

# Vulkanised 2025

The 7<sup>th</sup> Vulkan Developer Conference  
Cambridge, UK | February 11-13, 2025

## So you want to write a Vulkan Renderer in 2025

---

Charles Giessen  
LunarG, Inc.



# Who am I?

- Started learning Vulkan in 2017
- Joined LunarG in 2019
  - Maintain the Vulkan-Loader, Api dump, VkCube, Vulkaninfo, Vulkan-Utility-Libraries, SDK development, & more
- Joined the Vulkan Community Discord in 2018
  - Moderator since ~2021



Charles Giessen  
LunarG

The LunarG logo features the word "LUNAR" in a white, sans-serif font, followed by a large, stylized closing parenthesis symbol "G" that also serves as the letter "G". The logo is set against a background of a glowing blue and orange circuit board.

LUNAR)G®

GPU SOFTWARE SPECIALISTS

# Disclaimers

- Strictly my opinions
- My experience is with Desktop Vulkan
  - Android is my blindspot
- Vulkan is MASSIVE
  - Cannot cover everything!
- Not covering rendering algorithms
  - Focusing on the Vulkan parts of rendering

**Let's get started!**

# How to design a Vulkan Renderer

1. Pick the API Version
2. Pick the extensions to use
3. ????
4. Profit!!!!

# What version to target?

# Which version to target?

- Whatever your hardware minimum is!
- Depends on target platform
  - Desktop (Linux & Windows) can reliably use 1.3
  - Android lags behind with 1.1
  - MacOS (MoltenVK) is 1.2 but supports most everything in 1.3
- 1.4 still too new to recommend

# Three different version numbers

- Instance
  - From the Vulkan-Loader
  - `vkEnumerateInstanceVersion` (Added in 1.1)
- Physical Device
  - From the GPU Driver
  - `VkPhysicalDeviceProperties::apiVersion`
- Application
  - Defined by the application
  - `VkApplicationInfo::apiVersion`

```
VkApplicationInfo appInfo{};  
appInfo.apiVersion = VK_MAKE_API_VERSION(0, 1, X, 0);
```

# What extensions to use?

# Two Types of Extensions

- Device
  - Vast majority
  - From `vkEnumerateDeviceExtensions`
  - Many have been promoted into Core Versions
- Instance
  - Small number
  - From `vkEnumerateInstanceExtensions`
  - `VK_EXT_debug_utils`
  - WSI: `VK_KHR_<platform>_surface` & `VK_KHR_surface`

# 300+ published extensions

- Every extension solves a problem
  - Some fix just one site
  - Other rewrite half the API
- Depends on many things
  - Hardware minimum
  - Target version
  - Supported platforms
  - Maintenance constraints
- Way too many to cover each individually

# But which to choose!?

- Overwhelming to read about each one
- Many are aliases
  - To core functionality
  - To other extensions
- Many have dependencies on other extensions
- In addition, must enable feature bools
  - `VkPhysicalDeviceFeatures2` pNext chain
  - Core features in `VkPhysicalDeviceVulkan<Version#>Features`
  - Extension features in `VkPhysicalDeviceVulkan<Ext_name>Features<Ext_suffix>`
- More specifics on checking & enabling extensions
  - [https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/enabling\\_extensions.adoc](https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/enabling_extensions.adoc)

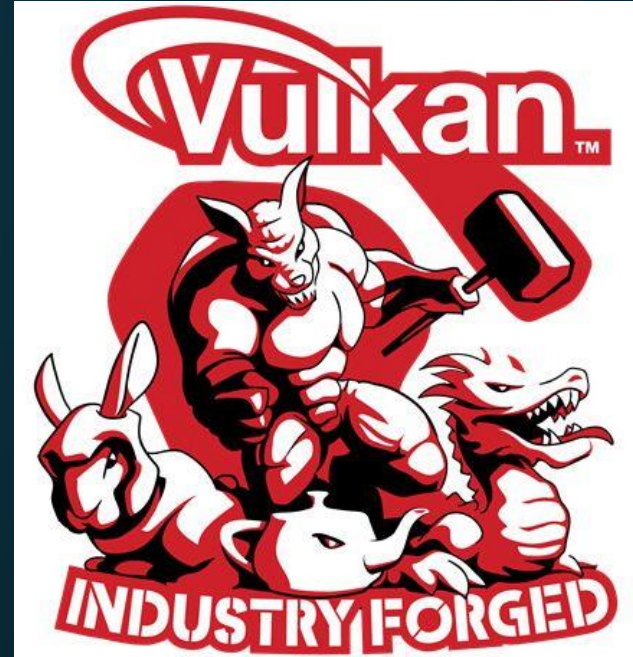
# Vulkan-Profiles

- Profiles define explicit extension & features requirements
  - Expressed in JSON
- Example: VP\_KHR\_roadmap\_2024
  - “This roadmap profile is intended to be supported by newer devices shipping in 2024 across mainstream smartphone, tablet, laptops, console and desktop devices.”
- Profiles-Library sets up VkDevice according to a given Profile
- Profiles are a great help in picking extensions & features
  - Just use what’s in a profile
  - Or *use* the profile directly!

**Why not just use Vulkan 1.0?**

# Vulkan 1.0

- Single target, no guesswork
- Supported everywhere
- Just learning 1.0 is hard enough
  - New things means more to learn
- Huge departure from OpenGL
- New concepts introduced
  - Render Passes
  - Synchronization
  - Descriptor Sets
  - Pipelines
  - And more!



# New paradigms, new problems

- Not all parts of Vulkan 1.0 are a success
  - Clunky interfaces
  - Complex interactions
  - Missing capabilities
- To be expected for an entirely new API!
- Picking a version and extensions is hard
- Yet using Vulkan 1.0 is even harder

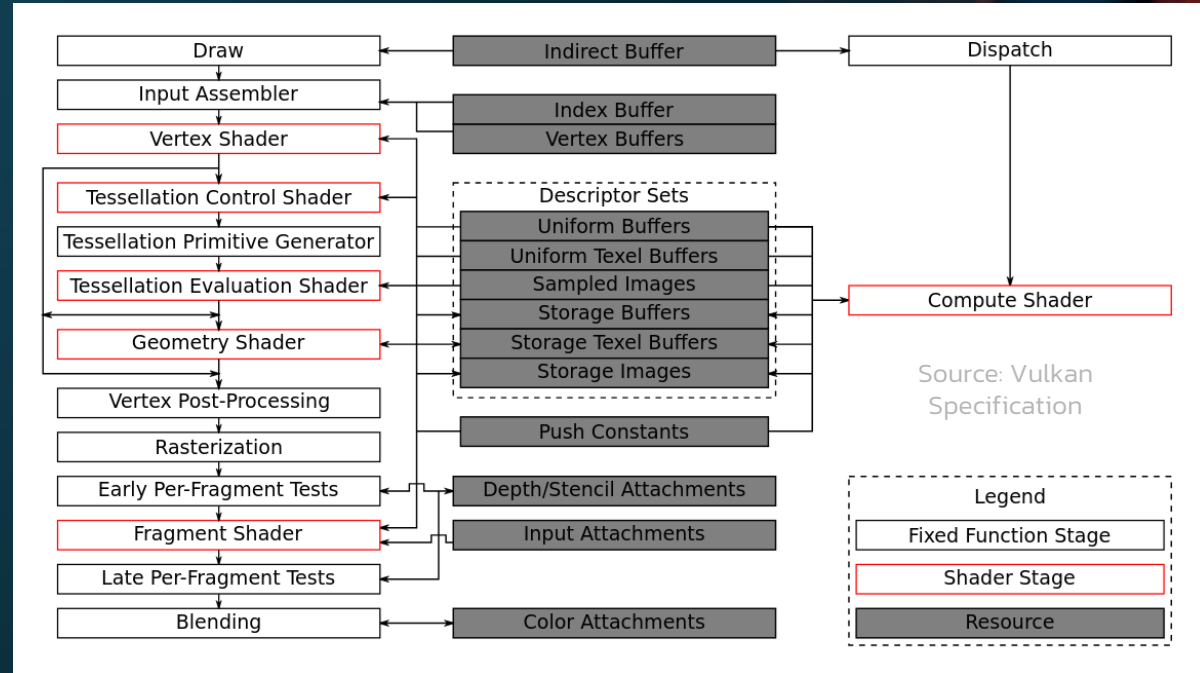
# We aren't stuck with Vulkan 1.0

- Good news is that this is 2025, not 2016
- Most of the glaring have been fixed
- Many more problems fixed in the following years
- The rest of the slides discuss these problems and their solutions

# Immutable Pipelines

# What even is a pipeline?

- Compiled binary of pipeline inputs
- Bake once
  - Costly
- Bind many times
  - Cheap
- Exchange time for space



# What's the catch?

- Pipelines require everything to be known ahead of time
- Each combination of inputs requires a dedicated pipeline
  - Shader, topology, blend mode, vertex layout, cull mode, etc
- Causes a combinatorial explosion of variants
  - 10,000's of pipelines for shipping titles
- Pipeline creating takes time
  - Creates stutters if done just-in-time

# VkDynamicState

- Not everything has to be immutable
- Set desired state while recording command buffers
- Over 70 states can be dynamic
- Great reference for all of it:
  - [https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/dynamic\\_state\\_map.adoc](https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/dynamic_state_map.adoc)

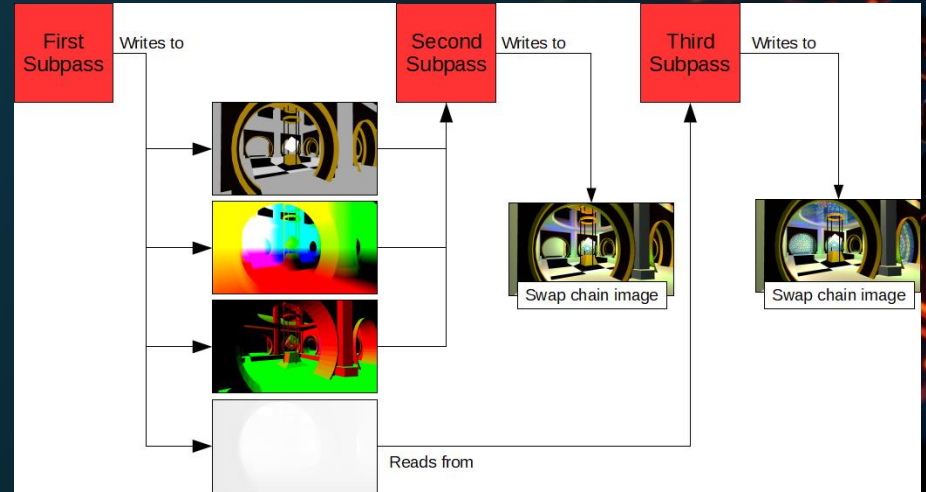
# Reducing compilation overhead

- `VK_EXT_graphics_pipeline_libraries`
  - Divide graphics pipeline into multiple parts
  - Link them into single binary right before binding
  - Diminishes cost of many variants
- `VK_EXT_shader_object`
  - Ditch pipelines entirely
  - Bind compiled shader stages
  - Currently only available on AMD & Nvidia
- Jury is still out on best way to do this

# Render Passes

# Render Passes and Sub Passes

- All drawing commands happen inside a “renderpass”
- Acts as a pseudo render graph
- Allows tiling GPU’s to use memory efficiently
- Describes image attachments
- Defines the subpasses
- Declare dependencies between subpasses



Source: Sascha Willems

Vulkan input attachments and sub passes

# Great in theory...

- Not so great to use in practice
- Single object with many responsibilities
  - Defines attachments
  - Defines memory barriers for attachments
  - Defines subpasses that read from and write to attachments
- Hard to architect into a renderer
  - Yet another input for pipelines
- Main benefit is for tiling based GPU's
  - Commonly found in mobile
- Requires using `VkFramebuffer`s
  - Only exists to combine images and renderpasses

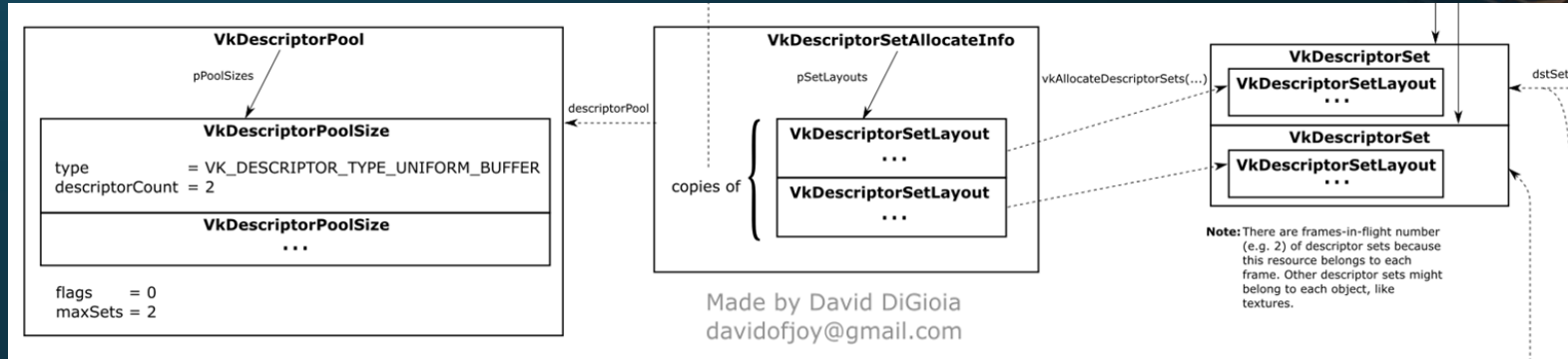
# Introducing Dynamic Rendering

- 1.3's dynamicRendering feature, VK\_KHR\_dynamic\_rendering
- Replaces VkRenderpass
- Describe renderpasses inline with command buffer recording
- Greatly simplifies application architecture
  - Creating pipelines only needs attachment descriptions
- Tiling GPU's aren't left behind either
- 1.4 dynamicRenderingLocalRead, VK\_KHR\_dynamic\_rendering\_local\_read
  - Enables efficient multi-pass rendering

# Descriptor Sets

# Descriptor Sets

- Organize shader inputs into “sets” by update frequency
- Update each set together
- Bind sets as needed
- Reasonable API for the gamut of existing hardware
- Small snippet of descriptor API:



# Descriptor Difficulties

- Cannot update descriptors after binding in a command buffer
- All descriptors must be valid, even if not used
- Descriptor arrays must be sampled uniformly
  - Different invocations can't use different indices
  - Can sample "dynamically uniform", eg runtime based index
- Upper limit on descriptor counts
- Discourages GPU-Driven rendering architectures

# Solution Space

- Descriptor Indexing
  - 1.3, optional in 1.2, or VK\_EXT\_descriptor\_indexing
  - Update descriptors after binding
  - Update unused descriptors
  - Relax requirement that all descriptors must be valid, even if unused
  - Non-uniform array indexing
- Buffer Device Address
  - 1.3, optional in 1.2, or VK\_KHR\_buffer\_device\_address
  - Directly access buffers through addresses without a descriptor
- Descriptor Buffers – VK\_EXT\_descriptor\_buffer
  - Manage descriptors directly
  - Similar to D3D12's descriptor model

# Shader Memory Layout

# What is the equivalent C Buffer of this?

```
layout(binding = 0) buffer block {  
    float a;  
    vec2 b;  
    vec2 c;  
};
```

# And of this?

```
layout(binding = 0) buffer block {  
    vec3 a;  
    vec2 b;  
    vec4 c;  
};
```

# Trick question!

- Didn't specify which layout to use
  - "Extended Alignment" AKA std140
  - "Base Alignment" AKA std430
  - "Scalar Block Layout" AKA scalar
- Each defines offset, alignment and padding
- So which should you use?

std140 / std430

```
struct C_Buffer {  
    float a;  
    float padding;  
    vec2 b;  
    vec2 c;  
    vec2 padding;  
};
```

# Scalar block layout!

- It uses C-like structure rules
- Matches expectations
- Enables easy sharing of data between host and device
  - No tedious padding and offsetting!
- Very commonly supported
- Great reference for all shader memory layouts
  - [https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/shader\\_memory\\_layout.adoc](https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/shader_memory_layout.adoc)

# Synchronization

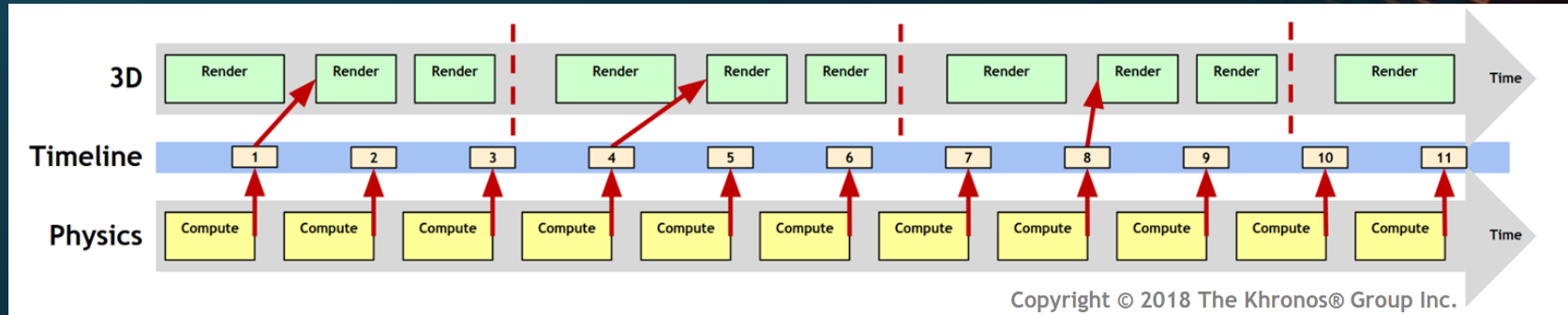
# Synchronization

- By far the hardest part of Vulkan
- Many different kinds of sync
  - Fence, Binary Semaphore, Event, Barrier
- Good synchronization are critical to good performance



# Timeline Semaphores

- Streamlines Host and Device sync
- Replaces fences and (binary) semaphores
- Is a monotonically increasing uint64\_t
- Have work wait for a value, increment to signal work is done
- Able to “Wait before signal”
- Does not currently work with swap chains



Copyright © 2018 The Khronos® Group Inc.

# Synchronization 2

- Improve usability and simplify the synchronization interface
- Specifies pipeline stages and access flags together
- More efficient Events
- Perform image memory barriers without transitions
- Makes using synchronization just that much easier
- Exhaustive discussion of the changes here
  - [https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/extensions/VK\\_KHR\\_synchronization2.adoc](https://github.com/KhronosGroup/Vulkan-Guide/blob/main/chapters/extensions/VK_KHR_synchronization2.adoc)

# Miscellaneous Features

# Shader Draw Parameters

- Adds additional shader builtins
  - BaseInstance
  - BaseVertex
  - DrawIndex
- Useful for indexing into buffers

# Indirect Rendering

- Generate draw commands on GPU
  - EX: Frustum culling
- But how many commands to use?
- DrawIndirectCount allows sourcing the count from GPU buffer

# VK\_EXT\_device\_generated\_commands

- New extension for GPU driven techniques
- Does way more than sourcing drawing parameters from GPU
  - Bind Vertex & Index buffers
  - Push constants
  - Pipelines & Shader objects
  - Draw calls
  - Compute dispatches
  - Raytracing
  - Mesh shading

# Conclusion

# Conclusion

- Vulkan-Guide is awesome
- Vulkan 1.0 was just a starting point
- The new stuff is worth the time and effort
  - All added for good reasons

# Thank you!

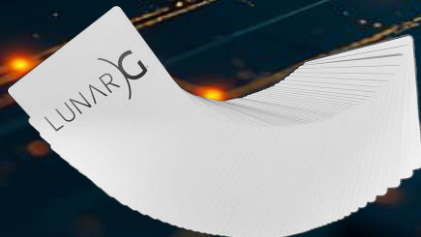
## Actions

**Download this  
Presentation**



<https://khr.io/1cr>

**Talk to us and  
get Swag!**



Visit the  
LunarG Sponsor Table

**Take the Annual  
Developers  
Survey**



<https://khr.io/1cq>

**Your Feedback  
Matters!**

Survey Results

- Are shared with the Khronos Vulkan Working Group
- Are used to drive development priorities throughout 2025

Survey Closes  
Wednesday, Feb. 19, 2025  
(GMT-7)