(float) 30
32(gl_PerVertex):      TypeStruct 7(fvec4) 6(float) 31
       33:      TypePointer Output 32(gl_PerVertex)
       34:   33(ptr) Variable Output
       36:      TypePointer Input 11(fvec3)
37(aVertex):    36(ptr) Variable Input
       39:   6(float) Constant 1065353216
       45:      TypePointer Output 7(fvec4)
       47:      TypePointer Output 11(fvec3)
48(vNormal):    47(ptr) Variable Output
       49:   16(int) Constant 3
```


This is the SPIR-V Assembly, Part III

102


```
#version 400
#extension GL_ARB_separate_shader_objects : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat3 uNormalMatrix;
} Matrices;

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

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

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

void
{

        gl_Position = PVM * vec4( aVertex, 1. );

        vNormal = Matrices.uNormalMatrix * aNormal;
        vColor = aColor;
        vTexCoord = aTexCoord;
}
```


```
       50:          TypePointer Uniform 12
53(aNormal):    36(ptr) Variable Input
56(vColor):     47(ptr) Variable Output
57(aColor):     36(ptr) Variable Input
       59:          TypeVector 6(float) 2
       60:          TypePointer Output 59(fvec2)
61(vTexCoord):  60(ptr) Variable Output
       62:          TypePointer Input 59(fvec2)
63(aTexCoord):  62(ptr) Variable Input
65(lightBuf):          TypeStruct 7(fvec4)
       66:          TypePointer Uniform 65(lightBuf)
67(Light):      66(ptr) Variable Uniform
4(main):            2 Function None 3
        5:          Label
10(PVM):        9(ptr) Variable Function
       19:   18(ptr) AccessChain 15(Matrices) 17
       20:          8 Load 19
       22:   18(ptr) AccessChain 15(Matrices) 21
       23:          8 Load 22
       24:          8 MatrixTimesMatrix 20 23
       26:   18(ptr) AccessChain 15(Matrices) 25
       27:          8 Load 26
       28:          8 MatrixTimesMatrix 24 27
                    Store 10(PVM) 28
       35:          8 Load 10(PVM)
       38:   11(fvec3) Load 37(aVertex)
       40:   6(float) CompositeExtract 38 0
       41:   6(float) CompositeExtract 38 1
       42:   6(float) CompositeExtract 38 2
       43:   7(fvec4) CompositeConstruct 40 41 42 39
       44:   7(fvec4) MatrixTimesVector 35 43
       46:   45(ptr) AccessChain 34 25
                    Store 46 44
       51:   50(ptr) AccessChain 15(Matrices) 49
       52:         12 Load 51
       54:   11(fvec3) Load 53(aNormal)
       55:   11(fvec3) MatrixTimesVector 52 54
                    Store 48(vNormal) 55
       58:   11(fvec3) Load 57(aColor)
                    Store 56(vColor) 58
       64:   59(fvec2) Load 63(aTexCoord)
                    Store 61(vTexCoord) 64
                    Return
                    FunctionEnd
```


SPIR-V: Printing the Configuration

103

glslangValidator –c

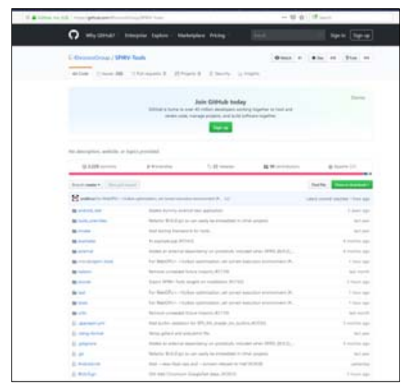
```
MaxLights 32
MaxClipPlanes 6
MaxTextureUnits 32
MaxTextureCoords 32
MaxVertexAttribs 64
MaxVertexUniformComponents 4096
MaxVaryingFloats 64
MaxVertexTextureImageUnits 32
MaxCombinedTextureImageUnits 80
MaxTextureImageUnits 32
MaxFragmentUniformComponents 4096
MaxDrawBuffers 32
MaxVertexUniformVectors 128
MaxVaryingVectors 8
MaxFragmentUniformVectors 16
MaxVertexOutputVectors 16
MaxFragmentInputVectors 15
MinProgramTexelOffset -8
MaxProgramTexelOffset 7
MaxClipDistances 8
MaxComputeWorkGroupCountX 65535
MaxComputeWorkGroupCountY 65535
MaxComputeWorkGroupCountZ 65535
MaxComputeWorkGroupSizeX 1024
MaxComputeWorkGroupSizeY 1024
MaxComputeWorkGroupSizeZ 64
MaxComputeUniformComponents 1024
MaxComputeTextureImageUnits 16
MaxComputeImageUniforms 8
MaxComputeAtomicCounters 8
MaxComputeAtomicCounterBuffers 1
MaxVaryingComponents 60
MaxVertexOutputComponents 64
MaxGeometryInputComponents 64
MaxGeometryOutputComponents 128
MaxFragmentInputComponents 128
MaxImageUnits 8
MaxCombinedImageUnitsAndFragmentOutputs 8
MaxCombinedShaderOutputResources 8
MaxImageSamples 0
MaxVertexImageUniforms 0
MaxTessControlImageUniforms 0
MaxTessEvaluationImageUniforms 0
MaxGeometryImageUniforms 0
MaxFragmentImageUniforms 81
```

```
MaxCombinedImageUniforms 8
MaxGeometryTextureImageUnits 16
MaxGeometryOutputVertices 256
MaxGeometryTotalOutputComponents 1024
MaxGeometryUniformComponents 1024
MaxGeometryVaryingComponents 64
MaxTessControlInputComponents 128
MaxTessControlOutputComponents 128
MaxTessControlTextureImageUnits 16
MaxTessControlUniformComponents 1024
MaxTessControlTotalOutputComponents 4096
MaxTessEvaluationInputComponents 128
MaxTessEvaluationOutputComponents 128
MaxTessEvaluationTextureImageUnits 16
MaxTessEvaluationUniformComponents 1024
MaxTessPatchComponents 120
MaxPatchVertices 32
MaxTessGenLevel 64
MaxViewports 16
MaxVertexAtomicCounters 0
MaxTessControlAtomicCounters 0
MaxTessEvaluationAtomicCounters 0
MaxGeometryAtomicCounters 0
MaxFragmentAtomicCounters 8
MaxCombinedAtomicCounters 8
MaxAtomicCounterBindings 1
MaxVertexAtomicCounterBuffers 0
MaxTessControlAtomicCounterBuffers 0
MaxTessEvaluationAtomicCounterBuffers 0
MaxGeometryAtomicCounterBuffers 0
MaxFragmentAtomicCounterBuffers 1
MaxCombinedAtomicCounterBuffers 1
MaxAtomicCounterBufferSize 16384
MaxTransformFeedbackBuffers 4
MaxTransformFeedbackInterleavedComponents 64
MaxCullDistances 8
MaxCombinedClipAndCullDistances 8
MaxSamples 4
nonInductiveForLoops 1
whileLoops 1
doWhileLoops 1
generalUniformIndexing 1
generalAttributeMatrixVectorIndexing 1
generalVaryingIndexing 1
generalSamplerIndexing 1
generalVariableIndexing 1
generalConstantMatrixVectorIndexing 1
```


SPIR-V: More Information

104

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



A Google-Wrapped Version of glslangValidator

The shaderc project from Google (https://github.com/google/shaderc) provides a glslangValidator wrapper program called ***glslc*** that has a much improved command-line interface. You use, basically, the same way:

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

There are several really nice features. The two I really like are:

1. You can #include files into your shader source
2. You can "#define" definitions on the command line like this:
   glslc.exe --target-env=vulkan -DNUMPONTS=4 sample-vert.vert -o sample-vert.spv

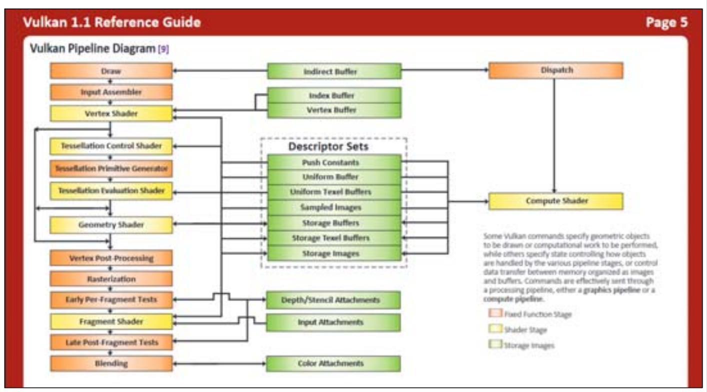
glslc is included in your Sample .zip file

Data Buffers

Mike Bailey
mjb@cs.oregonstate.edu

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

From the Quick Reference Card



Terminology Issues

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

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

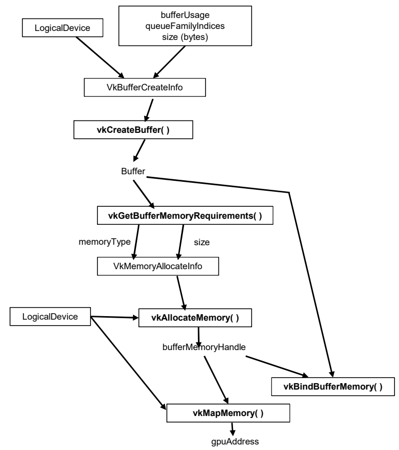
Vulkan calls these things "Buffers". But, Vulkan calls other things "Buffers", too, such as Texture Buffers and Command Buffers. So, I sometimes have taken to calling these things "Data Buffers" and have even gone to far as to override some of Vulkan's own terminology:

typedef VkBuffer VkDataBuffer;

This is probably a bad idea in the long run.

Creating and Filling Vulkan Data Buffers

109



Creating a Vulkan Data Buffer

110

```
VkBuffer  Buffer;

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

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


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

111

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


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

. . .

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

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

. . .

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );

        << do the memory copy >>

result = vkUnmapMemory( LogicalDevice, IN vdm );
```


Finding the Right Type of Memory

112

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

Finding the Right Type of Memory


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


Finding the Right Type of Memory


```
VkPhysicalDeviceMemoryProperties			vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
```

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

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


Sidebar: The Vulkan Memory Allocator (VMA)

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

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

This repository includes a smattering of documentation.

Sidebar: The Vulkan Memory Allocator (VMA)


```
#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"
. . .
VkBufferCreateInfo			vbci;
. . .
VmaAllocationCreateInfo		vaci;
	vaci.physicalDevice = PhysicalDevice;
	vaci.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

117

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

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

. . .

MyBuffer                     MyMatrixUniformBuffer;
```

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

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

mjb – July 24, 2020

Initializing a Data Buffer

118

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

```
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;
. . .
}
```

mjb – July 24, 2020

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

119

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

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

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

mjb – July 24, 2020

Filling those Uniform Variables

120

```
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!

mjb – July 24, 2020

The Parade of Buffer Data

MyBuffer MyMatrixUniformBuffer;

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

```
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;
. . .
}
```

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

Memory-mapped copy operation

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

struct matBuf Matrices;

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

Matrices.uModelMatrix = glm::mat4( );        // identity

Matrices.uViewMatrix = glm::lookAt( eye, look, up );

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

Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```

uniform matBuf Matrices;

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


Filling the Data Buffer


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

. . .

MyBuffer         MyMatrixUniformBuffer;
```

```
Init05UniformBuffer( sizeof(Matrices),     OUT &MyMatrixUniformBuffer );

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

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

Matrices.uModelMatrix = glm::mat4( );        // identity

Matrices.uViewMatrix = glm::lookAt( eye, look, up );

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

Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix ) );
```


Creating and Filling the Data Buffer – the Details


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

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

    VkMemoryAllocateInfo        vmai;
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        vmai.pNext = nullptr;
        vmai.allocationSize = vmr.size;
        vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );

    VkDeviceMemory              vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;

    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```


Creating and Filling the Data Buffer – the Details


```
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;
}
```

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

Creating and Filling the Data Buffer – the Details

125

```
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;
}
```

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

mjb – July 24, 2020

126

GLFW

Mike Bailey
mjb@cs.oregonstate.edu

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

mjb – July 24, 2020

127

http://www.glfw.org/

mjb – July 24, 2020

Setting Up GLFW

128

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

uint32_t                Width, Height;
VkSurfaceKHR            Surface;
        . . .

void
InitGLFW( )
{
        glfwInit( );
        if( ! glfwVulkanSupported( ) )
        {
                fprintf( stderr, "Vulkan is not supported on this system!\n" );
                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 );
}
```

mjb – July 24, 2020

You Can Also Query What Vulkan Extensions GLFW Requires


```
uint32_t count;
const char ** extensions = glfwGetRequiredInstanceExtensions (&count);

fprintf( FpDebug, "\nFound %d GLFW Required Instance Extensions:\n", count );

for( uint32_t  i = 0; i < count; i++ )
{
        fprintf( FpDebug, "\t%s\n", extensions[ i ] );
}
```

```
Found 2 GLFW Required Instance Extensions:
        VK_KHR_surface
        VK_KHR_win32_surface
```


GLFW Keyboard Callback


```
void
GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
        if( action == GLFW_PRESS )
        {
                switch( key )
                {
                        //case GLFW_KEY_M:
                        case 'm':
                        case 'M':
                                Mode++;
                                if( Mode >= 2 )
                                        Mode = 0;
                                break;

                        default:
                                fprintf( FpDebug, "Unknow key hit: 0x%04x = '%c'\n", key, key );
                                fflush(FpDebug);
                }
        }
}
```


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\n", button );
        }

        // button down sets the bit, up clears the bit:
        if( action == GLFW_PRESS )
        {
                double xpos, ypos;
                glfwGetCursorPos( window, &xpos, &ypos);
                Xmouse = (int)xpos;
                Ymouse = (int)ypos;
                ActiveButton |= b;              // set the proper bit
        }
        else
        {
                ActiveButton &= ~b;             // clear the proper bit
        }
}
```


GLFW Mouse Motion Callback


```
void
GLFWMouseMotion( GLFWwindow *window, double xpos, double ypos )
{
        int dx = (int)xpos - Xmouse;            // change in mouse coords
        int dy = (int)ypos - Ymouse;

        if( ( ActiveButton & LEFT ) != 0 )
        {
                Xrot += ( ANGFACT*dy );
                Yrot += ( ANGFACT*dx );
        }

        if( ( ActiveButton & MIDDLE ) != 0 )
        {
                Scale += SCLFACT * (float) ( dx - dy );

                // keep object from turning inside-out or disappearing:

                if( Scale < MINSCALE )
                        Scale = MINSCALE;
        }

        Xmouse = (int)xpos;                     // new current position
        Ymouse = (int)ypos;
}
```

Looping and Closing GLFW

133

```
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

mjb – July 24, 2020

Looping and Closing GLFW

134

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( );

mjb – July 24, 2020

135

GLM

Mike Bailey

mjb@cs.oregonstate.edu

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

mjb – July 24, 2020

What is GLM?

136

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

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

You can find it at:

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

You invoke GLM like this:

```
#define   GLM_FORCE_RADIANS

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

OpenGL treats all angles as given in *degrees*. This line forces GLM to treat all angles as given in *radians*. I recommend this so that *all* angles you create in *all* programming will be in radians.

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

mjb – July 24, 2020

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

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

The Most Useful GLM Variables, Operations, and Functions

// constructor:

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

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

// multiplications:

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

// emulating OpenGL transformations *with concatenation*:

glm::mat4 **glm::rotate**(glm::mat4 const & m, float angle, glm::vec3 const & axis);

glm::mat4 **glm::scale**(glm::mat4 const & m, glm::vec3 const & factors);

glm::mat4 **glm::translate**(glm::mat4 const & m, glm::vec3 const & translation);

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);

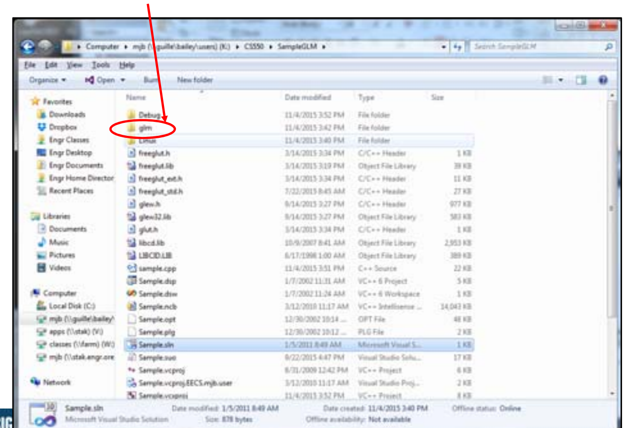
glm::mat4 **glm::frustum**(float left, float right, float bottom, float top, float near, float far);
glm::mat4 **glm::perspective**(float fovy, float aspect, float near, float far);

// viewing (assign, not concatenate):

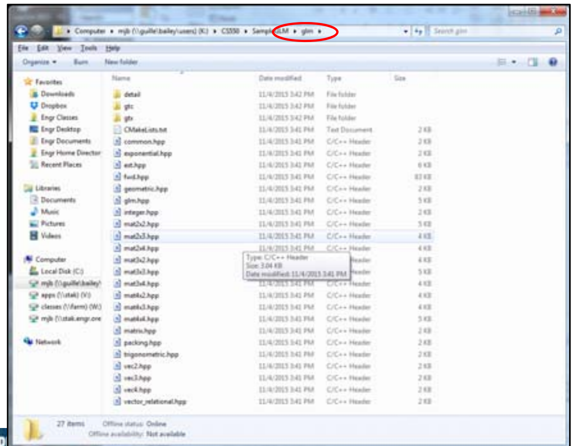
glm::mat4 **glm::lookAt**(glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up);

Installing GLM into your own space

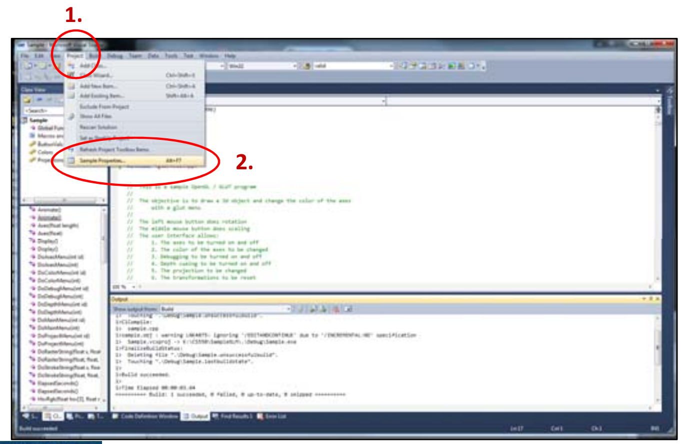
I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.



Here's what that GLM folder looks like

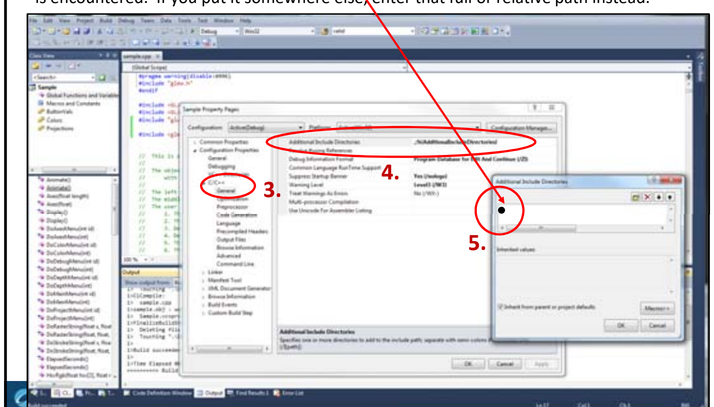


Telling Visual Studio about where the GLM folder is



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 );
```

How Does this Matrix Stuff Really Work?

$$x' = Ax + By + Cz + D$$
$$y' = Ex + Fy + Gz + H$$
$$z' = Ix + Jy + Kz + L$$

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.

Or, in matrix form:

$$\begin{array}{r} \\ \\ \text{x' producing row} \\ \text{y' producing row} \\ \text{z' producing row} \\ \\ \end{array} \begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = \begin{array}{cccc} \text{x consuming column} & \text{y consuming column} & \text{z consuming column} & \text{constant column} \\ \end{array} \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

Transformation Matrices

Translation

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

Rotation about X

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

Scaling

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

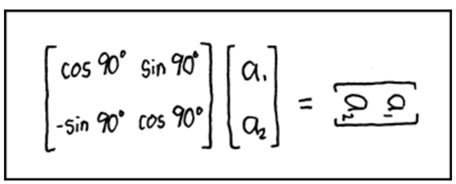
Rotation about Y

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

Rotation about Z

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

How it Really Works :-)



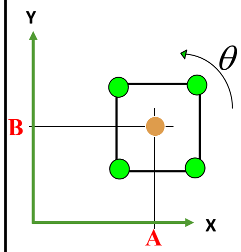
http://xkcd.com

The Rotation Matrix for an Angle (θ) about an Arbitrary Axis (Ax, Ay, Az)

$$[M] = \begin{bmatrix} A_xA_x + \cos\theta(1 - A_xA_x) & A_xA_y - \cos\theta(A_xA_y) - \sin\theta A_z & A_xA_z - \cos\theta(A_xA_z) + \sin\theta A_y \\ A_yA_x - \cos\theta(A_yA_x) + \sin\theta A_z & A_yA_y + \cos\theta(1 - A_yA_y) & A_yA_z - \cos\theta(A_yA_z) - \sin\theta A_x \\ A_zA_x - \cos\theta(A_zA_x) - \sin\theta A_y & A_zA_y - \cos\theta(A_zA_y) + \sin\theta A_x & A_zA_z + \cos\theta(1 - A_zA_z) \end{bmatrix}$$

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)?

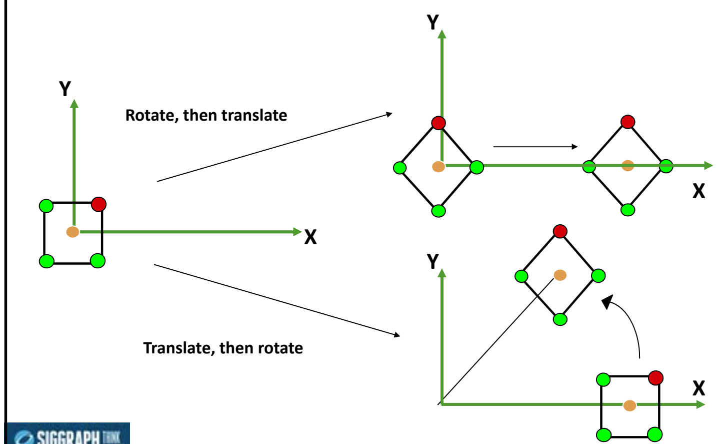
A: We create more than one matrix.

Write it →

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \left(\overset{\boxed{3}}{\left[T_{+A,+B} \right]} \cdot \left(\overset{\boxed{2}}{\left[R_\theta \right]} \cdot \left(\overset{\boxed{1}}{\left[T_{-A,-B} \right]} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \right) \right) \right)$$

← Say it

Matrix Multiplication *is not* Commutative



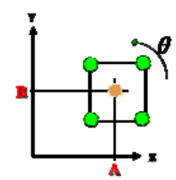
Matrix Multiplication *is* Associative

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \left(\left[T_{+A,+B} \right] \cdot \left(\left[R_\theta \right] \cdot \left(\left[T_{-A,-B} \right] \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \right) \right) \right)$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \underbrace{\left(\left[T_{+A,+B} \right] \cdot \left[R_\theta \right] \cdot \left[T_{-A,-B} \right] \right)}_{} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

One matrix –
the Current Transformation Matrix, or CTM

One Matrix to Rule Them All



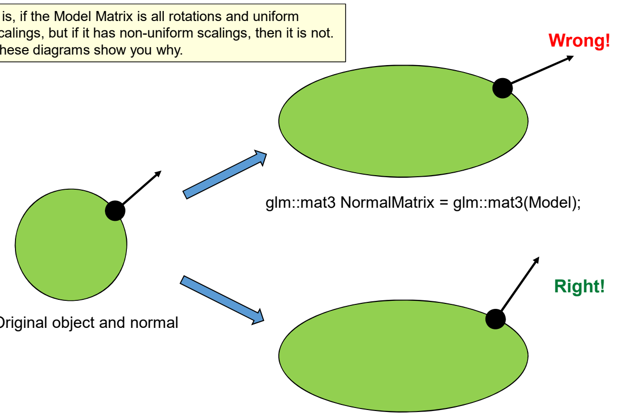
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \left(\left[T_{+A,+B} \right] \cdot \left[R_\theta \right] \cdot \left[T_{-A,-B} \right] \right) \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

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

glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);     glm::vec3  up(0.,1.,0.);
glm::mat4 View   = glm::lookAt( eye, look, up );

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

. . .
glm::mat3 Matrix = Projection * View * Model;
glm::mat3 NormalMatrix = glm::inverseTranspose(  glm::mat3(Model)  );
```

153

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

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



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

mjb – July 24, 2020

154

Instancing

Mike Bailey

mjb@cs.oregonstate.edu

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

mjb – July 24, 2020

155

Instancing – What and why?

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

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

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

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

mjb – July 24, 2020

156

Making each Instance look differently -- Approach #1

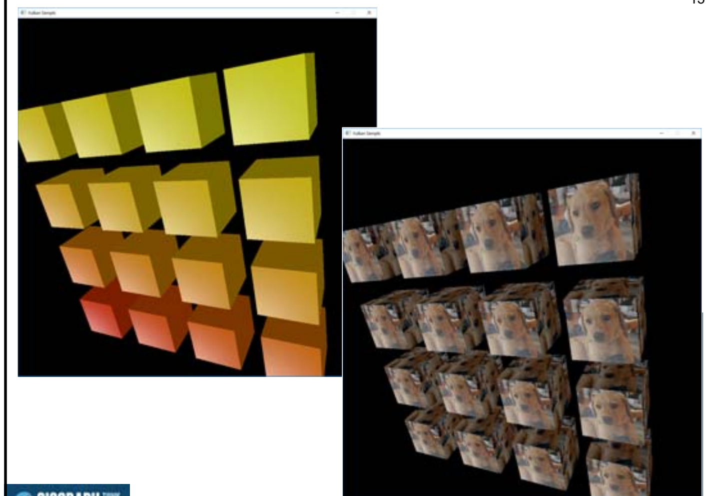
Use the built-in vertex shader variable **gl_InstanceIndex** to define a unique display property, such as position or color.

gl_InstanceIndex starts at 0

In the vertex shader:

```
out vec3 vColor;
const int    NUMINSTANCES = 16;
const float DELTA             =   3.0;

float xdelta = DELTA * float( gl_InstanceIndex % 4 );
float ydelta = DELTA * float( gl_InstanceIndex / 4 );
vColor = vec3( 1., 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]
```

mjb – July 24, 2020



Making each Instance look differently -- Approach #2

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]
```


Vulkan.

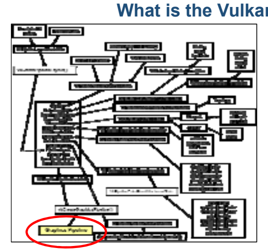
The Graphics Pipeline Data Structure

Mike Bailey

mjb@cs.oregonstate.edu

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

What is the Vulkan Graphics Pipeline?

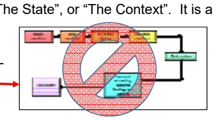


Don't worry if this is too small to read – a larger version is coming up.

There is also a Vulkan **Compute Pipeline Data Structure** – we will get to that later.

Here's what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call "The State", or "The Context". It is a ***data structure***.
2. The Vulkan Graphics Pipeline is *not* the processes that OpenGL would call "the graphics pipeline".

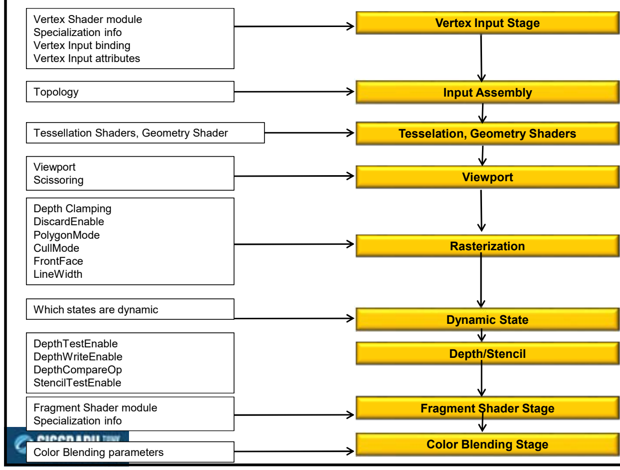


3. For the most part, the Vulkan Graphics Pipeline Data Structure is immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

Graphics Pipeline Stages and what goes into Them

161

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



mjb – July 24, 2020

The First Step: Create the Graphics Pipeline Layout

162

The Graphics Pipeline Layout is fairly static. Only the layout of the Descriptor Sets and information on the Push Constants need 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;
}
```

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

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.

mjb – July 24, 2020

A Pipeline Data Structure Contains the Following State Items:

163

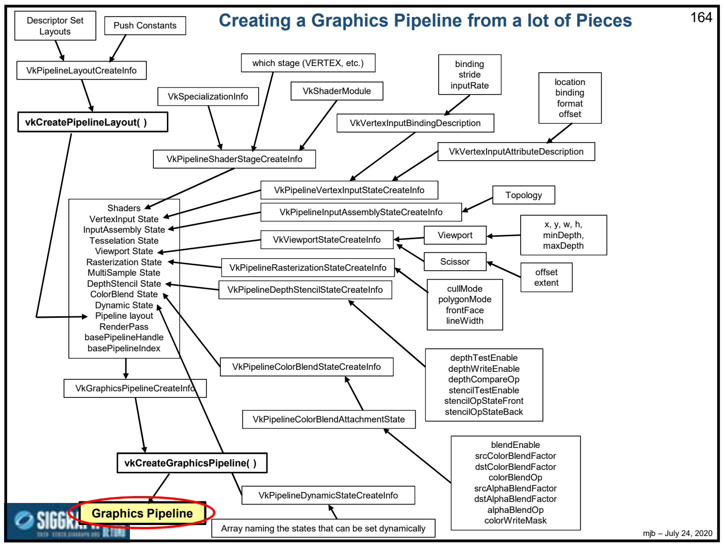
- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- ***Viewport***: x, y, w, h, minDepth, maxDepth
- ***Scissoring***: x, y, w, h
- Rasterization: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

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

mjb – July 24, 2020

Creating a Graphics Pipeline from a lot of Pieces

164



mjb – July 24, 2020

Creating a Typical Graphics Pipeline

165

```
VkResult
Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader,
                                      VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
#ifdef ASSUMPTIONS
        vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
        vprsci.depthClampEnable = VK_FALSE;
        vprsci.rasterizerDiscardEnable = VK_FALSE;
        vprsci.polygonMode = VK_POLYGON_MODE_FILL;
        vprsci.cullMode = VK_CULL_MODE_NONE;    // best to do this because of the projectionMatrix[1][1] *= -1.;
        vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
        vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_ONE_BIT;
        vpcbas.blendEnable = VK_FALSE;
        vpcbsci.logicOpEnable = VK_FALSE;
        vpdssci.depthTestEnable = VK_TRUE;
        vpdssci.depthWriteEnable = VK_TRUE;
        vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
#endif

    . . .
```

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

The Shaders to Use

166

```
VkPipelineShaderStageCreateInfo         vpssci[2];
        vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        vpssci[0].pNext = nullptr;
        vpssci[0].flags = 0;
        vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
        vpssci[0].module = vertexShader;
        vpssci[0].pName = "main";
        vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
#ifdef BITS
VK_SHADER_STAGE_VERTEX_BIT
VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
VK_SHADER_STAGE_GEOMETRY_BIT
VK_SHADER_STAGE_FRAGMENT_BIT
VK_SHADER_STAGE_COMPUTE_BIT
VK_SHADER_STAGE_ALL_GRAPHICS
VK_SHADER_STAGE_ALL
#endif
        vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
        vpssci[1].pNext = nullptr;
        vpssci[1].flags = 0;
        vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
        vpssci[1].module = fragmentShader;
        vpssci[1].pName = "main";
        vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    VkVertexInputBindingDescription         vvibd[1];   // an array containing one of these per buffer being used
        vvibd[0].binding = 0;       // which binding # this is
        vvibd[0].stride = sizeof( struct vertex );      // bytes between successive
        vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
#ifdef CHOICES
VK_VERTEX_INPUT_RATE_VERTEX
VK_VERTEX_INPUT_RATE_INSTANCE
#endif
```

Use one **vpssci** array member per shader module you are using

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

Link in the Per-Vertex Attributes

167

```
VkVertexInputAttributeDescription       vviad[4];       // an array containing one of these per vertex attribute in all bindings
        // 4 = vertex, normal, color, texture coord
        vviad[0].location = 0;          // location in the layout
        vviad[0].binding = 0;           // which binding description this is part of
        vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
        vviad[0].offset = offsetof( struct vertex, position );          // 0
#ifdef EXTRAS_DEFINED_AT_THE_TOP
// these are here for convenience and readability:
#define VK_FORMAT_VEC4          VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_XYZW          VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_VEC3          VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_STP           VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_XYZ           VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_VEC2          VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_ST            VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_XY            VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_FLOAT         VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_S             VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_X             VK_FORMAT_R32_SFLOAT
#endif
        vviad[1].location = 1;
        vviad[1].binding = 0;
        vviad[1].format = VK_FORMAT_VEC3;       // nx, ny, nz
        vviad[1].offset = offsetof( struct vertex, normal );            // 12

        vviad[2].location = 2;
        vviad[2].binding = 0;
        vviad[2].format = VK_FORMAT_VEC3;       // r, g, b
        vviad[2].offset = offsetof( struct vertex, color );             // 24

        vviad[3].location = 3;
        vviad[3].binding = 0;
        vviad[3].format = VK_FORMAT_VEC2;       // s, t
        vviad[3].offset = offsetof( struct vertex, texCoord );          // 36
```

Use one **vviad** array member per element in the struct for the array-of-structures element you are using as vertex input

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

168

```
VkPipelineVertexInputStateCreateInfo    vpvisci;        // used to describe the input vertex attributes
        vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
        vpvisci.pNext = nullptr;
        vpvisci.flags = 0;
        vpvisci.vertexBindingDescriptionCount = 1;
        vpvisci.pVertexBindingDescriptions = vvibd;
        vpvisci.vertexAttributeDescriptionCount = 4;
        vpvisci.pVertexAttributeDescriptions = vviad;

    VkPipelineInputAssemblyStateCreateInfo  vpiasci;
        vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
        vpiasci.pNext = nullptr;
        vpiasci.flags = 0;
        vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;;
#ifdef CHOICES
VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
#endif
        vpiasci.primitiveRestartEnable = VK_FALSE;

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

    VkPipelineGeometryStateCreateInfo       vpgsci;
        vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
        vptsci.pNext = nullptr;
        vptsci.flags = 0;
```

Declare the binding descriptions and attribute descriptions

Declare the vertex topology

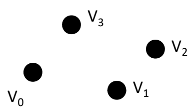
Tessellation Shader info

Geometry Shader info

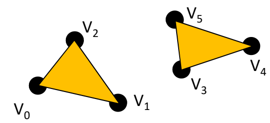
Options for vpiasci.topology

169

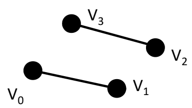
VK_PRIMITIVE_TOPOLOGY_POINT_LIST



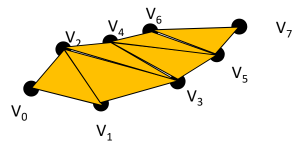
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST



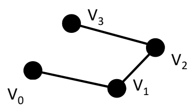
VK_PRIMITIVE_TOPOLOGY_LINE_LIST



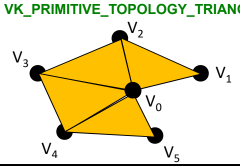
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP



VK_PRIMITIVE_TOPOLOGY_LINE_STRIP



VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN



SIGGRAPH

mjb – July 24, 2020

What is "Primitive Restart Enable"?

170

```
vpiasci.primitiveRestartEnable = VK_FALSE;
```

"Restart Enable" is used with:
- Indexed drawing.
- Triangle Fan and *Strip topologies

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

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

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

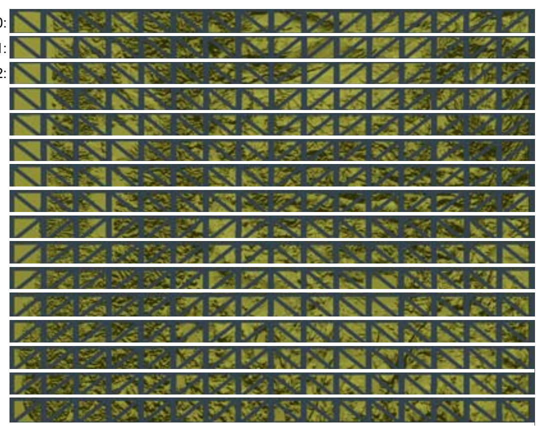
SIGGRAPH

mjb – July 24, 2020

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

171

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



SIGGRAPH

mjb – July 24, 2020

172

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

VkRect2D                            vr;
        vr.offset.x = 0;
        vr.offset.y = 0;
        vr.extent.width = Width;
        vr.extent.height = Height;

VkPipelineViewportStateCreateInfo        vpvsci;
        vpvsci.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
        vpvsci.pNext = nullptr;
        vpvsci.flags = 0;
        vpvsci.viewportCount = 1;
        vpvsci.pViewports = &vv;
        vpvsci.scissorCount = 1;
        vpvsci.pScissors = &vr;
```

Declare the viewport information

Declare the scissoring information

Group the viewport and scissor information together

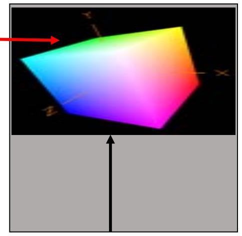
SIGGRAPH

mjb – July 24, 2020

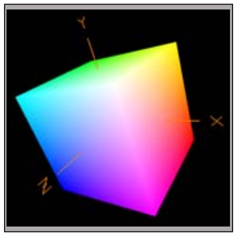
What is the Difference Between Changing the Viewport and Changing the Scissoring?

173

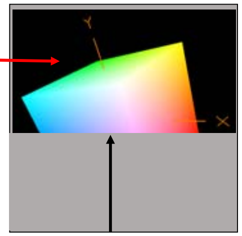
Viewport:
Viewporting operates on ***vertices*** and takes place right before the rasterizer. Changing the vertical part of the **viewport** causes the entire scene to get scaled (scrunched) into the viewport area.



Original Image



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



mjb – July 24, 2020

Setting the Rasterizer State

174

```
VkPipelineRasterizationStateCreateInfo          vprsci;
        vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
        vprsci.pNext = nullptr;
        vprsci.flags = 0;
        vprsci.depthClampEnable = VK_FALSE;
        vprsci.rasterizerDiscardEnable = VK_FALSE;
        vprsci.polygonMode = VK_POLYGON_MODE_FILL;
#ifdef CHOICES
VK_POLYGON_MODE_FILL
VK_POLYGON_MODE_LINE
VK_POLYGON_MODE_POINT
#endif
        vprsci.cullMode = VK_CULL_MODE_NONE;     // recommend this because of the projMatrix[1][1] *= -1.;
#ifdef CHOICES
VK_CULL_MODE_NONE
VK_CULL_MODE_FRONT_BIT
VK_CULL_MODE_BACK_BIT
VK_CULL_MODE_FRONT_AND_BACK_BIT
#endif
        vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
#ifdef CHOICES
VK_FRONT_FACE_COUNTER_CLOCKWISE
VK_FRONT_FACE_CLOCKWISE
#endif
        vprsci.depthBiasEnable = VK_FALSE;
        vprsci.depthBiasConstantFactor = 0.f;
        vprsci.depthBiasClamp = 0.f;
        vprsci.depthBiasSlopeFactor = 0.f;
        vprsci.lineWidth = 1.f;
```

Declare information about how the rasterization will take place

mjb – July 24, 2020

What is “Depth Clamp Enable”?

175

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

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

A good use for this is **Polygon Capping**:

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

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

mjb – July 24, 2020

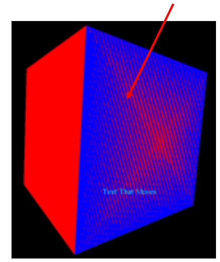
What is “Depth Bias Enable”?

176

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

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

Z-fighting



mjb – July 24, 2020

MultiSampling State

177

```
VkPipelineMultisampleStateCreateInfo          vpmsci;
        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.

mjb – July 24, 2020

Color Blending State for each Color Attachment *

178

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

```
VkPipelineColorBlendAttachmentState          vpcbas;
        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;
```

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

$$Color_{new} = (1.-\alpha) * Color_{existing} + \alpha * Color_{incoming}$$

$$0. \leq \alpha \leq 1.$$

*A "Color Attachment" is a framebuffer to be rendered into.
You can have as many of these as you want.

mjb – July 24, 2020

Raster Operations for each Color Attachment

179

```
VkPipelineColorBlendStateCreateInfo          vpcbsci;
        vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
        vpcbsci.pNext = nullptr;
        vpcbsci.flags = 0;
        vpcbsci.logicOpEnable = VK_FALSE;
        vpcbsci.logicOp = VK_LOGIC_OP_COPY;
#ifdef CHOICES
VK_LOGIC_OP_CLEAR
VK_LOGIC_OP_AND
VK_LOGIC_OP_AND_REVERSE
VK_LOGIC_OP_COPY
VK_LOGIC_OP_AND_INVERTED
VK_LOGIC_OP_NO_OP
VK_LOGIC_OP_XOR
VK_LOGIC_OP_OR
VK_LOGIC_OP_NOR
VK_LOGIC_OP_EQUIVALENT
VK_LOGIC_OP_INVERT
VK_LOGIC_OP_OR_REVERSE
VK_LOGIC_OP_COPY_INVERTED
VK_LOGIC_OP_OR_INVERTED
VK_LOGIC_OP_NAND
VK_LOGIC_OP_SET
#endif
        vpcbsci.attachmentCount = 1;
        vpcbsci.pAttachments = &vpcbas;
        vpcbsci.blendConstants[0] = 0;
        vpcbsci.blendConstants[1] = 0;
        vpcbsci.blendConstants[2] = 0;
        vpcbsci.blendConstants[3] = 0;
```

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

mjb – July 24, 2020

Which Pipeline Variables can be Set Dynamically

180

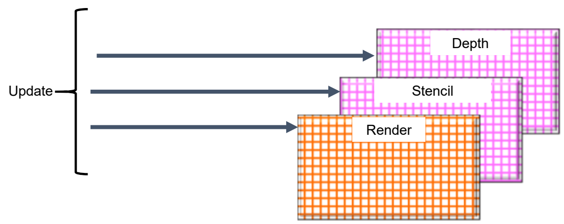
Just used as an example in the Sample Code

```
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;
        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;
```

mjb – July 24, 2020

The Stencil Buffer

181



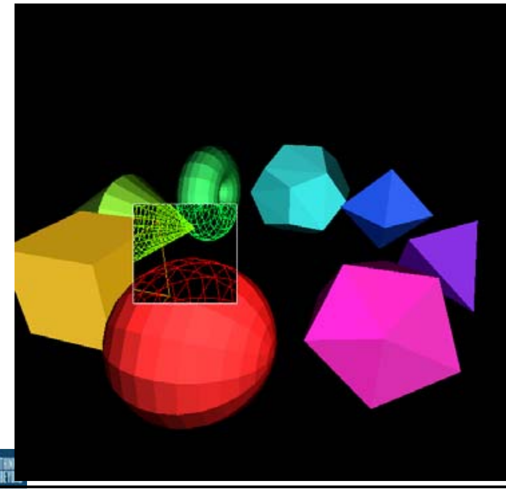
Here's how the Stencil Buffer works:

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

mjb – July 24, 2020

Using the Stencil Buffer to Create a *Magic Lens*

182

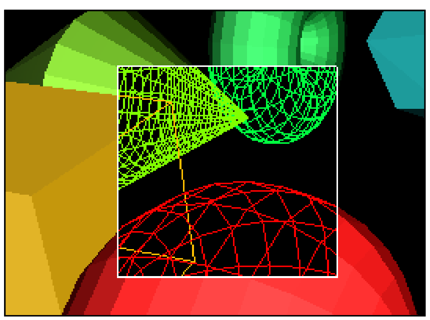


mjb – July 24, 2020

Using the Stencil Buffer to Create a *Magic Lens*

183

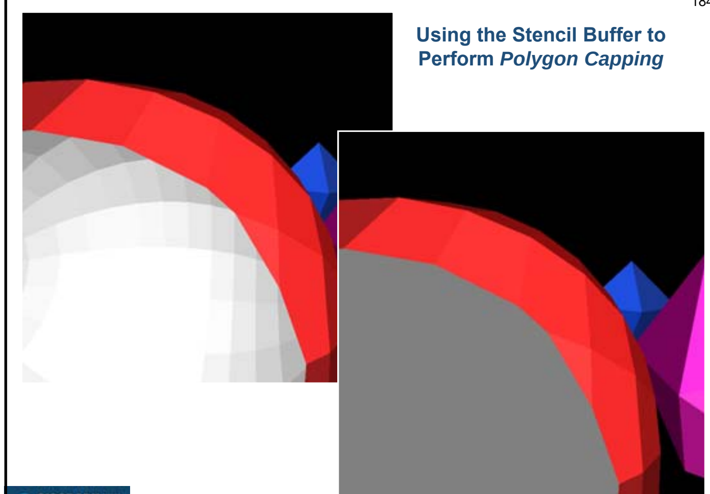
1. Clear the SB = 0
2. Write protect the color buffer
3. Fill a square, setting SB = 1
4. Write-enable the color buffer
5. Draw the solids wherever SB == 0
6. Draw the wireframes wherever SB == 1



mjb – July 24, 2020

Using the Stencil Buffer to Perform *Polygon Capping*

184

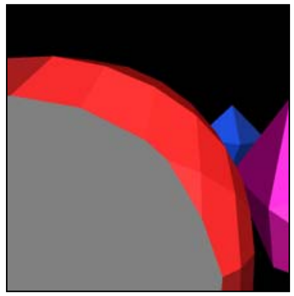


mjb – July 24, 2020

Using the Stencil Buffer to Perform *Polygon Capping*

185

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

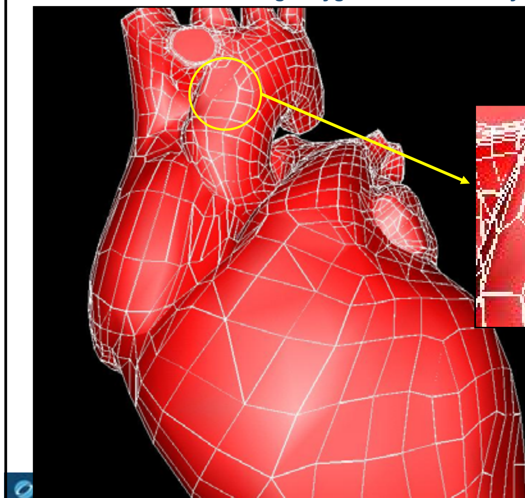


mjb – July 24, 2020

Outlining Polygons the Naïve Way

186

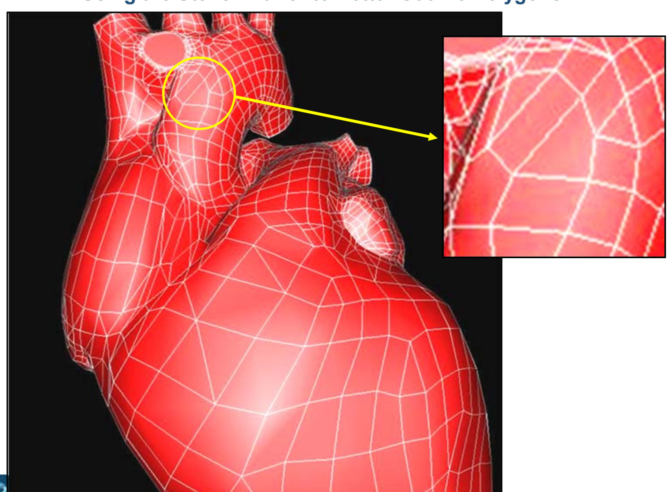
1. Draw the polygons
2. Draw the edges



mjb – July 24, 2020

Using the Stencil Buffer to Better Outline Polygons

187



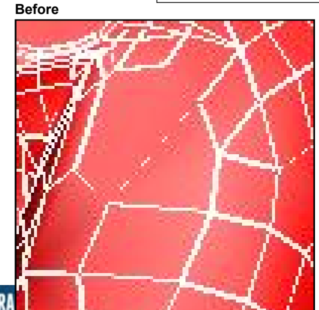
mjb – July 24, 2020

Using the Stencil Buffer to Better Outline Polygons

188

```
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
}
```

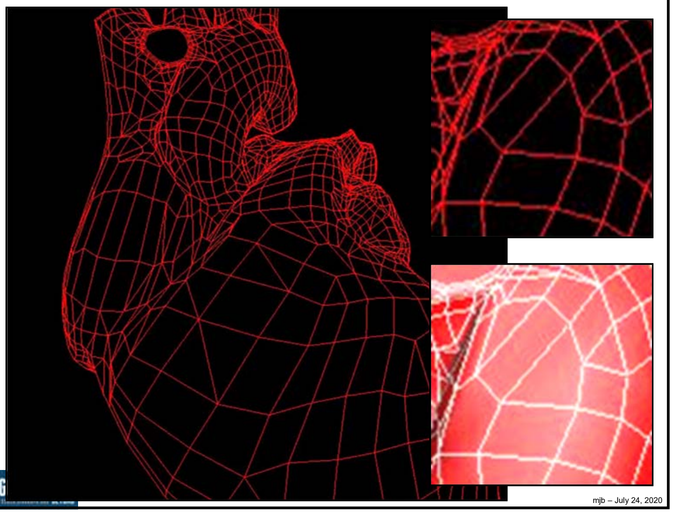
Before



After

Using the Stencil Buffer to Perform *Hidden Line Removal*

189



mjb – July 24, 2020

Stencil Operations for Front and Back Faces

190

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

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

mjb – July 24, 2020

Operations for Depth Values

191

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

mjb – July 24, 2020

Putting it all Together! (finally…)

192

```
VkPipeline   GraphicsPipeline;

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

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

    return result;
}
```

Group all of the individual state information and create the pipeline

mjb – July 24, 2020

Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

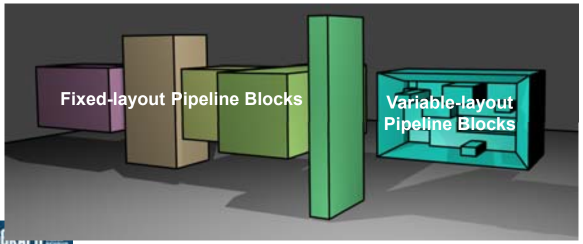

```
vkCmdBindPipeline( CommandBuffers[nextImageIndex],
                   VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```


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

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

```
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:



Vulkan.

Descriptor Sets

Mike Bailey

mjb@cs.oregonstate.edu

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

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?

```
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;
```

What are Descriptor Sets?

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

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

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


Descriptor Sets

Our example will assume the following shader uniform variables:

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

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

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

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


Descriptor Sets

CPU:

Uniform data created in a C++ data structure

- Knows the CPU data structure
- Knows where the data starts
- Knows the data's size

GPU:

Uniform data in a "blob"*

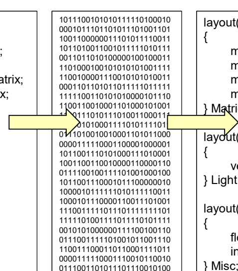
- Knows where the data starts
- Knows the data's size
- Doesn't know the CPU or GPU data structure

GPU:

Uniform data used in the shader

- Knows the shader data structure
- Doesn't know where each piece of data starts

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

struct lightBuf
{
        glm::vec4 uLightPos;
};

struct miscBuf
{
        float uTime;
        int   uMode;
};
```

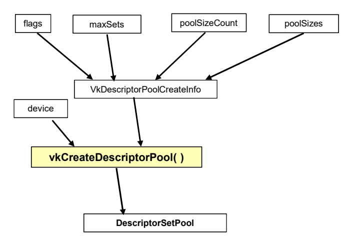


```
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat3 uNormalMatrix;
} Matrices;

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

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

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

* "binary large object"

Step 1: Descriptor Set Pools

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



201

```
VkResult
Init13DescriptorSetPool( )
{
	VkResult result;

	VkDescriptorPoolSize		vdps[4];
		vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
		vdps[0].descriptorCount = 1;
		vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
		vdps[1].descriptorCount = 1;
		vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
		vdps[2].descriptorCount = 1;
		vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
		vdps[3].descriptorCount = 1;

#ifdef CHOICES
VK_DESCRIPTOR_TYPE_SAMPLER
VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
#endif

	VkDescriptorPoolCreateInfo		vdpci;
		vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
		vdpci.pNext = nullptr;
		vdpci.flags = 0;
		vdpci.maxSets = 4;
		vdpci.poolSizeCount = 4;
		vdpci.pPoolSizes = &vdps[0];

	result = vkCreateDescriptorPool( LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);
	return result;
}
```


202

Step 2: Define the Descriptor Set Layouts

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

https://www.britishmuseum.org

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

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

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

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

MatrixSet DS Layout Binding:	LightSet DS Layout Binding:	MiscSet DS Layout Binding:	TexSamplerSet DS Layout Binding:
binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)
set = 0	set = 1	set = 2	set = 3

203

```
VkResult
Init13DescriptorSetLayouts( )
{
	VkResult result;

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

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

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

	// DS #3:
	VkDescriptorSetLayoutBinding		TexSamplerSet[1];
		TexSamplerSet[0].binding		= 0;
		TexSamplerSet[0].descriptorType	= VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
		TexSamplerSet[0].descriptorCount	= 1;
		TexSamplerSet[0].stageFlags		= VK_SHADER_STAGE_FRAGMENT_BIT;
		TexSamplerSet[0].pImmutableSamplers = (VkSampler *)nullptr;
```

```
uniform sampler2D   uSampler;
vec4 rgba = texture( uSampler, vST );
```


204

Step 2: Define the Descriptor Set Layouts

MatrixSet DS Layout Binding:	LightSet DS Layout Binding:	MiscSet DS Layout Binding:	TexSamplerSet DS Layout Binding:
binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)	binding descriptorType descriptorCount pipeline stage(s)
set = 0	set = 1	set = 2	set = 3

vdslc0 DS Layout CI:	vdslc1 DS Layout CI:	vdslc2 DS Layout CI:	vdslc3 DS Layout CI:
bindingCount type number of that type pipeline stage(s)	bindingCount type number of that type pipeline stage(s)	bindingCount type number of that type pipeline stage(s)	bindingCount type number of that type pipeline stage(s)

Array of Descriptor Set Layouts

Pipeline Layout

205

```
	VkDescriptorSetLayoutCreateInfo		vdslc0;
		vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
		vdslc0.pNext = nullptr;
		vdslc0.flags = 0;
		vdslc0.bindingCount = 1;
		vdslc0.pBindings = &MatrixSet[0];

	VkDescriptorSetLayoutCreateInfo		vdslc1;
		vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
		vdslc1.pNext = nullptr;
		vdslc1.flags = 0;
		vdslc1.bindingCount = 1;
		vdslc1.pBindings = &LightSet[0];

	VkDescriptorSetLayoutCreateInfo		vdslc2;
		vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
		vdslc2.pNext = nullptr;
		vdslc2.flags = 0;
		vdslc2.bindingCount = 1;
		vdslc2.pBindings = &MiscSet[0];

	VkDescriptorSetLayoutCreateInfo		vdslc3;
		vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
		vdslc3.pNext = nullptr;
		vdslc3.flags = 0;
		vdslc3.bindingCount = 1;
		vdslc3.pBindings = &TexSamplerSet[0];

	result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc0, PALLOCATOR, OUT &DescriptorSetLayouts[0] );
	result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc1, PALLOCATOR, OUT &DescriptorSetLayouts[1] );
	result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc2, PALLOCATOR, OUT &DescriptorSetLayouts[2] );
	result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc3, PALLOCATOR, OUT &DescriptorSetLayouts[3] );

	return result;
}
```

mjb – July 24, 2020

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

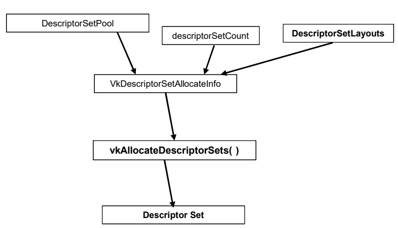
206

```
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;
}
```

mjb – July 24, 2020

Step 4: Allocating the Memory for Descriptor Sets

207



mjb – July 24, 2020

Step 4: Allocating the Memory for Descriptor Sets

208

```
VkResult
Init13DescriptorSets( )
{
	VkResult result;

	VkDescriptorSetAllocateInfo		vdsai;
		vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
		vdsai.pNext = nullptr;
		vdsai.descriptorPool = DescriptorPool;
		vdsai.descriptorSetCount = 4;
		vdsai.pSetLayouts = DescriptorSetLayouts;

	result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );
```

mjb – July 24, 2020

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

209

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

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

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

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

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

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

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

This struct identifies what texture sampler and image view it owns

mjb – July 24, 2020

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

210

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

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

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

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

mjb – July 24, 2020

Step 5: Tell the Descriptor Sets where their data is

211

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

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

uint32_t copyCount = 0;

// this could have been done with one call and an array of VkWriteDescriptorSets:

vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet *)nullptr );
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet *)nullptr );
```

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

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

mjb – July 24, 2020

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

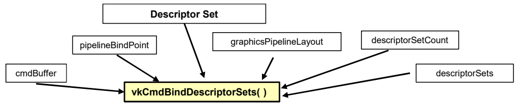
212

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

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

mjb – July 24, 2020

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

213



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

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

Sidebar: The Entire Collection of Descriptor Set Paths

214

VkDescriptorPoolCreateInfo
vkCreateDescriptorPool( )
Create the pool of Descriptor Sets for future use

VkDescriptorSetLayoutBinding
VkDescriptorSetLayoutCreateInfo
vkCreateDescriptorSetLayout( )
vkCreatePipelineLayout( )
Describe a particular Descriptor Set layout and use it in a specific Pipeline layout

VkDescriptorSetAllocateInfo
vkAllocateDescriptorSets( )
Allocate memory for particular Descriptor Sets

VkDescriptorBufferInfo
VkDescriptorImageInfo
VkWriteDescriptorSet
Tell a particular Descriptor Set where its CPU data is
vkUpdateDescriptorSets( )
Re-write CPU data into a particular Descriptor Set

vkCmdBindDescriptorSets( )
Make a particular Descriptor Set "current" for rendering

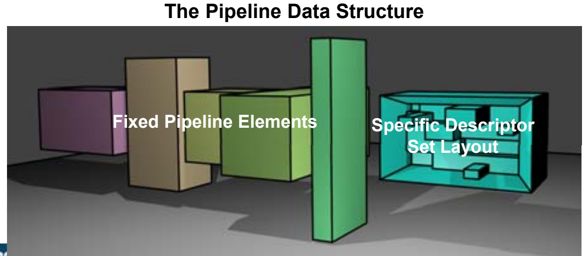
Sidebar: Why Do Descriptor Sets Need to Provide Layout Information to the Pipeline Data Structure?

215

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.

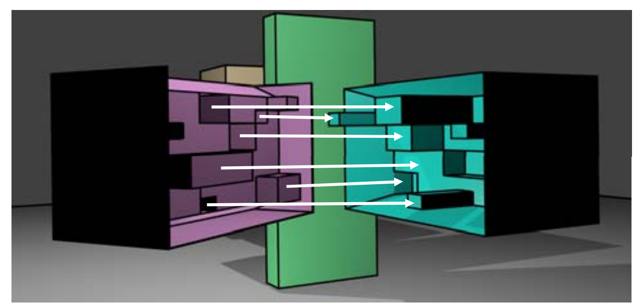
The Pipeline Data Structure



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

216

Any set of data that matches the Descriptor Set Layout can be plugged in there.



Textures

Mike Bailey

mjb@cs.oregonstate.edu

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

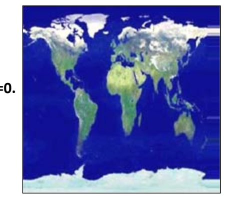
The Basic Idea

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

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

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

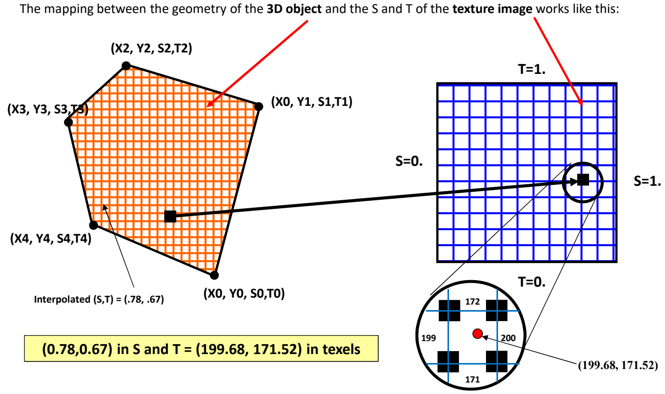
T=1.

S=0.  S=1.

T=0.

The Basic Idea

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



(0.78,0.67) in S and T = (199.68, 171.52) in texels

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.

In OpenGL terms: assigning an (s,t) to each vertex

Enable texture mapping:

glEnable(GL_TEXTURE_2D);

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

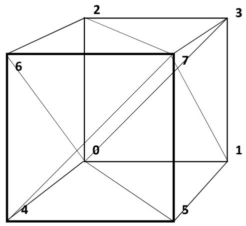
```
glBegin( GL_POLYGON );
    glTexCoord2f( s0, t0 );
    glNormal3f( nx0, ny0, nz0 );
    glVertex3f( x0, y0, z0 );

    glTexCoord2f( s1, t1 );
    glNormal3f( nx1, ny1, nz1 );
    glVertex3f( x1, y1, z1 );

    . . .
glEnd( );
```

Disable texture mapping:

glDisable(GL_TEXTURE_2D);

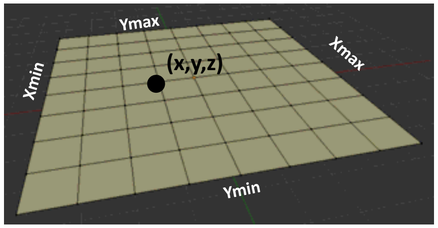
Triangles in an Array of Structures


```
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. }
        },
```



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

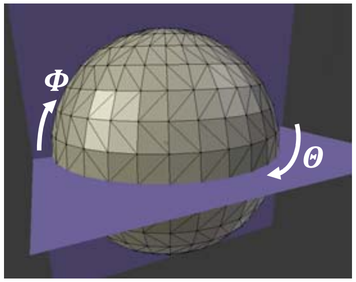
The easiest way to figure out what s and t are at a particular vertex is to figure out what *fraction* across the object the vertex is living at. For a plane,



$$s = \frac{x - Xmin}{Xmax - Xmin} \qquad t = \frac{y - Ymin}{Ymax - Ymin}$$

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

Or, for a sphere,

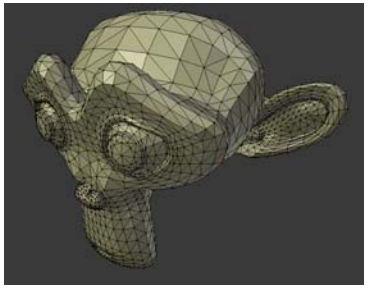


$$s = \frac{\Theta - (-\pi)}{2\pi} \qquad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}$$

```
s = ( lng + M_PI    ) / ( 2.*M_PI );
t  = ( lat + M_PI/2. ) / M_PI;
```

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

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



$s = ?$ $\qquad$ $t = ?$

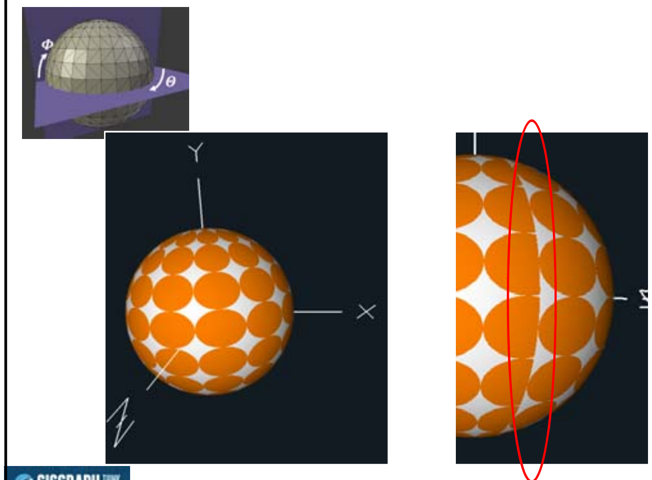
You really are at the mercy of whoever did the modeling…

225

mjb – July 24, 2020

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

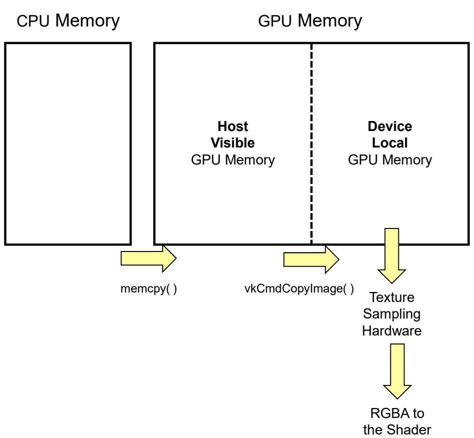
226



mjb – July 24, 2020

Memory Types

227



mjb – July 24, 2020

Memory Types

228

NVIDIA Discrete Graphics:

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

Intel Integrated Graphics:

3 Memory Types:
Memory 0: DeviceLocal
Memory 1: DeviceLocal HostVisible HostCoherent
Memory 2: DeviceLocal HostVisible HostCoherent HostCached

mjb – July 24, 2020

Texture Sampling Parameters

229

OpenGL

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

Vulkan

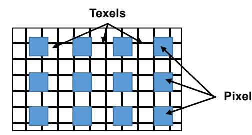
```
VkSamplerCreateInfo                vsci;
        vsci.magFilter = VK_FILTER_LINEAR;
        vsci.minFilter = VK_FILTER_LINEAR;
        vsci.mipmapMode    = VK_SAMPLER_MIPMAP_MODE_LINEAR;
        vsci.addressModeU  = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeV  = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

        . . .

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

mjb – July 24, 2020

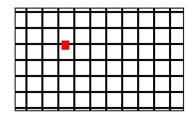
Textures' Undersampling Artifacts

230

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.



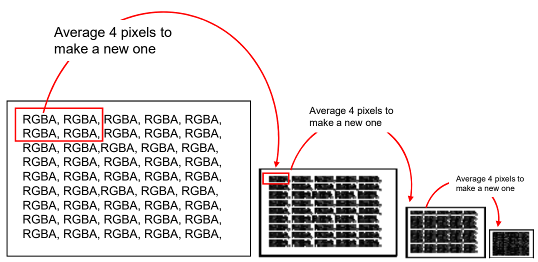
Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all *white*, with the red texel having never been included in the final image. The solution is to create lower-resolutions of the same texture so that the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



mjb – July 24, 2020

Texture Mip*-mapping

231



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

* Latin: *multim in parvo*, "many things in a small place"

mjb – July 24, 2020

232

```
VkResult
Init07TextureSampler( MyTexture * pMyTexture )
{
    VkResult result;

    VkSamplerCreateInfo                vsci;
        vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
        vsci.pNext = nullptr;
        vsci.flags = 0;
        vsci.magFilter = VK_FILTER_LINEAR;
        vsci.minFilter = VK_FILTER_LINEAR;
        vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
        vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
        vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
#ifdef CHOICES
VK_SAMPLER_ADDRESS_MODE_REPEAT
VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
#endif
        vsci.mipLodBias = 0.;
        vsci.anisotropyEnable = VK_FALSE;
        vsci.maxAnisotropy = 1.;
        vsci.compareEnable = VK_FALSE;
        vsci.compareOp = VK_COMPARE_OP_NEVER;
#ifdef CHOICES
VK_COMPARE_OP_NEVER
VK_COMPARE_OP_LESS
VK_COMPARE_OP_EQUAL
VK_COMPARE_OP_LESS_OR_EQUAL
VK_COMPARE_OP_GREATER
VK_COMPARE_OP_NOT_EQUAL
VK_COMPARE_OP_GREATER_OR_EQUAL
VK_COMPARE_OP_ALWAYS
#endif
        vsci.minLod = 0.;
        vsci.maxLod = 0.;
        vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
#ifdef CHOICES
VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
VK_BORDER_COLOR_INT_OPAQUE_BLACK
VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE
VK_BORDER_COLOR_INT_OPAQUE_WHITE
#endif
        vsci.unnormalizedCoordinates = VK_FALSE;      // VK_TRUE means we are use raw texels as the index
                                                      // VK_FALSE means we are using the usual 0. - 1.

    result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler );
```

mjb – July 24, 2020

233

```
VkResult
Init07TextureBuffer( INOUT MyTexture * pMyTexture)
{
	VkResult result;

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

	VkImage  stagingImage;
	VkImage  textureImage;

	// ************************************************************************************
	// this first {...} is to create the staging image:
	// ************************************************************************************
	{
		VkImageCreateInfo			vici;
			vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
			vici.pNext = nullptr;
			vici.flags = 0;
			vici.imageType = VK_IMAGE_TYPE_2D;
			vici.format = VK_FORMAT_R8G8B8A8_UNORM;
			vici.extent.width = texWidth;
			vici.extent.height = texHeight;
			vici.extent.depth = 1;
			vici.mipLevels = 1;
			vici.arrayLayers = 1;
			vici.samples = VK_SAMPLE_COUNT_1_BIT;
			vici.tiling = VK_IMAGE_TILING_LINEAR;
#ifdef CHOICES
VK_IMAGE_TILING_OPTIMAL
VK_IMAGE_TILING_LINEAR
#endif
			vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
#ifdef CHOICES
VK_IMAGE_USAGE_TRANSFER_SRC_BIT
VK_IMAGE_USAGE_TRANSFER_DST_BIT
VK_IMAGE_USAGE_SAMPLED_BIT
VK_IMAGE_USAGE_STORAGE_BIT
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
#endif
			vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

mjb – July 24, 2020

234

```
#ifdef CHOICES
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_PREINITIALIZED
#endif
			vici.queueFamilyIndexCount = 0;
			vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

		VkMemoryRequirements			vmr;
		vkGetImageMemoryRequirements( LogicalDevice, IN stagingImage, OUT &vmr);

		if (Verbose)
		{
			fprintf(FpDebug, "Image vmr.size = %lld\n", vmr.size);
			fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
			fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
			fflush(FpDebug);
		}

		VkMemoryAllocateInfo			vmai;
			vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
			vmai.pNext = nullptr;
			vmai.allocationSize = vmr.size;
			vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();   // because we want to mmap it

		VkDeviceMemory			vdm;
		result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
		pMyTexture->vdm = vdm;

		result = vkBindImageMemory( LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset

		// we have now created the staging image -- fill it with the pixel data:

		VkImageSubresource			vis;
			vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
			vis.mipLevel = 0;
			vis.arrayLayer = 0;

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

		if (Verbose)
		{
			fprintf(FpDebug, "Subresource Layout:\n");
			fprintf(FpDebug, "\toffset = %lld\n", vsl.offset);
			fprintf(FpDebug, "\tsize = %lld\n", vsl.size);
			fprintf(FpDebug, "\trowPitch = %lld\n", vsl.rowPitch);
			fprintf(FpDebug, "\tarrayPitch = %lld\n", vsl.arrayPitch);
			fprintf(FpDebug, "\tdepthPitch = %lld\n", vsl.depthPitch);
			fflush(FpDebug);
		|
```

mjb – July 24, 2020

235

```
		void * gpuMemory;
		vkMapMemory( LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
						// 0 and 0 = offset and memory map flags

		if (vsl.rowPitch == 4 * texWidth)
		{
			memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
		}
		else
		{
			unsigned char *gpuBytes = (unsigned char *)gpuMemory;
			for (unsigned int y = 0; y < texHeight; y++)
			{
				memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
			}
		}

		vkUnmapMemory( LogicalDevice, vdm);

	}
	// ************************************************************************************
```

mjb – July 24, 2020

236

```
	// ************************************************************************************
	// this second {...} is to create the actual texture image:
	// ************************************************************************************
	{
		VkImageCreateInfo			vici;
			vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
			vici.pNext = nullptr;
			vici.flags = 0;
			vici.imageType = VK_IMAGE_TYPE_2D;
			vici.format = VK_FORMAT_R8G8B8A8_UNORM;
			vici.extent.width  = texWidth;
			vici.extent.height = texHeight;
			vici.extent.depth = 1;
			vici.mipLevels = 1;
			vici.arrayLayers = 1;
			vici.samples = VK_SAMPLE_COUNT_1_BIT;
			vici.tiling = VK_IMAGE_TILING_OPTIMAL;
			vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
					// because we are transferring into it and will eventual sample from it
			vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
			vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
			vici.queueFamilyIndexCount = 0;
			vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

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

		VkMemoryRequirements			vmr;
		vkGetImageMemoryRequirements( LogicalDevice, IN textureImage, OUT &vmr);

		if( Verbose )
		{
			fprintf( FpDebug, "Texture vmr.size = %lld\n", vmr.size );
			fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment );
			fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );
			fflush( FpDebug );
		}
		VkMemoryAllocateInfo			vmai;
			vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
			vmai.pNext = nullptr;
			vmai.allocationSize = vmr.size;
			vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal( );  // because we want to sample from it

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

		result = vkBindImageMemory( LogicalDevice, IN textureImage, IN vdm, 0 );		// 0 = offset
	}
	// ************************************************************************************
```

mjb – July 24, 2020

237

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

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

result = vkBeginCommandBuffer( TextureCommandBuffer, IN &vcbbi);

// ****************************************************************************
// transition the staging buffer layout:
// ****************************************************************************
{
        VkImageSubresourceRange            visr;
                visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
                visr.baseMipLevel = 0;
                visr.levelCount = 1;
                visr.baseArrayLayer = 0;
                visr.layerCount = 1;

        VkImageMemoryBarrier            vimb;
                vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
                vimb.pNext = nullptr;
                vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
                vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
                vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.image = stagingImage;
                vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
                vimb.dstAccessMask = 0;
                vimb.subresourceRange = visr;

        vkCmdPipelineBarrier( TextureCommandBuffer,
                        VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0,
                        0, (VkMemoryBarrier *)nullptr,
                        0, (VkBufferMemoryBarrier *)nullptr,
                        1, IN &vimb );
}
// ****************************************************************************
```


238

```
// ****************************************************************************
// transition the texture buffer layout:
// ****************************************************************************
{
        VkImageSubresourceRange            visr;
                visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
                visr.baseMipLevel = 0;
                visr.levelCount = 1;
                visr.baseArrayLayer = 0;
                visr.layerCount = 1;

        VkImageMemoryBarrier            vimb;
                vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
                vimb.pNext = nullptr;
                vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
                vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
                vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.image = textureImage;
                vimb.srcAccessMask = 0;
                vimb.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
                vimb.subresourceRange = visr;

        vkCmdPipelineBarrier( TextureCommandBuffer,
                VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
                0, (VkMemoryBarrier *)nullptr,
                0, (VkBufferMemoryBarrier *)nullptr,
                1, IN &vimb);

        // now do the final image transfer:

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

        VkOffset3D                        vo3;
                vo3.x = 0;
                vo3.y = 0;
                vo3.z = 0;

        VkExtent3D                        ve3;
                ve3.width = texWidth;
                ve3.height = texHeight;
                ve3.depth = 1;
```


239

```
        VkImageCopy                        vic;
                vic.srcSubresource = visl;
                vic.srcOffset = vo3;
                vic.dstSubresource = visl;
                vic.dstOffset = vo3;
                vic.extent = ve3;

        vkCmdCopyImage(TextureCommandBuffer,
                stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
                textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);
}
// ****************************************************************************
```


240

```
// ****************************************************************************
// transition the texture buffer layout a second time:
// ****************************************************************************
        {
        VkImageSubresourceRange            visr;
                visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
                visr.baseMipLevel = 0;
                visr.levelCount = 1;
                visr.baseArrayLayer = 0;
                visr.layerCount = 1;

        VkImageMemoryBarrier            vimb;
                vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
                vimb.pNext = nullptr;
                vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
                vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
                vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
                vimb.image = textureImage;
                vimb.srcAccessMask = 0;
                vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
                vimb.subresourceRange = visr;

        vkCmdPipelineBarrier(TextureCommandBuffer,
                VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
                0, (VkMemoryBarrier *)nullptr,
                0, (VkBufferMemoryBarrier *)nullptr,
                1, IN &vimb);
        }
        // ****************************************************************************

        result = vkEndCommandBuffer( TextureCommandBuffer );

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

        result = vkQueueSubmit( Queue, 1, IN &vsi, VK_NULL_HANDLE );
        result = vkQueueWaitIdle( Queue );
```

241

```
// 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, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);

return result;
}
```

Access to an Image ⟷ Image View ⟷ The Actual Image Data

8 bits Red	8 bits Green	8 bits Blue	8 bits Alpha

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

Reading in a Texture from a BMP File

242

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

•••

MyTexture       MyPuppyTexture;
```

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

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

243

Vulkan.

Queues and Command Buffers

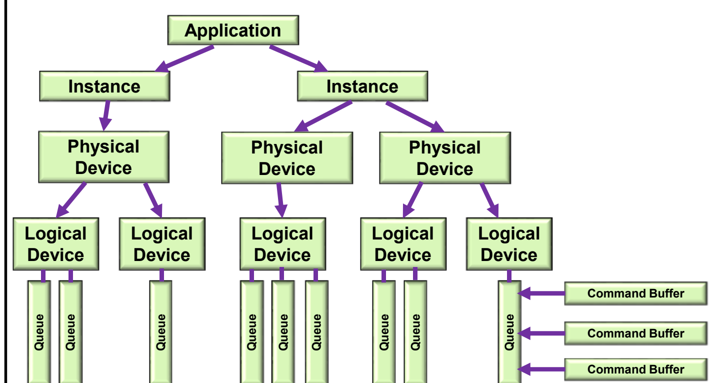
Mike Bailey

mjb@cs.oregonstate.edu

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

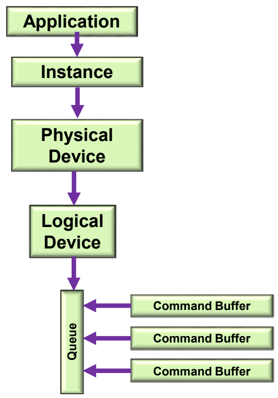
Vulkan: Overall Block Diagram

244



Simplified Block Diagram

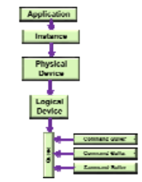
245

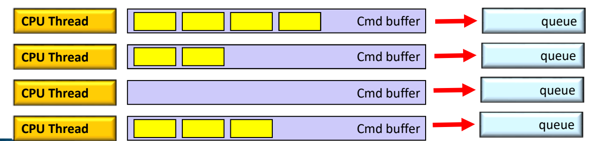


Vulkan Queues and Command Buffers

246

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





Querying what Queue Families are Available

247

```
uint32_t  count;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr );

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceFamilyProperties(  PhysicalDevice, &count, OUT &vqfp, );

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

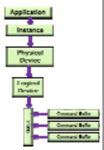
```
Found 3 Queue Families:
    0: Queue Family Count = 16  ;    Graphics Compute Transfer
    1: Queue Family Count =  1  ;    Transfer
    2: Queue Family Count =  8  ;    Compute
```


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

248

```
int
FindQueueFamilyThatDoesGraphics( )
{
        uint32_t count = -1;
        vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr );

        VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
        vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, IN &count, OUT vqfp );

        for( unsigned int i = 0; i < count; i++ )
        {
                if( ( vqfp[ i ].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
                        return i;
        }
        return -1;
}
```

Creating a Logical Device Needs to Know Queue Family Information

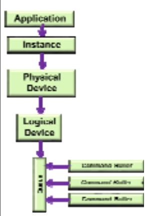


```
float  queuePriorities[ ] =
{
        1.              // one entry per queueCount
};

VkDeviceQueueCreateInfo  vdqci[1];
        vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
        vdqci[0].pNext = nullptr;
        vdqci[0].flags = 0;
        vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
        vdqci[0].queueCount = 1;
        vdqci[0].queuePriorities = (float *) queuePriorities;

VkDeviceCreateInfo  vdci;
        vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
        vdci.pNext = nullptr;
        vdci.flags = 0;
        vdci.queueCreateInfoCount = 1;                  // # of device queues wanted
        vdci.pQueueCreateInfos = IN &vdqci[0];          // array of VkDeviceQueueCreateInfo's
        vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
        vdci.ppEnabledLayerNames = myDeviceLayers;
        vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
        vdci.ppEnabledExtensionNames = myDeviceExtensions;
        vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures;     // already created

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

VkQueue Queue;
uint32_t  queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
uint32_t  queueIndex = 0;

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


Creating the Command Pool as part of the Logical Device


```
VkResult
Init06CommandPool( )
{
    VkResult result;

    VkCommandPoolCreateInfo          vcpci;
        vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
        vcpci.pNext = nullptr;
        vcpci.flags =       VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
                          | VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
#ifdef CHOICES
VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
#endif
        vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );

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

    return result;
}
```


Creating the Command Buffers

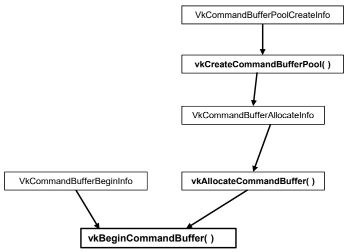



```
VkResult
Init06CommandBuffers( )
{
    VkResult result;

    // allocate 2 command buffers for the double-buffered rendering:

    {
        VkCommandBufferAllocateInfo          vcbai;
            vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
            vcbai.pNext = nullptr;
            vcbai.commandPool = CommandPool;
            vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
            vcbai.commandBufferCount = 2;         // 2, because of double-buffering

        result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
    }

    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:

    {
        VkCommandBufferAllocateInfo          vcbai;
            vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
            vcbai.pNext = nullptr;
            vcbai.commandPool = CommandPool;
            vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
            vcbai.commandBufferCount = 1;

        result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );
    }

    return result;
}
```


Beginning a Command Buffer – One per Image


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

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

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

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

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

        . . .

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

Beginning a Command Buffer



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


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


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


```
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers,
bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers,
            bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```



```
VkResult
RenderScene( )
{
    VkResult result;
    VkSemaphoreCreateInfo           vsci;
      vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
      vsci.pNext = nullptr;
      vsci.flags = 0;

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

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

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

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

257

```
VkClearColorValue                    vccv;
        vccv.float32[0] = 0.0;
        vccv.float32[1] = 0.0;
        vccv.float32[2] = 0.0;
        vccv.float32[3] = 1.0;

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

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

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

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

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

mjb – July 24, 2020

258

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

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

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

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

vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
                         GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
                                                      // dynamic offset count, dynamic offsets
vkCmdBindPushConstants( CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values );

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

VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );        // 0, 1 = firstBinding, bindingCount

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

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

24, 2020

Submitting a Command Buffer to a Queue for Execution

259

```
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;
        vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
```

mjb – July 24, 2020

The Entire Submission / Wait / Display Process

260

```
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 );
```

uly 24, 2020

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 Specification says:

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

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

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

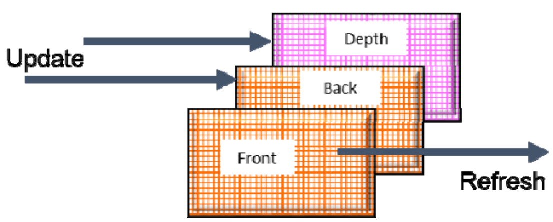
The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.

The Swap Chain

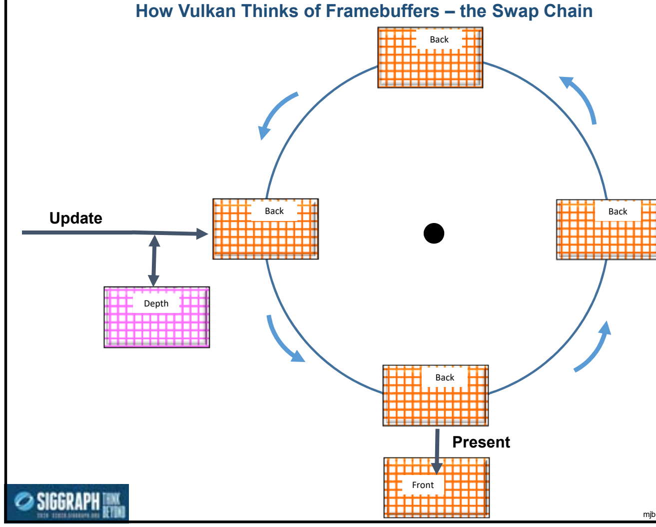
Mike Bailey
mjb@cs.oregonstate.edu

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

How OpenGL Thinks of Framebuffers



How Vulkan Thinks of Framebuffers – the Swap Chain



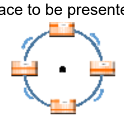
What is a Swap Chain?

265

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



Swap Chains are tightly coupled to the window system.

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

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

mjb – July 24, 2020

We Need to Find Out What our Display Capabilities Are

266

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

        . . .

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


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

        . . .

uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (VkPresentModeKHR *) nullptr );
VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, presentModes );
fprintf( FpDebug, "\nFound %d Present Modes:\n", presentModeCount );

        . . .
```

mjb – July 24, 2020

We Need to Find Out What our Display Capabilities Are

267

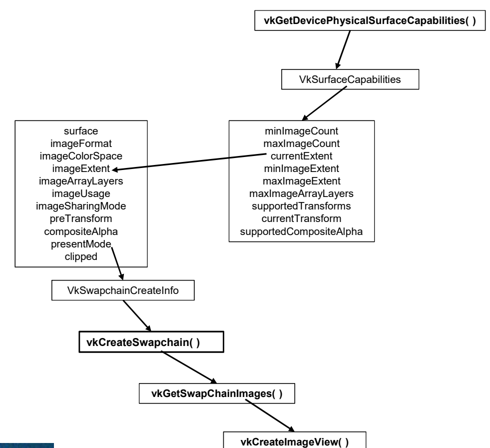
VulkanDebug.txt output:

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

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


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


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

mjb – July 24, 2020

Creating a Swap Chain

268



mjb – July 24, 2020

Creating a Swap Chain


```
VkSurfaceCapabilitiesKHR        vsc;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
VkExtent2D surfaceRes = vsc.currentExtent;

VkSwapchainCreateInfoKHR        vscci;
        vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
        vscci.pNext = nullptr;
        vscci.flags = 0;
        vscci.surface = Surface;
        vscci.minImageCount = 2;            // double buffering
        vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
        vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
        vscci.imageExtent.width = surfaceRes.width;
        vscci.imageExtent.height = surfaceRes.height;
        vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
        vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
        vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
        vscci.imageArrayLayers = 1;
        vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vscci.queueFamilyIndexCount = 0;
        vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
        vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
        vscci.oldSwapchain = VK_NULL_HANDLE;
        vscci.clipped = VK_TRUE;

result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
```


Creating the Swap Chain Images and Image Views


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

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

// present views for the double-buffering:

PresentImageViews = new VkImageView[ imageCount ];

for( unsigned int i = 0; i < imageCount; i++ )
{
    VkImageViewCreateInfo       vivci;
        vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        vivci.pNext = nullptr;
        vivci.flags = 0;
        vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
        vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        vivci.subresourceRange.baseMipLevel = 0;
        vivci.subresourceRange.levelCount = 1;
        vivci.subresourceRange.baseArrayLayer = 0;
        vivci.subresourceRange.layerCount = 1;
        vivci.image = PresentImages[ i ];

    result = vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[ i ] );
}
```

Rendering into the Swap Chain, I


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

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

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

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

    . . .

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

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

    . . .

vkCmdEndRenderPass( CommandBuffers[ nextImageIndex ] );
vkEndCommandBuffer(  CommandBuffers[ nextImageIndex ] );
```


Rendering into the Swap Chain, II


```
VkFenceCreateInfo           vfci;
        vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
        vfci.pNext = nullptr;
        vfci.flags = 0;

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

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

    . . .

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

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

Rendering into the Swap Chain, III


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

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 );
```


Push Constants

Mike Bailey

mjb@cs.oregonstate.edu

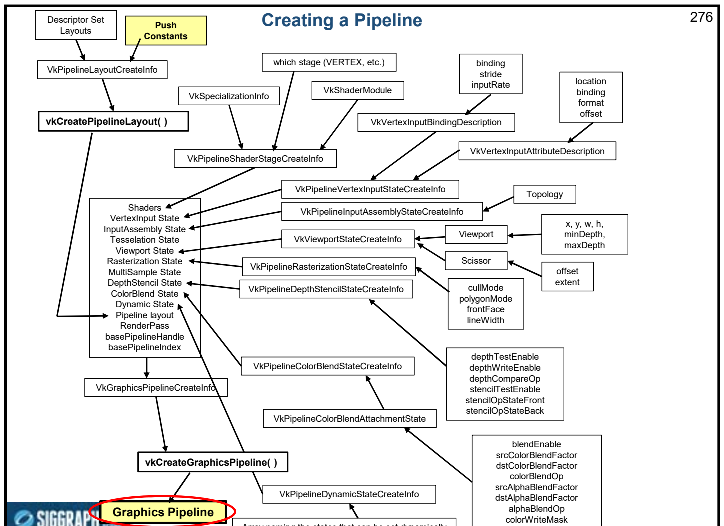
http://cs.oregonstate.edu/~mjb/vulkan

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.

Creating a Pipeline



Push Constants

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

```
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:

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

where:
stageFlags are or'ed bits of VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, etc.

size is in bytes

pValues is a void * pointer to the data, which, in this 4x4 matrix example, would be of type **glm::mat4**.

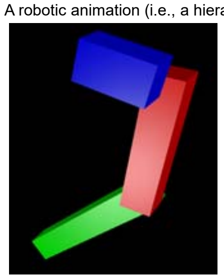
Setting up the Push Constants for the Pipeline Structure

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

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

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

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


An Robotic Example using Push Constants

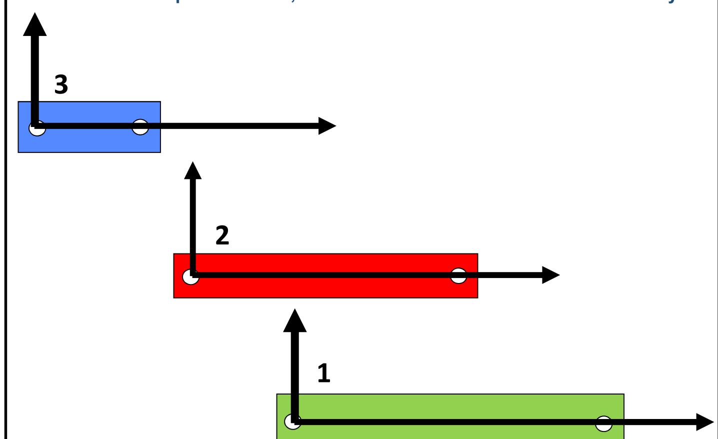
A robotic animation (i.e., a hierarchical transformation system)



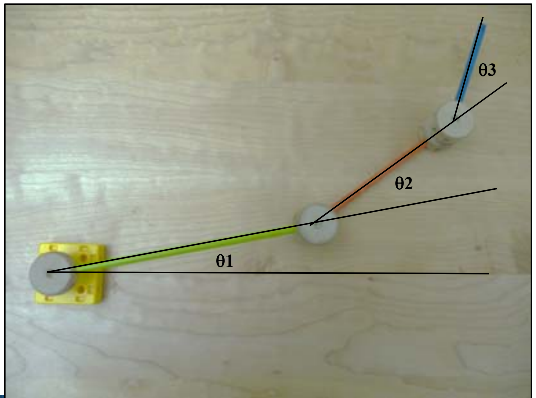
Where each arm is represented by:

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

struct armArm1;
struct armArm2;
struct armArm3;
```

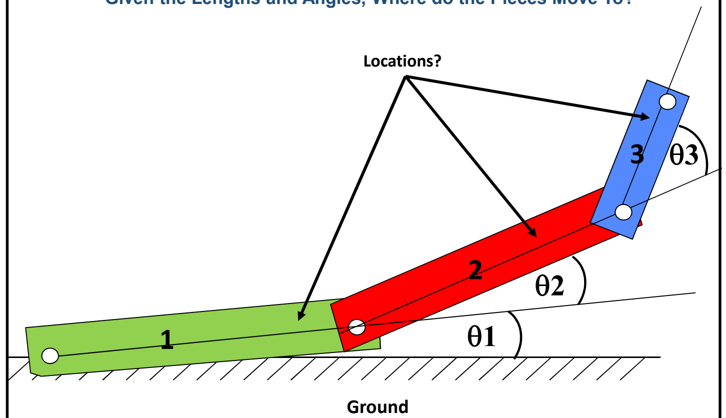

Forward Kinematics: You Start with Separate Pieces, all Defined in their Own Local Coordinate System



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



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



Positioning Part #1 With Respect to Ground

1. Rotate by Θ1
2. Translate by $T_{1/G}$

Write it

$$[M_{1/G}] = [T_{1/G}] * [R_{\theta 1}]$$

Say it

Why Do We Say it Right-to-Left?

Write it

$$[M_{1/G}] = [T_{1/G}] * [R_{\theta 1}]$$

Say it

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

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = [M_{1/G}] \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix} = [T_{1/G}] * [R_{\theta 1}] * \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

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 Θ2
2. Translate the length of part 1
3. Rotate by Θ 1
4. Translate by $T_{1/G}$

Write it

$$[M_{2/G}] = [T_{1/G}] * [R_{\theta 1}] * [T_{2/1}] * [R_{\theta 2}]$$

$$[M_{2/G}] = [M_{1/G}] * [M_{2/1}]$$

Say it

Positioning Part #3 With Respect to Ground

1. Rotate by Θ3
2. Translate the length of part 2
3. Rotate by Θ2
4. Translate the length of part 1
5. Rotate by Θ1
6. Translate by $T_{1/G}$

Write it

$$[M_{3/G}] = [T_{1/G}] * [R_{\theta 1}] * [T_{2/1}] * [R_{\theta 2}] * [T_{3/2}] * [R_{\theta 3}]$$

$$[M_{3/G}] = [M_{1/G}] * [M_{2/1}] * [M_{3/2}]$$

Say it

In the *Reset* Function


```
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;
```

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


```
VkPushConstantRange                    vpcr[1];
         vpcr[0].stageFlags =
                    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
                 |  VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
         vpcr[0].offset = 0;
         vpcr[0].size = sizeof( struct arm );

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

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

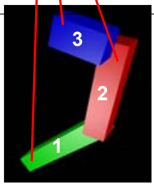
In the *UpdateScene* Functiom


```
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]

glm::mat4 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

glm::mat4 m32 = glm::mat4( 1. );     // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);                    // [T]*[R]
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));       // z-offset from previous arm

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


In the *RenderScene* Function

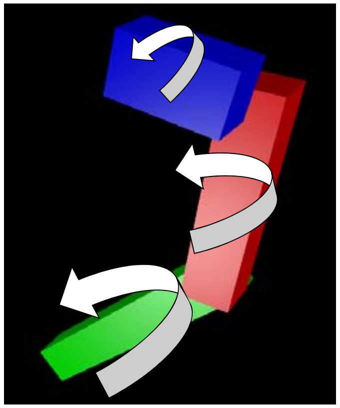

```
VkBuffer buffers[1]   = { MyVertexDataBuffer.buffer };

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


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

layout( location = 0 ) in vec3 aVertex;

      . . .

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

      . . .

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

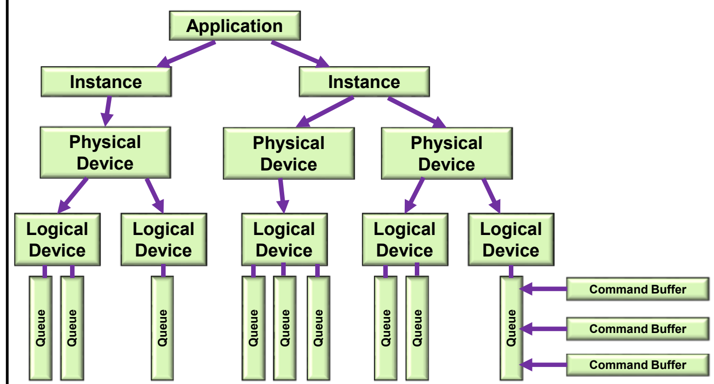



Physical Devices

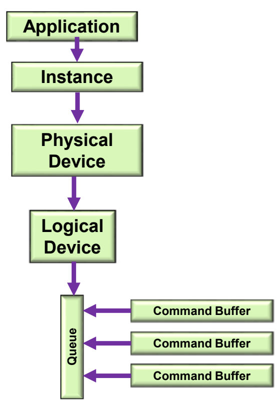
Mike Bailey
mjb@cs.oregonstate.edu

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

Vulkan: Overall Block Diagram



Vulkan: a More Typical (and Simplified) Block Diagram



Querying the Number of Physical Devices

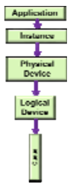
```
uint32_t  count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );

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

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

	How many total there are	Where to put them
result = vkEnumeratePhysicalDevices(Instance,	&count,	nullptr);
result = vkEnumeratePhysicalDevices(Instance,	&count,	physicalDevices);

Vulkan: Identifying the Physical Devices

297




```
VkResult result = VK_SUCCESS;

result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevice *)nullptr );
if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 )
{
        fprintf( FpDebug, "Could not count the physical devices\n" );
        return VK_SHOULD_EXIT;
}

fprintf(FpDebug, "\n%d physical devices found.\n", PhysicalDeviceCount);

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ PhysicalDeviceCount ];
result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDevices  );
if( result != VK_SUCCESS )
{
        fprintf( FpDebug, "Could not enumerate the %d physical devices\n", PhysicalDeviceCount );
        return VK_SHOULD_EXIT;
}
```

mjb – July 24, 2020

Which Physical Device to Use, I

298

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

        fprintf( FpDebug, " \n\nDevice %2d:\n", i );
        fprintf( FpDebug, "\tAPI version: %d\n", vpdp.apiVersion );
        fprintf( FpDebug, "\tDriver version: %d\n", vpdp.apiVersion );
        fprintf( FpDebug, "\tVendor ID: 0x%04x\n", vpdp.vendorID );
        fprintf( FpDebug, "\tDevice ID: 0x%04x\n", vpdp.deviceID );
        fprintf( FpDebug, "\tPhysical Device Type: %d =", vpdp.deviceType );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )     fprintf( FpDebug, " (Discrete GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )   fprintf( FpDebug, " (Integrated GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )      fprintf( FpDebug, " (Virtual GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )              fprintf( FpDebug, " (CPU)\n" );
        fprintf( FpDebug, "\tDevice Name: %s\n", vpdp.deviceName );
        fprintf( FpDebug, "\tPipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0] );
```

mjb – July 24, 2020

Which Physical Device to Use, II

299

```
        // need some logical here to decide which physical device to select:

        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
                discreteSelect = i;

        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
                integratedSelect = i;
}

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

mjb – July 24, 2020

Asking About the Physical Device's Features

300

```
VkPhysicalDeviceProperties  PhysicalDeviceFeatures;
vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );

fprintf( FpDebug, "\nPhysical Device Features:\n");
fprintf( FpDebug, "geometryShader = %2d\n", PhysicalDeviceFeatures.geometryShader);
fprintf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShader );
fprintf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect );
fprintf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines );
fprintf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport );
fprintf( FpDebug, "occlusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.occlusionQueryPrecise );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStatisticsQuery );
fprintf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64 );
fprintf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64 );
fprintf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16 );
```

mjb – July 24, 2020

Here's What the NVIDIA RTX 2080 Ti Produced


```
vkEnumeratePhysicalDevices:

Device  0:
        API version: 4198499
        Driver version: 4198499
        Vendor ID: 0x10de
        Device ID: 0x1e04
        Physical Device Type: 2 = (Discrete GPU)
        Device Name: RTX 2080 Ti
        Pipeline Cache Size: 206
Device #0 selected ('RTX 2080 Ti')

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


Here's What the Intel HD Graphics 520 Produced


```
vkEnumeratePhysicalDevices:

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

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


Asking About the Physical Device's Different Memories


```
VkPhysicalDeviceMemoryProperties                vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );

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

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");
}
```


Here's What I Got


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

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

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");
}
```


Here's What I Got


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

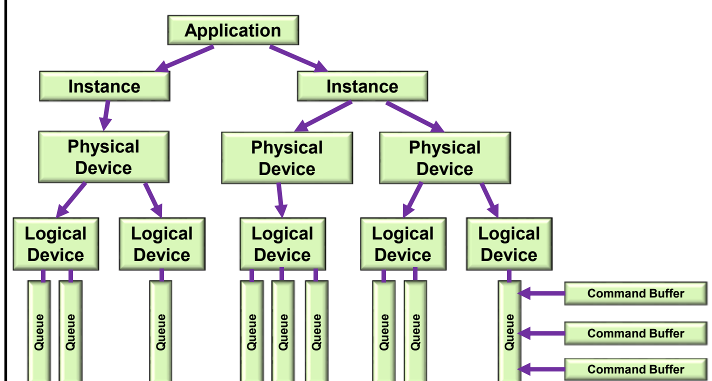

Logical Devices

Mike Bailey

mjb@cs.oregonstate.edu

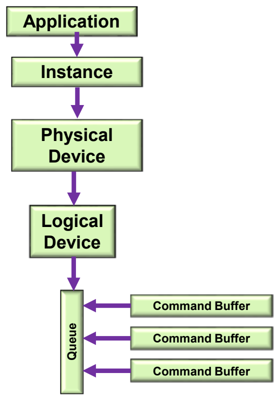
http://cs.oregonstate.edu/~mjb/vulkan

Vulkan: Overall Block Diagram



Vulkan: a More Typical (and Simplified) Block Diagram

309



Looking to See What Device Layers are Available

310

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

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

// see what device layers are available:

uint32_t  layerCount;
vkEnumerateDeviceLayerProperties(PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);

VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];

result = vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, deviceLayers);
```


Looking to See What Device Extensions are Available

311

```
// 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

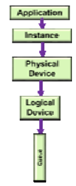
312

```
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

313

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




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

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

mjb – July 24, 2020

Vulkan: Creating the Logical Device's Queue

314

```
// get the queue for this logical device:

vkGetDeviceQueue( LogicalDevice, 0, 0,  OUT &Queue );          // 0, 0 = queueFamilyIndex, queueIndex
```

mjb – July 24, 2020

315

Dynamic State Variables

Mike Bailey

mjb@cs.oregonstate.edu

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

mjb – July 24, 2020

Creating a Pipeline with Dynamically Changeable State Variables

316

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.

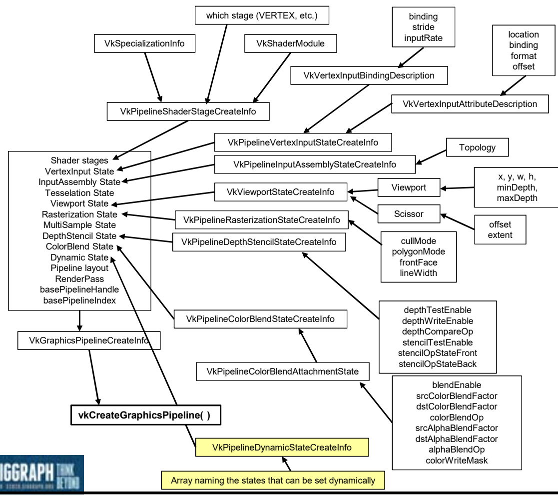
That isn't quite true. To a certain extent, Vulkan allows you to declare parts of the pipeline state changeable. This allows you to alter pipeline state information on the fly.

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

mjb – July 24, 2020

Creating a Pipeline

317



Which Pipeline State Variables can be Changed Dynamically

318

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

319

```
VkDynamicState                          vds[ ] =
{
        VK_DYNAMIC_STATE_VIEWPORT,
        VK_DYNAMIC_STATE_LINE_WIDTH
};

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

VkGraphicsPipelineCreateInfo            vgpci;
        . . .
        vgpci.pDynamicState = &vpdsci;
        . . .

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

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

Filling the Dynamic State Variables in the Command Buffer

320

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

Setting up Query Pools


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

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

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

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


Resetting, Filling, and Examining a Query Pool


```
vkCmdResetQueryPool( CommandBuffer, occlusionQueryPool, 0, 1 );

vkCmdBeginQuery( CommandBuffer, occlusionQueryPool, 0, VK_QUERY_CONTROL_PRECISE_BIT );
                                                    query index number
    . . .

vkCmdEndQuery( CommandBuffer, occlusionQueryPool, 0 );

#define DATASIZE      128
uint32_t     data[DATASIZE];

result = vkGetQueryPoolResults( LogicalDevice, occlusionQueryPool, 0, 1, DATASIZE*sizeof(uint32_t), data, stride, flags );
        // or'ed combinations of:
        // VK_QUERY_RESULT_64_BIT
        // VK_QUERY_RESULT_WAIT_BIT
        // VK_QUERY_RESULT_WITH_AVAILABILTY_BIT
        // VK_QUERY_RESULT_PARTIAL_BIT
        // stride is # of bytes in between each result
```

Occlusion Query

325

Occlusion Queries count the number of fragments drawn between the **vkCmdBeginQuery** and the **vkCmdEndQuery** that pass both the Depth and Stencil tests

This is commonly used to see what level-of-detail should be used when drawing a complicated object

Some hints:

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

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

mjb – July 24, 2020

Pipeline Statistics Query

326

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

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


// vqpci.pipelineStatistics = or'ed 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
```

mjb – July 24, 2020

Timestamp Query

327

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

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

mjb – July 24, 2020

Timestamp Query

328

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

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

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

mjb – July 24, 2020

329

Vulkan.

Compute Shaders

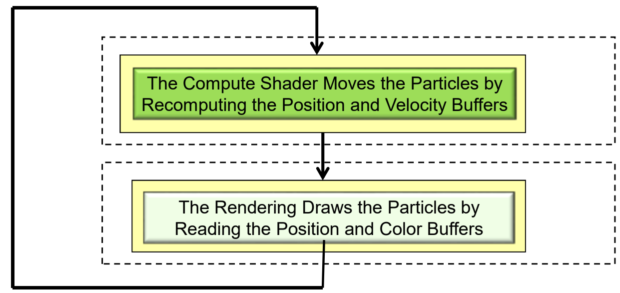
Mike Bailey

mjb@cs.oregonstate.edu

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

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

330



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

331

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

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


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

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

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

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

The Data in your Compute Shader will look like This

332

1

2

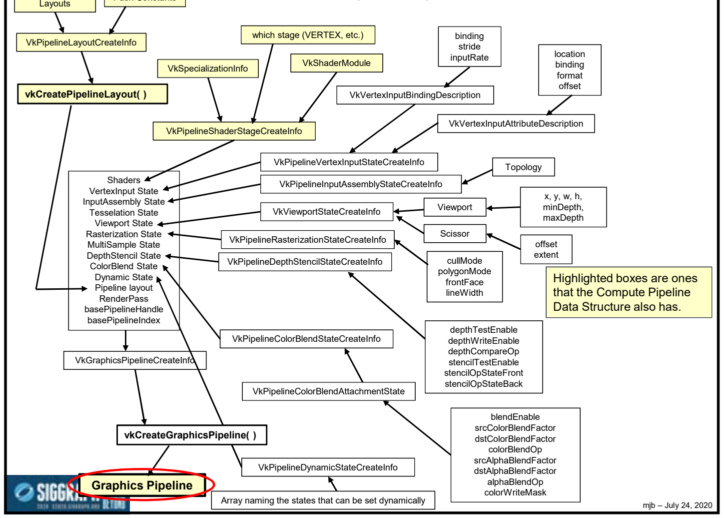
3

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

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

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

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

Remember the Graphics Pipeline Data Structure?

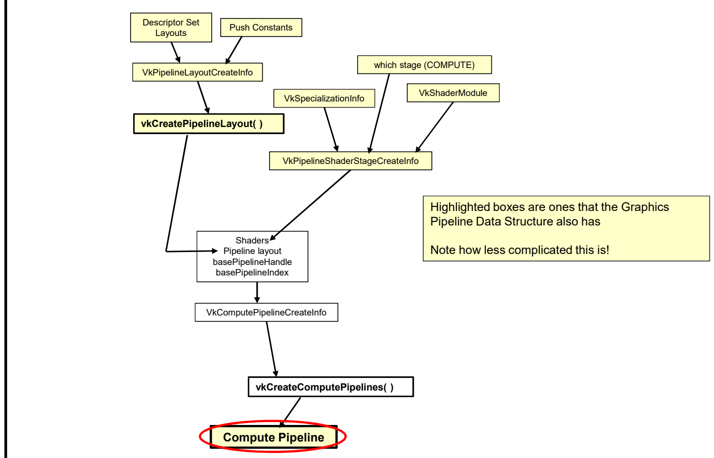
333



mjb – July 24, 2020

Here is how you create a Compute Pipeline Data Structure

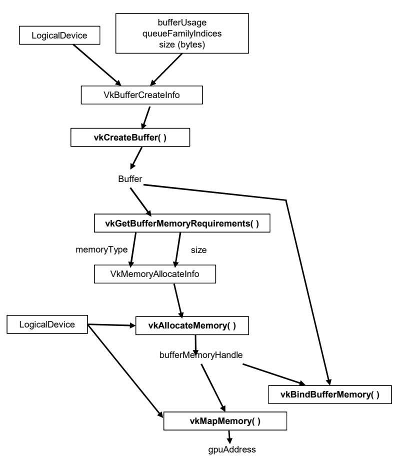
334



mjb – July 24, 2020

A Reminder about Data Buffers

335



mjb – July 24, 2020

Creating a Shader Storage Buffer

336

```
VkBuffer  PosBuffer;
. . .

VkBufferCreateInfo  vbci;
        vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        vbci.pNext = nullptr;
        vbci.flags = 0;
        vbci.size = NUM_PARTICLES * sizeof( glm::vec4 );
        vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
        vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vbci.queueFamilyIndexCount = 0;
        vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

mjb – July 24, 2020

Allocating Memory for a Buffer, Binding a Buffer to Memory, and Filling the Buffer

337

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


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

. . .

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

result = vkBindBufferMemory( LogicalDevice, PosBuffer, IN vdm, 0 );        // 0 is the offset
```

SIGGRAPH 2020 THINK BEYOND

mjb – July 24, 2020

Create the Compute Pipeline Layout

338

```
VkDescriptorSetLayoutBinding          ComputeSet[3];
        ComputeSet[0].binding              = 0;
        ComputeSet[0].descriptorType       = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ComputeSet[0].descriptorCount      = 1;
        ComputeSet[0].stageFlags           = VK_SHADER_STAGE_COMPUTE_BIT;
        ComputeSet[0].pImmutableSamplers = (VkSampler *)nullptr;

        ComputeSet[1].binding              = 1;
        ComputeSet[1].descriptorType       = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ComputeSet[1].descriptorCount      = 1;
        ComputeSet[1].stageFlags           = VK_SHADER_STAGE_COMPUTE_BIT;
        ComputeSet[1].pImmutableSamplers = (VkSampler *)nullptr;

        ComputeSet[2].binding              = 2;
        ComputeSet[2].descriptorType       = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        ComputeSet[2].descriptorCount      = 1;
        ComputeSet[2].stageFlags           = VK_SHADER_STAGE_COMPUTE_BIT;
        ComputeSet[2].pImmutableSamplers = (VkSampler *)nullptr;

VkDescriptorSetLayoutCreateInfo         vdslc;
        vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
        vdslc0.pNext = nullptr;
        vdslc0.flags = 0;
        vdslc0.bindingCount = 3;
        vdslc0.pBindings = &ComputeSet[0];
```

SIGGRAPH 2020 THINK BEYOND

mjb – July 24, 2020

Create the Compute Pipeline Layout

339

```
VkPipelineLayout              ComputePipelineLayout;
VkDescriptorSetLayout         ComputeSetLayout;

. . .

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

VkPipelineLayoutCreateInfo                vplci;
        vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        vplci.pNext = nullptr;
        vplci.flags = 0;
        vplci.setLayoutCount = 1;
        vplci.pSetLayouts = ComputeSetLayout;
        vplci.pushConstantRangeCount = 0;
        vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

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

SIGGRAPH 2020 THINK BEYOND

mjb – July 24, 2020

Create the Compute Pipeline

340

```
VkPipeline                    ComputePipeline;
. . .

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


VkComputePipelineCreateInfo              vcpci[1];
        vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
        vcpci[0].pNext = nullptr;
        vcpci[0].flags = 0;
        vcpci[0].stage = vpssci;
        vcpci[0].layout = ComputePipelineLayout;
        vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
        vcpci[0].basePipelineIndex = 0;

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

SIGGRAPH 2020 THINK BEYOND

mjb – July 24, 2020

Creating a Vulkan Data Buffer

341

```
VkBuffer  Buffer;

VkBufferCreateInfo  vbci;
        vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        vbci.pNext = nullptr;
        vbci.flags = 0;
        vbci.size = NUM_PARTICLES * sizeof( glm::vec4 );
        vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
        vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
        vbci.queueFamilyIndexCount = 0;
        vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

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

mjb – July 24, 2020

Allocating Memory and Binding the Buffer

342

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

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

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

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

MyBuffer   myPosBuffer;
        myPosBuffer.size    = vbci.size;
        myPosBuffer.buffer = PosBuffer;
        myPosBuffer.vdm   = vdm;
```

mjb – July 24, 2020

Fill the Buffers

343

```
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 );
```

mjb – July 24, 2020

Fill the Buffers

344

```
#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 ;
}
```

mjb – July 24, 2020

The Particle System Compute Shader

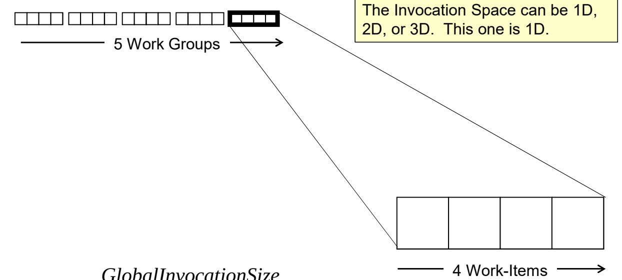

```
layout( std140, set = 0, binding = 0 )  buffer  Pos
{
        vec4  Positions[   ];              // array of structures
};

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

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

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

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

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

20 total items to compute:

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



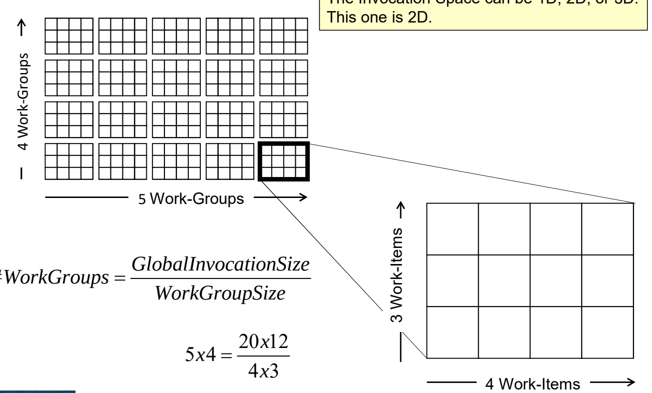
$$\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}$$

$$5x4 = \frac{20}{4}$$

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

20x12 (=240) total items to compute:

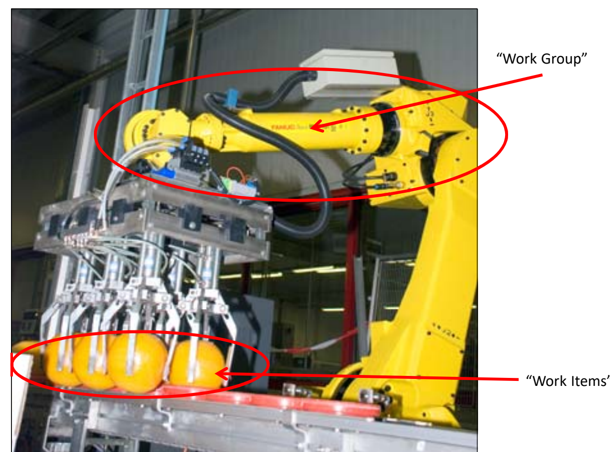
The Invocation Space can be 1D, 2D, or 3D. This one is 2D.



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

$$5x4 = \frac{20x12}{4x3}$$

A Mechanical Equivalent…

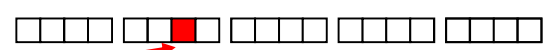


http://news.cision.com

The Particle System Compute Shader – The Physics

349

```
#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)

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

POINT      pp = p + v*DT + .5*DT*DT*G;
VELOCITY  vp = v + G*DT;

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

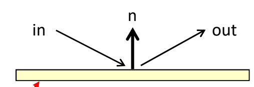
$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v' = v + G \cdot t$$

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

350

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

// plane equation:  Ax + By + Cz + D = 0
// ( it turns out that (A,B,C) is the normal )

VELOCITY
BouncePlane( POINT p, VELOCITY v, PLANE pl)
{
      VECTOR  n = normalize(  VECTOR( pl.xyz )  );
      return Bounce( v, n );
}

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

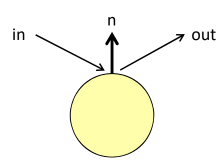


Note: a surface in the x-z plane has the equation:
0x + 1y + 0z + 0 = 0
and thus its normal vector is (0,1,0)

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

351

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

bool
IsInsideSphere( POINT p, SPHERE s )
{
      float r = length( p - s.xyz );
      return  (  r < s.w  );
}
```



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

352

```
uint  gid = gl_GlobalInvocationID.x;         // the .y and .z are both 1 in this case

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

POINT      pp = p + v*DT + .5*DT*DT*G;
VELOCITY  vp = v + G*DT;

if(  IsInsideSphere( pp, Sphere )  )
{
      vp = BounceSphere( p, v, S );
      pp = p + vp*DT + .5*DT*DT*G;
}

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

$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v' = v + G \cdot t$$

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

Dispatching the Compute Shader from the Command Buffer

353

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

     . . .

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );

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

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

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

mjb – July 24, 2020

Displaying the Particles

354

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

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

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

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

mjb – July 24, 2020

Displaying the Particles

355

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

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

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

mjb – July 24, 2020

Telling the Pipeline about its Input

356

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

VkPipelineInputAssemblyStateCreateInfo        vpiasci;
                vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
                vpiasci.pNext = nullptr;
                vpiasci.flags = 0;
                vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
```

mjb – July 24, 2020

Telling the Pipeline about its Input

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

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

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


Setting a Pipeline Barrier so the Drawing Waits for the Compute


```
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 =
        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
        IN &vbmb,    0,    nullptr
);
```


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 Compute Waits for the Drawing


```
VkBufferMemoryBarrier                   vbmb;
        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 =
        vbmb.offset = 0;
        vbmb.size = ??

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
        IN &vbmb,    0,    nullptr
);
```

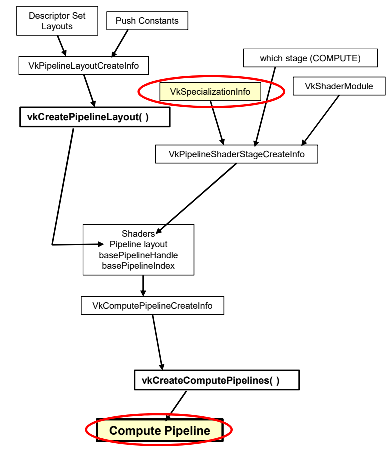
Specialization Constants

Mike Bailey

mjb@cs.oregonstate.edu

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

Remember the Compute Pipeline?



What Are Specialization Constants?

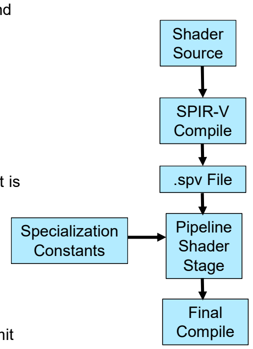
In Vulkan, all shaders get halfway-compiled into SPIR-V and then the rest-of-the-way compiled by the Vulkan driver.

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

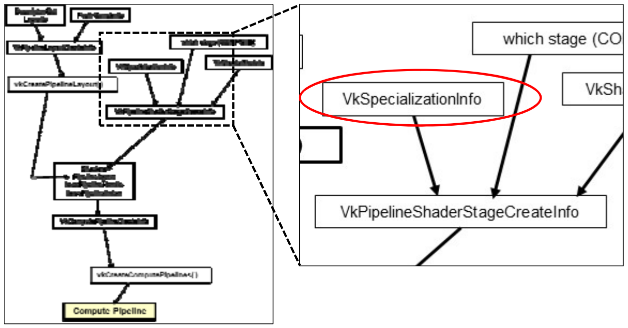


Why Do We Need Specialization Constants?

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



Specialization Constant Example -- Setting an Array Size

In the compute shader

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

int array[ASIZE];
```

In the Vulkan C/C++ program:

```
int asize = 64;

VkSpecializationMapEntry     vsme[1];     // one array element for each
                                          //      Specialization Constant
        vsme[0].constantID = 7;
        vsme[0].offset = 0;               // # bytes into the Specialization Constant
                                          //      array this one item is
        vsme[0].size = sizeof(asize);     // size of just this Specialization Constant

VkSpecializationInfo         vsi;
        vsi.mapEntryCount = 1;
        vsi.pMapEntries = &vsme[0];
        vsi.dataSize = sizeof(asize);     // size of all the Specialization Constants together
        vsi.pData = &asize;               // array of all the Specialization Constants
```


Linking the Specialization Constants into the Compute Pipeline


```
int asize = 64;

VkSpecializationMapEntry vsme[1];
        vsme[0].constantID = 7;
        vsme[0].offset = 0;
        vsme[0].size = sizeof(asize);

VkSpecializationInfo        vsi;
        vsi.mapEntryCount = 1;
        vsi.pMapEntries = &vsme[0];
        vsi.dataSize = sizeof(asize);
        vsi.pData = &asize;

VkPipelineShaderStageCreateInfo            vpssci;
            vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
            vpssci.pNext = nullptr;
            vpssci.flags = 0;
            vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
            vpssci.module = computeShader;
            vpssci.pName = "main";
            vpssci.pSpecializationInfo = &vsi;

VkComputePipelineCreateInfo               vcpci[1];
            vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
            vcpci[0].pNext = nullptr;
            vcpci[0].flags = 0;
            vcpci[0].stage = vpssci;
            vcpci[0].layout = ComputePipelineLayout;
            vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
            vcpci[0].basePipelineIndex = 0;

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


Specialization Constant Example – 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:

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

VkSpecializationMapEntry     vsme[3];
        vsme[0].constantID = 9;
        vsme[0].offset = offsetof( abc, a );
        vsme[0].size = sizeof(abc.a);
        vsme[1].constantID = 10;
        vsme[1].offset = offsetof( abc, b );
        vsme[1].size = sizeof(abc.b);
        vsme[2].constantID = 11;
        vsme[2].offset = offsetof( abc, c );
        vsme[2].size = sizeof(abc.c);

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

It's important to use sizeof( ) and offsetof( ) instead of hardcoding numbers!

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

369

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:

```
int numXworkItems = 64;

VkSpecializationMapEntry      vsme[1];
      vsme[0].constantID = 12;
      vsme[0].offset = 0;
      vsme[0].size = sizeof(int);

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

SIGGRAPH

mjb – July 24, 2020

370

Synchronization

Mike Bailey

mjb@cs.oregonstate.edu

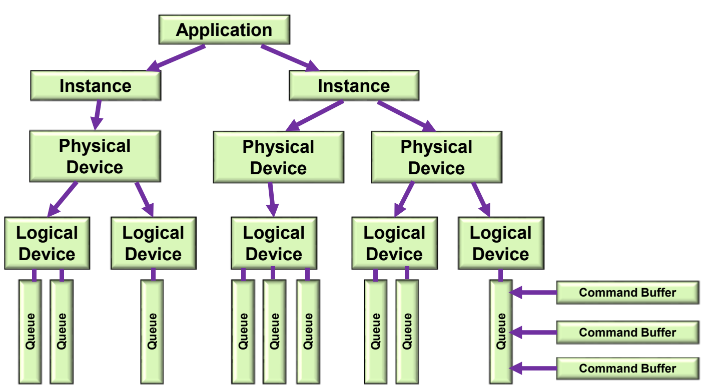
http://cs.oregonstate.edu/~mjb/vulkan

SIGGRAPH

mjb – July 24, 2020

Remember the Overall Block Diagram?

371

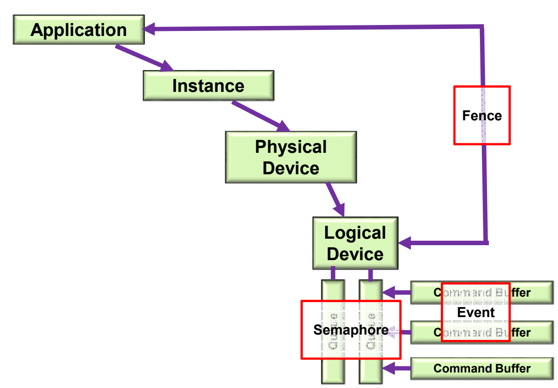


SIGGRAPH

mjb – July 24, 2020

Where Synchronization Fits in the Overall Block Diagram

372

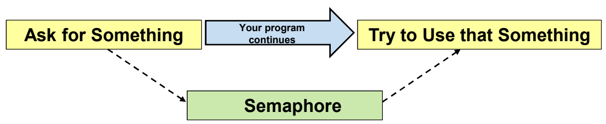


SIGGRAPH

mjb – July 24, 2020

Semaphores

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



Creating a Semaphore


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

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

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

Semaphores Example during the Render Loop


```
VkSemaphore imageReadySemaphore;

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

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

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 );
```

Set the semaphore

Wait on the semaphore

You do this to wait for an image to be ready to be rendered into

Fences

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

Fences

377

```
#define VK_FENCE_CREATE_UNSIGNALED_BIT        0

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

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

        , , ,

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

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

Set the fence

Wait on the fence(s)

mjb – July 24, 2020

Fence Example

378

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

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

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

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

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

    . . .

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

    . . .

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

mjb – July 24, 2020

Events

379

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

mjb – July 24, 2020

Controlling Events from the Host

380

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

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

result = vkSetEvent( LogicalDevice, IN event );

result = vkResetEvent( LogicalDevice, IN event );

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

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

mjb – July 24, 2020

Controlling Events from the Device


```
result = vkCmdSetEvent(     CommandBuffer, IN event, pipelineStageBits );

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

result = vkCmdWaitEvents( CommandBuffer, 1, &event,
        srcPipelineStageBits, dstPipelineStageBits,
          memoryBarrierCount, pMemoryBarriers,
          bufferMemoryBarrierCount, pBufferMemoryBarriers,
          imageMemoryBarrierCount, pImageMemoryBarriers
        );
```

Could be an array of events

Where signaled, where wait for the signal

Memory barriers get executed after events have been signaled

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

Vulkan.

Pipeline Barriers

Mike Bailey

mjb@cs.oregonstate.edu

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

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


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


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


```
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers,
        bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers,
            bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```

Potential Memory Race Conditions that Pipeline Barriers can Prevent

385

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

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

vkCmdPipelineBarrier( ) Function Call

386

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

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

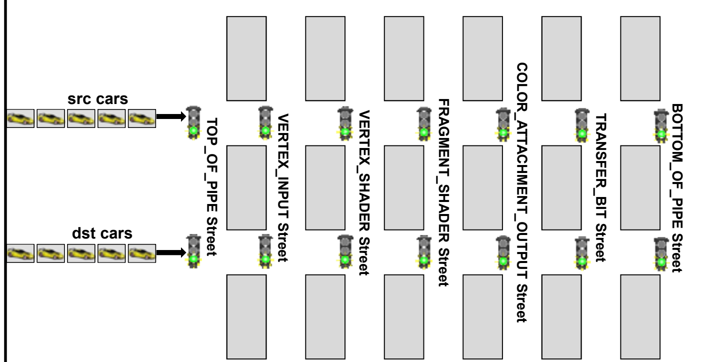
srcStageMask: Guarantee that *this* pipeline stage is completely done being used before …

dstStageMask: … allowing *this* pipeline stage to be used

pMemoryBarriers, pBufferMemoryBarriers, pImageMemoryBarriers: Defines what data we will be blocking on or un-blocking on

The Scenario

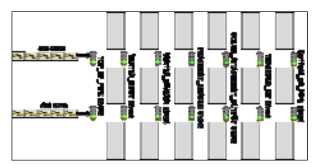
387



The Scenario

388

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?

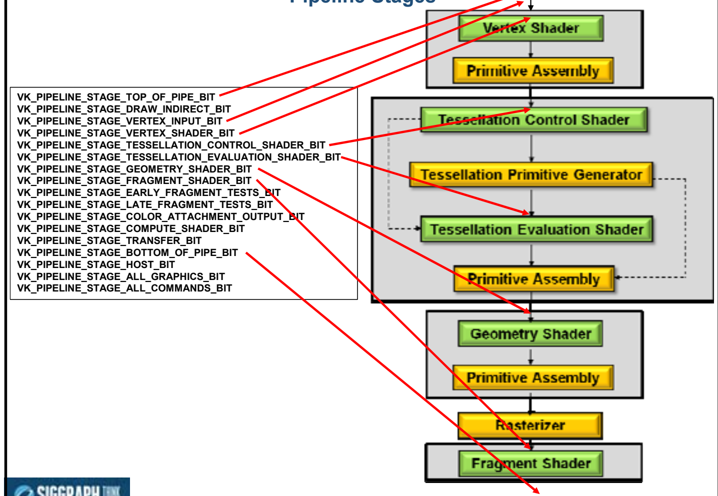
389

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

mjb – July 24, 2020

Pipeline Stages

390



mjb – July 24, 2020

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

391

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

mjb – July 24, 2020

Pipeline Stages and what Access Operations are Allowed

392

		VK_ACCESS_INDIRECT_COMMAND_READ_BIT	VK_ACCESS_INDEX_READ_BIT	VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT	VK_ACCESS_UNIFORM_READ_BIT	VK_ACCESS_INPUT_ATTACHMENT_READ_BIT	VK_ACCESS_SHADER_READ_BIT	VK_ACCESS_SHADER_WRITE_BIT	VK_ACCESS_COLOR_ATTACHMENT_READ_BIT	VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT	VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT	VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT	VK_ACCESS_TRANSFER_READ_BIT	VK_ACCESS_TRANSFER_WRITE_BIT	VK_ACCESS_HOST_READ_BIT	VK_ACCESS_HOST_WRITE_BIT	VK_ACCESS_MEMORY_READ_BIT	VK_ACCESS_MEMORY_WRITE_BIT
1	VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT																	
2	VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT	•																
3	VK_PIPELINE_STAGE_VERTEX_INPUT_BIT		•	•														
4	VK_PIPELINE_STAGE_VERTEX_SHADER_BIT				•		•	•										
5	VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT				•		•	•										
6	VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT				•		•	•										
7	VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT				•		•	•										
8	VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT										•	•						
9	VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT				•	•	•	•										
10	VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT										•	•						
11	VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT								•	•								
12	VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT																	
	VK_PIPELINE_STAGE_COMPUTE_SHADER				•		•	•										
	VK_PIPELINE_STAGE_TRANSFER_BIT												•	•				
	VK_PIPELINE_STAGE_HOST_BIT														•	•		

mjb – July 24, 2020

Access Operations and what Pipeline Stages they can be used In

	1 VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT	2 VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT	3 VK_PIPELINE_STAGE_VERTEX_INPUT_BIT	4 VK_PIPELINE_STAGE_VERTEX_SHADER_BIT	5 VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT	6 VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT	7 VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT	8 VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT	9 VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT	10 VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT	11 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT	12 VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT	VK_PIPELINE_STAGE_COMPUTE_SHADER	VK_PIPELINE_STAGE_TRANSFER_BIT	VK_PIPELINE_STAGE_HOST_BIT
VK_ACCESS_INDIRECT_COMMAND_READ_BIT		•													
VK_ACCESS_INDEX_READ_BIT			•												
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT			•												
VK_ACCESS_UNIFORM_READ_BIT				•	•	•	•		•				•		
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT									•						
VK_ACCESS_SHADER_READ_BIT				•	•	•	•		•				•		
VK_ACCESS_SHADER_WRITE_BIT				•	•	•	•		•				•		
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT											•				
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT											•				
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT								•		•					
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT								•		•					
VK_ACCESS_TRANSFER_READ_BIT														•	
VK_ACCESS_TRANSFER_WRITE_BIT														•	
VK_ACCESS_HOST_READ_BIT															•
VK_ACCESS_HOST_WRITE_BIT															•
VK_ACCESS_MEMORY_READ_BIT															
VK_ACCESS_MEMORY_WRITE_BIT															

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

Stages

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT ← **src**
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 ← **dst**
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

Access types

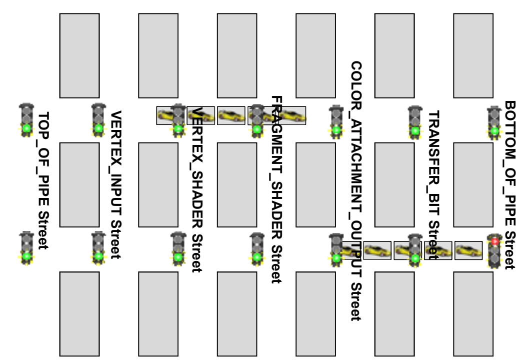
VK_ACCESS_INDIRECT_COMMAND_READ_BIT
VK_ACCESS_INDEX_READ_BIT
VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT
VK_ACCESS_UNIFORM_READ_BIT
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT
VK_ACCESS_SHADER_READ_BIT ← **src**
VK_ACCESS_SHADER_WRITE_BIT
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT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

dst (no access setting needed)

The Scenario

src cars are generating the image

dst cars are doing something with that image



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

Stages

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT ← **src**
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 ← **dst**
VK_PIPELINE_STAGE_HOST_BIT
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT

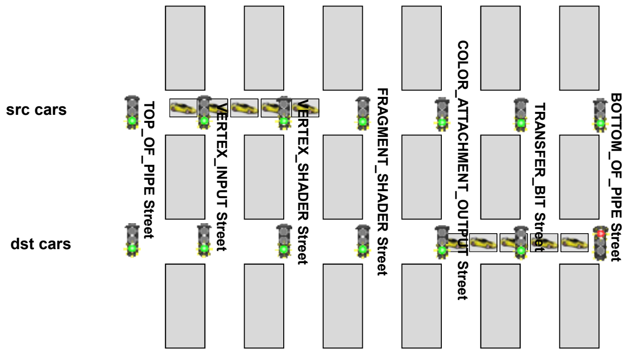
Access types

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

dst (no access setting needed)

The Scenario

397



mjb – July 24, 2020

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

398

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

VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL ← Used as a color attachment
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL ← Read into a shader as a texture
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL ← Copy from
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL ← Copy to
VK_IMAGE_LAYOUT_PREINITIALIZED
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR ← Show image to viewer
VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR

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

mjb – July 24, 2020

399

Antialiasing and Multisampling

Mike Bailey

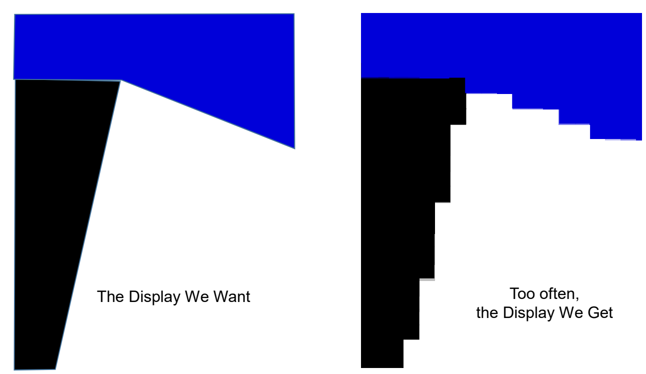
mjb@cs.oregonstate.edu

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

mjb – July 24, 2020

Aliasing

400

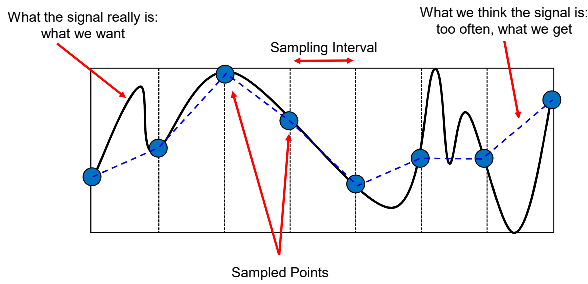


mjb – July 24, 2020

Aliasing

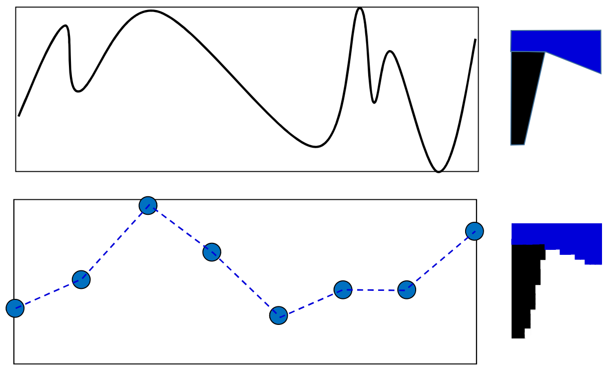
401

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



Aliasing

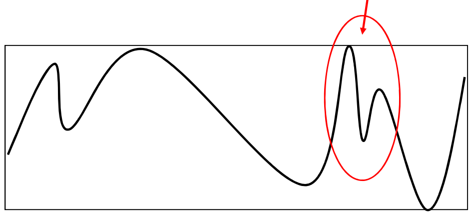
402



The Nyquist Criterion

403

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



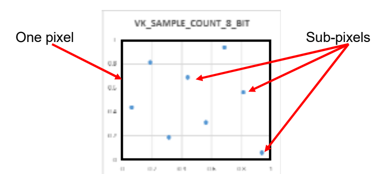
MultiSampling

404

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..

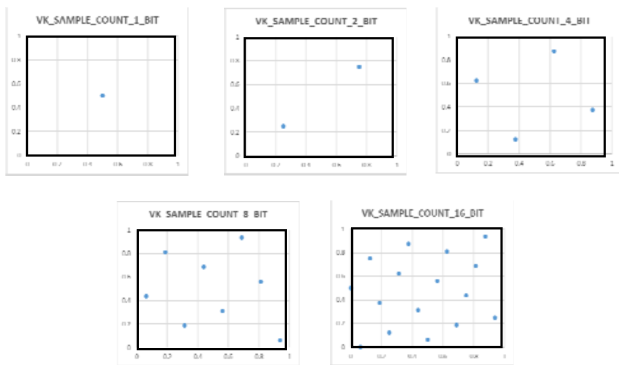


2. **Multisampling**: Pick some number of sub-pixels within that pixel that pass the depth and stencil tests. If any of them pass, then perform a single color render for the one pixel and assign that single color to all the sub-pixels that passed the depth and stencil tests.

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.

Vulkan Specification Distribution of Sampling Points within a Pixel

405



SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Vulkan Specification Distribution of Sampling Points within a Pixel

406

VK_SAMPLE_COUNT_2_BIT	VK_SAMPLE_COUNT_4_BIT	VK_SAMPLE_COUNT_8_BIT	VK_SAMPLE_COUNT_16_BIT
		(0.5625, 0.3125)	(0.5625, 0.5625)
	(0.375, 0.125)		(0.4375, 0.3125)
		(0.4375, 0.6875)	(0.3125, 0.625)
			(0.75, 0.4375)
(0.25,0.25)		(0.8125, 0.5625)	(0.1875, 0.375)
	(0.875, 0.375)		(0.625, 0.8125)
		(0.3125, 0.1875)	(0.8125, 0.6875)
			(0.6875, 0.1875)
		(0.1875, 0.8125)	(0.375, 0.875)
	(0.125, 0.625)		(0.5, 0.0625)
		(0.0625, 0.4375)	(0.25, 0.125)
(0.75,0.75)			(0.125, 0.75)
		(0.6875, 0.9375)	(0.0, 0.5)
	(0.625, 0.875)		(0.9375, 0.25)
		(0.9375, 0.0625)	(0.875, 0.9375)
			(0.0625, 0.0)

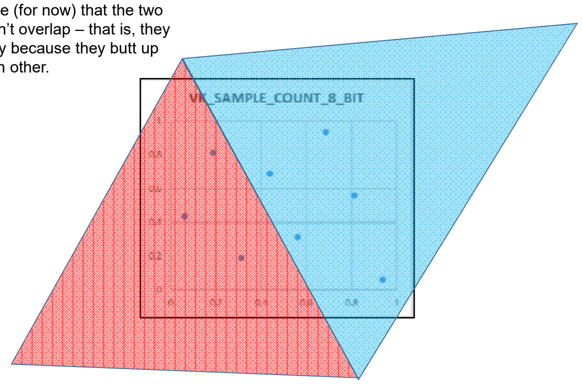
SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Consider Two Triangles Who Pass Through the Same Pixel

407

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.

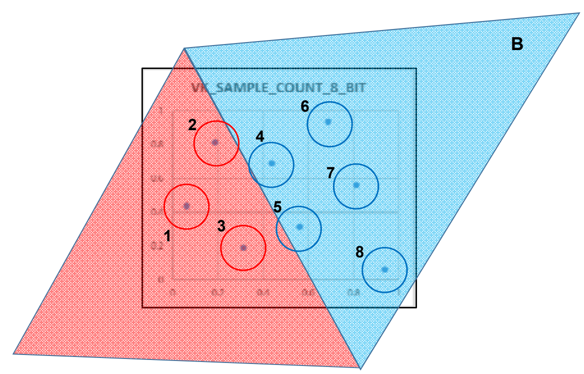


SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Supersampling

408



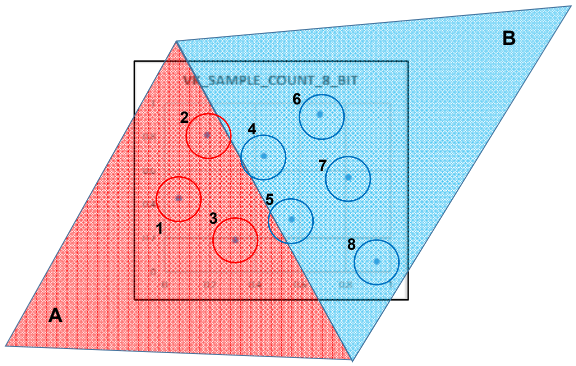
$$Final\ Pixel\ Color = \frac{\sum_{i=1}^{8} Color\ sample\ from\ subpixel_i}{8}$$

 # Fragment Shader calls = 8

SIGGRAPH THINK BEYOND

mjb – July 24, 2020

Multisampling

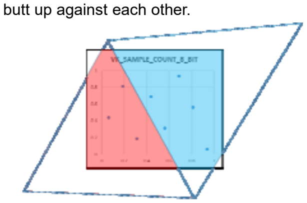


$$Final\ Pixel\ Color = \frac{3 * One\ color\ sample\ from\ A\ +\ 5 * One\ color\ sample\ from\ B}{8}$$

 # Fragment Shader calls = 2

Consider Two Triangles Who Pass Through the Same Pixel

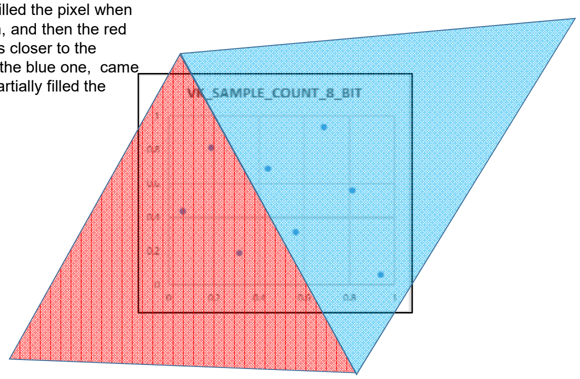
Let's assume (for now) that the two triangles don't overlap – that is, they look this way because they butt up against each other.



	Multisampling	Supersampling
Blue fragment shader calls	1	5
Red fragment shader calls	1	3

Consider Two Triangles Who Pass Through the Same Pixel

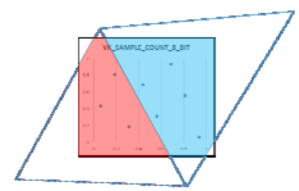
Q: What if the blue triangle completely filled the pixel when it was drawn, and then the red one, which is closer to the viewer than the blue one, came along and partially filled the pixel?



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

Consider Two Triangles Who Pass Through the Same Pixel

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



	Multisampling	Supersampling
Blue fragment shader calls	1	8
Red fragment shader calls	1	3

Setting up the Image

413

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

VkGraphicsPipelineCreateInfo          vgpci;
        vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
        vgpci.pNext = nullptr;
    . . .
        vgpci.pMultisampleState = &vpmsci;

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

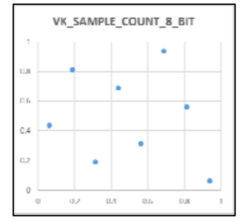
How dense is the sampling

VK_TRUE means to allow some sort of multisampling to take place

mjb – July 24, 2020

Setting up the Image

414




```
VkPipelineMultisampleStateCreateInfo          vpmsci;
    . . .

        vpmsci.minSampleShading = 0.5;

    . . .
```

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

0. produces simple multisampling

(0.,1.) produces partial supersampling

1. Produces complete supersampling

mjb – July 24, 2020

Setting up the Image

415

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

        vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
        vad[1].samples = VK_SAMPLE_COUNT_8_BIT;
        vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
        vad[1].flags = 0;

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

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

to next slide

mjb – July 24, 2020

Setting up the Image

416

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

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

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

from previous slide

mjb – July 24, 2020

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

417

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

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

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

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

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

For the *ImageLayout, use VK_IMAGE_LAYOUT_GENERAL

418

Multipass Rendering

Mike Bailey

mjb@cs.oregonstate.edu

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

Multipass Rendering uses Attachments -- What is a Vulkan *Attachment* Anyway?

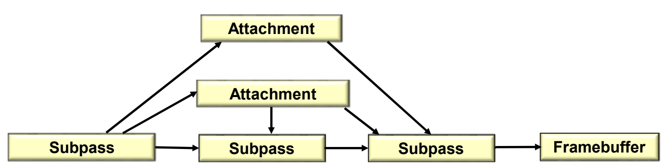
419

"[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:



What is an Example of Wanting to do This?

420

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)

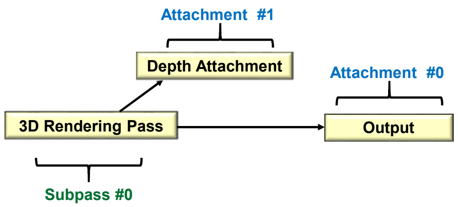
As well as:
- the current light source positions and colors
- the current eye position

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 known as the ***G-buffer Algorithm***.

Back in Our Single-pass Days

421

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



Here comes a quick reminder of how we did that.

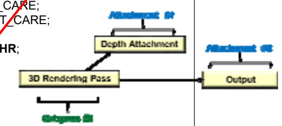
Afterwards, we will extend it.

mjb – July 24, 2020

Back in Our Single-pass Days, I

422

```
VkAttachmentDescription          vad [2 ];
        vad[0].flags = 0;
        vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
        vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
        vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
        vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
        vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

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

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

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



mjb – July 24, 2020

Back in Our Single-pass Days, II

423

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


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


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

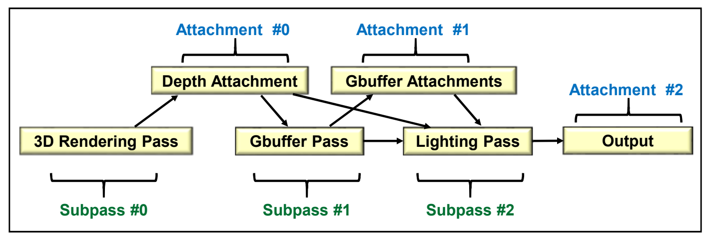
mjb – July 24, 2020

Multipass Rendering

424

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 subpass or subpasses.

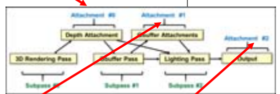
In this case, we will look at following up a 3D rendering with Gbuffer operations.



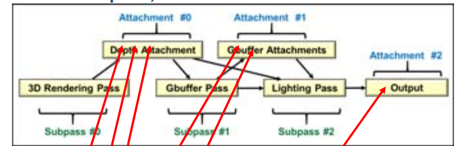
mjb – July 24, 2020

Multipass, I


```
VkAttachmentDescription                    vad [ 3 ];
        vad[0].flags = 0;
        vad[0].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
        vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
        vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[0].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        vad[0].finalLayout = VK_IMAGE_LAYOUT_UNDEFINED;

        vad[1].flags = 0;
        vad[1].format = VK_FORMAT_R32G32B32A32_UINT;
        vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
        vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        vad[1].finalLayout = VK_IMAGE_LAYOUT_UNDEFINED;

        vad[2].flags = 0;
        vad[2].format = VK_FORMAT_R8G8B8A8_SRGB;
        vad[2].samples = VK_SAMPLE_COUNT_1_BIT;
        vad[2].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad[2].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
        vad[2].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
        vad20].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
        vad[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
        vad[2].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC;
```



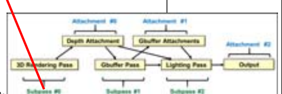
Multipass, II




```
VkAttachmentReference                  depthOutput;
        depthOutput.attachment = 0;          // depth
        depthOutput.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkAttachmentReference                  gbufferInput;
        gBufferInput.attachment = 0;         // depth
        gBufferInput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference                  gbufferOutput;
        gBufferOutput.attachment = 1;        // gbuffer
        gBufferOutput.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference                  lightingInput[2];
        lightingInput[0].attachment = 0;     // depth
        lightingInput[0].layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL;
        lightingInput[1].attachment = 1;     // gbuffer
        lightingInput[1].layout = VK_IMAGE_LAYOUT_SHADER_READ_OPTIMAL;

VkAttachmentReference                  lightingOutput;
        lightingOutput.attachment = 2;       // color rendering
        lightingOutput = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```


Multipass, III

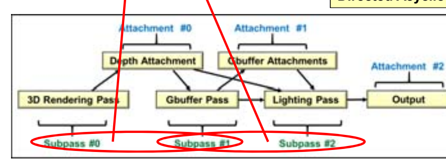

```
VkSubpassDescription                vsd[3];
        vsd[0].flags = 0;
        vsd[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
        vsd[0].inputAttachmentCount = 0;
        vsd[0].pInputAttachments = (VkAttachmentReference *)nullptr;
        vsd[0].colorAttachmentCount = 0;
        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
```



Multipass, IV


```
VkSubpassDependency                 vsdp[2];
        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;
```

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



Multipass, V

429

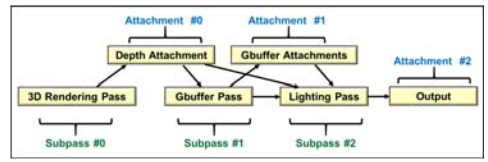
```
VkRenderPassCreateInfo                vrpci;
        vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
        vrpci.pNext = nullptr;
        vrpci.flags = 0;
        vrpci.attachmentCount = 3;     // depth, gbuffer, output
        vrpci.pAttachments = vad;
        vrpci.subpassCount = 3;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

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

mjb – July 24, 2020

Multipass, VI

430

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

// subpass #0 is automatically started here

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



mjb – July 24, 2020

431

Vulkan Ray Tracing

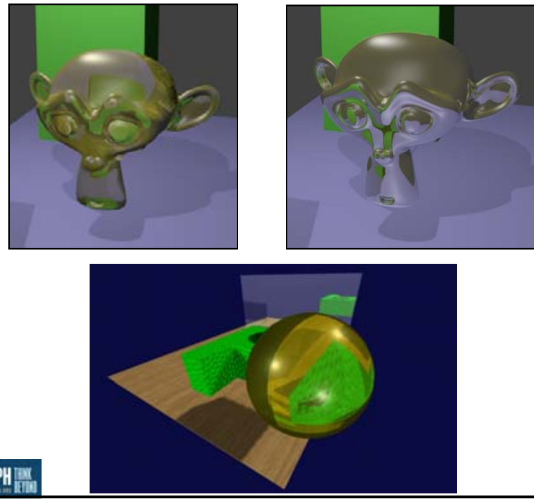
mjb – July 24, 2020

Analog Ray Tracing Example

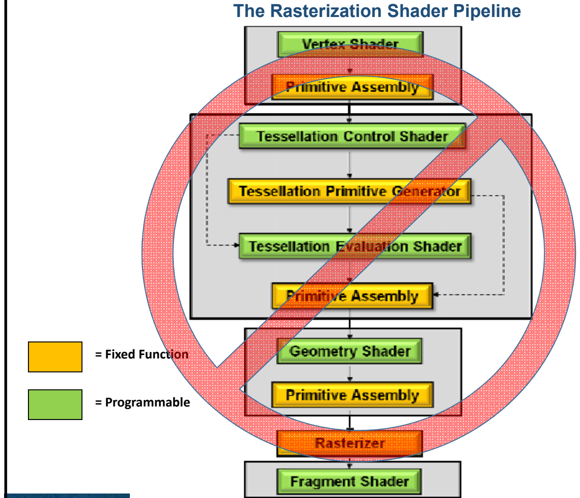
432

mjb – July 24, 2020

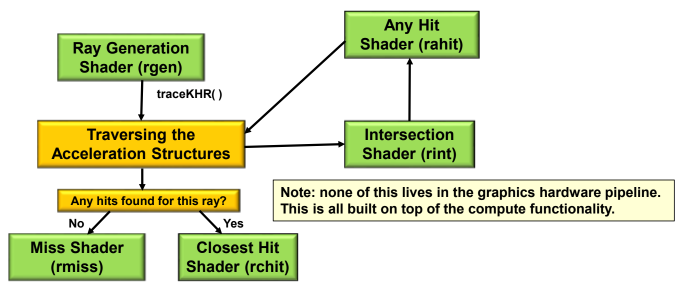
Digital Ray Tracing Examples



The Rasterization Shader Pipeline



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

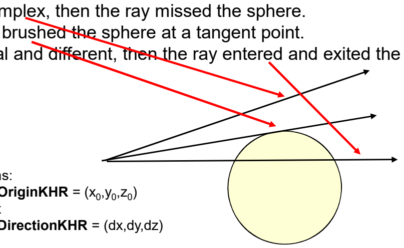
1. Sphere equation: $(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2$

2. Ray equation: $(x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz)$

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

$At^2 + Bt + C = 0$
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 = (x_0,y_0,z_0)
gl_HitKHR = t
gl_WorldRayDirectionKHR = (dx,dy,dz)

The Ray Intersection Process for a Cube

437

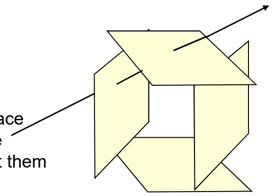
1. Plane equation: Ax + By + Cz + D = 0

2. Ray equation: $(x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz)$

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

At + B = 0
Solve for t

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

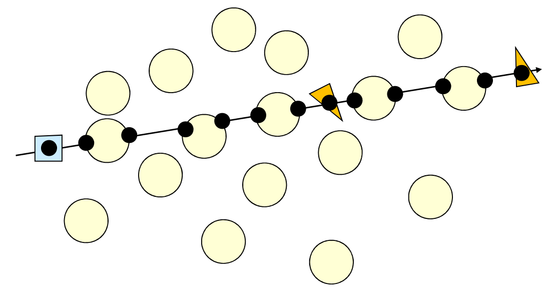


The ultimate entry and exit values are:

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

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

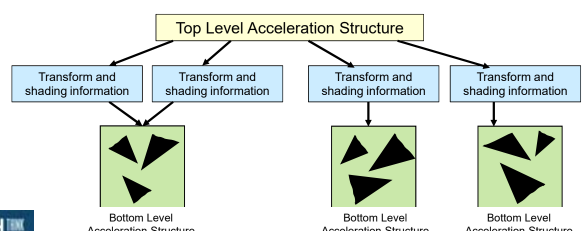
438



Acceleration Structures

439

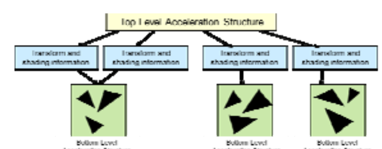
- Bottom-level Acceleration Structure (BLAS) holds the vertex data and is built from vertex and index VkBuffers
- The BLAS can also hold transformations, but it looks like usually the BLAS holds vertices in the original Model Coordinates.
- Top-level Acceleration Structure (TLAS) holds a pointer to elements 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

440

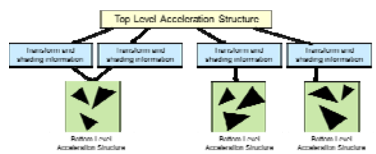
```
vkCreateAccelerationStructureKHR                    BottomLevelAccelerationStructure;

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, IN &vasci, PALLOCATOR, OUT &BottomLevelAcceleraionrStructure );
```



Creating *Top* Level Acceleration Structures

441

```
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, &TopLevelAcceleraionrStructure );
```



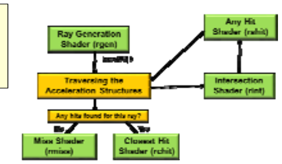
Ray Generation Shader

442

Gets all of the rays going and writes the final color to the pixel

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

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

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



A New Built-in Function

443

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

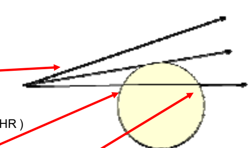
In Vulkan terms:
gl_WorldRayOriginKHR = (x_0,y_0,z_0)
gl_HitKHR = t
gl_WorldRayDirectionKHR = (dx,dy,dz)

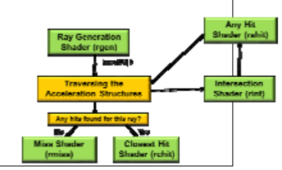
Intersection Shader

444

Intersect a ray with an arbitrary 3D object.
Passes data to the Any Hit shader.
There is a built-in ray-triangle Intersection Shader.

```
hitAttributeKHR  vec3  attribs

void main( )
{
        SpherePrimitive sph = spheres[ gl_PrimitiveID ];
        vec3 orig = gl_WorldRayOriginKHR;
        vec3 dir = normalize( gl_WorldRayDirectionKHR );
          . . .
        float discr = b*b – 4.*a*c;
        if( discr < 0. )
                return;

        float  tmp = ( -b - sqrt(discr) ) / (2.*a);
        if( gl_RayTminKHR < tmp  &&  tmp < gl_RayTmaxKHR )
        {
                vec3 p = orig + tmp * dir;
                attribs = p;
                reportIntersectionKHR( tmp, 0 );
                return;
        }
        tmp = ( -b + sqrt(discr) ) / (2.*a);
        if( gl_RayTminKHR < tmp  &&  tmp < gl_RayTmaxKHR )
        {
                vec3 p = orig + tmp * dir;
                attribs = p;
                reportIntersectionKHR( tmp, 0 );
                return;
        }
}
```



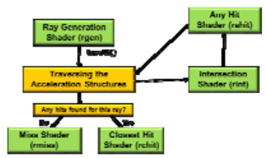


Miss Shader

445

Handle a ray that doesn't hit *any* objects

```
rayPayloadKHR myPayLoad
{
	vec4 color;
};

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



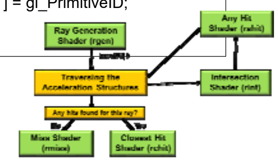
mjb – July 24, 2020

Any Hit Shader

446

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

```
layout( binding = 4, set = 0) buffer outputProperties
{
	float  outputValues[ ];
}  outputData;

layout(location = 0) rayPayloadInKHR uint outputId;
layout(location = 1) rayPayloadInKHR uint hitCounter;
hitAttributeKHR  vec 3 attribs;

void
main( )
{
	outputData.outputValues[ outputId + hitCounter ] = gl_PrimitiveID;
	hitCounter = hitCounter + 1;
}
```



mjb – July 24, 2020

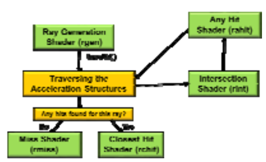
Closest Hit Shader

447

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

```
rayPayloadKHR myPayLoad
{
	vec4 color;
};

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

In Vulkan terms:
gl_WorldRayOriginKHR = (x_0,y_0,z_0)
gl_HitKHR = t
gl_WorldRayDirectionKHR =
(dx,dy,dz)



mjb – July 24, 2020

Other New Built-in Functions

448

void **terminateRayKHR**( );

void **ignoreIntersectionKHR**( );

Loosely equivalent to "discard"

void **reportIntersectionKHR**(float hit, uint hitKind);

mjb – July 24, 2020

Ray Trace Pipeline Data Structure

449

```
VkPipeline                  RaytracePipeline;
VkPipelineLayout            PipelineLayout;

VkPipelineLayoutCreateInfo              vplci;
        vplci.sType                     = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        vplci.pNext                     = nullptr;
        vplci.flags                     = 0;
        vplci.setLayoutCount            = 1;
        vplci.pSetLayouts               = &descriptorSetLayout;
        vplci.pushConstantRangeCount    = 0;
        vplci.pPushConstantRanges       = nullptr;

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

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

result = vkCreateRayTracingPipelinesKHR( LogicalDevice, PALLOCATOR, 1, IN &rvrtpci, nullptr, OUT &RaytracePipeline);
```

mjb – July 24, 2020

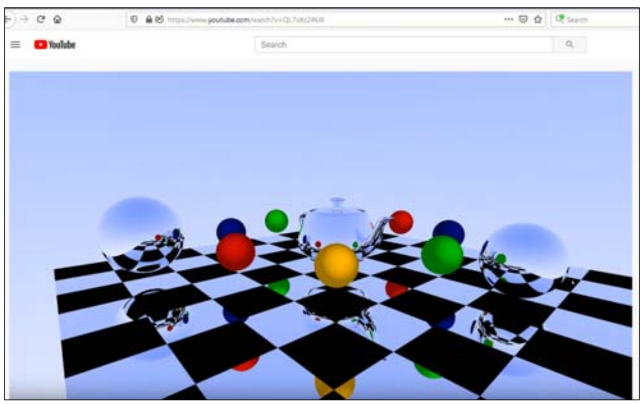
The Trigger comes from the Command Buffer: vlCmdBindPipeline( ) and vkCmdTraceRaysKHR( )

450

```
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 );,
```

mjb – July 24, 2020

451



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

mjb – July 24, 2020

452

Oregon State University Computer Graphics

Introduction to the Vulkan Computer Graphics API

Mike Bailey

mjb@cs.oregonstate.edu

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

FULL.pptx

mjb – July 24, 2020