Introduction

Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

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!

Oregon State University

Second, thanks to NVIDIA for all of their support!

NVIDIA

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

KHRONOS GROUP

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

Oregon State University
Computer Graphics

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

Oregon State
University
Computer Graphics

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

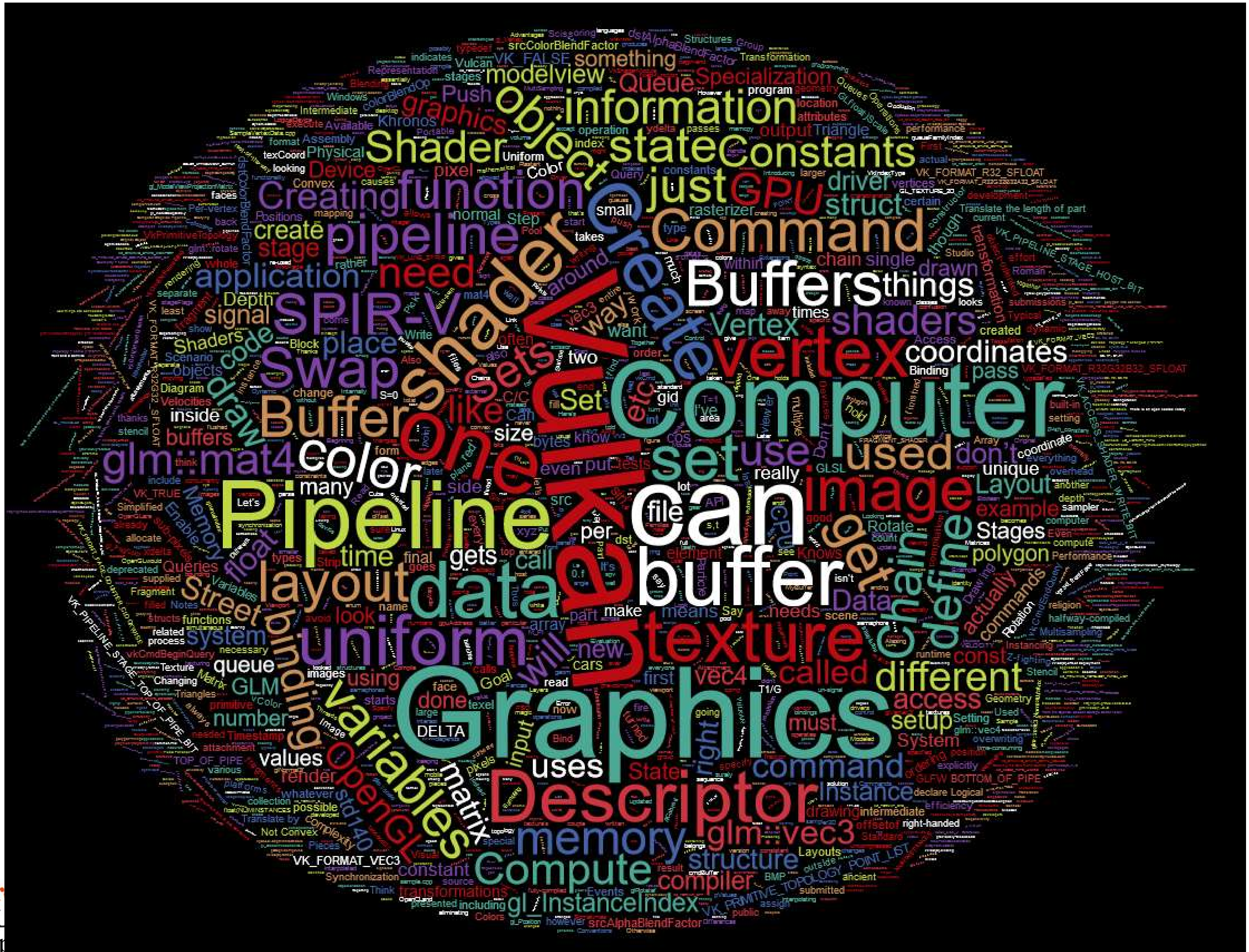
2020: Vulkan 1.2

Oregon State University
Computer Graphics

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

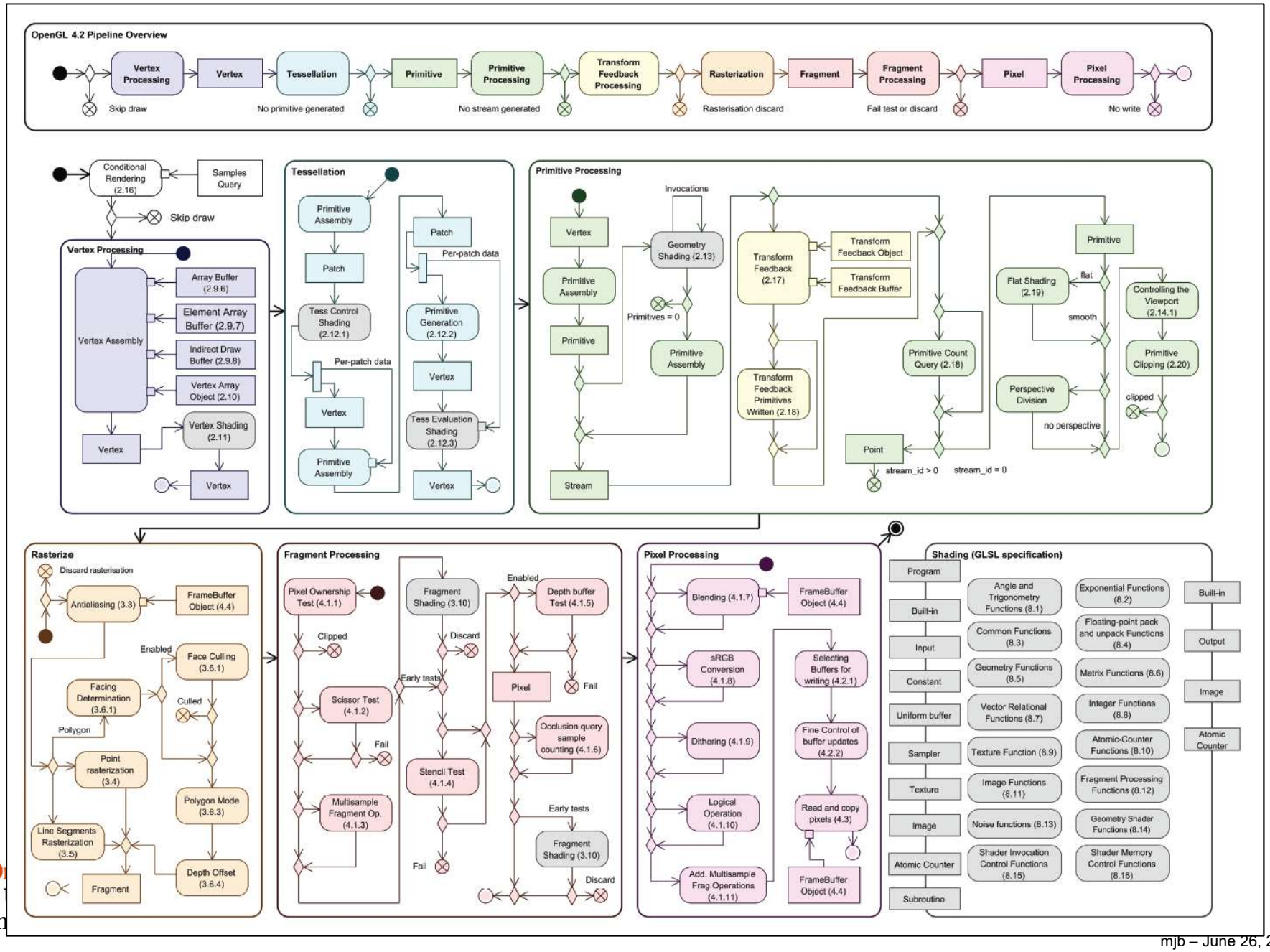
1. Performance
2. Performance
3. Performance

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	?	~484GBs	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.

Playing "Where's Waldo" with Khronos Membership

Oregon State University
Computer Graphics

Who's Been Specifically Working on Vulkan?

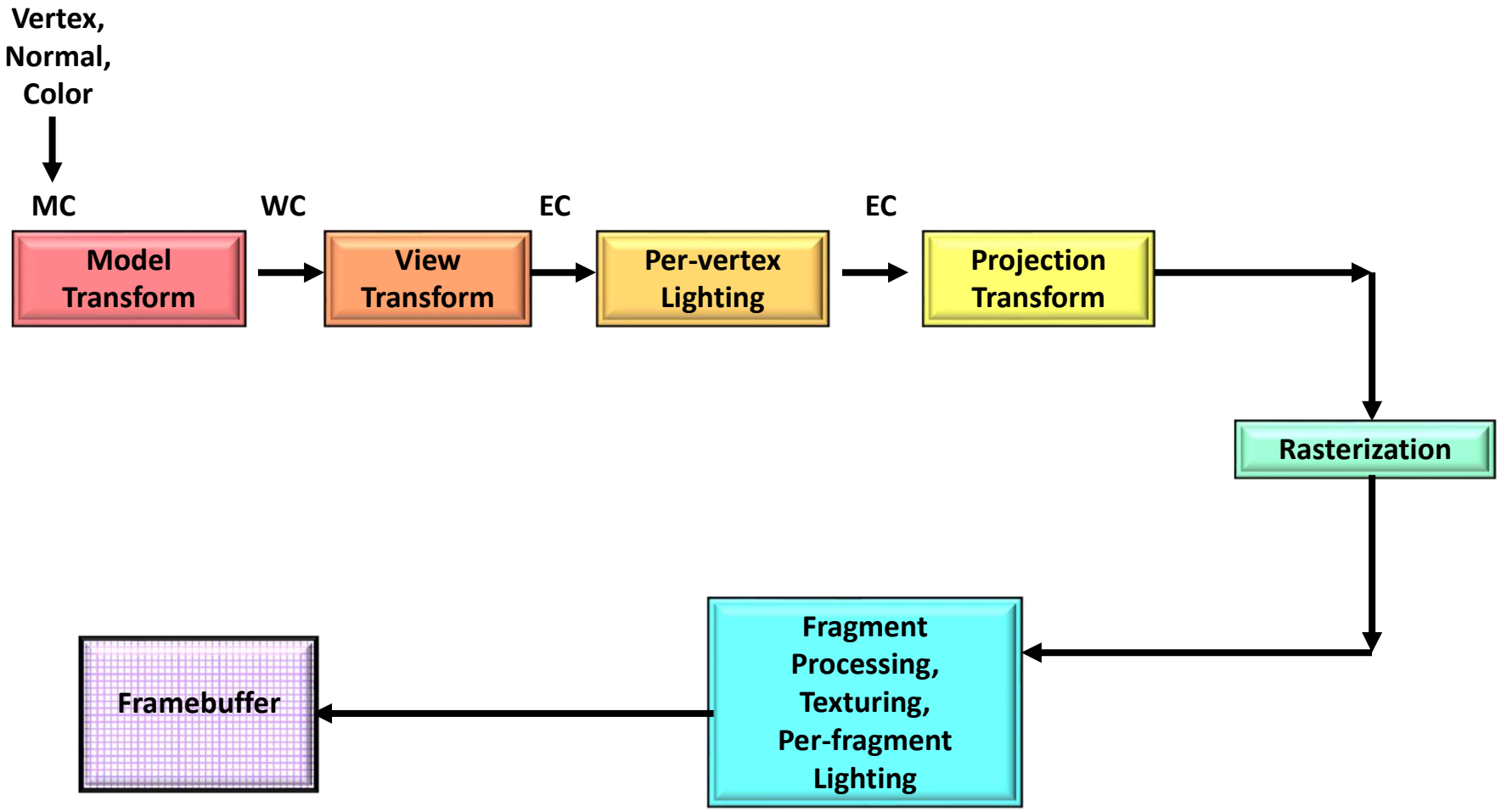
Vulkan

- 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

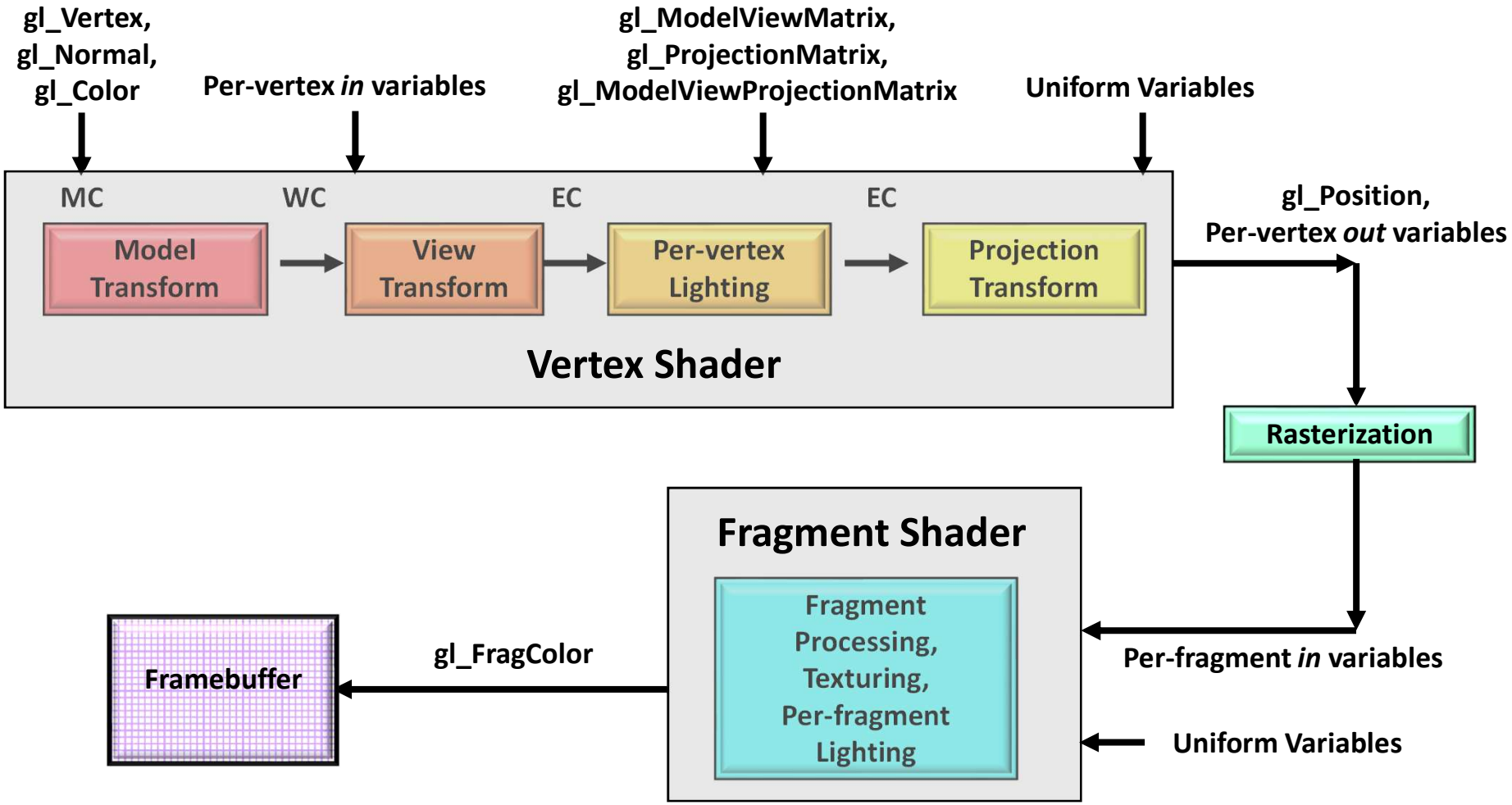
- More low-level information must be provided (by you!) in the application, rather than the driver
- Screen coordinate system is Y-down
- No “current state”, at least not one maintained by the driver
- All of the things that we have talked about being ***deprecated*** in OpenGL are *really* **deprecated** 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



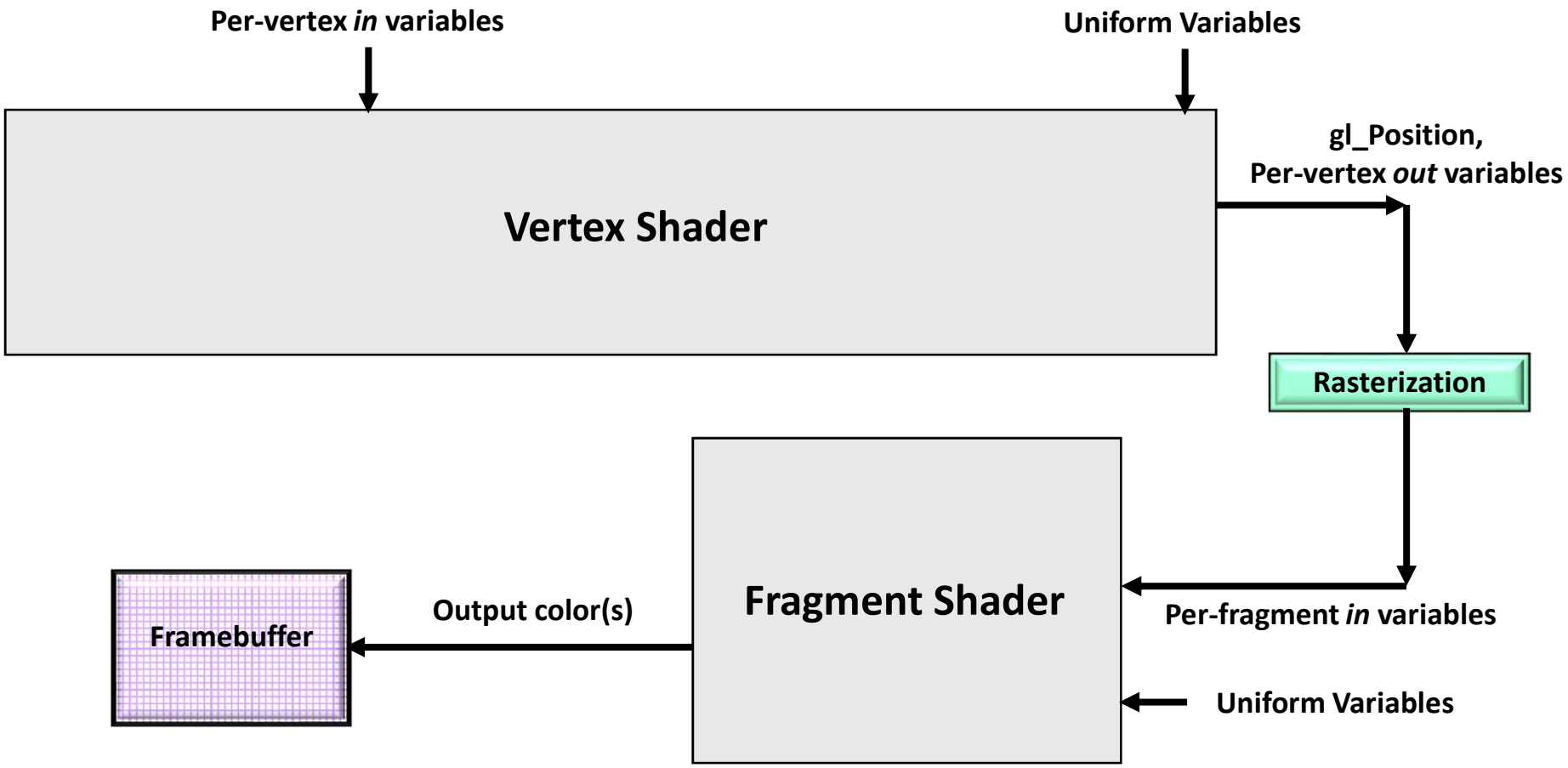
MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

The Basic Computer Graphics Pipeline, Shader-style



MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

The Basic Computer Graphics Pipeline, Vulkan-style



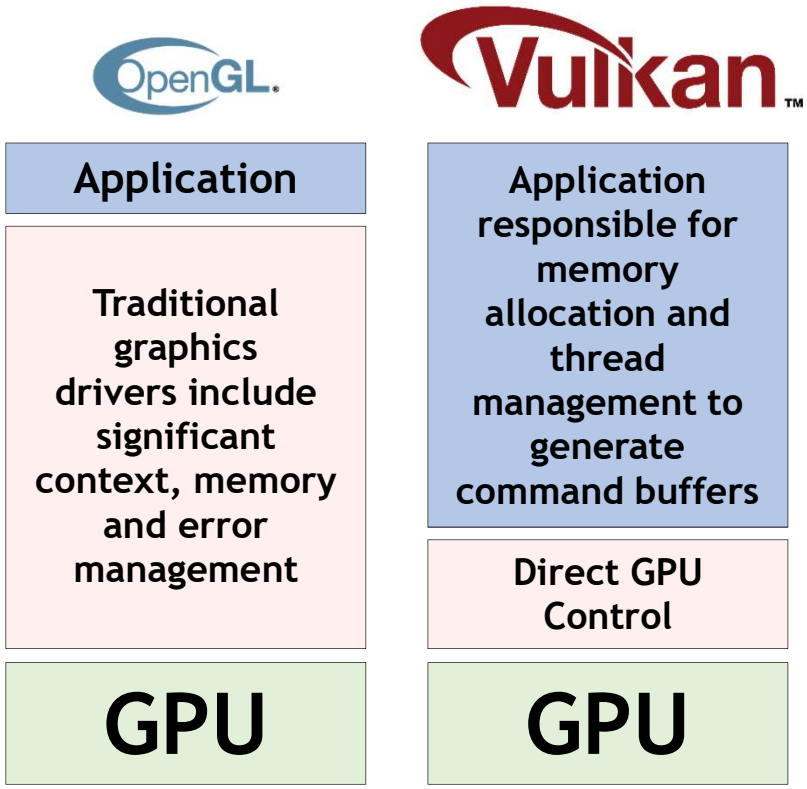
Moving part of the driver into the application

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets



Khronos Group

Simpler drivers for low-overhead efficiency and cross vendor portability

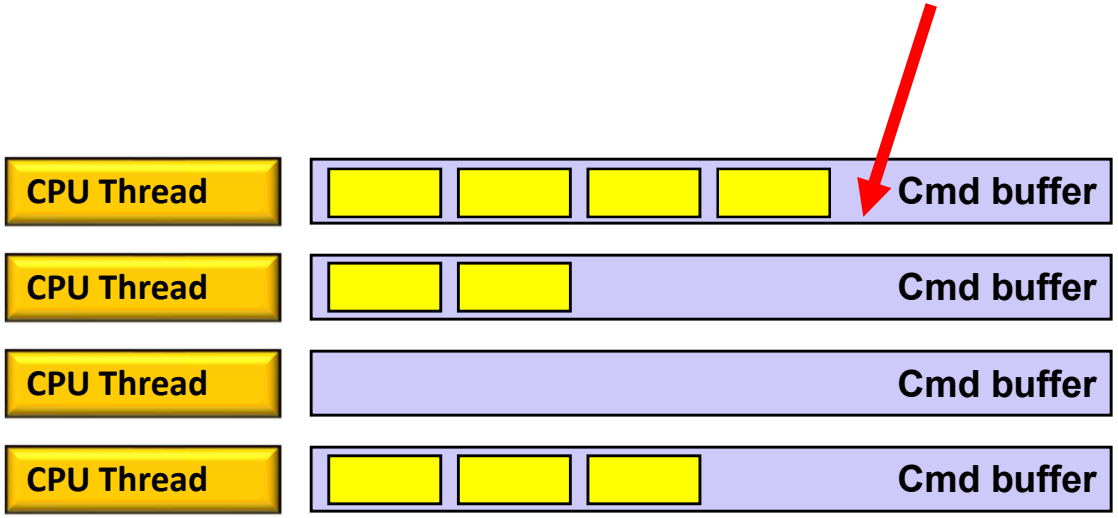
Layered architecture so validation and debug layers can be unloaded when not needed

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

Unified API for mobile, desktop, console and embedded platforms

Vulkan Highlights: Command Buffers

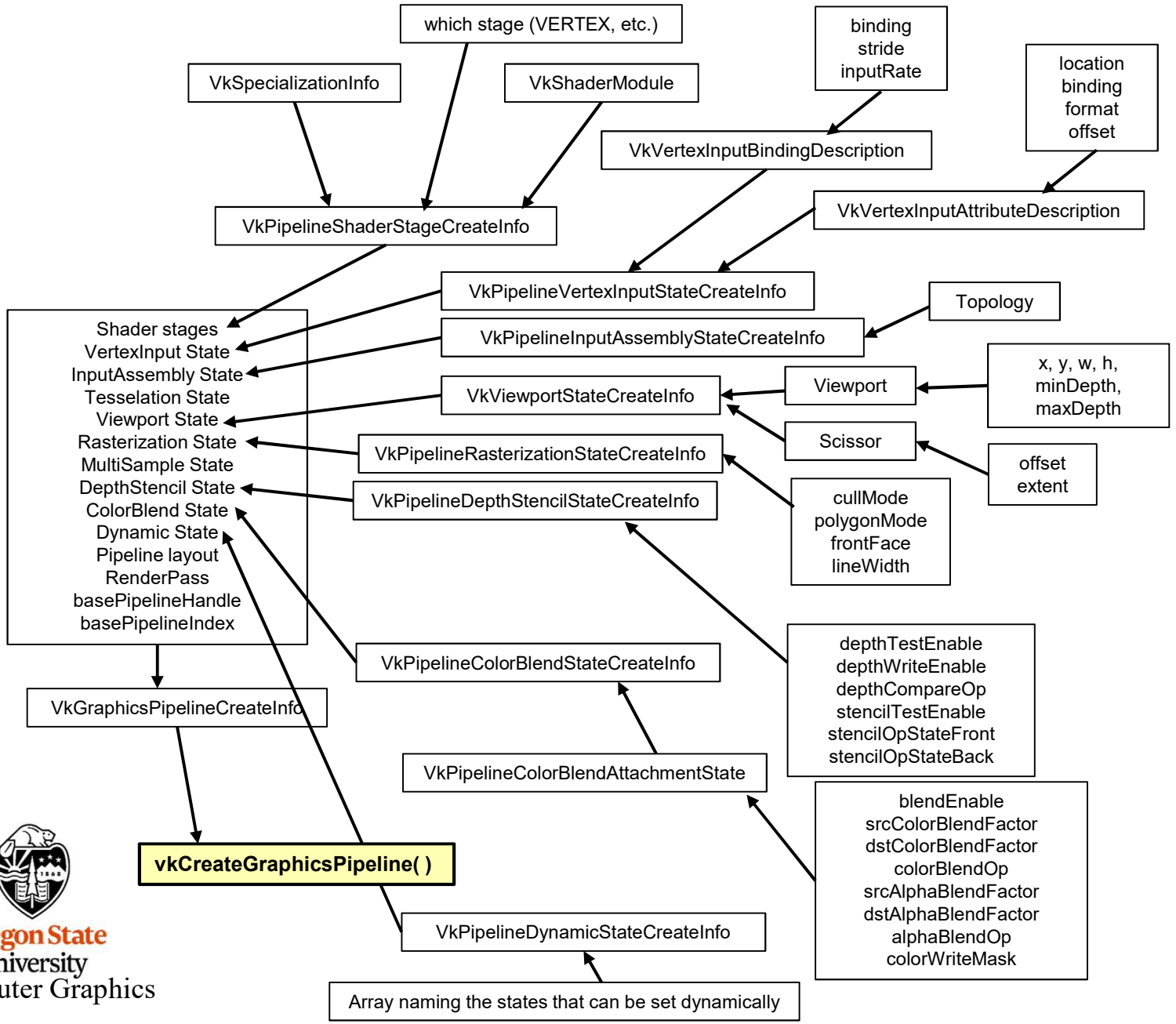
- Graphics commands are sent to command buffers
- E.g., *vkCmdDoSomething(cmdBuffer, …);*
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed to Queues when the application wants them to be flushed
- Each command buffer can be filled from a different thread



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



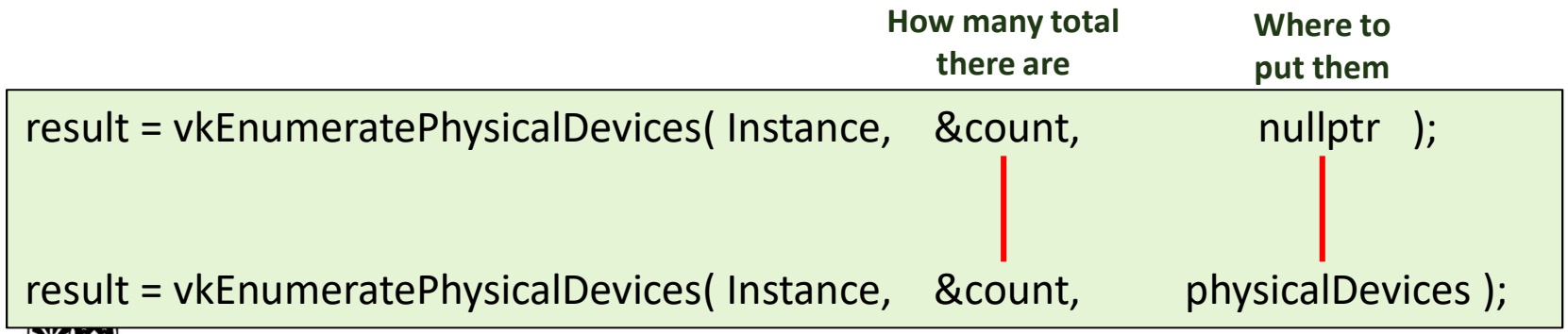
Oregon State University
Computer Graphics

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):



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!

Vulkan 1.1 Reference Guide — Page 1

Vulkan® is a graphics and compute API consisting of procedures and functions to specify shader programs, compute kernels, objects, and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. Vulkan is also a pipeline with programmable and state-driven fixed-function stages that are invoked by a set of specific drawing operations.

Specification and additional resources at **www.khronos.org/vulkan**

Color coded names as follows: function names and structure names
[n.n.n] Indicates sections and text in the Vulkan API 1.1 Specification.
A# Indicates a page in this reference guide for more information.
=0 Indicates reserved for future use.
pNext must either be NULL, or point to a valid structure which extends the base structure according to the valid usage rules of the base structure.

Return Codes [2.7.3]
Return codes are reported via VkResult return values.

Success Codes
Success codes are non-negative.
VK_SUCCESS, VK_NOT_READY, VK_TIMEOUT, VK_EVENT_{SET, RESET}, VK_INCOMPLETE, VK_SUBOPTIMAL_KHR

Error Codes
Error codes are negative.
VK_ERROR_OUT_OF_{HOST, DEVICE}_MEMORY
VK_ERROR_{INITIALIZATION, MEMORY_MAP}_FAILED
VK_ERROR_DEVICE_LOST
VK_ERROR_{EXTENSION, FEATURE, LAYER}_NOT_PRESENT
VK_ERROR_INCOMPATIBLE_DRIVER
VK_ERROR_TOO_MANY_OBJECTS
VK_ERROR_FORMAT_NOT_SUPPORTED
VK_ERROR_FRAGMENTED_POOL
VK_ERROR_OUT_OF_POOL_MEMORY
VK_ERROR_INVALID_EXTERNAL_HANDLE
VK_ERROR_SURFACE_LOST_KHR
VK_ERROR_NATIVE_WINDOW_IN_USE_KHR
VK_ERROR_OUT_OF_DATE_KHR
VK_ERROR_INCOMPATIBLE_DISPLAY_KHR

Command Function Pointers and Instances [3]

Command Function Pointers [3.1]
```
PFN_vkVoidFunction vkGetInstanceProcAddr(
    VkInstance instance, const char* pName);
PFN_vkVoidFunction vkGetDeviceProcAddr(
    VkDevice device, const char* pName);
  PFN_vkVoidFunction is:
    typedef void(VKAPI_PTR* PFN_vkVoidFunction)(void);
```

Instances [3.2]
```
VkResult vkEnumerateInstanceVersion(
    uint32_t* pApiVersion);
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator, A12
    VkInstance* pInstance);
typedef struct VkInstanceCreateInfo {
    VkStructureType sType; A15
    const void* pNext;
    VkInstanceCreateFlags flags; =0
    const VkApplicationInfo* pApplicationInfo;
    uint32_t enabledLayerCount;
    const char* const* ppEnabledLayerNames;
    uint32_t enabledExtensionCount;
    const char* const* ppEnabledExtensionNames;
} VkInstanceCreateInfo;
typedef struct VkApplicationInfo {
    VkStructureType sType; A15
    const void* pNext;
    const char* pApplicationName;
    uint32_t applicationVersion;
    const char* pEngineName;
    uint32_t engineVersion;
    uint32_t apiVersion;
} VkApplicationInfo;
void vkDestroyInstance(
    VkInstance instance,
    const VkAllocationCallbacks* pAllocator); A12
```

Devices and Queues [4]

Physical Devices [4.1]
```
VkResult vkEnumeratePhysicalDevices(
    VkInstance instance,
    uint32_t* pPhysicalDeviceCount,
    VkPhysicalDevice* pPhysicalDevices);
void vkGetPhysicalDeviceProperties(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties* pProperties); A14
void vkGetPhysicalDeviceProperties2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties2* pProperties);
typedef struct VkPhysicalDeviceProperties2 {
    VkStructureType sType; A15
    void* pNext;
    VkPhysicalDeviceProperties properties; A14
} VkPhysicalDeviceProperties2;
  pNext must be NULL or point to one of:
    VkPhysicalDeviceIDProperties A14
    VkPhysicalDeviceMaintenance3Properties A14
    VkPhysicalDeviceMultiviewProperties A14
    VkPhysicalDevicePointClippingProperties A14
    VkPhysicalDeviceProtectedMemoryProperties A15
    VkPhysicalDeviceSubgroupProperties A15
void vkGetPhysicalDeviceQueueFamilyProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pQueueFamilyPropertyCount,
    VkQueueFamilyProperties*
      pQueueFamilyProperties);
void vkGetPhysicalDeviceQueueFamilyProperties2(
    VkPhysicalDevice physicalDevice,
    uint32_t* pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2*pQueueFamilyProperties);
typedef struct VkQueueFamilyProperties {
    VkQueueFlags queueFlags;
    uint32_t queueCount;
    uint32_t timestampValidBits;
    VkExtent3D minImageTransferGranularity; A12
} VkQueueFamilyProperties;
  queueFlags:
    VK_QUEUE_X_BIT where X is GRAPHICS, COMPUTE,
    TRANSFER, PROTECTED, SPARSE_BINDING
typedef struct VkQueueFamilyProperties2 {
    VkStructureType sType; A15
    void* pNext; VkQueueFamilyProperties
      queueFamilyProperties;
} VkQueueFamilyProperties2;
```

Devices [4.2]
```
VkResult vkEnumeratePhysicalDeviceGroups(
    VkInstance instance,
    uint32_t* pPhysicalDeviceGroupCount,
    VkPhysicalDeviceGroupProperties*
      pPhysicalDeviceGroupProperties);
typedef struct VkPhysicalDeviceGroupProperties {
    VkStructureType sType; A15
    void* pNext;
    uint32_t physicalDeviceCount;
    VkPhysicalDevice physicalDevices[
      VK_MAX_DEVICE_GROUP_SIZE];
    VkBool32 subsetAllocation;
} VkPhysicalDeviceGroupProperties;
```

Device Creation [4.2.1]
```
VkResult vkCreateDevice(
    VkPhysicalDevice physicalDevice,
    const VkDeviceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator, A12
    VkDevice* pDevice);
typedef struct VkDeviceCreateInfo {
    VkStructureType sType; A15
    const void* pNext;
    VkDeviceCreateFlags flags; =0
    uint32_t queueCreateInfoCount;
    const VkDeviceQueueCreateInfo* pQueueCreateInfos;
    uint32_t enabledLayerCount;
    const char* const* ppEnabledLayerNames;
    uint32_t enabledExtensionCount;
    const char* const* ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures* pEnabledFeatures; A14
} VkDeviceCreateInfo;
  pNext must be NULL or point to one of:
    VkDeviceGroupDeviceCreateInfo A12
    VkPhysicalDevice16BitStorageFeatures A16
    VkPhysicalDeviceFeatures2 A16
    VkPhysicalDeviceMultiviewFeatures A16
    VkPhysicalDeviceProtectedMemoryFeatures A15
    VkPhysicalDeviceSamplerYcbcrConversionFeatures A15
    VkPhysicalDeviceVariablePointerFeatures A15
typedef struct VkDeviceGroupDeviceCreateInfo {
    VkStructureType sType; A15
    const void* pNext;
    uint32_t physicalDeviceCount;
    const VkPhysicalDevice* pPhysicalDevices;
} VkDeviceGroupDeviceCreateInfo;
```

Device Destruction [4.2.4]
```
void vkDestroyDevice(
    VkDevice device,
    const VkAllocationCallbacks* pAllocator); A12
```

Queues [4.3]
```
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType sType; A15
    const void* pNext;
    VkDeviceQueueCreateFlags flags;
    uint32_t queueFamilyIndex;
    uint32_t queueCount;
    const float* pQueuePriorities;
} VkDeviceQueueCreateInfo;
  flags: VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT
void vkGetDeviceQueue(VkDevice device,
    uint32_t queueFamilyIndex, uint32_t queueIndex,
    VkQueue* pQueue);
void vkGetDeviceQueue2(VkDevice device,
    const VkDeviceQueueInfo2* pQueueInfo,
    VkQueue* pQueue);
typedef struct VkDeviceQueueInfo2 {
    VkStructureType sType; A15
    const void* pNext;
    VkDeviceQueueCreateFlags flags;
    uint32_t queueFamilyIndex; uint32_t queueIndex;
} VkDeviceQueueInfo2;
  flags: VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT
```

Command Buffers [5]
Also see Command Buffer Lifecycle diagram. A5

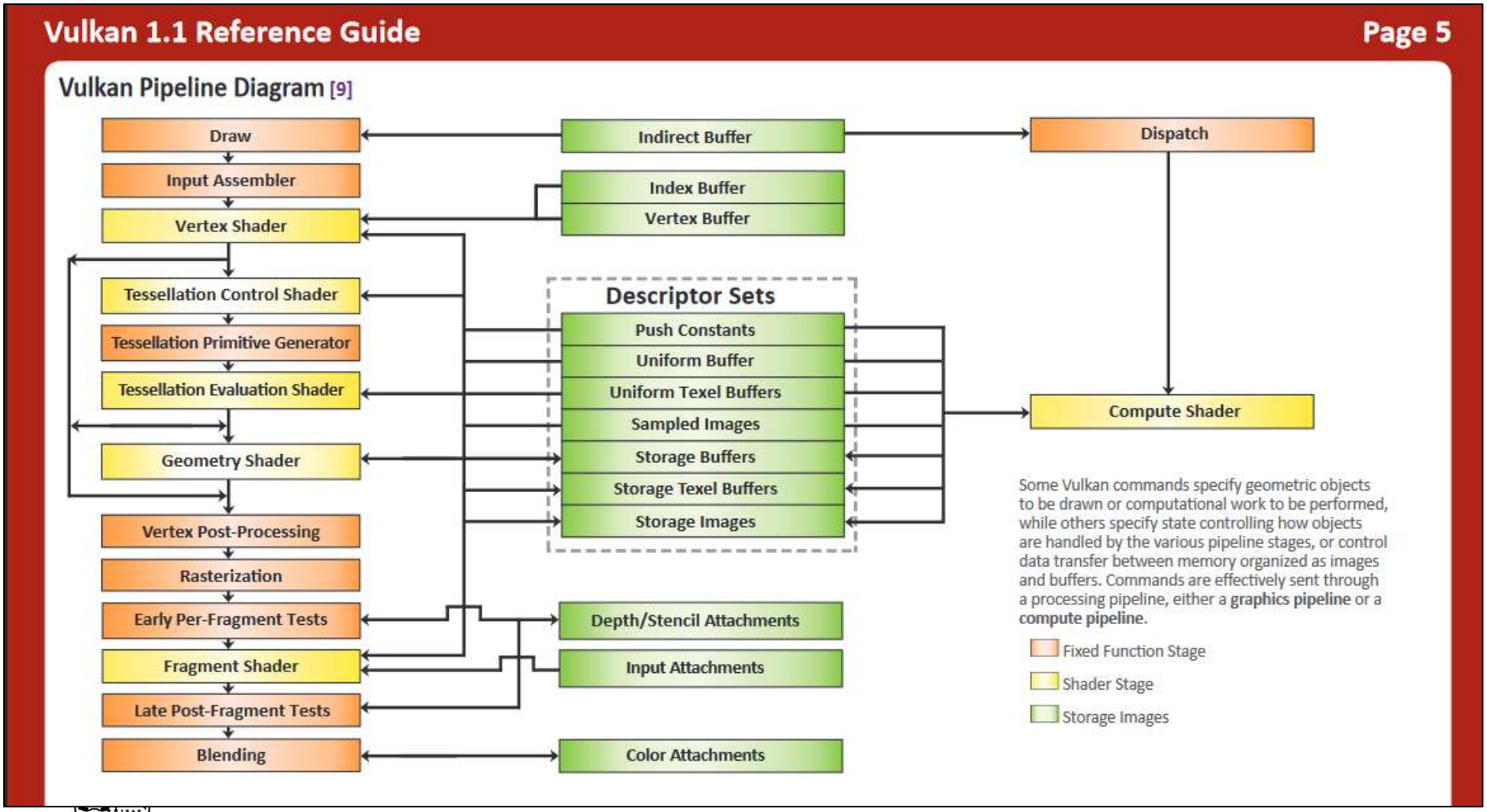
Command Pools [5.2]
```
VkResult vkCreateCommandPool(
    VkDevice device,
    const VkCommandPoolCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator, A12
    VkCommandPool* pCommandPool);
typedef struct VkCommandPoolCreateInfo {
    VkStructureType sType; A15
    const void* pNext;
    VkCommandPoolCreateFlags flags;
    uint32_t queueFamilyIndex;
} VkCommandPoolCreateInfo;
  flags: VK_COMMAND_POOL_CREATE_X_BIT where X is
    PROTECTED, RESET_COMMAND_BUFFER, TRANSIENT
void vkTrimCommandPool(VkDevice device,
    VkCommandPool commandPool,
    VkCommandPoolTrimFlags flags); =0
VkResult vkResetCommandPool(
    VkDevice device, VkCommandPool commandPool,
    VkCommandPoolResetFlags flags);
  flags: VK_COMMAND_POOL_RESET_RELEASE_-
    RESOURCES_BIT
void vkDestroyCommandPool(
    VkDevice device, VkCommandPool commandPool,
    const VkAllocationCallbacks* pAllocator); A12
```

Continued on next page >

Oregon State University Computer Graphics

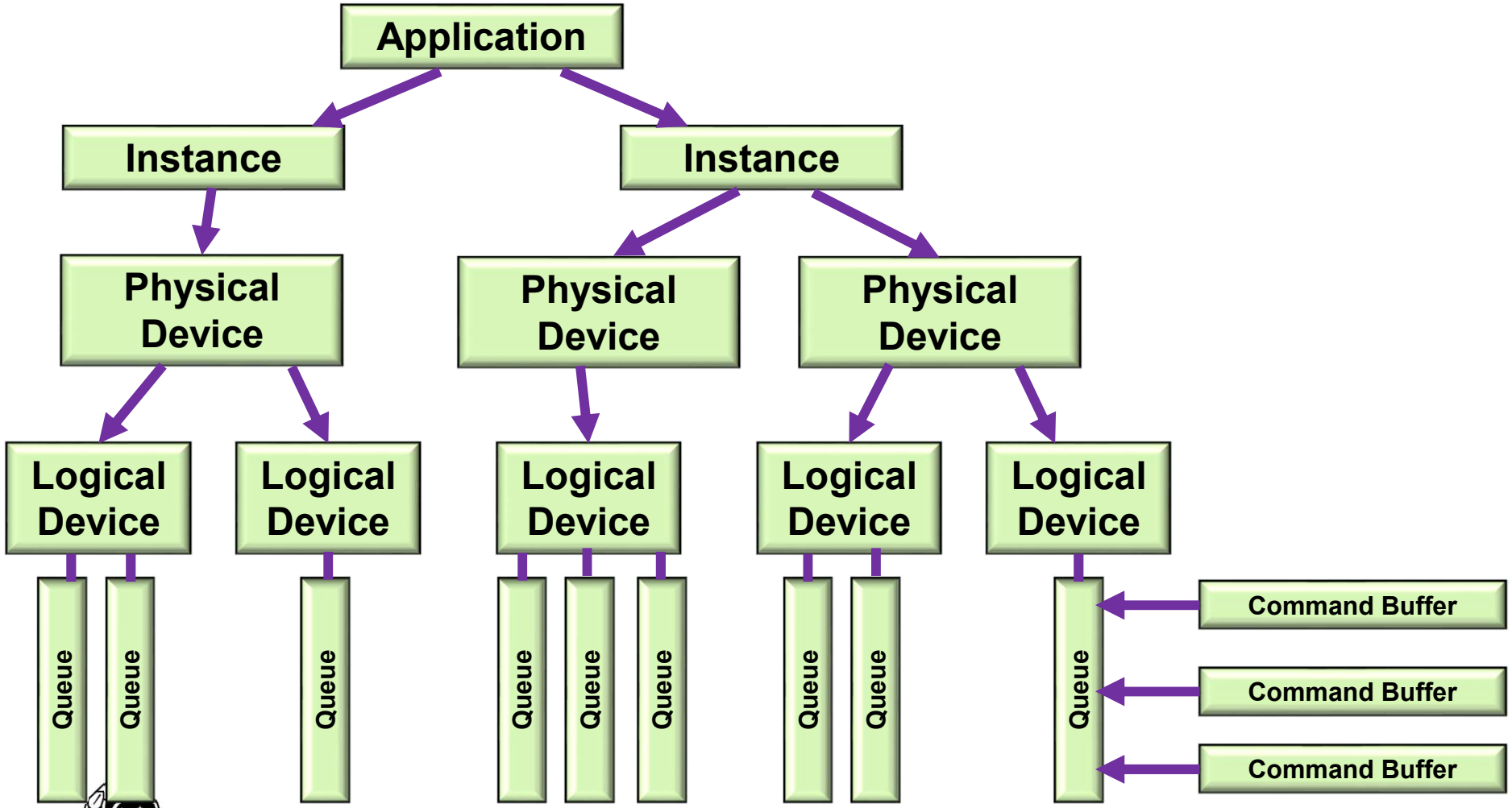
https://www.khronos.org/files/vulkan11-reference-guide.pdf

Vulkan Quick Reference Card

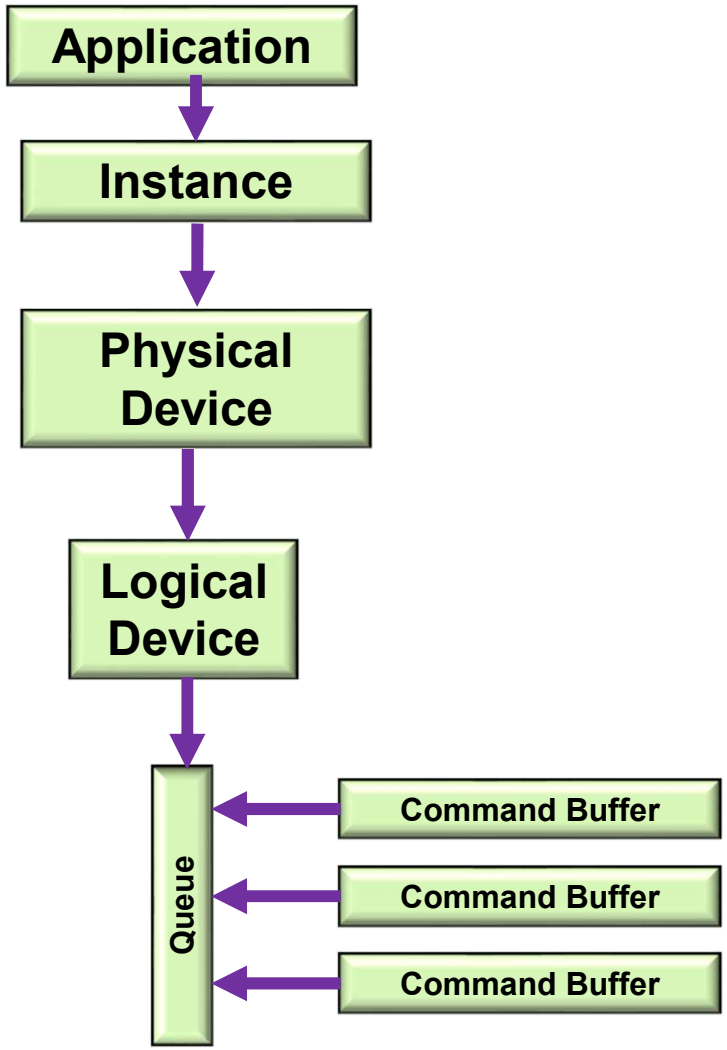


https://www.khronos.org/files/vulkan11-reference-guide.pdf

Vulkan Highlights: Overall Block Diagram



Vulkan Highlights: a More Typical Block Diagram



Steps in Creating Graphics using Vulkan

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

Vulkan GPU Memory

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

Vulkan Render Passes

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

Vulkan Compute Shaders

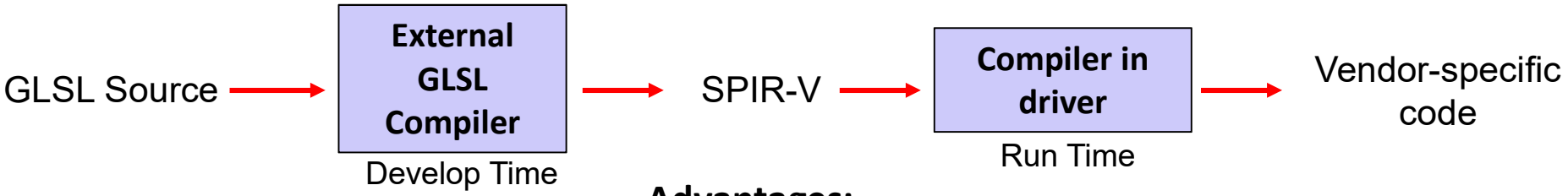
- Compute pipelines are allowed, but they are treated as something special (just like OpenGL treats them)
- Compute passes are launched through dispatches
- Compute command buffers can be run asynchronously

Vulkan Synchronization

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

Vulkan Shaders

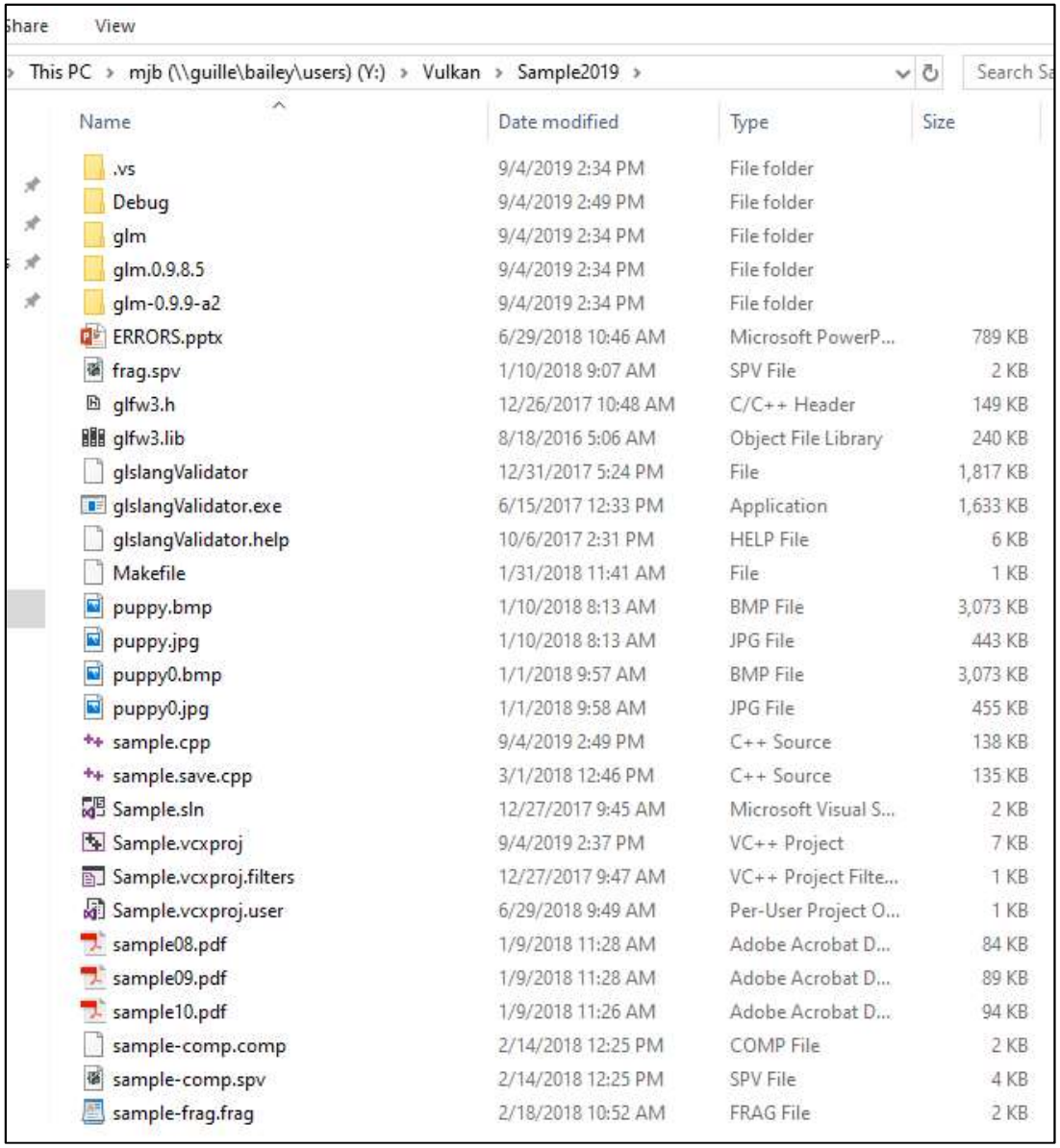
- GLSL is the same as before … almost
- For places it's not, an implied
  #define VULKAN 100
  is automatically supplied by the compiler
- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation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



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



Share View

This PC › mjb (\\guille\bailey\users) (Y:) › Vulkan › Sample2019 ›

Name	Date modified	Type	Size
.vs	9/4/2019 2:34 PM	File folder	
Debug	9/4/2019 2:49 PM	File folder	
glm	9/4/2019 2:34 PM	File folder	
glm.0.9.8.5	9/4/2019 2:34 PM	File folder	
glm-0.9.9-a2	9/4/2019 2:34 PM	File folder	
ERRORS.pptx	6/29/2018 10:46 AM	Microsoft PowerP...	789 KB
frag.spv	1/10/2018 9:07 AM	SPV File	2 KB
glfw3.h	12/26/2017 10:48 AM	C/C++ Header	149 KB
glfw3.lib	8/18/2016 5:06 AM	Object File Library	240 KB
glslangValidator	12/31/2017 5:24 PM	File	1,817 KB
glslangValidator.exe	6/15/2017 12:33 PM	Application	1,633 KB
glslangValidator.help	10/6/2017 2:31 PM	HELP File	6 KB
Makefile	1/31/2018 11:41 AM	File	1 KB
puppy.bmp	1/10/2018 8:13 AM	BMP File	3,073 KB
puppy.jpg	1/10/2018 8:13 AM	JPG File	443 KB
puppy0.bmp	1/1/2018 9:57 AM	BMP File	3,073 KB
puppy0.jpg	1/1/2018 9:58 AM	JPG File	455 KB
sample.cpp	9/4/2019 2:49 PM	C++ Source	138 KB
sample.save.cpp	3/1/2018 12:46 PM	C++ Source	135 KB
Sample.sln	12/27/2017 9:45 AM	Microsoft Visual S...	2 KB
Sample.vcxproj	9/4/2019 2:37 PM	VC++ Project	7 KB
Sample.vcxproj.filters	12/27/2017 9:47 AM	VC++ Project Filte...	1 KB
Sample.vcxproj.user	6/29/2018 9:49 AM	Per-User Project O...	1 KB
sample08.pdf	1/9/2018 11:28 AM	Adobe Acrobat D...	84 KB
sample09.pdf	1/9/2018 11:28 AM	Adobe Acrobat D...	89 KB
sample10.pdf	1/9/2018 11:26 AM	Adobe Acrobat D...	94 KB
sample-comp.comp	2/14/2018 12:25 PM	COMP File	2 KB
sample-comp.spv	2/14/2018 12:25 PM	SPV File	4 KB
sample-frag.frag	2/18/2018 10:52 AM	FRAG File	2 KB

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