

Introduction to



2023.08.09

Toshiba

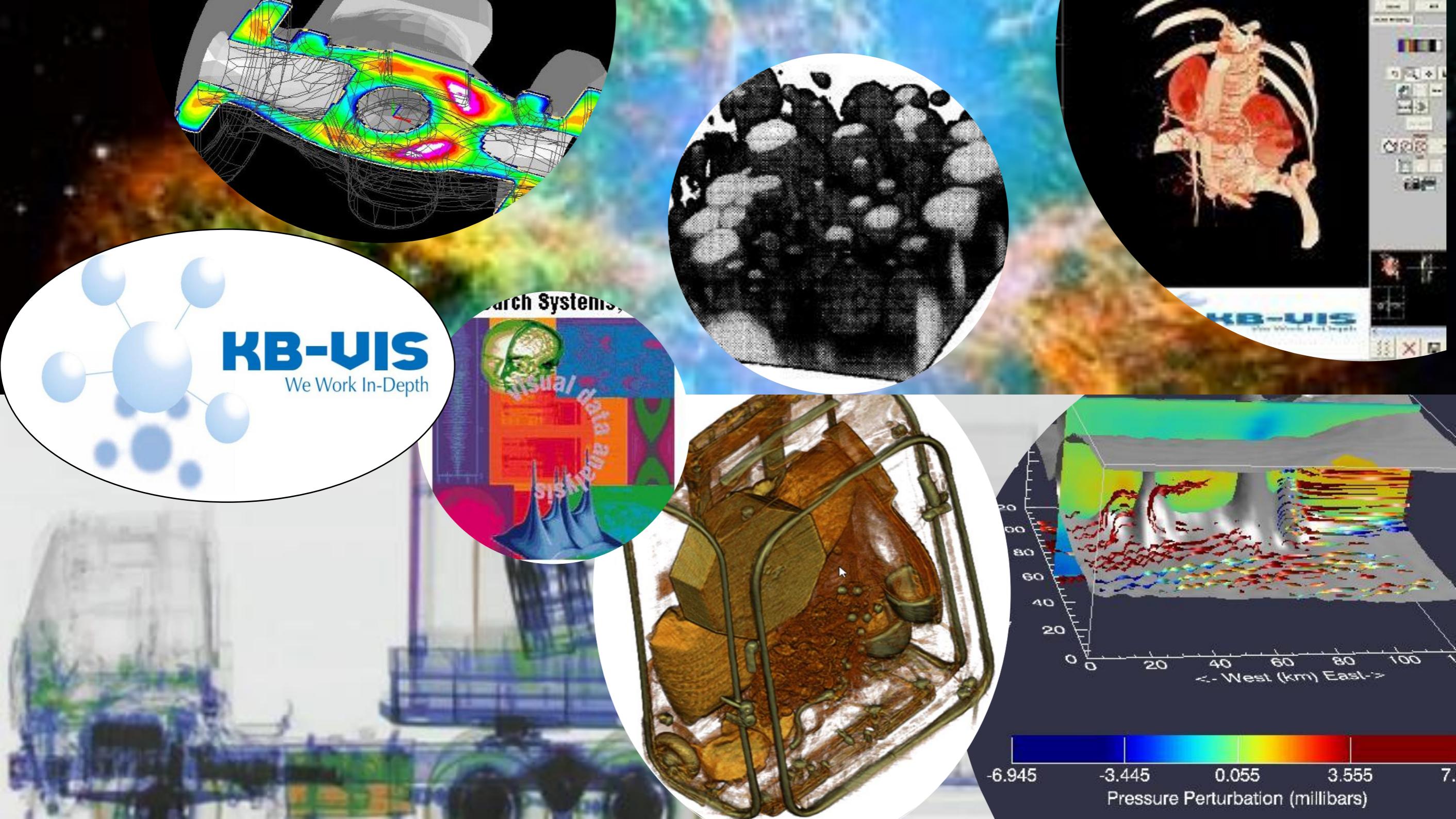


Karthik Bala
KBVIS Software Pty. Ltd. (Hobart, Australia)
3d@kbvis.com

Karthik Bala - Introduction



- B.E. (Hons.) Computer Science – BITS, Pilani, India
- M.S. Computer Science – Colorado State University, Fort Collins, USA
- Graphics Developer since 1995
- NVIDIA Registered Developer since 2001
- Developed OpenGL/WebGL/CUDA-based products for companies in USA, Australia, Europe, India
- Incorporated KBVIS (Private Limited) in 2005
- Creator of KBVIS' flagship product KB-Vol3D, installations in 14+ countries
- Rich experience in teaching, training, consulting, mentoring in organisations including Samsung, Honeywell, DSP Group, Triad Software, SpecIndia, Techtronix, EvolveIT, Barco, AshvaTech, CAS Tech, Manmar Tech, Walmart, Toshiba, Minetec, Merge Healthcare, Deswik Mining, Rapiscan Systems, RefleXion Medical and more.



DAY1 - Agenda

- Brief Overview of:
 - Graphics Application Software
 - Graphics Hardware Evolution – Fixed Pipeline to Programmable Hardware
 - OpenGL – Evolution and End-of-Life
- Vulkan Overview
 - Origin and Goals
 - Design Considerations
 - Ecosystem Overview
 - Overview of Vulkan Graphics Pipeline
- Introduction to Vulkan Programming
 - Setup
 - GLFW
 - Components of a Vulkan Program
 - Hello Triangle Example
 - Drawing, Coordinate System
 - Introduction to Shaders / GLSL / SPIR-V

Setup and Resources

- Course Content (will be updated progressively)

<https://github.com/kbvis3d/toshiba-vulkan-2023>

Graphics Primer

- Skim through the included introductory material to understand the primary motivation behind graphics APIs:
 - Perspective Projection and Rasterization
 - Spatial Transformations
 - The Rendering Equation
 - Depth and Transparency

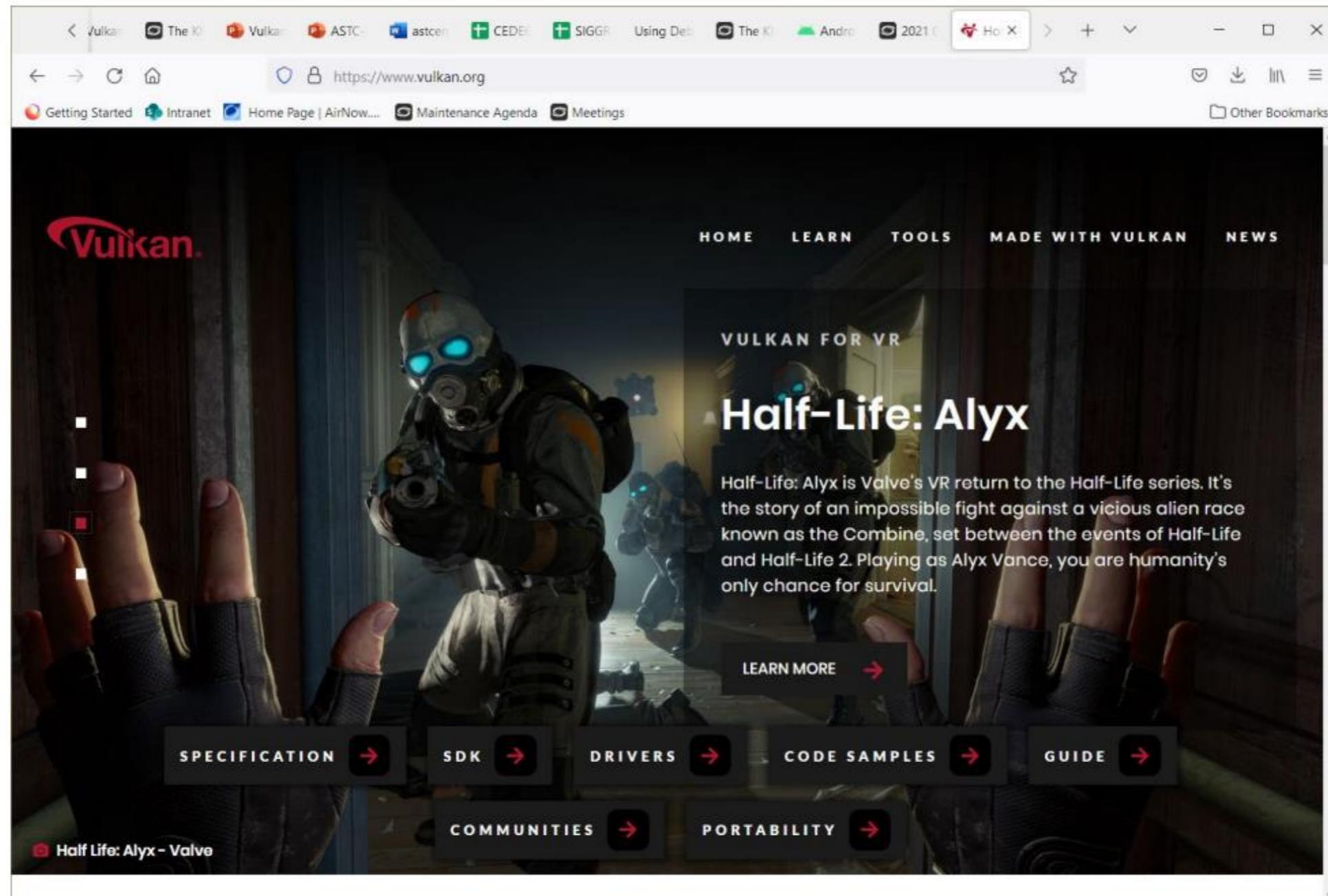
Note: The idea is to get a sense of the tasks Vulkan is meant to perform and pick up salient graphics vocabulary

For the full course material visit the CMU course page:

<http://15462.courses.cs.cmu.edu/spring2023/>



New website: <https://www.vulkan.org>



- Home page for Vulkan on the web
- Make tools and resources easier to find
- Highlight new Vulkan content
- Updated regularly

<https://www.vulkan.org>

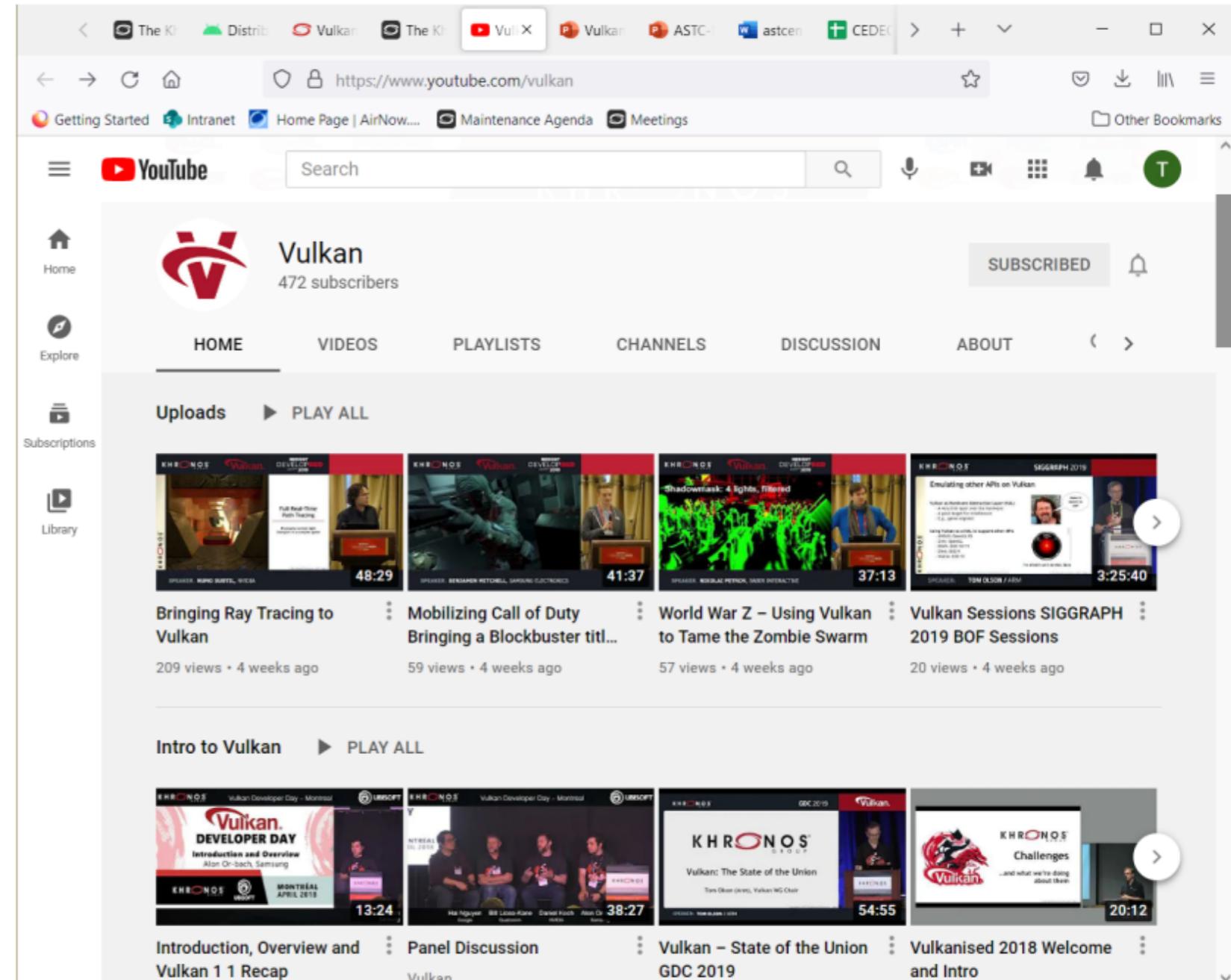
The screenshot shows a web browser window displaying the Vulkan.org website. The URL in the address bar is <https://www.vulkan.org/tools#download-these-essential-development-tools>. The page title is "SDK" and the main heading is "Download these essential development tools". Below the heading, a sub-headline reads "Essentials tools, documentation and libraries for every Vulkan developer". A section titled "Vulkan SDK for Windows, Linux, and macOS" is highlighted with a red minus sign icon. A call-to-action button labeled "FIND OUT MORE" with a right-pointing arrow is visible. To the right of the text area is a large, atmospheric image from the video game Detroit: Become Human, featuring a man in a black suit standing in a futuristic, multi-tiered stadium-like arena filled with many people in white uniforms. The image has a dark, moody aesthetic with blue and white lighting. At the bottom of the image, there is a small caption that reads "Detroit Become Human - Quantic Dream". The browser interface includes a toolbar at the top with various icons and tabs, and a vertical scroll bar on the right side of the page.

New Youtube Channel

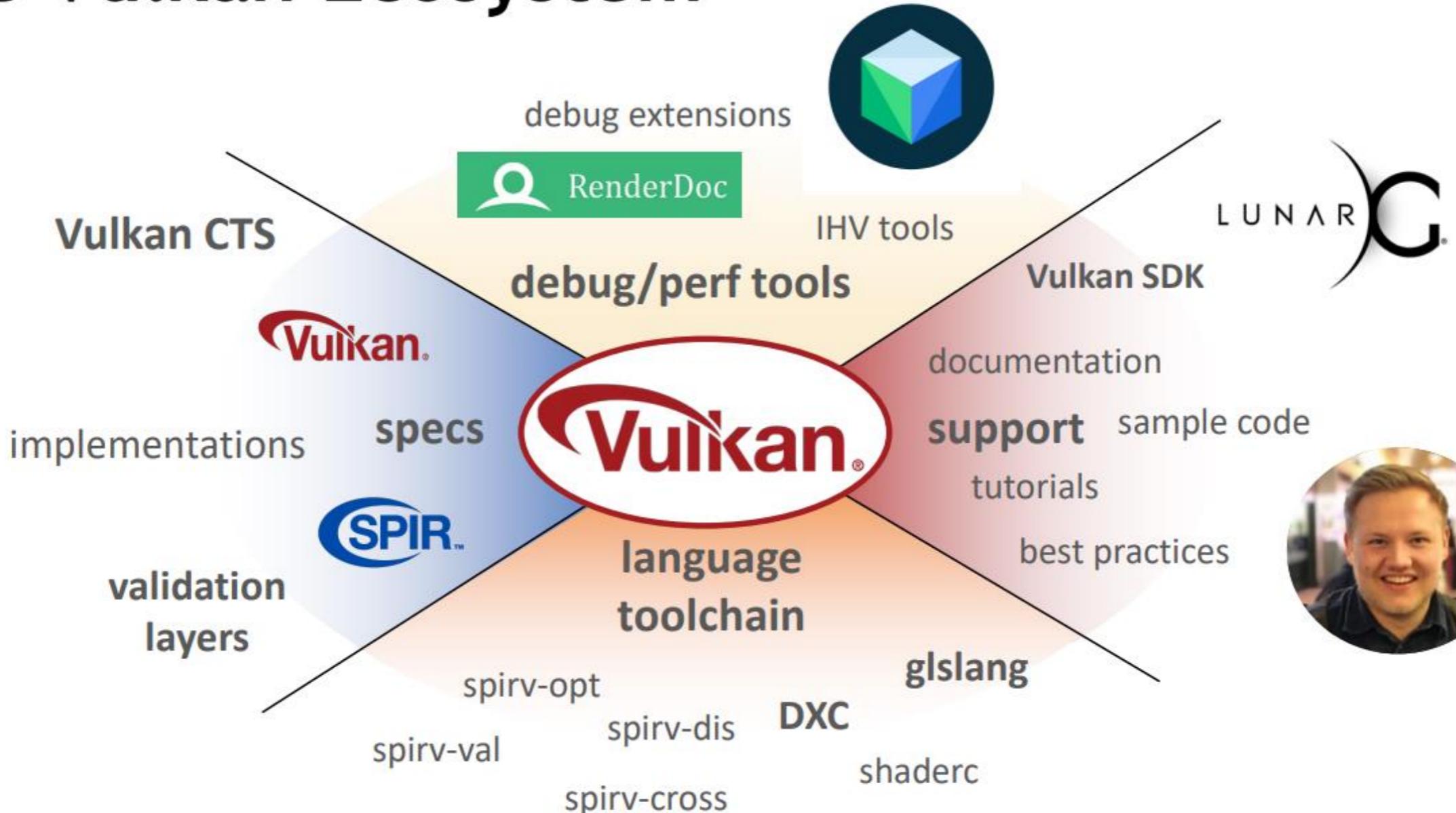
- www.youtube.com/vulkan

- Videos from Vulkan events

- GDC
- CEDEC
- SIGGRAPH
- Vulkanised!
- Reboot Develop
- ...



The Vulkan Ecosystem



Khronos Visual Computing Ecosystem

REAL-TIME 2D / 3D

Cross-platform gaming & UI
VR and AR displays
CAD and product design
Safety-critical displays



VR, VISION, NEURAL NETWORKS

VR system portability
Tracking and odometry
Scene analysis/understanding
Neural Network inferencing



3D FOR THE WEB

3D apps and games in-browser
Runtime delivery of 3D assets



PARALLEL COMPUTATION

Machine learning acceleration
Embedded vision processing
High Performance Computing (HPC)



Vulkan Samples Repository

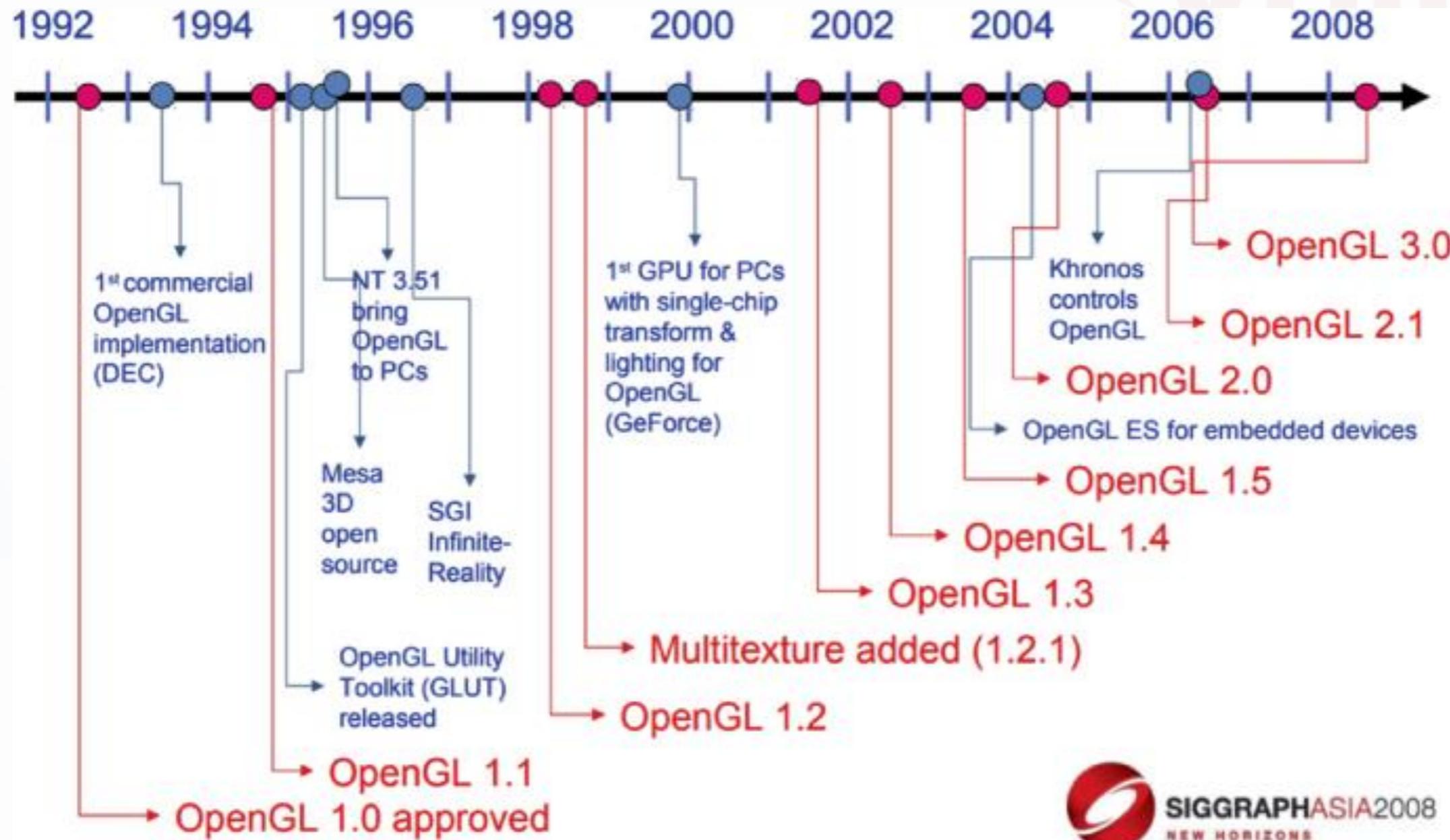
- A home for Vulkan sample code
 - Intended to help you learn to use Vulkan effectively
 - GPU, OS, and platform neutral, tested on a wide variety of implementations
 - Open Source under the Apache 2.0 license
- Three classes of samples
 - API Samples - Demonstrate how to use the API
 - Performance Samples - Show the impact of good and bad choices
 - Extension Samples - Show how to use new functionality
- Collected / maintained by Khronos Dev Rel together with
 - Community members including Sascha Willems and others
 - Khronos member ISVs and IHVs
 - Developed on GitHub - all are welcome to contribute and participate

What is Vulkan?

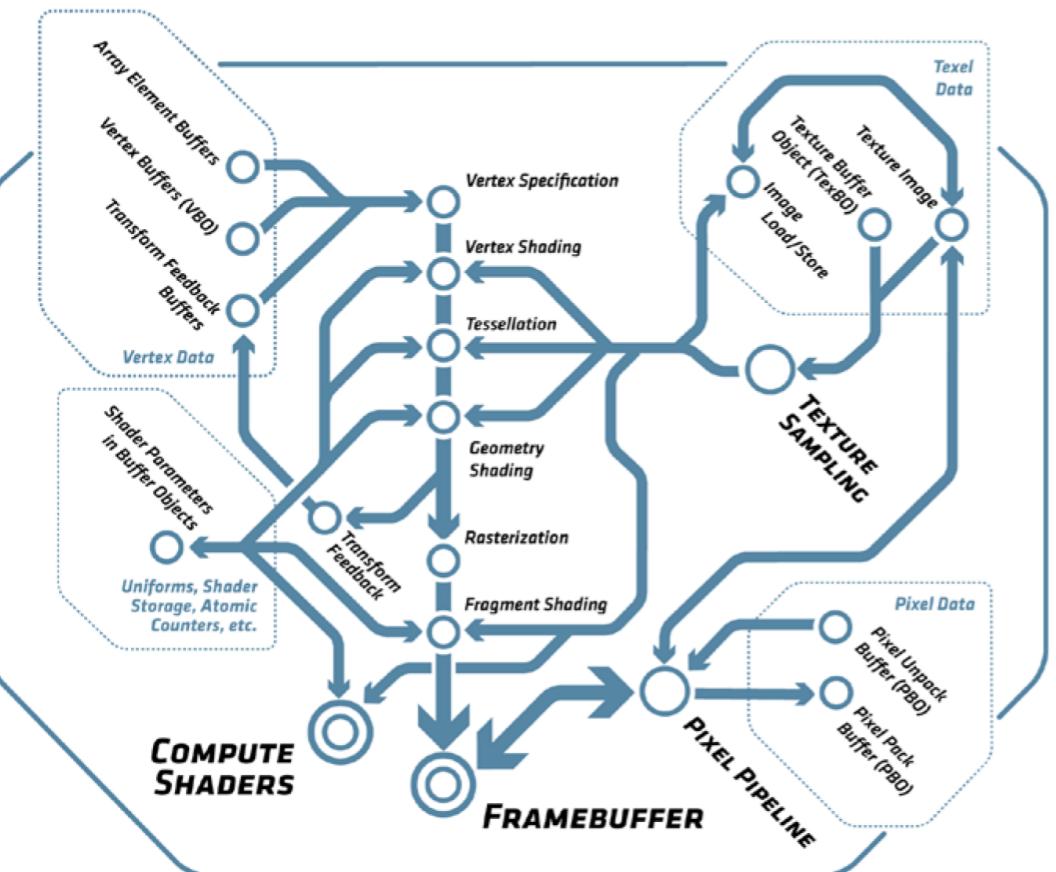
- A cross-platform graphics and compute API for modern GPUs
 - Logical successor to OpenGL / ES
 - Focus on real-time applications
 - One API across all markets and platforms
- Key features
 - Programming model matches the hardware
 - Multi-thread and multi-core friendly
 - Error checking and compilation are outside the driver
 - Much lower CPU overhead
- Our bargain with developers
 - You take more responsibility and do more work
 - You get better and more predictable performance



Pre-Vulkan - History of OpenGL



SIGGRAPH ASIA 2008
NEW HORIZONS



Bringing state-of-the-art
functionality to cross-
platform graphics

OpenGL 4.5

OpenGL 4.4

OpenGL 4.3

OpenGL 4.2

OpenGL 4.1

OpenGL 3.3/4.0

OpenGL 3.2

OpenGL 3.1

OpenGL 2.0

OpenGL 2.1

OpenGL 3.0

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

DirectX
9.0c

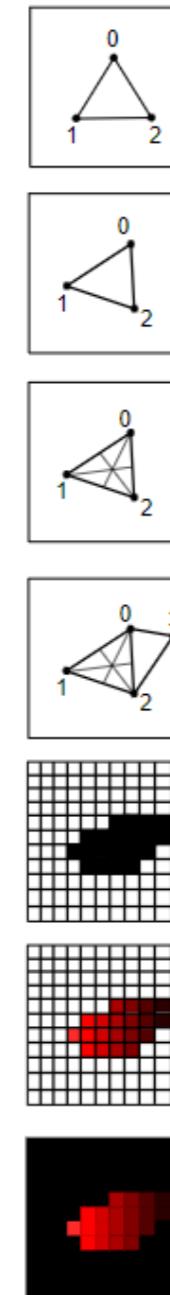
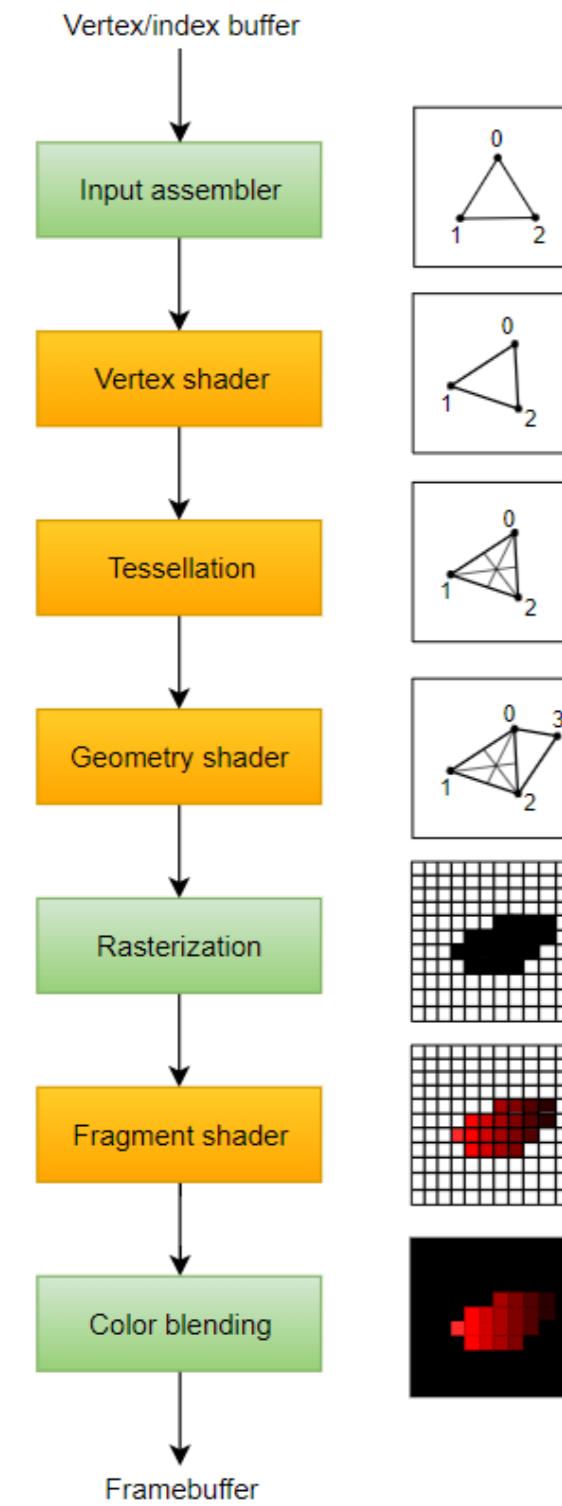
DirectX
10.0

DirectX
10.1

DirectX
11

DirectX
11.1

DirectX
11.2

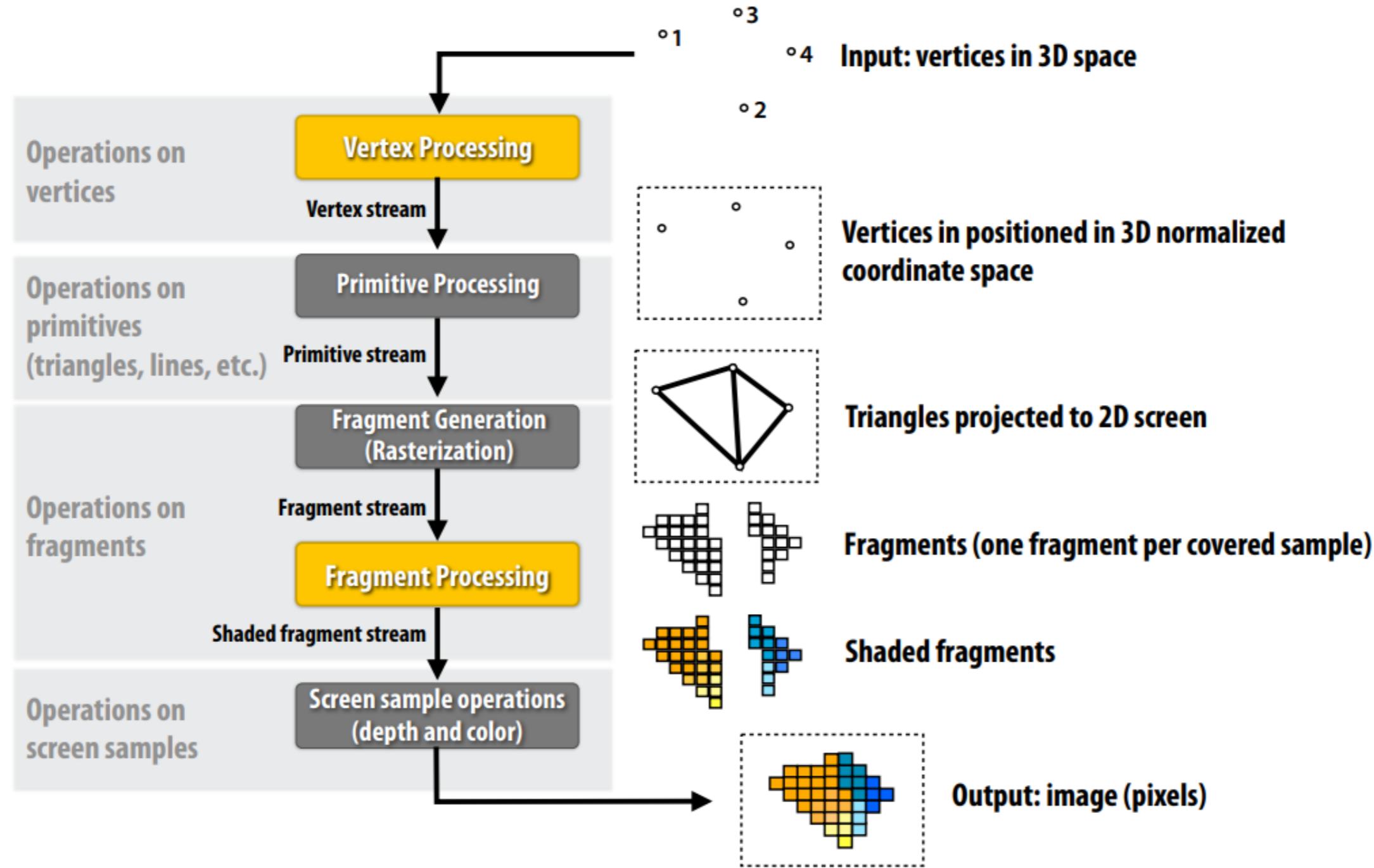


Graphics Pipeline

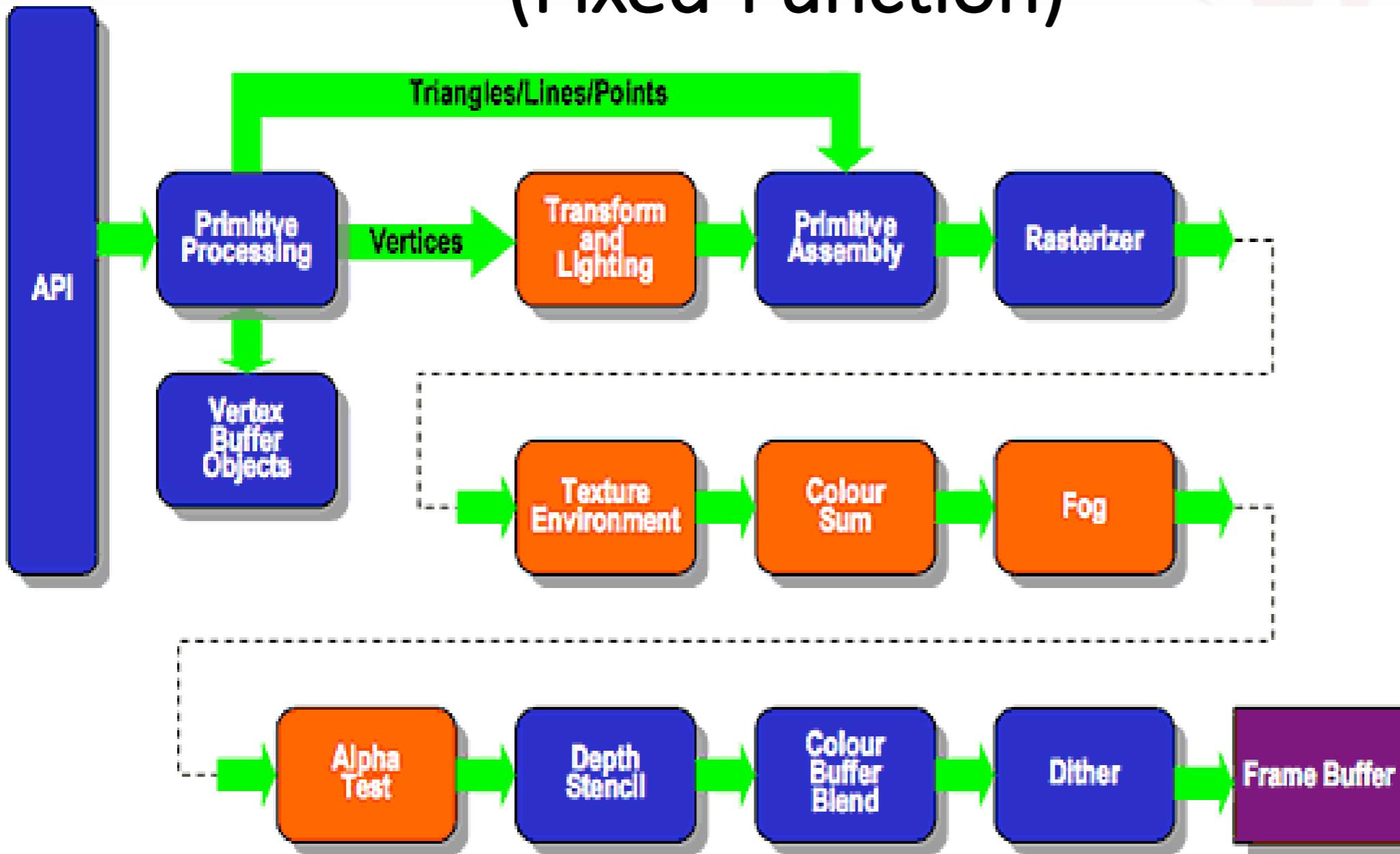
Sequence of operations that take the vertices and textures of your dataset all the way to the pixels in the render target(s)

We Work In-Depth

Note: Refer to graphics primers for explanations of these operations



OpenGL Pipeline (Fixed-Function)



Simpler Times

```
#include <opengl.h>

main()
{
    InitializeOpenGLWindow();
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);

    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd();
}
```

- Example 1.1 (OpenGL Red Book)

Vulkan®

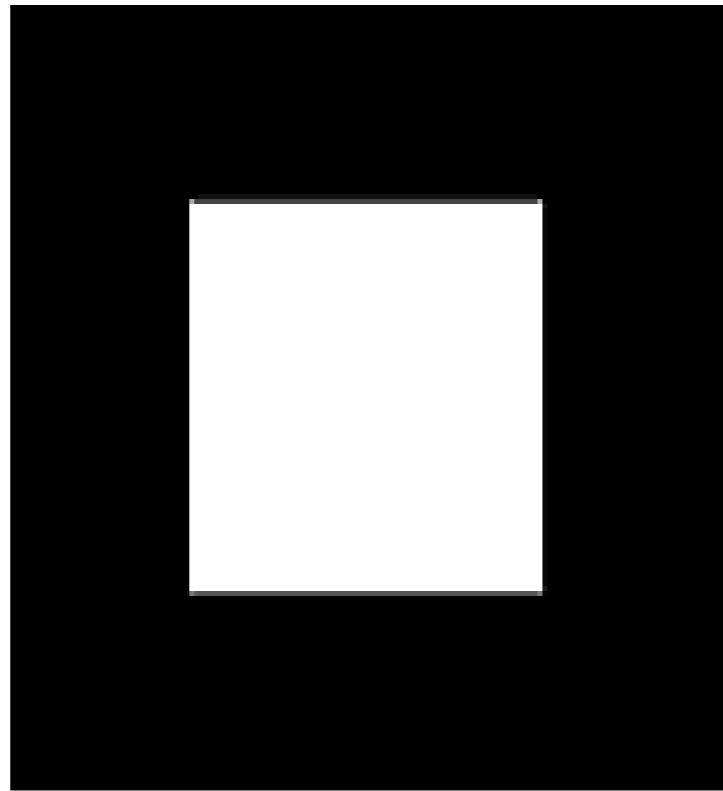
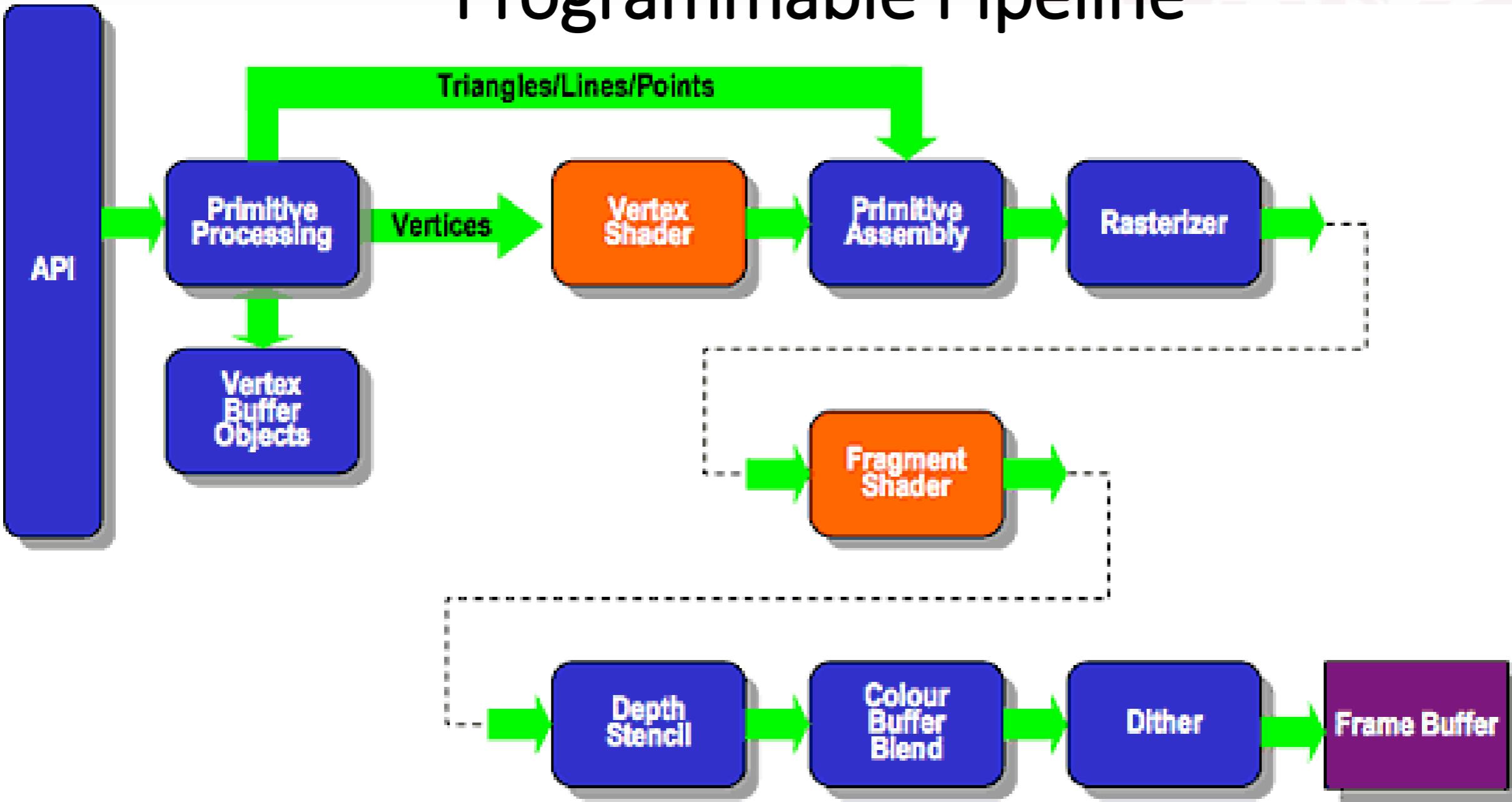


Figure 1-1 : White Rectangle on a Black Background

Programmable Pipeline



Shaders

Vertex Shader:

```
uniform mat4 mvpMatrix;  
in vec4 vertex;  
in vec4 color;  
out vec4 varyingColor;  
  
void main(void)  
{  
    varyingColor = color;  
    gl_Position = mvpMatrix * vertex;  
}
```

Fragment Shader:

```
in vec4 varyingColor;  
out vec4 fragColor;  
  
void main(void)  
{  
    fragColor = varyingColor;  
}
```

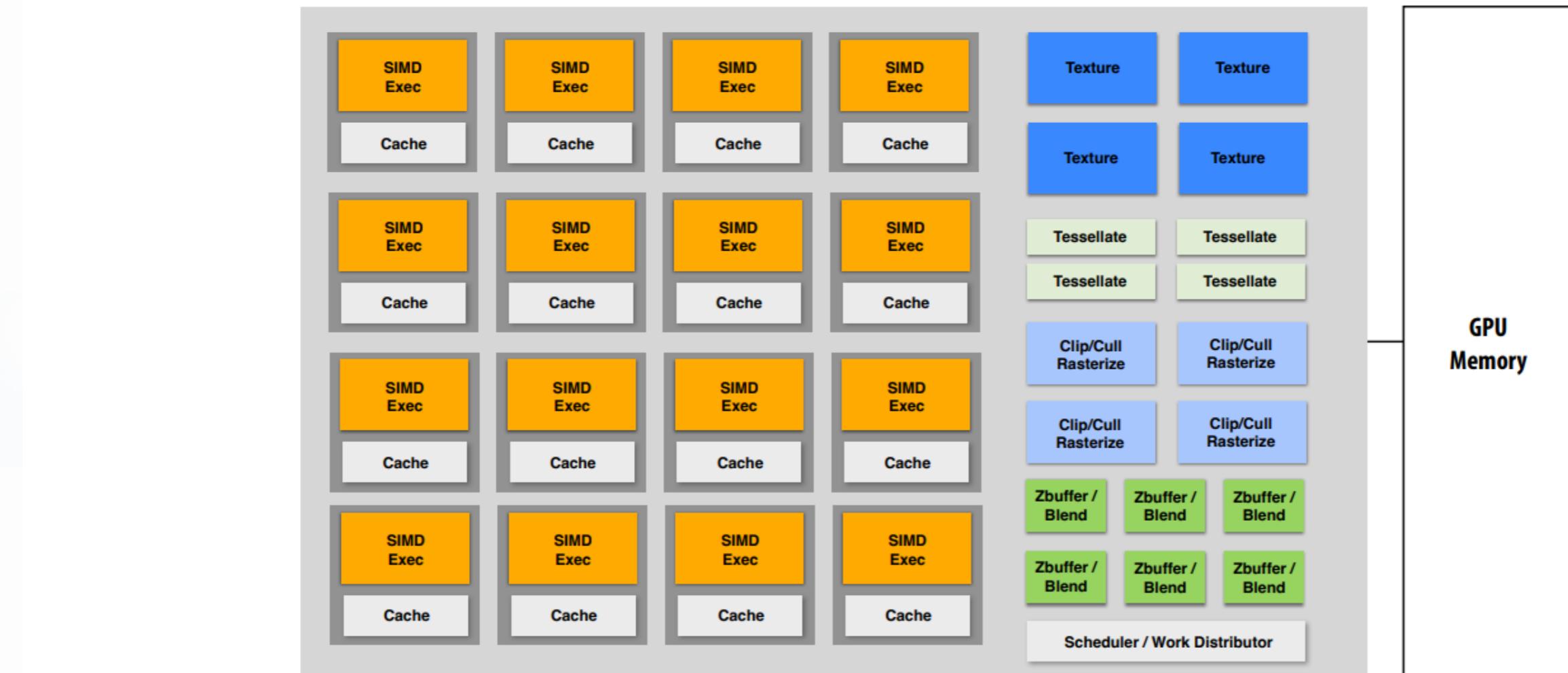
Fixed vs. Programmable

- Modern GPU architecture has rendered Fixed-Function pipeline obsolete
- Fixed-Function
 - separate hardware units for T&L, Fog, Color/Blend/Alpha, Texture etc.
 - dedicated API calls for each function
 - less scalable, inflexible - functionality enhanced only by adding new API calls, parameters, or GL states
- Modern GPUS
 - many general compute cores, massively parallel
 - graphics operations programmable using shaders
 - some functionality needs to be implemented by the programmer now
 - e.g. matrix transform, lighting & shading computations

GPU: heterogeneous, multi-core processor

Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")

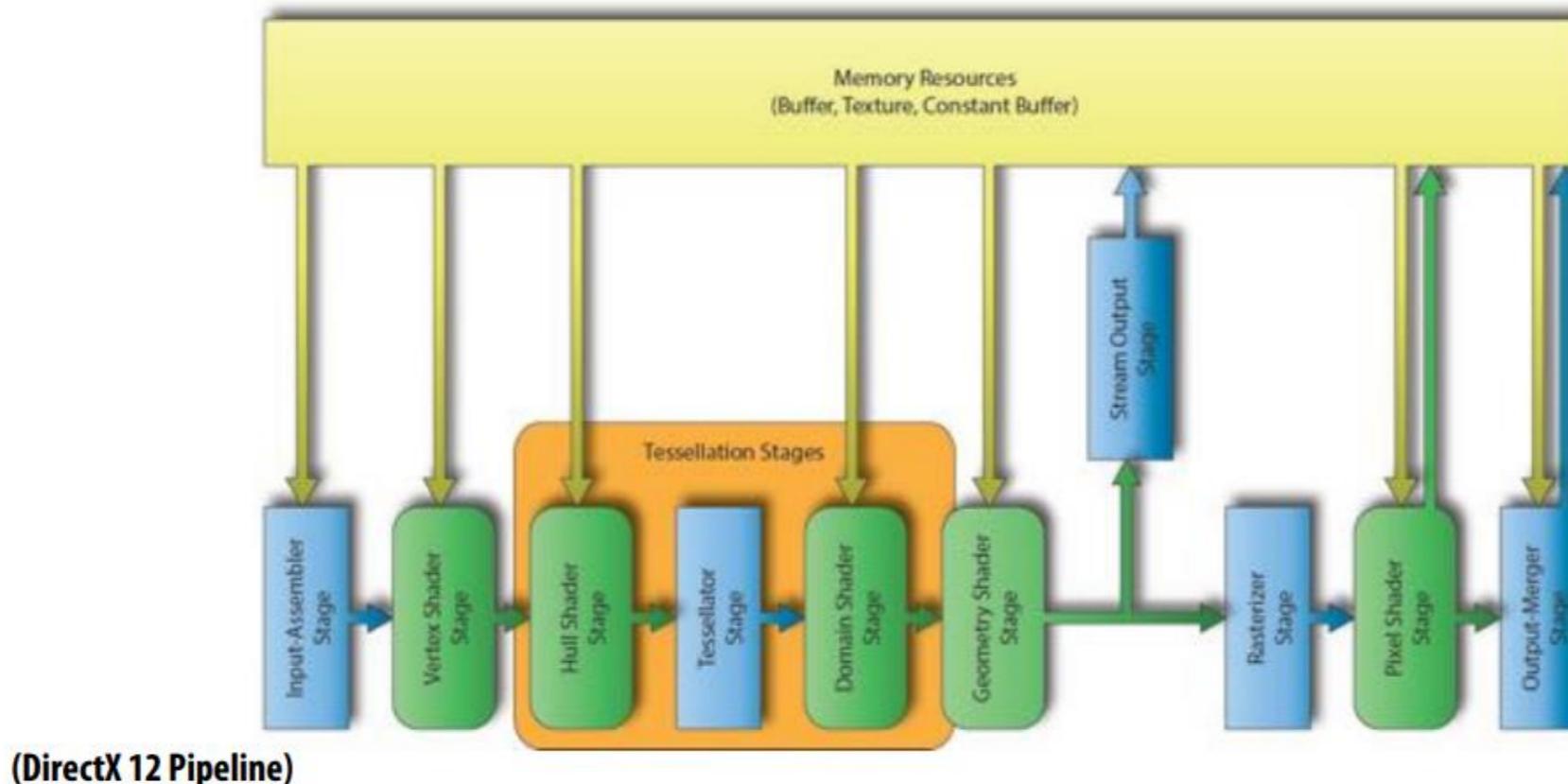
still enormous amount of fixed-function compute over here



This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!

Modern Rasterization Pipeline

- Trend toward more generic (but still highly parallel!) computation:
 - make stages programmable
 - replace fixed function vertex, fragment processing
 - add geometry, tessellation shaders
 - generic “compute” shaders (whole other story...)
 - more flexible scheduling of stages





**Originally architected for graphics workstations
with direct renderers and split memory**

**Matches architecture of modern platforms
including mobile platforms with unified memory, tiled rendering**

**Driver does lots of work: state validation, dependency tracking,
error checking. Limits and randomizes performance**

**Explicit API – the application has direct, predictable control
over the operation of the GPU**

**Threading model doesn't enable generation of graphics
commands in parallel to command execution**

**Multi-core friendly with multiple command buffers
that can be created in parallel**

**Syntax evolved over twenty years – complex API choices can
obscure optimal performance path**

**Removing legacy requirements simplifies API design,
reduces specification size and enables clear usage guidance**

**Shader language compiler built into driver.
Only GLSL supported. Have to ship shader source**

**SPIR-V as compiler target simplifies driver and enables front-end
language flexibility and reliability**

**Despite conformance testing developers must often handle
implementation variability between vendors**

**Simpler API, common language front-ends, more rigorous
testing increase cross vendor functional/performance portability**

Vulkan Explicit GPU Control

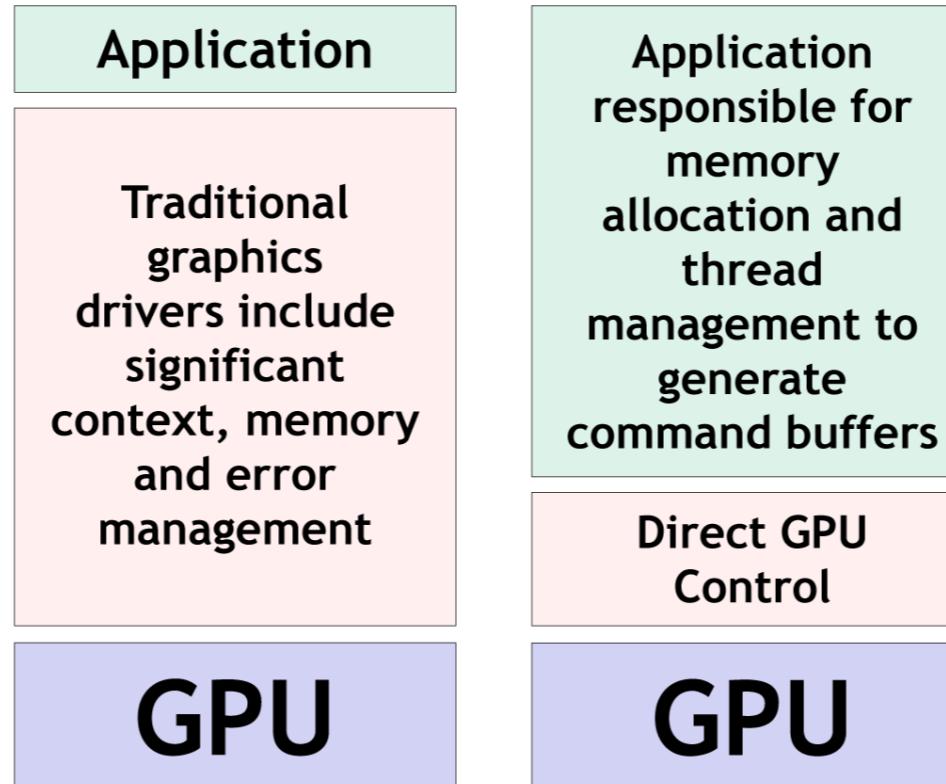


Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets



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 delivers the maximized performance and cross platform portability needed by sophisticated engines, middleware and apps

10 Lines
of
code

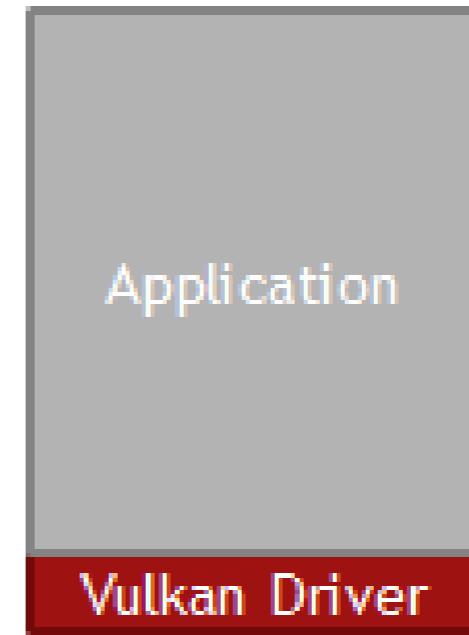
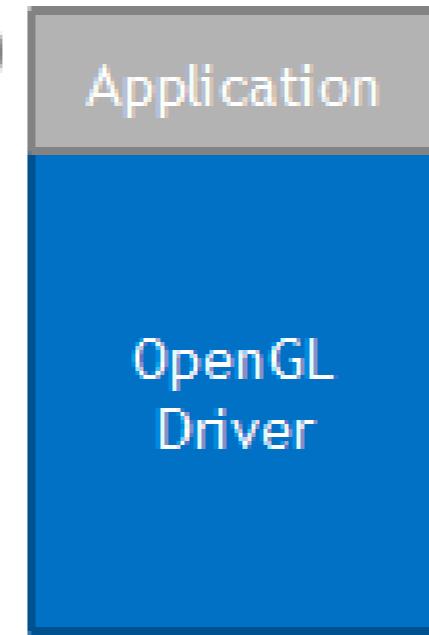
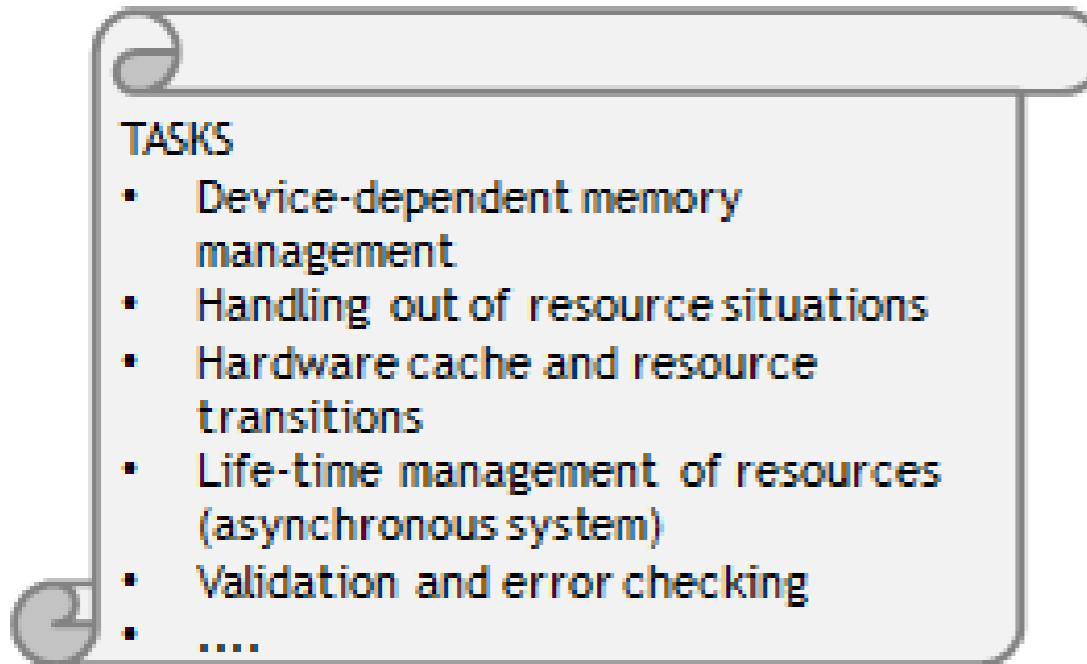


Code Complexity:

Vulkan is a much more verbose API. It can be faster due to additional information supplied by the application, however with more control comes more responsibility to do it right.

<code>glCreateTextures</code>	}	Query memory type/size for format and use.
<code>glTextureStorage</code>		Manage memory allocation...
<code>glCreateFramebuffers</code>	}	Setup renderpass, dependencies, attachment load/store behavior...
<code>glNamedFramebufferTexture</code>		
<code>glCreateProgram</code>	}	Generate SPIR-V, create pipeline object requiring full state definition, renderpass...
<code>glCreateShader (x 2)</code>		
<code>glShaderSource</code>	}	
<code>glCompileShader</code>		
<code>glAttachShader</code>	}	
<code>glBindFramebuffer</code>		
<code>glViewport</code>	}	
<code>glUseProgram</code>		
<code>glDrawArrays(GL_TRIANGLES, 0, 3)</code>	}	Create commandbuffer and its memory backing. record commands and submit to queue





Application Responsibility:

The OpenGL driver manages a lot of tasks for the application, that move to the application's responsibility in Vulkan. This can be positive or negative depending on the level of engagement for such tasks.

VULKAN

THE BEGINNINGS

A simplified view of a typical **OpenGL** or <DX 10 app rendering pipeline:

► **CPU thread:**



► **GPU usage:**



Why?

VULKAN

THE BEGINNINGS

A simplified view of a typical OpenGL or <DX 10 app rendering pipeline:



Why?

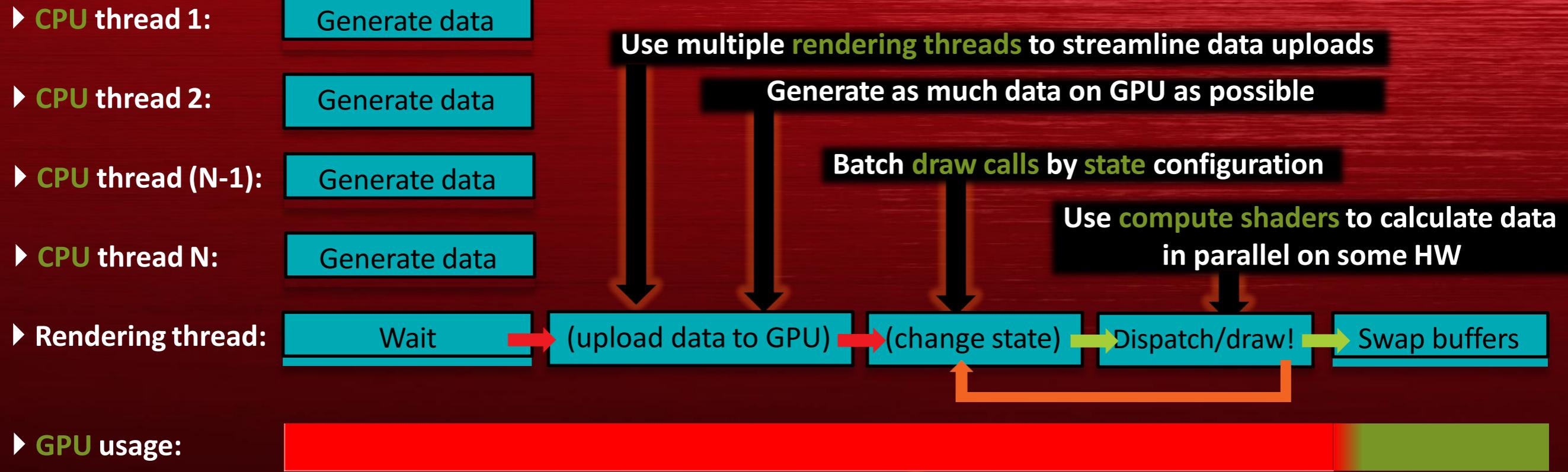
- CPU->GPU command submission is **time-consuming**:
 - Only **submit** and start executing GPU-side if:
 - All **commands** for a **frame** have been submitted..
 - Command buffer **fills up**
 - Upon app's **explicit request**.
- App can submit **different commands every frame**:
 - Cannot bake **command buffers in advance**!

Typical OpenGL or <DX 10 app rendering pipeline (more advanced apps):



Can we do any better?

Typical OpenGL or <DX 10 app rendering pipeline (more advanced apps):



VULKAN

THE BEGINNINGS

- These workarounds **do not solve the biggest problem:**
 - GPUs are **highly asynchronous** constructs:
 - Designed to perform many kinds of tasks **in parallel**:
 - Computations
 - DMA transfers
 - Rasterization
 - Other (eg. accelerated image data conversion)
- But from API standpoint:
 - GPU can only be requested to execute **work chunks** from **one rendering thread!**
 - Apps **cannot be trusted** – CPU time spent on API **call validation..**

VULKAN

THE BEGINNINGS

■ Do we really care?

- More and more **CPU-bound apps** are showing up on the market.
 - Driver thread(s) consuming CPU time
 - Increasing app complexity
- No **easy way** to address these in a **cross-platform way**.
- **Tilers** cannot leverage their full power on **OpenGL ES**.
- Only **vendor-specific** solutions exist (eg. **Pixel Local Storage**)

■ Not to mention use cases like:

- Multiple **GPU** support
- VR

VULKAN

DO I NEED IT?

- **Vulkan** addresses all of the discussed issues:
 - Exposes **GPU** as a set of **command queue families**.
 - **Command buffers** can be submitted to **queues** from **multiple threads**.
 - **Application is responsible for**:
 - submitting **work chunks** to the right **command queues**.
 - **synchronization** of **GPU jobs' execution**.
 - Exposes available **GPU memory** as a set of **memory heaps**.
 - Application is responsible for **flushes / invalidation / management**.
- **Applications are required** to adapt to the **running GPU's capabilities**.
- Misbehave and **hang the GPU**.

VULKAN

DO I NEED IT?

■ Who NEEDS Vulkan?

– CPU-bound applications:

- Vast majority of information required to compute / render – prebaked at loading time.
- A frame can be rendered with just **two commands!**
- **No driver-side validation** = more **CPU time** for stuff that really matters.

– GPU-bound applications:

– Improve **GPU utilization** by:

- Submitting compute / graphics jobs to relevant **queue families**.
- Performing VRAM -> VRAM & RAM <-> VRAM copy ops with **transfer queues**.

– No sudden performance drops or spikes:

- All **GPU-side caches** are **flushed**, according to app-specified information, at predictable times.
- Driver **no longer** needs to do any **guess-work**.

VULKAN

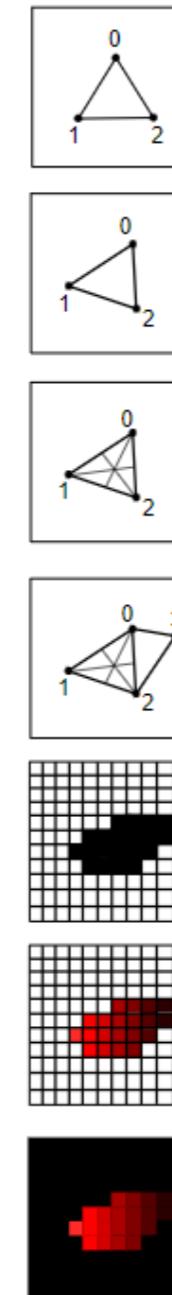
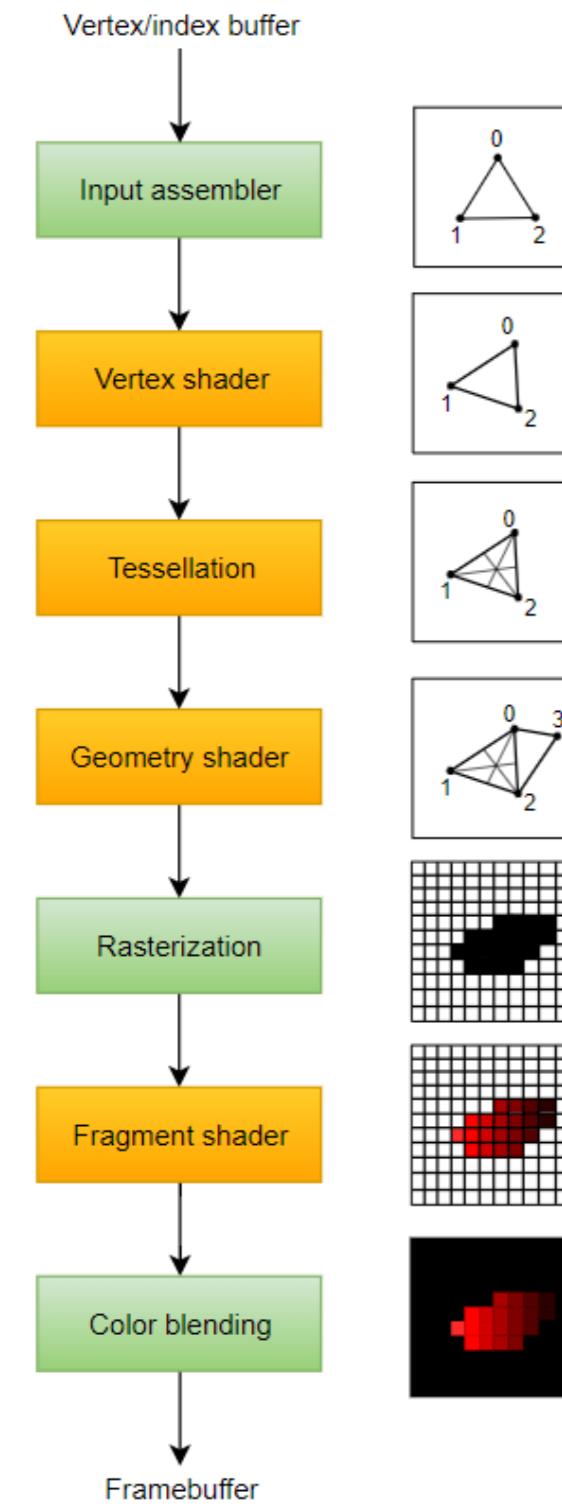
DO I NEED IT?

■ Who MAY need Vulkan?

- Existing **GL 4.x / <= DX 11** applications:
 - Moving to Vulkan **may or may not** bring performance benefits.
 - Likely to spend less CPU power.

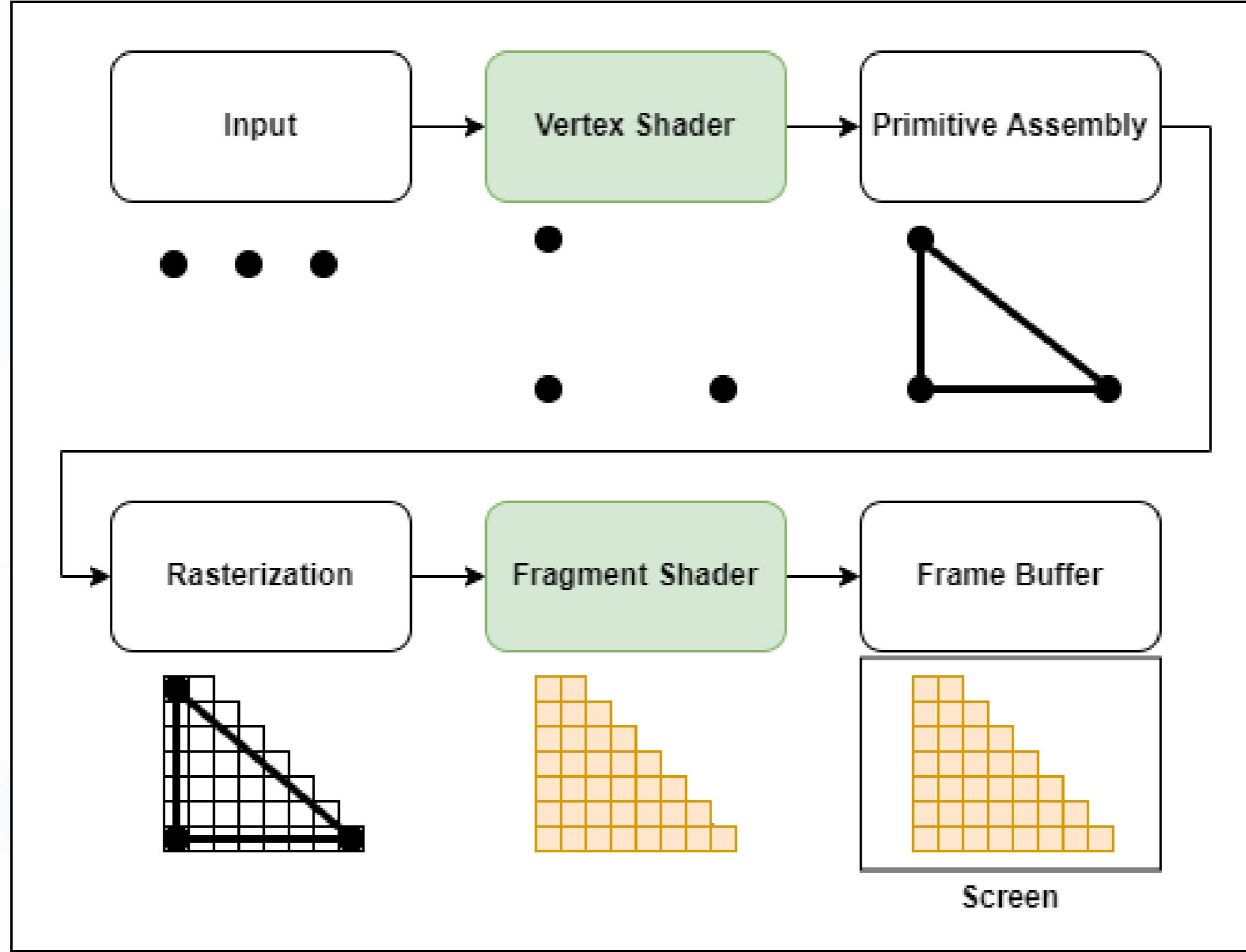
■ Who does NOT need Vulkan?

- Prototype applications requiring **rapid development time**:
 - Validation layers do **not** cover whole specification **yet**.
 - Many **incorrect use cases** are still **not** detected.
 - **Steep** learning curve.
- **Simple applications** which are **not CPU- or GPU-bound**:
 - Unless for **learning purposes**, these are unlikely to benefit from Vulkan.

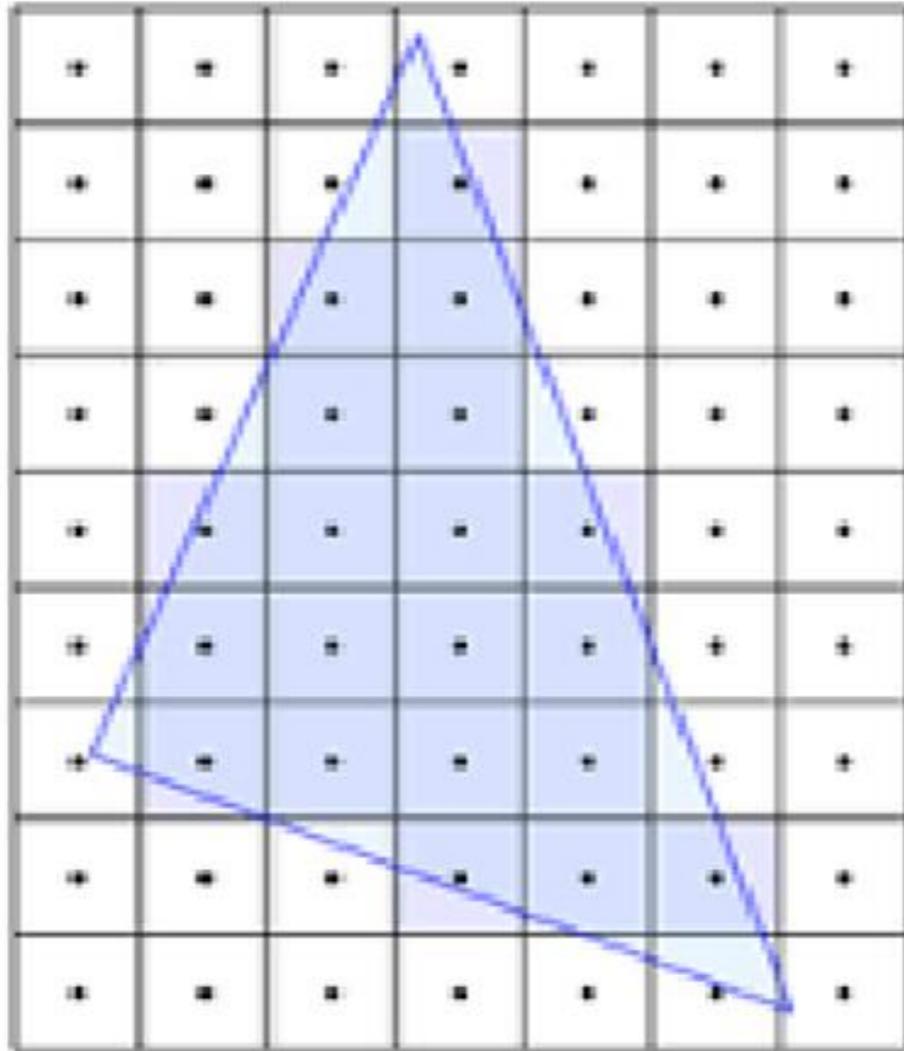


Hardware-accelerated Graphics Pipeline

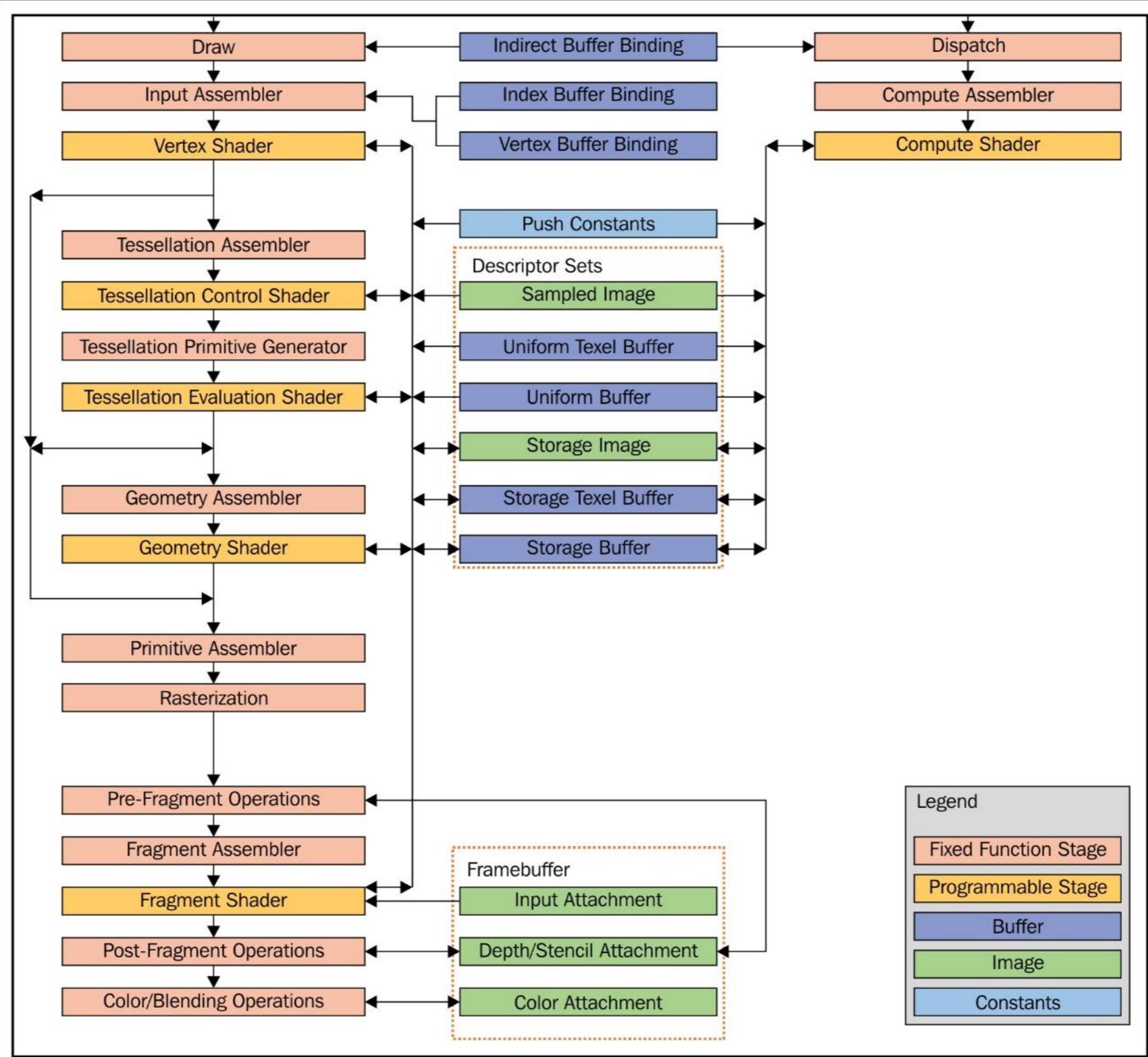
Sequence of operations that take the vertices and textures of your dataset all the way to the pixels in the render target(s)

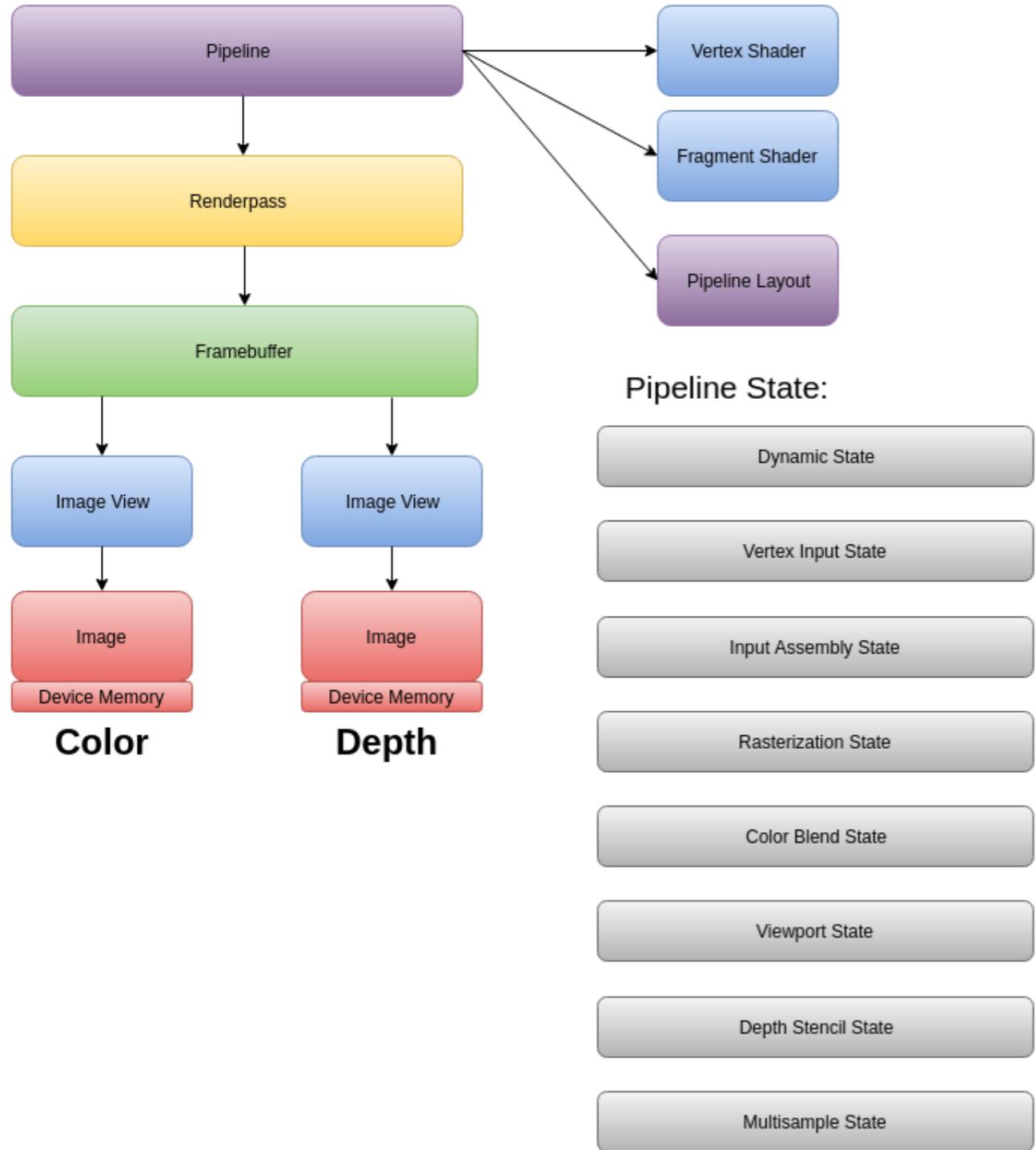


Rasterization



- Frame buffer is subdivided into a grid of pixels
- Determine which pixels are covered by the primitive
- Fragment – data relevant to a pixel, finally resolves to a colour/output value (RGB, luminance etc.)





Vulkan Graphics Pipeline

Pipeline State:

Dynamic State

Vertex Input State

Input Assembly State

Rasterization State

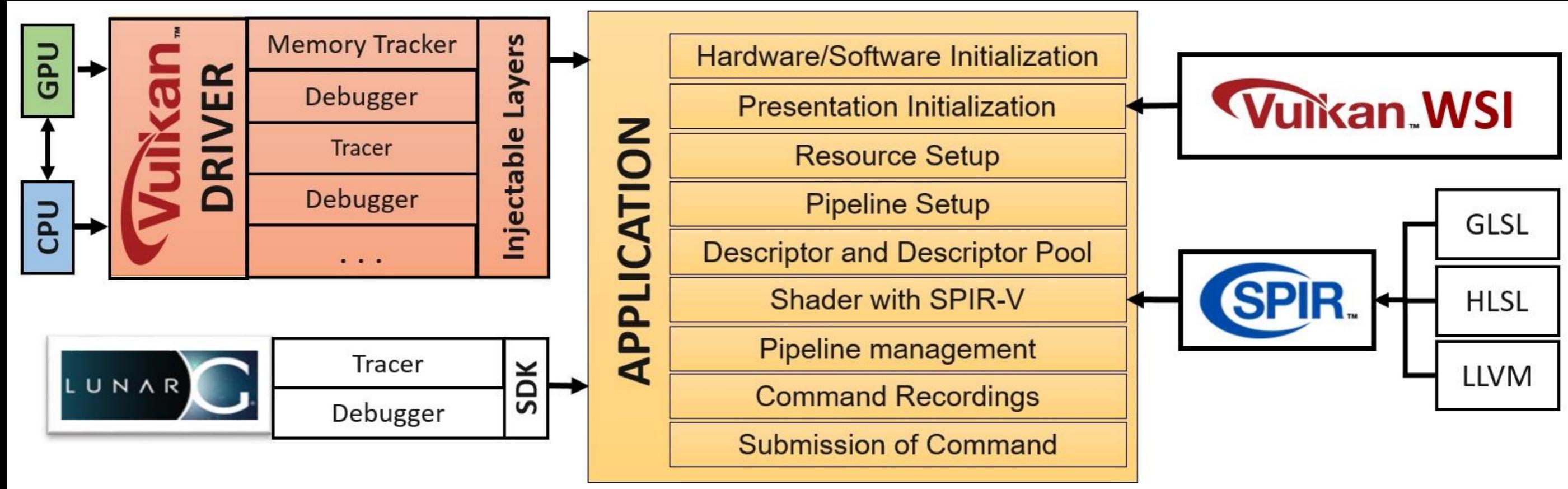
Color Blend State

Viewport State

Depth Stencil State

Multisample State

The Vulkan Application



Vulkan Sample – Hello Triangle

- Development Environment
 - VS2017/C++17
 - Vulkan 1.3.250.1
 - GLFW 3.0.4
 - GLM
 - STB (for texture load)
- Source: <https://vulkan-tutorial.com>

GLFW

As mentioned before, Vulkan by itself is a platform agnostic API and does not include tools for creating a window to display the rendered results. To benefit from the cross-platform advantages of Vulkan and to avoid the horrors of Win32, we'll use the [GLFW library](#) to create a window, which supports Windows, Linux and MacOS. There are other libraries available for this purpose, like [SDL](#), but the advantage of GLFW is that it also abstracts away some of the other platform-specific things in Vulkan besides just window creation.

See:

<https://www.glfw.org/docs/3.3/index.html>



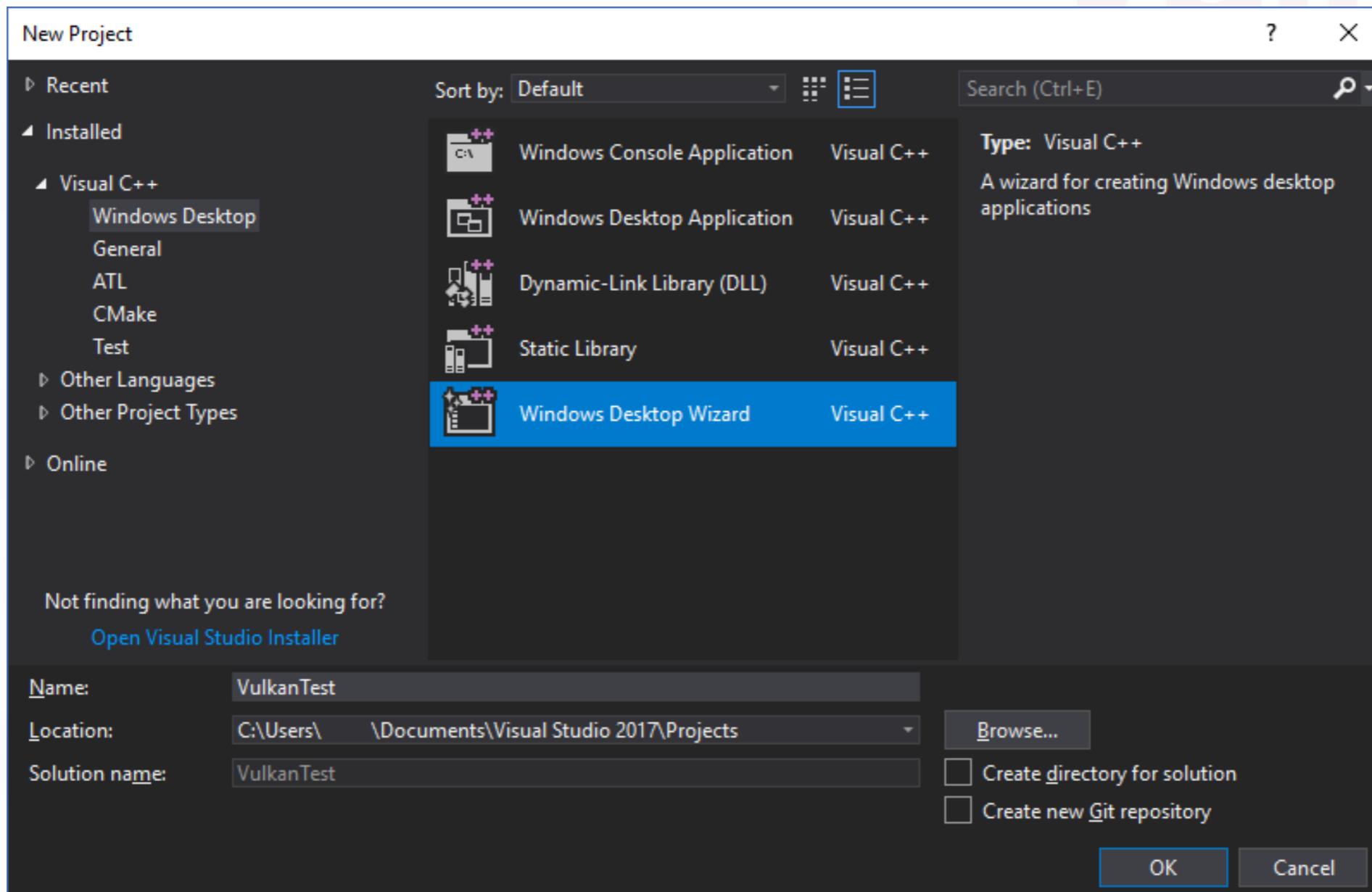
Gives you a window and OpenGL context with just **two function calls**

Support for OpenGL, OpenGL ES, Vulkan and related options, flags and extensions

Support for multiple windows, multiple monitors, high-DPI and gamma ramps

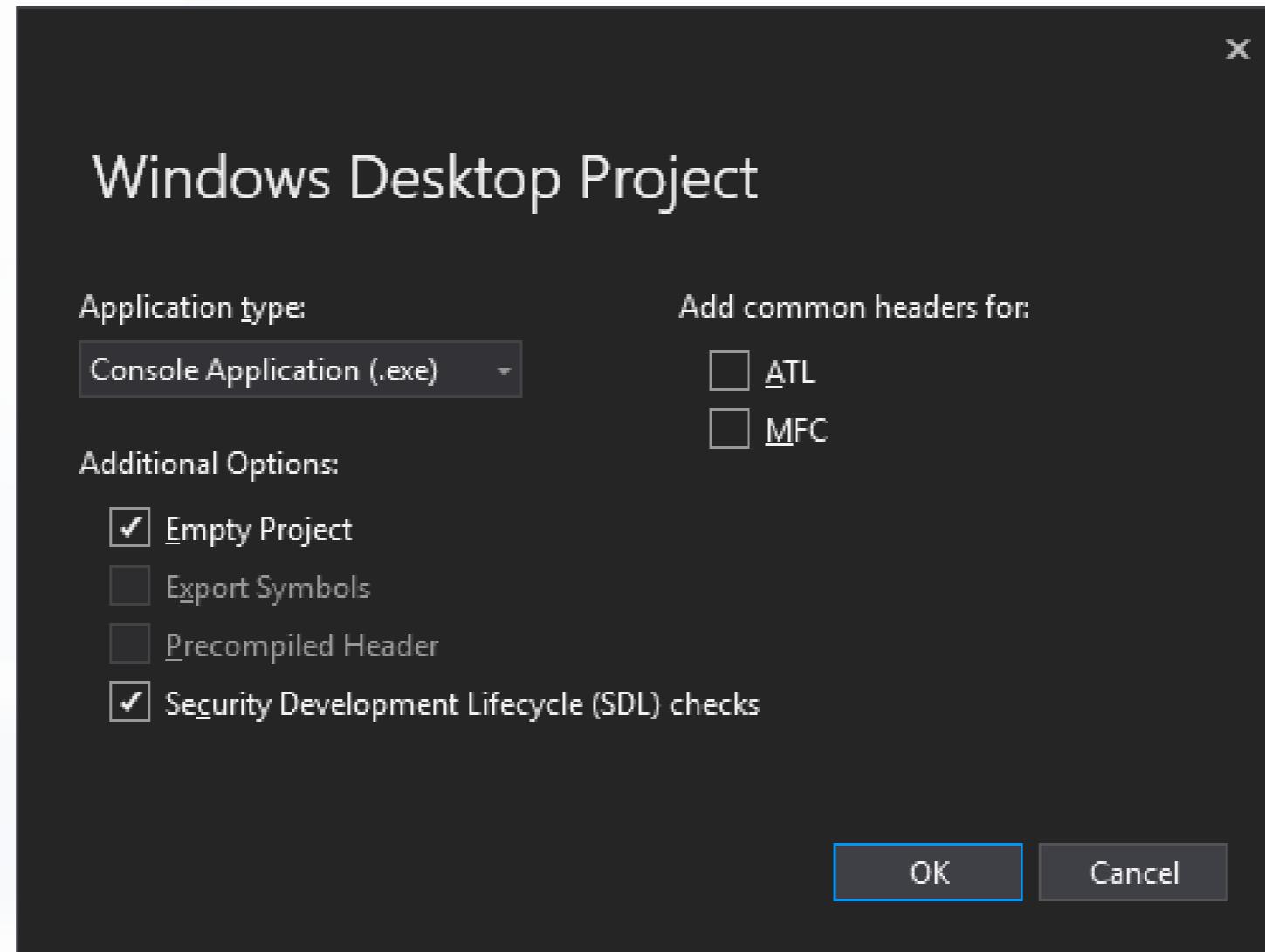
Support for keyboard, mouse, gamepad, time and window event input, via polling or callbacks

Visual Studio Setup

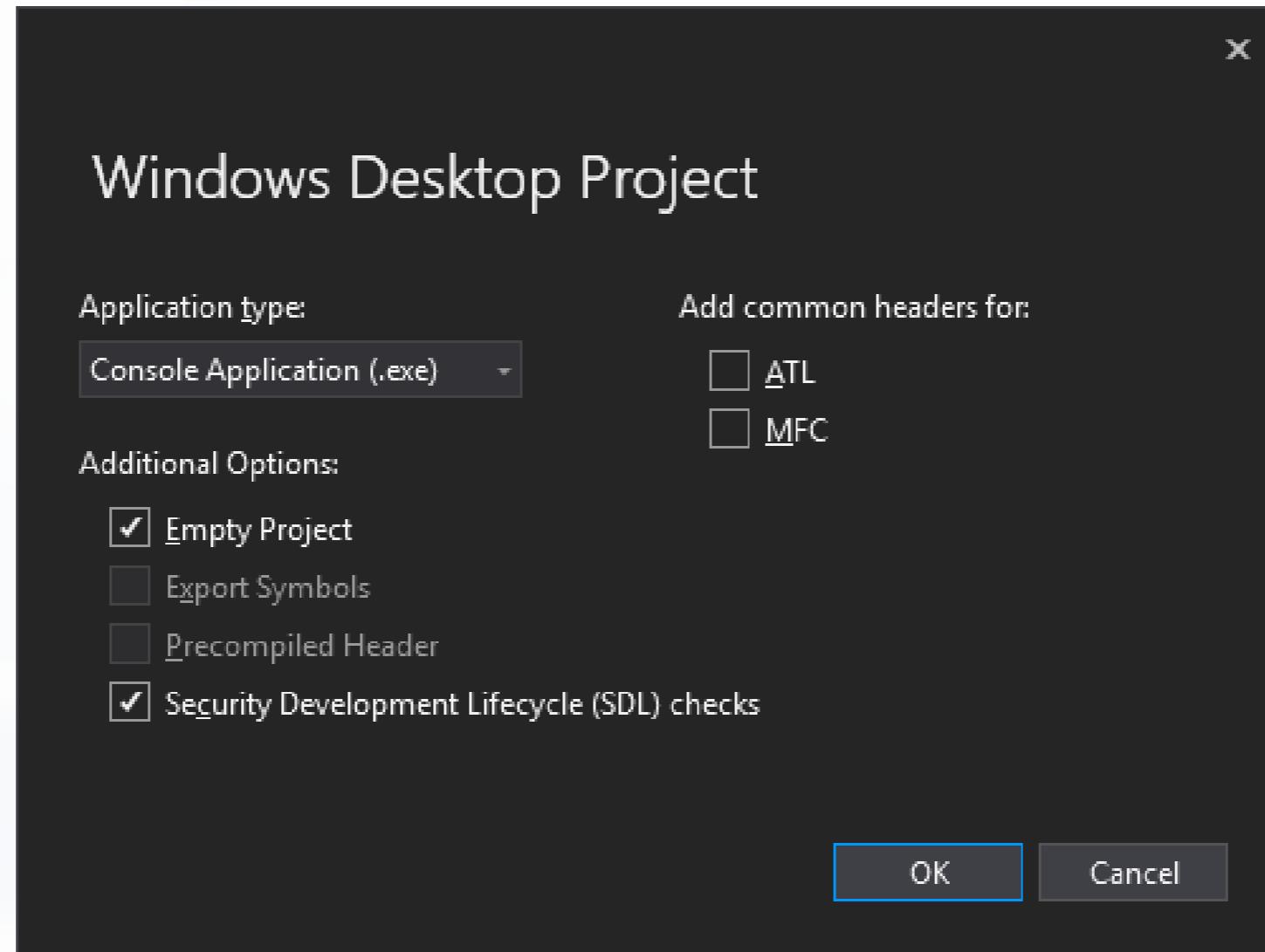


Visual Studio Setup

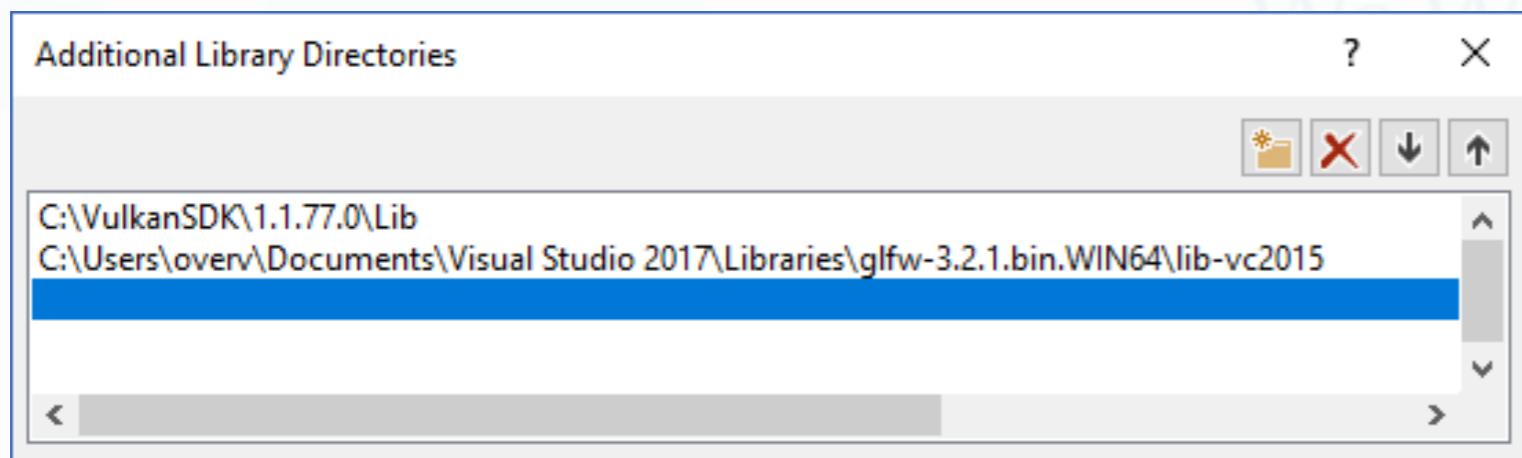
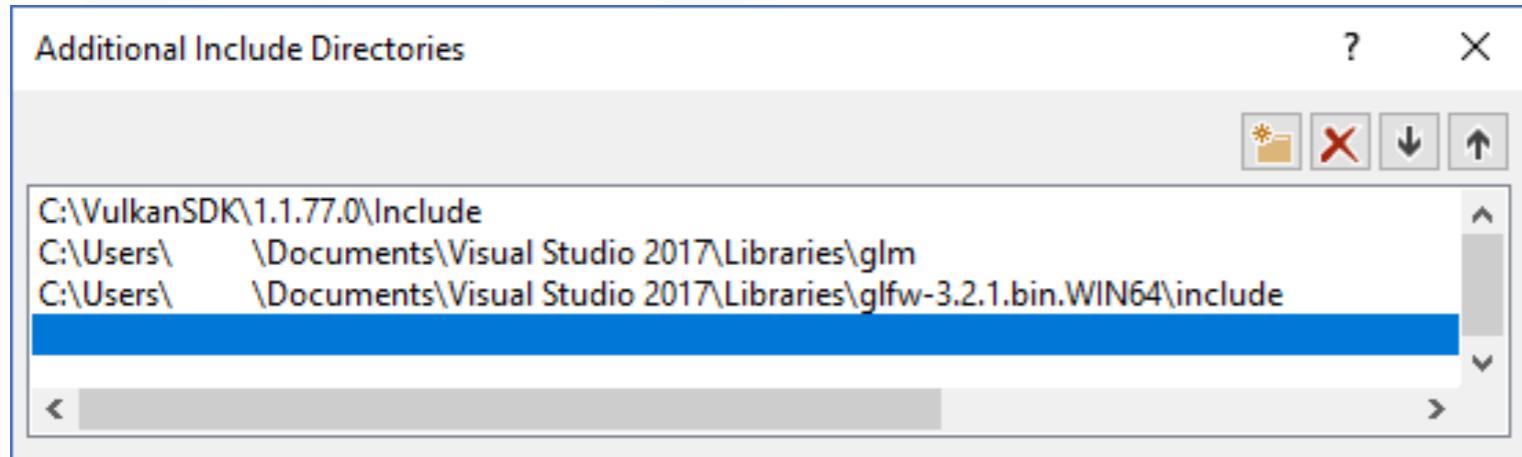
Vulkan®



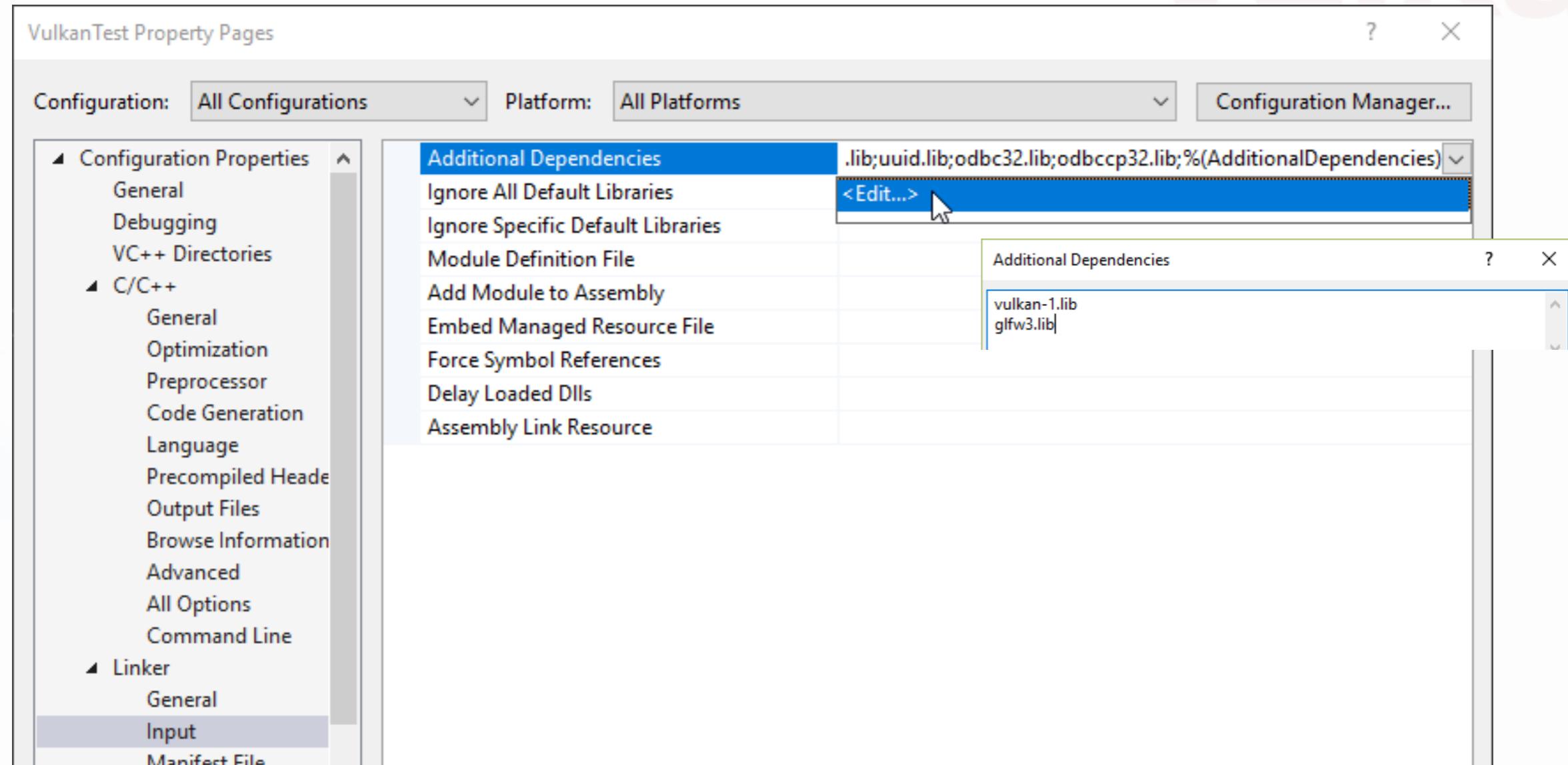
Visual Studio Setup



Visual Studio Directories



Visual Studio Linker

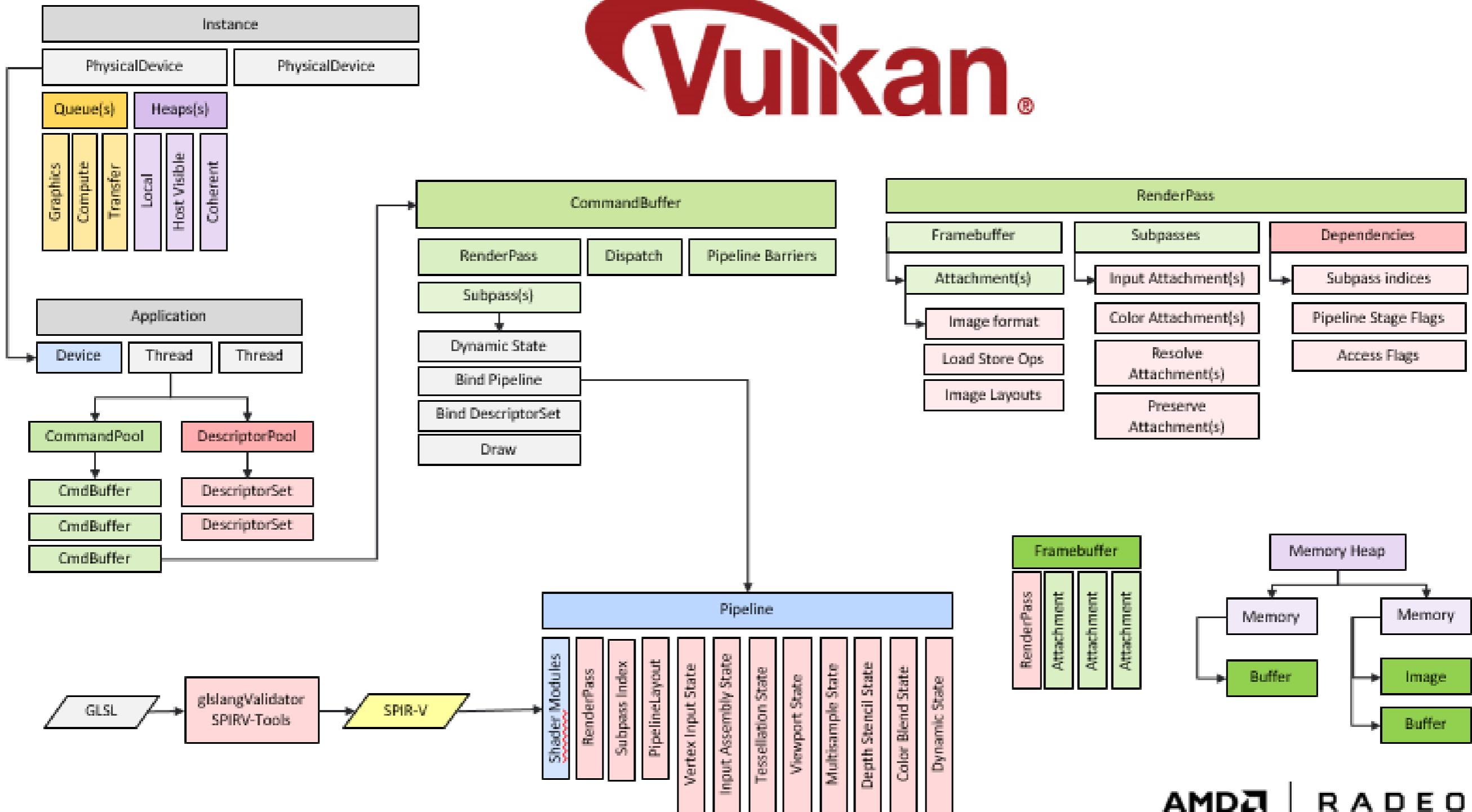


C++17

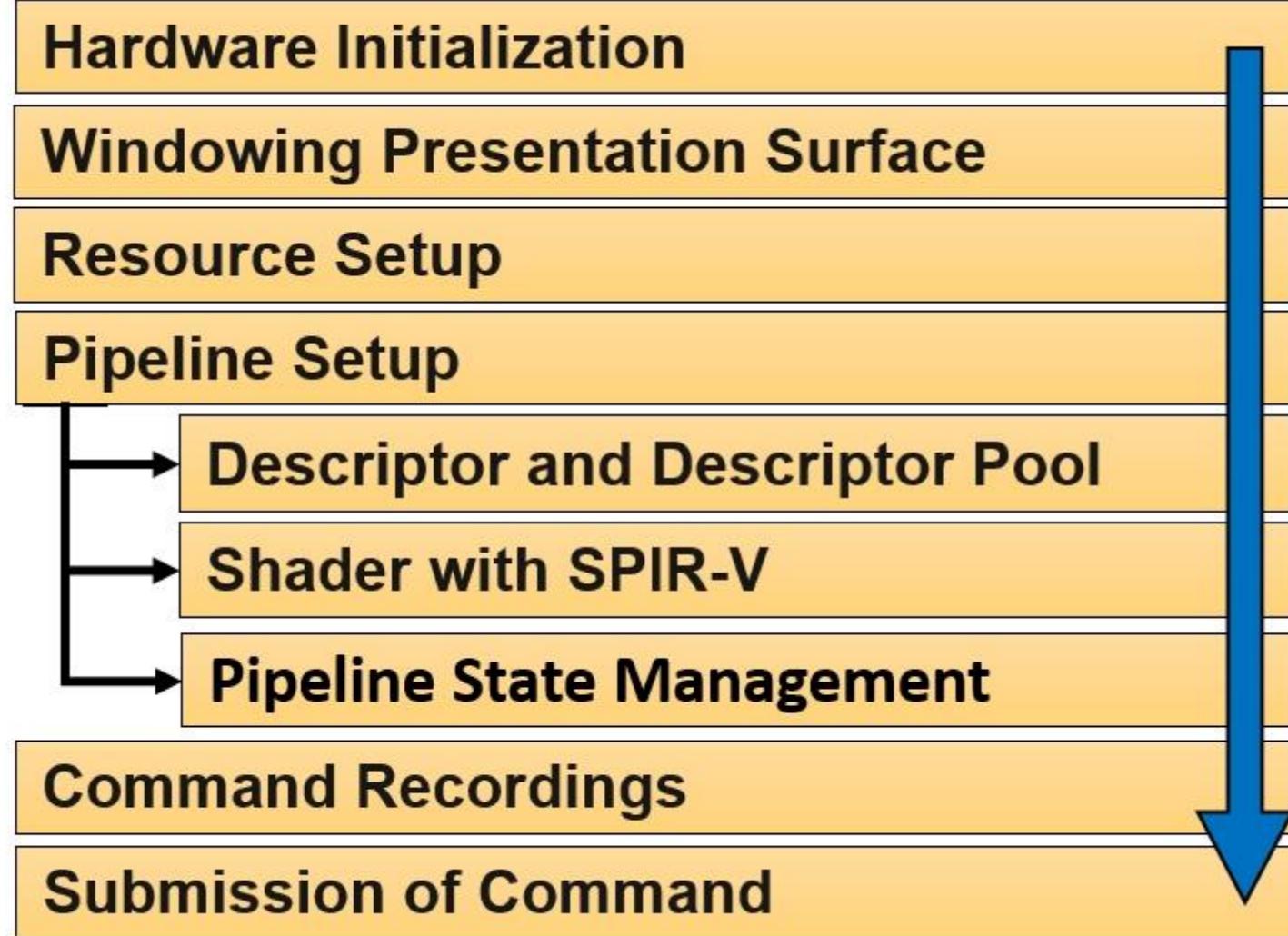
The screenshot shows the Visual Studio C/C++ Properties dialog. On the left, a tree view under the 'C/C++' node has 'Language' selected, indicated by a blue background. The main area displays properties for the 'Language' category. A dropdown menu for 'C++ Language Standard' is open, showing three options: 'ISO C++14 Standard (/std:c++14)', 'ISO C++17 Standard (/std:c++17)' (which is highlighted with a blue selection bar and has a cursor arrow pointing to it), and 'ISO C++ Latest Draft Standard (/std:c++latest)'. Other visible properties include 'Remove unreferenced code and data' set to 'Yes (/Zc:inline)', 'Enforce type conversion rules', 'Enable Run-Time Type Information', and 'Open MP Support'.

Property	Value
Remove unreferenced code and data	Yes (/Zc:inline)
Enforce type conversion rules	
Enable Run-Time Type Information	
Open MP Support	
C++ Language Standard	
Enable C++ Modules (experimental)	ISO C++14 Standard (/std:c++14) ISO C++17 Standard (/std:c++17) ISO C++ Latest Draft Standard (/std:c++latest)

Vulkan.

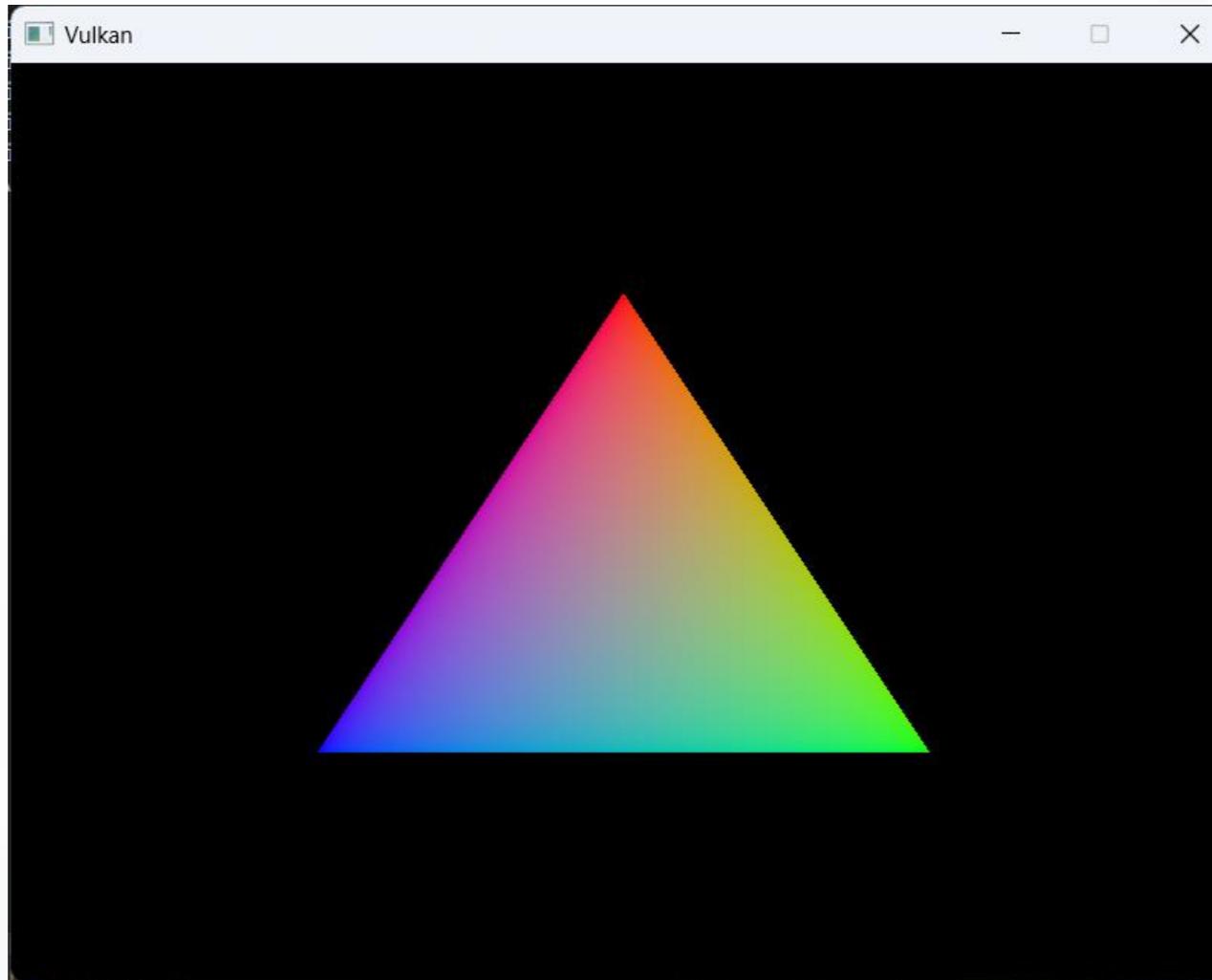


Vulkan Application Programming Model



Hello Triangle

Vulkan®



https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Base_code

General Structure

VulkanTutorial\00_base_code

```
#include <vulkan/vulkan.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

class HelloTriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    void initVulkan() {

    }

    void mainLoop() {

    }

    void cleanup() {

    };
};
```

Creating an Instance (VulkanTutorial\15_hello_triangle)

Start by adding a `createInstance` function and invoking it in the `initVulkan` function.

```
void initVulkan() {
    createInstance();
}
```

Additionally add a data member to hold the handle to the instance:

```
private:
VkInstance instance;
```

Now, to create an instance we'll first have to fill in a struct with some information about our application. This data is technically optional, but it may provide some useful information to the driver in order to optimize our specific application (e.g. because it uses a well-known graphics engine with certain special behavior). This struct is called `VkApplicationInfo`:

```
void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;
}
```

Creating an Instance

```
VkInstanceCreateInfo createInfo{};  
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;  
createInfo.pApplicationInfo = &appInfo;
```

The first two parameters are straightforward. The next two layers specify the desired global extensions. As mentioned in the overview chapter, Vulkan is a platform agnostic API, which means that you need an extension to interface with the window system. GLFW has a handy built-in function that returns the extension(s) it needs to do that which we can pass to the struct:

```
uint32_t glfwExtensionCount = 0;  
const char** glfwExtensions;  
  
glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);  
  
createInfo.enabledExtensionCount = glfwExtensionCount;  
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

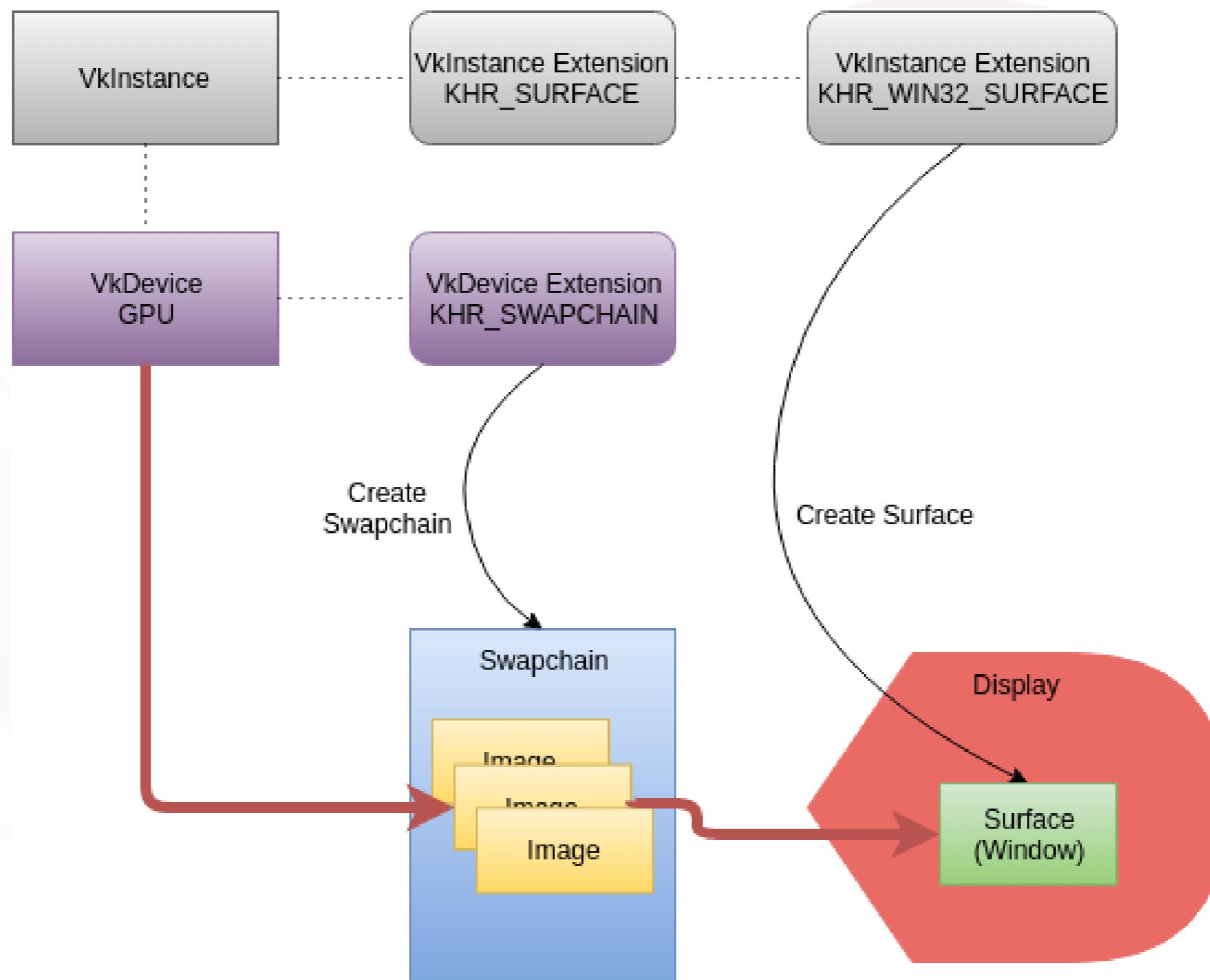
The last two members of the struct determine the global validation layers to enable. We'll talk about these more in-depth in the next chapter, so just leave these empty for now.

```
createInfo.enabledLayerCount = 0;
```

We've now specified everything Vulkan needs to create an instance and we can finally issue the `vkCreateInstance` call:

```
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

kan.



Physical devices and queue families

an®

- [Selecting a physical device](#)
- [Base device suitability checks](#)
- [Queue families](#)

Selecting a physical device

After initializing the Vulkan library through a `VkInstance` we need to look for and select a graphics card in the system that supports the features we need. In fact we can select any number of graphics cards and use them simultaneously, but in this tutorial we'll stick to the first graphics card that suits our needs.

We'll add a function `pickPhysicalDevice` and add a call to it in the `initVulkan` function.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
}

void pickPhysicalDevice() {
```

Physical devices and queue families

an®

- [Selecting a physical device](#)
- [Base device suitability checks](#)
- [Queue families](#)

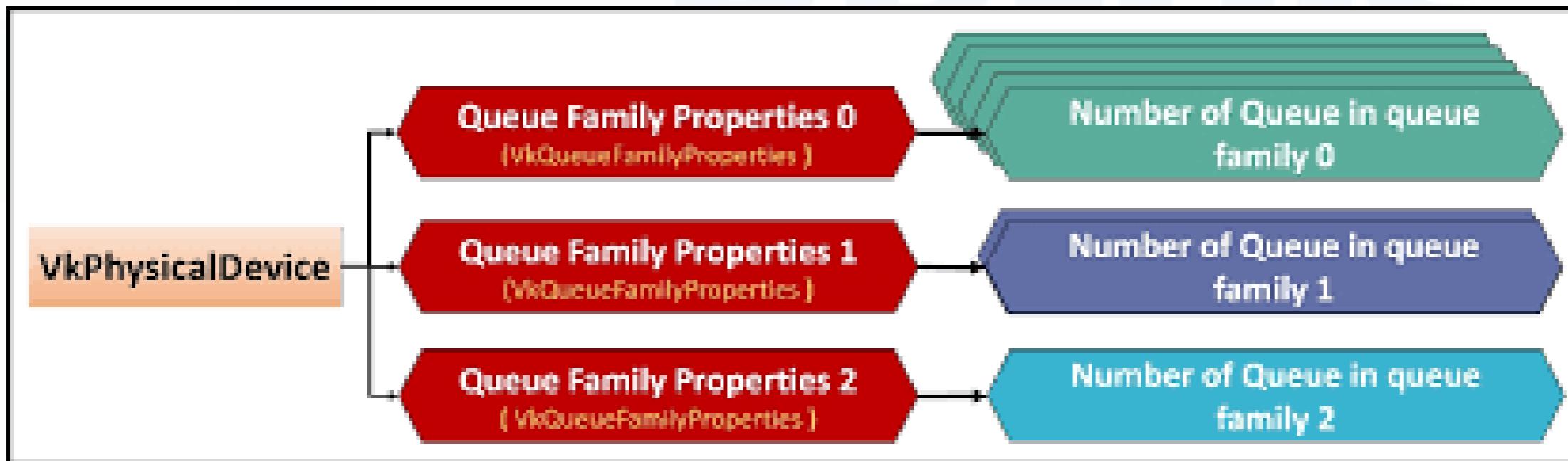
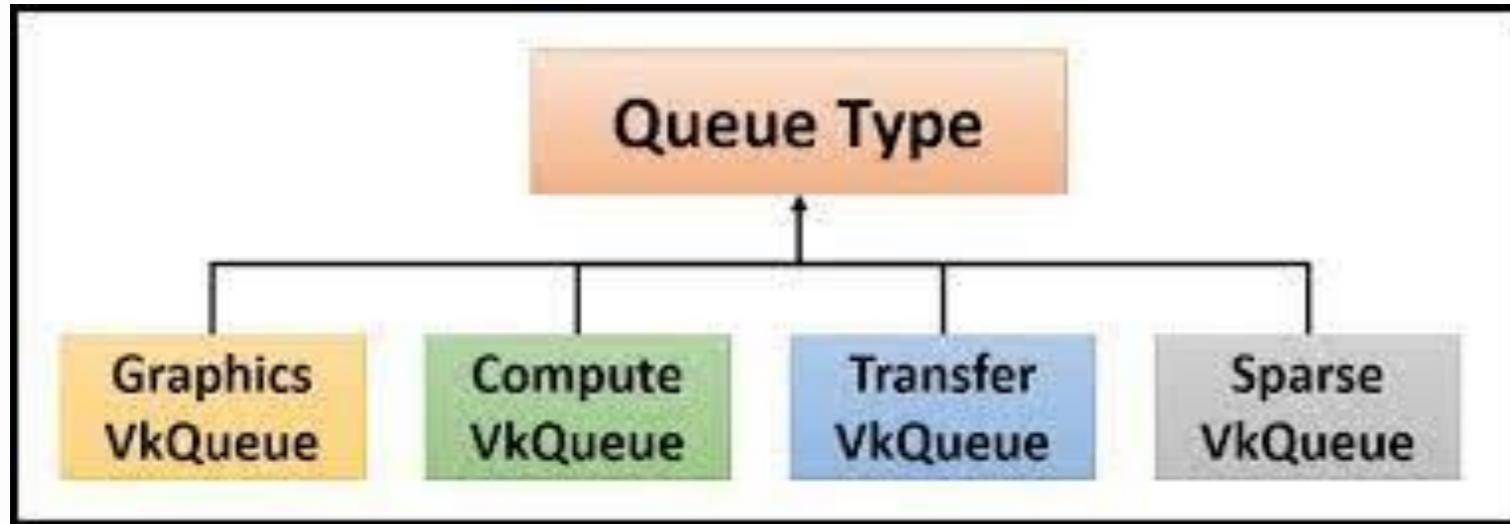
Selecting a physical device

After initializing the Vulkan library through a `VkInstance` we need to look for and select a graphics card in the system that supports the features we need. In fact we can select any number of graphics cards and use them simultaneously, but in this tutorial we'll stick to the first graphics card that suits our needs.

We'll add a function `pickPhysicalDevice` and add a call to it in the `initVulkan` function.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
}

void pickPhysicalDevice() {
```

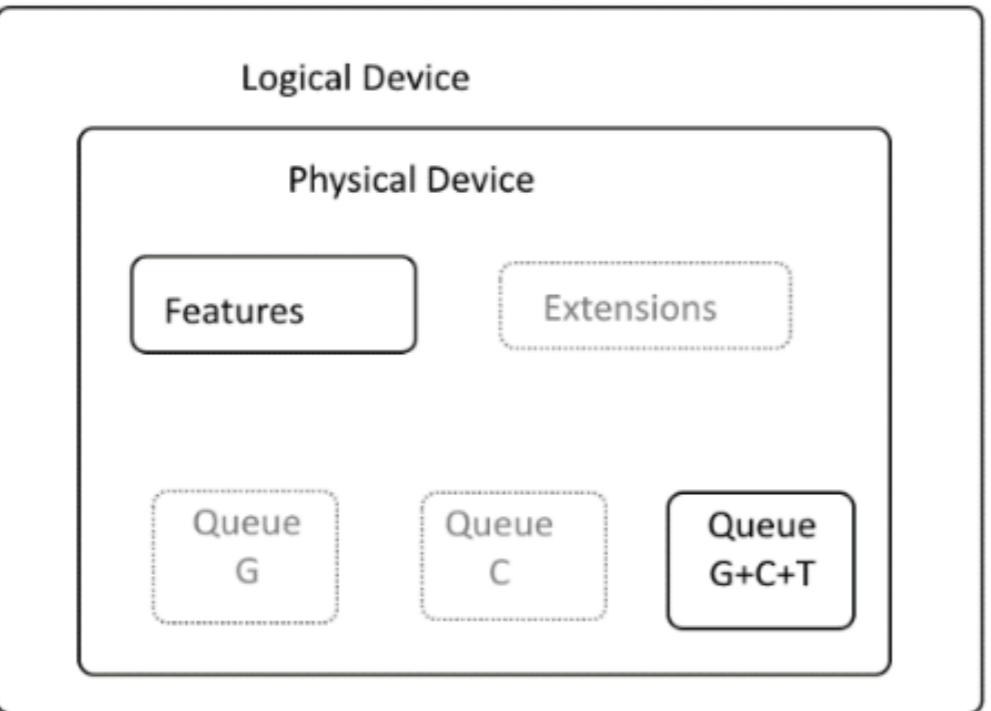


Logical device and queues

- [Introduction](#)
- [Specifying the queues to be created](#)
- [Specifying used device features](#)
- [Creating the logical device](#)
- [Retrieving queue handles](#)

Introduction

After selecting a physical device to use we need to set up a *logical device* to interface with it. The logical device creation process is similar to the instance creation process and describes the features we want to use. We also need to specify which queues to create now that we've queried which queue families are available. You can even create multiple logical devices from the same physical device if you have varying requirements.



The logical device allows us to perform almost all the work typically done in rendering applications, such as creating images and buffers, setting the pipeline state, or loading shaders. The most important ability it gives us is recording commands (such as issuing draw calls or dispatching computational work) and submitting them to queues, where they are executed and processed by the given hardware. After such execution, we acquire the results of the submitted operations. These can be a set of values calculated by compute shaders, or other data (not necessarily an image) generated by draw calls. All this is performed on a logical device, so now we will look at how to create one.

```
VkPhysicalDeviceFeatures deviceFeatures{};
```

Creating the logical device

With the previous two structures in place, we can start filling in the main [VkDeviceCreateInfo](#) structure.

```
VkDeviceCreateInfo createInfo{};  
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

First add pointers to the queue creation info and device features structs:

```
createInfo.pQueueCreateInfos = &queueCreateInfo;  
createInfo.queueCreateInfoCount = 1;  
  
createInfo.pEnabledFeatures = &deviceFeatures;
```

The remainder of the information bears a resemblance to the [VkInstanceCreateInfo](#) struct and requires you to specify extensions and validation layers. The difference is that these are device specific this time.

```
createInfo.enabledExtensionCount = 0;

if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
```

We won't need any device specific extensions for now.

That's it, we're now ready to instantiate the logical device with a call to the appropriately named [vkCreateDevice](#) function.

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical device!");
}
```

The parameters are the physical device to interface with, the queue and usage info we just specified, the optional allocation callbacks pointer and a pointer to a variable to store the logical device handle in. Similarly to the instance creation function, this call can return errors based on enabling non-existent extensions or specifying the desired usage of unsupported features.

Retrieving queue handles

The queues are automatically created along with the logical device, but we don't have a handle to interface with them yet. First add a class member to store a handle to the graphics queue:

```
VkQueue graphicsQueue;
```

Device queues are implicitly cleaned up when the device is destroyed, so we don't need to do anything in [cleanup](#).

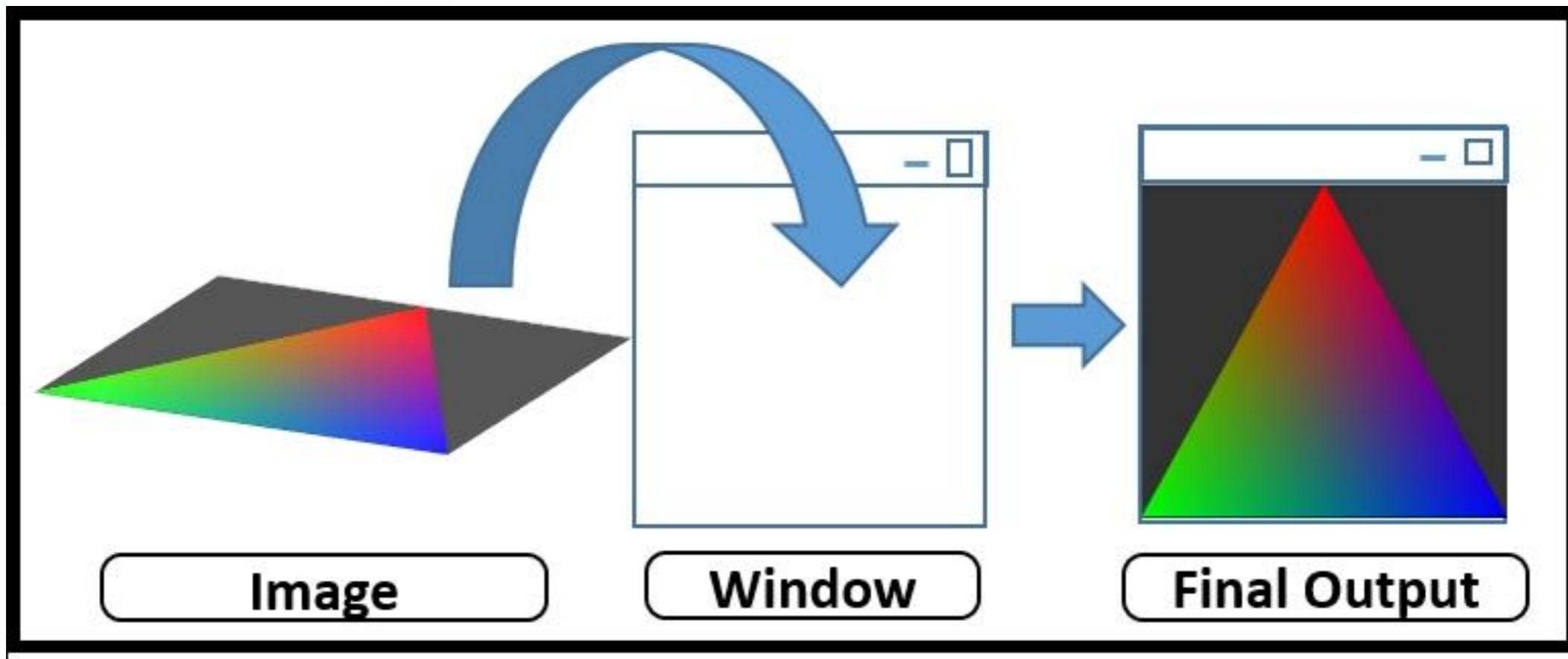
We can use the [vkGetDeviceQueue](#) function to retrieve queue handles for each queue family. The parameters are the logical device, queue family, queue index and a pointer to the variable to store the queue handle in. Because we're only creating a single queue from this family, we'll simply use index [0](#).

```
vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
```

With the logical device and queue handles we can now actually start using the graphics card to do things! In the next few chapters we'll set up the resources to present results to the window system.

Presentation

Vulkan®

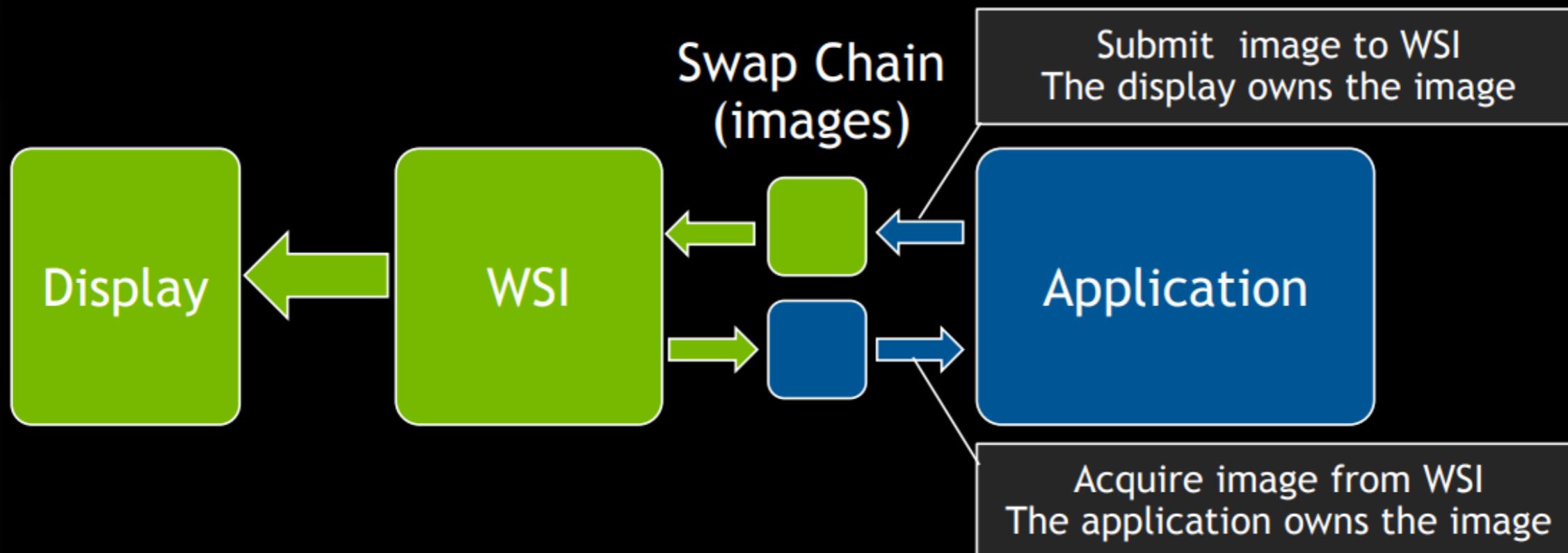


VULKAN WINDOW SYSTEM INTEGRATION (WSI)

WSI manages the ownership of images via a swap chain

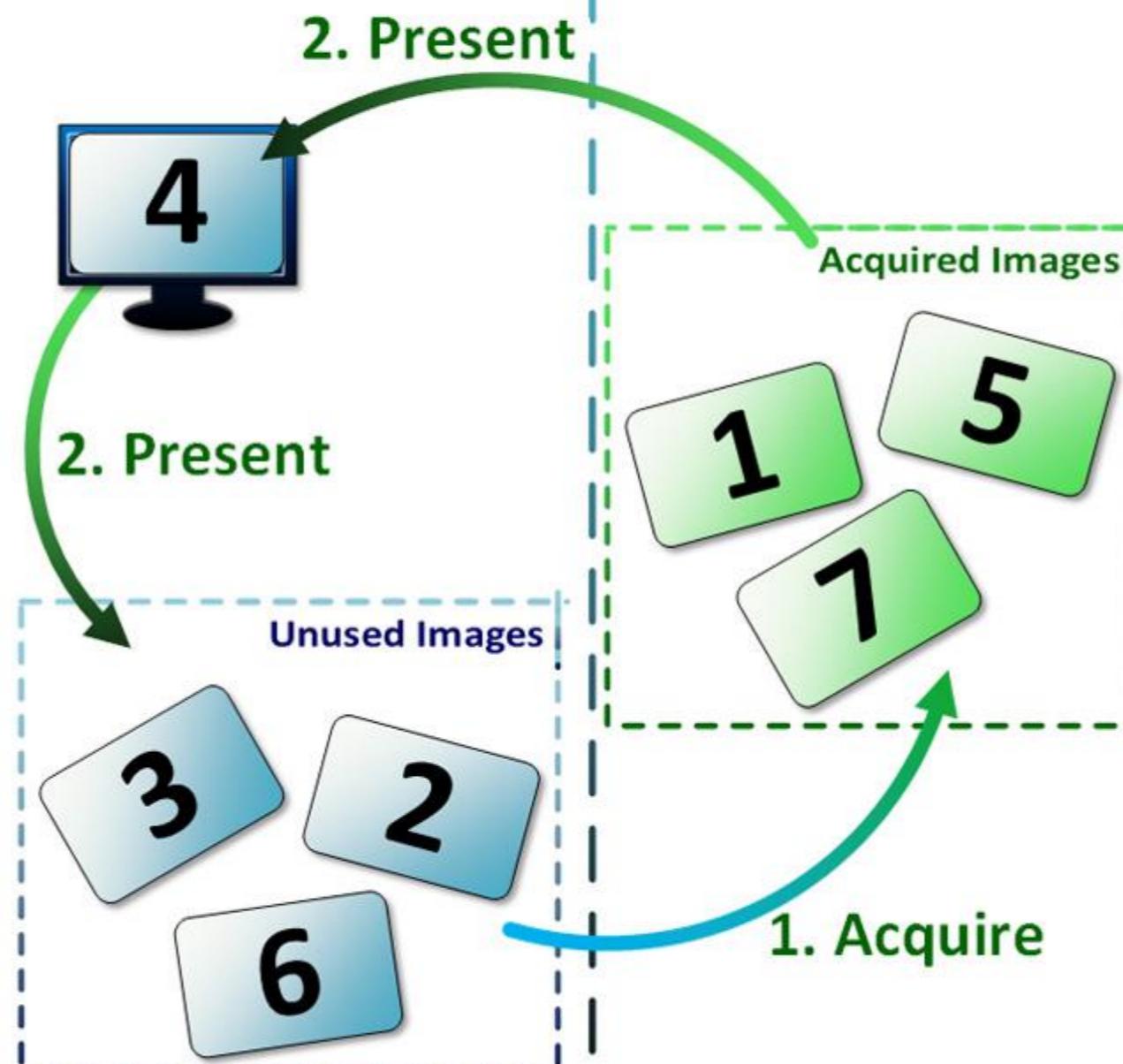
One image is presented while the other is rendered to

WSI is a Vulkan Extension



Presentation
Engine

Application



Window surface

- [Window surface creation](#)
- [Querying for presentation support](#)
- [Creating the presentation queue](#)

Since Vulkan is a platform agnostic API, it can not interface directly with the window system on its own. To establish the connection between Vulkan and the window system to present results to the screen, we need to use the WSI (Window System Integration) extensions. In this chapter we'll discuss the first one, which is `VK_KHR_surface`. It exposes a `VkSurfaceKHR` object that represents an abstract type of surface to present rendered images to. The surface in our program will be backed by the window that we've already opened with GLFW.

The `VK_KHR_surface` extension is an instance level extension and we've actually already enabled it, because it's included in the list returned by `glfwGetRequiredInstanceExtensions`. The list also includes some other WSI extensions that we'll use in the next couple of chapters.

The window surface needs to be created right after the instance creation, because it can actually influence the physical device selection. The reason we postponed this is because window surfaces are part of the larger topic of render targets and presentation for which the explanation would have cluttered the basic setup. It should also be noted that window surfaces are an entirely optional component in Vulkan, if you just need off-screen rendering. Vulkan allows you to do that without hacks like creating an invisible window (necessary for OpenGL).

To access native platform functions, you need to update the includes at the top:

```
#define VK_USE_PLATFORM_WIN32_KHR
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
#define GLFW_EXPOSE_NATIVE_WIN32
#include <GLFW/glfw3native.h>
```

The GLFW call takes simple parameters instead of a struct which makes the implementation of the function very straightforward:

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
        throw std::runtime_error("failed to create window surface!");
    }
}
```

The parameters are the `VkInstance`, GLFW window pointer, custom allocators and pointer to `VkSurfaceKHR` variable. It simply passes through the `VkResult` from the relevant platform call.

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    uint32_t queueFamilyCount = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

    std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());

    int i = 0;
    for (const auto& queueFamily : queueFamilies) {
        if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
            indices.graphicsFamily = i;
        }

        VkBool32 presentSupport = false;
        vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);

        if (presentSupport) {
            indices.presentFamily = i;
        }

        if (indices.isComplete()) {
            break;
        }

        i++;
    }

    return indices;
}
```

If the queue families are the same, then we only need to pass its index once. Finally, add a call to retrieve the queue handle:

```
vkGetDeviceQueue(device, indices.presentFamily.value(), 0, &presentQueue);
```

In case the queue families are the same, the two handles will most likely have the same value now. In the next chapter we're going to look at swap chains and how they give us the ability to present images to the surface.

Swap chain

- [Checking for swap chain support](#)
- [Enabling device extensions](#)
- [Querying details of swap chain support](#)
- [Choosing the right settings for the swap chain](#)
 - [Surface format](#)
 - [Presentation mode](#)
 - [Swap extent](#)
- [Creating the swap chain](#)
- [Retrieving the swap chain images](#)

Vulkan does not have the concept of a "default framebuffer", hence it requires an infrastructure that will own the buffers we will render to before we visualize them on the screen. This infrastructure is known as the *swap chain* and must be created explicitly in Vulkan. The swap chain is essentially a queue of images that are waiting to be presented to the screen. Our application will acquire such an image to draw to it, and then return it to the queue. How exactly the queue works and the conditions for presenting an image from the queue depend on how the swap chain is set up, but the general purpose of the swap chain is to synchronize the presentation of images with the refresh rate of the screen.

Querying details of swap chain support

Just checking if a swap chain is available is not sufficient, because it may not actually be compatible with our window surface. Creating a swap chain also involves a lot more settings than instance and device creation, so we need to query for some more details before we're able to proceed.

There are basically three kinds of properties we need to check:

- Basic surface capabilities (min/max number of images in swap chain, min/max width and height of images)
- Surface formats (pixel format, color space)
- Available presentation modes

Similar to `findQueueFamilies`, we'll use a struct to pass these details around once they've been queried. The three aforementioned types of properties come in the form of the following structs and lists of structs:

```
struct SwapChainSupportDetails {  
    VkSurfaceCapabilitiesKHR capabilities;  
    std::vector<VkSurfaceFormatKHR> formats;  
    std::vector<VkPresentModeKHR> presentModes;  
};
```

```
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface, &details.capabilities);
```

This function takes the specified `VkPhysicalDevice` and `VkSurfaceKHR` window surface into account when determining the supported capabilities. All of the support querying functions have these two as first parameters because they are the core components of the swap chain.

The next step is about querying the supported surface formats. Because this is a list of structs, it follows the familiar ritual of 2 function calls:

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);

if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data());
}
```

Make sure that the vector is resized to hold all the available formats. And finally, querying the supported presentation modes works exactly the same way with `vkGetPhysicalDeviceSurfacePresentModesKHR`:

```
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, nullptr);

if (presentModeCount != 0) {
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, details.presentModes.data());
}
```

Choosing the right settings for the swap chain

If the `swapChainAdequate` conditions were met then the support is definitely sufficient, but there may still be many different modes of varying optimality. We'll now write a couple of functions to find the right settings for the best possible swap chain. There are three types of settings to determine:

- Surface format (color depth)
- Presentation mode (conditions for "swapping" images to the screen)
- Swap extent (resolution of images in swap chain)

For each of these settings we'll have an ideal value in mind that we'll go with if it's available and otherwise we'll create some logic to find the next best thing.

Surface format

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>& availableFormats) {
    for (const auto& availableFormat : availableFormats) {
        if (availableFormat.format == VK_FORMAT_B8G8R8A8_SRGB && availableFormat.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
            return availableFormat;
        }
    }

    return availableFormats[0];
}
```

Presentation mode

The presentation mode is arguably the most important setting for the swap chain, because it represents the actual conditions for showing images to the screen. There are four possible modes available in Vulkan:

- `VK_PRESENT_MODE_IMMEDIATE_KHR`: Images submitted by your application are transferred to the screen right away, which may result in tearing.
- `VK_PRESENT_MODE_FIFO_KHR`: The swap chain is a queue where the display takes an image from the front of the queue when the display is refreshed and the program inserts rendered images at the back of the queue. If the queue is full then the program has to wait. This is most similar to vertical sync as found in modern games. The moment that the display is refreshed is known as "vertical blank".
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: This mode only differs from the previous one if the application is late and the queue was empty at the last vertical blank. Instead of waiting for the next vertical blank, the image is transferred right away when it finally arrives. This may result in visible tearing.
- `VK_PRESENT_MODE_MAILBOX_KHR`: This is another variation of the second mode. Instead of blocking the application when the queue is full, the images that are already queued are simply replaced with the newer ones. This mode can be used to render frames as fast as possible while still avoiding tearing, resulting in fewer latency issues than standard vertical sync. This is commonly known as "triple buffering", although the existence of three buffers alone does not necessarily mean that the framerate is unlocked.

Only the `VK_PRESENT_MODE_FIFO_KHR` mode is guaranteed to be available, so we'll again have to write a function that looks for the best mode that is available:

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR>& availablePresentModes) {
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

Creating the swap chain

```
void createSwapChain() {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(physicalDevice);

    VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
    VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes);
    VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);

    uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
    if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount > swapChainSupport.capabilities.maxImageCount) {
        imageCount = swapChainSupport.capabilities.maxImageCount;
    }

    VkSwapchainCreateInfoKHR createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
    createInfo.surface = surface;

    createInfo.minImageCount = imageCount;
    createInfo.imageFormat = surfaceFormat.format;
    createInfo.imageColorSpace = surfaceFormat.colorSpace;
    createInfo.imageExtent = extent;
    createInfo.imageArrayLayers = 1;
    createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
uint32_t queueFamilyIndices[] = {indices.graphicsFamily.value(), indices.presentFamily.value()};

if (indices.graphicsFamily != indices.presentFamily) {
    createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
    createInfo.queueFamilyIndexCount = 2;
    createInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
    createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
}

createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;

if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
    throw std::runtime_error("failed to create swap chain!");
}

vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, swapChainImages.data());

swapChainImageFormat = surfaceFormat.format;
swapChainExtent = extent;
}

void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat);
    }
}
```

Image views

To use any `VkImage`, including those in the swap chain, in the render pipeline we have to create a `VkImageView` object. An image view is quite literally a view into an image. It describes how to access the image and which part of the image to access, for example if it should be treated as a 2D texture depth texture without any mipmapping levels.

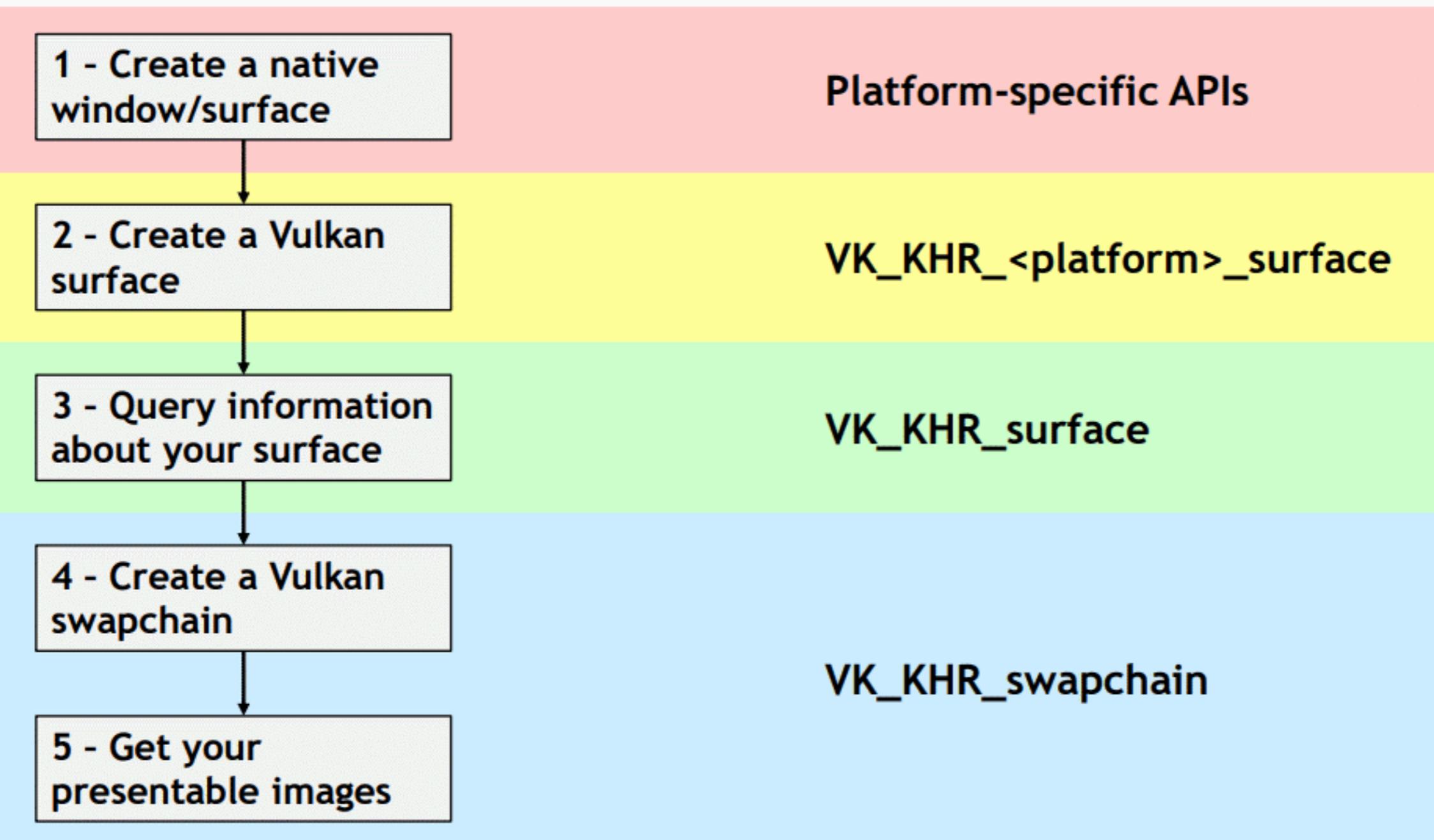
In this chapter we'll write a `createImageViews` function that creates a basic image view for every image in the swap chain so that we can use them as color targets later on.

```
VkImageView createImageView(VkImage image, VkFormat format) {
    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;
    if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
        throw std::runtime_error("failed to create image view!");
    }

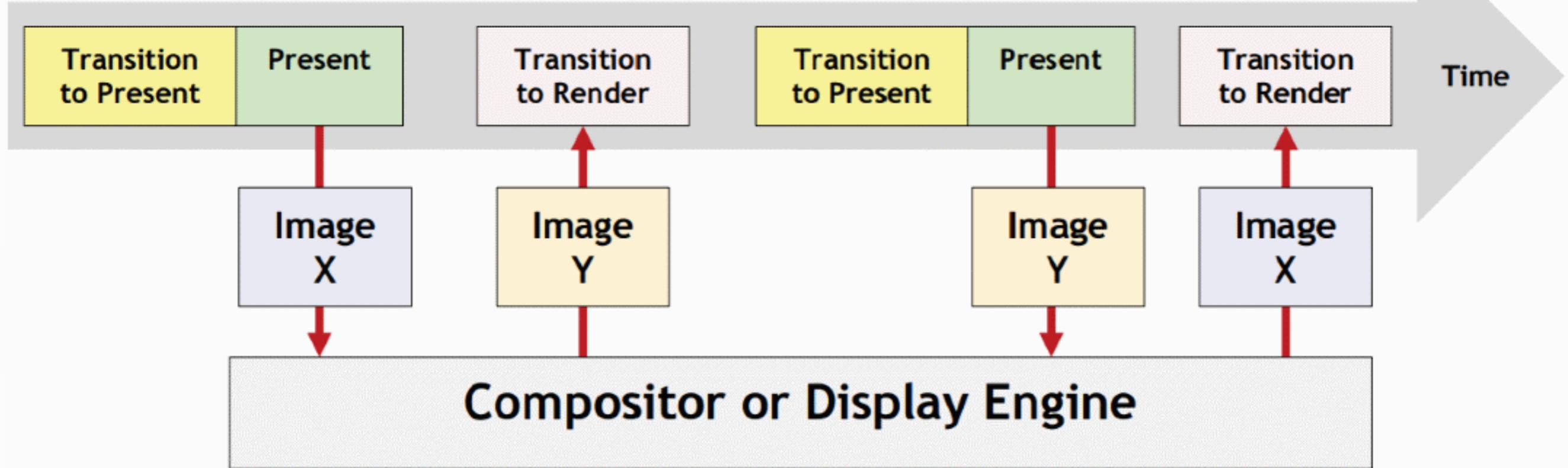
    return imageView;
}
```

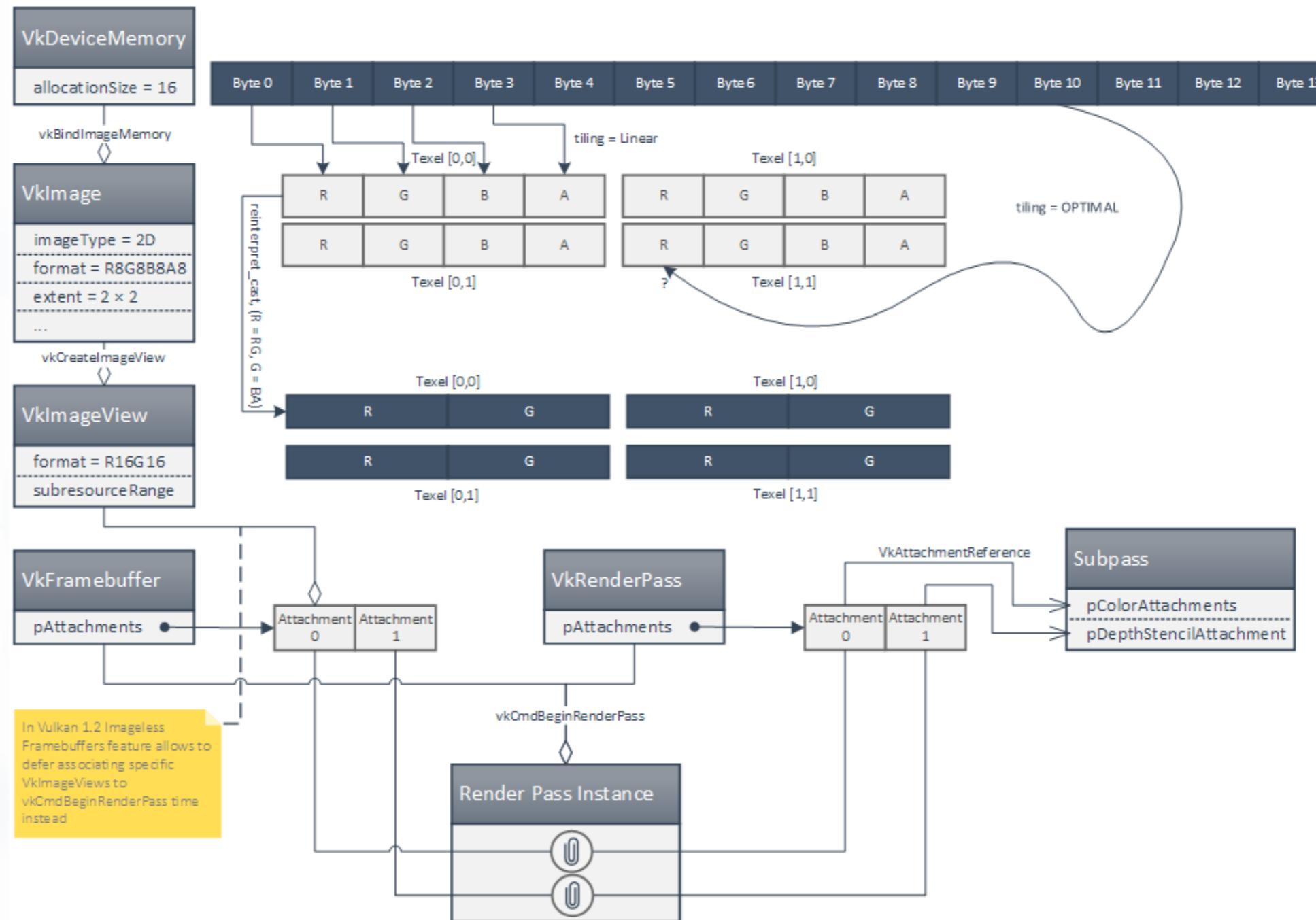
Steps to setup your presentable images





VkQueue





Shader modules

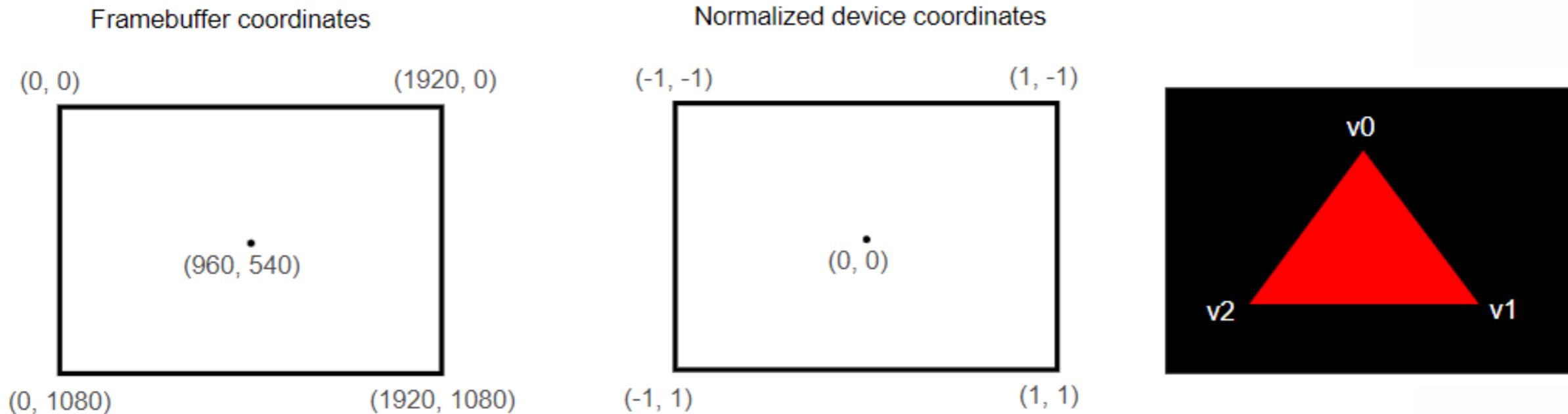
- [Vertex shader](#)
- [Fragment shader](#)
- [Per-vertex colors](#)
- [Compiling the shaders](#)
- [Loading a shader](#)
- [Creating shader modules](#)
- [Shader stage creation](#)

Unlike earlier APIs, shader code in Vulkan has to be specified in a bytecode format as opposed to human-readable syntax like [GLSL](#) and [HLSL](#). This bytecode format is called [SPIR-V](#) and is designed to be used with both Vulkan and OpenCL (both Khronos APIs). It is a format that can be used to write graphics and compute shaders, but we will focus on shaders used in Vulkan's graphics pipelines in this tutorial.

Vertex shader

The vertex shader processes each incoming vertex. It takes its attributes, like world position, color, normal and texture coordinates as input. The output is the final position in clip coordinates and the attributes that need to be passed on to the fragment shader, like color and texture coordinates. These values will then be interpolated over the fragments by the rasterizer to produce a smooth gradient.

A *clip coordinate* is a four dimensional vector from the vertex shader that is subsequently turned into a *normalized device coordinate* by dividing the whole vector by its last component. These normalized device coordinates are **homogeneous coordinates** that map the framebuffer to a [-1, 1] by [-1, 1] coordinate system that looks like the following:



GLSL

Shaders are written in the C-like language GLSL. GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms and its `main` function. Each shader's entry point is at its `main` function where we process any input variables and output the results in its output variables. Don't worry if you don't know what uniforms are, we'll get to those shortly.

A shader typically has the following structure:

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
    // process input(s) and do some weird graphics stuff
    ...
    // output processed stuff to output variable
    out_variable_name = weird_stuff_we_processed;
}
```

Types

GLSL has, like any other programming language, data types for specifying what kind of variable we want to work with. GLSL has most of the default basic types we know from languages like C: **int**, **float**, **double**, **uint** and **bool**. GLSL also features two container types that we'll be using a lot, namely **vectors** and **matrices**. We'll discuss matrices in a later chapter.

Vectors

A vector in GLSL is a 2,3 or 4 component container for any of the basic types just mentioned. They can take the following form (**n** represents the number of components):

- **vecn**: the default vector of **n** floats.
- **bvecn**: a vector of **n** booleans.
- **ivec n** : a vector of **n** integers.
- **uvec n** : a vector of **n** unsigned integers.
- **dvec n** : a vector of **n** double components.

Most of the time we will be using the basic **vecn** since floats are sufficient for most of our purposes.

Components of a vector can be accessed via **vec.x** where **x** is the first component of the vector. You can use **.x**, **.y**, **.z** and **.w** to access their first, second, third and fourth component respectively. GLSL also allows you to use **rgba** for colors or **stpq** for texture coordinates, accessing the same components.

Ins and outs

Shaders are nice little programs on their own, but they are part of a whole and for that reason we want to have inputs and outputs on the individual shaders so that we can move stuff around. GLSL defined the **in** and **out** keywords specifically for that purpose. Each shader can specify inputs and outputs using those keywords and wherever an output variable matches with an input variable of the next shader stage they're passed along. The vertex and fragment shader differ a bit though.

The vertex shader **should** receive some form of input otherwise it would be pretty ineffective. The vertex shader differs in its input, in that it receives its input straight from the vertex data. To define how the vertex data is organized we specify the input variables with location metadata so we can configure the vertex attributes on the CPU. We've seen this in the previous chapter as **layout (location = 0)**. The vertex shader thus requires an extra layout specification for its inputs so we can link it with the vertex data.

It is also possible to omit the **layout (location = 0)** specifier and query for the attribute locations in your OpenGL code via [`glGetAttribLocation`](#), but I'd prefer to set them in the vertex shader. It is easier to understand and saves you (and OpenGL) some work.

The other exception is that the fragment shader requires a **vec4** color output variable, since the fragment shaders needs to generate a final output color. If you fail to specify an output color in your fragment shader, the color buffer output for those fragments will be undefined (which usually means OpenGL will render them either black or white).

Uniforms

Uniforms are another way to pass data from our application on the CPU to the shaders on the GPU. Uniforms are however slightly different compared to vertex attributes. First of all, uniforms are global. Global, meaning that a uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program. Second, whatever you set the uniform value to, uniforms will keep their values until they're either reset or updated.

To declare a uniform in GLSL we simply add the `uniform` keyword to a shader with a type and a name. From that point on we can use the newly declared uniform in the shader. Let's see if this time we can set the color of the triangle via a uniform:

```
1 #version 450
2
3 layout(binding = 0) uniform UniformBufferObject {
4     mat4 model;
5     mat4 view;
6     mat4 proj;
7 } ubo;
```

```
#version 450

vec2 positions[3] = vec2[](  
    vec2(0.0, -0.5),  
    vec2(0.5, 0.5),  
    vec2(-0.5, 0.5)  
);  
  
void main() {  
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);  
}
```

The `main` function is invoked for every vertex. The built-in `gl_VertexIndex` variable contains the index of the current vertex. This is usually an index into the vertex buffer, but in our case it will be an index into a hardcoded array of vertex data. The position of each vertex is accessed from the constant array in the shader and combined with dummy `z` and `w` components to produce a position in clip coordinates. The built-in variable `gl_Position` functions as the output.

Fragment shader

The triangle that is formed by the positions from the vertex shader fills an area on the screen with fragments. The fragment shader is invoked on these fragments to produce a color and depth for the framebuffer (or framebuffers). A simple fragment shader that outputs the color red for the entire triangle looks like this:

```
#version 450

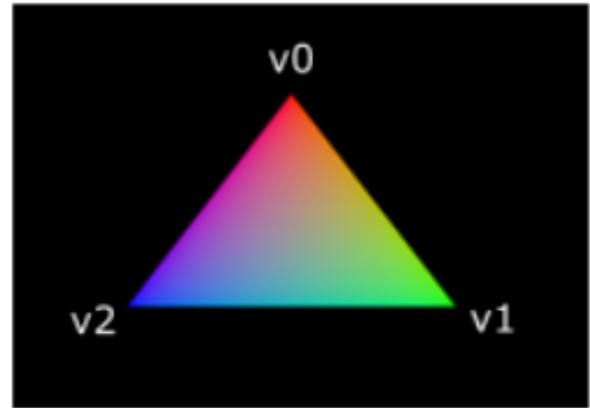
layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

The `main` function is called for every fragment just like the vertex shader `main` function is called for every vertex. Colors in GLSL are 4-component vectors with the R, G, B and alpha channels within the [0, 1] range. Unlike `gl_Position` in the vertex shader, there is no built-in variable to output a color for the current fragment. You have to specify your own output variable for each framebuffer where the `layout(location = 0)` modifier specifies the index of the framebuffer. The color red is written to this `outColor` variable that is linked to the first (and only) framebuffer at index `0`.

Per-vertex colors

Making the entire triangle red is not very interesting, wouldn't something like the following look a lot nicer?



We have to make a couple of changes to both shaders to accomplish this. First off, we need to specify a distinct color for each of the three vertices. The vertex shader should now include an array with colors just like it does for positions:

```
vec3 colors[3] = vec3[](  
    vec3(1.0, 0.0, 0.0),  
    vec3(0.0, 1.0, 0.0),  
    vec3(0.0, 0.0, 1.0)  
,
```

Now we just need to pass these per-vertex colors to the fragment shader so it can output their interpolated values to the framebuffer. Add an output for color to the vertex shader and write to it in the `main` function:

```
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

Next, we need to add a matching input in the fragment shader:

```
layout(location = 0) in vec3 fragColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

The input variable does not necessarily have to use the same name, they will be linked together using the indexes specified by the `location` directives. The `main` function has been modified to output the color along with an alpha value. As shown in the image above, the values for `fragColor` will be automatically interpolated for the fragments between the three vertices, resulting in a smooth gradient.

Shader Compilation to SPIR-V



We're now going to compile these into SPIR-V bytecode using the `glslc` program.

Windows

Create a `compile.bat` file with the following contents:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv  
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv  
pause
```

Replace the path to `glslc.exe` with the path to where you installed the Vulkan SDK. Double click the file to run it.

Note: You could instead use **glslangValidator**

Shader stage creation

To actually use the shaders we'll need to assign them to a specific pipeline stage through `VkPipelineShaderStageCreateInfo` structures as part of the actual pipeline creation process.

We'll start by filling in the structure for the vertex shader, again in the `createGraphicsPipeline` function.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};  
vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
```

The first step, besides the obligatory `sType` member, is telling Vulkan in which pipeline stage the shader is to be used. There is an enum value for each of the programmable stages described in the previous chapter.

```
vertShaderStageInfo.module = vertShaderModule;  
vertShaderStageInfo.pName = "main";
```

Vertex input

The `VkPipelineVertexInputStateCreateInfo` structure describes the format of the vertex data that will be passed to the vertex shader. It describes this in roughly two ways:

- Bindings: spacing between data and whether the data is per-vertex or per-instance (see [instancing](#))
- Attribute descriptions: type of the attributes passed to the vertex shader, which binding to load them from and at which offset

Because we're hard coding the vertex data directly in the vertex shader, we'll fill in this structure to specify that there is no vertex data to load for now. We'll get back to it in the vertex buffer chapter.

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};  
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;  
vertexInputInfo.vertexBindingDescriptionCount = 0;  
vertexInputInfo.pVertexBindingDescriptions = nullptr; // Optional  
vertexInputInfo.vertexAttributeDescriptionCount = 0;  
vertexInputInfo.pVertexAttributeDescriptions = nullptr; // Optional
```

Input assembly

The `VkPipelineInputAssemblyStateCreateInfo` struct describes two things: what kind of geometry will be drawn from the vertices and if primitive restart should be enabled. The former is specified in the `topology` member and can have values like:

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`: points from vertices
- `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`: line from every 2 vertices without reuse
- `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`: the end vertex of every line is used as start vertex for the next line
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`: triangle from every 3 vertices without reuse
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`: the second and third vertex of every triangle are used as first two vertices of the next triangle

Normally, the vertices are loaded from the vertex buffer by index in sequential order, but with an *element buffer* you can specify the indices to use yourself. This allows you to perform optimizations like reusing vertices. If you set the `primitiveRestartEnable` member to `VK_TRUE`, then it's possible to break up lines and triangles in the `_STRIP` topology modes by using a special index of `0xFFFF` or `0xFFFFFFFF`.

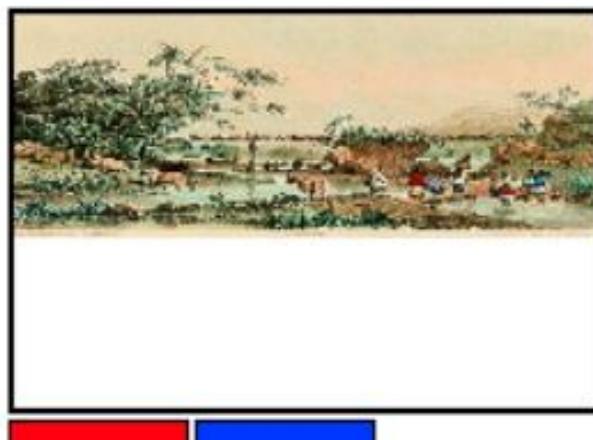
We intend to draw triangles throughout this tutorial, so we'll stick to the following data for the structure:

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly{};  
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;  
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;  
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

Viewports and scissors

A viewport basically describes the region of the framebuffer that the output will be rendered to. This will almost always be `(0, 0)` to `(width, height)` and in this tutorial that will also be the case.

```
VkViewport viewport{};  
viewport.x = 0.0f;  
viewport.y = 0.0f;  
viewport.width = (float) swapChainExtent.width;  
viewport.height = (float) swapChainExtent.height;  
viewport.minDepth = 0.0f;  
viewport.maxDepth = 1.0f;
```



Viewport
Scissor
rectangle



Viewport
Scissor
rectangle

We Work In-Depth

Rasterizer

The rasterizer takes the geometry that is shaped by the vertices from the vertex shader and turns it into fragments to be colored by the fragment shader. It also performs [depth testing](#), [face culling](#) and the scissor test, and it can be configured to output fragments that fill entire polygons or just the edges (wireframe rendering). All this is configured using the [`VkPipelineRasterizationStateCreateInfo`](#) structure.

```
VkPipelineRasterizationStateCreateInfo rasterizer{};  
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;  
rasterizer.depthClampEnable = VK_FALSE;
```

If `depthClampEnable` is set to `VK_TRUE`, then fragments that are beyond the near and far planes are clamped to them as opposed to discarding them. This is useful in some special cases like shadow maps. Using this requires enabling a GPU feature.

```
rasterizer.rasterizerDiscardEnable = VK_FALSE;
```

If `rasterizerDiscardEnable` is set to `VK_TRUE`, then geometry never passes through the rasterizer stage. This basically disables any output to the framebuffer.

```
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
```

Color blending

After a fragment shader has returned a color, it needs to be combined with the color that is already in the framebuffer. This transformation is known as color blending and there are two ways to do it:

- Mix the old and new value to produce a final color
- Combine the old and new value using a bitwise operation

There are two types of structs to configure color blending. The first struct, [VkPipelineColorBlendAttachmentState](#), contains the configuration per attached framebuffer and the second struct, [VkPipelineColorBlendStateCreateInfo](#), contains the *global* color blending settings. In our case we only have one framebuffer:

```
VkPipelineColorBlendAttachmentState colorBlendAttachment{};  
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_  
COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;  
colorBlendAttachment.blendEnable = VK_FALSE;  
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional  
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional  
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional  
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional  
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional  
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```

GRAPHICS PIPELINE

Snapshot of all States

Including Shaders

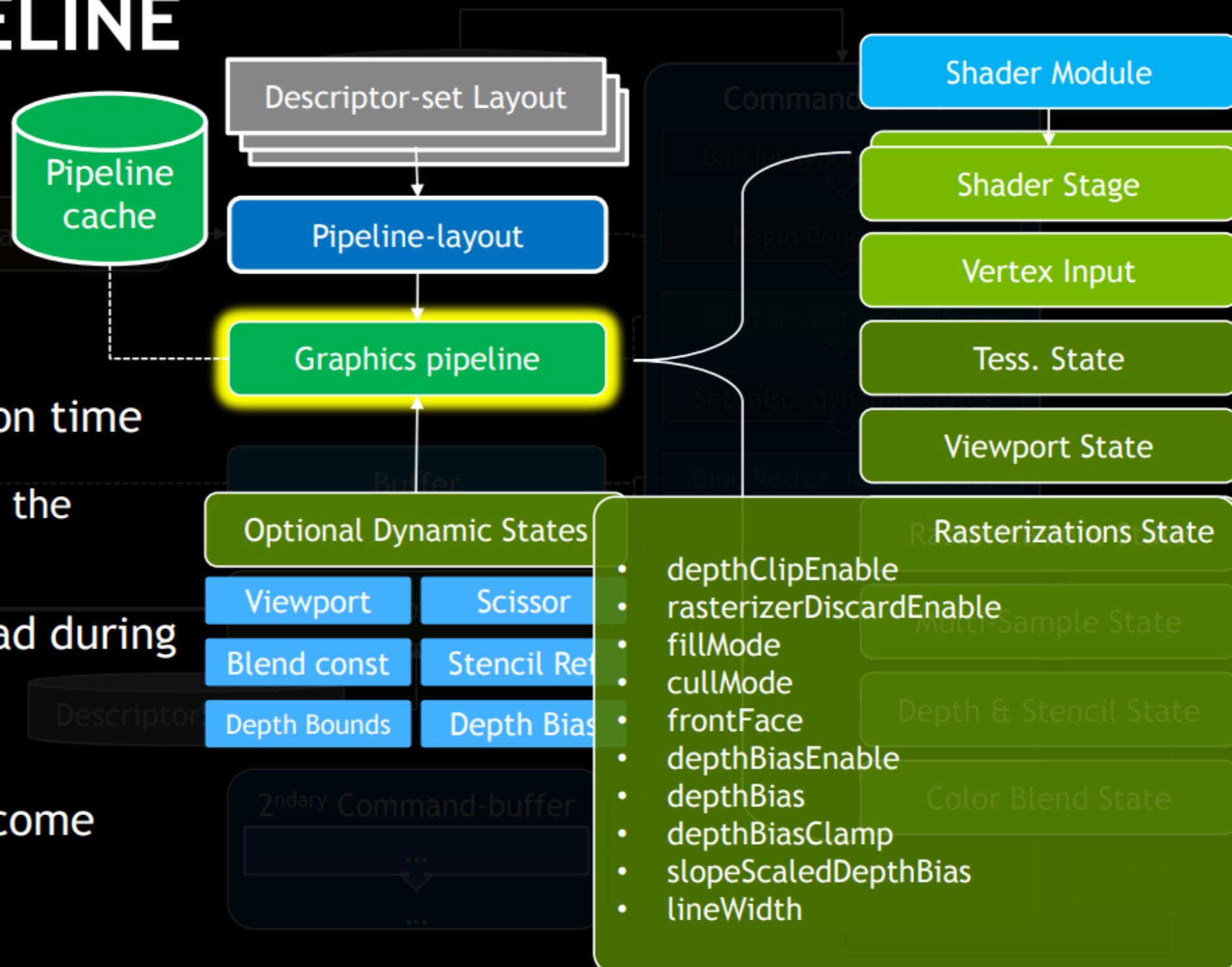
Pre-compiled & Immutable

Ideally: done at Initialization time

Ok at render-time *if* using the Pipeline-Cache

Prevents validation overhead during rendering loop

Some Render-states can be excluded from it: they become “Dynamic” States



GRAPHICS PIPELINE

Graphics Pipeline must be consistent with shaders

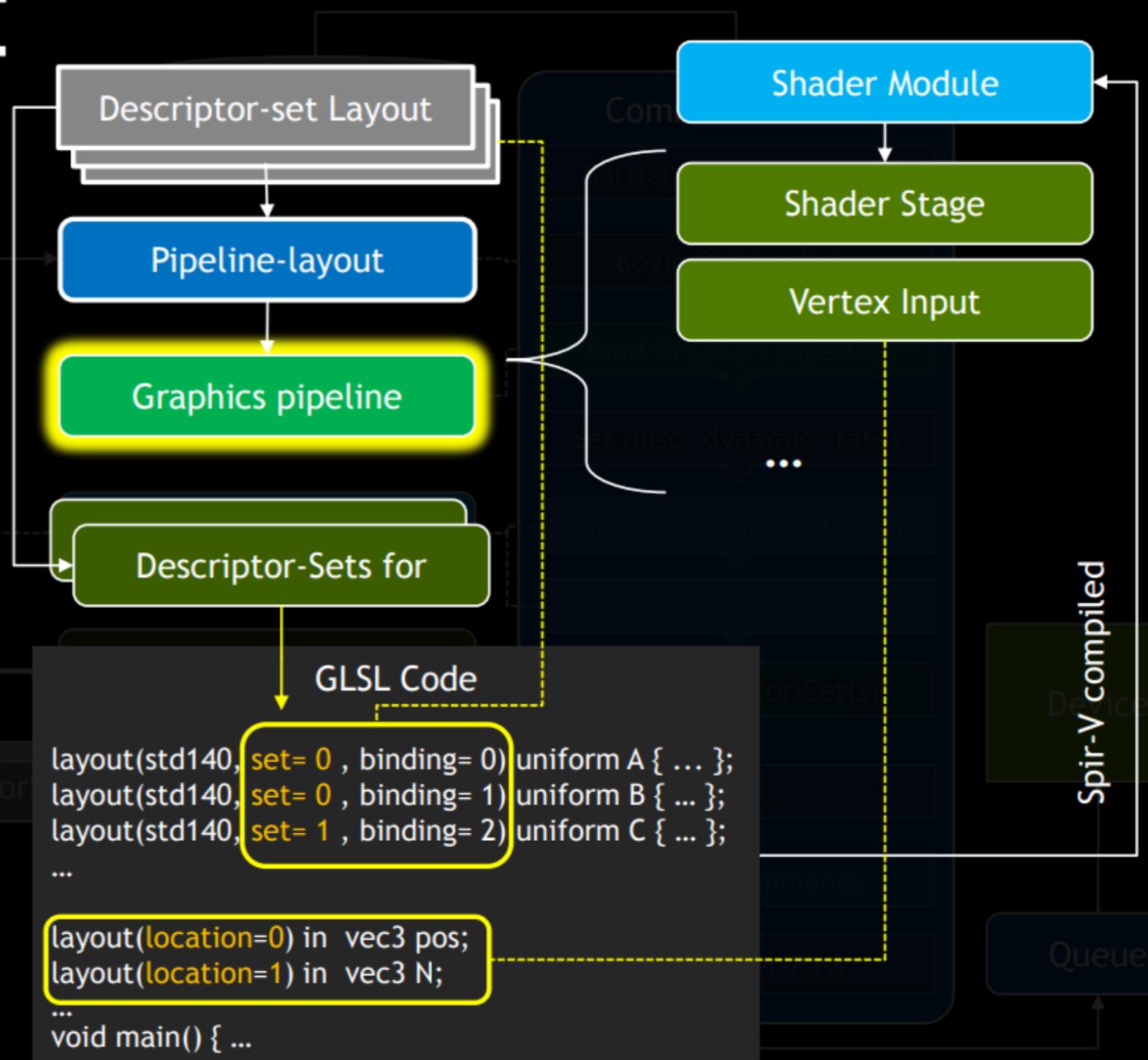
No “introspection”, so everything known & prepared in advance

Vertex Input:

tells how Attributes: Locations are attached to which Vertex Buffer at which offset

Pipeline Layout:

Tells how to map Sets and Bindings for the shaders at each stage (Vtx, Fragment, Geom...)



Pipeline layout

You can use `uniform` values in shaders, which are globals similar to dynamic state variables that can be changed at drawing time to alter the behavior of your shaders without having to recreate them. They are commonly used to pass the transformation matrix to the vertex shader, or to create texture samplers in the fragment shader.

These uniform values need to be specified during pipeline creation by creating a `VkPipelineLayout` object. Even though we won't be using them until a future chapter, we are still required to create an empty pipeline layout.

Create a class member to hold this object, because we'll refer to it from other functions at a later point in time:

```
VkPipelineLayout pipelineLayout;
```

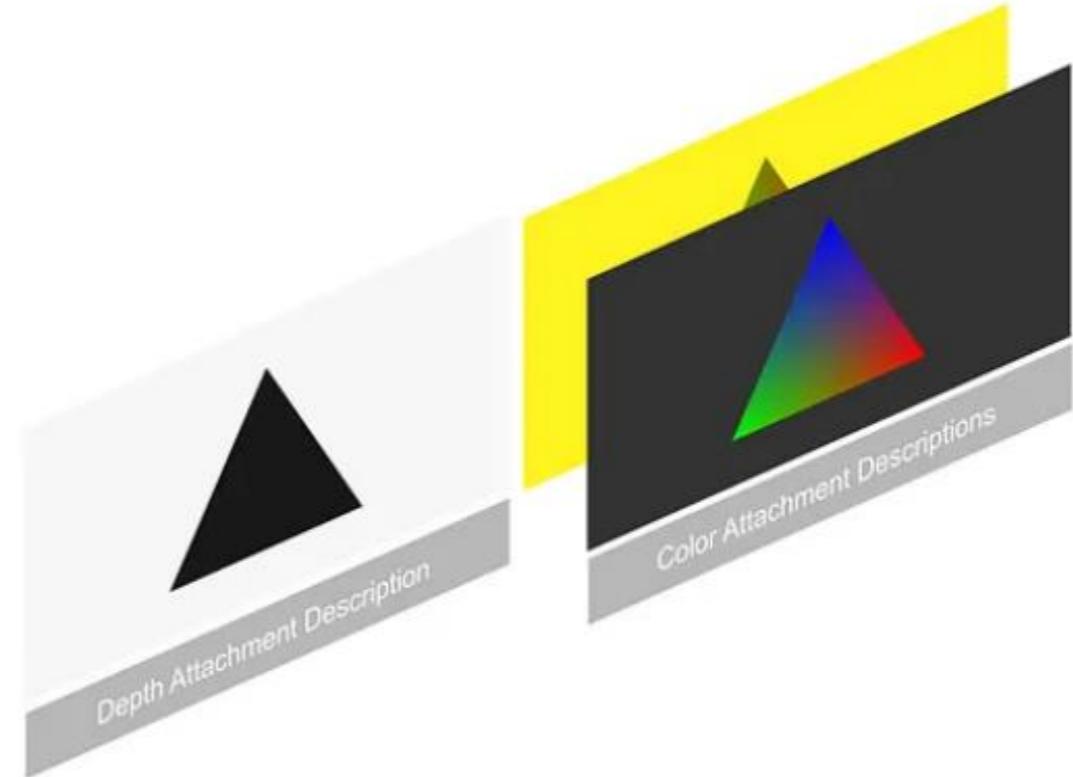
And then create the object in the `createGraphicsPipeline` function:

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optional
pipelineLayoutInfo.pSetLayouts = nullptr; // Optional
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optional

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {
    throw std::runtime_error("failed to create pipeline layout!");
}
```

Render passes

- [Setup](#)
- [Attachment description](#)
- [Subpasses and attachment references](#)
- [Render pass](#)



Setup

Before we can finish creating the pipeline, we need to tell Vulkan about the framebuffer attachments that will be used while rendering. We need to specify how many color and depth buffers there will be, how many samples to use for each of them and how their contents should be handled throughout the rendering operations. All of this information is wrapped in a *render pass* object, for which we'll create a new `createRenderPass` function. Call this function from `initVulkan` before `createGraphicsPipeline`.

Framebuffers

The attachments specified during render pass creation are bound by wrapping them into a `VkFramebuffer` object. A framebuffer object references all of the `VkImageView` objects that represent the attachments. In our case that will be only a single one: the color attachment. However, the image that we have to use for the attachment depends on which image the swap chain returns when we retrieve one for presentation. That means that we have to create a framebuffer for all of the images in the swap chain and use the one that corresponds to the retrieved image at drawing time.



Command buffers

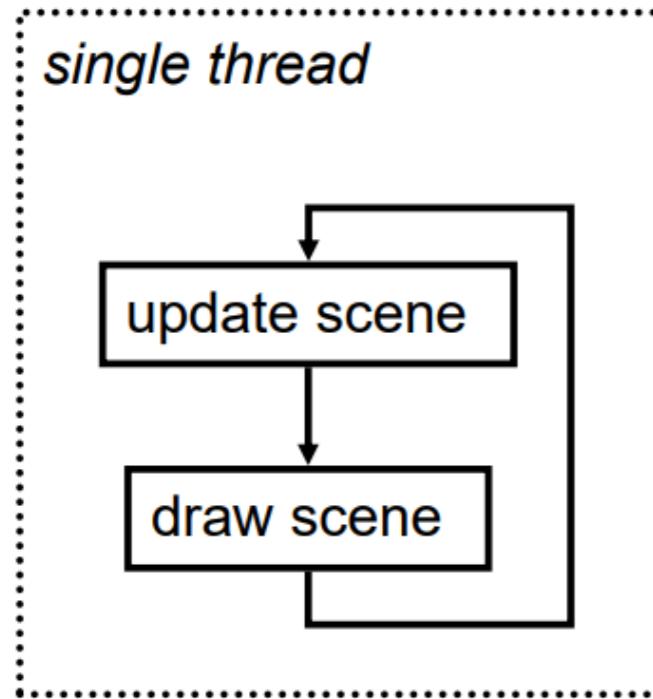
- [Command pools](#)
- [Command buffer allocation](#)
- [Command buffer recording](#)
- [Starting a render pass](#)
- [Basic drawing commands](#)
- [Finishing up](#)

Commands in Vulkan, like drawing operations and memory transfers, are not executed directly using function calls. You have to record all of the operations you want to perform in command buffer objects. The advantage of this is that when we are ready to tell the Vulkan what we want to do, all of the commands are submitted together and Vulkan can more efficiently process the commands since all of them are available together. In addition, this allows command recording to happen in multiple threads if so desired.

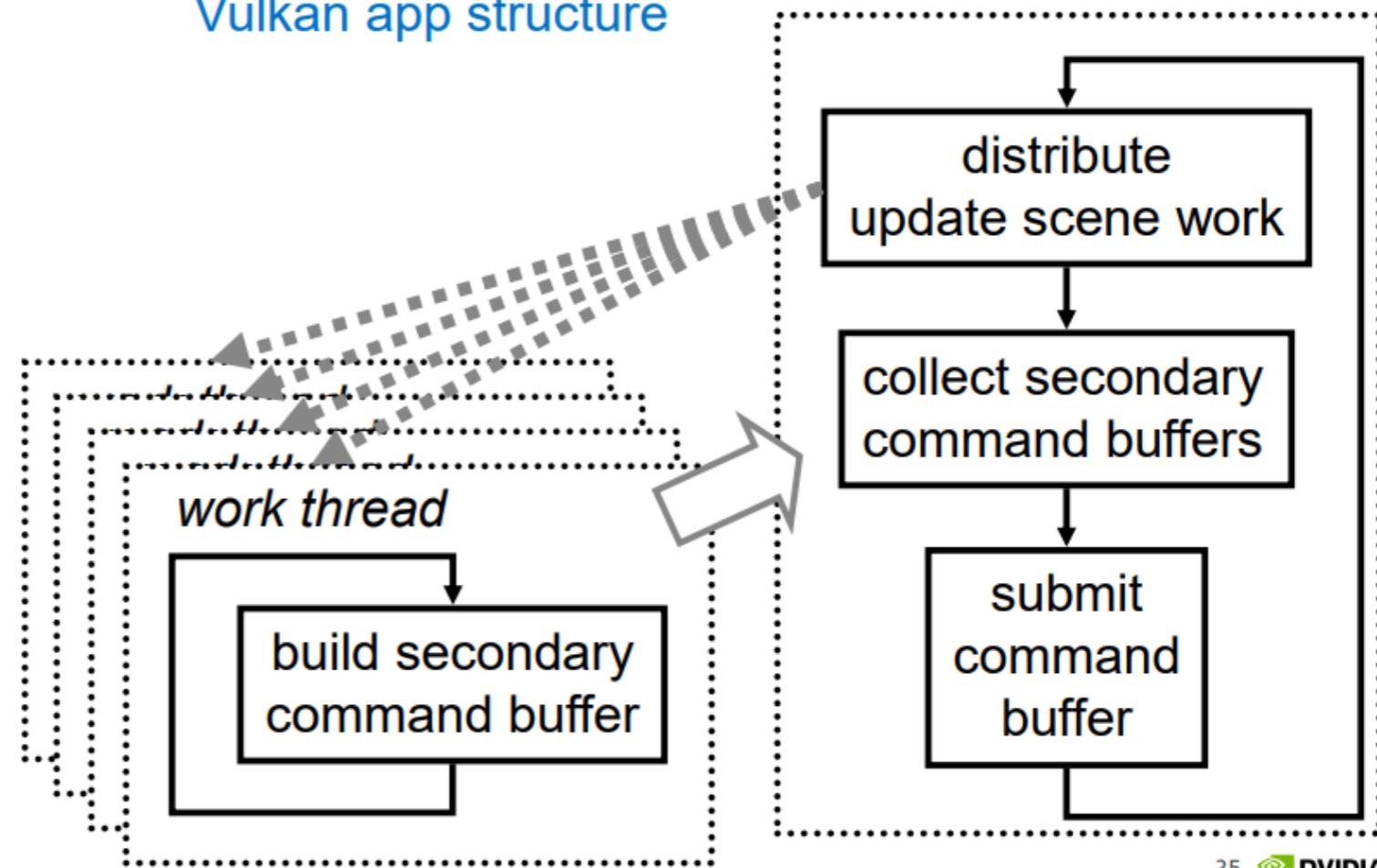
Vulkan Application Structure

Parallel Command Buffer Generation

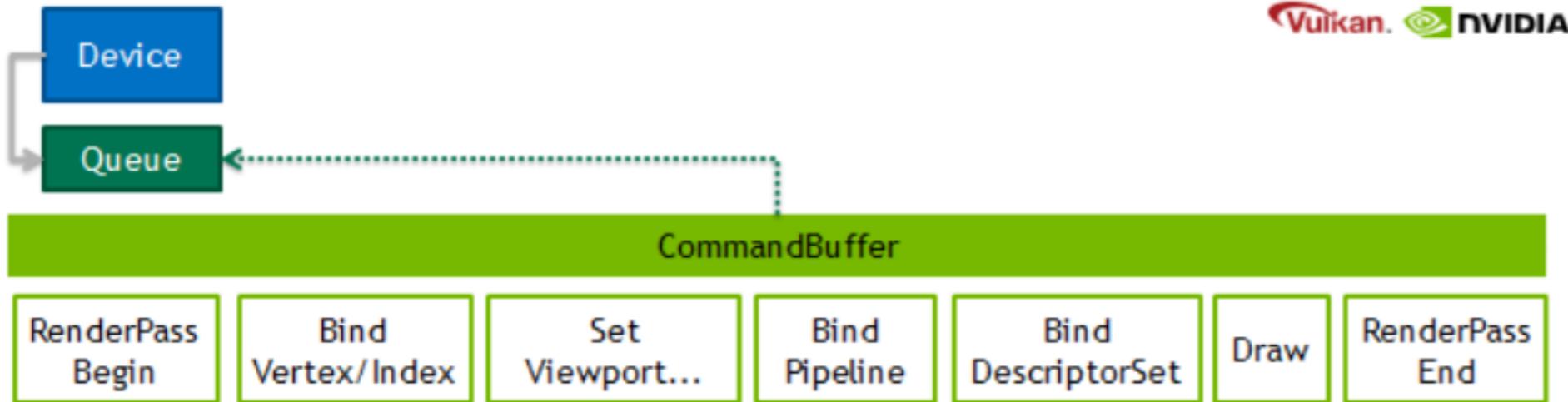
Traditional single-threaded
OpenGL app structure



Possible multi-threaded
Vulkan app structure



Command Submission



Vulkan. NVIDIA.

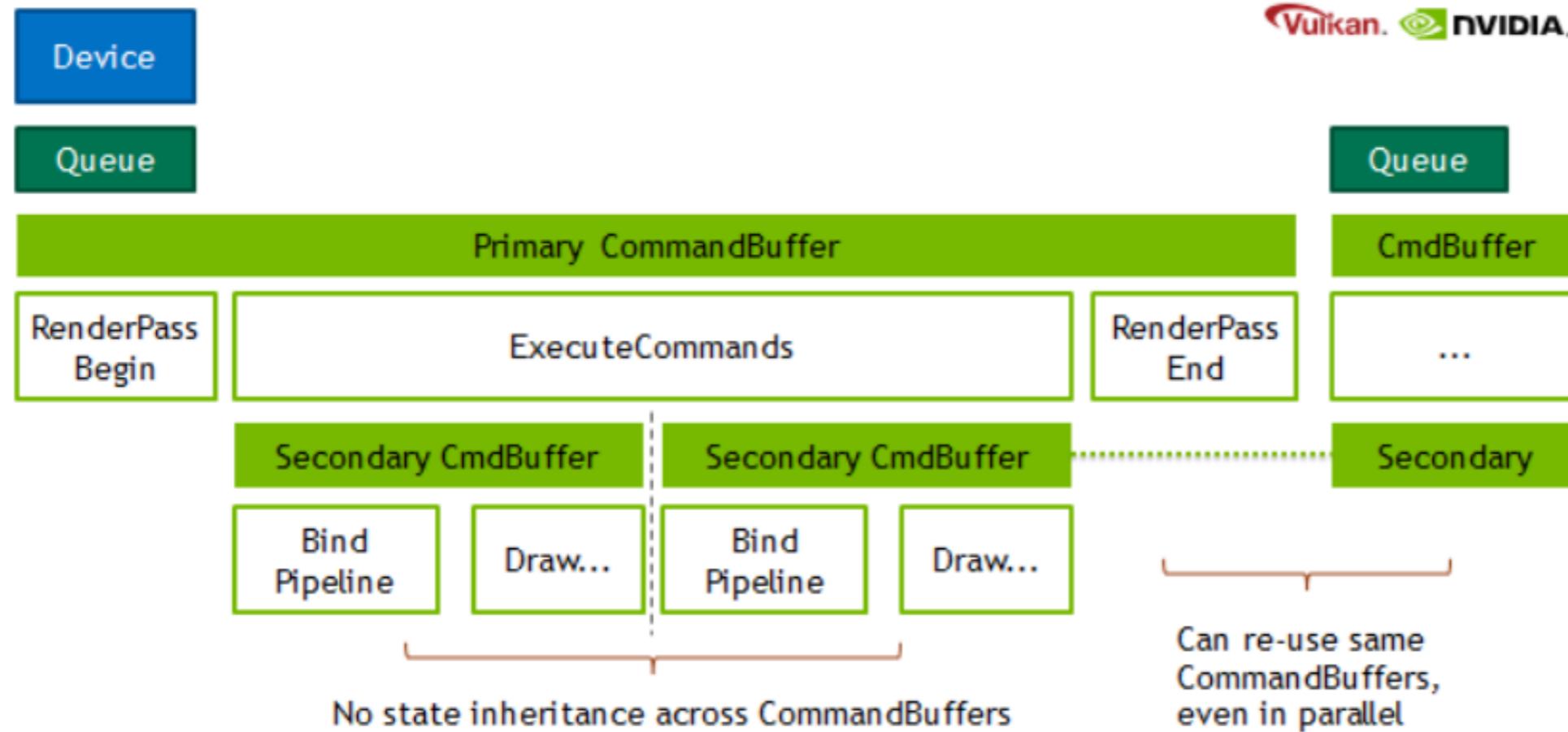
Where OpenGL's state and drawing commands are often immediate, for Vulkan most of these operations are deferred.

The **CommandBuffer** hosts the typical set of commands to setup rendering state and is then submitted to the **Queue** for execution.

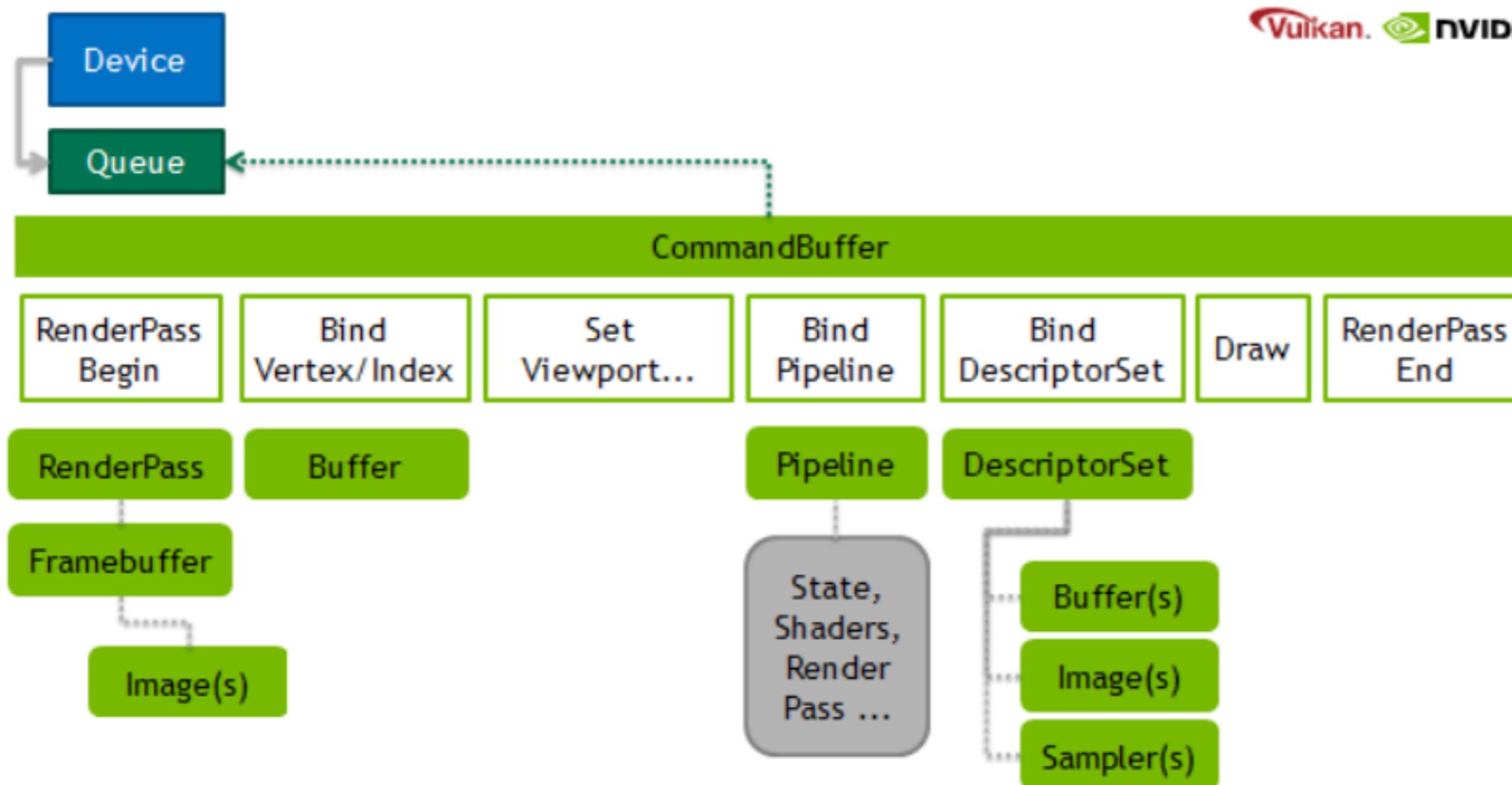


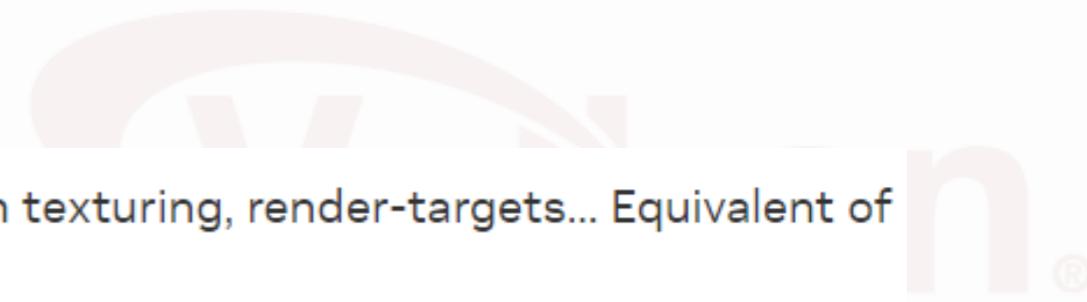
The actual operations within the CommandBuffer should not sound too unfamiliar. A RenderPass is similar to framebuffer-object binding, and a DescriptorSet handles uniform bindings (buffer, texture...), more about those later.

- **Device:** The device is used to query information, and to create most of Vulkan's API objects.
- **Queue:** A device can expose multiple queues. For example, there can be dedicated queue to copying data, or the compute and/or graphics queue. Operations on a single queue are typically processed in-order, but multiple queues can overlap in parallel.
- **CommandBuffer:** Here we record the general commands such as setting state, executing work like drawing from vertex-buffers, dispatching compute grids, copying between buffers... function wise nothing fundamentally different. While there are still costs for building, the submission to the queue will be rather quick.



- **Primary CommandBuffer** always handles RenderPass setup. All the other typical rendering operations can be either directly recorded, or provided by **secondary CommandBuffer**.
- **Secondary CommandBuffer** can encode a subset of commands.





- **Image:** Represents formatted data organized in regular grids used in texturing, render-targets... Equivalent of an OpenGL texture.
- **FrameBuffer:** A set of Image attachments that are being rendered into. It must match the configuration of the RenderPass it is used with.
- **RenderPass:** In principle encodes the format of the framebuffer attachments, what type of clears, whether we do multi-pass effects, pass dependencies... This is one of the bigger new features that Vulkan has to offer which will be subject of a later blog post.
- **Buffer:** Represents raw linear memory used for vertex, index, uniform data... Equivalent to an OpenGL buffer.
- **Pipeline:** Encodes rendering state such as shaders being used, depth-testing, blending operations... All captured into a single monolithic object. Because all important state is provided upfront at the creation time of the object, its later usage can be very quick. OpenGL's internal validation may have to do state-dependent compilation of shaders that at worst could create stuttering at draw-time. With Vulkan you have precise control over when such validation is triggered.
- **DescriptorSet:** A set of bindings for shader inputs. Instead of binding resources individually in OpenGL, Vulkan organizes them in groups. You can re-use such a binding group as well. In a later blog post we will cover the various ways how to provide uniform data to your compute or draw calls.

- **CommandBufferPool**: The CommandBuffers and their content are allocated from these pools.
- **DescriptorPool**: Many DescriptorSets can be allocated from a single pool.
- **Heap**: The device comes with fixed amount of limited heaps, which memory is allocated from.
- **Memory**: Buffers and Images are bound to Memory depending on their requirements and the developers preference. This allows manual sub-allocation of resources from a bigger block of memory or aliasing the memory with different resources.

The pools simplify deletion of many resources that were allocated from them at once and they also ensure allocations can be done lock-free by using per-thread pools. For example one can use a different CommandBufferPool per-frame and create all temporary CommandBuffers from it. After a few frames when all these CommandBuffers have been completed by the GPU, the pool can be reset and new temporaries generated from it.

Memory management also allows for greater control and new use-cases such as aliasing memory. A memory allocation is rather costly and some operating systems also have fixed overhead for how many allocations are active at once. We therefore encourage developers to sub-allocate resources from larger chunks of memory.

Rendering and presentation

- Outline of a frame
- Synchronization
 - Semaphores
 - Fences
 - What to choose?
- Creating the synchronization objects
- Waiting for the previous frame
- Acquiring an image from the swap chain
- Recording the command buffer
- Submitting the command buffer
- Subpass dependencies
- Presentation
- Conclusion



Render Pass

What does a Vulkan Render Pass Provide?

Describes the list attachments the render pass involves

Each attachment can specify

- How the attachment state is initialized (loaded, cleared, dont-care)

- How the attach state is stored (store, or dont-care)

 - Don't-care allows framebuffer intermediates to be discarded

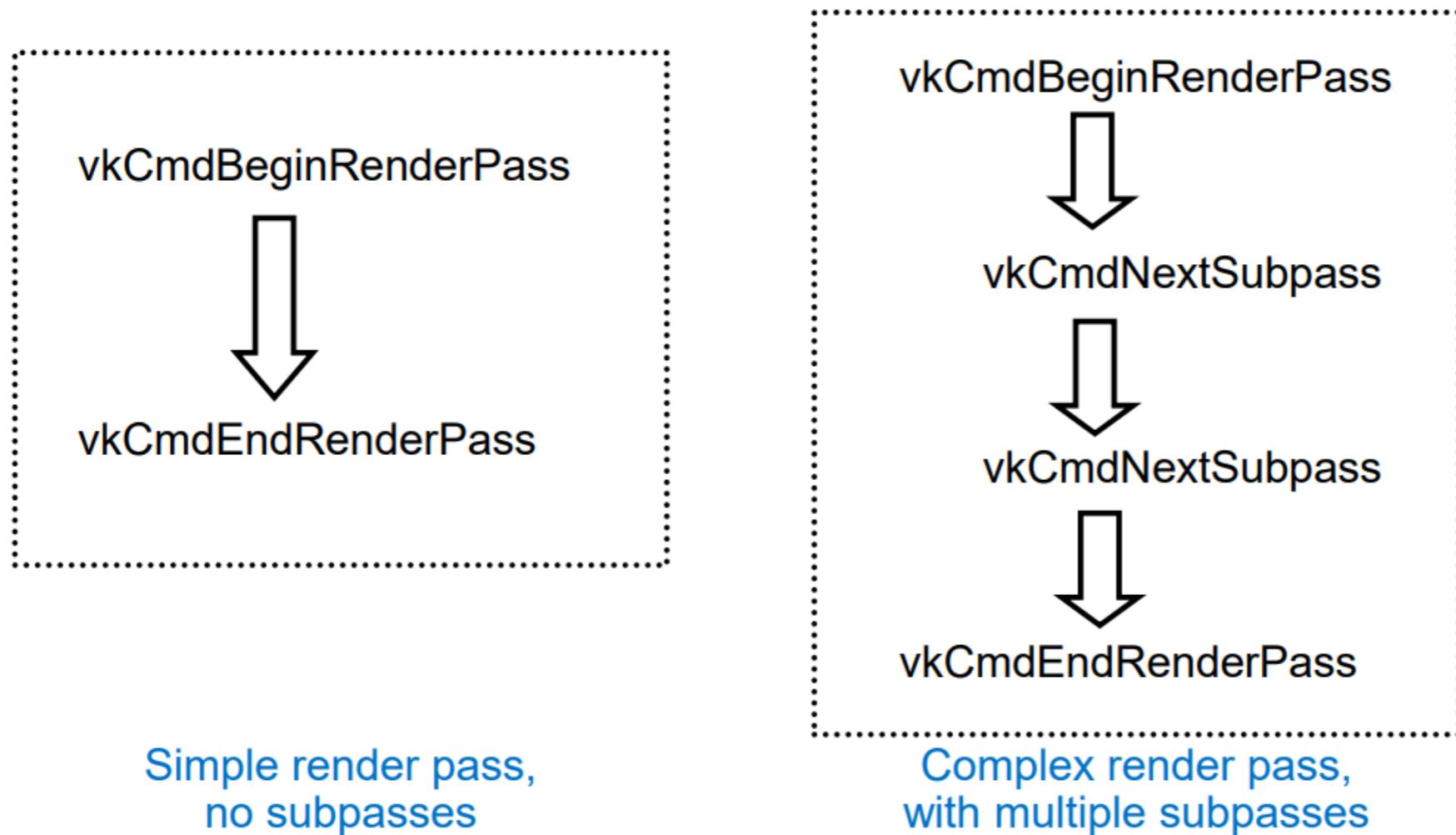
 - E.g. depth buffer not needed after the render pass

- How the layout of attachment could change

Sub-pass dependencies indicate involvement of attachments within a subpass

Render Passes

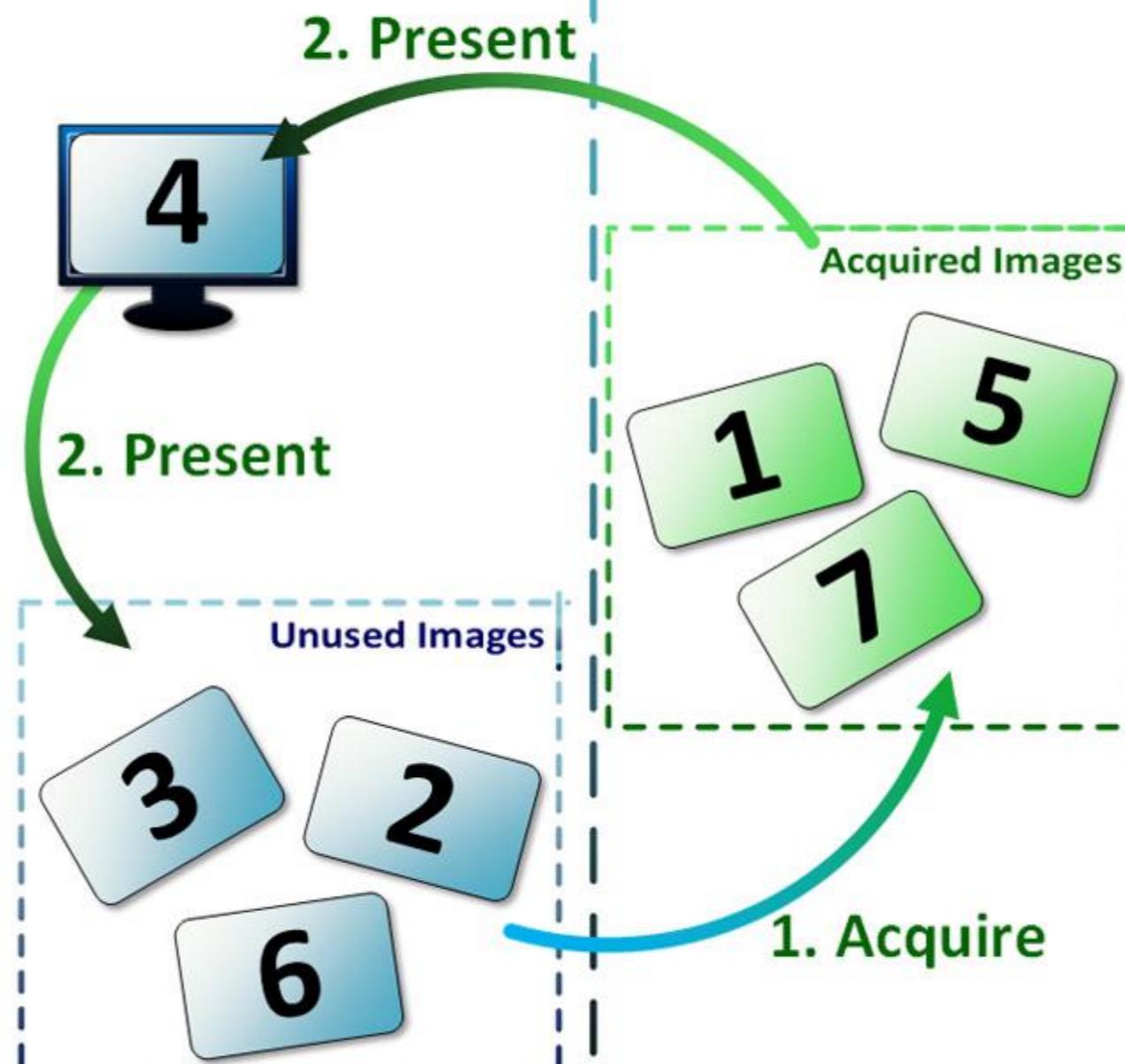
Monolithic or Could Have Sub-passes



*Each subpass
has its own
description and
dependencies*

Presentation
Engine

Application



Synchronization

A core design philosophy in Vulkan is that synchronization of execution on the GPU is explicit. The order of operations is up to us to define using various synchronization primitives which tell the driver the order we want things to run in. This means that many Vulkan API calls which start executing work on the GPU are asynchronous, the functions will return before the operation has finished.

In this chapter there are a number of events that we need to order explicitly because they happen on the GPU, such as:

- Acquire an image from the swap chain
- Execute commands that draw onto the acquired image
- Present that image to the screen for presentation, returning it to the swapchain

Each of these events is set in motion using a single function call, but are all executed asynchronously. The function calls will return before the operations are actually finished and the order of execution is also undefined. That is unfortunate, because each of the operations depends on the previous one finishing. Thus we need to explore which primitives we can use to achieve the desired ordering.

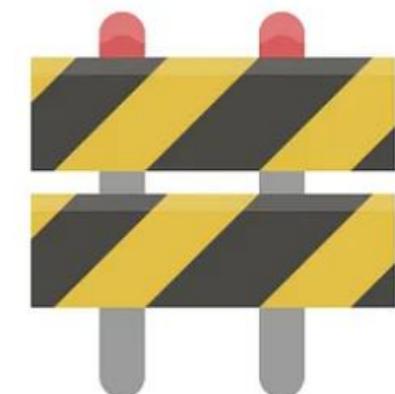
Semaphores

A semaphore is used to add order between queue operations. Queue operations refer to the work we submit to a queue, either in a command buffer or from within a function as we will see later. Examples of queues are the graphics queue and the presentation queue. Semaphores are used both to order work inside the same queue and between different queues.



Fences

A fence has a similar purpose, in that it is used to synchronize execution, but it is for ordering the execution on the CPU, otherwise known as the host. Simply put, if the host needs to know when the GPU has finished something, we use a fence.

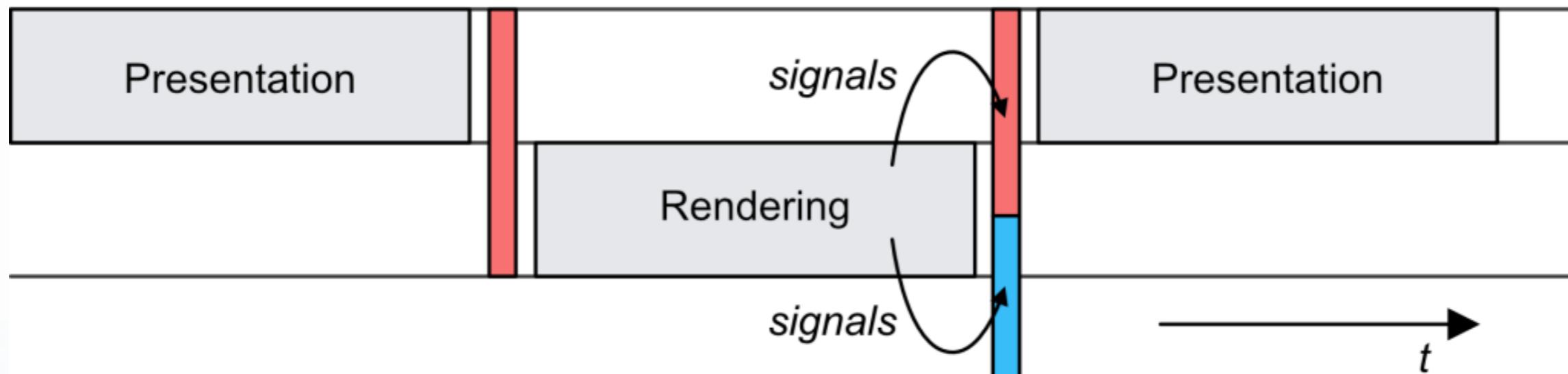


Similar to semaphores, fences are either in a signaled or unsignaled state.

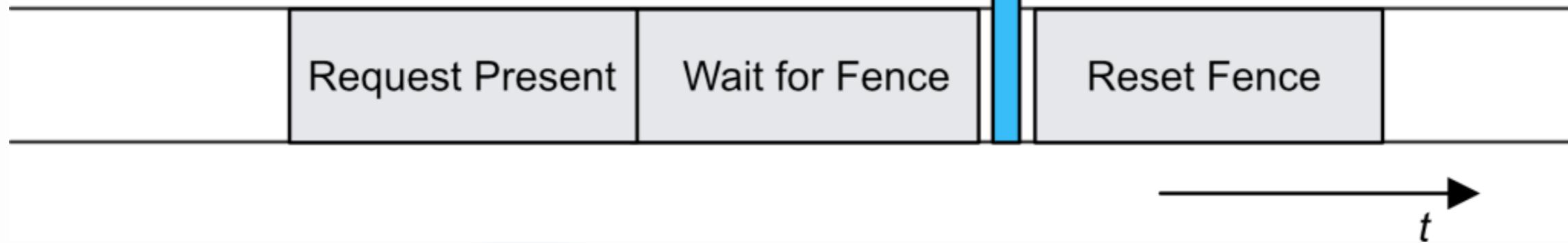
Whenever we submit work to execute, we can attach a fence to that work. When the work is finished, the fence will be signaled. Then we can make the host wait for the fence to be signaled, guaranteeing that the work has finished before the host continues.

Frames in flight

GPU Timeline



CPU Timeline



Creating the synchronization objects

We'll need one semaphore to signal that an image has been acquired from the swapchain and is ready for rendering, another one to signal that rendering has finished and presentation can happen, and a fence to make sure only one frame is rendering at a time.

Create three class members to store these semaphore objects and fence object:

```
VkSemaphore imageAvailableSemaphore;  
VkSemaphore renderFinishedSemaphore;  
VkFence inFlightFence;
```

We Work In-Depth

Waiting for the previous frame

At the start of the frame, we want to wait until the previous frame has finished, so that the command buffer and semaphores are available to use. To do that, we call `vkWaitForFences`:

```
void drawFrame() {
    vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
}
```

The `vkWaitForFences` function takes an array of fences and waits on the host for either any or all of the fences to be signaled before returning. The `VK_TRUE` we pass here indicates that we want to wait for all fences, but in the case of a single one it doesn't matter. This function also has a timeout parameter that we set to the maximum value of a 64 bit unsigned integer, `UINT64_MAX`, which effectively disables the timeout.

After waiting, we need to manually reset the fence to the unsignaled state with the `vkResetFences` call:

```
vkResetFences(device, 1, &inFlightFence);
```

Acquiring an image from the swap chain

The next thing we need to do in the `drawFrame` function is acquire an image from the swap chain. Recall that the swap chain is an extension feature, so we must use a function with the `vk*KHR` naming convention:

```
void drawFrame() {
    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore, VK_NULL_HANDLE, &imageIndex);
}
```

The first two parameters of `vkAcquireNextImageKHR` are the logical device and the swap chain from which we wish to acquire an image. The third parameter specifies a timeout in nanoseconds for an image to become available. Using the maximum value of a 64 bit unsigned integer means we effectively disable the timeout.

The next two parameters specify synchronization objects that are to be signaled when the presentation engine is finished using the image. That's the point in time where we can start drawing to it. It is possible to specify a semaphore, fence or both. We're going to use our `imageAvailableSemaphore` for that purpose here.

The last parameter specifies a variable to output the index of the swap chain image that has become available. The index refers to the `VkImage` in our `swapChainImages` array. We're going to use that index to pick the `VkFrameBuffer`.

Recording the command buffer

With the `imageIndex` specifying the swap chain image to use in hand, we can now record the command buffer. First, we call `vkResetCommandBuffer` on the command buffer to make sure it is able to be recorded.

```
vkResetCommandBuffer(commandBuffer, 0);
```

The second parameter of `vkResetCommandBuffer` is a `VkCommandBufferResetFlagBits` flag. Since we don't want to do anything special, we leave it as 0.

Now call the function `recordCommandBuffer` to record the commands we want.

```
recordCommandBuffer(commandBuffer, imageIndex);
```

With a fully recorded command buffer, we can now submit it.

Submitting the command buffer

Queue submission and synchronization is configured through parameters in the `VkSubmitInfo` structure.

```
VkSubmitInfo submitInfo{};  
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;  
  
VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};  
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};  
submitInfo.waitSemaphoreCount = 1;  
submitInfo.pWaitSemaphores = waitSemaphores;  
submitInfo.pWaitDstStageMask = waitStages;
```

```
void drawFrame() {
    vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
    vkResetFences(device, 1, &inFlightFence);

    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore, VK_NULL_HANDLE, &imageIndex);

    vkResetCommandBuffer(commandBuffer, /*VkCommandBufferResetFlagBits*/ 0);
    recordCommandBuffer(commandBuffer, imageIndex);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
    VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;

    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
    submitInfo.signalSemaphoreCount = 1;
    submitInfo.pSignalSemaphores = signalSemaphores;

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFence) != VK_SUCCESS) {
        throw std::runtime_error("failed to submit draw command buffer!");
    }
}
```

Presentation

The last step of drawing a frame is submitting the result back to the swap chain to have it eventually show up on the screen. Presentation is configured through a `VkPresentInfoKHR` structure at the end of the `drawFrame` function.

```
VkPresentInfoKHR presentInfo{};  
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;  
  
presentInfo.waitSemaphoreCount = 1;  
presentInfo.pWaitSemaphores = signalSemaphores;
```

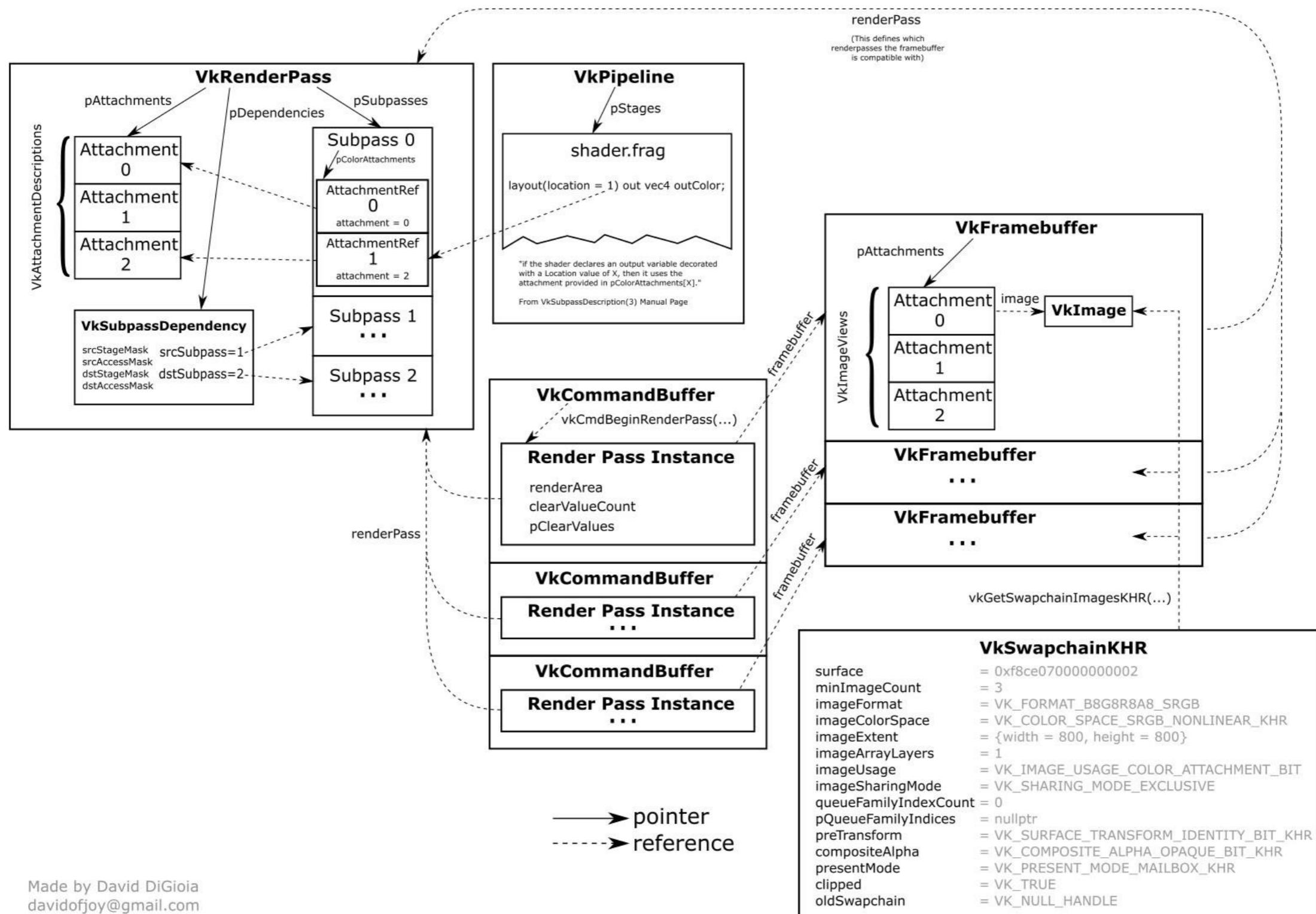
The first two parameters specify which semaphores to wait on before presentation can happen, just like `VkSubmitInfo`. Since we want to wait on the command buffer to finish execution, thus our triangle being drawn, we take the semaphores which will be signalled and wait on them, thus we use `signalSemaphores`.

```
VkSwapchainKHR swapChains[] = {swapChain};  
presentInfo.swapchainCount = 1;  
presentInfo.pSwapchains = swapChains;  
presentInfo.pImageIndices = &imageIndex;
```

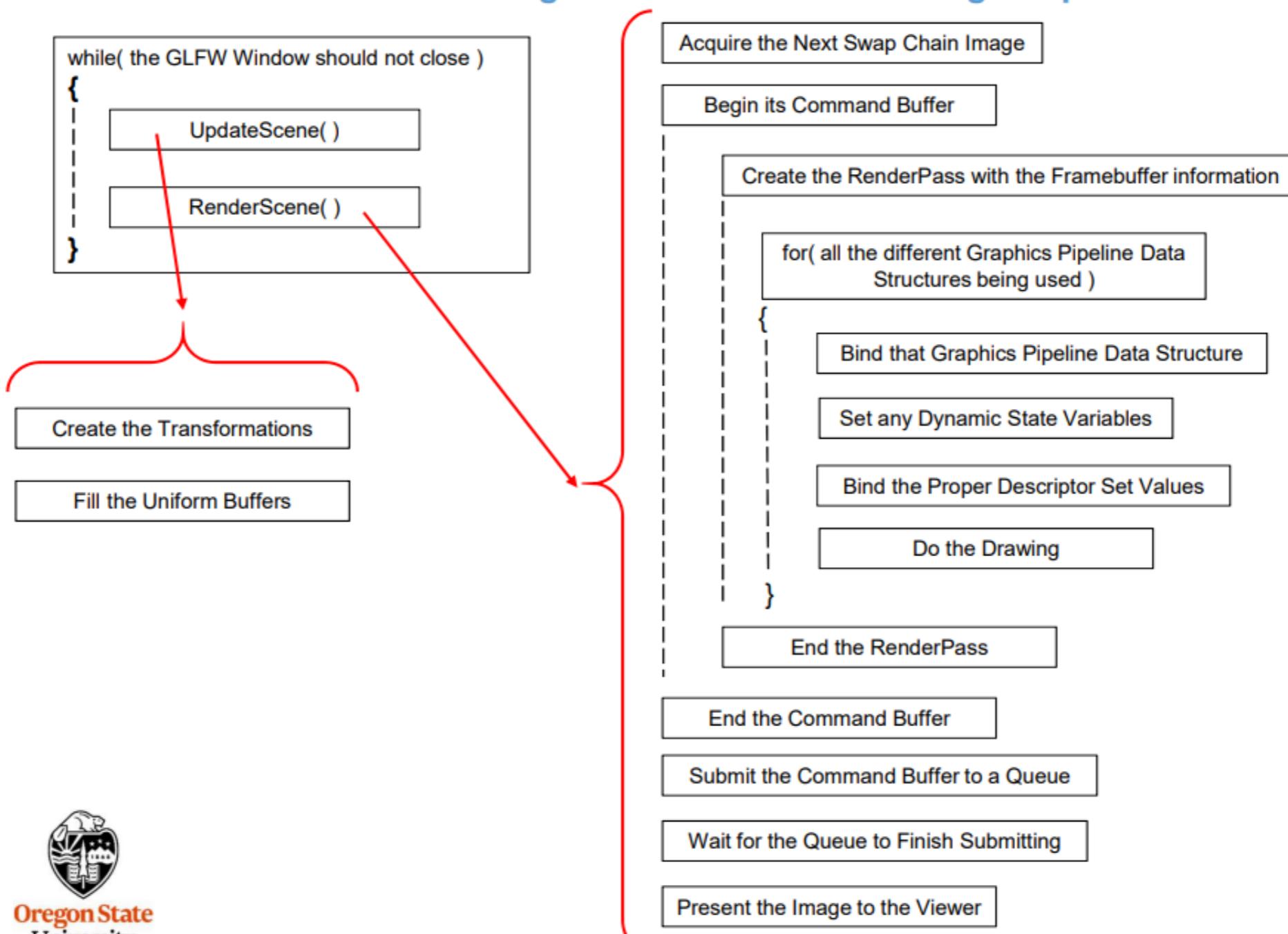
The next two parameters specify the swap chains to present images to and the index of the image for each swap chain. This will almost always be a single one.

```
presentInfo.pResults = nullptr; // Optional  
  
vkQueuePresentKHR(presentQueue, &presentInfo);
```

```
VkPresentInfoKHR presentInfo{};  
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;  
  
presentInfo.waitSemaphoreCount = 1;  
presentInfo.pWaitSemaphores = signalSemaphores;  
  
VkSwapchainKHR swapChains[] = {swapChain};  
presentInfo.swapchainCount = 1;  
presentInfo.pSwapchains = swapChains;  
  
presentInfo.pImageIndices = &imageIndex;  
  
vkQueuePresentKHR(presentQueue, &presentInfo);  
}
```



Vulkan Program Flow – the Rendering Loop



```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffer();
    createSyncObjects();
}

void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }

    vkDeviceWaitIdle(device);
}
```



DAY 2 - Agenda

- Recap of Hello Triangle
- Swap Chain and Presentation
- Synchronization

<BREAK>

- Introduction to Texturing
- Textures, Images, Samplers
- Mipmapping
- Layouts and Transitions
- Synchronization
- Hello Texture Example
- Vertex and Uniform Buffers
- GLM
- Q&A, Exercises



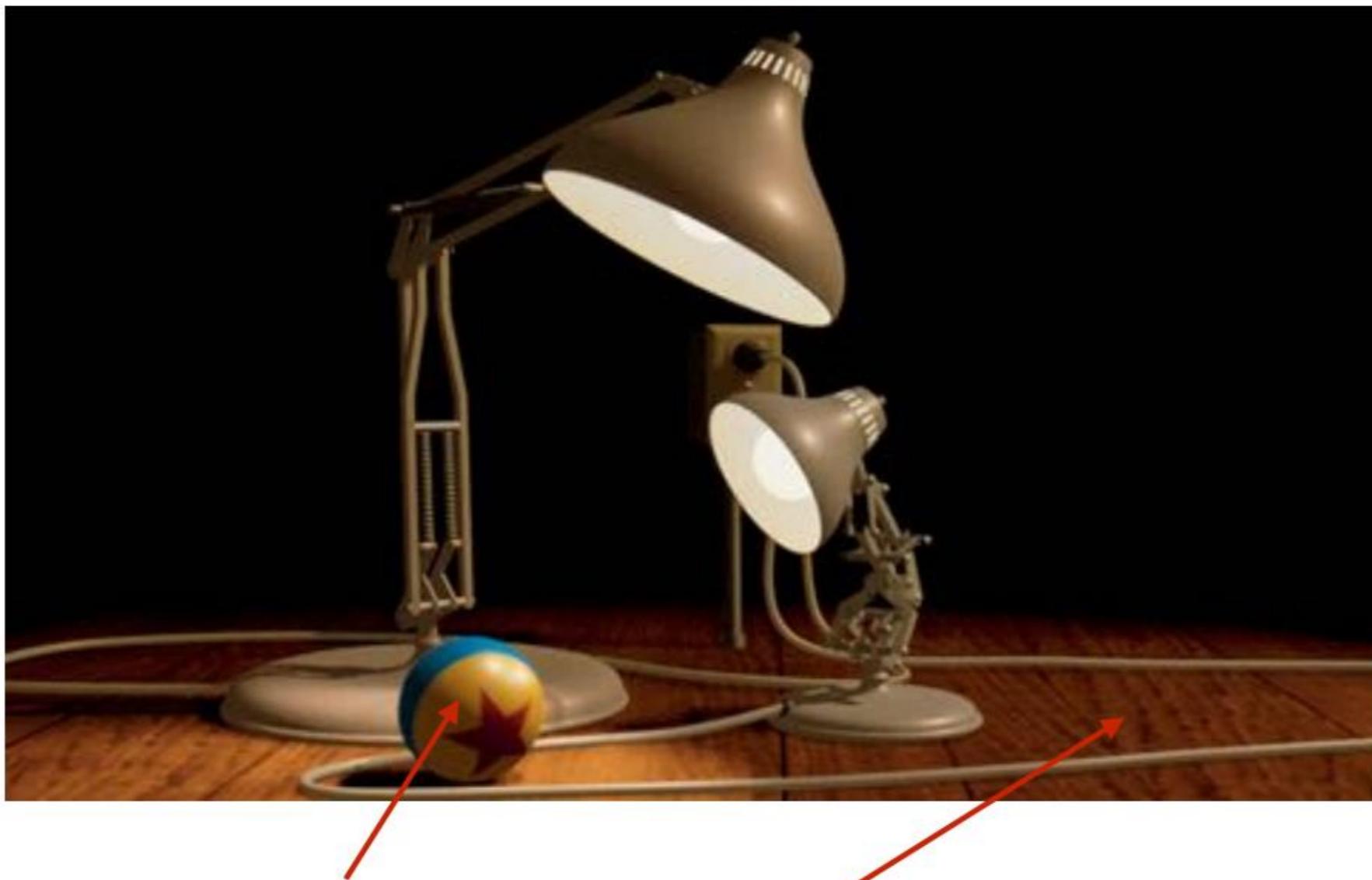
Texture Mapping



an.[®]

Many uses of texture mapping

Define variation in surface reflectance



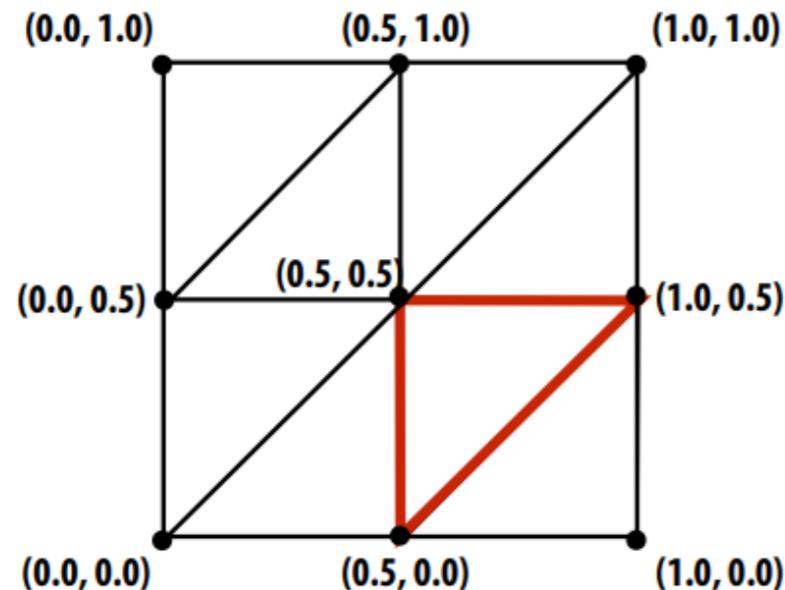
Pattern on ball

Wood grain on floor

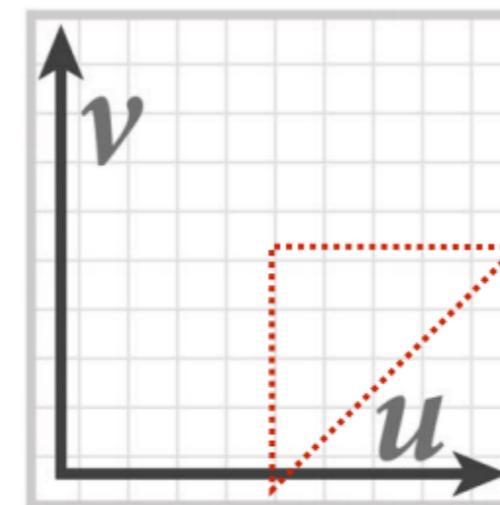
Texture coordinates

- “Texture coordinates” define a mapping from surface coordinates to points in texture domain
- Often defined by linearly interpolating texture coordinates at triangle vertices

Suppose each cube face is split into eight triangles, with texture coordinates (u, v) at each vertex

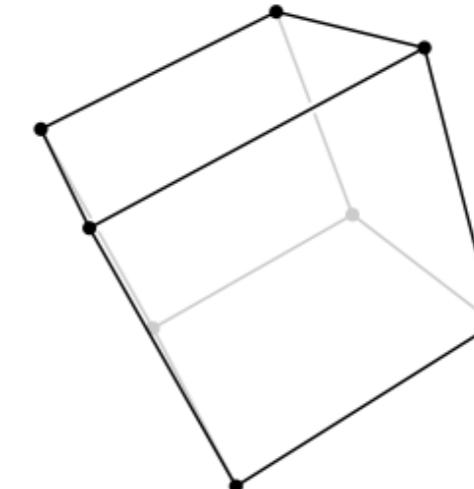


A texture on the $[0,1]^2$ domain can be specified by a 2048x2048 image

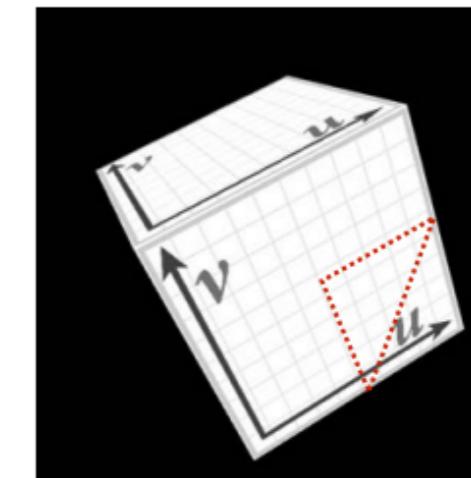


(location of highlighted triangle in texture space shown in red)

example: texture this cube



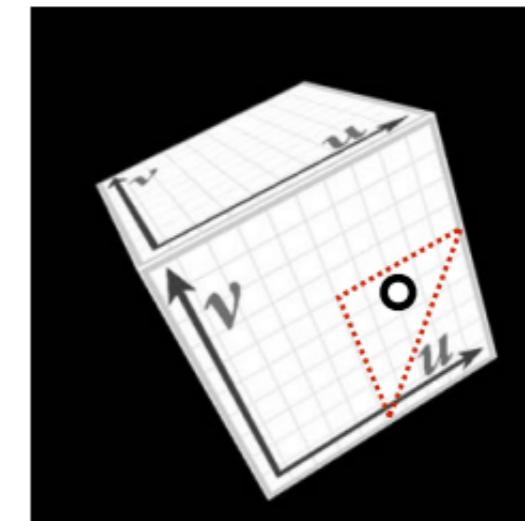
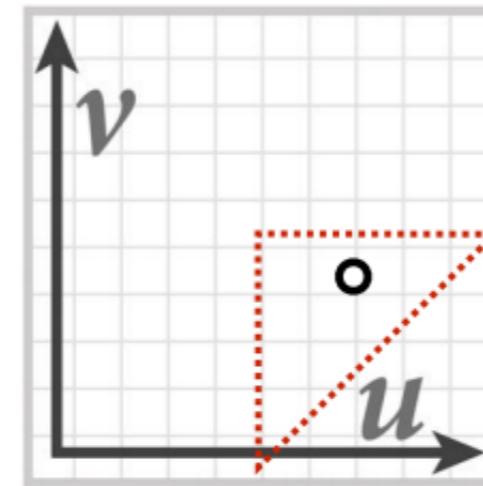
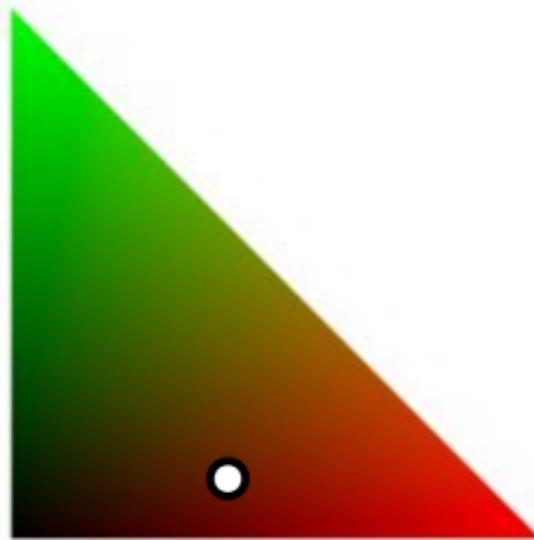
Linearly interpolating texture coordinates & “looking up” color in texture gives this image:



Texture Sampling 101

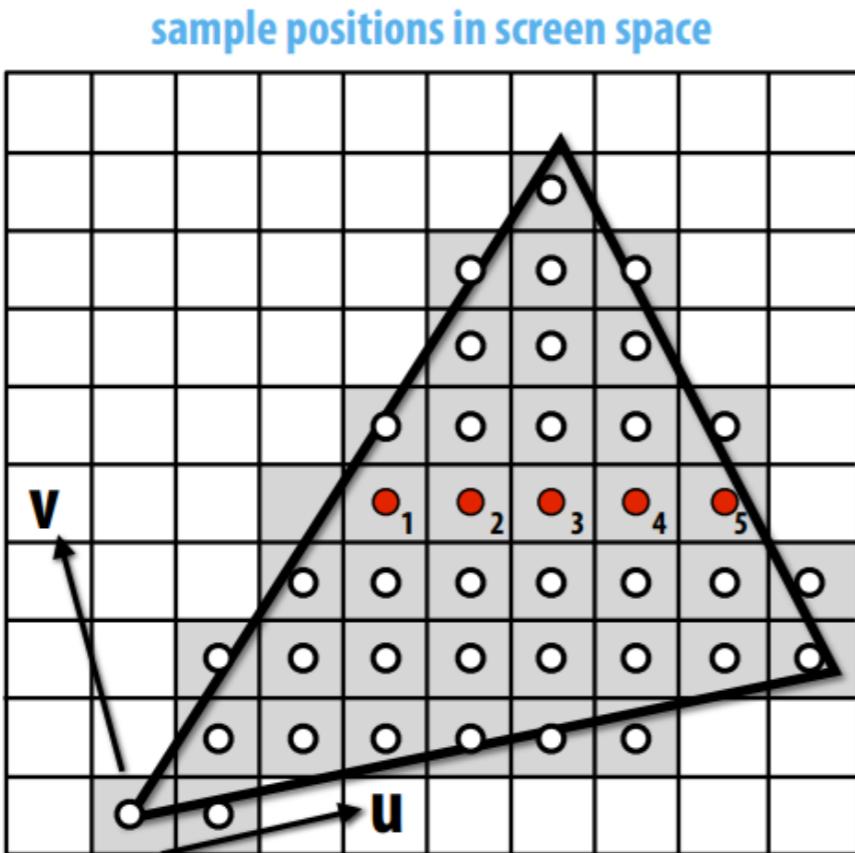
■ Basic algorithm for texture mapping:

- for each pixel in the rasterized image:
 - interpolate (u, v) coordinates across triangle
 - sample (evaluate) texture at interpolated (u, v)
 - set color of fragment to sampled texture value

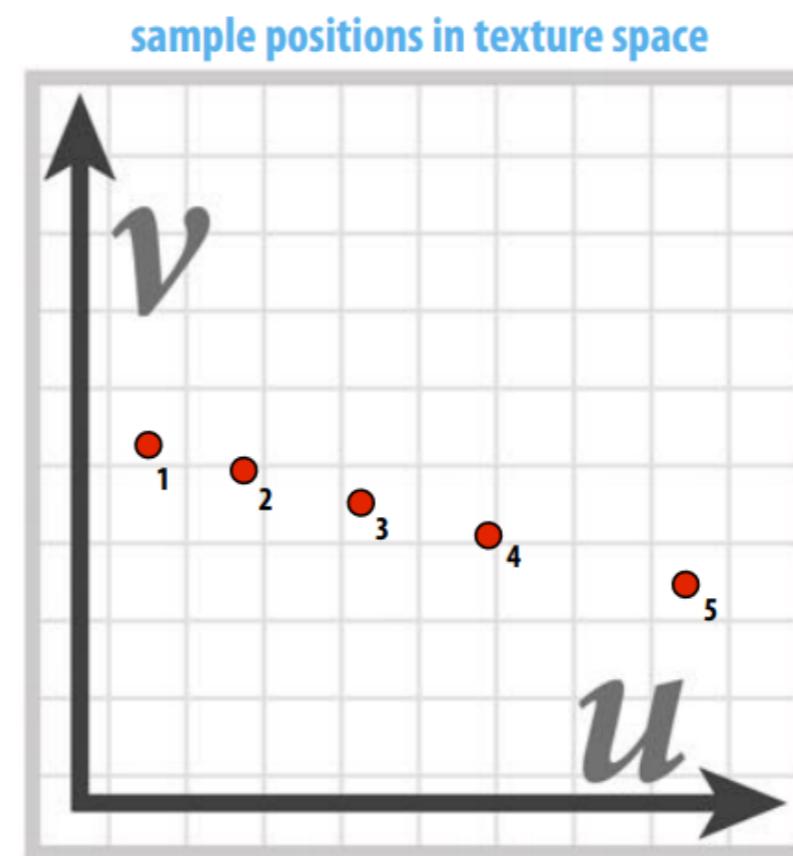


Visualizing texture samples

Since triangles are projected from 3D to 2D, pixels in screen space will correspond to regions of varying size & location in texture



Sample positions are uniformly distributed in screen space
(rasterizer samples triangle's appearance at these locations)

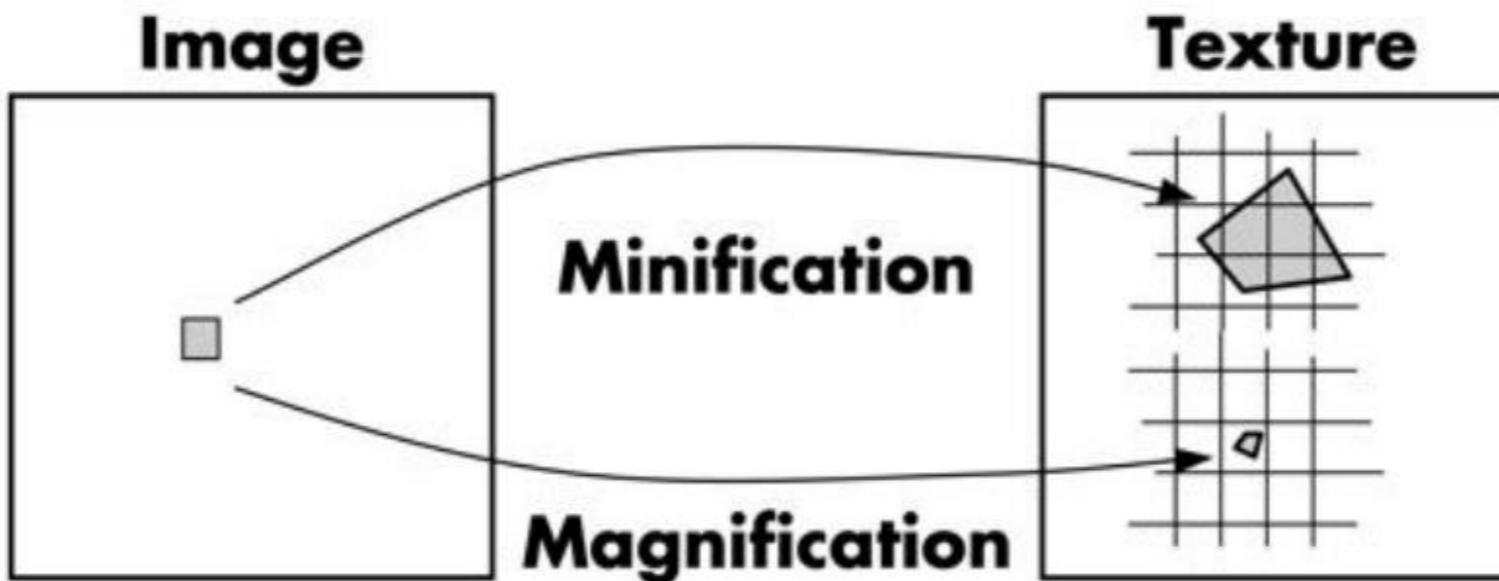


Sample positions in texture space are not uniform
(texture function is sampled at these locations)

Irregular sampling pattern makes it hard to avoid aliasing!

Magnification vs. Minification

an.[®]



■ Magnification (easier):

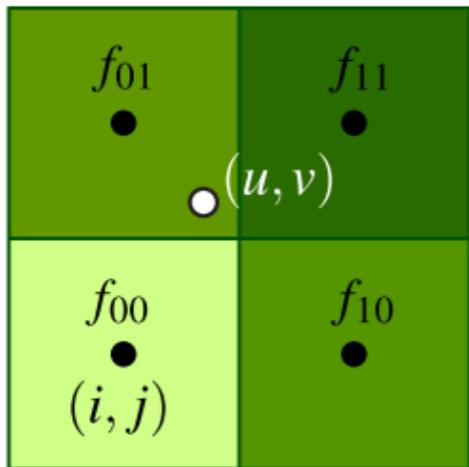
- Example: camera is very close to scene object
- Single screen pixel maps to tiny region of texture
- Can just interpolate value at screen pixel center

■ Minification (harder):

- Example: scene object is very far away
- Single screen pixel maps to large region of texture
- Need to compute average texture value over pixel to avoid aliasing

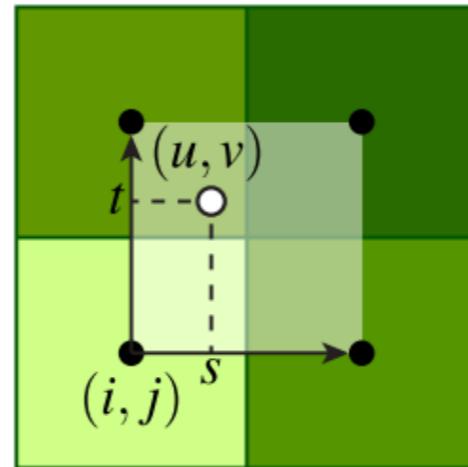
Bilinear interpolation (magnification)

How can we “look up” a texture value at a non-integer location (u, v) ?



$$i = \lfloor u - \frac{1}{2} \rfloor$$

$$j = \lfloor v - \frac{1}{2} \rfloor$$



$$s = u - (i + \frac{1}{2}) \in [0, 1]$$

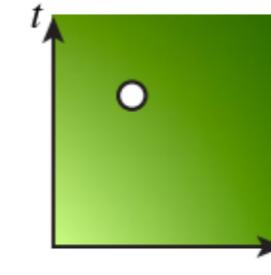
$$t = v - (j + \frac{1}{2}) \in [0, 1]$$

linear (each row)

$$(1 - s)f_{01} + sf_{11}$$

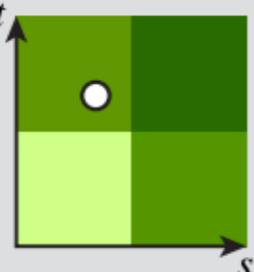
$$(1 - s)f_{00} + sf_{10}$$

bilinear



$$(1 - t)((1 - s)f_{00} + sf_{10}) \\ + t((1 - s)f_{01} + sf_{11})$$

nearest neighbor



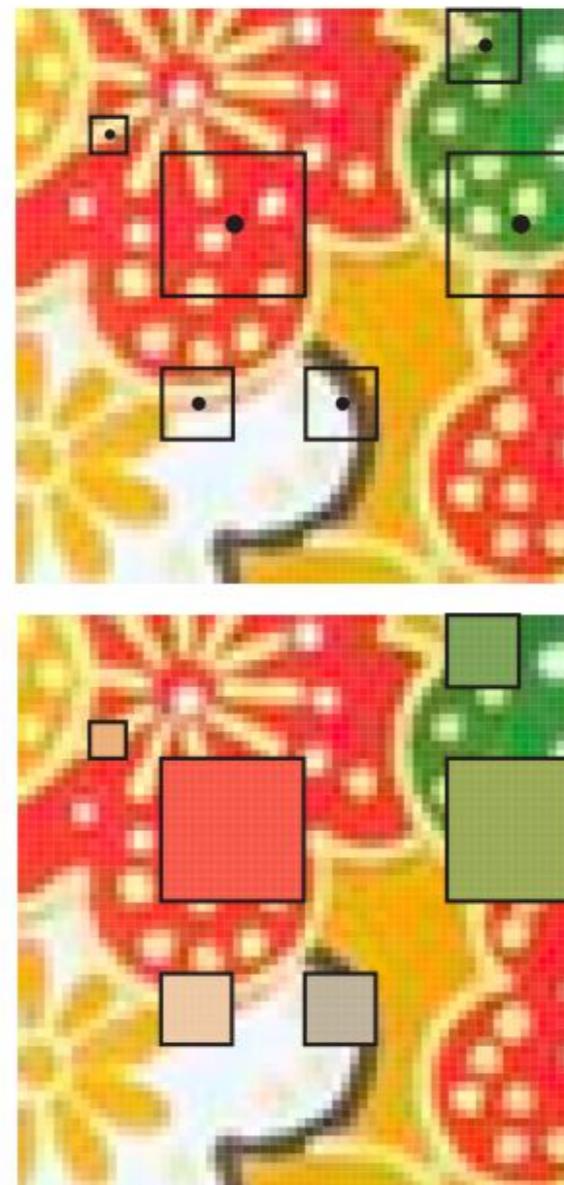
fast but ugly:

just grab value of nearest
“texel” (texture pixel)

**Q: What happens if we
interpolate vertically first?**

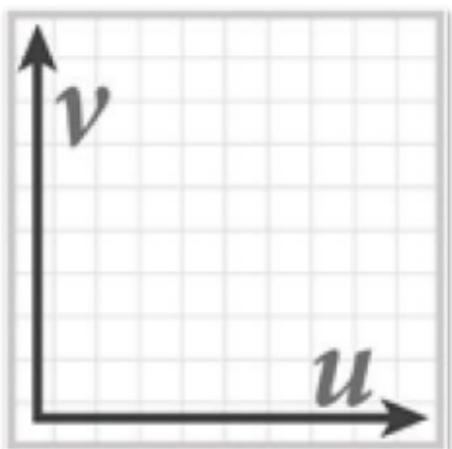
Texture prefiltering — basic idea

- Texture aliasing often occurs because a single pixel on the **screen** covers many pixels of the **texture**
- If we just grab the texture value at the center of the pixel, we get aliasing (get a “random” color that changes if the sample moves even very slightly)
- Ideally, would use the average texture value—but this is expensive to compute
- Instead, we can pre-compute the averages (once) and just look up these averages (many times) at run-time

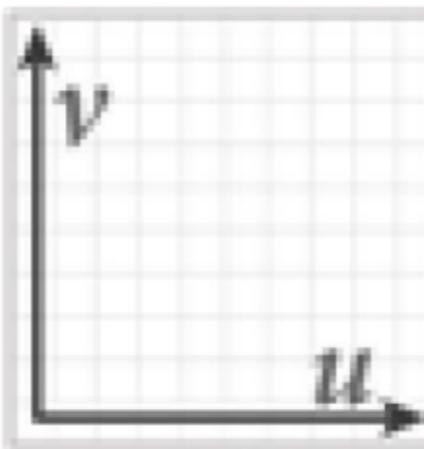


But which averages should we store? Can't precompute them all!

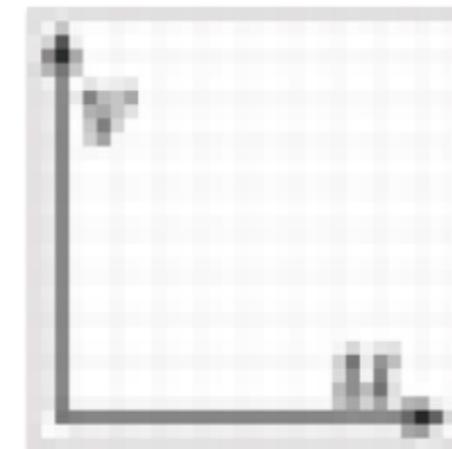
MIP map (L. Williams 83)



Level 0 = 128x128



Level 1 = 64x64



Level 2 = 32x32



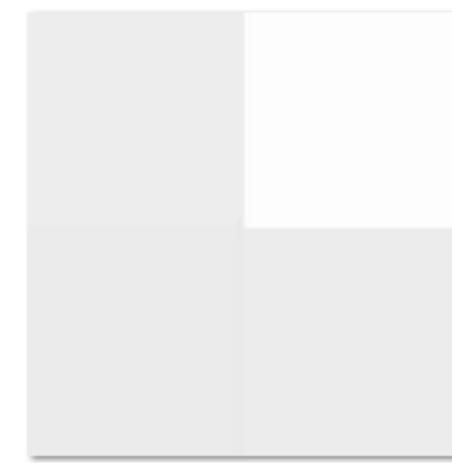
Level 3 = 16x16



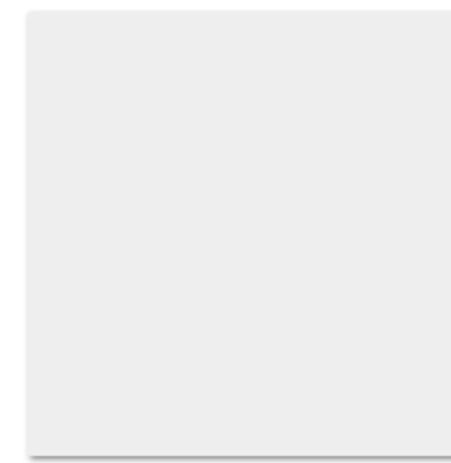
Level 4 = 8x8



Level 5 = 4x4



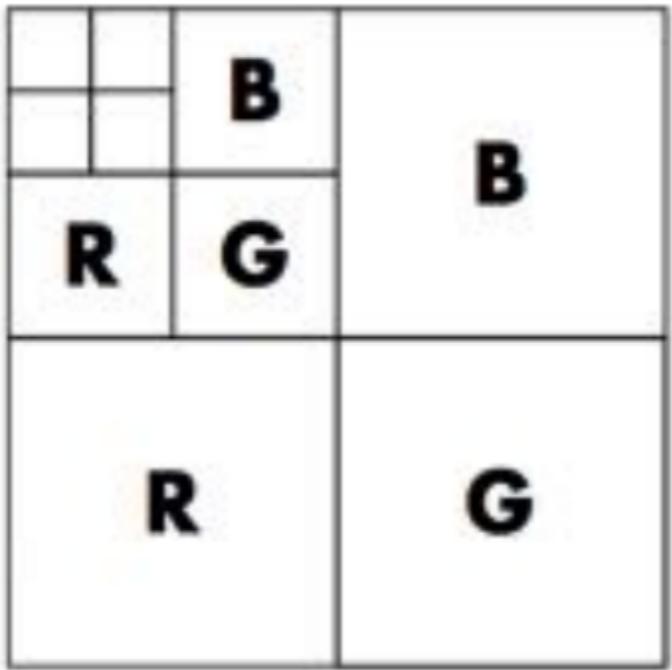
Level 6 = 2x2



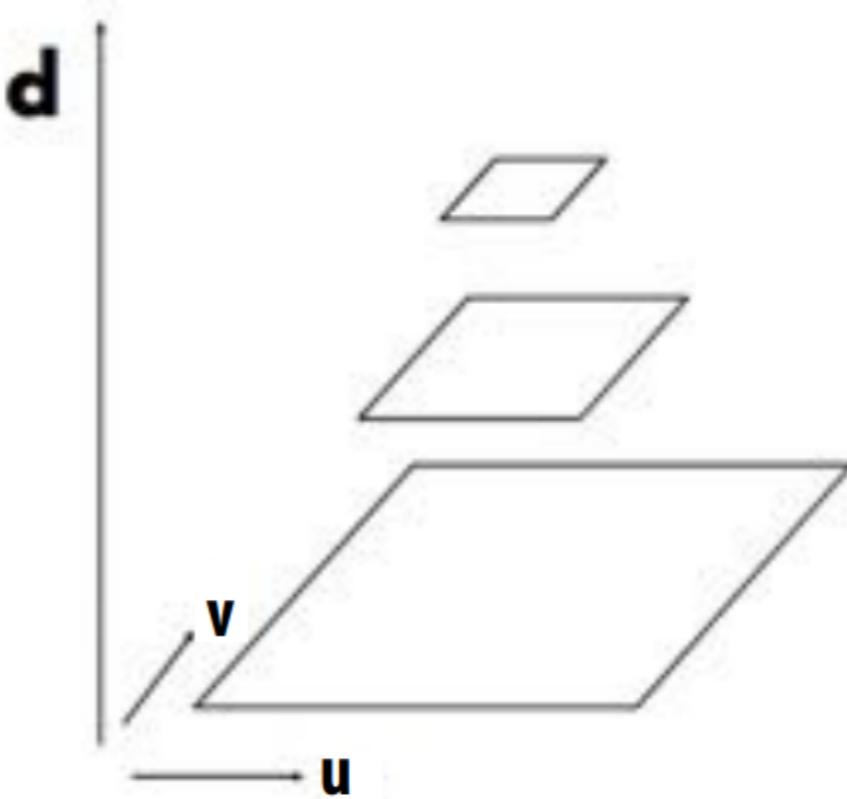
Level 7 = 1x1

- Rough idea: store prefiltered image at “every possible scale”
- Texels at higher levels store average of texture over a region of texture space (downsampled)
- Later: look up a single pixel from MIP map of appropriate size

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout

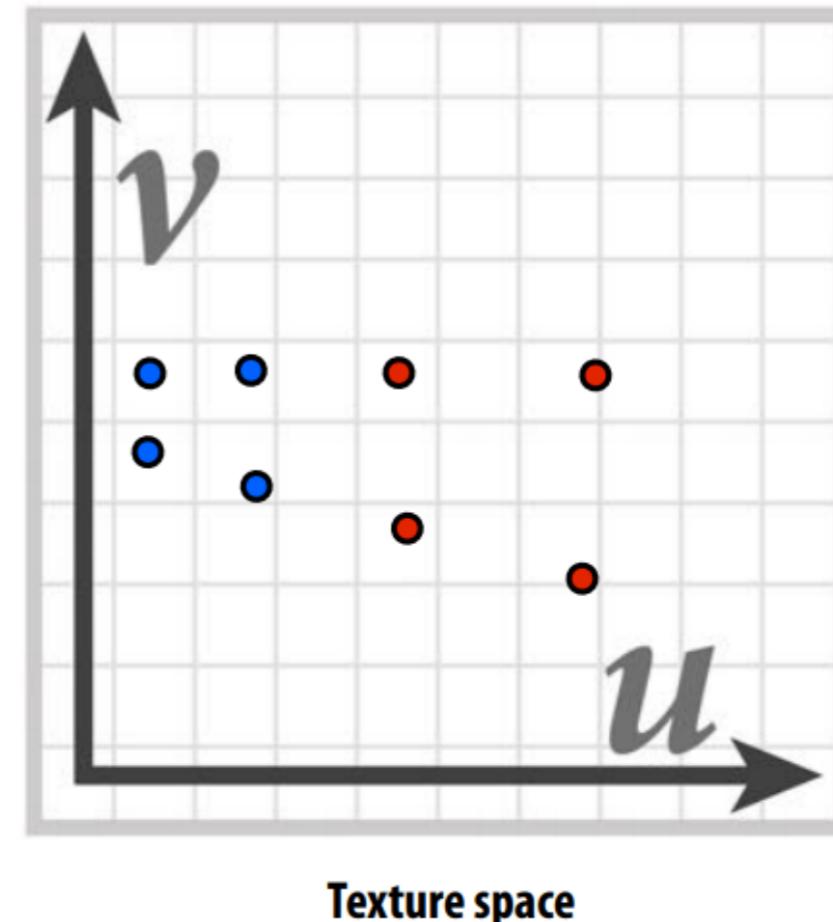
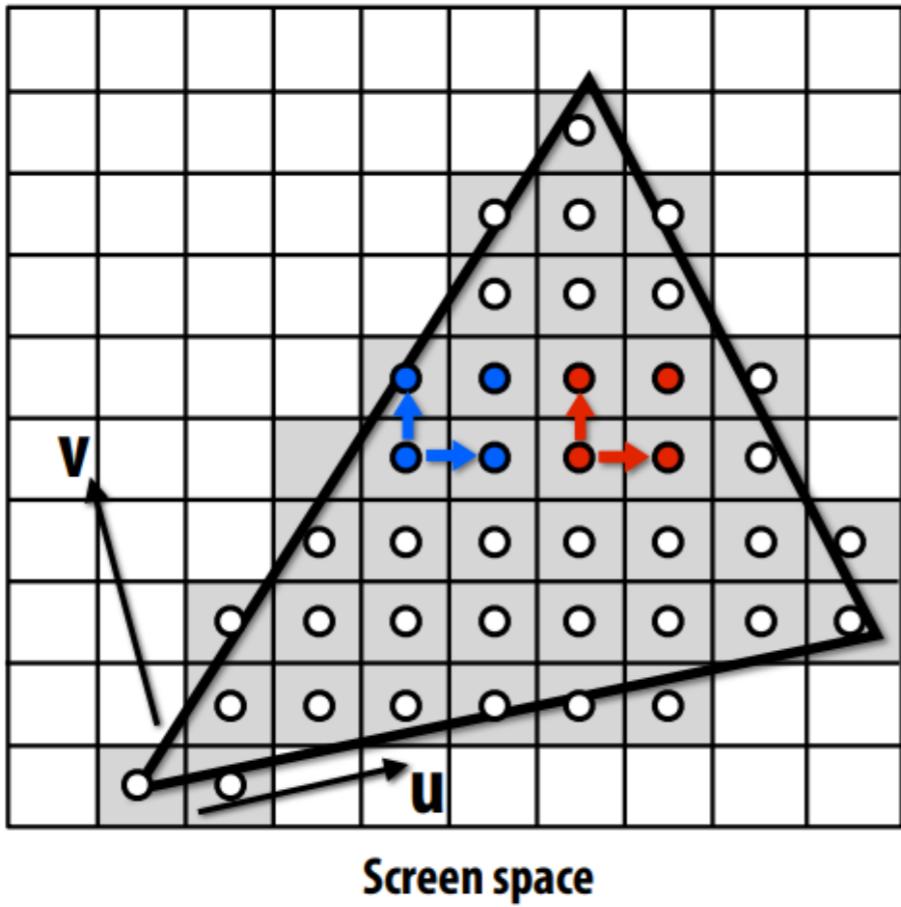


"Mip hierarchy"
level = d

Q: What's the storage overhead of a mipmap?

Computing MIP Map Level

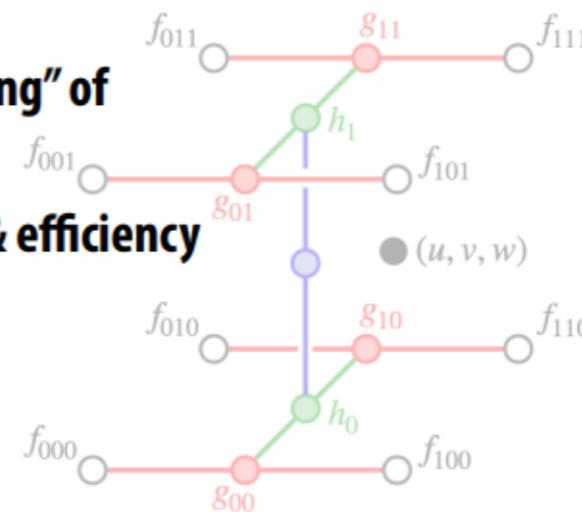
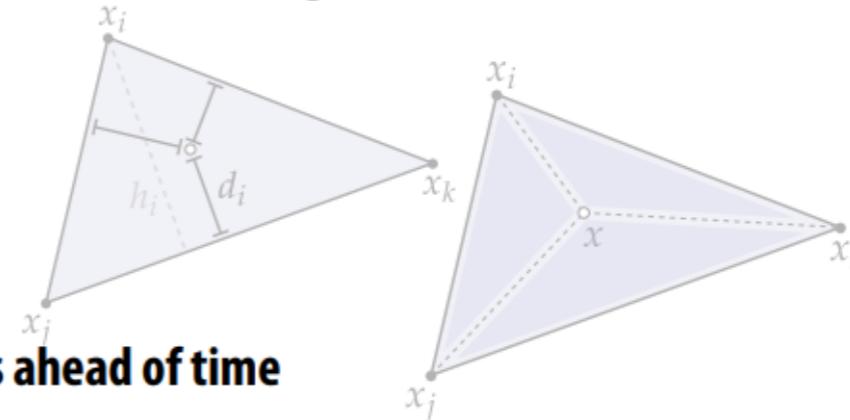
Even within a single triangle, may want to sample from different MIP map levels:



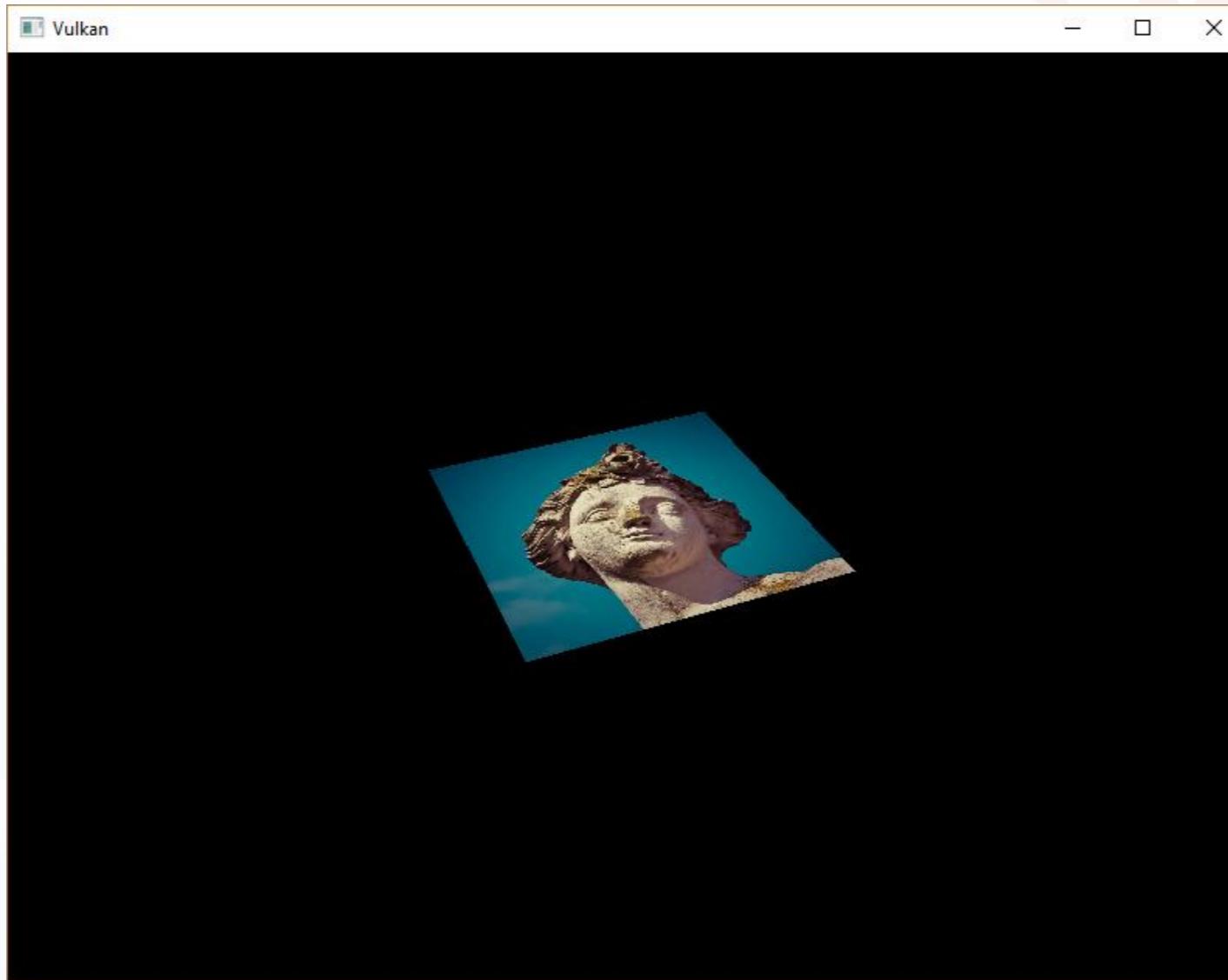
Q: Which pixel should sample from a coarser MIP map level: the blue one, or the red one?

Texture Mapping—Summary

- Once we have 2D primitives, can interpolate attributes across vertices using **barycentric coordinates**
- Important example: **texture coordinates**, used to copy pieces of a 2D image onto a 3D surface
- Careful **texture filtering** is needed to avoid aliasing
 - Key idea:** what's the average color covered by a pixel?
 - For **magnification**, can just do a **bilinear** lookup
 - For **minification**, use **prefiltering** to compute averages ahead of time
 - a **MIP map** stores averages at different levels
 - blend between levels using **trilinear filtering**
 - At grazing angles, **anisotropic filtering** needed to deal w/ "stretching" of samples
 - In general, **no perfect solution to aliasing!** Try to balance quality & efficiency



Hello Texture – VulkanTutorial\\26_texture_mapping



https://vulkan-tutorial.com/Texture_mapping/Images



Introduction

The geometry has been colored using per-vertex colors so far, which is a rather limited approach. In this part of the tutorial we're going to implement texture mapping to make the geometry look more interesting. This will also allow us to load and draw basic 3D models in a future chapter.

Adding a texture to our application will involve the following steps:

- Create an image object backed by device memory
- Fill it with pixels from an image file
- Create an image sampler
- Add a combined image sampler descriptor to sample colors from the texture

Vertex Shader



```
#version 450

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Fragment Shader



A combined image sampler descriptor is represented in GLSL by a sampler uniform. Add a reference to it in the fragment shader:

```
layout(binding = 1) uniform sampler2D texSampler;
```

There are equivalent `sampler1D` and `sampler3D` types for other types of images. Make sure to use the correct binding here.

```
void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

Textures are sampled using the built-in `texture` function. It takes a `sampler` and coordinate as arguments. The sampler automatically takes care of the filtering and transformations in the background. You should now see the texture on the square when you run the application:

IMAGES AND IMAGEVIEW

Images represent all kind of ‘pixel-like’ arrays

Textures: Color or Depth-Stencil

Heap

Render targets : Color and Depth-Stencil

Even Compute data

Memory
Shader Load/Store (imgLoadStore)

ImageView required to expose Images properly when specific format required

For Shaders

For Framebuffers

Sampler

Cmd.Buffer Pool

Command-buffer

Barrier synchronization

Begin Render-Pass

Bind Graphics-pipeline

Set misc. dynamic states

Bind Vertex/Idx Buffer(s)

Update Buffer

Bind Descriptor-Set(s)

Draw...

Execute Commands

End Render-Pass

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

...

Device

Queue

HOW DOES IT LOOK ?

Simple texture creation

Way more complex than OpenGL !

- Load image
- Create an Image (1D/2D/3D/Cube...)
- Create an Image-View
- Aggregate layers/mipmap layers info (offsets, sizes) in a structure (VkBufferImageCopy)
- Aggregate layers & mipmap data to contiguous memory

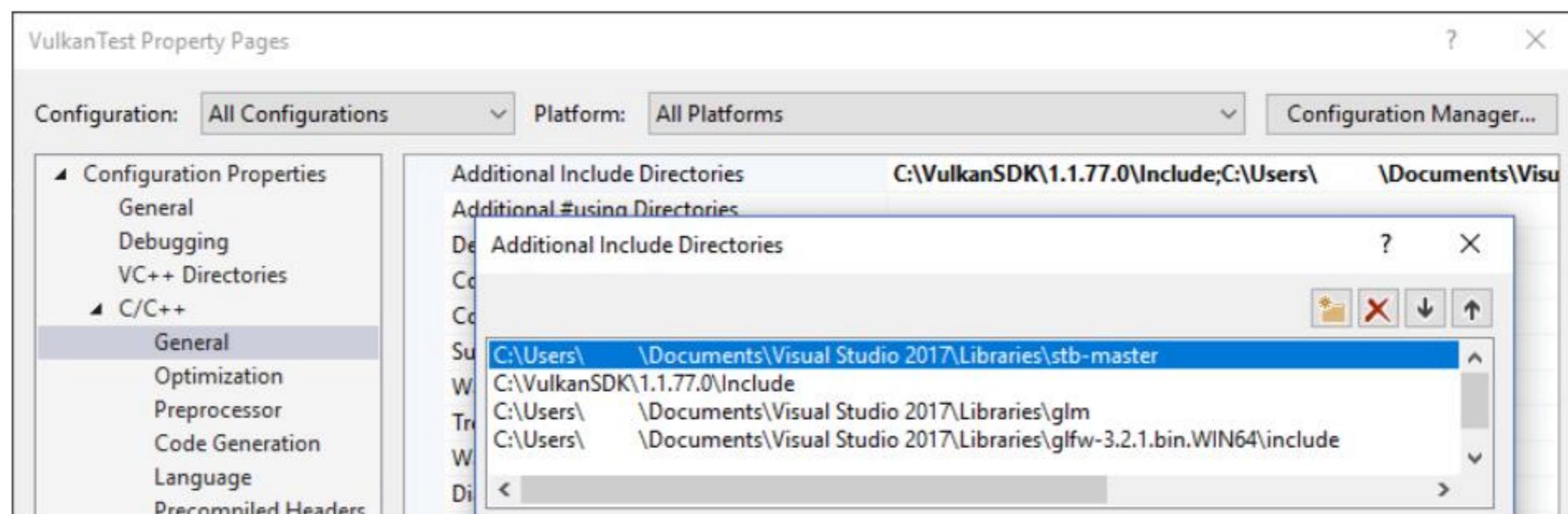
- Create staging buffer + bind memory + copy data in it
- Use command-buffer to copy to the image: layers and mipmaps
 - Layout transition of image for copy
 - `vkCmdCopyBufferToImage`
 - Layout transition of image for use by shader
- Enqueue command buffer and execute

Image library

There are many libraries available for loading images, and you can even write your own code to load simple formats like BMP and PPM. In this tutorial we'll be using the `stb_image` library from the [stb collection](#). The advantage of it is that all of the code is in a single file, so it doesn't require any tricky build configuration. Download `stb_image.h` and store it in a convenient location, like the directory where you saved GLFW and GLM. Add the location to your include path.

Visual Studio

Add the directory with `stb_image.h` in it to the `Additional Include Directories` paths.



Loading the Image



```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }
}
```

The `stbi_load` function takes the file path and number of channels to load as arguments. The `STBI_rgb_alpha` value forces the image to be loaded with an alpha channel, even if it doesn't have one, which is nice for consistency with other textures in the future. The middle three parameters are outputs for the width, height and actual number of channels in the image. The pointer that is returned is the first element in an array of pixel values. The pixels are laid out row by row with 4 bytes per pixel in the case of `STBI_rgb_alpha` for a total of `texWidth * texHeight * 4` values.

Staging buffer

We're now going to create a buffer in host visible memory so that we can use `vkMapMemory` and copy the pixels to it. Add variables for this temporary buffer to the `createTextureImage` function:

```
VkBuffer stagingBuffer;  
VkDeviceMemory stagingBufferMemory;
```

The buffer should be in host visible memory so that we can map it and it should be usable as a transfer source so that we can copy it to an image later on:

```
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);
```

We can then directly copy the pixel values that we got from the image loading library to the buffer:

```
void* data;  
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);  
memcpy(data, pixels, static_cast<size_t>(imageSize));  
vkUnmapMemory(device, stagingBufferMemory);
```

Texture Image

Although we could set up the shader to access the pixel values in the buffer, it's better to use image objects in Vulkan for this purpose. Image objects will make it easier and faster to retrieve colors by allowing us to use 2D coordinates, for one. Pixels within an image object are known as texels and we'll use that name from this point on. Add the following new class members:

```
VkImage textureImage;  
VkDeviceMemory textureImageMemory;
```

The parameters for an image are specified in a [VkImageCreateInfo](#) struct:

```
VkImageCreateInfo imageInfo{};  
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;  
imageInfo.imageType = VK_IMAGE_TYPE_2D;  
imageInfo.extent.width = static_cast<uint32_t>(texWidth);  
imageInfo.extent.height = static_cast<uint32_t>(texHeight);  
imageInfo.extent.depth = 1;  
imageInfo.mipLevels = 1;  
imageInfo.arrayLayers = 1;
```

```
void createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling, VkImageUsageFlags usage, VkMemoryPropertyFlags properties, VkImage& image, VkDeviceMemory& imageMemory) {
    VkImageCreateInfo imageInfo{};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = usage;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateImage(device, &imageInfo, nullptr, &image) != VK_SUCCESS) {
        throw std::runtime_error("failed to create image!");
    }
}
```



```
VkMemoryRequirements memRequirements;
vkGetImageMemoryRequirements(device, image, &memRequirements);

VkMemoryAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

if (vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate image memory!");
}

vkBindImageMemory(device, image, imageMemory, 0);
}
```

```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
    memcpy(data, pixels, static_cast<size_t>(imageSize));
    vkUnmapMemory(device, stagingBufferMemory);

    stbi_image_free(pixels);

    createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory);
}
```

Various Vulkan Objects “inside” a Texture

No such thing as “`VkTexture`”

Instead

Texture state in Vulkan = `VkDescriptorImageInfo`

Combines: `VkSampler` + `VkImageView` + `VkImageLayout`

Sampling state + Image state + Current image layout

*OpenGL textures are opaque,
so lacks an exposed image layout*

Texture binding in Vulkan = part of `VkDescriptorSetLayoutBinding`

Contained with `VkDescriptorSetLayout`

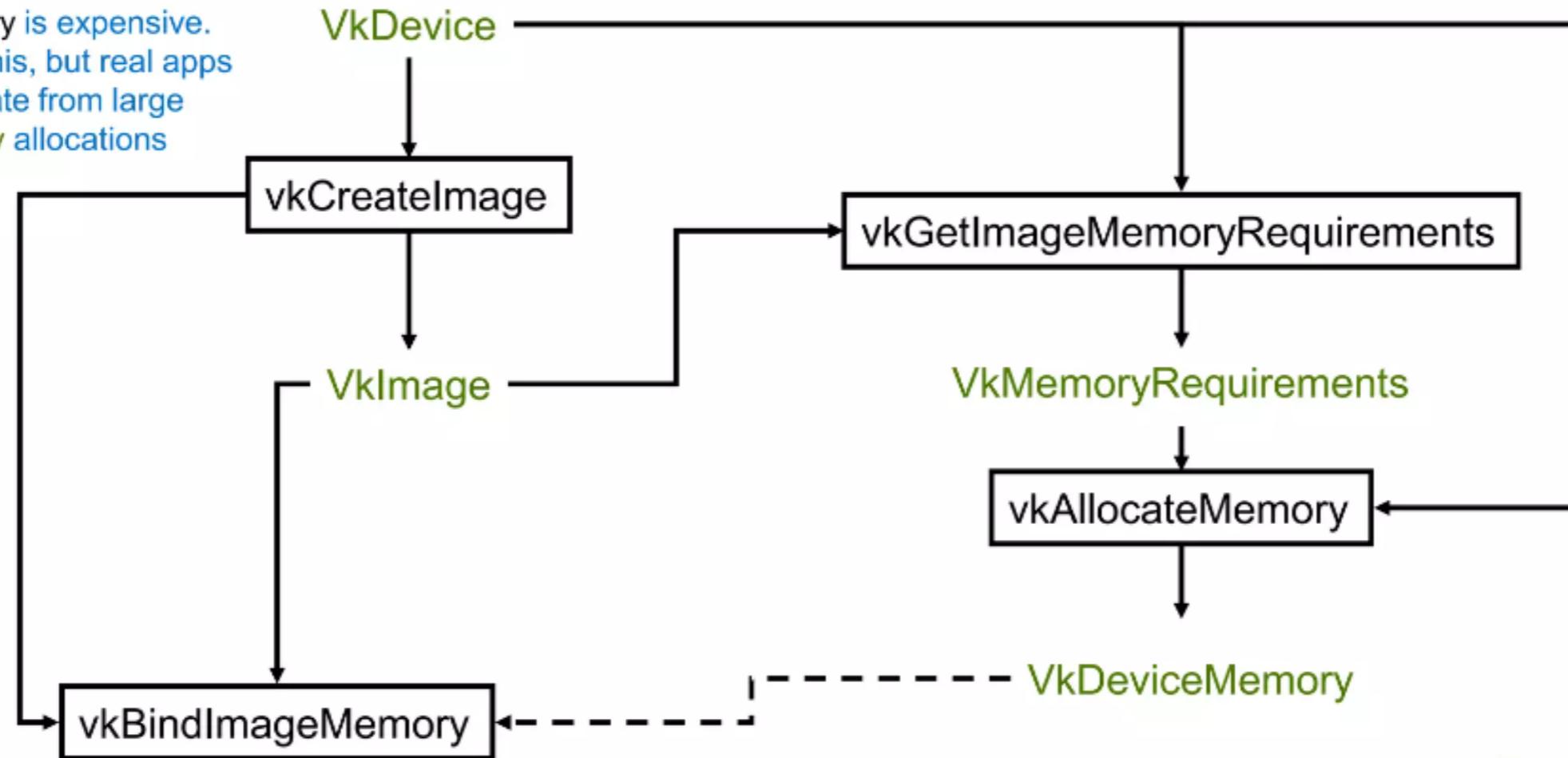
Allocating Image Memory for Texture

With Vulkan

Naïve approach!

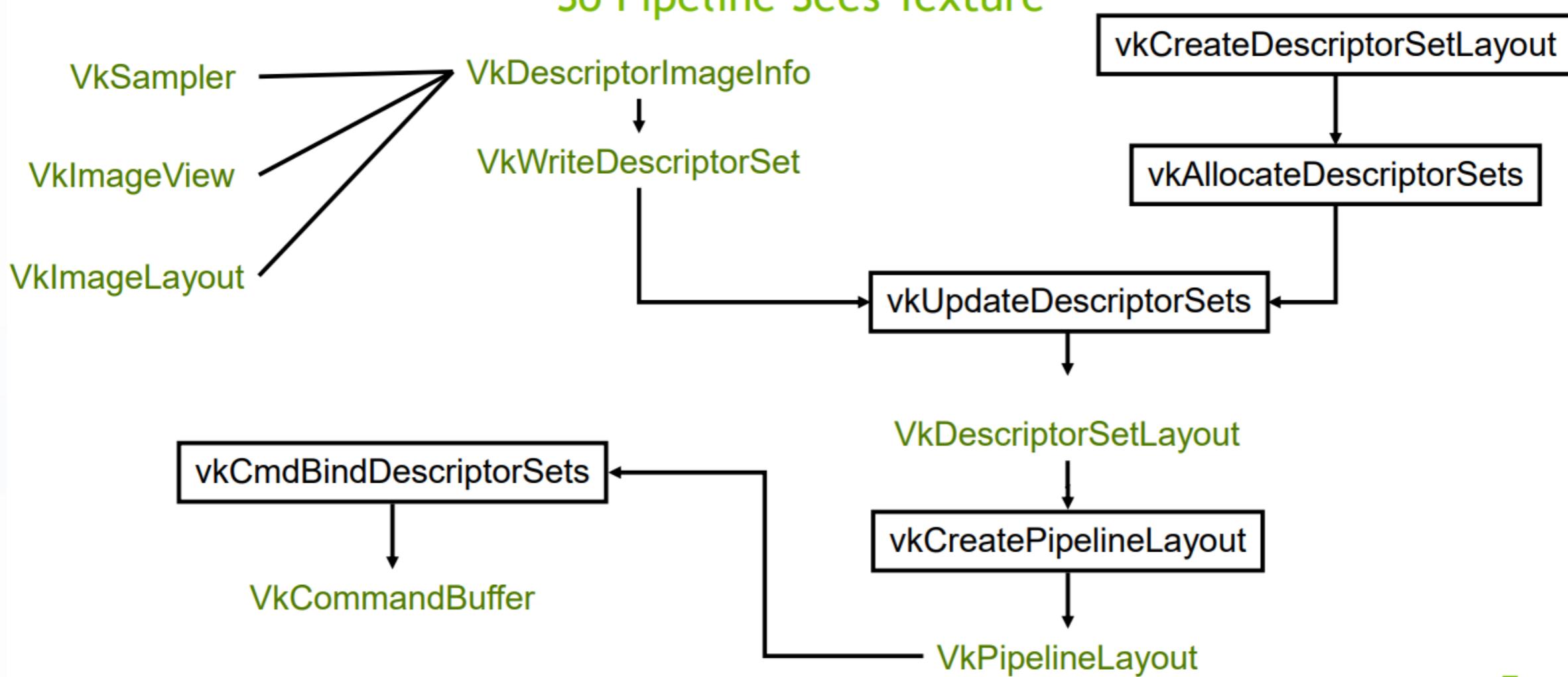
`vkAllocateMemory` is expensive.
Demos may do this, but real apps
should sub-allocate from large
`VkDeviceMemory` allocations

See next slide...



Binding Descriptor Sets for a Texture

So Pipeline Sees Texture



Establishing Sampler Bindings for a Pipeline Object

Vulkan Pipelines Need to Know How They Will Bind Samplers

`VkDescriptorSetLayoutBinding` *for a sampler binding*

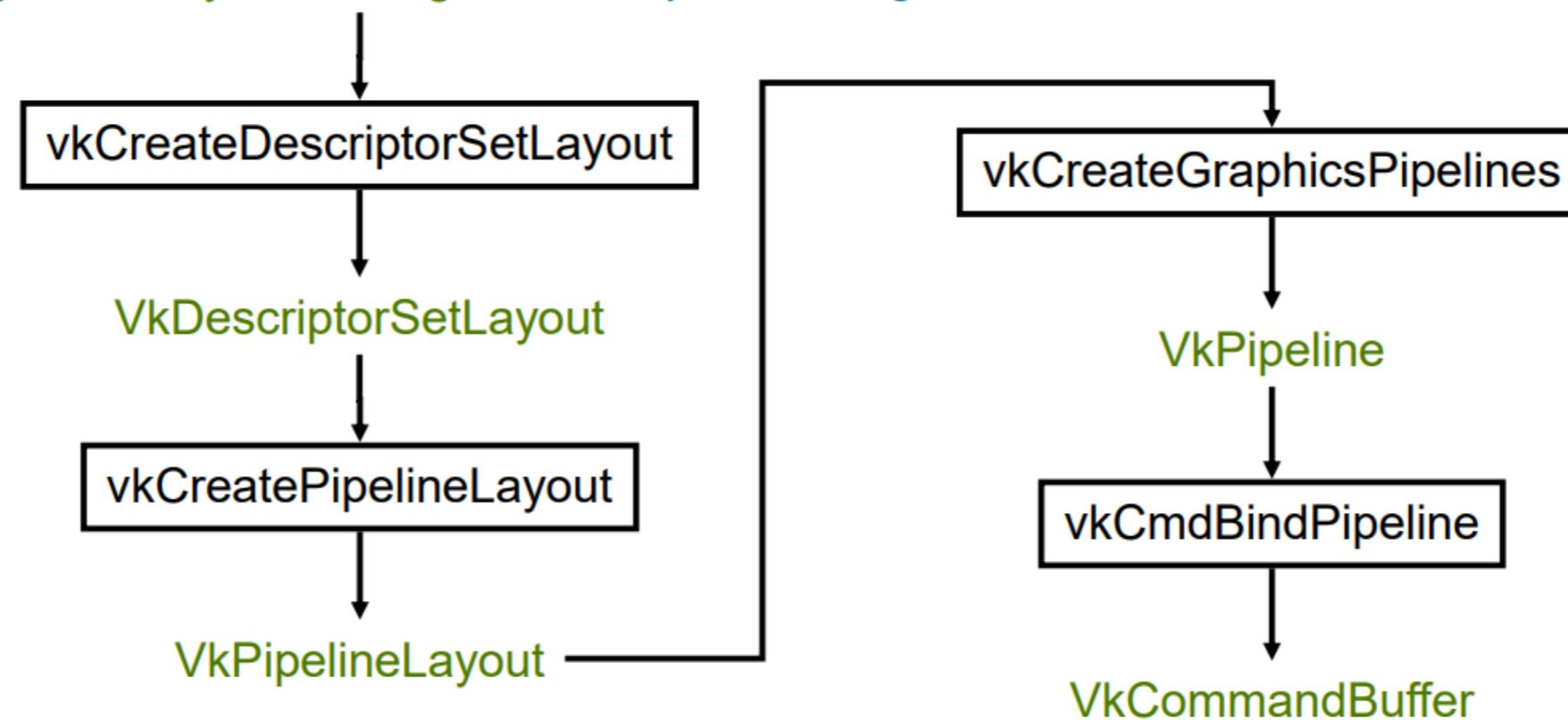


Image view and sampler

- [Texture image view](#)
- [Samplers](#)
- [Anisotropy device feature](#)

In this chapter we're going to create two more resources that are needed for the graphics pipeline to sample an image. The first resource is one that we've already seen before while working with the swap chain images, but the second one is new - it relates to how the shader will read texels from the image.

Texture image view

We've seen before, with the swap chain images and the framebuffer, that images are accessed through image views rather than directly. We will also need to create such an image view for the texture image.

```
VkImageView createImageView(VkImage image, VkFormat format) {
    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;
    if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
        throw std::runtime_error("failed to create texture image view!");
    }

    return imageView;
}
```



```
void createTextureImageView() {
    textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB);
}
```

And `createImageViews` can be simplified to:

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

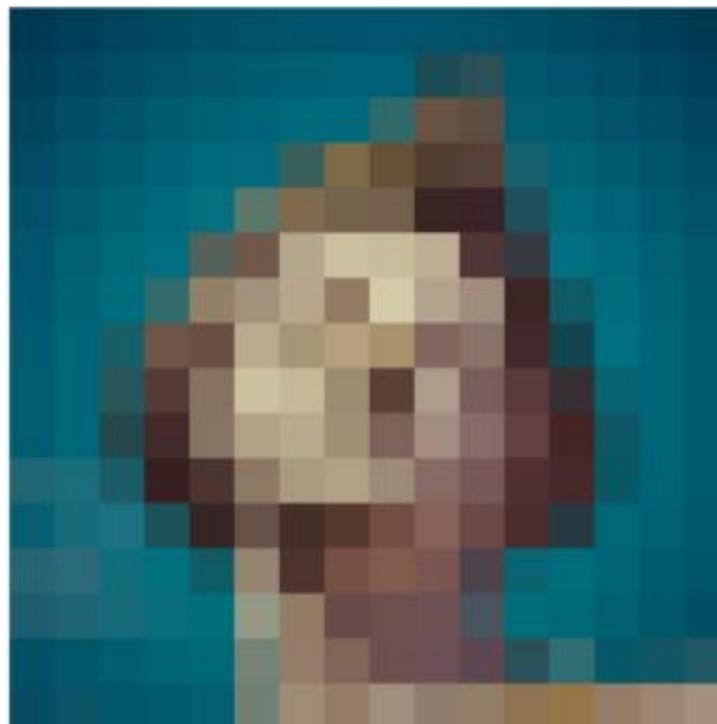
    for (uint32_t i = 0; i < swapChainImages.size(); i++) {
        swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat);
    }
}
```

Samplers

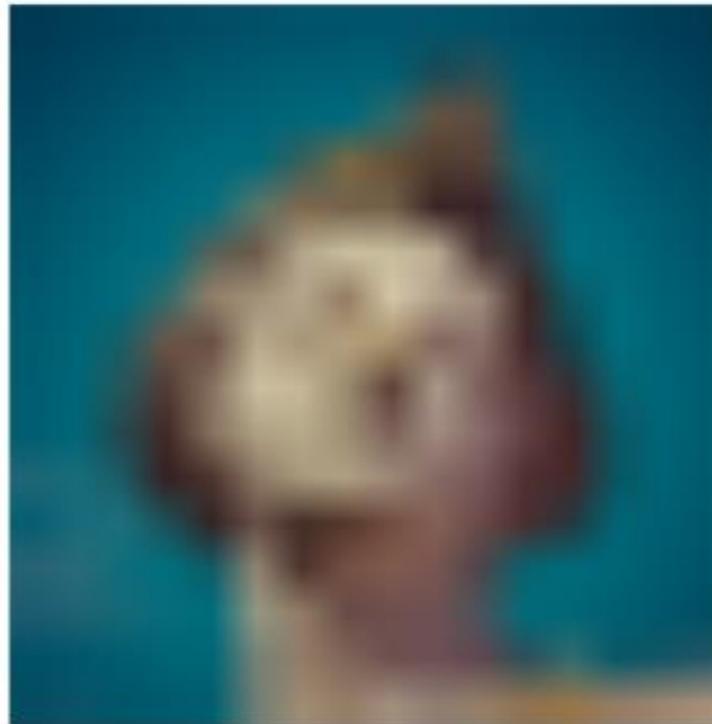
It is possible for shaders to read texels directly from images, but that is not very common when they are used as textures. Textures are usually accessed through samplers, which will apply filtering and transformations to compute the final color that is retrieved.

These filters are helpful to deal with problems like oversampling. Consider a texture that is mapped to geometry with more fragments than texels. If you simply took the closest texel for the texture coordinate in each fragment, then you would get a result like the first image:

No filtering

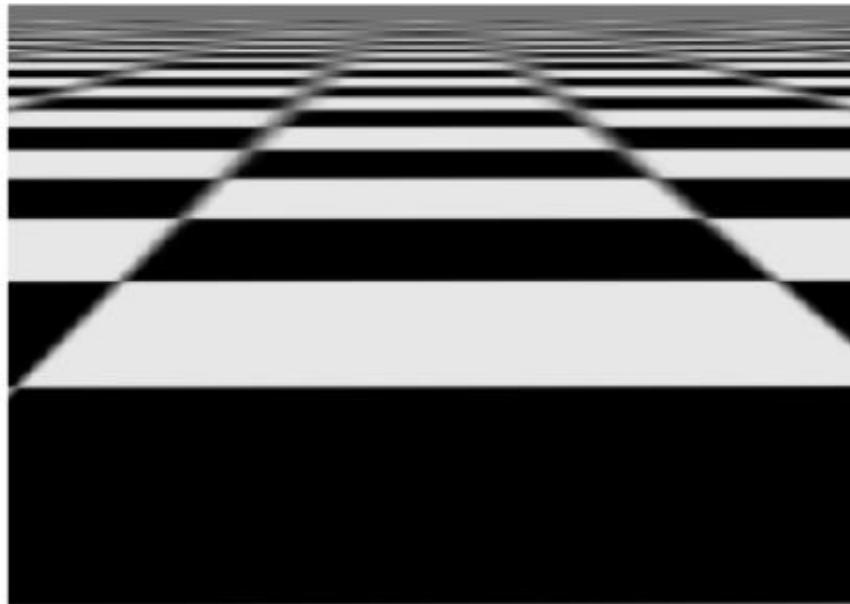


Bilinear filtering

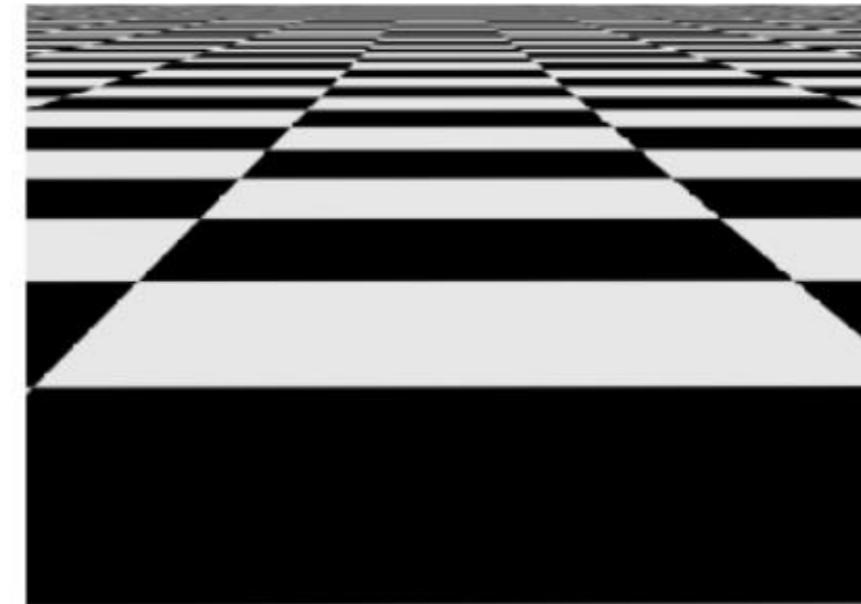


an.[®]

No anisotropic filtering



16x anisotropic filtering



As shown in the left image, the texture turns into a blurry mess in the distance. The solution to this is [anisotropic filtering](#), which can also be applied automatically by a sampler.

Aside from these filters, a sampler can also take care of transformations. It determines what happens when you try to read texels outside the image through its *addressing mode*. The image below displays some of the possibilities:

Repeat



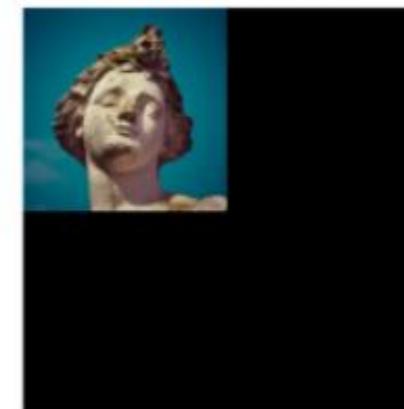
Mirrored repeat



Clamp to edge



Clamp to border



Vertex input description

- [Introduction](#)
- [Vertex shader](#)
- [Vertex data](#)
- [Binding descriptions](#)
- [Attribute descriptions](#)
- [Pipeline vertex input](#)

Vertex shader

First change the vertex shader to no longer include the vertex data in the shader code itself. The vertex shader takes input from a vertex buffer using the `in` keyword.

```
#version 450

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

The `inPosition` and `inColor` variables are *vertex attributes*. They're properties that are specified per-vertex in the vertex buffer, just like we manually specified a position and color per vertex using the two arrays. Make sure to recompile the vertex shader!



Vertex data

We're moving the vertex data from the shader code to an array in the code of our program. Start by including the GLM library, which provides us with linear algebra related types like vectors and matrices. We're going to use these types to specify the position and color vectors.

```
#include <glm/glm.hpp>
```

Create a new structure called `Vertex` with the two attributes that we're going to use in the vertex shader inside it:

```
struct Vertex {  
    glm::vec2 pos;  
    glm::vec3 color;  
};
```

GLM conveniently provides us with C++ types that exactly match the vector types used in the shader language.

```
const std::vector<Vertex> vertices = {  
    {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},  
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},  
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}  
};
```

Binding descriptions

The next step is to tell Vulkan how to pass this data format to the vertex shader once it's been uploaded into GPU memory. There are two types of structures needed to convey this information.

The first structure is `VkVertexInputBindingDescription` and we'll add a member function to the `Vertex` struct to populate it with the right data.

A vertex binding describes at which rate to load data from memory throughout the vertices. It specifies the number of bytes between data entries and whether to move to the next data entry after each vertex or after each instance.

```
VkVertexInputBindingDescription bindingDescription{};  
bindingDescription.binding = 0;  
bindingDescription.stride = sizeof(Vertex);  
bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

All of our per-vertex data is packed together in one array, so we're only going to have one binding. The `binding` parameter specifies the index of the binding in the array of bindings. The `stride` parameter specifies the number of bytes from one entry to the next, and the `inputRate` parameter can have one of the following values:

- `VK_VERTEX_INPUT_RATE_VERTEX`: Move to the next data entry after each vertex
- `VK_VERTEX_INPUT_RATE_INSTANCE`: Move to the next data entry after each instance

Attribute descriptions

The second structure that describes how to handle vertex input is [VkVertexInputAttributeDescription](#). We're going to add another helper function to `Vertex` to fill in these structs.

```
attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);
```

The `binding` parameter tells Vulkan from which binding the per-vertex data comes. The `location` parameter references the `location` directive of the input in the vertex shader. The input in the vertex shader with location `0` is the position, which has two 32-bit float components.

The `format` parameter describes the type of data for the attribute. A bit confusingly, the formats are specified using the same enumeration as color formats. The following shader types and formats are commonly used together:

- `float` : `VK_FORMAT_R32_SFLOAT`
- `vec2` : `VK_FORMAT_R32G32_SFLOAT`
- `vec3` : `VK_FORMAT_R32G32B32_SFLOAT`
- `vec4` : `VK_FORMAT_R32G32B32A32_SFLOAT`

As you can see, you should use the format where the amount of color channels matches the number of components in the shader data type. It is allowed to use more channels than the number of components in the shader, but they will be silently discarded. If the number of channels is lower than the number of components, then the BGA components will use default values of `(0, 0, 1)`. The color type (`SFLOAT`, `UINT`, `SINT`) and bit width should also match the type of the shader input. See the following examples:

- `ivec2` : `VK_FORMAT_R32G32_SINT`, a 2-component vector of 32-bit signed integers
- `uvec4` : `VK_FORMAT_R32G32B32A32_UINT`, a 4-component vector of 32-bit unsigned integers
- `double` : `VK_FORMAT_R64_SFLOAT`, a double-precision (64-bit) float

The `format` parameter implicitly defines the byte size of attribute data and the `offset` parameter specifies the number of bytes since the start of the per-vertex data to read from. The binding is loading one `Vertex` at a time and the position attribute (`pos`) is at an offset of `0` bytes from the beginning of this struct. This is automatically calculated using the `offsetof` macro.

```
attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);
```

The color attribute is described in much the same way.

Pipeline vertex input

We now need to set up the graphics pipeline to accept vertex data in this format by referencing the structures in `createGraphicsPipeline`. Find the `vertexInputInfo` struct and modify it to reference the two descriptions:

```
auto bindingDescription = Vertex::getBindingDescription();
auto attributeDescriptions = Vertex::getAttributeDescriptions();

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = static_cast<uint32_t>(attributeDescriptions.size());
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();
```

The pipeline is now ready to accept vertex data in the format of the `vertices` container and pass it on to our vertex shader. If you run the program now with validation layers enabled, you'll see that it complains that there is no vertex buffer bound to the binding. The next step is to create a vertex buffer and move the vertex data to it so the GPU is able to access it.

Vertex buffer creation

- [Introduction](#)
- [Buffer creation](#)
- [Memory requirements](#)
- [Memory allocation](#)
- [Filling the vertex buffer](#)
- [Binding the vertex buffer](#)

Introduction

Buffers in Vulkan are regions of memory used for storing arbitrary data that can be read by the graphics card. They can be used to store vertex data, which we'll do in this chapter, but they can also be used for many other purposes that we'll explore in future chapters. Unlike the Vulkan objects we've been dealing with so far, buffers do not automatically allocate memory for themselves. The work from the previous chapters has shown that the Vulkan API puts the programmer in control of almost everything and memory management is one of those things.

```
void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties, VkBuffer& buffer, VkDeviceMemory& bufferMemory)
{
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }

    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Memory requirements

The first step of allocating memory for the buffer is to query its memory requirements using the aptly named `vkGetBufferMemoryRequirements` function.

```
VkMemoryRequirements memRequirements;  
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
```

The `VkMemoryRequirements` struct has three fields:

- `size`: The size of the required amount of memory in bytes, may differ from `bufferInfo.size`.
- `alignment`: The offset in bytes where the buffer begins in the allocated region of memory, depends on `bufferInfo.usage` and `bufferInfo.flags`.
- `memoryTypeBits`: Bit field of the memory types that are suitable for the buffer.

Graphics cards can offer different types of memory to allocate from. Each type of memory varies in terms of allowed operations and performance characteristics. We need to combine the requirements of the buffer and our own application requirements to find the right type of memory to use.

```
uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {
    VkPhysicalDeviceMemoryProperties memProperties;
    vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);

    for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
        if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].propertyFlags & properties) == properties) {
            return i;
        }
    }

    throw std::runtime_error("failed to find suitable memory type!");
}
```



Filling the vertex buffer

It is now time to copy the vertex data to the buffer. This is done by [mapping the buffer memory](#) into CPU accessible memory with [vkMapMemory](#).

```
void* data;  
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
```

This function allows us to access a region of the specified memory resource defined by an offset and size. The offset and size here are `0` and `bufferInfo.size`, respectively. It is also possible to specify the special value `VK_WHOLE_SIZE` to map all of the memory. The second to last parameter can be used to specify flags, but there aren't any available yet in the current API. It must be set to the value `0`. The last parameter specifies the output for the pointer to the mapped memory.

```
void* data;  
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);  
memcpy(data, vertices.data(), (size_t) bufferInfo.size);  
vkUnmapMemory(device, vertexBufferMemory);
```

You can now simply `memcpy` the vertex data to the mapped memory and unmap it again using [vkUnmapMemory](#). Unfortunately the driver may not immediately copy the data into the buffer memory, for example because of caching. It is also possible that writes to the buffer are not visible in the mapped memory yet. There are two ways to deal with that problem:

- Use a memory heap that is host coherent, indicated with `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- Call [vkFlushMappedMemoryRanges](#) after writing to the mapped memory, and call [vkInvalidateMappedMemoryRanges](#) before reading from the mapped memory

Staging buffer

- [Introduction](#)
- [Transfer queue](#)
- [Abstracting buffer creation](#)
- [Using a staging buffer](#)
- [Conclusion](#)

Transfer queue

The buffer copy command requires a queue family that supports transfer operations, which is indicated using `VK_QUEUE_TRANSFER_BIT`. The good news is that any queue family with `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT` capabilities already implicitly support `VK_QUEUE_TRANSFER_BIT` operations. The implementation is not required to explicitly list it in `queueFlags` in those cases.

If you like a challenge, then you can still try to use a different queue family specifically for transfer operations. It will require you to make the following modifications to your program:

- Modify `QueueFamilyIndices` and `findQueueFamilies` to explicitly look for a queue family with the `VK_QUEUE_TRANSFER_BIT` bit, but not the `VK_QUEUE_GRAPHICS_BIT`.
- Modify `createLogicalDevice` to request a handle to the transfer queue
- Create a second command pool for command buffers that are submitted on the transfer queue family
- Change the `sharingMode` of resources to be `VK_SHARING_MODE_CONCURRENT` and specify both the graphics and transfer queue families
- Submit any transfer commands like `vkCmdCopyBuffer` (which we'll be using in this chapter) to the transfer queue instead of the graphics queue

```

void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
                VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                stagingBuffer, stagingBufferMemory);

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
        memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer, vertexBufferMemory);

    copyBuffer(stagingBuffer, vertexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}

```

We're now using a new `stagingBuffer` with `stagingBufferMemory` for mapping and copying the vertex data. In this chapter we're going to use two new buffer usage flags:

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT`: Buffer can be used as source in a memory transfer operation.
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT`: Buffer can be used as destination in a memory transfer operation.

The `vertexBuffer` is now allocated from a memory type that is device local, which generally means that we're not able to use `vkMapMemory`. However, we can copy data from the `stagingBuffer` to the `vertexBuffer`. We have to indicate that we intend to do that by specifying the transfer source flag for the `stagingBuffer` and the transfer destination flag for the `vertexBuffer`, along with the vertex buffer usage flag.

We're now going to write a function to copy the contents from one buffer to another, called `copyBuffer`.

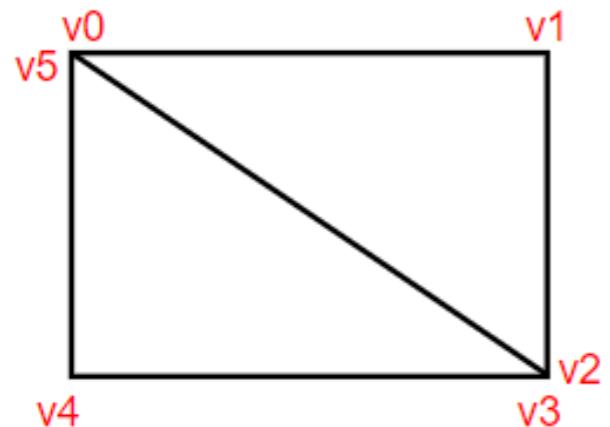
Index buffer

- Introduction
- Index buffer creation
- Using an index buffer

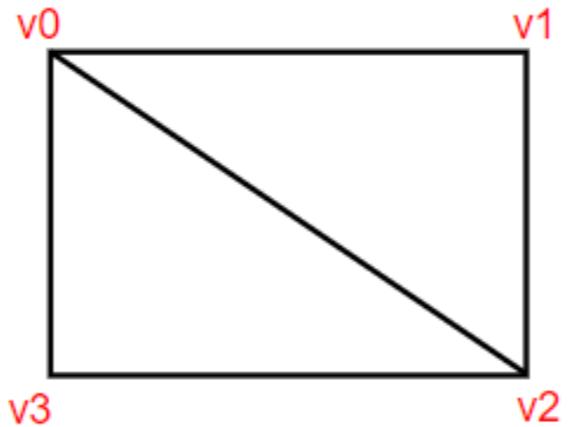
Introduction

The 3D meshes you'll be rendering in a real world application will often share vertices between multiple triangles. This already happens even with something simple like drawing a rectangle:

Vertex buffer only



Vertex + index buffer

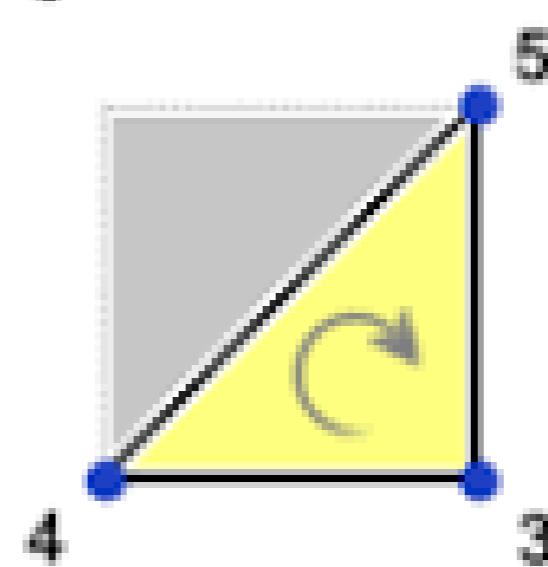
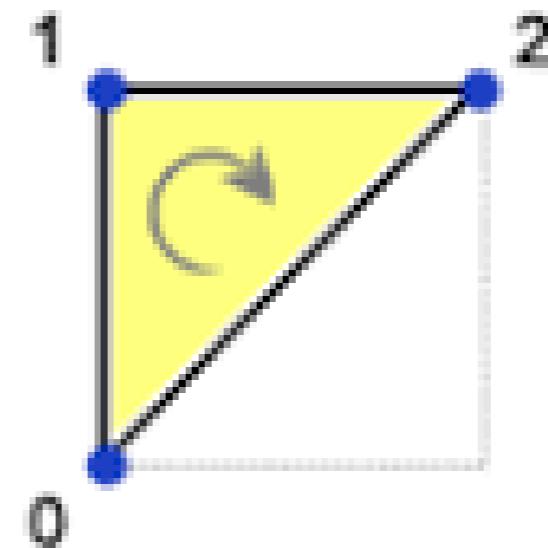


Indices
 $\{0, 1, 2, 2, 3, 0\}$

Vertex Buffer

VB Index : 0

0	(-1, -1)
1	(-1, 1)
2	(1, 1)
3	(1, -1)
4	(-1, -1)
5	(1, -1)



Index Buffer

IB Index:	0
1	1
2	2
3	3
4	0
5	2

Vertex Buffer

VB Index:	0
1	(-1, -1)
2	(-1, 1)
3	(1, 1)
4	(1, -1)

Index buffer creation

In this chapter we're going to modify the vertex data and add index data to draw a rectangle like the one in the illustration. Modify the vertex data to represent the four corners:

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}
};
```

The top-left corner is red, top-right is green, bottom-right is blue and the bottom-left is white. We'll add a new array `indices` to represent the contents of the index buffer. It should match the indices in the illustration to draw the upper-right triangle and bottom-left triangle.

```
const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0
};
```

Using an index buffer

Using an index buffer for drawing involves two changes to `recordCommandBuffer`. We first need to bind the index buffer, just like we did for the vertex buffer. The difference is that you can only have a single index buffer. It's unfortunately not possible to use different indices for each vertex attribute, so we do still have to completely duplicate vertex data even if just one attribute varies.

```
vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, offsets);

vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT16);
```

An index buffer is bound with `vkCmdBindIndexBuffer` which has the index buffer, a byte offset into it, and the type of index data as parameters. As mentioned before, the possible types are `VK_INDEX_TYPE_UINT16` and `VK_INDEX_TYPE_UINT32`.

Just binding an index buffer doesn't change anything yet, we also need to change the drawing command to tell Vulkan to use the index buffer. Remove the `vkCmdDraw` line and replace it with `vkCmdDrawIndexed`:

```
vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Uniforms

Uniforms are another way to pass data from our application on the CPU to the shaders on the GPU. Uniforms are however slightly different compared to vertex attributes. First of all, uniforms are global. Global, meaning that a uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program. Second, whatever you set the uniform value to, uniforms will keep their values until they're either reset or updated.

To declare a uniform in GLSL we simply add the `uniform` keyword to a shader with a type and a name. From that point on we can use the newly declared uniform in the shader. Let's see if this time we can set the color of the triangle via a uniform:

```
1 #version 450
2
3 layout(binding = 0) uniform UniformBufferObject {
4     mat4 model;
5     mat4 view;
6     mat4 proj;
7 } ubo;
```

```
#version 450

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

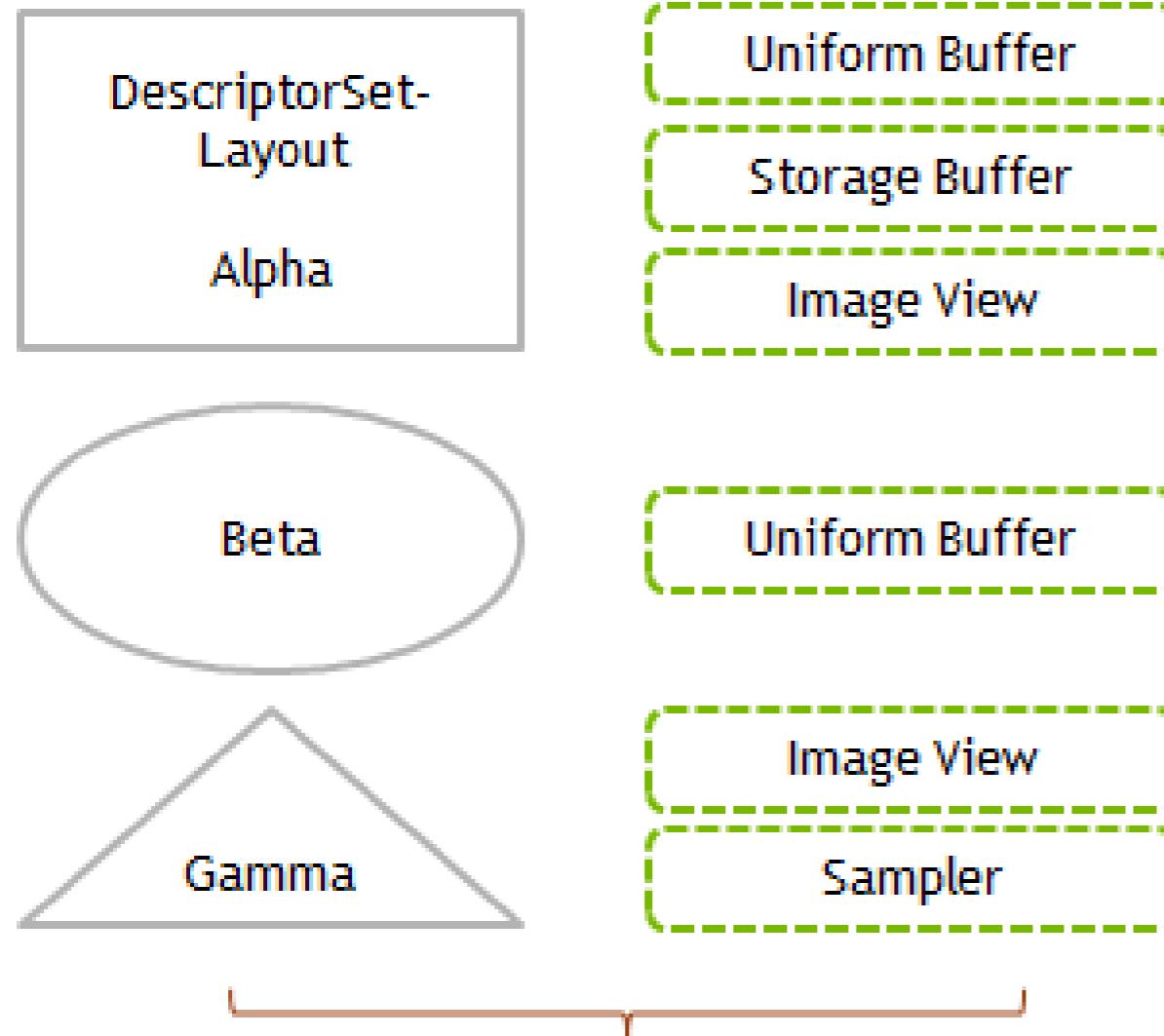
void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Descriptor layout and buffer

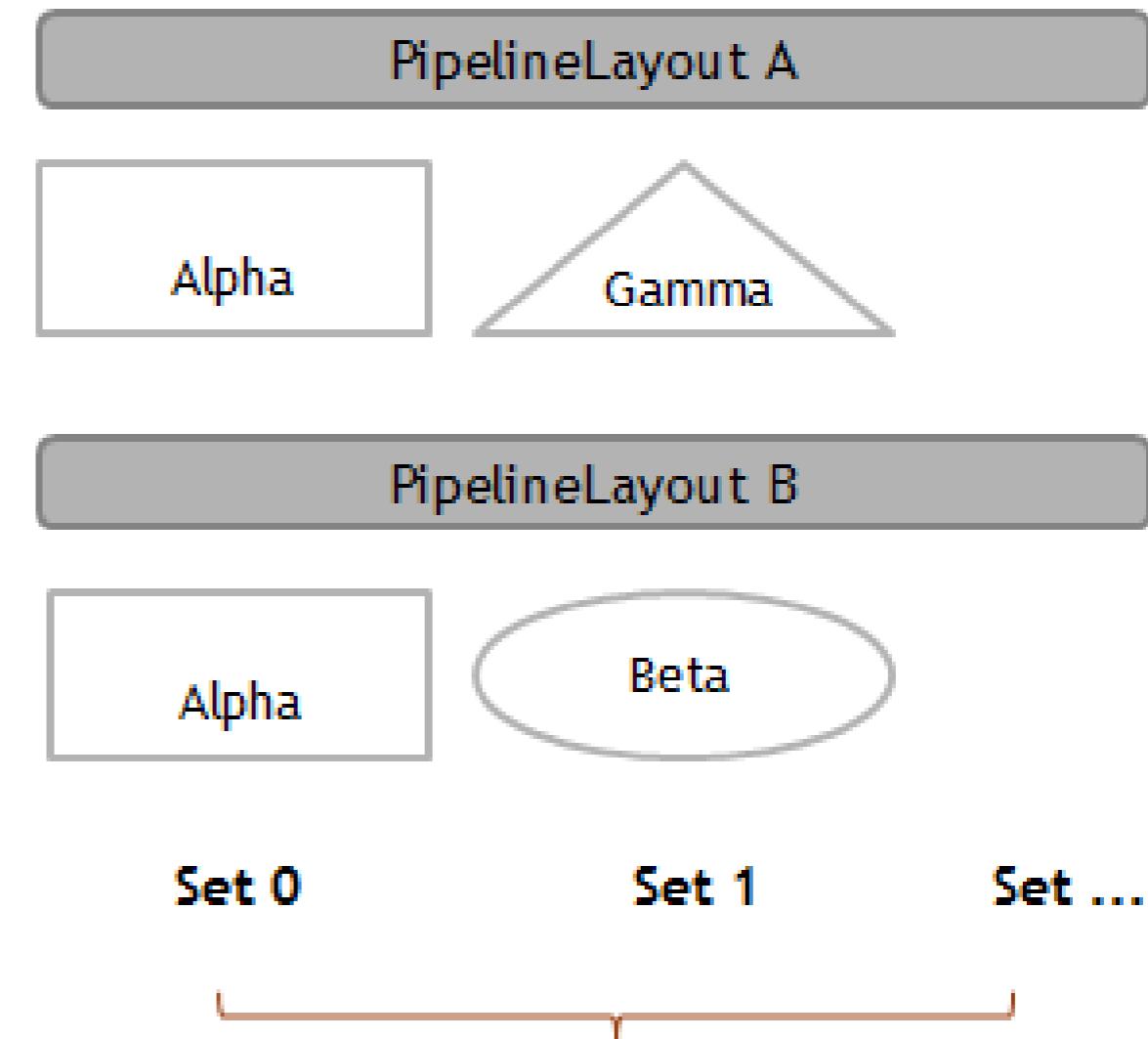
- [Introduction](#)
- [Vertex shader](#)
- [Descriptor set layout](#)
- [Uniform buffer](#)
- [Updating uniform data](#)



KB-UIS
We Work In-Depth

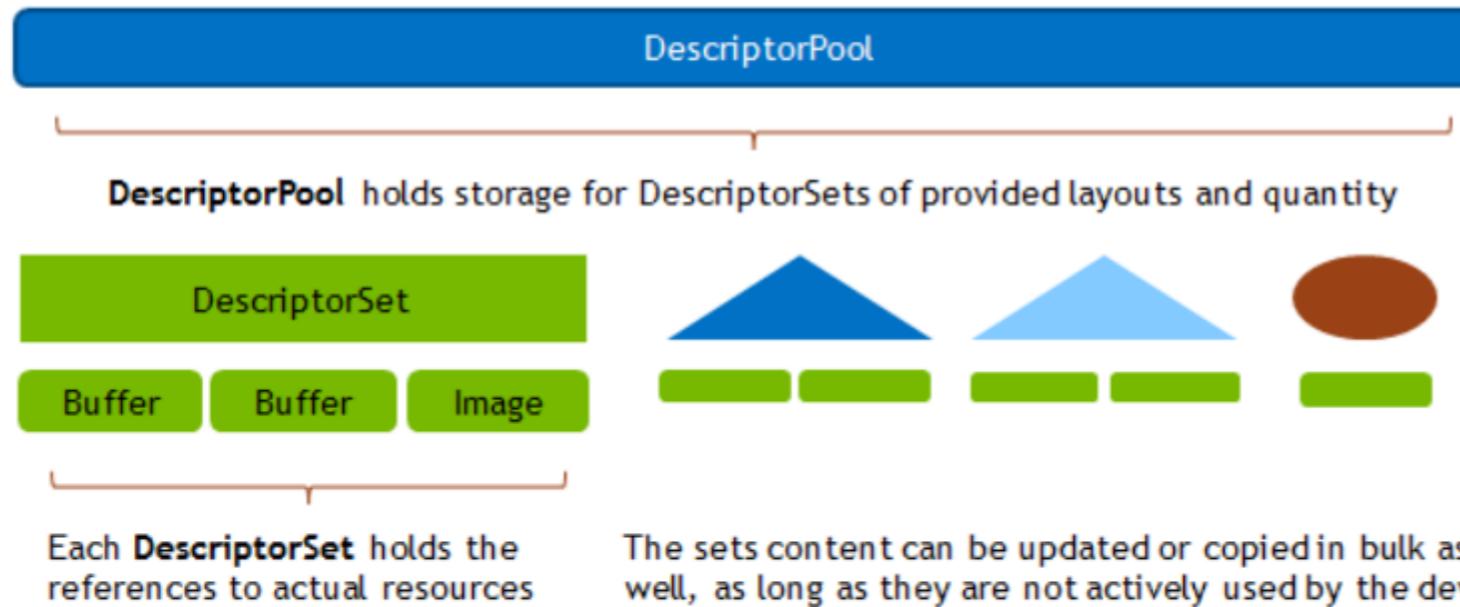


DescriptorSetLayouts define what type of resources are bound within the group

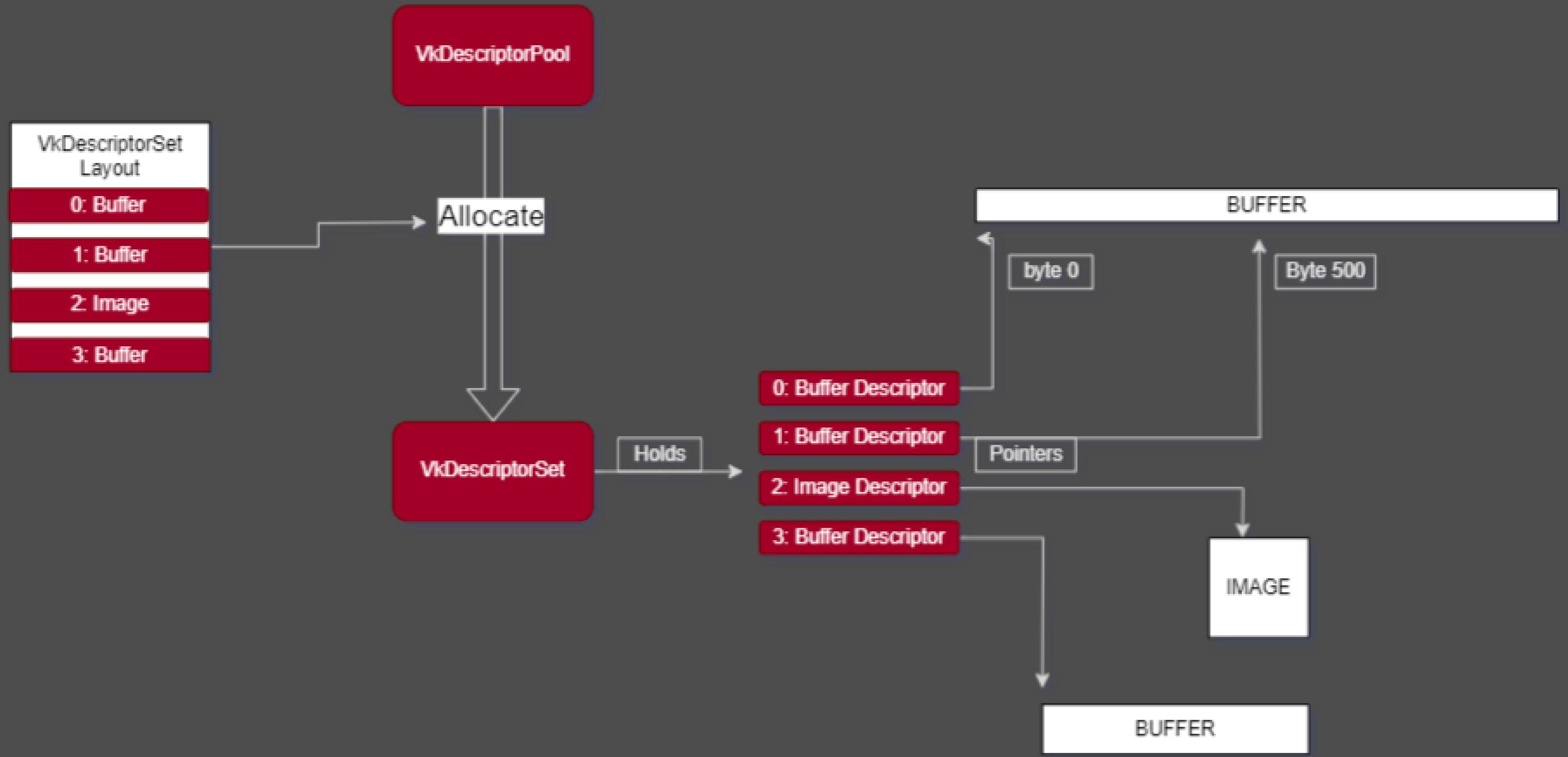


PipelineLayout defines which sets may be used by a Pipeline and at which set number.

- **DescriptorSetLayout:** This object describes which bindings are in the DescriptorSet for every shader stage. For example, we define at binding 0 a constant buffer used by both vertex and fragment stages, at binding 1 a storage buffer and at 2 an image only for fragment stage. It is the developer's responsibility to ensure the shaders (SPIR-V) have compatible definitions for the DescriptorSet.
- **PipelineLayout:** As a pipeline (shader and most important rendering state) can have multiple DescriptorSets, this object defines which DescriptorSetLayouts are used with each set binding number. Using the same DescriptorSetLayouts at the same units across pipelines, has some performance benefits, more about that later.



- **DescriptorSet:** This is the object that we will later use to bind the resources, therefore it will contain the actual data we reference. Image, Sampler or Buffers are referenced.
- **DescriptorPool:** The DescriptorSets are allocated from the pool. Which layouts and how many sets for each can be allocated is defined by the developer at creation time of the pool.



Descriptor set layout

The next step is to define the UBO on the C++ side and to tell Vulkan about this descriptor in the vertex shader.

```
struct UniformBufferObject {  
    glm::mat4 model;  
    glm::mat4 view;  
    glm::mat4 proj;  
};
```

We can exactly match the definition in the shader using data types in GLM. The data in the matrices is binary compatible with the way the shader expects it, so we can later just `memcpy` a `UniformBufferObject` to a `VkBuffer`.

We need to provide details about every descriptor binding used in the shaders for pipeline creation, just like we had to do for every vertex attribute and its `location` index. We'll set up a new function to define all of this information called `createDescriptorSetLayout`. It should be called right before pipeline creation, because we're going to need it there.

Every binding needs to be described through a `VkDescriptorSetLayoutBinding` struct.

```
void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding{};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.descriptorCount = 1;
}
```

The first two fields specify the `binding` used in the shader and the type of descriptor, which is a uniform buffer object. It is possible for the shader variable to represent an array of uniform buffer objects, and `descriptorCount` specifies the number of values in the array. This could be used to specify a transformation for each of the bones in a skeleton for skeletal animation, for example. Our MVP transformation is in a single uniform buffer object, so we're using a `descriptorCount` of `1`.

```
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

We also need to specify in which shader stages the descriptor is going to be referenced. The `stageFlags` field can be a combination of `VkShaderStageFlagBits` values or the value `VK_SHADER_STAGE_ALL_GRAPHICS`. In our case, we're only referencing the descriptor from the vertex shader.



```
void createUniformBuffers() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    uniformBuffers.resize(MAX_FRAMES_IN_FLIGHT);
    uniformBuffersMemory.resize(MAX_FRAMES_IN_FLIGHT);
    uniformBuffersMapped.resize(MAX_FRAMES_IN_FLIGHT);

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
                    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                    uniformBuffers[i], uniformBuffersMemory[i]);

        vkMapMemory(device, uniformBuffersMemory[i], 0, bufferSize, 0, &uniformBuffersMapped[i]);
    }
}
```

```
void updateUniformBuffer(uint32_t currentImage) {
    static auto startTime = std::chrono::high_resolution_clock::now();

    auto currentTime = std::chrono::high_resolution_clock::now();
    float time = std::chrono::duration<float, std::chrono::seconds::period>(currentTime - startTime).count();

    UniformBufferObject ubo{};
    ubo.model = glm::rotate(glm::mat4(1.0f), time * glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
    ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f));
    ubo.proj = glm::perspective(glm::radians(45.0f), swapChainExtent.width / (float) swapChainExtent.height, 0.1f, 10.0f);
    ubo.proj[1][1] *= -1;

    memcpy(uniformBuffersMapped[currentImage], &ubo, sizeof(ubo));
}

void drawFrame() {
    vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);

    uint32_t imageIndex;
    VkResult result = vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
        imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

    if (result == VK_ERROR_OUT_OF_DATE_KHR) {
        recreateSwapChain();
        return;
    } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
        throw std::runtime_error("failed to acquire swap chain image!");
    }

    updateUniformBuffer(currentFrame);
}
```

Combined image sampler

- [Introduction](#)
- [Updating the descriptors](#)
- [Texture coordinates](#)
- [Shaders](#)

Introduction

We looked at descriptors for the first time in the uniform buffers part of the tutorial. In this chapter we will look at a new type of descriptor: *combined image sampler*. This descriptor makes it possible for shaders to access an image resource through a sampler object like the one we created in the previous chapter.

We'll start by modifying the descriptor layout, descriptor pool and descriptor set to include such a combined image sampler descriptor. After that, we're going to add texture coordinates to `Vertex` and modify the fragment shader to read colors from the texture instead of just interpolating the vertex colors.

Updating the descriptors

Browse to the `createDescriptorSetLayout` function and add a `VkDescriptorSetLayoutBinding` for a combined image sampler descriptor. We'll simply put it in the binding after the uniform buffer:

```
VkDescriptorSetLayoutBinding samplerLayoutBinding{};  
samplerLayoutBinding.binding = 1;  
samplerLayoutBinding.descriptorCount = 1;  
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;  
samplerLayoutBinding.pImmutableSamplers = nullptr;  
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;  
  
std::array<VkDescriptorSetLayoutBinding, 2> bindings = {uboLayoutBinding, samplerLayoutBinding};  
VkDescriptorSetLayoutCreateInfo layoutInfo{};  
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;  
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());  
layoutInfo.pBindings = bindings.data();
```

Make sure to set the `stageFlags` to indicate that we intend to use the combined image sampler descriptor in the fragment shader. That's where the color of the fragment is going to be determined. It is possible to use texture sampling in the vertex shader, for example to dynamically deform a grid of vertices by a [heightmap](#).

We must also create a larger descriptor pool to make room for the allocation of the combined image sampler by adding another `VkPoolSize` of type `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` to the `VkDescriptorPoolCreateInfo`. Go to the `createDescriptorPool` function and modify it to include a `VkDescriptorPoolSize` for this descriptor:

```
std::array<VkDescriptorPoolSize, 2> poolSizes{};  
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;  
poolSizes[0].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);  
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;  
poolSizes[1].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);  
  
VkDescriptorPoolCreateInfo poolInfo{};  
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;  
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());  
poolInfo.pPoolSizes = poolSizes.data();  
poolInfo.maxSets = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
```

The final step is to bind the actual image and sampler resources to the descriptors in the descriptor set. Go to the [createDescriptorSets](#) function.

```
for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {  
    VkDescriptorBufferInfo bufferInfo{};  
    bufferInfo.buffer = uniformBuffers[i];  
    bufferInfo.offset = 0;  
    bufferInfo.range = sizeof(UniformBufferObject);  
  
    VkDescriptorImageInfo imageInfo{};  
    imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;  
    imageInfo.imageView = textureImageView;  
    imageInfo.sampler = textureSampler;  
  
    ...  
}
```

The resources for a combined image sampler structure must be specified in a [VkDescriptorImageInfo](#) struct, just like the buffer resource for a uniform buffer descriptor is specified in a [VkDescriptorBufferInfo](#) struct. This is where the objects from the previous chapter come together.



```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};  
  
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;  
descriptorWrites[0].dstSet = descriptorSets[i];  
descriptorWrites[0].dstBinding = 0;  
descriptorWrites[0].dstArrayElement = 0;  
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;  
descriptorWrites[0].descriptorCount = 1;  
descriptorWrites[0].pBufferInfo = &bufferInfo;  
  
descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;  
descriptorWrites[1].dstSet = descriptorSets[i];  
descriptorWrites[1].dstBinding = 1;  
descriptorWrites[1].dstArrayElement = 0;  
descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;  
descriptorWrites[1].descriptorCount = 1;  
descriptorWrites[1].pImageInfo = &imageInfo;  
  
vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()), descriptorWrites.data(), 0, nullptr);
```

Texture Coordinate Input Attribute

```
attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);

attributeDescriptions[2].binding = 0;
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, texCoord);
```

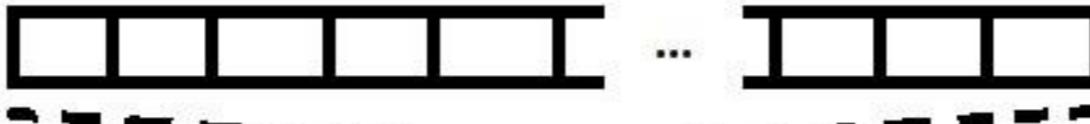
```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
};
```

Layout Transitions



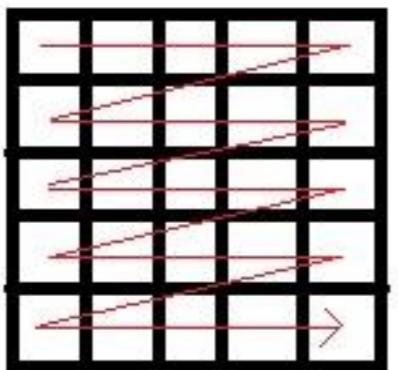
Raw memory

Array of bytes :

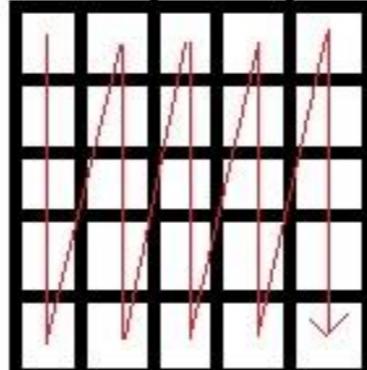


View

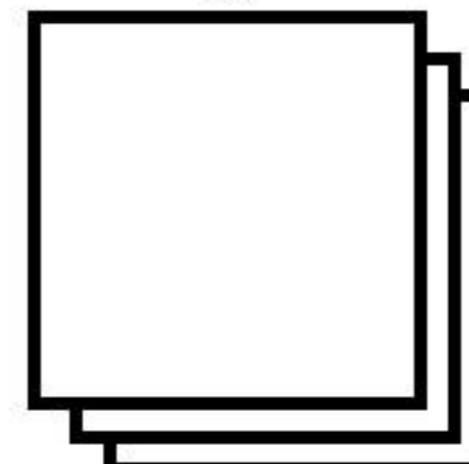
Analysis :



Row-major

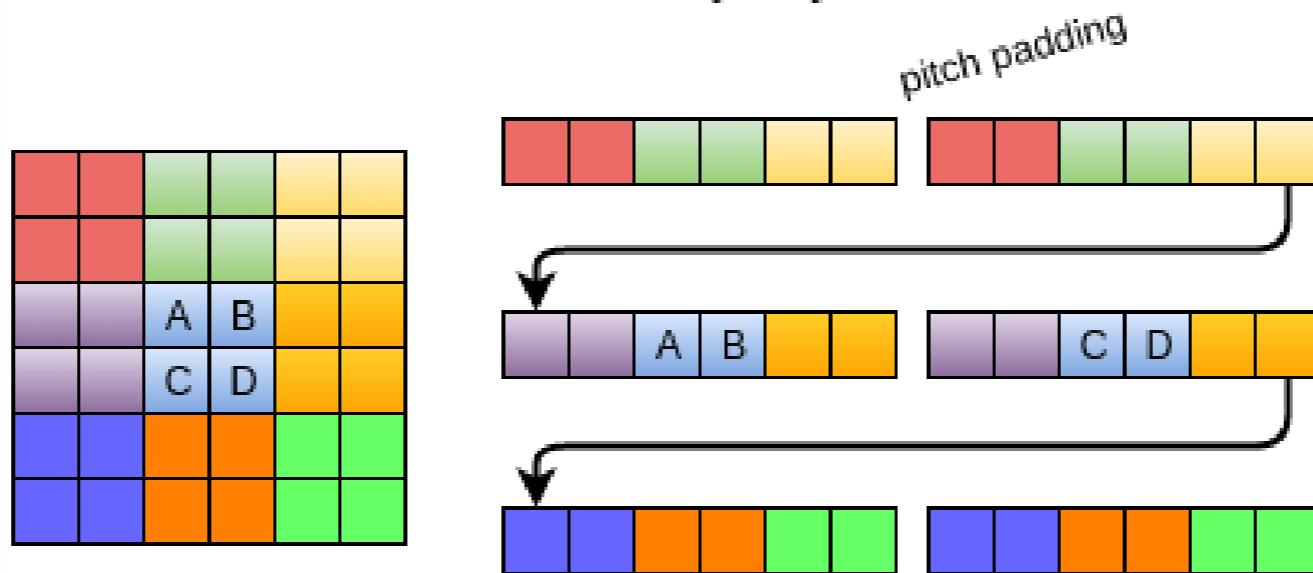


Column-major

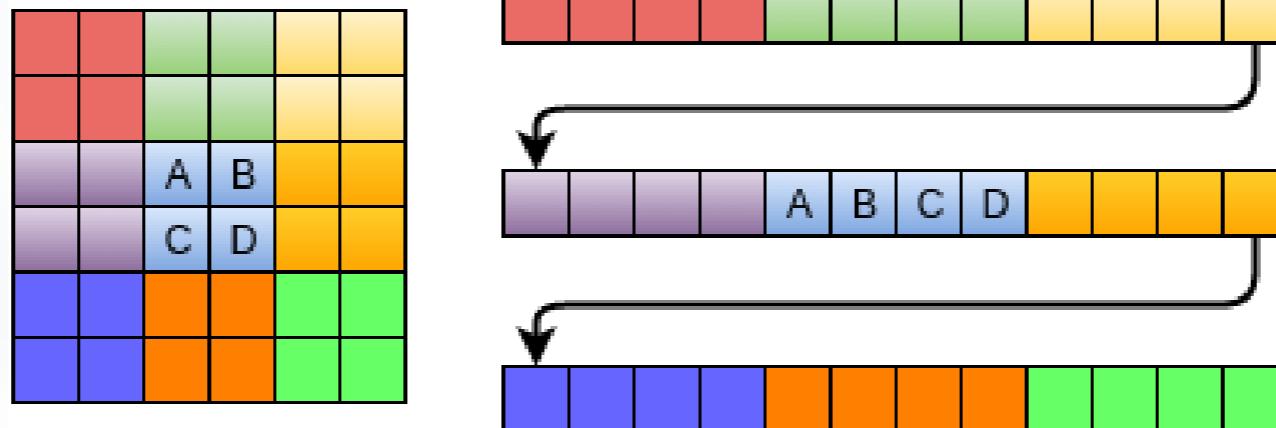


Layered

Linear Memory Layout



Optimal (Tiled) Memory Layout



Layout Transitions

```
void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageLayout newLayout) {  
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();  
  
    VkImageMemoryBarrier barrier{};  
    barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;  
    barrier.oldLayout = oldLayout;  
    barrier.newLayout = newLayout;  
    barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;  
    barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;  
    barrier.image = image;  
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;  
    barrier.subresourceRange.baseMipLevel = 0;  
    barrier.subresourceRange.levelCount = 1;  
    barrier.subresourceRange.baseArrayLayer = 0;  
    barrier.subresourceRange.layerCount = 1;  
  
    VkPipelineStageFlags sourceStage;  
    VkPipelineStageFlags destinationStage;
```

```
if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL)
barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else {
    throw std::invalid_argument("unsupported layout transition!");
}

vkCmdPipelineBarrier(
    commandBuffer,
    sourceStage, destinationStage,
    0,
    0, nullptr,
    0, nullptr,
    1, &barrier
);

endSingleTimeCommands(commandBuffer);
```

Preparing the texture image

We now have all of the tools we need to finish setting up the texture image, so we're going back to the `createTextureImage` function. The last thing we did there was creating the texture image. The next step is to copy the staging buffer to the texture image. This involves two steps:

- Transition the texture image to `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`
- Execute the buffer to image copy operation

This is easy to do with the functions we just created:

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);
copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHeight));
```

The image was created with the `VK_IMAGE_LAYOUT_UNDEFINED` layout, so that one should be specified as old layout when transitioning `textureImage`. Remember that we can do this because we don't care about its contents before performing the copy operation.

To be able to start sampling from the texture image in the shader, we need one last transition to prepare it for shader access:

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
```

Transition barrier masks

```
VkPipelineStageFlags sourceStage;
VkPipelineStageFlags destinationStage;

if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

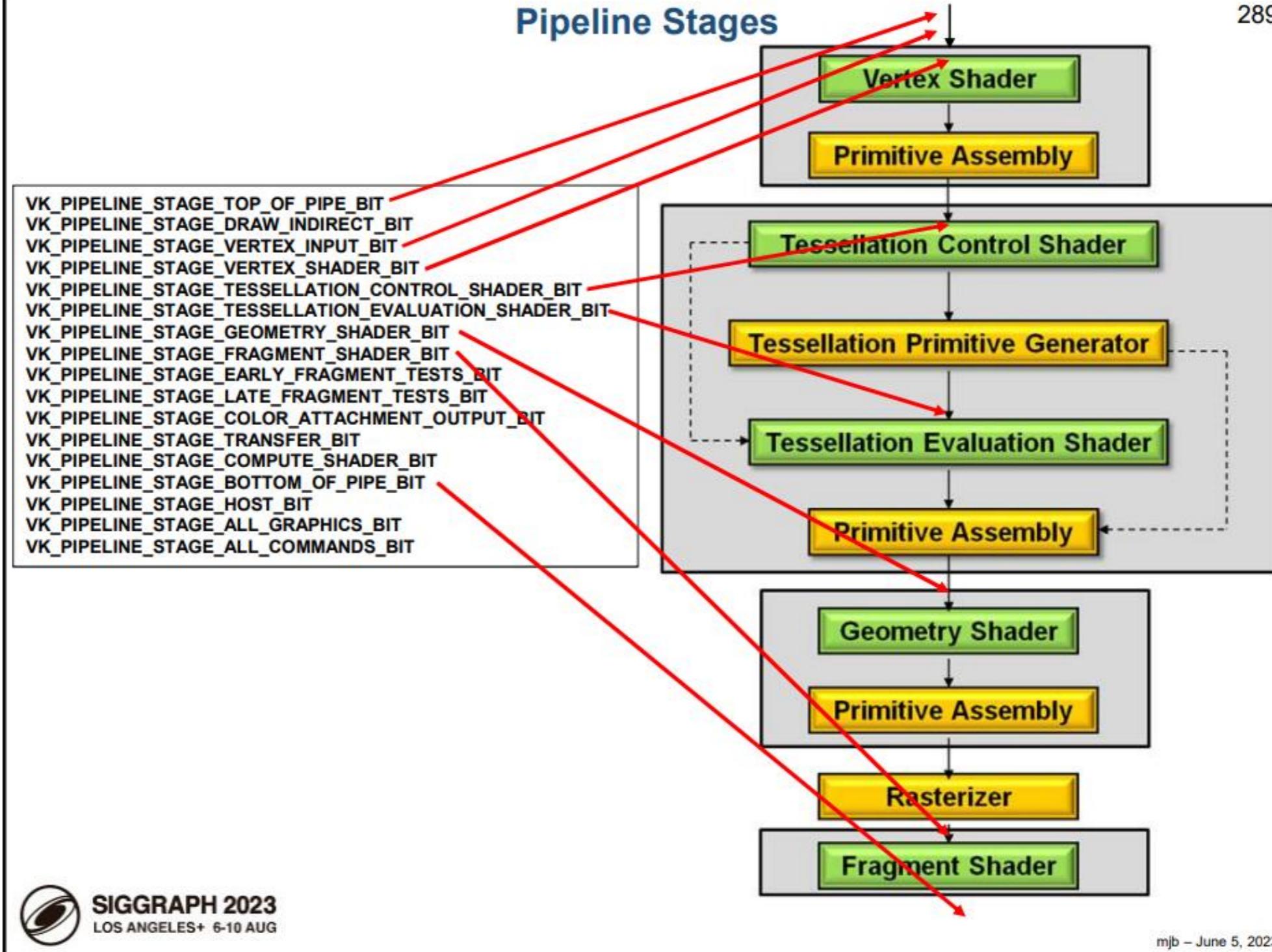
    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
    destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else {
    throw std::invalid_argument("unsupported layout transition!");
}

vkCmdPipelineBarrier(
    commandBuffer,
    sourceStage, destinationStage,
    0,
    0, nullptr,
    0, nullptr,
    1, &barrier
);
```

Pipeline Stages

289



SIGGRAPH 2023
LOS ANGELES+ 6-10 AUG

Example: Be sure we are done writing an Output image before using it as a Fragment Shader Texture

293

```
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
```

Stages

src
dst

```
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
```

Access types

dst
src



Si
Lo

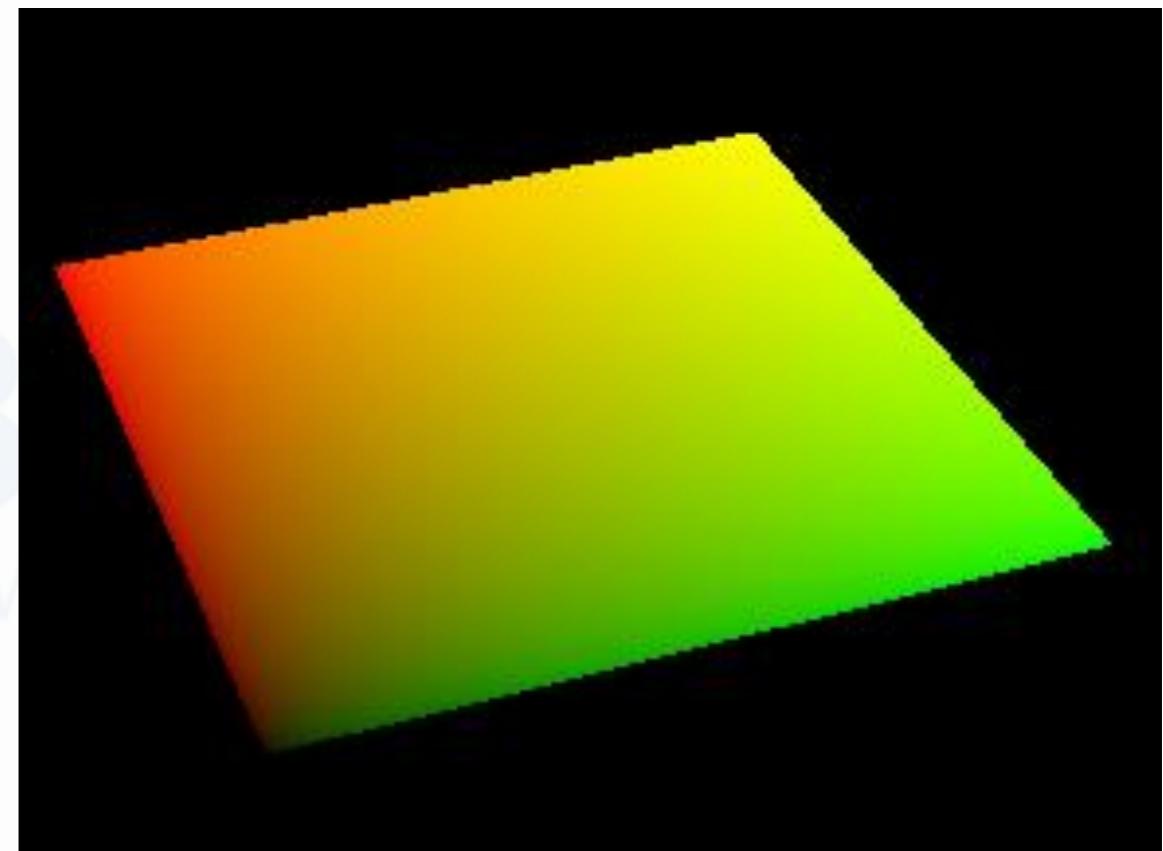
Exercise #1

Use equal weightage of texture
and (currently ignored) vertex
color to compute the final color



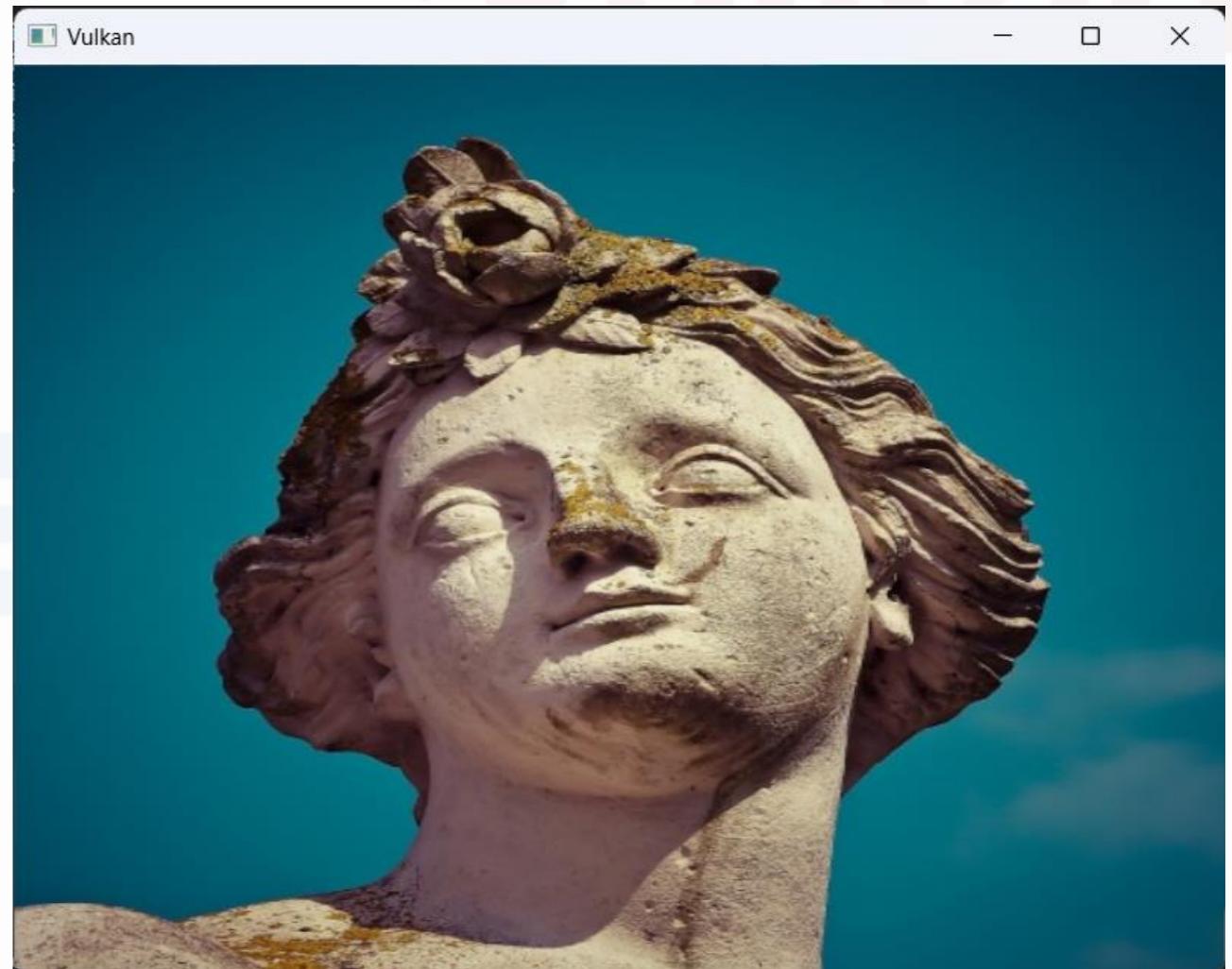
Exercise #2

Use texture coordinates to color
the rectangle (red and green)
instead of the texture sample

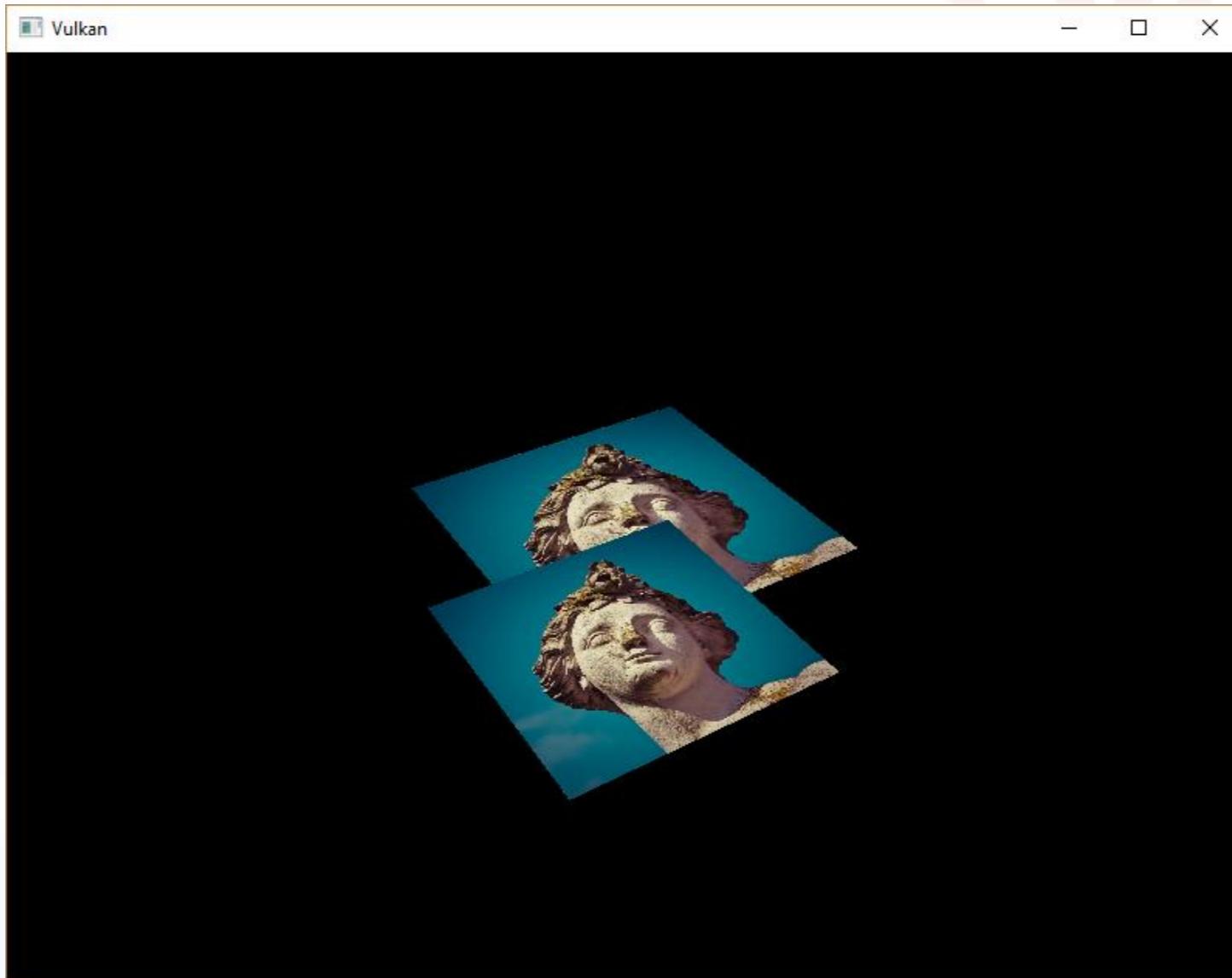


Exercise #3

Disable the transformation and projection (rotation and perspective), display as 2D image filling the frame

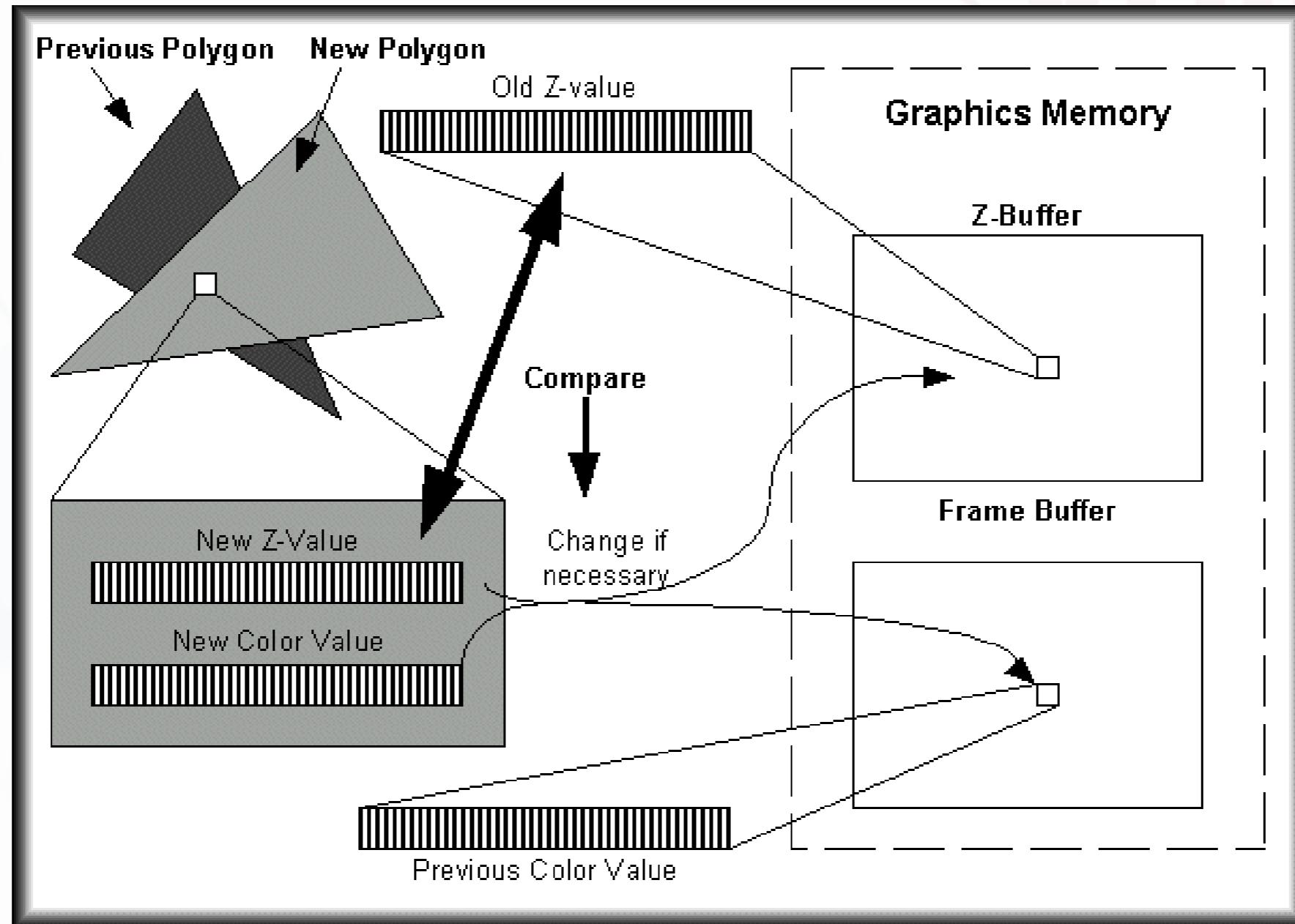


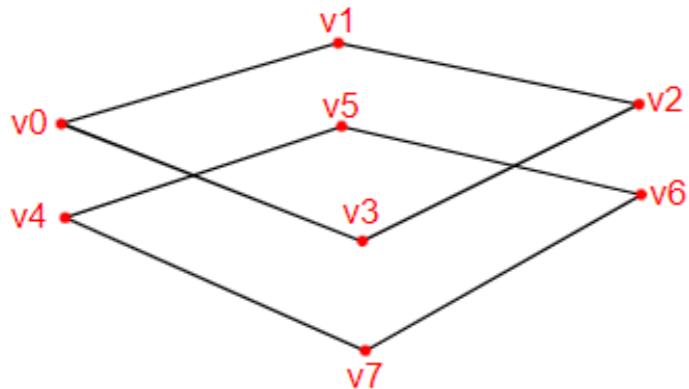
Depth Buffering – VulkanTutorial\27_depth_buffering



https://vulkan-tutorial.com/Depth_buffering

Depth-ordering of fragments via the Depth Buffer





Use Z coordinates of `-0.5f` and add the appropriate indices for the extra square:

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 0.0f}, {0.0f, 1.0f}},

    {{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 0.0f}, {0.0f, 1.0f}}
};

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4
};
```

```
void createDepthResources() {
    VkFormat depthFormat = findDepthFormat();

    createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTIMAL,
               VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
    depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);
}

VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling, VkFormatFeatureFlags features) {
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
            return format;
        } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) == features) {
            return format;
        }
    }

    throw std::runtime_error("failed to find supported format!");
}

VkFormat findDepthFormat() {
    return findSupportedFormat(
{VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
        VK_IMAGE_TILING_OPTIMAL,
        VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
    );
}
```

Depth and stencil state

The depth attachment is ready to be used now, but depth testing still needs to be enabled in the graphics pipeline. It is configured through the `VkPipelineDepthStencilStateCreateInfo` struct:

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};  
depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;  
depthStencil.depthTestEnable = VK_TRUE;  
depthStencil.depthWriteEnable = VK_TRUE;
```

The `depthTestEnable` field specifies if the depth of new fragments should be compared to the depth buffer to see if they should be discarded. The `depthWriteEnable` field specifies if the new depth of fragments that pass the depth test should actually be written to the depth buffer.

```
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

The `depthCompareOp` field specifies the comparison that is performed to keep or discard fragments. We're sticking to the convention of lower depth = closer, so the depth of new fragments should be *less*.

Depth Attachment

```
void createRenderPass() {  
  
    VkAttachmentDescription depthAttachment{};  
    depthAttachment.format = VK_FORMAT_D32_SFLOAT;  
    depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;  
    depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;  
    depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;  
    depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;  
    depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;  
    depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;  
    depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;  
  
    VkAttachmentReference depthAttachmentRef{};  
    depthAttachmentRef.attachment = 1;  
    depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Depth Render Subpass Dependency

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
dependency.srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
```

Dependencies can be confusing. A dependency translates to this:

- Wait for work done (specified by **srcAccessMask**) in the pipeline stage specified by **srcStageMask**.
- Unblock work done in the pipeline stage specified by **dstStageMask** for operations specified by **dstAccessMask**.

For the depth attachment here:

- We wait on (i.e. are blocked by) the **srcStage**. As the depth attachment is shared across frames, we wait for the last stage that uses depth – **LATE_FRAGMENT_TESTS**. This will wait for the depth storeOp to finish.
- We block all depth operations in/after the **dstStage** i.e. we block the earliest use of depth until we are finished – **EARLY_FRAGMENT_TESTS**. This will block any loadOp until we are finished writing to the depth buffer.
- ***WARNING!** I have corrected the code in this sample, and it differs from the original vulkan-tutorial.com sample. The author's assumption that a single depth attachment can be shared across frames, as the render subpasses will not overlap is incorrect. The semaphores are per-frame and commands execute out-of-order in Vulkan. We need to explicitly synchronize via pipeline execution barriers, render subpass dependencies, and/or memory barriers wherever necessary. The simple solution is to just have per-frame resources, including the depth-buffer, unless resource availability is constrained on the target platform.*

DAY 3 - Agenda

- Review of Exercises, Q&A
- CUDA interop
 - Sample: vulkanImageCUDA
 - Sample: simpleVulkan
- Introduction to Vulkan Compute
 - Vulkan Compute Pipeline
 - Compute Queue Families
 - Compute Shaders
 - Shader Storage Buffers
 - Dispatch
 - Synchronization
 - Sample: GPU Particle System
- Q&A, Exercises

Vulkan®

3Dapis
We Work In-Depth

CUDA-Vulkan Interop – CUDA\vulkanImageCUDA



Sample: vulkanImageCUDA

- This sample demonstrates Vulkan Image - CUDA Interop. CUDA imports the Vulkan image buffer, performs box filtering over it, and synchronizes with Vulkan through vulkan semaphores imported by CUDA.
- `setCudaVkDevice()`: calls `cudaSetDevice()` on the device already picked for Vulkan.
- `getKhrExtensionsFn()` retrieves the `vkGetSemaphoreWin32HandleKHR` extension function pointer
- `createSyncObjectsExt()`: uses Win32 extensions to create semaphores for export
- `cudaVkImportImageMem()`: uses `cudaImportExternalMemory()` to import memory exported by Vulkan (as Win32 handles)
- `cudaVkImportSemaphore()`: uses `cudaImportExternalSemaphore()` and `vkGetSemaphoreWin32HandleKHR()` to import semaphores exported by Vulkan (as Win32 handles)
- In `drawFrame()`:
 - `submitVulkan()` handles the first frame (loads the image and draws) without waiting for CUDA
 - `submitVulkanCuda()` handles subsequent frames by:
 - Waiting on `cudaUpdateVkSemaphore` so that the next box-filtered image data is available
 - Signaling via `vkUpdateCudaSemaphore` that CUDA may perform box-filtering on the current frame
- `cudaUpdateVkImage()` performs the box-filtering on the current frame by:
 - Using `cudaWaitExternalSemaphoresAsync` to wait on `cudaExtVkUpdateCudaSemaphore` (imported from Vulkan) i.e. until Vulkan has finished rendering the previous frame.
 - Launching the `d_boxfilter_rgba` kernel on the current frame
 - Using `cudaSignalExternalSemaphoresAsync` to signal `cudaExtCudaUpdateVkSemaphore` (imported from Vulkan) i.e. let Vulkan know that CUDA has finished filtering the current frame, and Vulkan may render it.

Using the Vulkan Mipmapped array in CUDA

- `cudaVkImportImageMem()` uses `cudaExternalMemoryGetMappedMipmappedArray()` to map a CUDA mipmapped array onto the imported Vulkan memory object.
- This is then used by `cudaCreateTextureObject()` to create `textureObjMipMapInput` for use as input for the filtering.
- `cudaGetMipmappedArrayLevel()` is used to retrieve each mipmapped level of the CUDA mipmapped array which is then used by `cudaCreateSurfaceObject()`
- `cudaMemcpy2DArrayToArray()` and `cudaCreateTextureObject()` are used to populate the `textureObjMipMapInput` `cudaTextureObject_t` for use as original image input texture for the kernel.
- Two `cudaSurfaceObject_t` lists are used:
 - `d_surfaceObjectListTemp` is the destination for the horizontal filter pass, using the `cudaTextureObject_t textureObjMipMapInput` as the source
 - `d_surfaceObjectList` is the destination for the vertical filter pass, using `d_surfaceObjectListTemp` as the source from the horizontal pass. As `d_surfaceObjectList` points to the shared external Vulkan texture, the rendered image shows the filtered output.
 - Hence the two `cudaMallocMipmappedArray()` calls – one for the horizontal filter pass, and the other to cache the original image.

Matching device UUIDs

- When importing memory and synchronization objects exported by Vulkan, they must be imported and mapped on the same device as they were created on.
- The CUDA device that corresponds to the Vulkan physical device on which the objects were created can be determined by comparing the UUID of a CUDA device with that of the Vulkan physical device, as in `setCudaVkDevice()`.

```
// Find the GPU which is selected by Vulkan
while (current_device < device_count) {
    cudaGetDeviceProperties(&deviceProp, current_device);

    if ((deviceProp.computeMode != cudaComputeModeProhibited)) {
        // Compare the cuda device UUID with vulkan UUID
        int ret = memcmp(&deviceProp.uuid, &vkDeviceUUID, VK_UUID_SIZE);
        if (ret == 0) {
            checkCudaErrors(cudaSetDevice(current_device));
            checkCudaErrors(cudaGetDeviceProperties(&deviceProp, current_device));
            printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n",
                   current_device, deviceProp.name, deviceProp.major,
                   deviceProp.minor);

            return current_device;
        }
    } else {
        devices_prohibited++;
    }
}
```

Importing Vulkan Memory Objects in CUDA

- A Vulkan memory object exported using `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT` can be imported into CUDA using the NT handle associated with that object as shown below. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT` specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

```
void cudaVkImportImageMem() {
    cudaExternalMemoryHandleDesc cudaExtMemHandleDesc;
    memset(&cudaExtMemHandleDesc, 0, sizeof(cudaExtMemHandleDesc));
    cudaExtMemHandleDesc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
    cudaExtMemHandleDesc.handle.win32.handle = getVkImageMemHandle(VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT);
    cudaExtMemHandleDesc.size = totalImageMemSize;

    checkCudaErrors(cudaImportExternalMemory(&cudaExtMemImageBuffer, &cudaExtMemHandleDesc));
```

Mapping Mipmapped Arrays onto Imported Memory Objects

- A CUDA mipmapped array can be mapped onto the imported memory object. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Vulkan API. Additionally, if the mipmapped array is bound as a color target in Vulkan, the flag `cudaArrayColorAttachment` must be set (we are only using it as a sampled texture in Vulkan).
- All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`

```
cudaExternalMemoryMipmappedArrayDesc externalMemoryMipmappedArrayDesc;

memset(&externalMemoryMipmappedArrayDesc, 0, sizeof(externalMemoryMipmappedArrayDesc));

cudaExtent extent = make_cudaExtent(imageWidth, imageHeight, 0);
cudaChannelFormatDesc formatDesc;
formatDesc.x = 8;
formatDesc.y = 8;
formatDesc.z = 8;
formatDesc.w = 8;
formatDesc.f = cudaChannelFormatKindUnsigned;

externalMemoryMipmappedArrayDesc.offset = 0;
externalMemoryMipmappedArrayDesc.formatDesc = formatDesc;
externalMemoryMipmappedArrayDesc.extent = extent;
externalMemoryMipmappedArrayDesc.flags = 0;
externalMemoryMipmappedArrayDesc.numLevels = mipLevels;

checkCudaErrors(cudaExternalMemoryGetMappedMipmappedArray(
    &cudaMipmappedImageArray, cudaExtMemImageBuffer,
    &externalMemoryMipmappedArrayDesc));
```

```
_host_ cudaError_t cudaGetMipmappedArrayLevel ( cudaArray_t* levelArray , cudaMipmappedArray_const_t mipmappedArray , unsigned int level )
```

Gets a mipmap level of a CUDA mipmapped array.

Parameters

levelArray

- Returned mipmap level CUDA array

mipmappedArray

- CUDA mipmapped array

level

- Mipmap level

Returns

[cudaSuccess](#), [cudaErrorInvalidValue](#)[cudaErrorInvalidResourceHandle](#)

Description

Returns in `*levelArray` a CUDA array that represents a single mipmap level of the CUDA mipmapped array `mipmappedArray`.

Importing Synchronization Objects

- A Vulkan semaphore object exported using `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT` can be imported into CUDA using the NT handle associated with that object as in `cudaVkImportSemaphore()`. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
void cudaVkImportSemaphore() {
    cudaExternalSemaphoreHandleDesc externalSemaphoreHandleDesc;
    memset(&externalSemaphoreHandleDesc, 0, sizeof(externalSemaphoreHandleDesc));
    externalSemaphoreHandleDesc.type = cudaExternalSemaphoreHandleTypeOpaqueWin32;
    externalSemaphoreHandleDesc.handle.win32.handle = getVkSemaphoreHandle(
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT,
        cudaUpdateVkSemaphore);
    externalSemaphoreHandleDesc.flags = 0;

    checkCudaErrors(cudaImportExternalSemaphore(&cudaExtCudaUpdateVkSemaphore, &externalSemaphoreHandleDesc));

    memset(&externalSemaphoreHandleDesc, 0, sizeof(externalSemaphoreHandleDesc));
    externalSemaphoreHandleDesc.type = cudaExternalSemaphoreHandleTypeOpaqueWin32;
    externalSemaphoreHandleDesc.handle.win32.handle = getVkSemaphoreHandle(
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT,
        vkUpdateCudaSemaphore);
    externalSemaphoreHandleDesc.flags = 0;
    checkCudaErrors(cudaImportExternalSemaphore(&cudaExtVkUpdateCudaSemaphore, &externalSemaphoreHandleDesc));
    printf("CUDA Imported Vulkan semaphore\n");
}
```

Waiting on Imported Synchronization Objects

- An imported Vulkan semaphore object can be waited on as in `cudaUpdateVkImage()`. Waiting on such a semaphore object waits until it reaches the signaled state and then resets it back to the unsignaled state. The corresponding signal that this wait is waiting on must be issued in Vulkan as in `submitVulkanCuda()`. Additionally, the signal must be issued before this wait can be issued

```
void submitVulkanCuda(uint32_t imageIndex) {
    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    VkSemaphore waitSemaphores[] = {imageAvailableSemaphores[currentFrame], cudaUpdateVkSemaphore};
    VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, VK_PIPELINE_STAGE_ALL_COMMANDS_BIT};
    submitInfo.waitSemaphoreCount = 2;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffers[imageIndex];

    VkSemaphore signalSemaphores[] = {renderFinishedSemaphores[currentFrame], vkUpdateCudaSemaphore};

    submitInfo.signalSemaphoreCount = 2;
    submitInfo.pSignalSemaphores = signalSemaphores;

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
        throw std::runtime_error("failed to submit draw command buffer!");
    }
}
```

Signaling on Imported Synchronization Objects

- An imported Vulkan semaphore object can be signaled as in `cudaUpdateVkImage()`. Signaling such a semaphore object sets it to the signaled state. The corresponding wait that waits on this signal must be issued in Vulkan. Additionally, the wait that waits on this signal must be issued after this signal has been issued.

```
void cudaUpdateVkImage() {
    cudaVkSemaphoreWait(cudaExtVkUpdateCudaSemaphore);

    int nthreads = 128;

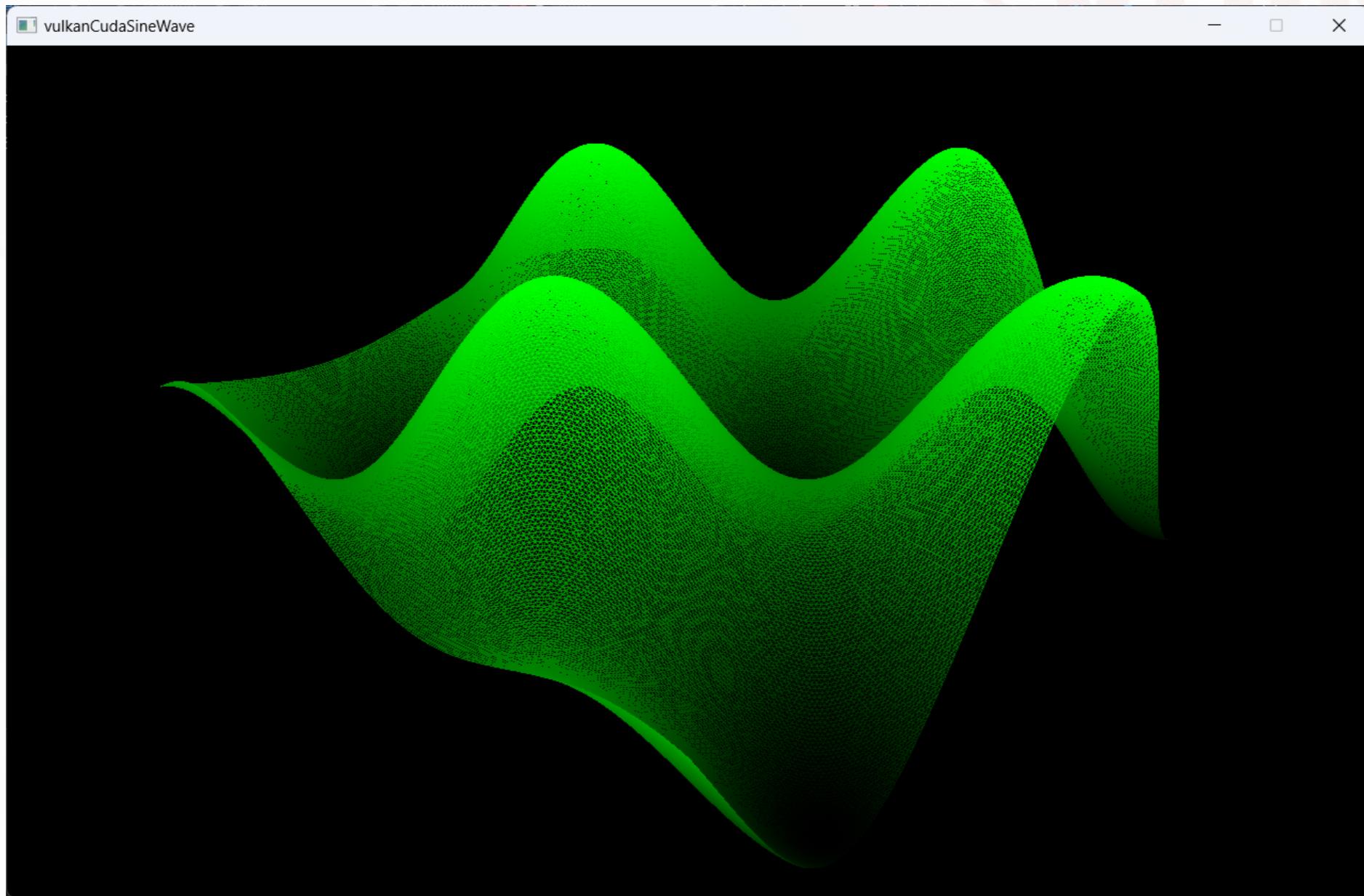
    /*Perform 2D box filter on image using CUDA */
    d_boxfilter_rgba_x<<<imageHeight / nthreads, nthreads, 0, streamToRun>>>(
        d_surfaceObjectListTemp, textureObjMipMapInput, imageWidth, imageHeight,
        mipLevels, filter_radius);

    d_boxfilter_rgba_y<<<imageWidth / nthreads, nthreads, 0, streamToRun>>>(
        d_surfaceObjectList, d_surfaceObjectListTemp, imageWidth, imageHeight,
        mipLevels, filter_radius);

    varySigma();

    cudaVkSemaphoreSignal(cudaExtCudaUpdateVkSemaphore);
}
```

CUDA-Vulkan Interop – CUDA\simpleVulkan



```
void drawFrame() {
    static chrono_tp startTime = std::chrono::high_resolution_clock::now();

    chrono_tp currentTime = std::chrono::high_resolution_clock::now();
    float time = std::chrono::duration<float, std::chrono::seconds::period>(currentTime - startTime).count();

    if (m_currentFrame == 0) {
        m_lastTime = startTime;
    }

    float frame_time = std::chrono::duration<float, std::chrono::seconds::period>(currentTime - m_lastTime).count();

    // Have vulkan draw the current frame...
    VulkanBaseApp::drawFrame();

    static uint64_t waitValue = 1;
    static uint64_t signalValue = 2;

    cudaExternalSemaphoreWaitParams waitParams = {};
    waitParams.flags = 0;
    waitParams.params.fence.value = waitValue;

    cudaExternalSemaphoreSignalParams signalParams = {};
    signalParams.flags = 0;
    signalParams.params.fence.value = signalValue;
    // Wait for vulkan to complete it's work
    checkCudaErrors(cudaWaitExternalSemaphoresAsync(&m_cudaTimelineSemaphore, &waitParams, 1, m_stream));
    // Now step the simulation
    m_sim.stepSimulation(time, m_stream);
    // Signal vulkan to continue with the updated buffers
    checkCudaErrors(cudaSignalExternalSemaphoresAsync(&m_cudaTimelineSemaphore, &signalParams, 1, m_stream));

    waitValue += 2;
    signalValue += 2;
```

```
__host__ cudaError_t cudaWaitExternalSemaphoresAsync ( const cudaExternalSemaphore_t* extSemArray , const
cudaExternalSemaphoreWaitParams* paramsArray , unsigned int numExtSems , cudaStream_t stream = 0 )
```

Waits on a set of external semaphore objects.

Parameters

extSemArray

- External semaphores to be waited on

paramsArray

- Array of semaphore parameters

numExtSems

- Number of semaphores to wait on

stream

- Stream to enqueue the wait operations in

Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#)[cudaErrorTimeout](#)

Description

Enqueues a wait operation on a set of externally allocated semaphore object in the specified stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of waiting on a semaphore depends on the type of the object.

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeOpaqueFd](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt](#) then waiting on the semaphore will wait until the semaphore reaches the signaled state. The semaphore will then be reset to the unsignaled state. Therefore for every signal operation, there can only be one wait operation.

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeD3D12Fence](#), [cudaExternalSemaphoreHandleTypeD3D11Fence](#), [cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd](#),

[cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32](#) then waiting on the semaphore will wait until the value of the semaphore is greater than or equal to `cudaExternalSemaphoreWaitParams::params::fence::value`.

```
__host__ cudaError_t cudaSignalExternalSemaphoresAsync ( const cudaExternalSemaphore_t* extSemArray , const
cudaExternalSemaphoreSignalParams* paramsArray , unsigned int numExtSems , cudaStream_t stream = 0 )
```

Signals a set of external semaphore objects.

Parameters

extSemArray

- Set of external semaphores to be signaled

paramsArray

- Array of semaphore parameters

numExtSems

- Number of semaphores to signal

stream

- Stream to enqueue the signal operations in

Returns

[cudaSuccess](#), [cudaErrorInvalidResourceHandle](#)

Description

Enqueues a signal operation on a set of externally allocated semaphore object in the specified stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of signaling a semaphore depends on the type of the object.

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeOpaqueFd](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32](#), [cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt](#) then signaling the semaphore will set it to the signaled state.

If the semaphore object is any one of the following types: [cudaExternalSemaphoreHandleTypeD3D12Fence](#), [cudaExternalSemaphoreHandleTypeD3D11Fence](#), [cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd](#), [cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32](#) then the semaphore will be set to the value specified in `cudaExternalSemaphoreSignalParams::params::fence::value`.

```
void VulkanBaseApp::drawFrame() {
    static uint64_t waitValue = 0;
    static uint64_t signalValue = 1;

    VkSemaphoreWaitInfo semaphoreWaitInfo = {};
    semaphoreWaitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
    semaphoreWaitInfo.pSemaphores = &m_vkTimelineSemaphore;
    semaphoreWaitInfo.semaphoreCount = 1;
    semaphoreWaitInfo.pValues = &waitValue;
    vkWaitSemaphores(m_device, &semaphoreWaitInfo, std::numeric_limits<uint64_t>::max());

    uint32_t imageIndex;
    VkResult result = vkAcquireNextImageKHR(
        m_device, m_swapChain, std::numeric_limits<uint64_t>::max(),
        m_vkPresentationSemaphore, VK_NULL_HANDLE, &imageIndex);
    if (result == VK_ERROR_OUT_OF_DATE_KHR) {
        recreateSwapChain();
    } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
        throw std::runtime_error("Failed to acquire swap chain image!");
    }

    updateUniformBuffer(imageIndex);
```

The fence from our earlier examples has been replaced by a timeline semaphore.

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

std::vector<VkSemaphore> waitSemaphores;
std::vector<VkPipelineStageFlags> waitStages;

waitSemaphores.push_back(m_vkTimelineSemaphore);
waitStages.push_back(VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT);

submitInfo.waitSemaphoreCount = (uint32_t)waitSemaphores.size();
submitInfo.pWaitSemaphores = waitSemaphores.data();
submitInfo.pWaitDstStageMask = waitStages.data();

submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &m_commandBuffers[imageIndex];
```

The wait-to-render semaphore from our earlier examples has been replaced by the same timeline semaphore used for the image acquisition wait.

```

std::vector<VkSemaphore> signalSemaphores;
signalSemaphores.push_back(m_vkTimelineSemaphore);
submitInfo.signalSemaphoreCount = (uint32_t)signalSemaphores.size();
submitInfo.pSignalSemaphores = signalSemaphores.data();

VkTimelineSemaphoreSubmitInfo timelineInfo = {};
timelineInfo.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
timelineInfo.waitSemaphoreValueCount = 1;
timelineInfo.pWaitSemaphoreValues = &waitForValue;
timelineInfo.signalSemaphoreValueCount = 1;
timelineInfo.pSignalSemaphoreValues = &signalValue;

submitInfo.pNext = &timelineInfo;

if (vkQueueSubmit(m_graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}

```

The signal-after-render semaphore from our earlier examples has been replaced by the same timeline semaphore used for wait-to-render i.e. we have replaced the fence and both semaphores with a single shared timeline semaphore.

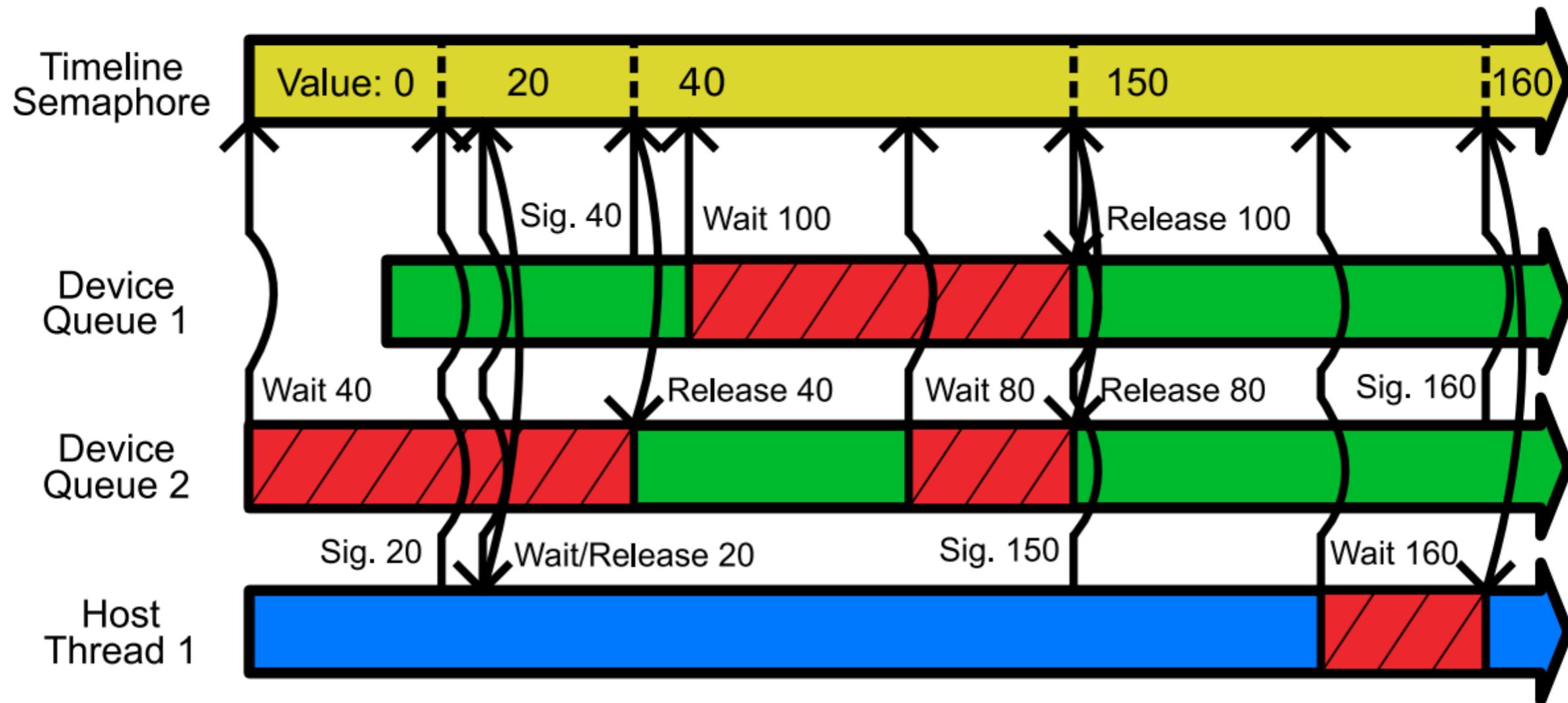
Timeline Semaphores

Original Vulkan Synchronization Model

- Two Coarse-Grained Primitives: VkSemaphore and VkFence
- VkSemaphore: Device->Device Synchronization
 - Binary State
 - Auto-Reset - 1:1 signal:wait relationship
 - Queue operations wait on and signal an arbitrary number of semaphores
 - Reusable, but only in the unsignaled state
 - Signal must be queued before wait is queued
- VkFence: Device->Host Synchronization
 - Binary State
 - Manual Reset – 1:<N> signal:wait relationship
 - Queue operations signal at most one fence
 - Reusable, but only in the unsignaled state

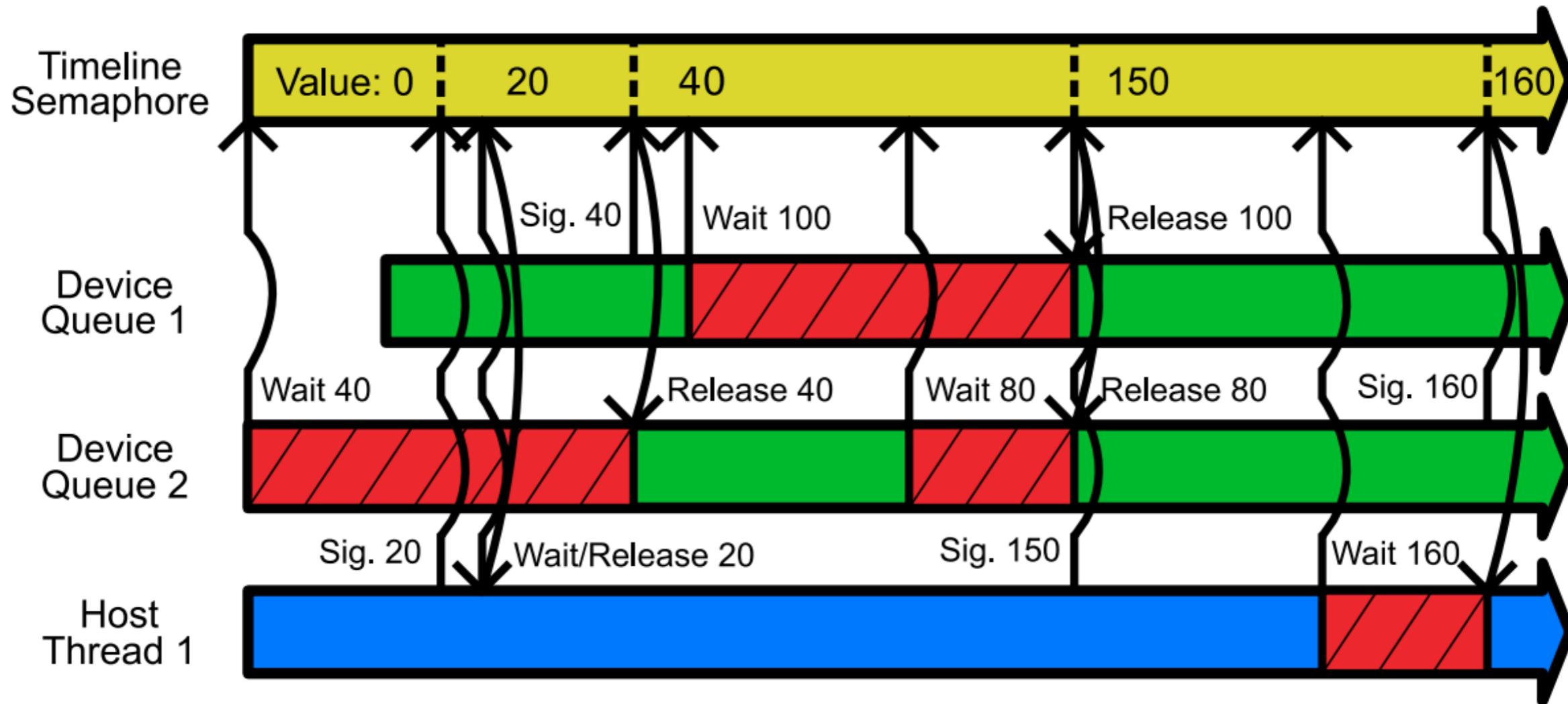
New Model: Timeline Semaphore Primitive

- From ~10 sync primitives to 1
 - Some of these operations not directly expressible with existing sync primitives



New Model: Timeline Semaphore Primitive

- Allows Multiple In-Flight Signals and Asynchronous Waits on One Semaphore
 - Subsequent signals do not impact existing or future waits on prior signals



What Is A Vulkan Timeline Semaphore?

- Extends Existing VkSemaphore API
 - Supported by all core VkQueue operations that use VkSemaphore objects
 - Export and Import across processes or APIs using existing VkSemaphore APIs
- Functional Superset of Both Binary-Type VkSemaphores and VkFence
- 64-bit Monotonically Increasing Counter Replaces Binary VkSemaphore State
 - Values can now be descriptive, e.g. a monotonic timestamp, frame count, etc.
- New APIs to Signal and Wait From Host Threads
- Broad OS Support
 - Initially Windows 7 through 10, Linux, Android

Remaining Limitations and Compromises

- **No Window-System Integration API Support**
 - `vkQueuePresentKHR()` and `vkAcquireNextImageKHR()` not supported
 - The OS/Window System infrastructure is not ready everywhere yet
 - Several API-semantics issues to work through as well (Input Welcome!)
- **Import/Export Not a Required Feature**
 - Relies on kernel-level support that is not available everywhere yet
 - Works on Windows 10+, and Linux/Android devices with newer kernels/drivers
- **64-bit Values, but Sometimes Only 31-bit Range Between Outstanding Operations**
 - Allows Implementations to hide wrapping when using 32-bit HW or OS primitives
 - Still allows use of full 64-bit range if gap between signals and waits is reasonable

```
if (vkQueueSubmit(m_graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}

VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &m_vkPresentationSemaphore;

VkSwapchainKHR swapChains[] = {m_swapChain};
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;

result = vkQueuePresentKHR(m_presentQueue, &presentInfo);
if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR || m_framebufferResized) {
    recreateSwapChain();
    m_framebufferResized = false;
} else if (result != VK_SUCCESS) {
    throw std::runtime_error("Failed to acquire swap chain image!");
}

m_currentFrame++;

waitValue += 2;
signalValue += 2;
```

To reuse the timeline semaphore in the next frame, the wait and signal values are incremented.

To signal a semaphore created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` with a particular counter value, on the host, call:

```
// Provided by VK_VERSION_1_2
VkResult vkSignalSemaphore(
    VkDevice                                     device,
    const VkSemaphoreSignalInfo*                 pSignalInfo);
```

The `VkSemaphoreSignalInfo` structure is defined as:

```
// Provided by VK_VERSION_1_2
typedef struct VkSemaphoreSignalInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkSemaphore          semaphore;
    uint64_t              value;
} VkSemaphoreSignalInfo;
```

C++

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &m_vkPresentationSemaphore;

VkSwapchainKHR swapChains[] = {m_swapChain};
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;

result = vkQueuePresentKHR(m_presentQueue, &presentInfo);
if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR || m_framebufferResized) {
    recreateSwapChain();
    m_framebufferResized = false;
} else if (result != VK_SUCCESS) {
    throw std::runtime_error("Failed to acquire swap chain image!");
}

m_currentFrame++;

waitForValue += 2;
signalValue += 2;
}
```

```
void initVulkanApp() {
    m_sim.initCudaLaunchConfig(cuda_device);

    // Create the cuda stream we'll be using
    checkCudaErrors(cudaStreamCreateWithFlags(&m_stream, cudaStreamNonBlocking));

    const size_t nVerts = m_sim.getWidth() * m_sim.getHeight();
    const size_t nInds = (m_sim.getWidth() - 1) * (m_sim.getHeight() - 1) * 6;

    // Create the height map cuda will write to
    createExternalBuffer(
        nVerts * sizeof(float),
        VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT,
        m_heightBuffer, m_heightMemory);

    // Create the vertex buffer that will hold the xy coordinates for the grid
    createBuffer(nVerts * sizeof(vec2), VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, m_xyBuffer, m_xyMemory);

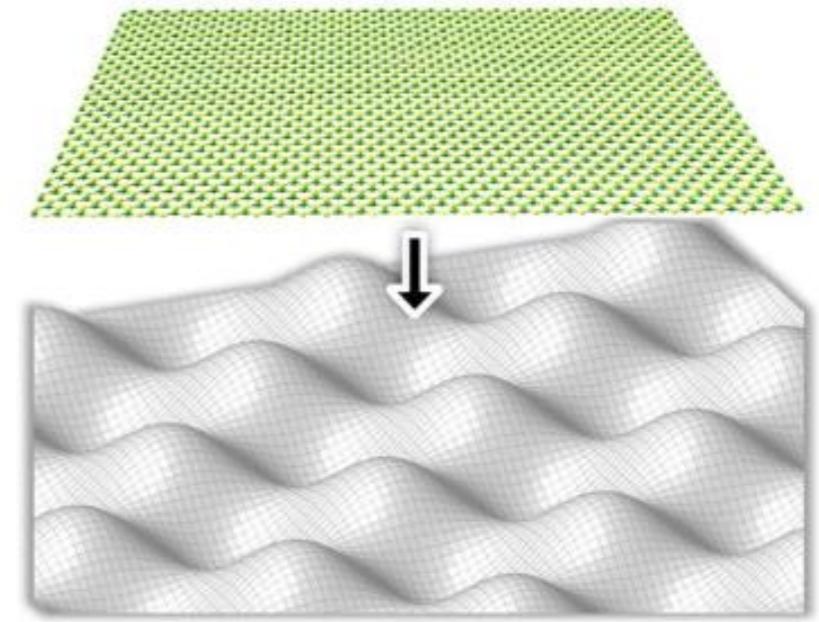
    // Create the index buffer that references from both buffers above
    createBuffer(
        nInds * sizeof(uint32_t),
        VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, m_indexBuffer, m_indexMemory);

    // Import the height map into cuda and retrieve a device pointer to use
    importCudaExternalMemory((void **)&m_cudaHeightMap, m_cudaVertMem, m_heightMemory, nVerts * sizeof(*m_cudaHeightMap),
        VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT);
    // Set the height map to use in the simulation
    m_sim.initSimulation(m_cudaHeightMap);
```

```
// Import the height map into cuda and retrieve a device pointer to use
importCudaExternalMemory((void **)&m_cudaHeightMap, m_cudaVertMem, m_heightMemory, nVerts * sizeof(*m_cudaHeightMap),
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT);
// Set the height map to use in the simulation
m_sim.initSimulation(m_cudaHeightMap);
```

```
void SineWaveSimulation::initSimulation(float *heights) {
    m_heightMap = heights;
}

void SineWaveSimulation::stepSimulation(float time, cudaStream_t stream) {
    sinewave<<~m_blocks, m_threads, 0, stream>>(m_heightMap, m_width, m_height, time);
    getLastCudaError("Failed to launch CUDA simulation");
}
```



The sinewave kernel sets the height every step, so the vertices move up and down (z coordinate) sinusoidally.

Initialize and populate Vertex Buffer (XY grid)

```
// Set up the initial values for the vertex buffers with Vulkan
void *stagingBase;
VkBuffer stagingBuffer;
VkDeviceMemory stagingMemory;
VkDeviceSize stagingSz = std::max(nVerts * sizeof(vec2), nInds * sizeof(uint32_t));
createBuffer(stagingSz, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
             VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
             stagingBuffer, stagingMemory);

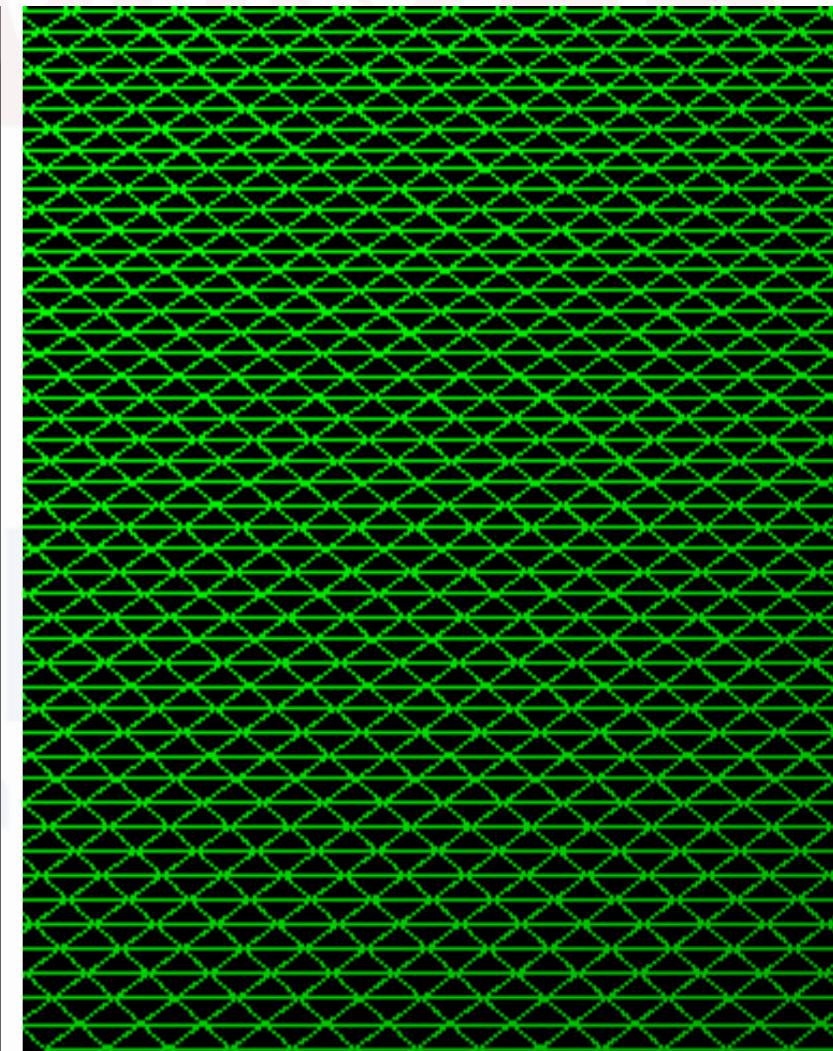
vkMapMemory(m_device, stagingMemory, 0, stagingSz, 0, &stagingBase);

memset(stagingBase, 0, nVerts * sizeof(float));
copyBuffer(m_heightBuffer, stagingBuffer, nVerts * sizeof(float));

for (size_t y = 0; y < m_sim.getHeight(); y++) {
    for (size_t x = 0; x < m_sim.getWidth(); x++) {
        vec2 *stagedVert = (vec2 *)stagingBase;
        stagedVert[y * m_sim.getWidth() + x][0] = (2.0f * x) / (m_sim.getWidth() - 1) - 1;
        stagedVert[y * m_sim.getWidth() + x][1] = (2.0f * y) / (m_sim.getHeight() - 1) - 1;
    }
}
copyBuffer(m_xyBuffer, stagingBuffer, nVerts * sizeof(vec2));
```

Populate Index Buffer

```
{  
    uint32_t *indices = (uint32_t *)stagingBase;  
    for (size_t y = 0; y < m_sim.getHeight() - 1; y++) {  
        for (size_t x = 0; x < m_sim.getWidth() - 1; x++) {  
            indices[0] = (uint32_t)((y + 0) * m_sim.getWidth() + (x + 0));  
            indices[1] = (uint32_t)((y + 1) * m_sim.getWidth() + (x + 0));  
            indices[2] = (uint32_t)((y + 0) * m_sim.getWidth() + (x + 1));  
            indices[3] = (uint32_t)((y + 1) * m_sim.getWidth() + (x + 0));  
            indices[4] = (uint32_t)((y + 1) * m_sim.getWidth() + (x + 1));  
            indices[5] = (uint32_t)((y + 0) * m_sim.getWidth() + (x + 1));  
            indices += 6;  
        }  
    }  
    copyBuffer(m_indexBuffer, stagingBuffer, nInds * sizeof(uint32_t));  
  
    vkUnmapMemory(m_device, stagingMemory);  
    vkDestroyBuffer(m_device, stagingBuffer, nullptr);  
    vkFreeMemory(m_device, stagingMemory, nullptr);  
}
```

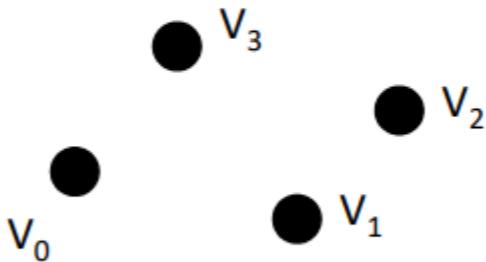


For each cell in the XY grid, two triangles are added to the index buffer.

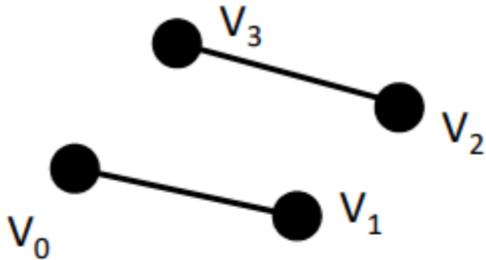
Primitive Topology

Vulkan Topologies

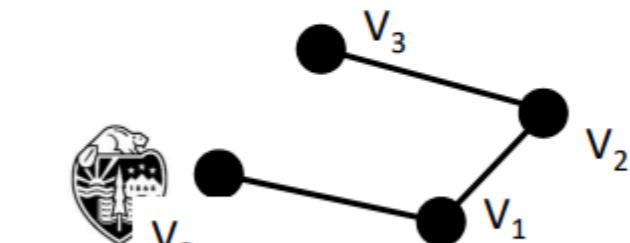
`VK_PRIMITIVE_TOPOLOGY_POINT_LIST`



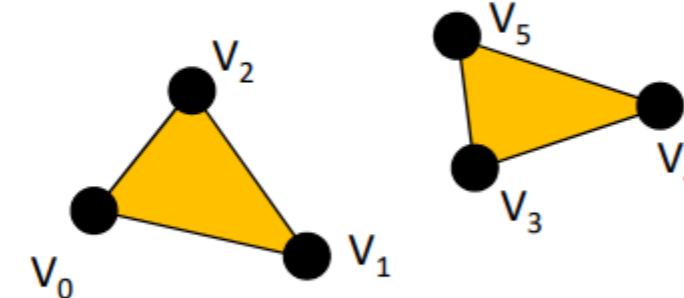
`VK_PRIMITIVE_TOPOLOGY_LINE_LIST`



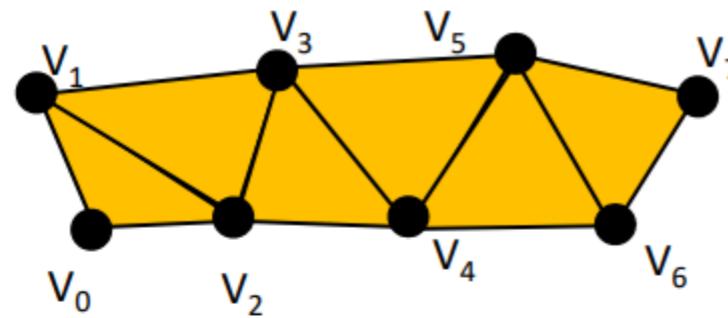
`VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`



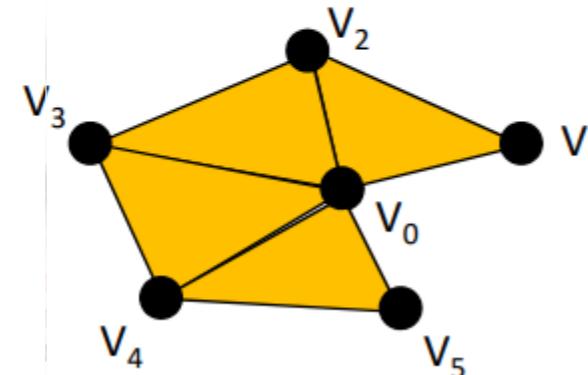
`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`



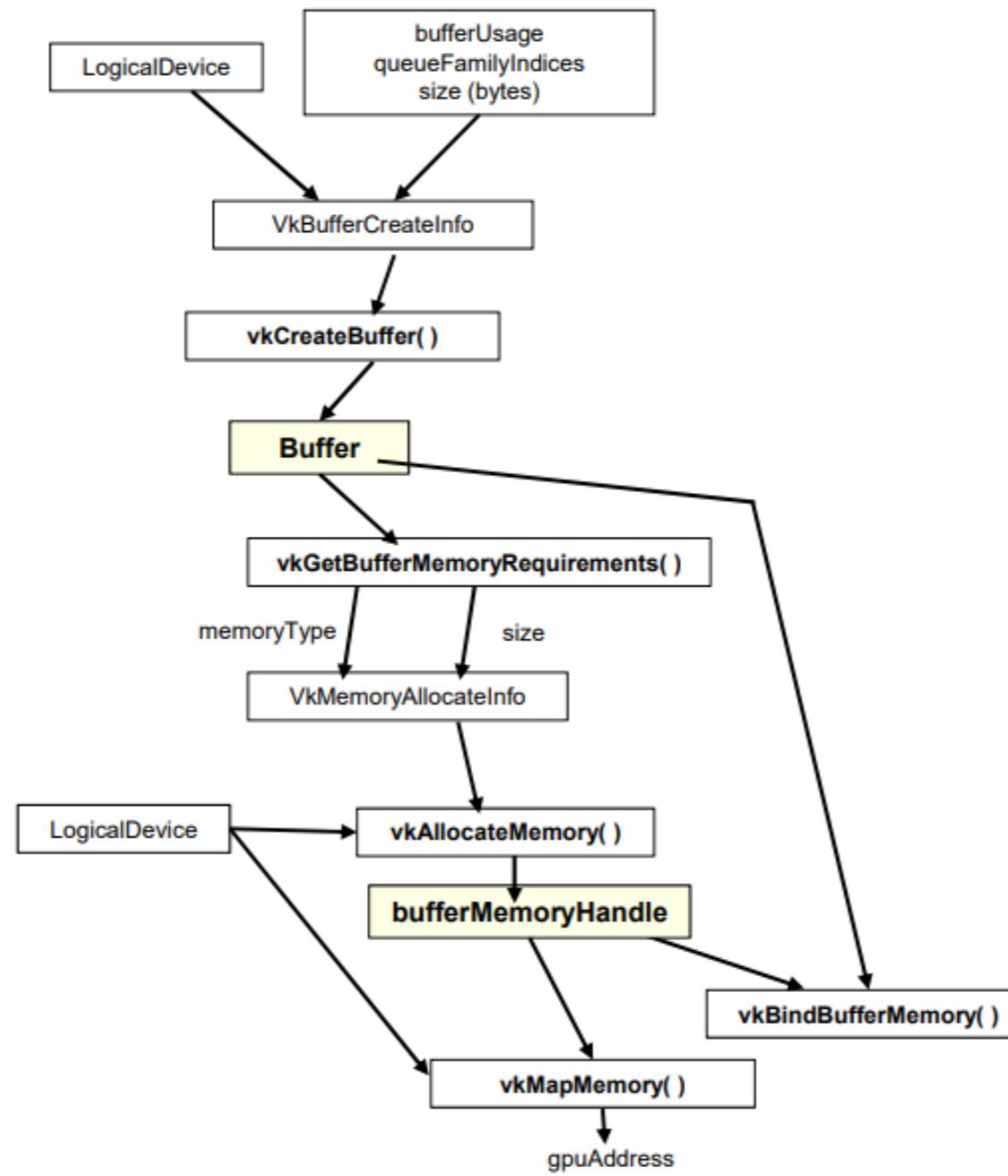
`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`



`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`



Creating and Filling Vulkan Data Buffers



Vertex Shader

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 modelViewProj;
} ubo;

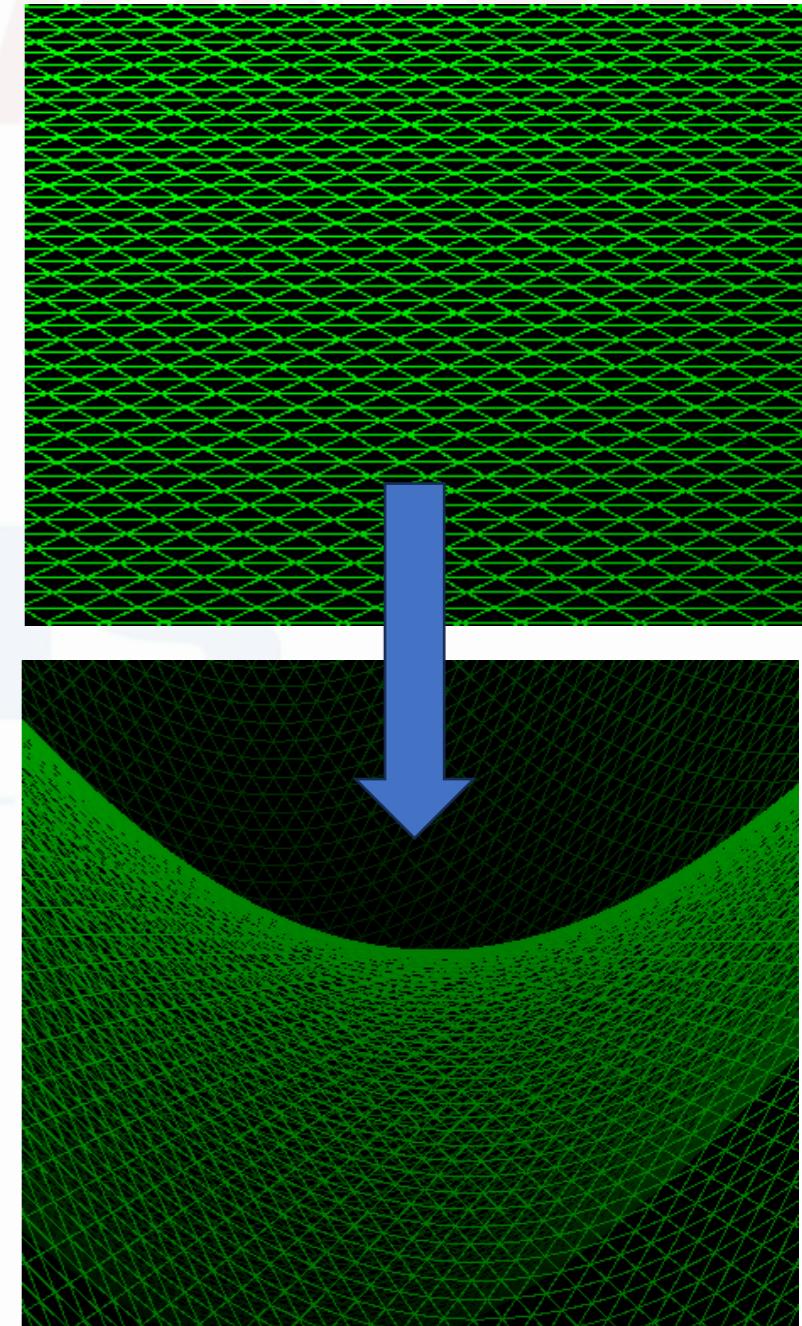
layout(location = 0) in float height;
layout(location = 1) in vec2 xyPos;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = ubo.modelViewProj * vec4(xyPos.xy, height, 1.0f);
    fragColor = vec3(0.0f, (height + 0.5f), 0.0f);
}
```

The Z value comes into the vertex shader from the CUDA height field, and the surface undulates sinusoidally.

Note: `GL_ARB_separate_shader_objects` is no longer required



```
void VulkanBaseApp::createExternalBuffer(
    VkDeviceSize size, VkBufferUsageFlags usage,
    VkMemoryPropertyFlags properties,
    VkExternalMemoryHandleTypeFlagsKHR extMemHandleType, VkBuffer &buffer,
    VkDeviceMemory &bufferMemory) {
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    VkExternalMemoryBufferCreateInfo externalMemoryBufferInfo = {};
    externalMemoryBufferInfo.sType = VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO;
    externalMemoryBufferInfo.handleTypes = extMemHandleType;
    bufferInfo.pNext = &externalMemoryBufferInfo;

    if (vkCreateBuffer(m_device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }
```

```
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(m_device, buffer, &memRequirements);

WindowsSecurityAttributes winSecurityAttributes;

VkExportMemoryWin32HandleInfoKHR vulkanExportMemoryWin32HandleInfoKHR = {};
vulkanExportMemoryWin32HandleInfoKHR.sType = VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR;
vulkanExportMemoryWin32HandleInfoKHR.pNext = NULL;
vulkanExportMemoryWin32HandleInfoKHR.pAttributes = &winSecurityAttributes;
vulkanExportMemoryWin32HandleInfoKHR.dwAccess = DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE;
vulkanExportMemoryWin32HandleInfoKHR.name = (LPCWSTR)NULL;
VkExportMemoryAllocateInfoKHR vulkanExportMemoryAllocateInfoKHR = {};
vulkanExportMemoryAllocateInfoKHR.sType = VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHR;
vulkanExportMemoryAllocateInfoKHR.pNext = extMemHandleType & VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR
? &vulkanExportMemoryWin32HandleInfoKHR
: NULL;
vulkanExportMemoryAllocateInfoKHR.handleTypes = extMemHandleType;

VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.pNext = &vulkanExportMemoryAllocateInfoKHR;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(m_physicalDevice, memRequirements.memoryTypeBits, properties);

if (vkAllocateMemory(m_device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
| throw std::runtime_error("failed to allocate external buffer memory!");
}

vkBindBufferMemory(m_device, buffer, bufferMemory, 0);
```

```
void *VulkanBaseApp::getMemHandle(VkDeviceMemory memory, VkExternalMemoryHandleTypeFlagBits handleType) {
    HANDLE handle = 0;

    VkMemoryGetWin32HandleInfoKHR vkMemoryGetWin32HandleInfoKHR = {};
    vkMemoryGetWin32HandleInfoKHR.sType = VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR;
    vkMemoryGetWin32HandleInfoKHR.pNext = NULL;
    vkMemoryGetWin32HandleInfoKHR.memory = memory;
    vkMemoryGetWin32HandleInfoKHR.handleType = handleType;

    PFN_vkGetMemoryWin32HandleKHR fpGetMemoryWin32HandleKHR;
    fpGetMemoryWin32HandleKHR = (PFN_vkGetMemoryWin32HandleKHR)vkGetDeviceProcAddr(m_device, "vkGetMemoryWin32HandleKHR");
    if (!fpGetMemoryWin32HandleKHR) {
        throw std::runtime_error("Failed to retrieve vkGetMemoryWin32HandleKHR!");
    }
    if (fpGetMemoryWin32HandleKHR(m_device, &vkMemoryGetWin32HandleInfoKHR, &handle) != VK_SUCCESS) {
        throw std::runtime_error("Failed to retrieve handle for buffer!");
    }
    return (void *)handle;
}
```

To export a Windows handle representing the payload of a Vulkan device memory object, call:

```
// Provided by VK_KHR_external_memory_win32
VkResult vkGetMemoryWin32HandleKHR(
    VkDevice                                     device,
    const VkMemoryGetWin32HandleInfoKHR* pGetWin32HandleInfo,
    HANDLE*                                       pHandle);
```

Parameters

- `device` is the logical device that created the device memory being exported.
- `pGetWin32HandleInfo` is a pointer to a `VkMemoryGetWin32HandleInfoKHR` structure containing parameters of the export operation.
- `pHandle` will return the Windows handle representing the payload of the device memory object.

```
void importCudaExternalMemory(void **cudaPtr, cudaExternalMemory_t &cudaMem,
                             VkDeviceMemory &vkMem, VkDeviceSize size,
                             VkExternalMemoryHandleTypeFlagBits handleType) {
    cudaExternalMemoryHandleDesc externalMemoryHandleDesc = {};
    externalMemoryHandleDesc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
    externalMemoryHandleDesc.size = size;
    externalMemoryHandleDesc.handle.win32.handle = (HANDLE)getMemHandle(vkMem, handleType);

    checkCudaErrors(cudaImportExternalMemory(&cudaMem, &externalMemoryHandleDesc));

    cudaExternalMemoryBufferDesc externalMemBufferDesc = {};
    externalMemBufferDesc.offset = 0;
    externalMemBufferDesc.size = size;
    externalMemBufferDesc.flags = 0;

    checkCudaErrors(cudaExternalMemoryGetMappedBuffer(cudaPtr, cudaMem, &externalMemBufferDesc));
}
```

```
void VulkanBaseApp::createExternalSemaphore(VkSemaphore &semaphore, VkExternalSemaphoreHandleTypeFlagBits handleType) {
    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    VkExportSemaphoreCreateInfoKHR exportSemaphoreCreateInfo = {};
    exportSemaphoreCreateInfo.sType = VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR;

    VkSemaphoreTypeCreateInfo timelineCreateInfo;
    timelineCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO;
    timelineCreateInfo.pNext = NULL;
    timelineCreateInfo.semaphoreType = VK_SEMAPHORE_TYPE_TIMELINE;
    timelineCreateInfo.initialValue = 0;
    exportSemaphoreCreateInfo.pNext = &timelineCreateInfo;
    exportSemaphoreCreateInfo.handleTypes = handleType;
    semaphoreInfo.pNext = &exportSemaphoreCreateInfo;

    if (vkCreateSemaphore(m_device, &semaphoreInfo, nullptr, &semaphore) != VK_SUCCESS) {
        throw std::runtime_error("failed to create synchronization objects for a CUDA-Vulkan!");
    }
}
```

```
void *VulkanBaseApp::getSemaphoreHandle(VkSemaphore semaphore, VkExternalSemaphoreHandleTypeFlagBits handleType) {
    HANDLE handle;

    VkSemaphoreGetWin32HandleInfoKHR semaphoreGetWin32HandleInfoKHR = {};
    semaphoreGetWin32HandleInfoKHR.sType = VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR;
    semaphoreGetWin32HandleInfoKHR.pNext = NULL;
    semaphoreGetWin32HandleInfoKHR.semaphore = semaphore;
    semaphoreGetWin32HandleInfoKHR.handleType = handleType;

    PFN_vkGetSemaphoreWin32HandleKHR fpGetSemaphoreWin32HandleKHR;
    fpGetSemaphoreWin32HandleKHR = (PFN_vkGetSemaphoreWin32HandleKHR)vkGetDeviceProcAddr(m_device, "vkGetSemaphoreWin32HandleKHR");
    if (!fpGetSemaphoreWin32HandleKHR) {
        throw std::runtime_error("Failed to retrieve vkGetMemoryWin32HandleKHR!");
    }
    if (fpGetSemaphoreWin32HandleKHR(m_device, &semaphoreGetWin32HandleInfoKHR, &handle) != VK_SUCCESS) {
        throw std::runtime_error("Failed to retrieve handle for buffer!");
    }

    return (void *)handle;
}
```

```
void importCudaExternalSemaphore(  
    cudaExternalSemaphore_t &cudaSem, VkSemaphore &vkSem,  
    VkExternalSemaphoreHandleTypeFlagBits handleType) {  
    cudaExternalSemaphoreHandleDesc externalSemaphoreHandleDesc = {};  
  
    externalSemaphoreHandleDesc.type = cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32;  
    externalSemaphoreHandleDesc.handle.win32.handle = (HANDLE)getSemaphoreHandle(vkSem, handleType);  
    externalSemaphoreHandleDesc.flags = 0;  
  
    checkCudaErrors(cudaImportExternalSemaphore(&cudaSem, &externalSemaphoreHandleDesc));  
}
```

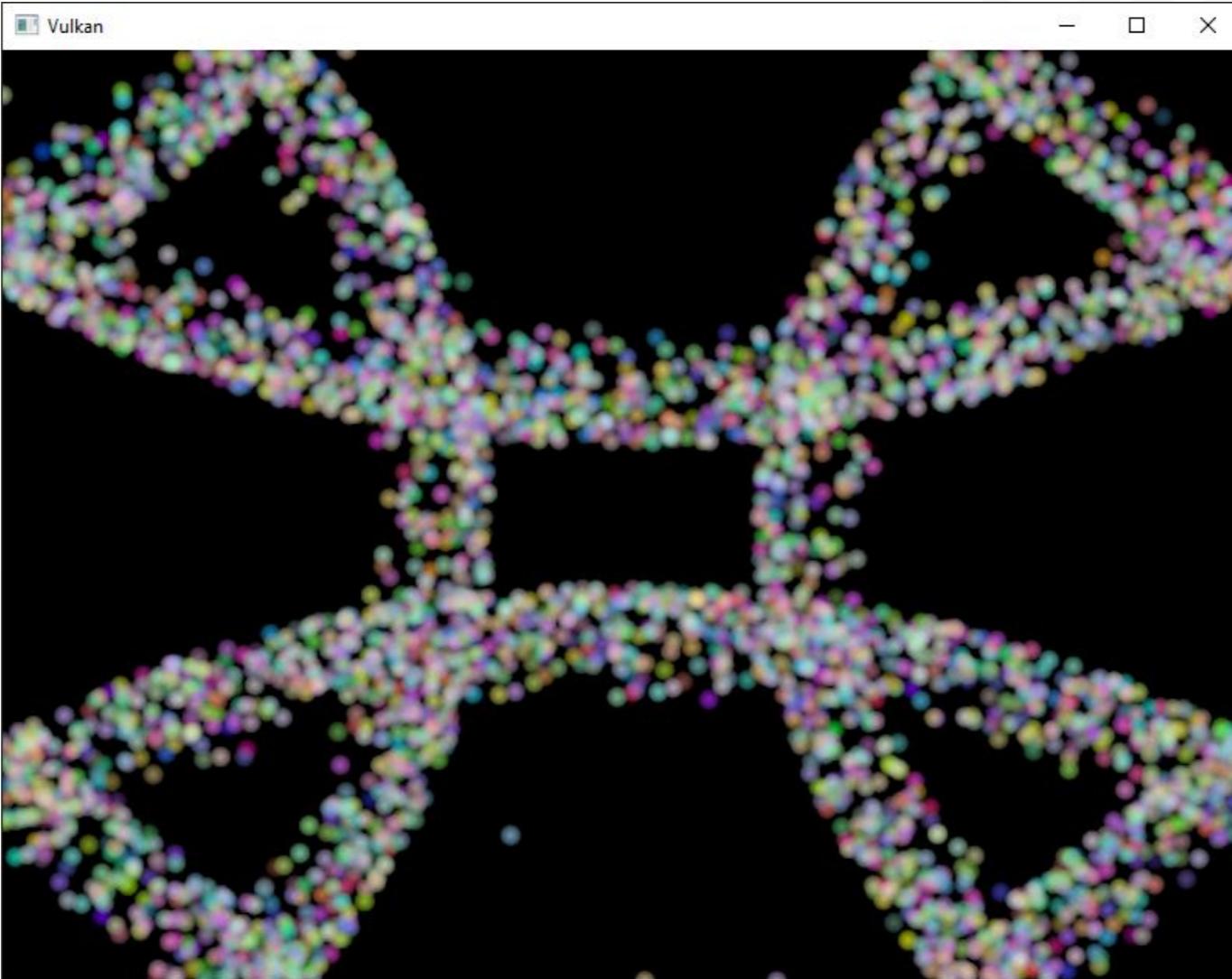


Compute Shader

- [Introduction](#)
- [Advantages](#)
- [The Vulkan pipeline](#)
- [An example](#)
- [Data manipulation](#)
 - [Shader storage buffer objects \(SSBO\)](#)
 - [Storage images](#)
- [Compute queue families](#)
- [The compute shader stage](#)
- [Loading compute shaders](#)
- [Preparing the shader storage buffers](#)
- [Descriptors](#)
- [Compute pipelines](#)
- [Compute space](#)
- [Compute shaders](#)
- [Running compute commands](#)
 - [Dispatch](#)
 - [Submitting work](#)
 - [Synchronizing graphics and compute](#)
- [Drawing the particle system](#)
- [Conclusion](#)

https://vulkan-tutorial.com/Compute_Shader

GPU Particles – VulkanTutorial\31_compute_shader



https://vulkan-tutorial.com/Compute_Shader

GPU Particle System



An easy to understand example that we will implement in this chapter is a GPU based particle system. Such systems are used in many games and often consist of thousands of particles that need to be updated at interactive frame rates. Rendering such a system requires 2 main components: vertices, passed as vertex buffers, and a way to update them based on some equation.

The "classical" CPU based particle system would store particle data in the system's main memory and then use the CPU to update them. After the update, the vertices need to be transferred to the GPU's memory again so it can display the updated particles in the next frame. The most straight-forward way would be recreating the vertex buffer with the new data for each frame. This is obviously very costly. Depending on your implementation, there are other options like mapping GPU memory so it can be written by the CPU (called "resizable BAR" on desktop systems, or unified memory on integrated GPUs) or just using a host local buffer (which would be the slowest method due to PCI-E bandwidth). But no matter what buffer update method you choose, you always require a "round-trip" to the CPU to update the particles.

With a GPU based particle system, this round-trip is no longer required. Vertices are only uploaded to the GPU at the start and all updates are done in the GPU's memory using compute shaders. One of the main reasons why this is faster is the much higher bandwidth between the GPU and its local memory. In a CPU based scenario, you'd be limited by main memory and PCI-express bandwidth, which is often just a fraction of the GPU's memory bandwidth.

Shader storage buffer objects (SSBO)

A shader storage buffer (SSBO) allows shaders to read from and write to a buffer. Using these is similar to using uniform buffer objects. The biggest differences are that you can alias other buffer types to SSBOs and that they can be arbitrarily large.

Going back to the GPU based particle system, you might now wonder how to deal with vertices being updated (written) by the compute shader and read (drawn) by the vertex shader, as both usages would seemingly require different buffer types.

But that's not the case. In Vulkan you can specify multiple usages for buffers and images. So for the particle vertex buffer to be used as a vertex buffer (in the graphics pass) and as a storage buffer (in the compute pass), you simply create the buffer with those two usage flags:

```
VkBufferCreateInfo bufferInfo{};  
...  
bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT;  
...  
  
if (vkCreateBuffer(device, &bufferInfo, nullptr, &shaderStorageBuffers[i]) != VK_SUCCESS) {  
    throw std::runtime_error("failed to create vertex buffer!");  
}
```

The two flags `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` and `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` set with `bufferInfo.usage` tell the implementation that we want to use this buffer for two different scenarios: as a vertex buffer in the vertex shader and as a store buffer. Note that we also added the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` flag in here so we can transfer data from the host to the GPU. This is crucial as we want the shader storage buffer to stay in GPU memory only (`VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`) we need to transfer data from the host to this buffer.

Here is the same code using the `createBuffer` helper function:

```
createBuffer(bufferSize, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, shaderStorageBuffers[i], shaderStorageBuffersMemory[i]);
```

The GLSL shader declaration for accessing such a buffer looks like this:

```
struct Particle {  
    vec2 position;  
    vec2 velocity;  
    vec4 color;  
};  
  
layout(std140, binding = 1) readonly buffer ParticleSSBOIn {  
    Particle particlesIn[ ];  
};  
  
layout(std140, binding = 2) buffer ParticleSSBOout {  
    Particle particlesOut[ ];  
};
```

In this example we have a typed SSBO with each particle having a position and velocity value (see the [Particle](#) struct). The SSBO then contains an unbound number of particles as marked by the `[]`. Not having to specify the number of elements in an SSBO is one of the advantages over e.g. uniform buffers. `std140` is a memory layout qualifier that determines how the member elements of the shader storage buffer are aligned in memory. This gives us certain guarantees, required to map the buffers between the host and the GPU.

Writing to such a storage buffer object in the compute shader is straight-forward and similar to how you'd write to the buffer on the C++ side:

```
particlesOut[index].position = particlesIn[index].position + particlesIn[index].velocity.xy * ubo.deltaTime;
```

Compute queue families

In the [physical device and queue families chapter](#) we already learned about queue families and how to select a graphics queue family. Compute uses the queue family properties flag bit `VK_QUEUE_COMPUTE_BIT`. So if we want to do compute work, we need to get a queue from a queue family that supports compute.

Note that Vulkan requires an implementation which supports graphics operations to have at least one queue family that supports both graphics and compute operations, but it's also possible that implementations offer a dedicated compute queue. This dedicated compute queue (that does not have the graphics bit) hints at an asynchronous compute queue. To keep this tutorial beginner friendly though, we'll use a queue that can do both graphics and compute operations. This will also save us from dealing with several advanced synchronization mechanisms.

For our compute sample we need to change the device creation code a bit:

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());

int i = 0;
for (const auto& queueFamily : queueFamilies) {
    if ((queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) && (queueFamily.queueFlags & VK_QUEUE_COMPUTE_BIT)) {
        indices.graphicsAndComputeFamily = i;
    }

    i++;
}
```

The compute shader stage

In the graphics samples we have used different pipeline stages to load shaders and access descriptors. Compute shaders are accessed in a similar way by using the `VK_SHADER_STAGE_COMPUTE_BIT` pipeline. So loading a compute shader is just the same as loading a vertex shader, but with a different shader stage. We'll talk about this in detail in the next paragraphs. Compute also introduces a new binding point type for descriptors and pipelines named `VK_PIPELINE_BIND_POINT_COMPUTE` that we'll have to use later on.

Loading compute shaders

Loading compute shaders in our application is the same as loading any other other shader. The only real difference is that we'll need to use the `VK_SHADER_STAGE_COMPUTE_BIT` mentioned above.

```
auto computeShaderCode = readFile("shaders/compute.spv");

VkShaderModule computeShaderModule = createShaderModule(computeShaderCode);

VkPipelineShaderStageCreateInfo computeShaderStageInfo{};
computeShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
computeShaderStageInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
computeShaderStageInfo.module = computeShaderModule;
computeShaderStageInfo.pName = "main";
...
```

Initialize the Particle System

```
// Initialize particles
std::default_random_engine rndEngine((unsigned)time(nullptr));
std::uniform_real_distribution<float> rndDist(0.0f, 1.0f);

// Initial particle positions on a circle
std::vector<Particle> particles(PARTICLE_COUNT);
for (auto& particle : particles) {
    float r = 0.25f * sqrt(rndDist(rndEngine));
    float theta = rndDist(rndEngine) * 2 * 3.14159265358979323846;
    float x = r * cos(theta) * HEIGHT / WIDTH;
    float y = r * sin(theta);
    particle.position = glm::vec2(x, y);
    particle.velocity = glm::normalize(glm::vec2(x,y)) * 0.00025f;
    particle.color = glm::vec4(rndDist(rndEngine), rndDist(rndEngine), rndDist(rndEngine), 1.0f);
}
```

We then create a **staging buffer** in the host's memory to hold the initial particle properties:

```
VkDeviceSize bufferSize = sizeof(Particle) * PARTICLE_COUNT;

VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;
createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

void* data;
vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
memcpy(data, particles.data(), (size_t)bufferSize);
vkUnmapMemory(device, stagingBufferMemory);
```

Using this staging buffer as a source we then create the per-frame shader storage buffers and copy the particle properties from the staging buffer to each of these:

```
for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
    createBuffer(bufferSize, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_B
IT | VK_BUFFER_USAGE_TRANSFER_DST_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, shaderStorageBuffers[i],
shaderStorageBuffersMemory[i]);
    // Copy data from the staging buffer (host) to the shader storage buffer (GPU)
    copyBuffer(stagingBuffer, shaderStorageBuffers[i], bufferSize);
}
```

Descriptors

Setting up descriptors for compute is almost identical to graphics. The only difference is that descriptors need to have the `VK_SHADER_STAGE_COMPUTE_BIT` set to make them accessible by the compute stage:

```
std::array<VkDescriptorSetLayoutBinding, 3> layoutBindings{};  
layoutBindings[0].binding = 0;  
layoutBindings[0].descriptorCount = 1;  
layoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;  
layoutBindings[0].pImmutableSamplers = nullptr;  
layoutBindings[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;  
...
```

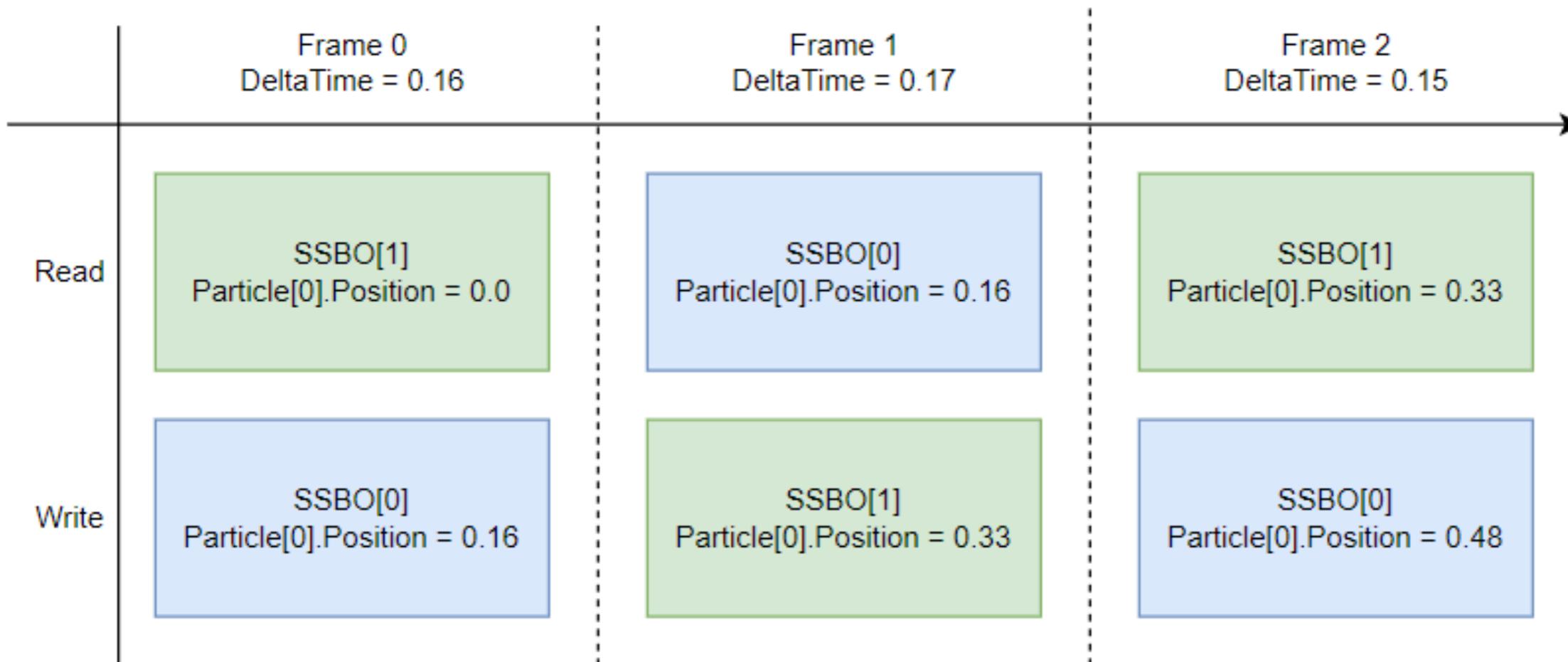
Note that you can combine shader stages here, so if you want the descriptor to be accessible from the vertex and compute stage, e.g. for a uniform buffer with parameters shared across them, you simply set the bits for both stages:

```
layoutBindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_COMPUTE_BIT;
```

Here is the descriptor setup for our sample. The layout looks like this:

```
std::array<VkDescriptorSetLayoutBinding, 3> layoutBindings{};  
layoutBindings[0].binding = 0;  
layoutBindings[0].descriptorCount = 1;  
layoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;  
layoutBindings[0].pImmutableSamplers = nullptr;  
layoutBindings[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;  
  
layoutBindings[1].binding = 1;  
layoutBindings[1].descriptorCount = 1;  
layoutBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;  
layoutBindings[1].pImmutableSamplers = nullptr;  
layoutBindings[1].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;  
  
layoutBindings[2].binding = 2;  
layoutBindings[2].descriptorCount = 1;  
layoutBindings[2].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;  
layoutBindings[2].pImmutableSamplers = nullptr;  
layoutBindings[2].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;  
  
VkDescriptorSetLayoutCreateInfo layoutInfo{};  
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;  
layoutInfo.bindingCount = 3;  
layoutInfo.pBindings = layoutBindings.data();  
  
if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &computeDescriptorsetLayout) != VK_SUCCESS) {  
    throw std::runtime_error("failed to create compute descriptor set layout!");  
}
```

Looking at this setup, you might wonder why we have two layout bindings for shader storage buffer objects, even though we'll only render a single particle system. This is because the particle positions are updated frame by frame based on a delta time. This means that each frame needs to know about the last frames' particle positions, so it can update them with a new delta time and write them to its own SSBO:



For that, the compute shader needs to have access to the last and current frame's SSBOs. This is done by passing both to the compute shader in our descriptor setup. See the `storageBufferInfoLastFrame` and `storageBufferInfoCurrentFrame`:

Compute pipelines

As compute is not a part of the graphics pipeline, we can't use `vkCreateGraphicsPipelines`. Instead we need to create a dedicated compute pipeline with `vkCreateComputePipelines` for running our compute commands. Since a compute pipeline does not touch any of the rasterization state, it has a lot less state than a graphics pipeline:

```
VkComputePipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
pipelineInfo.layout = computePipelineLayout;
pipelineInfo.stage = computeShaderStageInfo;

if (vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline) != VK_SUCCESS) {
    throw std::runtime_error("failed to create compute pipeline!");
}
```

The setup is a lot simpler, as we only require one shader stage and a pipeline layout. The pipeline layout works the same as with the graphics pipeline:

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = &computeDescriptorsetLayout;

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &computePipelineLayout) != VK_SUCCESS) {
    throw std::runtime_error("failed to create compute pipeline layout!");
}
```

```
#version 450

layout (binding = 0) uniform ParameterUBO {
    float deltaTime;
} ubo;

struct Particle {
    vec2 position;
    vec2 velocity;
    vec4 color;
};

layout(std140, binding = 1) readonly buffer ParticleSSBOIn {
    Particle particlesIn[ ];
};

layout(std140, binding = 2) buffer ParticleSSBOOut {
    Particle particlesOut[ ];
};

layout (local_size_x = 256, local_size_y = 1, local_size_z = 1) in;

void main()
{
    uint index = gl_GlobalInvocationID.x;

    Particle particleIn = particlesIn[index];

    particlesOut[index].position = particleIn.position + particleIn.velocity.xy * ubo.deltaTime;
    particlesOut[index].velocity = particleIn.velocity;
    ...
}
```

- Memory layout std140 is commonly used as it provides a standard layout with guaranteed packing order
- No need to query offsets of data members
- Array stride is always rounded up vec4 size (16 bytes)

Running compute commands

an®

Dispatch

Now it's time to actually tell the GPU to do some compute. This is done by calling `vkCmdDispatch` inside a command buffer. While not perfectly true, a dispatch is for compute as a draw call like `vkCmdDraw` is for graphics. This dispatches a given number of compute work items in at max. three dimensions.

```
VkCommandBufferBeginInfo beginInfo{};
beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

if (vkBeginCommandBuffer(commandBuffer, &beginInfo) != VK_SUCCESS) {
    throw std::runtime_error("failed to begin recording command buffer!");
}

...

vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
computeDescriptorSets[i], 0, 0);

vkCmdDispatch(computeCommandBuffer, PARTICLE_COUNT / 256, 1, 1);

...

if (vkEndCommandBuffer(commandBuffer) != VK_SUCCESS) {
    throw std::runtime_error("failed to record command buffer!");
}
```

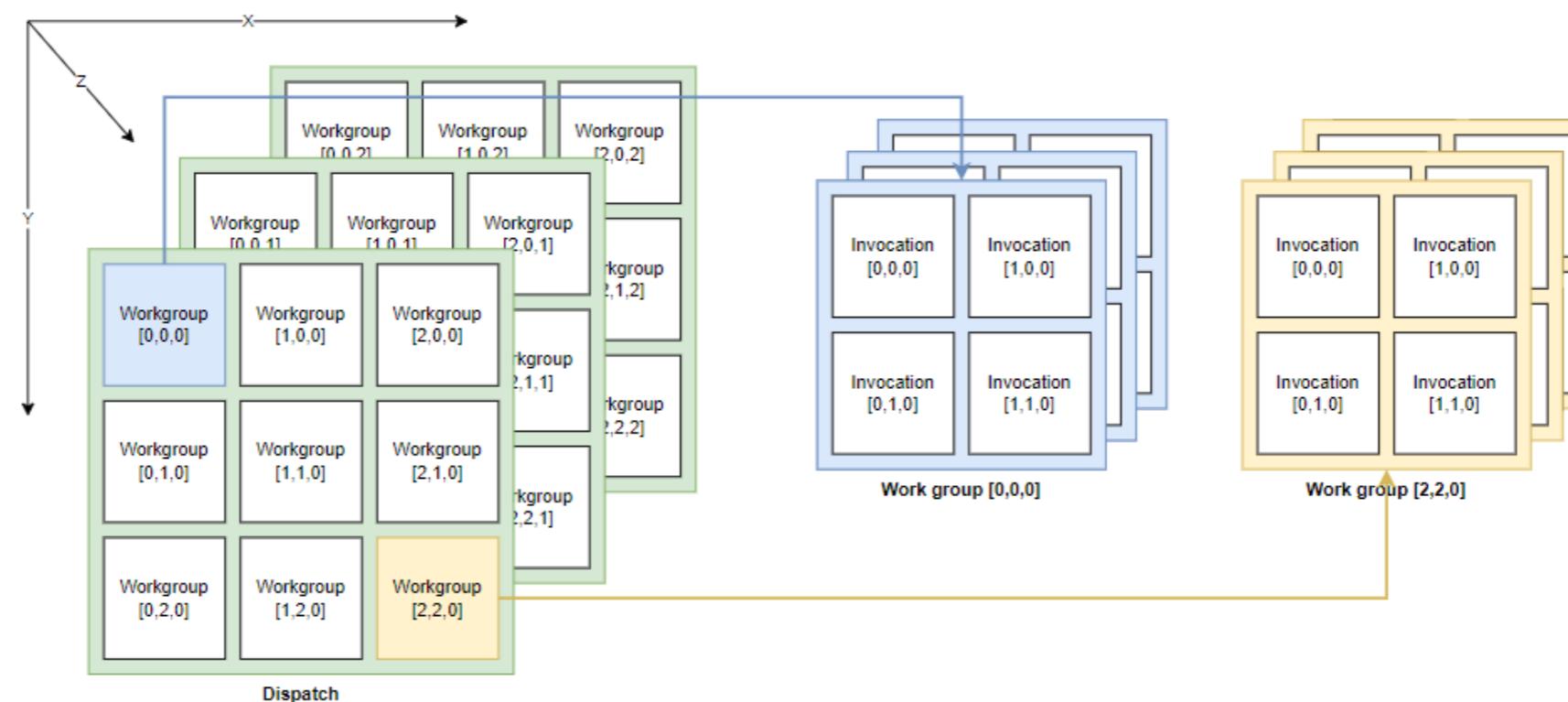
Compute space

Before we get into how a compute shader works and how we submit compute workloads to the GPU, we need to talk about two important compute concepts: **work groups** and **invocations**. They define an abstract execution model for how compute workloads are processed by the compute hardware of the GPU in three dimensions (x, y, and z).

Work groups define how the compute workloads are formed and processed by the the compute hardware of the GPU. You can think of them as work items the GPU has to work through. Work group dimensions are set by the application at command buffer time using a dispatch command.

And each work group then is a collection of **invocations** that execute the same compute shader. Invocations can potentially run in parallel and their dimensions are set in the compute shader. Invocations within a single workgroup have access to shared memory.

This image shows the relation between these two in three dimensions:



The `vkCmdDispatch` will dispatch `PARTICLE_COUNT / 256` local work groups in the x dimension. As our particles array is linear, we leave the other two dimensions at one, resulting in a one-dimensional dispatch. But why do we divide the number of particles (in our array) by 256? That's because in the previous paragraph we defined that every compute shader in a work group will do 256 invocations. So if we were to have 4096 particles, we would dispatch 16 work groups, with each work group running 256 compute shader invocations. Getting the two numbers right usually takes some tinkering and profiling, depending on your workload and the hardware you're running on. If your particle size would be dynamic and can't always be divided by e.g. 256, you can always use `gl_GlobalInvocationID` at the start of your compute shader and return from it if the global invocation index is greater than the number of your particles.

And just as was the case for the compute pipeline, a compute command buffer contains a lot less state than a graphics command buffer. There's no need to start a render pass or set a viewport.

Submitting work

As our sample does both compute and graphics operations, we'll be doing two submits to both the graphics and compute queue per frame (see the `drawFrame` function):

```
...
if (vkQueueSubmit(computeQueue, 1, &submitInfo, nullptr) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit compute command buffer!");
}
...
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}
```

The first submit to the compute queue updates the particle positions using the compute shader, and the second submit will then use that updated data to draw the particle system.

Synchronizing graphics and compute

Synchronization is an important part of Vulkan, even more so when doing compute in conjunction with graphics. Wrong or lacking synchronization may result in the vertex stage starting to draw (=read) particles while the compute shader hasn't finished updating (=write) them (read-after-write hazard), or the compute shader could start updating particles that are still in use by the vertex part of the pipeline (write-after-read hazard).

So we must make sure that those cases don't happen by properly synchronizing the graphics and the compute load. There are different ways of doing so, depending on how you submit your compute workload but in our case with two separate submits, we'll be using [semaphores](#) and [fences](#) to ensure that the vertex shader won't start fetching vertices until the compute shader has finished updating them.

This is necessary as even though the two submits are ordered one-after-another, there is no guarantee that they execute on the GPU in this order. Adding in wait and signal semaphores ensures this execution order.

So we first add a new set of synchronization primitives for the compute work in [createSyncObjects](#). The compute fences, just like the graphics fences, are created in the signaled state because otherwise, the first draw would time out while waiting for the fences to be signaled as detailed [here](#):

```
std::vector<VkFence> computeInFlightFences;
std::vector<VkSemaphore> computeFinishedSemaphores;
...
computeInFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
computeFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);

VkSemaphoreCreateInfo semaphoreInfo{};
semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

VkFenceCreateInfo fenceInfo{};
fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
    ...
    if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &computeFinishedSemaphores[i]) != VK_SUCCESS ||
        vkCreateFence(device, &fenceInfo, nullptr, &computeInFlightFences[i]) != VK_SUCCESS) {
        throw std::runtime_error("failed to create compute synchronization objects for a frame!");
    }
}
```

Compute Submission



```
// Compute submission
vkWaitForFences(device, 1, &computeInFlightFences[currentFrame], VK_TRUE, UINT64_MAX);

updateUniformBuffer(currentFrame);

vkResetFences(device, 1, &computeInFlightFences[currentFrame]);

vkResetCommandBuffer(computeCommandBuffers[currentFrame], /*VkCommandBufferResetFlagBits*/ 0);
recordComputeCommandBuffer(computeCommandBuffers[currentFrame]);

submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &computeCommandBuffers[currentFrame];
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &computeFinishedSemaphores[currentFrame];

if (vkQueueSubmit(computeQueue, 1, &submitInfo, computeInFlightFences[currentFrame]) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit compute command buffer!");
};
```

Graphics Submission

```
// Graphics submission
vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);

...
vkResetFences(device, 1, &inFlightFences[currentFrame]);

vkResetCommandBuffer(commandBuffers[currentFrame], /*VkCommandBufferResetFlagBits*/ 0);
recordCommandBuffer(commandBuffers[currentFrame], imageIndex);

VkSemaphore waitSemaphores[] = { computeFinishedSemaphores[currentFrame], imageAvailableSemaphores[currentFrame] };
VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

submitInfo.waitSemaphoreCount = 2;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[currentFrame];
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &renderFinishedSemaphores[currentFrame];

if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}
```

Similar to the sample in the [semaphores chapter](#), this setup will immediately run the compute shader as we haven't specified any wait semaphores. This is fine, as we are waiting for the compute command buffer of the current frame to finish execution before the compute submission with the `vkWaitForFences` command.

The graphics submission on the other hand needs to wait for the compute work to finish so it doesn't start fetching vertices while the compute buffer is still updating them. So we wait on the `computeFinishedSemaphores` for the current frame and have the graphics submission wait on the `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` stage, where vertices are consumed.

But it also needs to wait for presentation so the fragment shader won't output to the color attachments until the image has been presented. So we also wait on the `imageAvailableSemaphores` on the current frame at the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage.

Exercise: Replace binary semaphores with timeline semaphores

Drawing the particle system

Earlier on, we learned that buffers in Vulkan can have multiple use-cases and so we created the shader storage buffer that contains our particles with both the shader storage buffer bit and the vertex buffer bit. This means that we can use the shader storage buffer for drawing just as we used "pure" vertex buffers in the previous chapters.

We first setup the vertex input state to match our particle structure:

```
struct Particle {  
    ...  
  
    static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {  
        std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions{};  
  
        attributeDescriptions[0].binding = 0;  
        attributeDescriptions[0].location = 0;  
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;  
        attributeDescriptions[0].offset = offsetof(Particle, position);  
  
        attributeDescriptions[1].binding = 0;  
        attributeDescriptions[1].location = 1;  
        attributeDescriptions[1].format = VK_FORMAT_R32G32B32A32_SFLOAT;  
        attributeDescriptions[1].offset = offsetof(Particle, color);  
  
        return attributeDescriptions;  
    }  
};
```

Note that we don't add `velocity` to the vertex input attributes, as this is only used by the compute shader.

We then bind and draw it like we would with any vertex buffer:

```
vkCmdBindVertexBuffers(commandBuffer, 0, 1, &shaderStorageBuffer[currentFrame], offsets);  
  
vkCmdDraw(commandBuffer, PARTICLE_COUNT, 1, 0, 0);
```

DAY 4 - Agenda

- Review of Exercises, Q&A
- Texturing
 - Manipulating texture coordinates in the shader
 - Handling aspect ratio in shaders
 - Repeat, Border, Clamp
- Viewport vs. Scissor
- Vulkan Compute – N-body Simulation
 - Specialization Constants
 - VMA
- RDMA Sample
- Overview of References and Resources for further learning
- Q&A

Vulkan External Memory RDMA

- Check if physical device memory supports
`VK_MEMORY_PROPERTY_RDMA_CAPABLE_BIT_NV` and
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_RDMA_ADDRESS_BIT_NV`
- Create external buffer using `vkCreateBuffer` and
`VkExternalMemoryBufferCreateInfo` as in the CUDA interop examples.
- Request handle type of
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_RDMA_ADDRESS_BIT_NV`
- Find the memory type index from memory properties to use in
`VkMemoryAllocateInfo`
- Allocate and bind memory the external buffer.
- Use `vkGetDeviceProcAddr` to get the extension function pointer for
`vkGetMemoryRemoteAddressNV`
- Use `vkGetMemoryRemoteAddressNV` and `VkMemoryGetRemoteAddressInfoNV` to
retrieve the RDMA address for use by the external device.
- https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_NV_external_memory_rdma.html

The `VkPhysicalDeviceExternalMemoryRDMAFeaturesNV` structure is defined as:

```
// Provided by VK_NV_external_memory_rdma
typedef struct VkPhysicalDeviceExternalMemoryRDMAFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            externalMemoryRDMA;
} VkPhysicalDeviceExternalMemoryRDMAFeaturesNV;
```

Members

This structure describes the following feature:

Description

- `sType` is a `VkStructureType` value identifying this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `externalMemoryRDMA` indicates whether the implementation has support for the `VK_MEMORY_PROPERTY_RDMA_CAPABLE_BIT_NV` memory property and the `VK_EXTERNAL_MEMORY_HANDLE_TYPE_RDMA_ADDRESS_BIT_NV` external memory handle type.

If the `VkPhysicalDeviceExternalMemoryRDMAFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceExternalMemoryRDMAFeaturesNV` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

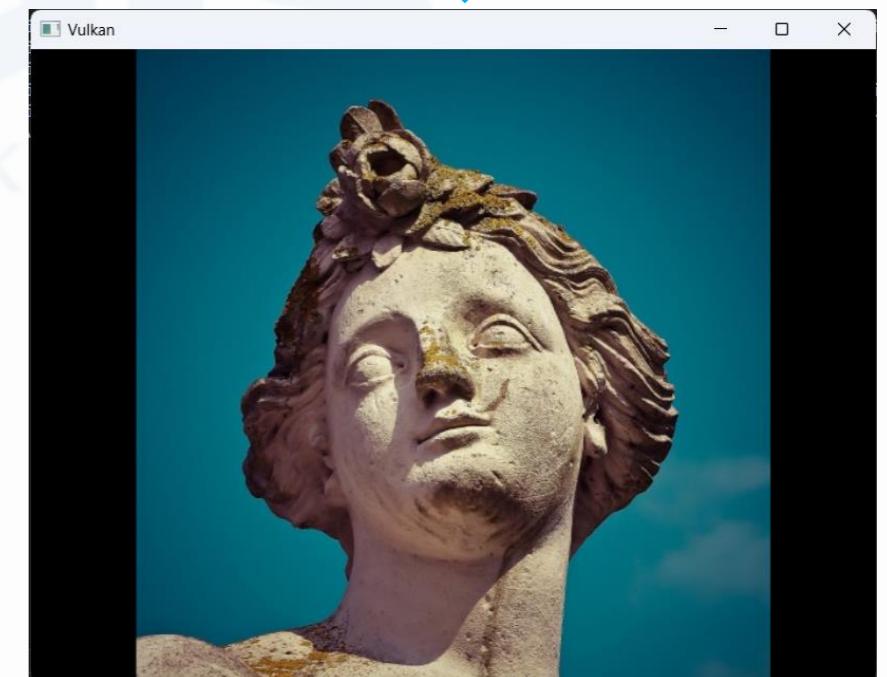
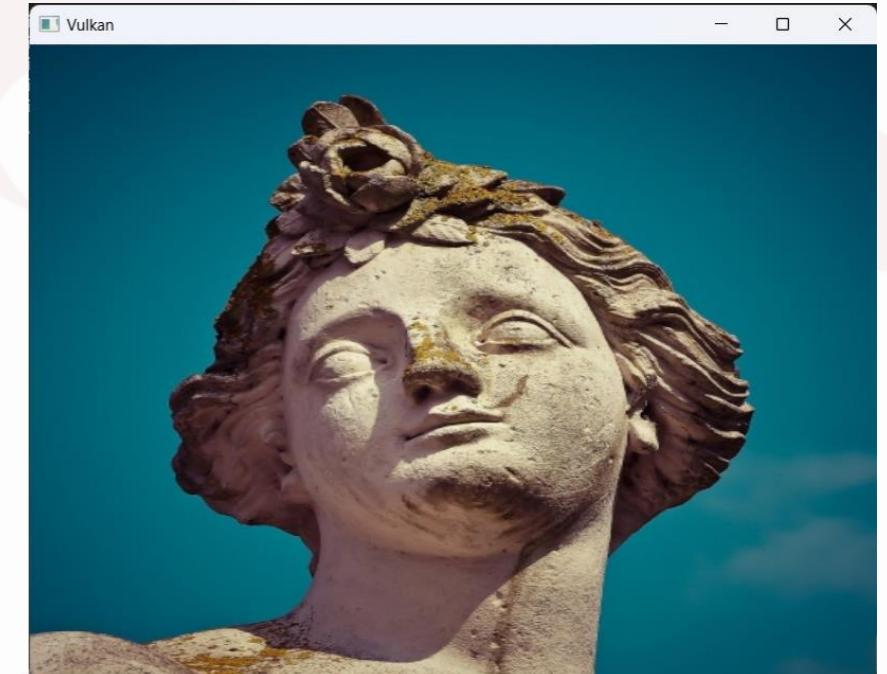
Exercise #4

Handle non-square displays i.e. Aspect Ratio not equalling 1.0

In our example, the display is 800x600. We need to correct for the display's aspect ratio.

- One possible solution – adjust texture coordinates:

```
// Scale vertex position to cover view.  
// Assumes input geometry is (-0.5,-0.5)->(0.5,0.5)  
// For generality, this should be encoded into the MVP matrix as in the 3D case.  
gl_Position = vec4(2.0 * inPosition, 0.0, 1.0);  
// Scale and shift texture coordinates to correct for window aspect ratio  
if(aspectRatio > 1.0){  
    scale = vec2(aspectRatio, 1.0);  
    shift = 0.5 * vec2(aspectRatio - 1.0, 0.0);  
}else{  
    scale = vec2(1.0, 1.0 / aspectRatio);  
    shift = 0.5 * vec2(0.0, 1.0 / aspectRatio - 1.0);  
}  
fragTexCoord = inTexCoord * scale - shift;
```



Exercise – Handle Window Aspect Ratio (600x800 here): Display Mode 0: 3D view, as in original example code

```
#version 450

#define DISPLAY_3D 0
#define DISPLAY_TEXTURE_FIT 1
#define DISPLAY_QUAD_FIT 2
#define DISPLAY_STRETCH_FIT 3

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
    vec2 windowSize;
    uint displayMode; // 0 - use MVP, 1 - scale the texture, 2 - scale the image quad
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 2) in vec2 inTexCoord;

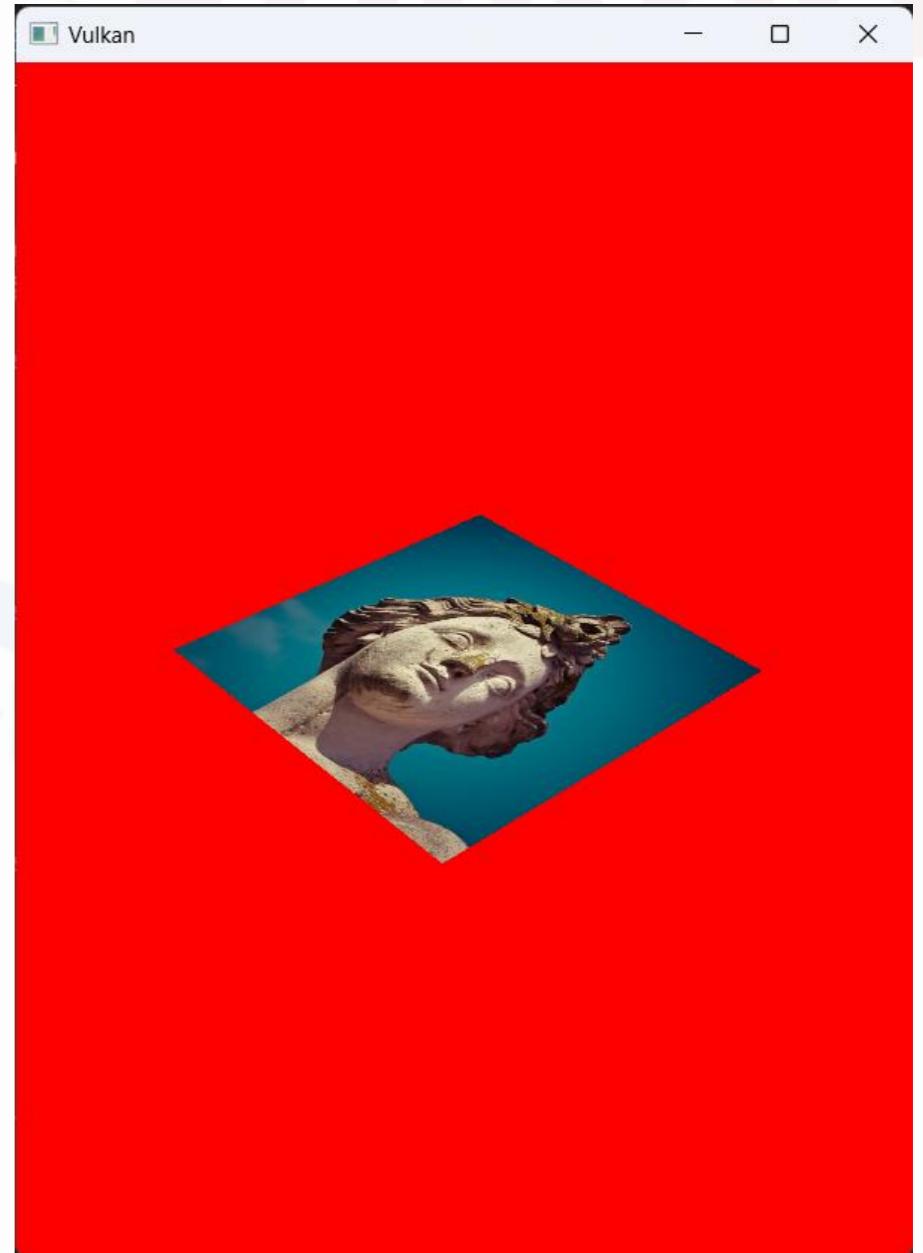
layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;

void main() {

    fragColor = inColor;

    float aspectRatio = ubo.windowSize.x / ubo.windowSize.y;
    vec2 scale, shift;

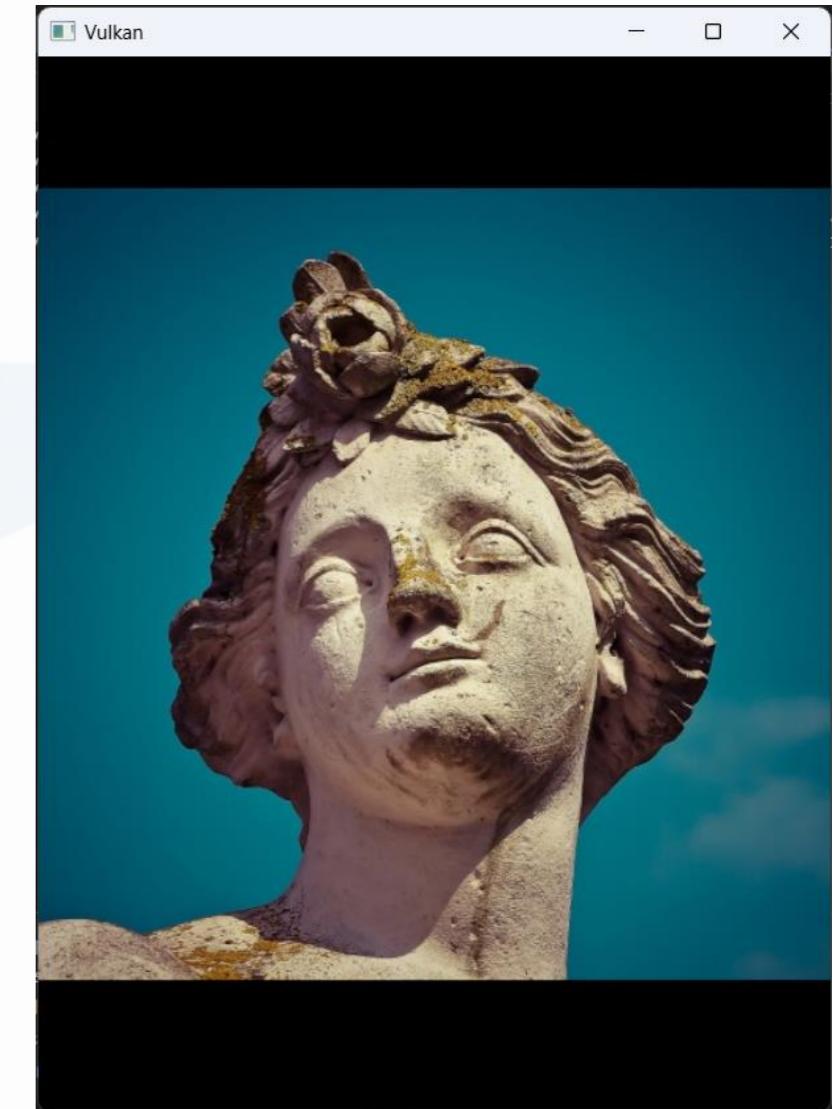
    switch(ubo.displayMode){
        case DISPLAY_3D:
            gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
            fragTexCoord = inTexCoord;
            break;
    }
}
```



Exercise – Handle Window Aspect Ratio (600x800 here):

Display Mode 1: Texture Fit (fit the texture within the quad to maintain aspect ratio)

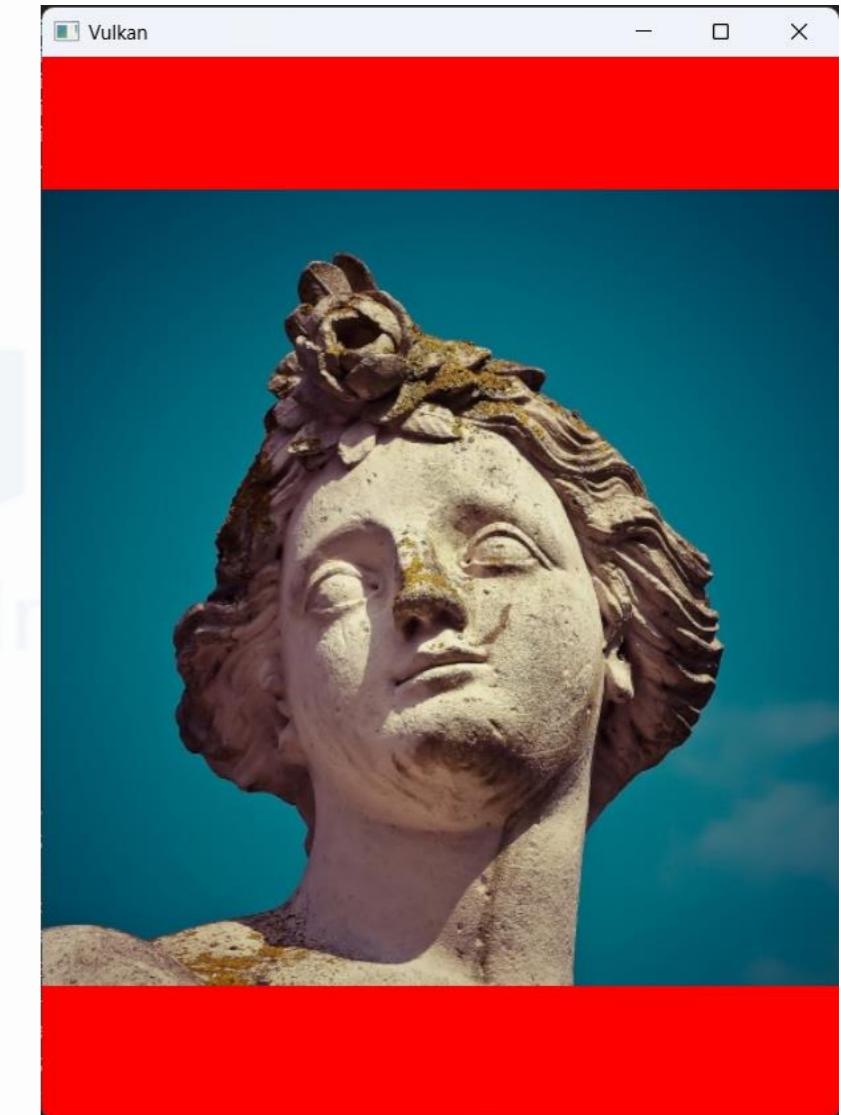
```
case DISPLAY_TEXTURE_FIT:  
    // Scale vertex position to cover view. Assumes input geometry is (-0.5,-0.5)->(0.5,0.5)  
    // For generality, this should be encoded into the MVP matrix as in the 3D case.  
    gl_Position = vec4(2.0 * inPosition, 0.0, 1.0);  
    // Scale and shift texture coordinates to correct for window aspect ratio  
    if(aspectRatio > 1.0){  
        scale = vec2(aspectRatio, 1.0);  
        shift = 0.5 * vec2(aspectRatio - 1.0, 0.0);  
    }else{  
        scale = vec2(1.0, 1.0 / aspectRatio);  
        shift = 0.5 * vec2(0.0, 1.0 / aspectRatio - 1.0);  
    }  
    fragTexCoord = inTexCoord * scale - shift;  
    break;
```



Exercise – Handle Window Aspect Ratio (600x800 here):

Display Mode 2: Quad Fit (Fit the textured quad within the window to maintain aspect ratio)

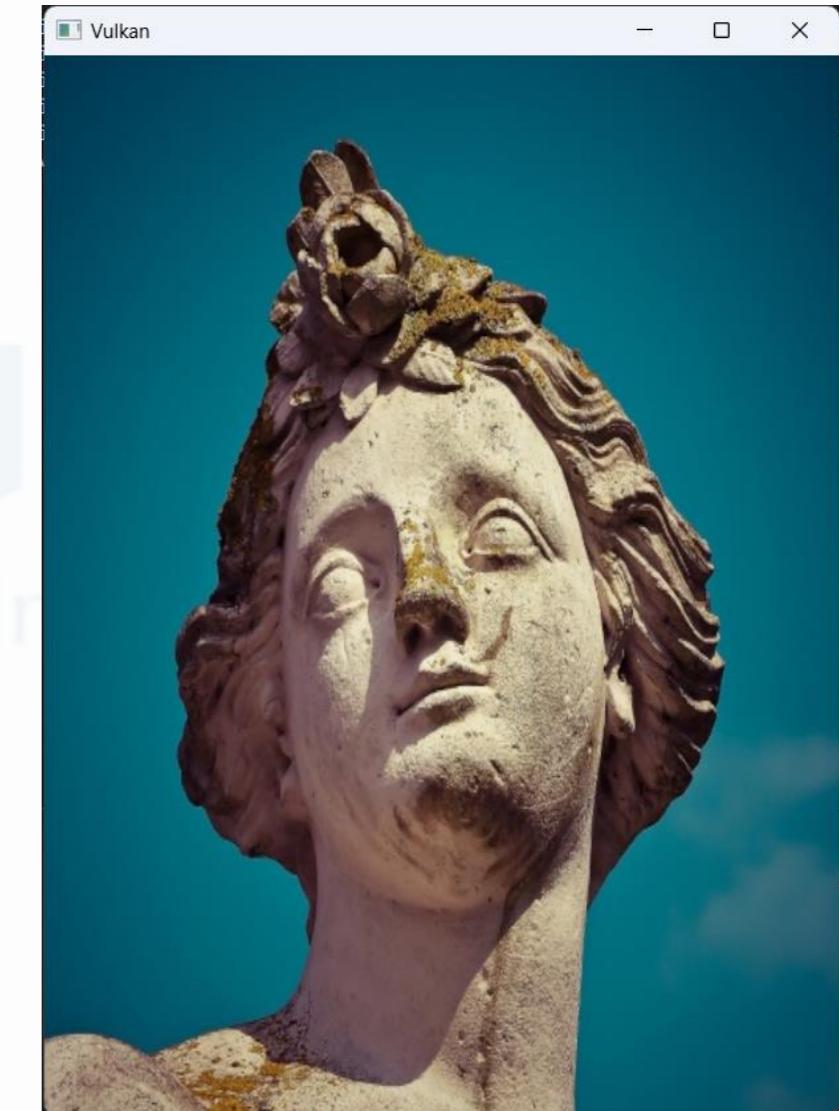
```
case DISPLAY_QUAD_FIT:  
    scale = aspectRatio > 1.0 ?  
        vec2(1.0 / aspectRatio, 1.0) :  
        vec2(1.0, aspectRatio);  
    // Scale vertex position to cover view, with aspect ratio correction.  
    // Assumes input geometry is (-0.5,-0.5)->(0.5,0.5)  
    // For generality, this should be encoded into the MVP matrix as in the 3D case.  
    gl_Position = vec4(2.0 * inPosition * scale, 0.0, 1.0);  
    fragTexCoord = inTexCoord;  
    break;
```



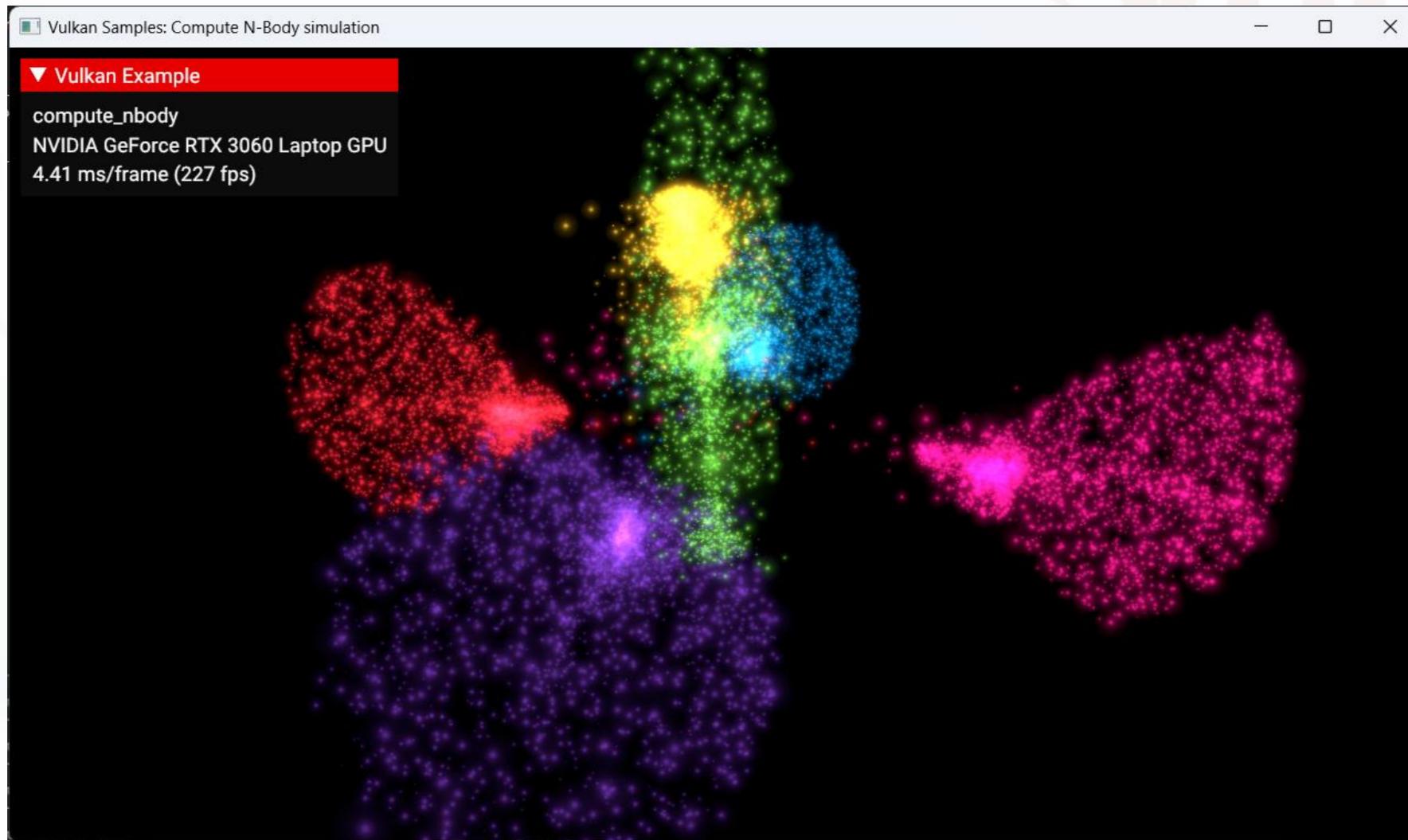
Exercise – Handle Window Aspect Ratio (600x800 here):

Display Mode 3: Stretch Fit (Stretch the textured quad to the window regardless of aspect ratio)

```
case DISPLAY_STRETCH_FIT:  
    // Scale vertex position to cover view (no aspect ratio correction).  
    // Assumes input geometry is (-0.5,-0.5)->(0.5,0.5)  
    // For generality, this should be encoded into the MVP matrix as in the 3D case.  
    gl_Position = vec4(2.0 * inPosition, 0.0, 1.0);  
    fragTexCoord = inTexCoord;  
    break;
```



Compute Example: N-body Simulation



- From Khronos Vulkan-Samples:
<https://github.com/KhronosGroup/Vulkan-Samples>

`samples/api/compute_nbody`

- Compute shader example that uses two passes and shared compute shader memory for simulating a N-Body particle system.
- Key Takeaways:
 - Two compute shader pipelines for the two passes
 - Uses **Specialization Constants** to set constant parameters and compute `local_size`
 - Uses **VMA** for memory allocation
 - Uses **ImGui** for GUI (**GLFW** is simpler for most beginners to prototype though)

ImGui



- Dear ImGui is a **bloat-free graphical user interface library for C++**. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline-enabled application. It is fast, portable, renderer agnostic, and self-contained (no external dependencies).
- Dear ImGui is designed to **enable fast iterations** and to **empower programmers** to create **content creation tools and visualization / debug tools** (as opposed to UI for the average end-user). It favors simplicity and productivity toward this goal and lacks certain features commonly found in more high-level libraries.
- Dear ImGui is particularly suited to integration in game engines (for tooling), real-time 3D applications, fullscreen applications, embedded applications, or any applications on console platforms where operating system features are non-standard.

VMA (Vulkan Memory Allocator)

- The Vulkan Memory Allocator, developed by AMD, is a set of functions to simplify your view of allocating buffer memory.
- The github link is:
 - <https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator>
- This repository includes documentation and sample programs.

10.8. Specialization Constants

Specialization constants are a mechanism whereby constants in a SPIR-V module **can** have their constant value specified at the time the `VkPipeline` is created. This allows a SPIR-V module to have constants that **can** be modified while executing an application that uses the Vulkan API.



Note

Specialization constants are useful to allow a compute shader to have its local workgroup size changed at runtime by the user, for example.

Each `VkPipelineShaderStageCreateInfo` structure contains a `pSpecializationInfo` member, which **can** be `NULL` to indicate no specialization constants, or point to a `VkSpecializationInfo` structure.

The `VkSpecializationInfo` structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationInfo {
    uint32_t                      mapEntryCount;
    const VkSpecializationMapEntry* pMapEntries;
    size_t                         dataSize;
    const void*                    pData;
} VkSpecializationInfo;
```

- `mapEntryCount` is the number of entries in the `pMapEntries` array.
- `pMapEntries` is a pointer to an array of `VkSpecializationMapEntry` structures, which map constant IDs to offsets in `pData`.
- `dataSize` is the byte size of the `pData` buffer.
- `pData` contains the actual constant values to specialize with.



Specialization Constants



Oregon State
University
Mike Bailey

mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons](#)
[Attribution-NonCommercial-NoDerivatives 4.0](#)
[International License](#)



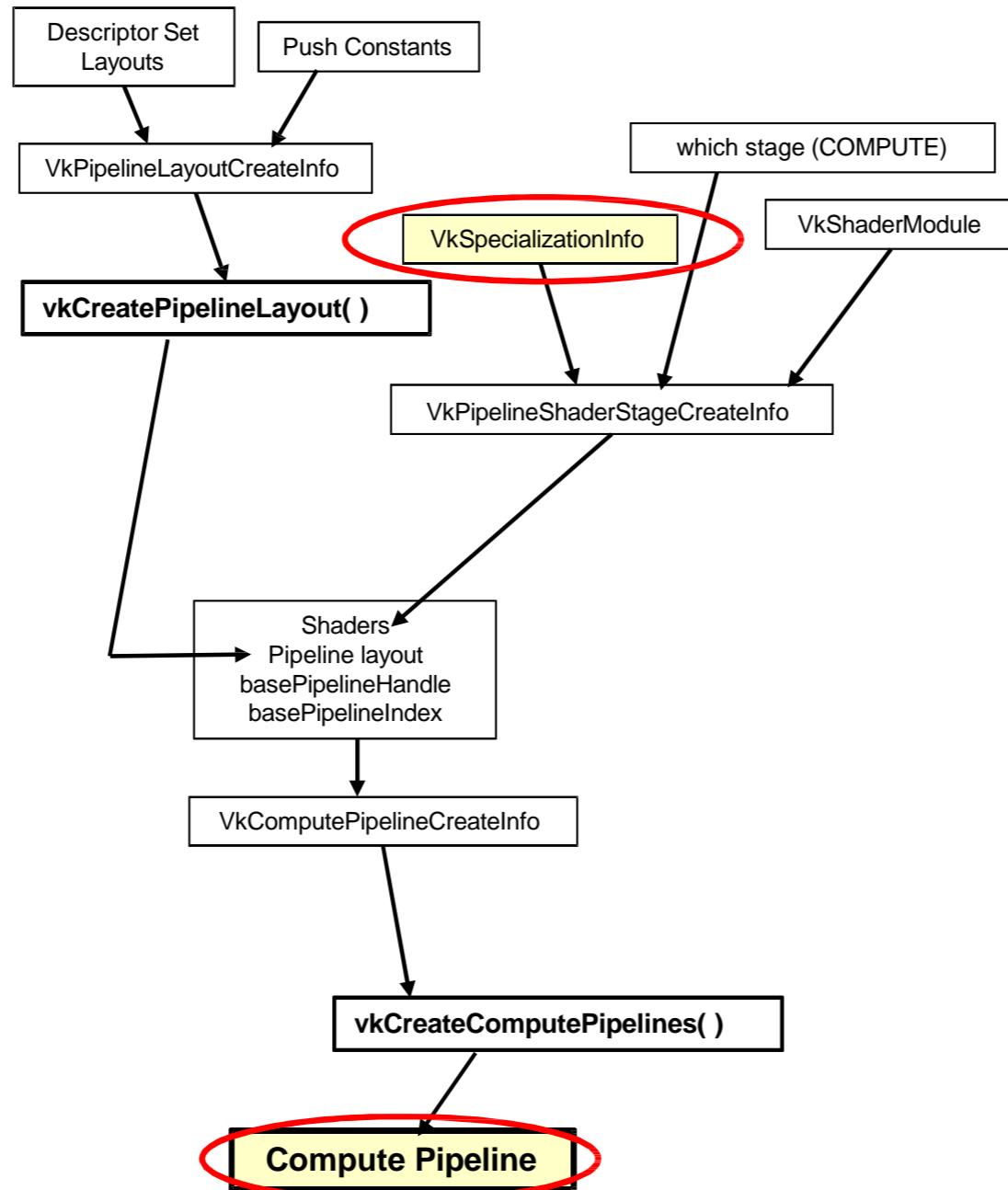
Oregon State
University



Oregon State
University
Computer Graphics

Remember the Compute Pipeline?

2



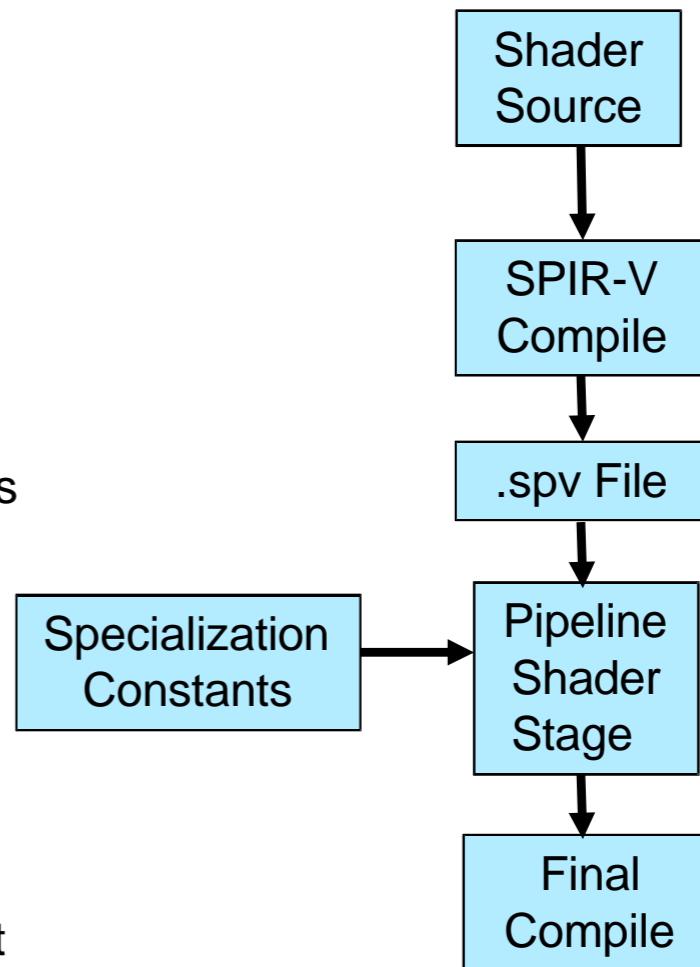
In Vulkan, all shaders get halfway-compiled into SPIR-V and then the rest-of-the-way compiled by the Vulkan driver.

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()**

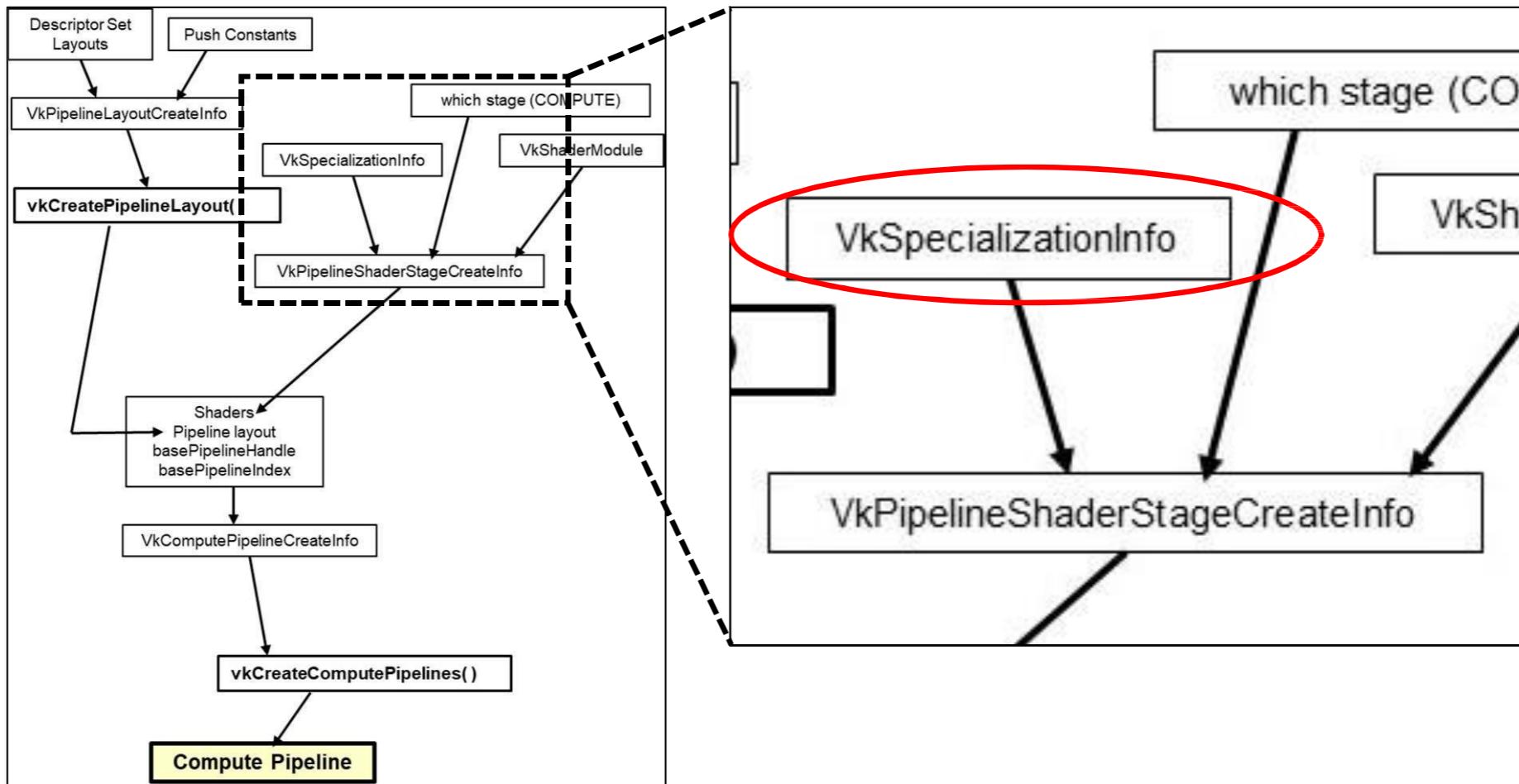
Without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago



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 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[0] constantID = 7;
    vsme[0].offset = 0;
    vsme[0].size = sizeof(asize); // # bytes into the Specialization Constant
                                // array this one item is
                                // 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
                            // array of all the Specialization Constants
```

```
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 );
```



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);
```

It's important to use `sizeof()`
and `offsetof()` instead of
hardcoding numbers!

```
VkSpecializationInfo vsi;  
vsi.mapEntryCount = 3;  
vsi.pMapEntries = &vsme[0];  
vsi.dataSize = sizeof(abc);  
vsi.pData = &abc;
```

// size of *all* the Specialization Constants together
// array of *all* the Specialization Constants



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

In the compute shader

```
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;
```





The Vulkan Memory Allocator (VMA)



Oregon State
University
Mike Bailey

mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons](#)
[Attribution-NonCommercial-NoDerivatives 4.0](#)
[International License](#)



Oregon State
University



Oregon State
University
Computer Graphics

The **Vulkan Memory Allocator**, developed by AMD, is a set of functions to simplify your view of allocating buffer memory. It is all included in our class VMA sample code, but if you want to go get it for yourself, the github link is:

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

This repository includes a smattering of documentation and sample programs.



Do this in just one .cpp file:

```
#define VMA_IMPLEMENTATION  
#define VMA_VULKAN_VERSION 1001000      // if vulkan version 1.1  
#include "vk_mem_alloc.h"
```

Do this in all the other .cpp files:

```
#define VMA_VULKAN_VERSION 1001000      // if vulkan version 1.1  
#include "vk_mem_alloc.h"
```

Do the usual Vulkan setup for:

```
VkPhysicalDevice    PhysicalDevice;  
VkLogicalDevice     LogicalDevice;  
VkInstance          Instance;
```

Add one new global variable for VMA:

```
VmaAllocator        Allocator;
```

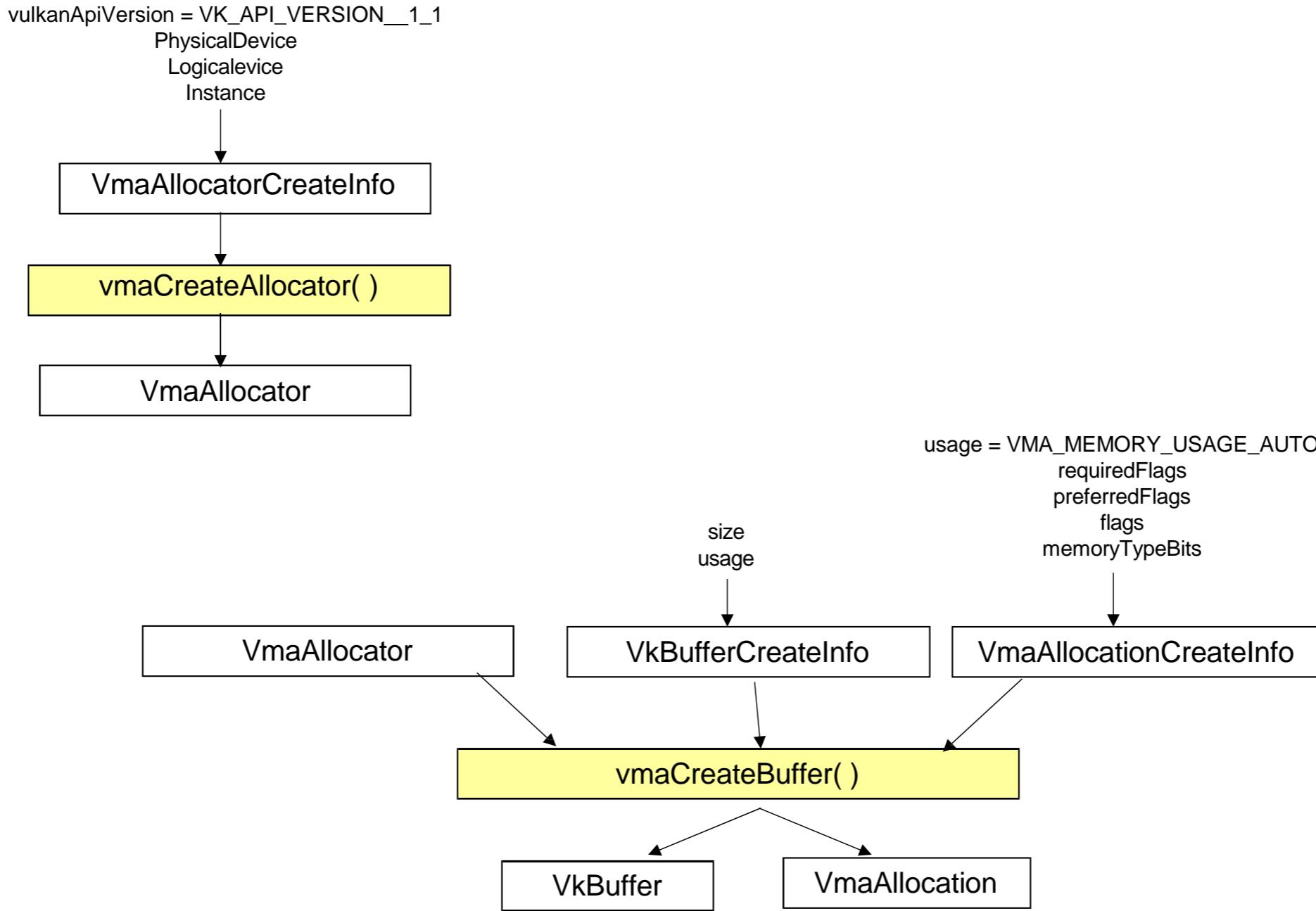
Add one new global variable for each VMA allocation you do:

```
VmaAllocation       Allocation;
```



Vulkan Memory Allocator (VMA)

4



```
VmaAllocator Allocator; // global  
.  
.  
.  
VmaAllocatorCreateInfo vrci;  
vrci.vulkanApiVersion = VK_API_VERSION_1_1;  
vrci.flags = 0; // VmaAllocatorCreateFlagBits enum  
vrci.physicalDevice = PhysicalDevice; // from usual vulkan setup  
vrci.device = LogicalDevice; // from usual vulkan setup  
vrci.instance = Instance; // from usual vulkan setup  
vrci.pVulkanFunctions = nullptr;
```

```
vmaCreateAllocator( IN &vrci, OUT &Allocator );
```

The Allocator acts as the keeper of
the system knowledge for VMA



```
VMA_ALLOCATOR_CREATE_EXTERNALLY_SYNCHRONIZED_BIT  
VMA_ALLOCATOR_CREATE_KHR_DEDICATED_ALLOCATION_BIT  
VMA_ALLOCATOR_CREATE_KHR_BIND_MEMORY2_BIT VMA_ALLOCATOR_CREATE_EXT_ME  
VMA_ALLOCATOR_CREATE_AMD_DEVICE_COHERENT_MEMORY_BIT  
VMA_ALLOCATOR_CREATE_BUFFER_DEVICE_ADDRESS_BIT  
VMA_ALLOCATOR_CREATE_EXT_MEMORY_PRIORITY_BIT
```



```
VkBuffer Buffer;           // or "VkDataBuffer Buffer"

VkBufferCreateInfo vbc;
```

vbc.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbc.pNext = nullptr;
vbc.flags = 0;
vbc.size = << buffer size in bytes >>
vbc.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

vbc.sharingMode = << one of: >>
VK_SHARING_MODE_EXCLUSIVE
VK_SHARING_MODE_CONCURRENT

vbc.queueFamilyIndexCount = 0;
vbc.pQueueFamilyIndices = (const int32_t) nullptr;

// DO NOT CREATE THE BUFFER – LET VMA DO IT!
// result = vkCreateBuffer(LogicalDevice, IN &vbc, PALLOCATOR, OUT &Buffer);

“or” these bits
together to specify
how this buffer will be
used



```
#include "vk_mem_alloc.h"
...
VkBuffer           Buffer;          // global
...

VmaAllocationCreateInfo vaci;
    vaci.usage      = VMA_MEMORY_USAGE_AUTO; // select what it thinks is the best type (recommended)
    vaci.requiredFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT;
    vaci.preferredFlags = VK_MEMORY_PROPERTY_HOST_COHERENT_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT;
    vaci.flags       = VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT;

vmaCreateBuffer( IN Allocator, IN &vbc, IN &vaci, OUT &Buffer, OUT &Allocation, nullptr );
```

Both allocates and binds in one call

The Allocation acts as the keeper of
this specific buffer knowledge for VMA

```
vmaDestroyBuffer( Allocator, Buffer, Allocation );
```

```
vmaDestroyAllocator( Allocator );
```



```
VMA_MEMORY_USAGE_UNKNOWN  
VMA_MEMORY_USAGE_GPU_ONLY  
VMA_MEMORY_USAGE_CPU_ONLY  
VMA_MEMORY_USAGE_CPU_TO_GPU  
VMA_MEMORY_USAGE_GPU_TO_CPU  
VMA_MEMORY_USAGE_CPU_COPY  
VMA_MEMORY_USAGE_GPU_LAZILY_ALLOCATED  
VMA_MEMORY_USAGE_AUTO  
VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE  
VMA_MEMORY_USAGE_AUTO_PREFER_HOST
```

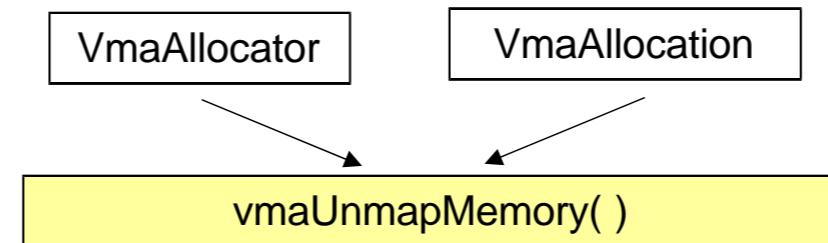
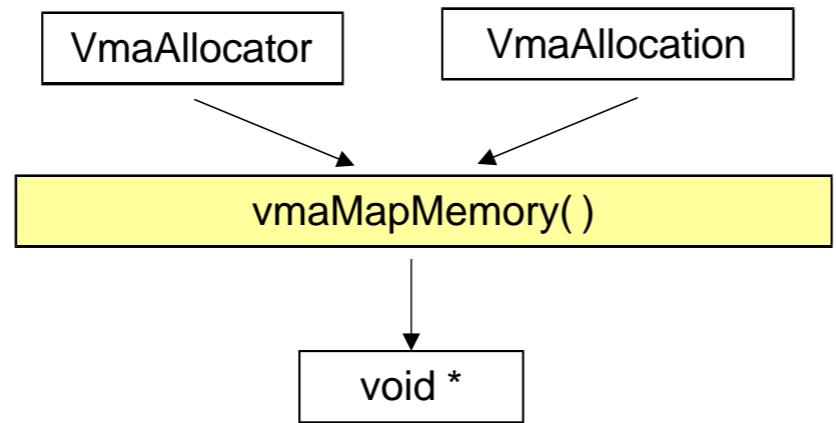


```
VMA_ALLOCATION_CREATE_DEDICATED_MEMORY_BIT VMA_ALLOCATION_CREATE_NEVER_ALLOCATE_BIT  
VMA_ALLOCATION_CREATE_MAPPED_BIT VMA_ALLOCATION_CREATE_USER_DATA_COPY_STRATEGY_BIT  
VMA_ALLOCATION_CREATE_UPPER_ADDRESS_BIT VMA_ALLOCATION_CREATE_DONT_BIND_BIT  
VMA_ALLOCATION_CREATE_WITHIN_BUDGET_BIT VMA_ALLOCATION_CREATE_CAN_ALIAS_BIT  
VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT  
VMA_ALLOCATION_CREATE_HOST_ACCESS_RANDOM_BIT  
VMA_ALLOCATION_CREATE_HOST_ACCESS_ALLOW_TRANSFER_INSTEAD_BIT  
VMA_ALLOCATION_CREATE_STRATEGY_MIN_MEMORY_BIT VMA_ALLOCATION_CREATE_STRATEGY_MIN_OFFSET_BIT
```



```
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT  
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT VK_MEMORY_PROPERTY_HOST_CACHED_BIT  
VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT VK_MEMORY_PROPERTY_PROTECT_BIT  
VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD  
VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD  
VK_MEMORY_PROPERTY_RDMA_CAPABLE_BIT_NV
```





```
void *mappedDataAddr;  
  
vmaMapMemory( Allocator, Allocation, OUT &mappedDataAddr );  
  
    memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );  
  
vmaUnmapMemory( Allocator, Allocation );
```



```
struct vertex *vp;

vmaMapMemory( Allocator, Allocation, OUT (void *)&vp );

for( int i = 0; i < numTrianglesInObjFile; i++ )      // number of triangles
{
    for( int j = 0; j < 3; j++ )                      // 3 vertices per triangle
    {
        vp->position = glm::vec3( ... );
        vp->normal = glm::vec3( ... );
        vp->color = glm::vec3( ... );
        vp->texCoord = glm::vec2( ... );
        vp++;
    }
}

vmaUnmapMemory( Allocator, Allocation );
```



Vulkan Samples

- Khronos Samples
 - <https://github.com/KhronosGroup/Vulkan-Samples>
- NVIDIA CUDA Samples
 - <https://github.com/NVIDIA/cuda-samples>
- Sascha Willems Samples (more advanced)
 - <https://github.com/SaschaWillems/Vulkan>

Resources and Reference



- Vulkan Tutorial
 - <https://vulkan-tutorial.com/>
- Vulkan Lecture Series from TUW
 - <https://youtube.com/playlist?list=PLmlqTlJ6KsE1Jx5HV4sd2jOe3V1KMHHgn>
- Vulkan Reference
 - <https://registry.khronos.org/vulkan/specs/1.3-extensions/html/>
- Mike Bailey's Vulkan Page
 - <https://web.engr.oregonstate.edu/~mjb/vulkan/>
- Window System Integration (WSI)
 - <https://registry.khronos.org/vulkan/site/guide/latest/wsi.html>
- API without Secrets: Introduction to Vulkan
 - <https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-preface.html>
- Tips and Tricks: Vulkan Dos and Don'ts
 - <https://developer.nvidia.com/blog/vulkan-dos-donts/>
- NVIDIA Developer Zone
 - <https://developer.nvidia.com/>
- Computer Graphics CMU Course
 - <http://15462.courses.cs.cmu.edu/spring2023/courseinfo>