SIGGRAPH 2026
Los Angeles 19–23 JUL

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

How to write a Vulkan application in 2026

Iago Calvo Lista (Arm)
Ralph Potter (Samsung & Khronos Group)
Jose Emilio Muñoz Lopez (Arm)

Meet the Presenters

Ralph Potter

Chief Engineer (Samsung)
Chair of the Vulkan Working Group

Iago Calvo Lista

Senior Software Engineer (Arm)

Jose Emilio Muñoz Lopez

Staff Software Engineer (Arm)

Special thanks to: Elvar Unnthorsson

Senior Software Engineer (Arm)

What is Vulkan?

- Vulkan is an explicit, low-level GPU API — it exposes hardware concepts directly rather than abstracting them away
- Designed for predictable performance: the application owns memory allocation, synchronization, command submission, and hardware state
- Cross-platform: Windows, Linux, Android, and macOS
- Maintained by Khronos Group as an open, royalty-free standard
- Runs on a wide range of hardware: discrete desktop GPUs, mobile SoCs, and integrated graphics

Supplementary Materials

- This course is built around a simple Vulkan application that demonstrates the key concepts covered throughout
- Code, materials, and slides are available at: tinyurl.com/24fsppy9

Why This Course, Why Now?

- Vulkan launched in 2016 — a decade of evolution separates the API of today from most tutorials and textbooks
- The majority of available learning material targets Vulkan 1.0 patterns: verbose, brittle boilerplate written around worst-case hardware assumptions
- Best practices have changed substantially: features that once required extensions are now core; patterns once considered advanced are now the recommended baseline
- This course is explicitly aimed at 2026: we will teach you how to write Vulkan that is clean, maintainable, and takes advantage of what the spec now provides
- If you learned Vulkan before ~2022, some of what you know is worth unlearning

What Will We Cover?

Topic	Description
Device Creation and Initialization	Initializing your GPU, connecting a swapchain, vk-bootstrap
Vulkan Resources	Introducing images and buffers
Command Recording and Submission	Recording and submitting work to your GPU
Shaders	Authoring shaders and intro to Slang
Pipeline Management - Shader Objects	VK_EXT_shader_object
Drawing / Dynamic Rendering	Dynamic rendering
Descriptor Management	VK_EXT_descriptor_heap
Synchronization	Basic primitives, sync2 and timeline semaphore
Debugging	Tools for debugging your application

Supported Hardware

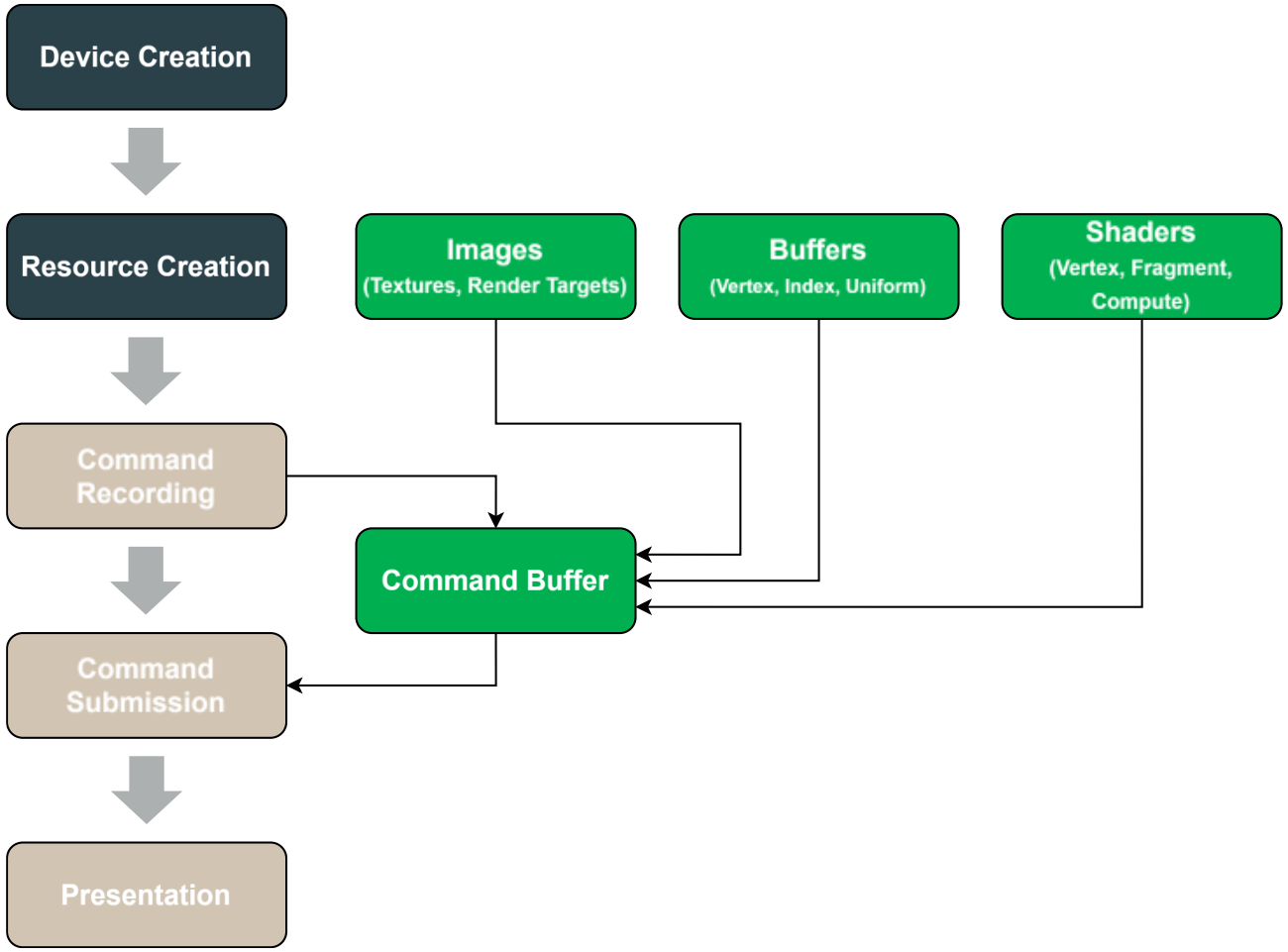
- This course showcases the latest Vulkan extensions. The level of support will vary based on your target hardware.
  - NVIDIA (Windows): Full support on the latest production drivers
  - AMD (Windows): Full support on the latest production drivers
  - Intel (Windows): Future support planned
  - MESA (Linux): Supported on AMD, Intel and lavapipe from Mesa 26.2 (August). Experimental support in earlier releases
  - Current production mobile drivers lack support for descriptor heaps and shader objects. Support for KHR versions of the extensions planned
- The features recommended in this course significantly ease usability.
  - For non-commercial projects, we recommend starting with the feature set described in this course
  - For commercial projects, you may need to review your target hardware and adjust accordingly

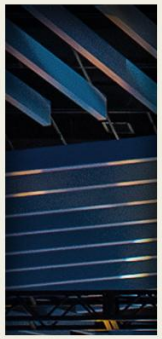
Abstraction Libraries

- Abstraction libraries can save you from writing a lot of mechanical boilerplate code. Strongly recommended while you learning

Library	Provides
Vulkan-HPP	Official C++ bindings for Vulkan - https://github.com/KhronosGroup/Vulkan-Headers
vk-bootstrap	Abstracts instance/device enumeration and creation - https://github.com/charles-lunarg/vk-bootstrap
GLFW	Abstracts platform-specific window management - https://www.glfw.org/

Vulkan Application Workflow







SIGGRAPH 2026
Los Angeles 19–23 JUL

Device Creation and Initialization

Device Creation And Initialization

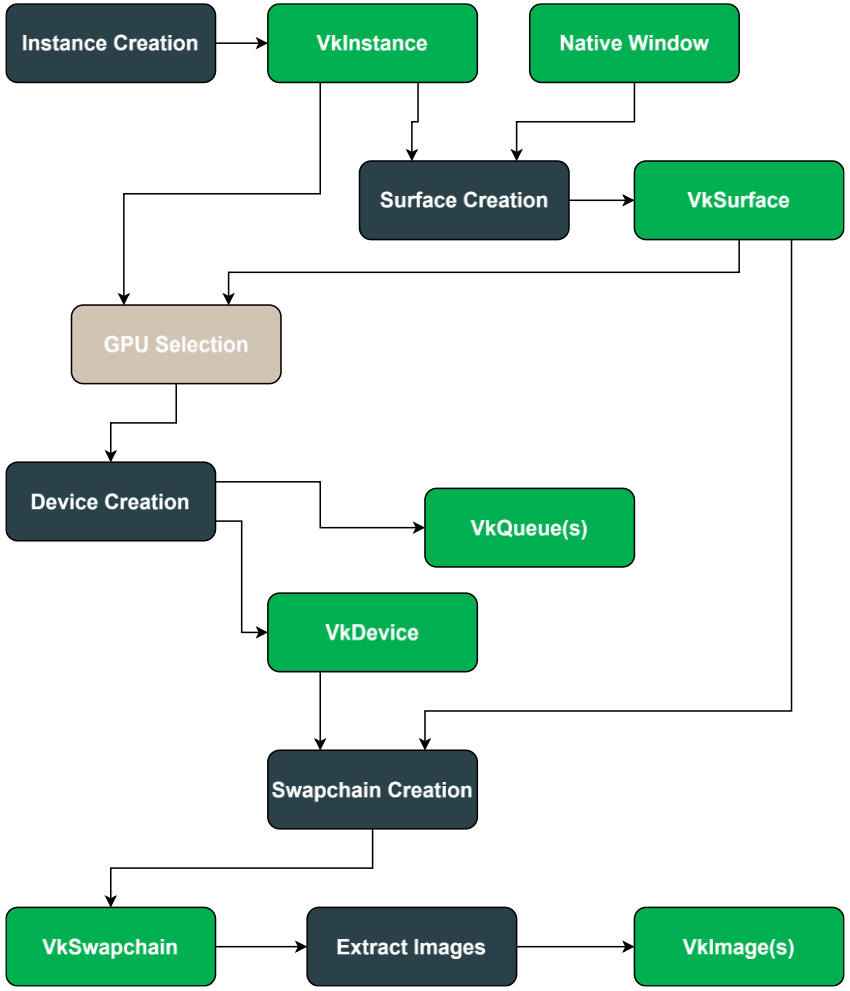
GOALS

- Initialize the Vulkan API
- Select a suitable GPU and configure the required features and extensions
- Connect the GPU to the window system for presentation
- Vulkan setup is notoriously verbose – libraries like vk-bootstrap can massively reduce the verbosity

Choosing Your API Baseline

Requirement	Provides
Vulkan 1.4	Baseline functionality (dynamic rendering, sync2, etc.)
VK_KHR_surface/VK_swapchain + platform specific extension	Manages presentation to display/application windows
VK_EXT_descriptor_heap	New resource binding model
VK_EXT_shader_object	Replaces monolithic pipelines with individual shaders
VK_EXT_extended_dynamic_state3	Makes more GPU state dynamic
VK_EXT_vertex_input_dynamic_state	Makes vertex pipeline input dynamic
VK_KHR_unified_image_layouts	Reduces the need to manage GPU image layouts and synchronization

Device Creation And Initialization



INITIALIZATION FLOW

- Create the instance and surface before GPU selection
- Select a GPU that supports both features and presentation
- Create the logical device and fetch the queues from it
  - **Host**: issue commands – usually CPU
  - **Device**: execute commands – usually GPU
    - Physical device (HW capabilities) and device (enabled capabilities)
- Build the swapchain, then create views for its images

Device Creation And Initialization

```
// vk-bootstrap greatly simplifies instance
// extension/layer selection and debug messenger setup.
auto instanceBuilder =
            vkb::InstanceBuilder()
                .set_app_name("Vulkan SIGGRAPH Tutorial")
                .set_engine_name("My Engine")
                .require_api_version(1, 4, 0)
                .request_validation_layers(true)
                .use_default_debug_messenger();
auto instance = instanceBuilder.build();
```

INSTANCE CREATION

- Set our supported Vulkan version
- Enable validation layers, and the default reporting mechanism
- Also implicitly enables platform specific WSI extensions

- Abstraction over vkCreateInstance / vkEnumerateInstanceExtensionProperties / vkEnumerateInstanceLayerProperties

Device Creation and Initialization

```
// We need to create the surface before physical-
// device selection so we can check presentation
// support when selecting the physical device.
VkSurfaceKHR surface = VK_NULL_HANDLE;
VkResult result =
        glfwCreateWindowSurface(instance, window,
                                nullptr,
                                &surface);
```

SURFACE CREATION

- VkSurface provides a platform-agnostic bridge between the platform-specific window system and Vulkan
- Vulkan surface creation APIs (vkCreateWin32SurfaceKHR / vkCreateAndroidSurfaceKHR) are platform-specific, but allow share the same design
- For beginners, we recommend using GLFW or SDL for your window and surface creation

Device Creation And Initialization

```
// Start from devices that meet the basic Vulkan requirements
vkb::PhysicalDeviceSelector selector { instance };

// Add requirements for required extensions, compatibility with our Window surface and minimum Vulkan version
const std::array requiredExtensions{
    VK_EXT_SHADER_OBJECT_EXTENSION_NAME,
    VK_EXT_VERTEX_INPUT_DYNAMIC_STATE_EXTENSION_NAME,
    . . .,
};
selector.set_surface(surface).set_minimum_version(1, 4)
    .add_required_extensions(requiredExtensions.size(),
                             requiredExtensions.data())

// Collect the matching GPU candidates
const auto physicalDevices = selector.select_devices();
```

PHYSICAL DEVICE SELECTION

- Your machine may have multiple physical GPUs
- Each has a unique handle, and an immutable set of properties/features/supported functionality
- We need to select one that meets all the requirements of our application
- vk-bootstrap excels here:
  - Add required API version, extensions, features, device type etc.
  - Populate a list of matching physical devices and perform further logic
  - Or just accept the first matching device

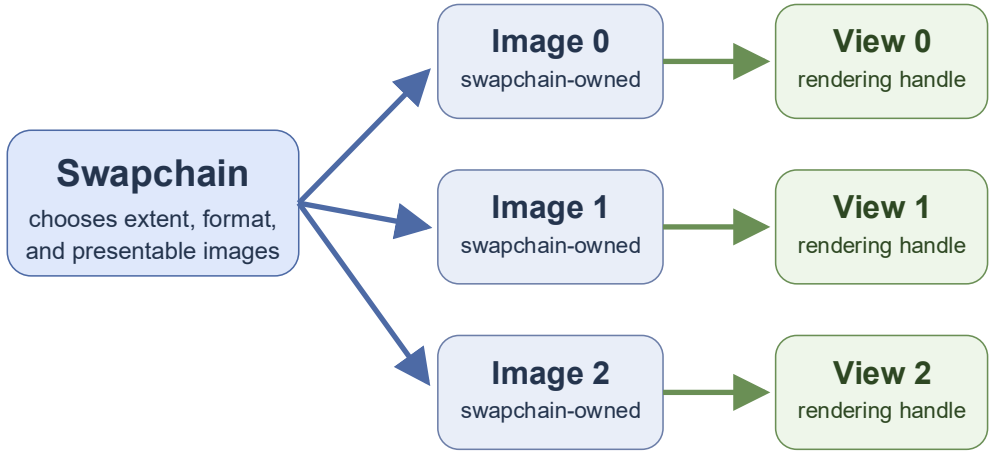
Device Creation And Initialization

```
// Build the logical device from the selected GPU
const auto deviceResult =
  vkb::DeviceBuilder(physicalDevice).build();
 vkbAssert(deviceResult, "Creating Vulkan logical
device");
 m_vkbData.m_device = deviceResult.value();

 // Fetch the queue handles used later for submission and
present
 const auto graphicsQueueResult =
   m_vkbData.m_device.get_queue(vkb::QueueType::graphics);
 const auto presentQueueResult =
   m_vkbData.m_device.get_queue(vkb::QueueType::present);
```

LOGICAL DEVICE AND QUEUES

- A logical device is your application's sandboxed interface to a physical GPU
- We need to enable the same set of extensions/features that we used in our selection query on the previous step
- vk-bootstrap will do this for us implicitly
- We also retrieve graphics and present queues from the device for future command submission

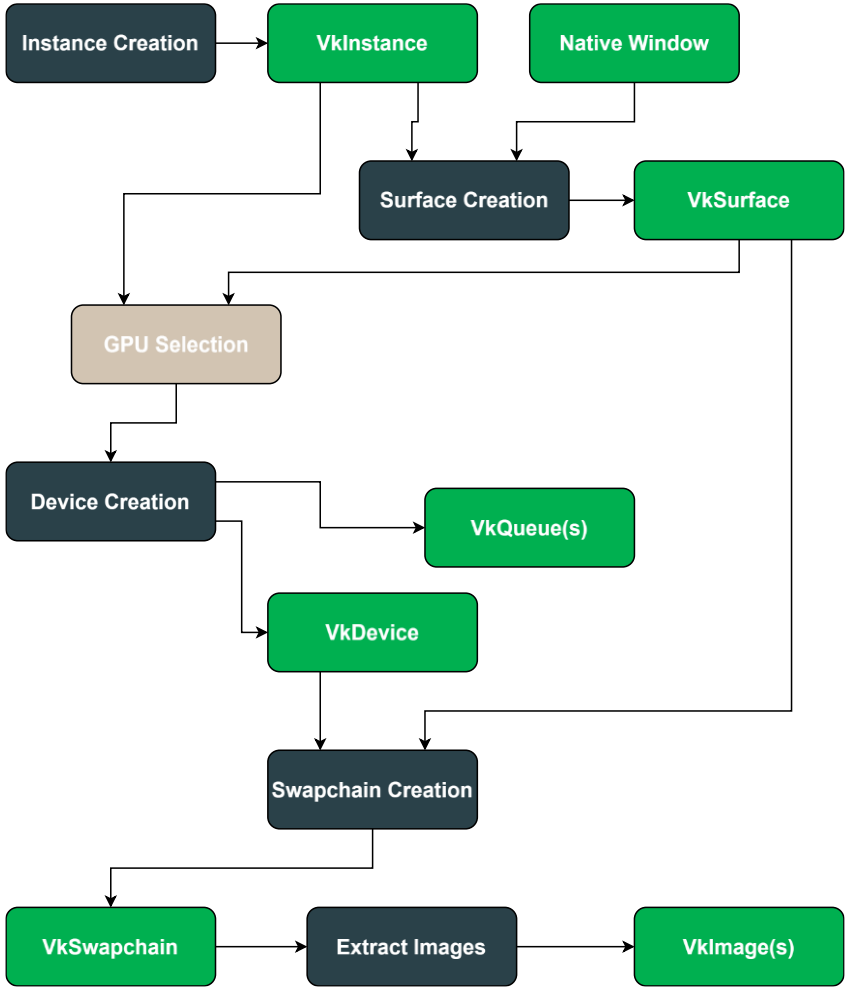
Device Creation And Initialization



SWAPCHAIN AND IMAGE VIEWS

- A swapchain is logically an array of images compatible with your systems display compositor/hardware
- Render loop consists of cycling through those images, rendering to the associated view, and submitting to a presentation queue on the device

Device Creation And Initialization



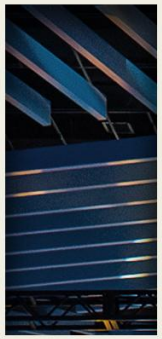
RECAP

- Create an VkInstance (top-level API interface)
- Create a VkSurface (connection to OS display system)
- Select a physical device (GPU) based on application requirements
- Create a logical device (VkDevice), enabling the same requirements
- Retrieve queues for future command submission
- Create a VkSwapchain to present output to the surface/window/display system
- Retrieve images from swapchain as targets for your rendering

Device Creation And Initialization

PRACTICAL TAKEAWAYS

- Device creation is verbose and time consuming, but largely mechanical
- GLFW (https://www.glfw.org/) or SDL (https://www.libsdl.org/) can help you abstract window and surface creation
- vk-bootstrap (https://github.com/charles-lunarg/vk-bootstrap) can vastly simplify device selection and creation
- For beginners, don’t waste your time handcrafting a perfect solution here – just use the libraries



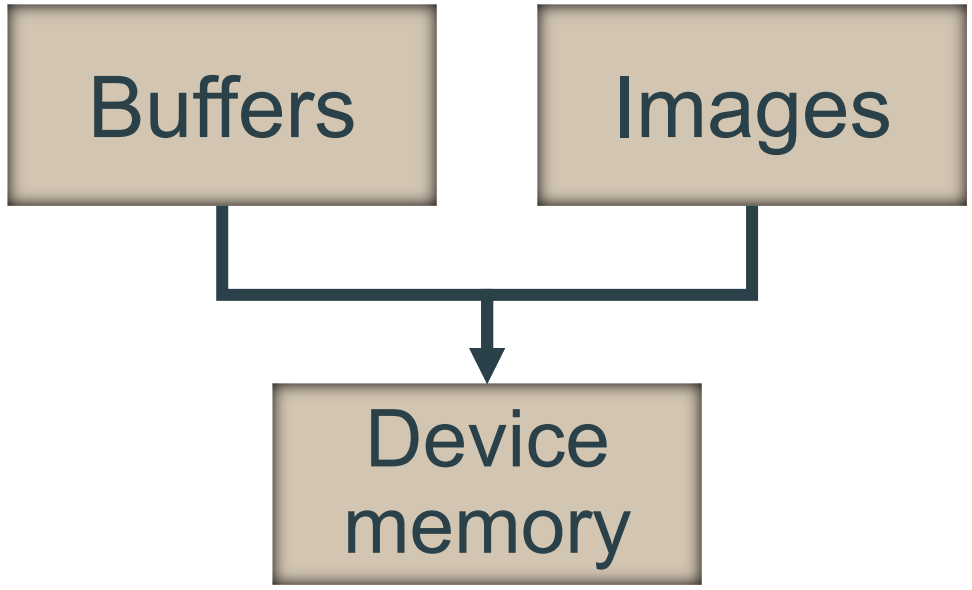


SIGGRAPH 2026
Los Angeles 19–23 JUL

Vulkan resources

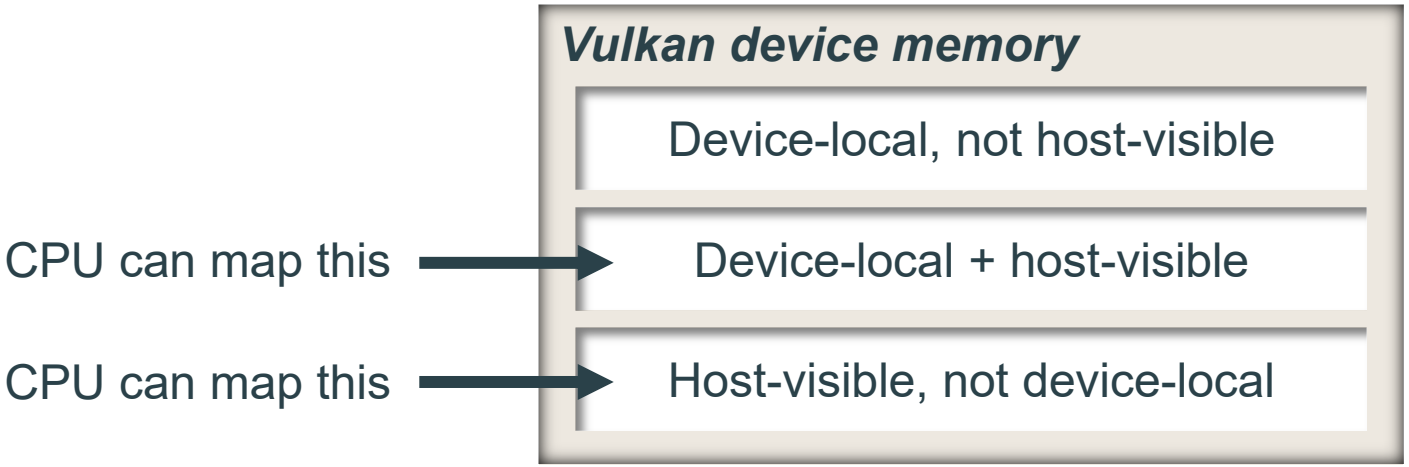
Vulkan resources: buffers and images

- Buffers and images hold GPU data
  - Buffers hold linear data (vertices, indices, constants)
  - Images hold formatted multidimensional data (textures, attachments)
- Device memory is allocated separately and bound to them
- The application manages object lifetime



Vulkan resources: device memory and host access

- Host-visible: the CPU can map it
- Device-local: intended for efficient device access
- Host visibility and device locality are independent properties
- Choose based on who reads and writes the data



Vulkan resources: memory flags

Property	Meaning
`eDeviceLocal`	Preferred for GPU-heavy access
`eHostVisible`	CPU can map the allocation
`eHostCoherent`	No explicit flush/invalidate for host cache visibility

- Host coherence removes cache maintenance, not synchronization

Vulkan resources: creating a buffer

```
const vk::BufferCreateInfo bufferInfo{
  .size = sizeof(CameraData), .usage = vk::BufferUsageFlagBits::eUniformBuffer,
  .sharingMode = vk::SharingMode::eExclusive};

vk::UniqueBuffer buffer = logicalDevice.createBufferUnique(bufferInfo);

vk::MemoryRequirements2 requirements =
  logicalDevice.getBufferMemoryRequirements2({.buffer = *buffer});

auto memory = logicalDevice.allocateMemoryUnique({
  .allocationSize = requirements.memoryRequirements.size,
  .memoryTypeIndex = findMemoryType(
    requirements.memoryRequirements.memoryTypeBits,
    vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent)});

logicalDevice.bindBufferMemory2({.buffer = *buffer, .memory = *memory});
```

Create

Query

Allocate

Bind

- Production applications typically suballocate memory with libraries such as Vulkan Memory Allocator (VMA)

Vulkan resources: creating an image

```
const vk::ImageCreateInfo imageInfo{
  .imageType = vk::ImageType::e2D, .format = textureFormat,
  .extent = { width, height, 1 }, .mipLevels = mipLevels, .arrayLayers = 1,
  .samples = vk::SampleCountFlagBits::e1, .tiling = vk::ImageTiling::eOptimal,
  .usage = vk::ImageUsageFlagBits::eTransferDst | vk::ImageUsageFlagBits::eSampled,
  .initialLayout = vk::ImageLayout::eUndefined};

vk::UniqueImage image = logicalDevice.createImageUnique(imageInfo);

vk::MemoryRequirements2 requirements =
  logicalDevice.getImageMemoryRequirements2({.image = *image});

auto memory = logicalDevice.allocateMemoryUnique({
  .allocationSize = requirements.memoryRequirements.size,
  .memoryTypeIndex = findMemoryType(
    requirements.memoryRequirements.memoryTypeBits,
    vk::MemoryPropertyFlagBits::eDeviceLocal)});

logicalDevice.bindImageMemory2({.image = *image, .memory = *memory }});
```

Create

Query

Allocate

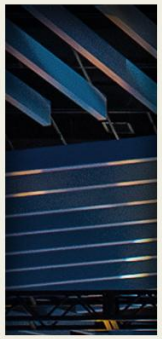
Bind

- An image view then selects how an image is accessed

Vulkan resources: staging buffers

- CPU writes the mapped staging buffer
- GPU copies into the final resource
- The staging resource remains alive until the GPU copy completes

```
memcpy(stagingBuffer.addressCPU, data.data(), data.size());

commandBuffer.copyBufferToImage2(copyInfo);
```

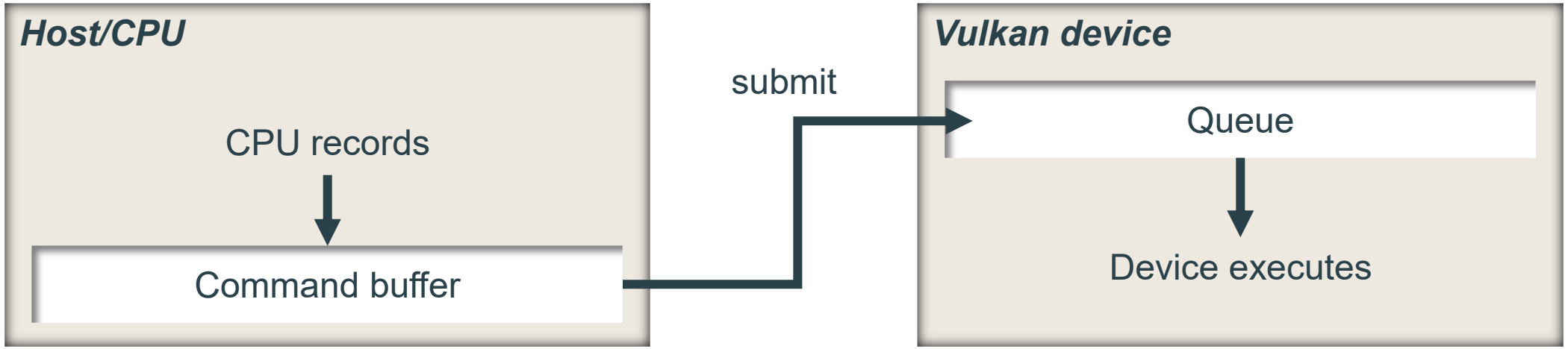




SIGGRAPH 2026
Los Angeles 19–23 JUL

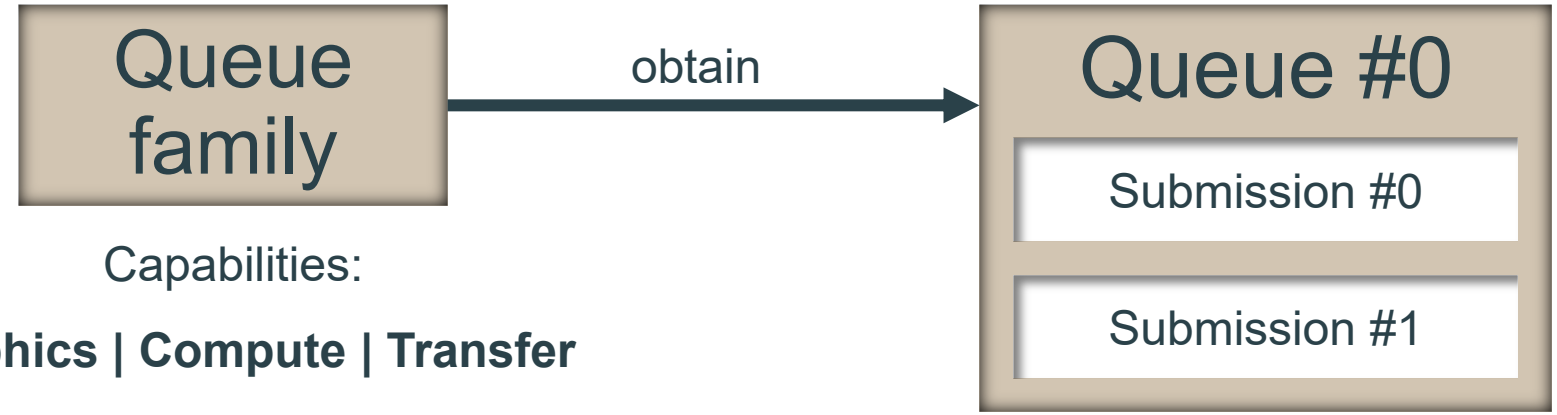
Command recording and submission

Command buffers



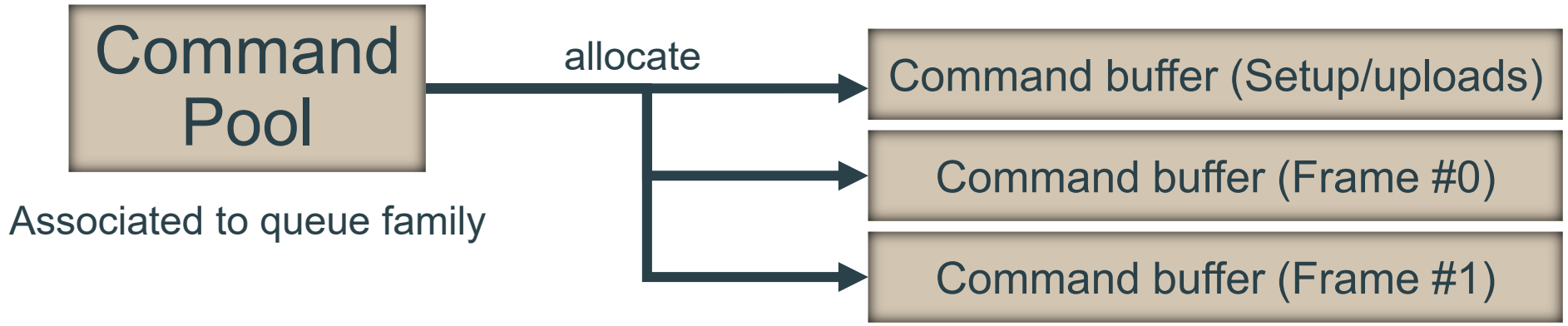
- Recording describes work, it does not execute it
- Completed command buffers are submitted to a queue
- CPU and device normally proceed asynchronously

Queues



- Queue families advertise capabilities
- Queues accept compatible submissions
- A single busy queue can keep many GPU stages working
- Multiple queues help only when the hardware and workload benefit
- This course submits command buffers to one graphics queue

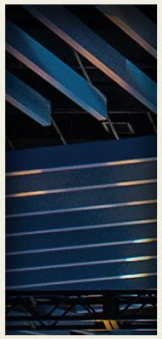
Command pools




```
auto m_commandPool = m_logicalDevice.createCommandPoolUnique({
  .flags = vk::CommandPoolCreateFlagBits::eResetCommandBuffer,
  .queueFamilyIndex = graphicsQueueFamily});

auto commandBuffers = m_logicalDevice.allocateCommandBuffers({
  .commandPool = *m_commandPool, .level = vk::CommandBufferLevel::ePrimary,
  .commandBufferCount = frameCount + 1});
```

Record a command buffer

```
// Record
commandBuffer.begin({});

commandBuffer.copyBufferToImage2(copyInfo);
// Other GPU operations...

commandBuffer.end();

const vk::CommandBufferSubmitInfo commandInfo{.commandBuffer = commandBuffer};

const vk::SubmitInfo2 submitInfo{
  .commandBufferInfoCount = 1,
  .pCommandBufferInfos = &commandInfo
};

// Submit (synchronization omitted here)
m_graphicsQueue.submit2(submitInfo);
```





SIGGRAPH 2026
Los Angeles 19–23 JUL

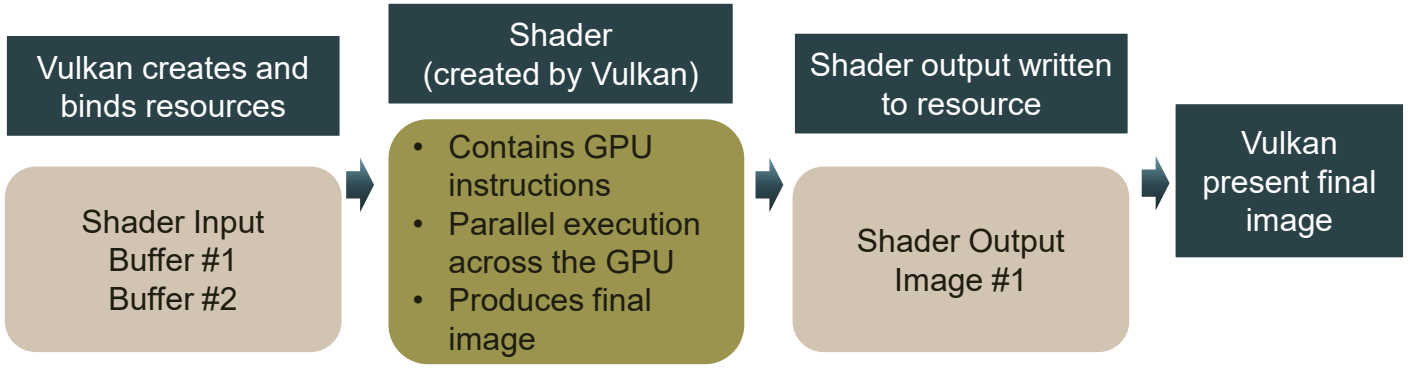
Authoring shaders

Shaders

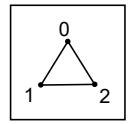
INTRODUCTION

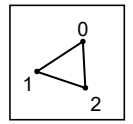
- Shaders are ~~small~~ programs that run on the GPU.
  - Massively parallel threads (SIMD-like)
- Offer explicit control over operations done by the GPU.
- They are the main way to define GPU operations.
  - Generate the visual output
- APIs like Vulkan focus on feeding data and state
  - Shaders consume data and produce useful outputs
  - Vulkan is still in charge of creating the shaders

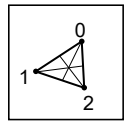
Vulkan host API calls	Shader code
Provide state and resources	Code that runs on the GPU
Bind descriptors and manage memory	Read input and writes output
Submit work and manage synchronization	Massively parallel

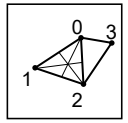


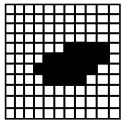
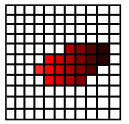
Shaders










Input Assembler

Vertex Shader

Tessellation

Geometry Shader

Rasterization

Fragment Shader

Color Blending

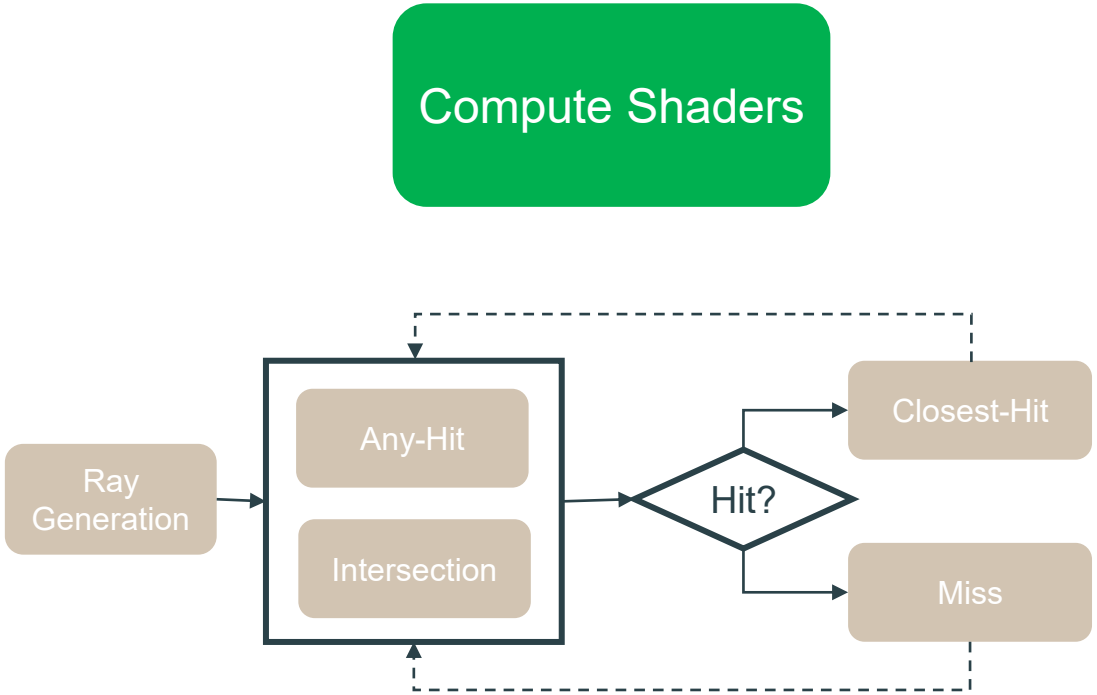
SHADERS IN VULKAN (GRAPHICS)

- Vulkan Graphics pipeline is divided in multiple stages
  - Fixed function: Configurable
  - Programmable: Application can provide shader code
- Main Programmable Stages:
  - Vertex shader: Runs once per input vertex, outputs position and attributes
  - Fragment shader: Runs for rasterized fragment (potential pixel) outputs depth and targets (color)
- Other stages: tessellation and geometry
  - Optional, not recommended

Shaders

SHADERS IN VULKAN (OTHER)

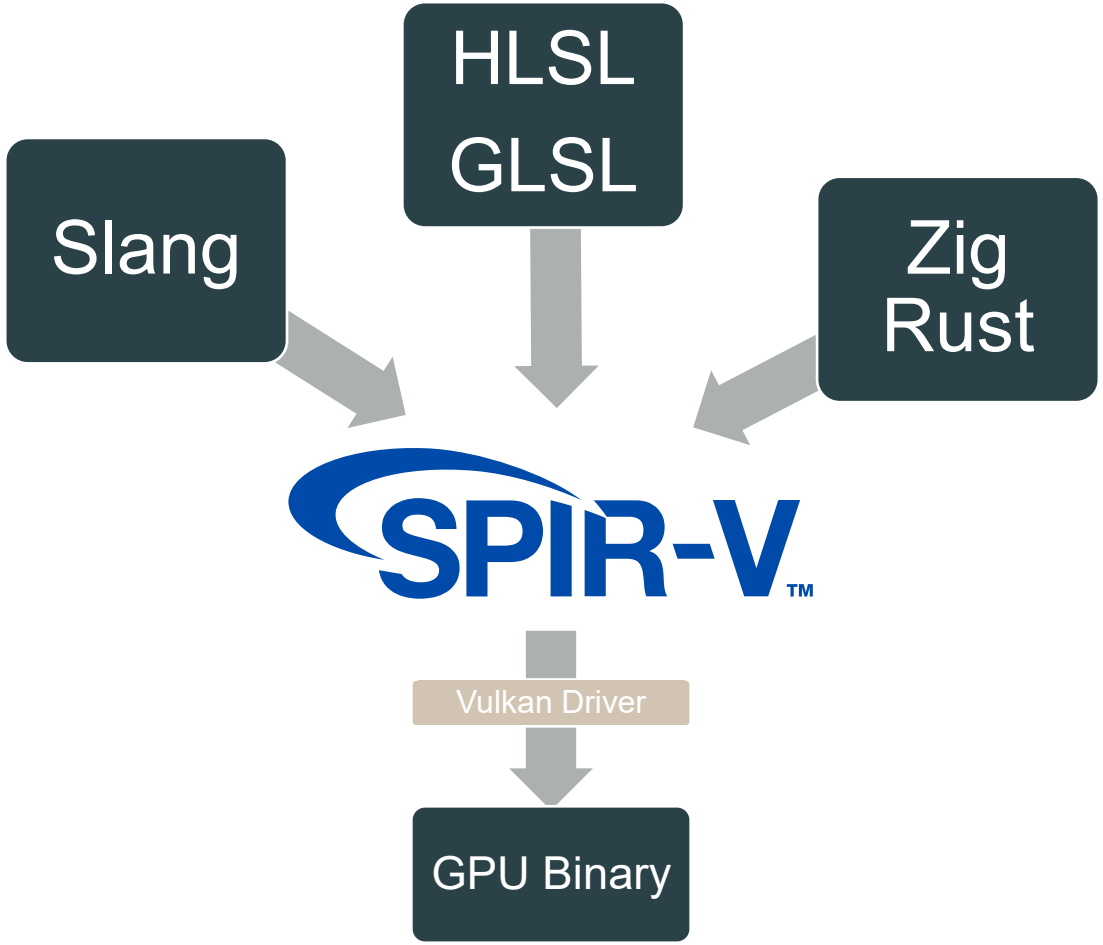
- Classic graphics pipeline is not the only option
- Compute shader: general-purpose GPU workloads
  - Not focus of this tutorial
  - Useful for parallel tasks such as simulation and advanced workloads
- Ray tracing shaders: Ray tracing pipeline
- Mesh and task shaders



Shaders

SHADER LANGUAGES (SPIR-V)

- Vulkan consumes SPIR-V an intermediate representation
- Developers use high-level languages, then it is compiled to SPIR-V
- Vulkan is language neutral to allow shader language experimentation and evolution
  - Support traditional languages like GLSL and HLSL
  - Experimentation with languages like Zig and Rust.



Shaders

SHADER LANGUAGES (SLANG)

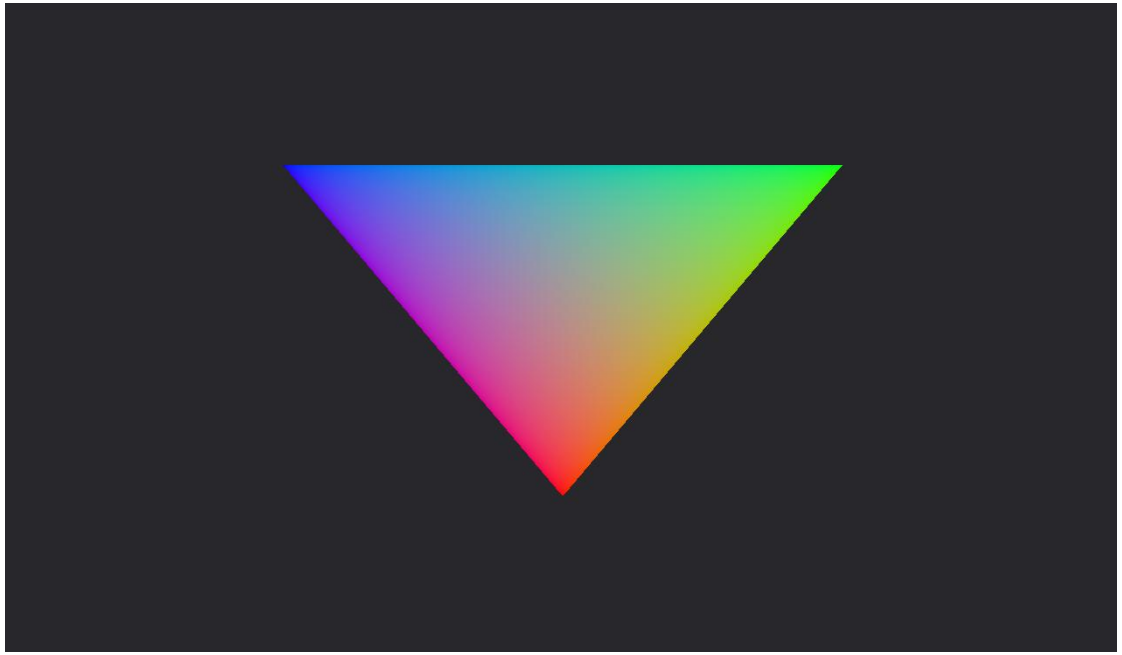
- Slang is a modern shader language hosted by the Khronos group.
  - Compiles to SPIR-V and other targets.
- Modern features.
  - Type inference, namespaces, interfaces, generics, etc.
- Advanced features:
  - Reflection API and binding generation.
  - Link time specialization.
  - Automatic differentiation
- This is not a shaders or Slang tutorial
  - Objective: show basics necessary to write shaders in Slang.
  - Shader authoring is a complex topic.

LEARN MORE ABOUT SLANG:

- https://shader-slang.org/slang/user-guide/index.html
- SIGGRAPH 2025 Introduction to Slang
  - https://www.youtube.com/watch?v=F7OS9Zvztmw

Shaders

```
struct VSOutput
{
  float4 position : SV_Position;
  float3 color;
};

[shader("vertex")]
VSOutput vertexMain(uint vertexID : SV_VertexID)
{
  float2 positions[3] = { float2( 0.0, 0.5), float2( 0.5, -0.5), float2(-0.5, -0.5) };
  float3 colors[3]    = { float3(1.0, 0.0, 0.0), float3(0.0, 1.0, 0.0), float3(0.0, 0.0, 1.0)};

  VSOutput output;
  output.position = float4(positions[vertexID], 0.0, 1.0);
  output.color    = colors[vertexID];

  return output;
}

[shader("fragment")]
float4 fragmentMain(VSOutput input)
{
  return float4(input.color, 1.0);
}
```



Shaders

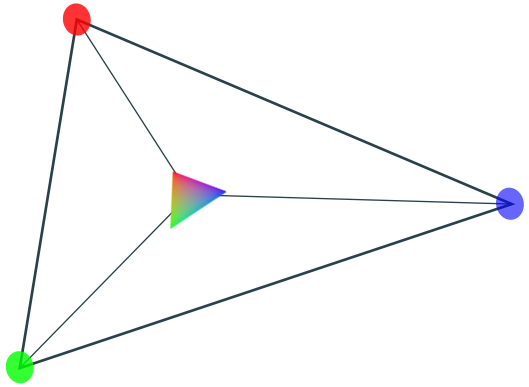
```
struct VSOutput
{
  float4 position : SV_Position;
  float3 color;
};

[shader("vertex")]
VSOutput vertexMain(uint vertexID : SV_VertexID)
{
  float2 positions[3] = { float2( 0.0, 0.5), float2( 0.5, -0.5), float2(-0.5, -0.5) };
  float3 colors[3]    = { float3(1.0, 0.0, 0.0), float3(0.0, 1.0, 0.0), float3(0.0, 0.0, 1.0)};

  VSOutput output;
  output.position = float4(positions[vertexID], 0.0, 1.0);
  output.color    = colors[vertexID];

  return output;
}

[shader("fragment")]
float4 fragmentMain(VSOutput input)
{
  return float4(input.color, 1.0);
}
```

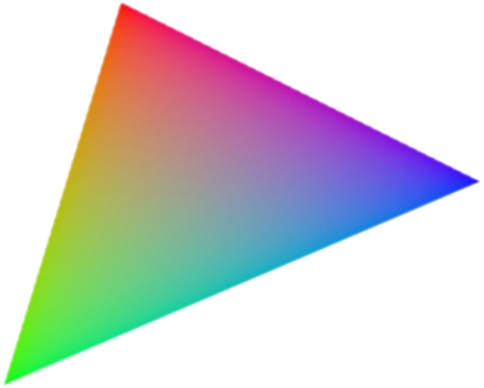
- Vertex shaders run once per vertex
- We return a struct
  - Defines the values passed from vertex to fragment shader.

Shaders

```
struct VSOutput
{
  float4 position : SV_Position;
  float3 color;
};

[shader("vertex")]
VSOutput vertexMain(uint vertexID : SV_VertexID)
{
  float2 positions[3] = { float2( 0.0, 0.5), float2( 0.5, -0.5), float2(-0.5, -0.5) };
  float3 colors[3]    = { float3(1.0, 0.0, 0.0), float3(0.0, 1.0, 0.0), float3(0.0, 0.0, 1.0)};

  VSOutput output;
  output.position = float4(positions[vertexID], 0.0, 1.0);
  output.color    = colors[vertexID];

  return output;
}

[shader("fragment")]
float4 fragmentMain(VSOutput input)
{
  return float4(input.color, 1.0);
}
```

- We are required to set the position of the vertex.
  - 4D Homogeneous coordinates
- We set values to be interpolated across the triangle





Shaders

```
struct VSOutput
{
  float4 position : SV_Position;
  float3 color;
};

[shader("vertex")]
VSOutput vertexMain(uint vertexID : SV_VertexID)
{
  float2 positions[3] = { float2( 0.0, 0.5), float2( 0.5, -0.5), float2(-0.5, -0.5) };
  float3 colors[3]    = { float3(1.0, 0.0, 0.0), float3(0.0, 1.0, 0.0), float3(0.0, 0.0, 1.0)};

  VSOutput output;
  output.position = float4(positions[vertexID], 0.0, 1.0);
  output.color    = colors[vertexID];

  return output;
}

[shader("fragment")]
float4 fragmentMain(VSOutput input)
{
  return float4(input.color, 1.0);
}
```

- The example is using a hardcoded position and attributes
  - No descriptors needed
- Fragment shader runs once per fragment (pixel)
- Fragment shader receives the interpolated values
- Fragment shader returns a color for each fragment

Shaders

```
[[vk::binding(0, 0)]]
ConstantBuffer<CameraData> g_camera;

[[vk::binding(1, 0)]]
StructuredBuffer<ObjectDataRecord> g_objects[];

[shader("vertex")]
VertexOutput vertexMain(
  AppVertexInput input,
  uint instanceIndex : SV_InstanceID)
{
  ObjectData objectData = g_objects[instanceIndex][0];

  VertexOutput output;
  float4 worldPosition =
       mul(objectData.model, float4(input.pos, 1.0));
  output.position = mul(g_camera.viewProjection, worldPosition);
  output.uv       = input.uv;

  return output;
}
```

```
struct AppVertexInput
{
 [[vk::location(0)]] float3 pos;
 [[vk::location(1)]] float2 uv;
};
```

```
struct CameraData
{
  float4x4 viewProjection;
};
struct ObjectData
{
  float4x4 model;
};
```

- Usually, vertex attributes provide per-vertex input values
  - Contain properties for each input vertex, like UV, color, position
- Bindings connect Slang shader variables to Vulkan resources.
- Slang binding and locations are optional
- Resource arrays / bindless resources are possible
  - Advanced version with `DescriptorHandle`
- Descriptor setup and mapping is covered later

Shaders

```
[[vk::binding(2, 0)]]
Sampler2D g_albedoTexture;

struct FragmentInput
{
  float2 uv;
};

[shader("fragment")]
float4 fragmentMain(FragmentInput input)
{
  float4 texel = g_albedoTexture.Sample(input.uv);
  if (texel.a < 0.5) {
    discard;
  }
  return float4(texel.rgb, 1.0);
}
```

- Images and textures are accessed through samplers
- Interpolate the texture using UV coordinates
  - Handles mip levels and derivatives
- Use discard to drop unwanted fragments
- This is enough to read the more complex shader examples.
  - The code repo has some PBR examples
  - Learning Slang is not the objective of the course

Shaders

```
import materialLighting; // expose MaterialsLight namespace

typealias MatProperties = MaterialsLight::MaterialProperties;

StructuredBuffer<DescriptorHandle<StructuredBuffer<MatProperties>>> materials;

interface IMaterial {
  [Differentiable] float3 eval(float2 uv);
}

struct TextureMaterial : IMaterial {
  MatProperties mat;

  [Differentiable] float3 eval(float2 uv) {
    return MaterialsLight::lambert(mat, uv);
  }
}

[Differentiable] float3 shade<T : IMaterial>(T material, float2 uv) {
  return material.eval(uv);
}

[shader("fragment")]
float4 fragmentMain(FragmentInput input) {
  MatProperties mat = materials[NonUniformResourceIndex(input.id)][0];
  DifferentialPair<float2> duv = diffPair(input.uv, float2(1.0, 0.0));
  var color = fwd_diff(shade<TextureMaterial>)(TextureMaterial(mat), duv).p;
  return float4(color, 1.0);
}
```

- Note: This is a very complex shader. The objective is to showcase Slang features rather than producing useful results

Shaders

```
import materialLighting; // expose MaterialsLight namespace

typealias MatProperties = MaterialsLight::MaterialProperties;
[[vk::binding(0, 0)]] // Annotations can be autogenerated from reflection
StructuredBuffer<DescriptorHandle<StructuredBuffer<MatProperties>>> materials;

interface IMaterial {
  [Differentiable] float3 eval(float2 uv);
}

struct TextureMaterial : IMaterial {
  MatProperties mat;

  [Differentiable] float3 eval(float2 uv) {
    return MaterialsLight::Lambert(mat, uv);
  }
}

[Differentiable] float3 shade<T : IMaterial>(T material, float2 uv) {
  return material.eval(uv);
}

[shader("fragment")]
float4 fragmentMain(FragmentInput input) {
  MatProperties mat = materials[NonUniformResourceIndex(input.id)][0];
  DifferentialPair<float2> duv = diffPair(input.uv, float2(1.0, 0.0));
  var color = fwd_diff(shade<TextureMaterial>)(TextureMaterial(mat), duv).p;
  return float4(color, 1.0);
}
```

- Note: This is a very complex shader. The objective is to showcase Slang features rather than producing useful results

Modern Language Features

- Modules and imports
- Aliases
- Generics and Interfaces
- Namespaces
- Type inference / deduction
- Operator Overloading
- Tuple, optional, pointer, properties

Advanced toolchain options

- Binding auto-deduction & Reflections
- Multiple entry points in SPIR-V
- Compile to SPIR-V, DXIL (DX12), WSL(WebGPU), etc.

Machine Learning: Automatic differentiation & SlangPy

- Slang is only an option: HLSL and GLSL support SPIR-V

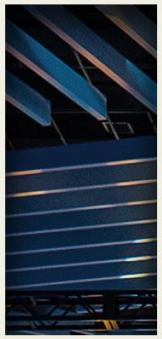
Shaders

FURTHER LEARNING

- **Graphics pipeline**
  - Vulkan tutorial: Graphics pipeline introduction - https://docs.vulkan.org/tutorial/latest/03_Drawing_a_triangle/02_Graphics_pipeline_basics/00_Introduction.html
- **Slang**
  - Slang user guide - https://shader-slang.org/slang/user-guide/index.html
  - SIGGRAPH 2025 Introduction to Slang - https://www.youtube.com/watch?v=F7OS9Zvztmw
- **Compute shaders**
  - Vulkan tutorial: Compute shaders - https://docs.vulkan.org/tutorial/latest/11_Compute_Shader.html

SLANG AT SIGGRAPH 2026

- Tuesday, 21 July 2026 10:15am - 11:45am PDT
  - Hands-on Course: Introduction to Slang: The Next-Generation Shading Language
    - https://s2026.conference-schedule.org/presentation/?id=gensub_172&sess=sess258
  - Hands-on Course: NVIDIA: Hands-On Machine Learning with Slang: Building High-Performance GPU Inference Pipelines
    - https://s2026.conference-schedule.org/presentation/?id=gensub_263&sess=sess261
- Slang in Action: Modern Shading for Graphics, Compute, and AI
  - https://s2026.conference-schedule.org/presentation/?id=bof_122&sess=sess300
  - Wednesday, 22 July 2026 1:30pm - 2:00pm PDT, JW Marriott - Plaza 1
  - Birds of the Feather





SIGGRAPH 2026
Los Angeles 19–23 JUL

Pipeline and Shader Management

Introducing VK_EXT_shader_object

Pipeline Management

```
slangc \
    "shaders/basic.vert.slang" \
    -entry "vertexMain" \
    -stage vertex \
    -target spirv  \
    -profile spirv_1_5 -matrix-layout-column-major \
    -reflection-json "albedo.vert.json" \
    -o "albedo.vert.spv"
```

```
slangc --help
```

HOW TO USE OUR SHADERS

- We have written our Slang shaders
- `slangc` supports multiple options
- Shader compilation to SPIR-V can be costly

Pipeline Management

```
function(add_slang_shader target_name source_file entry_point stage output_file)
  set(reflection_json "${output_file}.json")
  add_custom_command(
    OUTPUT "${output_file}" "${reflection_json}"
    COMMAND "${SLANGC_EXECUTABLE}" # slangc executable path
            "${source_file}"
            -entry "${entry_point}" # main function (IE: vertexMain fragmentMain)
            -stage "${stage}" # vertex, fragment, etc.
            -target spirv -profile spirv_1_5 -matrix-layout-column-major
            -reflection-json "${reflection_json}" # Generate reflection data
            -o "${output_file}"
    DEPENDS "${source_file}"
    VERBATIM)
  add_custom_target("${target_name}"
    DEPENDS "${output_file}" "${reflection_json}")
endfunction()
```

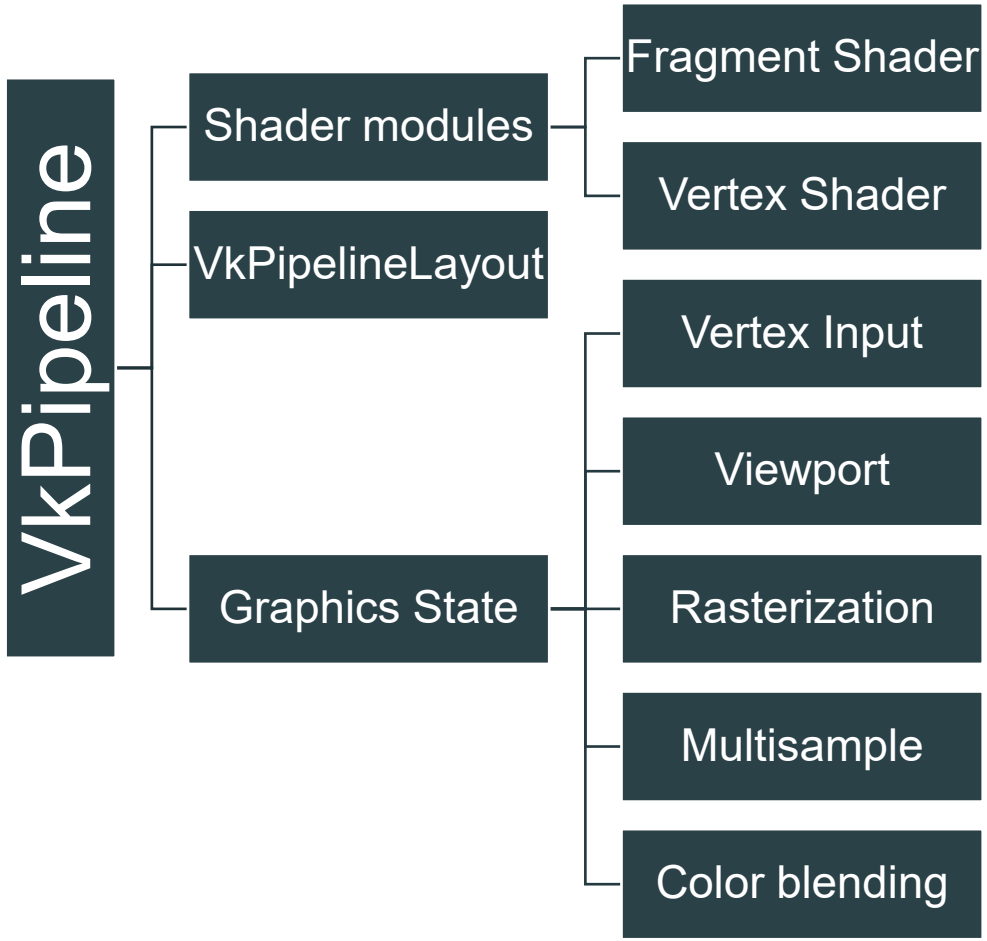
HOW TO USE OUR SHADERS

- We have written our Slang shaders
- `slangc` supports multiple options
- Shader compilation to SPIR-V can be costly
  - `slangc` part of the build process (CMake)
- How do we use the compiled SPIR-V?

Pipeline Management

PIPELINES

- The traditional way to use our shader is to create a `VkPipeline`
- Pipelines are monolithic
  - Most graphics state needed to render is baked into the pipeline
- Unpredictable shader compilation stutters caused problems in OpenGL
- Vulkan 1.0 expected most state to be given upfront
  - GPU compiler has more optimization opportunities
  - Compiling a shader is predictable



Pipeline Management

```
// Create vk::ShaderModule objects using the SPIR-V
vk::ShaderModuleCreateInfo vertShaderModuleInfo {
    .codeSize = vertSpirv.size(), .pCode = vertSpirv.data() /* ... */ };
vk::ShaderModule vertShaderModule = device.createShaderModule(vertShaderModuleInfo);

// Describe shader stages
vk::PipelineShaderStageCreateInfo vertShaderStage {
    .stage  = vk::ShaderStageFlagBits::eVertex,
    .module = vertShaderModule, .pName  = "main" };

// Describe fixed-function pipeline state
vk::PipelineVertexInputStateCreateInfo   vertexInput;
vk::PipelineInputAssemblyStateCreateInfo inputAssembly;
// ...

// Create pipeline layout if using descriptors
vk::PipelineLayoutCreateInfo layoutInfo{/* ... */};
vk::PipelineLayout pipelineLayout = device.createPipelineLayout(layoutInfo);

// Create the graphics pipeline
vk::GraphicsPipelineCreateInfo pipelineInfo {
  /* Set fixed function state, shader stages, layout, ... */ };
vk::Pipeline pipeline = device.createGraphicsPipeline(pipelineCache, pipelineInfo).value;
```

- Note: This sample uses `VkPipeline`, we recommend shader objects

PROBLEM WITH PIPELINES

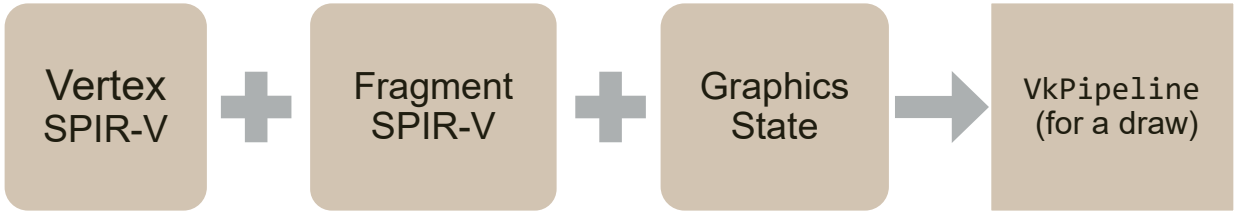
- Lots of boilerplate code to create a pipeline
- Difficult to manage the cache: cache on top of `VkPipelineCache`
- Applications are expected to specify all state upfront
  - Games are very dynamic: Difficult to know all pipelines upfront
  - Small state changes require recompiling the entire pipeline
  - Huge number of Pipeline State Objects (PSOs) to manage

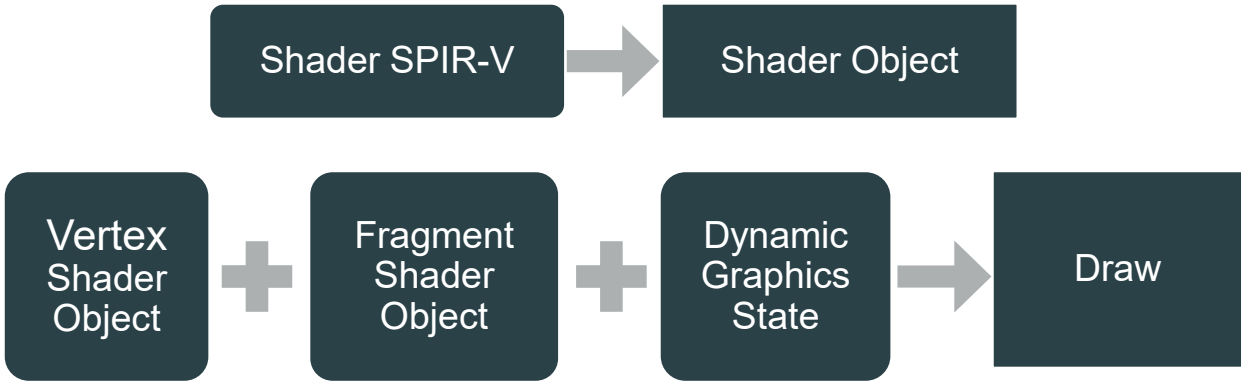
Pipeline Management

SHADER OBJECTS

- Separate graphics state from shader code
- We can compile each shader separately
  - Bind each shader to a command buffer
  - Shader decisions are more dynamic
- Reduce number of PSOs

VkPipelines	Shader Objects
Set most graphics state at creation	Separate shaders and graphics state
All shader states compiled together	Shaders can be compiled separately





Pipeline Management (Shader Objects)

```
// Enable VK_EXT_shader_object in device creation

ShaderInfo createShaderObjects() {
  // Descriptor mapping not needed if shaders do not use resources, explained later
  vk::ShaderDescriptorSetAndBindingMappingInfoEXT* mappingInfo = nullptr;
  std::vector<vk::ShaderCreateInfoEXT> createInfos = {
      vk::ShaderCreateInfoEXT{
          .pNext = mappingInfo /* = nullptr*/,
          .flags = vk::ShaderCreateFlagBitsEXT::eDescriptorHeap,
          .stage = vk::ShaderStageFlagBits::eVertex,
          .codeType = vk::ShaderCodeTypeEXT::eSpirV,
          .codeSize = vertexShaderSpirv.size(),
          .pCode = vertexShaderSpirv.data(),
          // ...
      },
      vk::ShaderCreateInfoEXT{
          // ...
          .stage = vk::ShaderStageFlagBits::eFragment,
          // ...
      }
  };
  auto shaders = m_logicalDevice.createShadersEXTUnique(createInfos).value;
  return ShaderInfo{.m_vertexShader = std::move(shaders[0]),
                    .m_fragmentShader = std::move(shaders[1])};
}
```

SHADER OBJECTS (CREATION)

- You can create a shader directly from SPIR-V
  - Removes a lot of boilerplate
  - Easier and more straightforward alternative

Pipeline Management (Shader Objects)

```
void recordRenderingCommands(vk::CommandBuffer cmdBuf) {

  // Set dynamic state ...

  for (const auto& draw : drawData) {
    if (draw.shaders != currentShaders) {
      // Bind fragment shader to the command buffer
      std::array stages{ vk::ShaderStageFlagBits::eFragment};
      std::array<vk::ShaderEXT, 1> shaders{
             draw.shaders.m_fragmentShader.get()};
      cmdBuf.bindShadersEXT(stages, shaders);

      // Bind vertex shader. Could have been combined with previous call.
      stages = { vk::ShaderStageFlagBits::eVertex };
      shaders = {draw.shaders.m_vertexShader.get()};
      cmdBuf.bindShadersEXT(stages, shaders);

      currentShaders = draw.shaders;
    }

    // ...
    // Draw ...
  }
//...
}
```

SHADER OBJECTS (USE)

- You use new methods to set the state
  - Dynamic and more flexible than `VkPipeline`
- `bindShadersEXT`: Can change individual shaders
  - Shaders are bound directly to a command buffer
  - Supports changing multiple shaders in one call

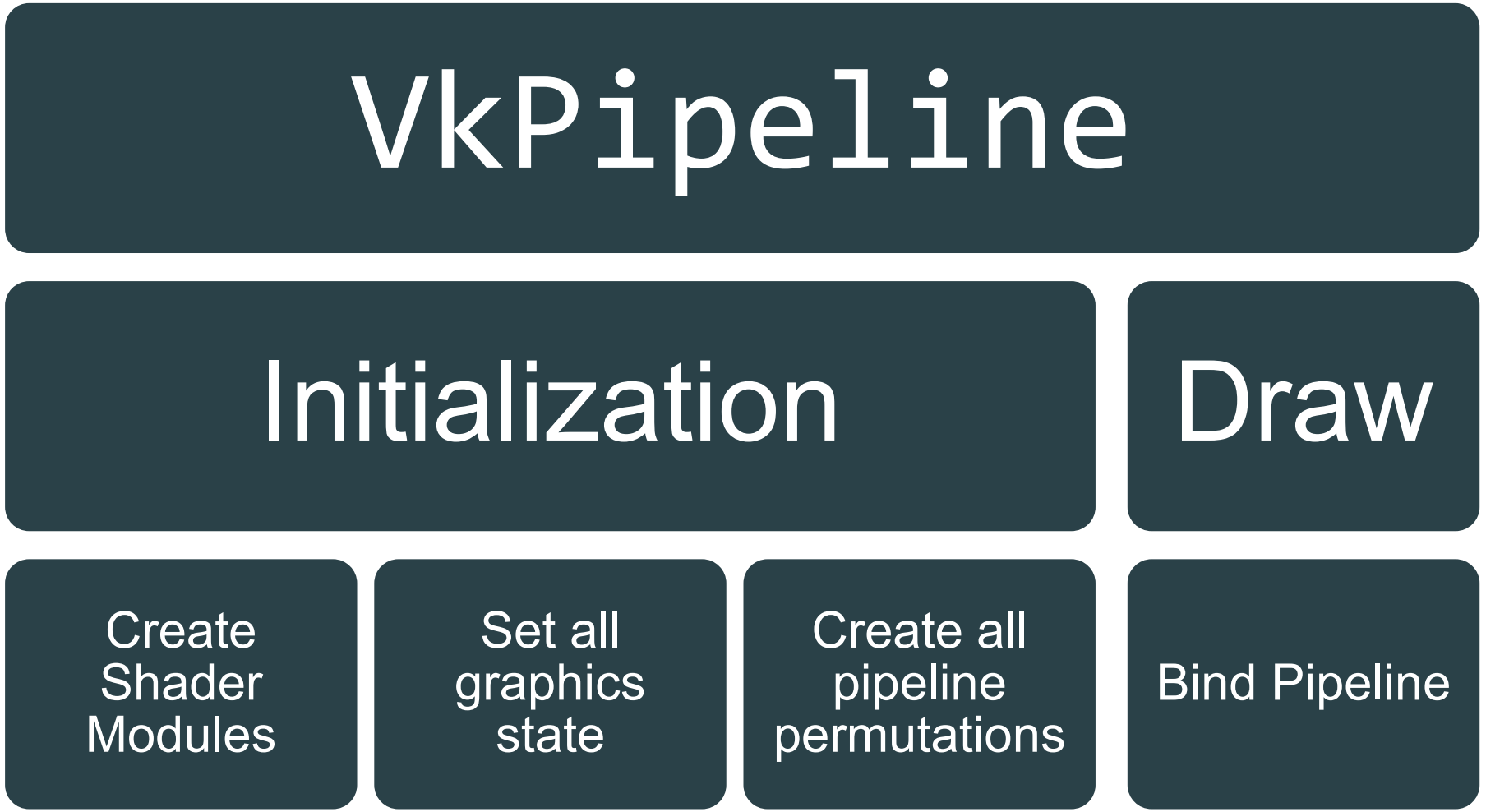
Pipeline Management (Shader Objects)

```
// Enable dynamic state extensions at device creation.
// Using shader objects so we do not need to set it at pipeline creation

void recordRenderingCommands(vk::CommandBuffer cmdBuf) {
    // Set common dynamic state
    cmdBuf.setCullModeEXT(vk::CullModeFlagBits::eBack);
    cmdBuf.setFrontFaceEXT(vk::FrontFace::eClockwise);
    cmdBuf.setPrimitiveTopologyEXT(vk::PrimitiveTopology::eTriangleList);
    // ...

    for (const auto& draw : drawData) {
      // Some draw objects might have different state
      if (draw.mesh.vertexInputState != currentVertexInputState) {
         cmdBuf.setVertexInputEXT(...);
         currentVertexInputState = draw.mesh.vertexInputState;
      }
      if (drawDataIt.shaders != currentShaders) {
         cmdBuf.bindShadersEXT(...);
         currentShaders = draw.shaders;
      }
     // ...
    // Draw
  }
//...
}
```

DYNAMIC STATE

- Shader objects move state from pipeline creation to command recording.
  - Dynamic state: we cannot bake state into the `VkPipeline`
- Dynamic state can be used with `VkPipeline`
  - We can specify some state as dynamic at creation and omit setting it up front.
  - Easy to change specific state per call
- Dynamic state is very flexible
  - Reduce the number of pipelines to create
  - GPU overhead on some implementations
- Vulkan has added a lot of dynamic state using extensions
  - `VK_EXT_extended_dynamic_state / state2 / state3`
  - `VK_EXT_vertex_input_dynamic_state`
  - `VK_EXT_attachment_feedback_loop_dynamic_state`
  - `VK_EXT_color_write_enable`

Pipeline Management (Shader Objects)



- Very rigid
- We set all state and PSO permutations at the start

Pipeline Management (Shader Objects)

Shader Objects

Initialization

Create Shader Objects

Draw

Bind shaders

Set Dynamic state

- Flexible
- We set graphics state during command recording
- Easy to change state for some draw objects
  - Per draw data: has shaders and graphics state
- Easy to combine vertex and fragment shaders

Pipeline Management (Shader Objects)

VkPipeline

- Initialization
  - Create Shader Modules
  - Set all graphics state
  - Create all pipeline permutations
- Draw
  - Bind Pipeline

Shader Objects

- Initialization
  - Create Shader Objects
- Draw
  - Bind shaders
  - Set Dynamic State

Pipeline Management (Shader Objects)

```
vk::ShaderCreateInfoEXT shaderObjectCI(
  const ShaderCreationData& shaderData, bool enableLinking) {
  return vk::ShaderCreateInfoEXT{
      // ...
      .stage = shaderData.stage,
      .flags = vk::ShaderCreateFlagBitsEXT::eDescriptorHeap |
               (enableLinking ? vk::ShaderCreateFlagBitsEXT::eLinkStage
                              : vk::ShaderCreateFlagBitsEXT(0)),
      // We are ignoring mesh, tessellation, geometry shaders
      // but the logic in nextStage is similar for them
      .nextStage = (shaderData.stage == vk::ShaderStageFlagBits::eVertex )
                 ? vk::ShaderStageFlagBits::eFragment : vk::ShaderStageFlagBits{}
      // ...
  };
}
ShaderInfo createShaderObjects() {
 const bool enableLinking = true;
 std::array createInfos = {
   shaderObjectCI({vk::ShaderStageFlagBits::eVertex, vertexShaderSpirv}, enableLinking),
   shaderObjectCI({vk::ShaderStageFlagBits::eFragment, fragShaderSpirv}, enableLinking)
 };
 // If we have enableLinking, we need to send all linked shaders in the same call
 shaderInfo = m_logicalDevice.createShadersEXTUnique(createInfos).value;
```

LINKING SHADER OBJECTS

- Shader objects can be linked together
  - **Unlinked**: Easy to reuse shaders, fewer permutations.
  - **Linked**: More optimization opportunities → Better GPU performance
- Linking shaders:
  - Specific flags `ShaderCreateFlagBitsEXT::eLinkStage`
  - Need to be together in the same call to `createShadersEXT`
  - Need to specify the required information
  - Better GPU performance
  - Need to be bound together

Pipeline Management (Shader Objects)

```
ShaderInfoJob createShaderObjects() {
  // Check if the shader group is already compiled
  if (availableInCache(shaders, shaderInfoJob.optimized)) {
      return shaderInfoJob;
  }

  // Try to compile the individual shaders without linking
  shaderInfoJob.fallback.m_vertexShader =
      getAsFuture(createShaderObjectGroupIfNotInCache(shaders.vert));
  shaderInfoJob.fallback.m_fragmentShader =
      getAsFuture(createShaderObjectGroupIfNotInCache(shaders.frag));

  // Launch a job to compile the shaders with linking
  shaderInfoJob.optimized =
      getAsFuture(createShaderObjectGroupIfNotInCache(shaders));
  return shaderInfoJob;
}
void recordRenderingCommands(vk::CommandBuffer cmdBuf) {
for (const auto& drawDataIt : drawData) {
  if (drawDataIt.shaders.optimized.ready()) {
    cmdBuf.bindShadersEXT(stages, drawDataIt.shaders.optimized.getReady());
  } else {
    // Fallback shaders are expected to be ready, if not, we wait for them.
    cmdBuf.bindShadersEXT(stages, drawDataIt.shaders.fallback.getWait());
  }
 // ...
```

WHEN TO LINK SHADER OBJECTS

- For most projects, it is fine to always link shaders
- If you have a lot of shaders:
  - First compile shaders separately
    - Individual shaders are more likely to be reused.
  - Then link shaders together for better GPU performance.

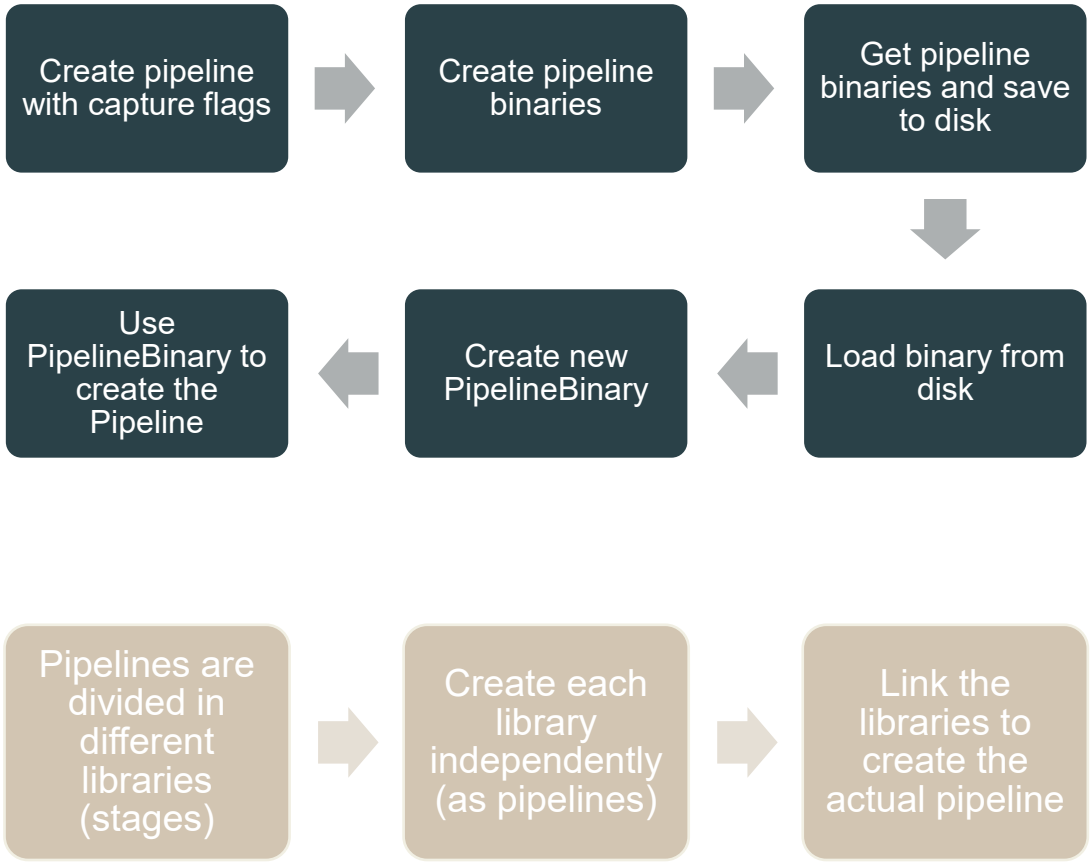
Pipeline Management (Shader Objects)

```
const uint64_t shaderBinaryKey = calculateShaderBinaryKey(
    soDeviceProperties.shaderBinaryUUID,
    soDeviceProperties.shaderBinaryVersion,
    spirvShaders,
    descriptorHeapMapping
    /* ... Add all shader creation state affecting binary compatibility */);

const bool hasShaderBinary = shaderCacheLoad(shaderBinaryKey, shaderBinaryData);

vk::ShaderCreateInfoEXT shaderCreateInfo{
  .codeType = hasShaderBinary ? vk::ShaderCodeTypeEXT::eBinary
                              : vk::ShaderCodeTypeEXT::eSpirV,
  .codeSize = hasShaderBinary ? shaderBinaryData.size() : spirvData.size(),
  .pCode = hasShaderBinary ? shaderBinaryData.data() : spirvData.data(),
  // ...
};

auto shaders = m_logicalDevice.createShadersEXTUnique(createInfos).value;
// Need to fall back to SPIR-V if binary not in cache or createShadersEXT fails

if (!hasShaderBinary) {
    const std::vector<std::uint8_t> shaderBinary =
          m_logicalDevice.getShaderBinaryDataEXT(*shaders[0]);
   shaderCacheStore(shaderBinaryKey, shaderBinary);
}
```

BINARY SHADER OBJECTS

- Shader objects can expose compiled binaries for application-managed caching.
  - `VkPipelineCache` can prove insufficient for complex applications
- Shader cache key can be complex
  - Device properties:
    - `VkPhysicalDeviceShaderObjectPropertiesEXT::shaderBinaryUUID`
    - `VkPhysicalDeviceShaderObjectPropertiesEXT::shaderBinaryVersion`
  - Descriptor mappings, flags, layers, linking, etc.
- `getShaderBinaryDataEXT` allows to get the compiled shader binary data
- `ShaderCreateInfoEXT::codeType` = `ShaderCodeTypeEXT::eBinary`
- Some platforms might have an internal or cloud cache

Pipeline Management



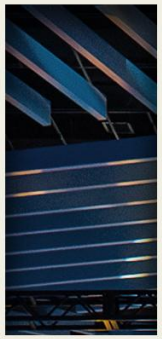
SHADER OBJECTS ALTERNATIVES

- Shader objects are the recommended future looking option
  - Small changes might still be necessary (working on a KHR)
  - Might still lack support on some platforms
- Alternatives to improve `VkPipeline` compilation
  - **Extended dynamic state**: Compatible with `VkPipeline`
  - `KHR_pipeline_binary`: Control pipeline cache directly
    - Access binaries like `getShaderBinaryDataEXT`
  - `KHR_pipeline_library` / `EXT_graphics_pipeline_library`
    - Allows creating pipeline libraries for different stages of the pipeline
    - Allows compiling stages of a Vulkan pipeline independently
    - Shader Objects offers an easier more future looking API

Pipeline Management

SHADER OBJECTS VS VKPIPELINES

VkPipelines	Shader Objects
Create a pipeline with all state upfront	Create shader objects with SPIR-V and set state dynamically
Use `VkPipelineCache` to cache pipelines	Use `getShaderBinaryDataEXT` to get compiled shader binary
Set most state at pipeline creation	Set state dynamically with new methods
Boilerplate code to create pipelines, shader modules, etc.	More flexible, dynamic and simple
Universal support	Limited support
Easy for driver to optimize	Limited optimization opportunities, Specially if shaders are not linked



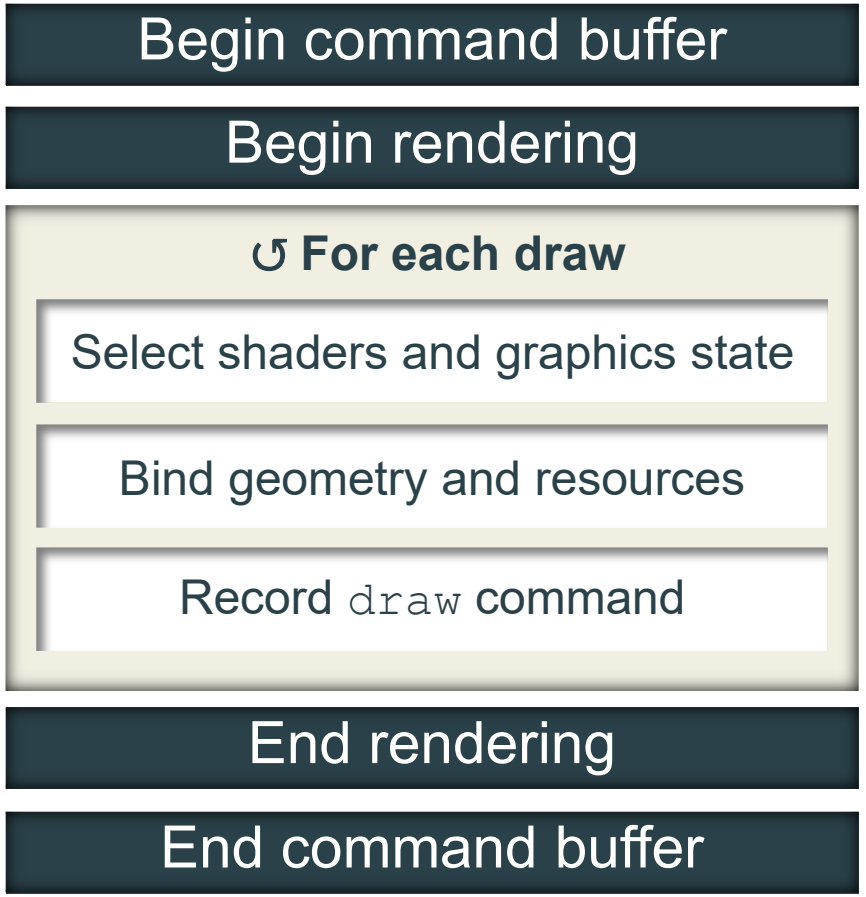


SIGGRAPH 2026
Los Angeles 19–23 JUL

Draws and geometry

Recording draws

- Draws consume the current state and bound resources
- Many draws can share one rendering scope



Geometry data: vertices and indices

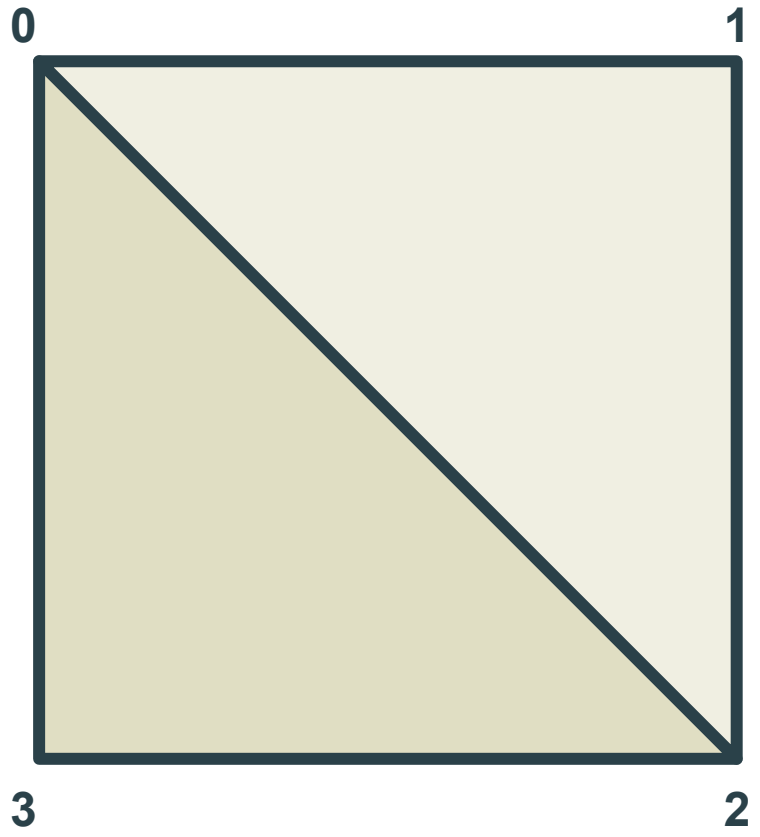
- Vertex buffers store attributes for each vertex
- Index buffers select and reuse complete vertex records
- Topology defines how vertices form primitives

Vertex buffer

0	position | UV | normal | tangent
1	position | UV | normal | tangent
2	position | UV | normal | tangent
3	position | UV | normal | tangent

Index buffer

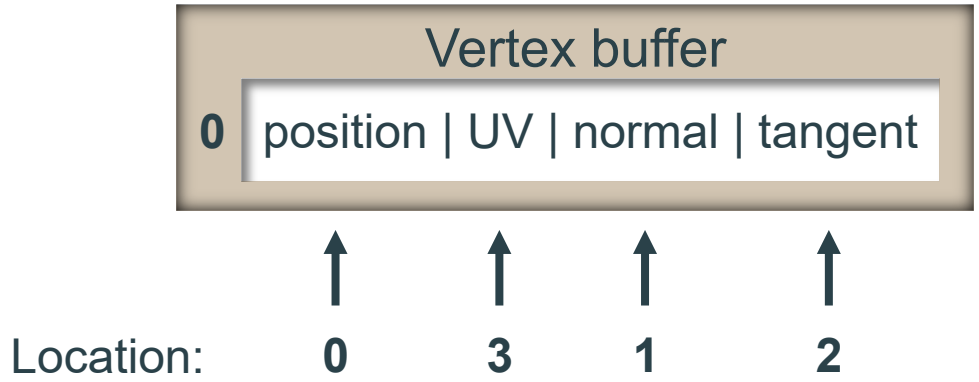
0, 1, 2
2, 3, 0



Geometry data: shader inputs

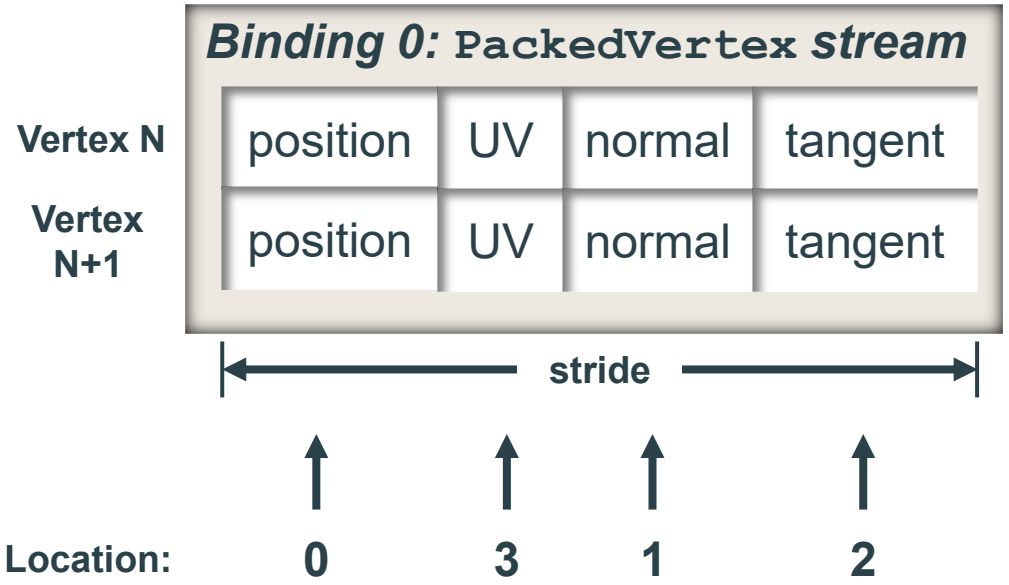
```
struct VertexInput {
    [[vk::location(0)]]
    float3 position

    [[vk::location(1)]]
    float3 normal;

    [[vk::location(2)]]
    float4 tangent;

    [[vk::location(3)]]
    float2 uv;
};
```

- Vulkan matches attributes by location
- The application maps buffer data to those locations



Geometry data: describing the vertex layout

```
struct PackedVertex {
  glm::vec3 position;
  glm::vec2 uv;
  glm::vec3 normal;
  glm::vec4 tangent;
};
```




```
const vk::VertexInputBindingDescription2EXT
  bindingDescription{
    .binding   = 0,
    .stride    = sizeof(PackedVertex),
    .inputRate = vk::VertexInputRate::eVertex,
    .divisor   = 1
};
```

```
const vk::VertexInputAttributeDescription2EXT
  positionAttribute{
    .location = 0,
    .binding  = 0,
    .format   = vk::Format::eR32G32B32Sfloat,
    .offset   = offsetof(PackedVertex, position)
};
```

One attribute description per shader input

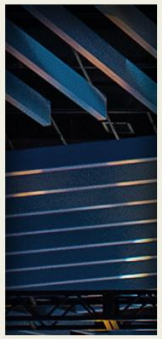
Geometry data: binding the buffers

Set the vertex layout

```
const std::array bindingDescriptions{
  bindingDescription
};


const std::array attributeDescriptions{
  positionAttribute,
  normalAttribute,
  tangentAttribute,
  uvAttribute
};

commandBuffer.setVertexInputEXT(
  bindingDescriptions,
  attributeDescriptions
);
```

Bind the current mesh

```
// Attach the current mesh’s vertex buffer
// to binding 0
commandBuffer.bindVertexBuffers2(
  0, // first binding
  vertexBuffers,
  vertexOffsets,
  vertexSizes,
  vertexStrides); // sizeof(PackedVertex)

// Bind the mesh's indices separately
commandBuffer.bindIndexBuffer2(
  meshIndexBuffer,
  indexOffset,
  indexBufferSize,
  vk::IndexType::eUint32);
```

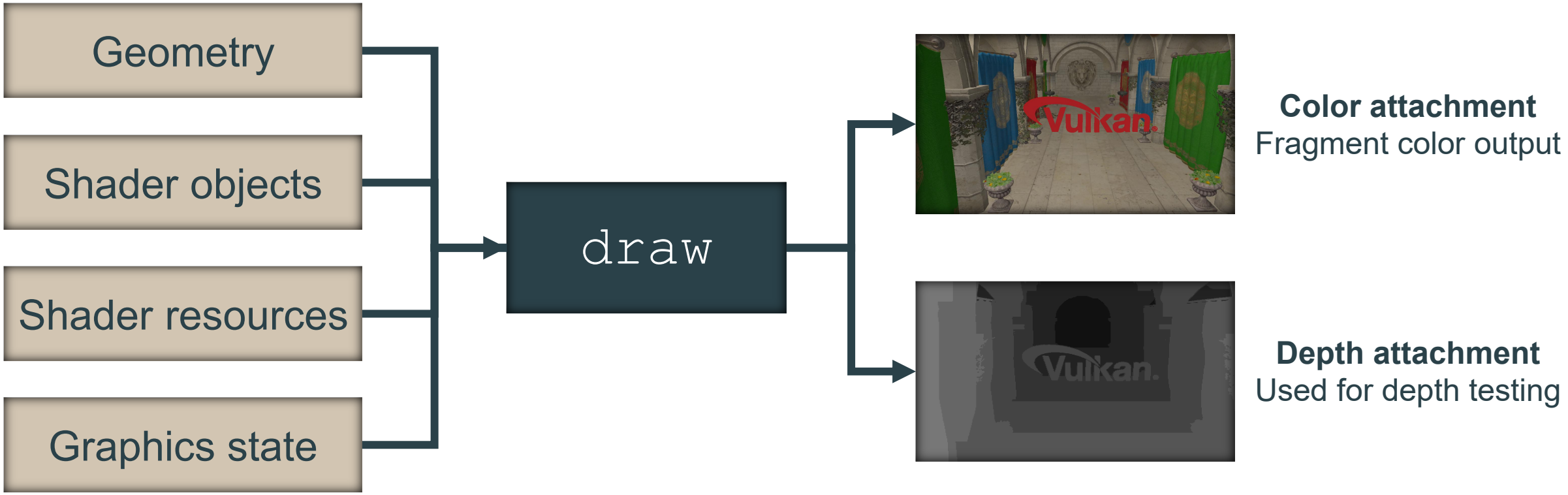




SIGGRAPH 2026
Los Angeles 19–23 JUL

Rendering scopes and attachments

Rendering: attachments



- An attachment is an image view used during rendering for color, depth, or stencil data

Dynamic rendering

Color image view

Depth image view


```
commandBuffer.beginRendering(renderingInfo)
```

Active rendering scope

Draw 0 | Draw 1 | Draw 2

State, resources and geometry may change between draws

```
commandBuffer.endRendering()
```

Begin the rendering scope

```
const vk::RenderingAttachmentInfo colorAttachment{
  .imageView   = swapchainImageView,
  .imageLayout = colorAttachmentLayout,
  .loadOp      = vk::AttachmentLoadOp::eClear,
  .storeOp     = vk::AttachmentStoreOp::eStore,
  .clearValue  = colorClearValue
};
```

```
const vk::RenderingAttachmentInfo depthAttachment{
  .imageView   = depthImageView,
  .imageLayout = depthAttachmentLayout,
  .loadOp      = vk::AttachmentLoadOp::eClear,
  .storeOp     = vk::AttachmentStoreOp::eDontCare,
  .clearValue  = depthClearValue
};
```

```
const vk::RenderingInfo renderingInfo{
  .renderArea           = {{0, 0}, swapchainExtent},
  .layerCount           = 1,
  .colorAttachmentCount = 1,
  .pColorAttachments    = &colorAttachment,
  .pDepthAttachment     = &depthAttachment
};

commandBuffer.beginRendering(renderingInfo);
```

Dynamic rendering and older render pass objects

Dynamic rendering

Before recording

Images / Image views

While recording

RenderingInfo

```
beginRendering()
draws
endRendering()
```

Render pass object API

Before recording

Attachment image views:

Attachment descriptions

Subpasses

Dependencies

`VkRenderPass`

`VkFramebuffer #2`

`VkFramebuffer #1`

`VkFramebuffer #0`

While recording

```
beginRenderPass()
draws / nextSubpass()
endRenderPass()
```

Same rendering scope, without render pass or framebuffer objects

Subpass-like attachment-local reads are also supported with dynamic rendering

Rendering: putting the draw sequence together

Rendering scope

```
commandBuffer.beginRendering(renderingInfo);
```

State shared by these draws

```
commandBuffer.setVertexInputEXT(
  bindingDescriptions, attributeDescriptions);
```

For each draw:

- **Select shaders + state**
- **Provide shader data**
- **Bind geometry**
- **Record draw**

```
for (const SceneDraw& draw : draws) {
  commandBuffer.bindShadersEXT(...);

  commandBuffer.pushDataEXT(...);

  commandBuffer.bindVertexBuffers2(...);
  commandBuffer.bindIndexBuffer2(...);

  commandBuffer.drawIndexed(...);
}
```

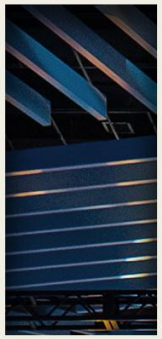
```
commandBuffer.endRendering();
```

10-minute break

Course materials
tinyurl.com/24fsppy9

How to write a Vulkan application in 2026

Back in: **10:00**

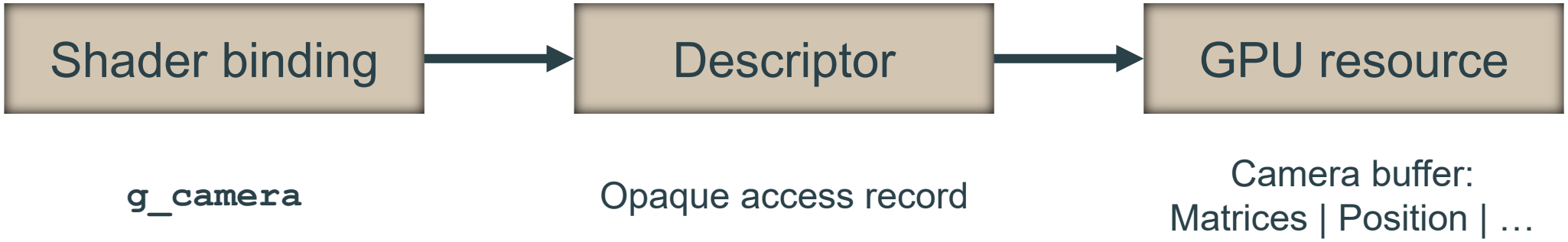




SIGGRAPH 2026
Los Angeles 19–23 JUL

Descriptor management

Descriptors connect shaders to resources



- Descriptors describe access, resource data remains in its buffer or image

Classic descriptor sets and bindings

Example:

Set 0: Global

- Binding 0: Camera
- Binding 1: Lights

Set 1: Material

- Binding 0: Albedo texture
- Binding 1: Normal texture

Set 2: Draw

- Binding 0: Per-draw data

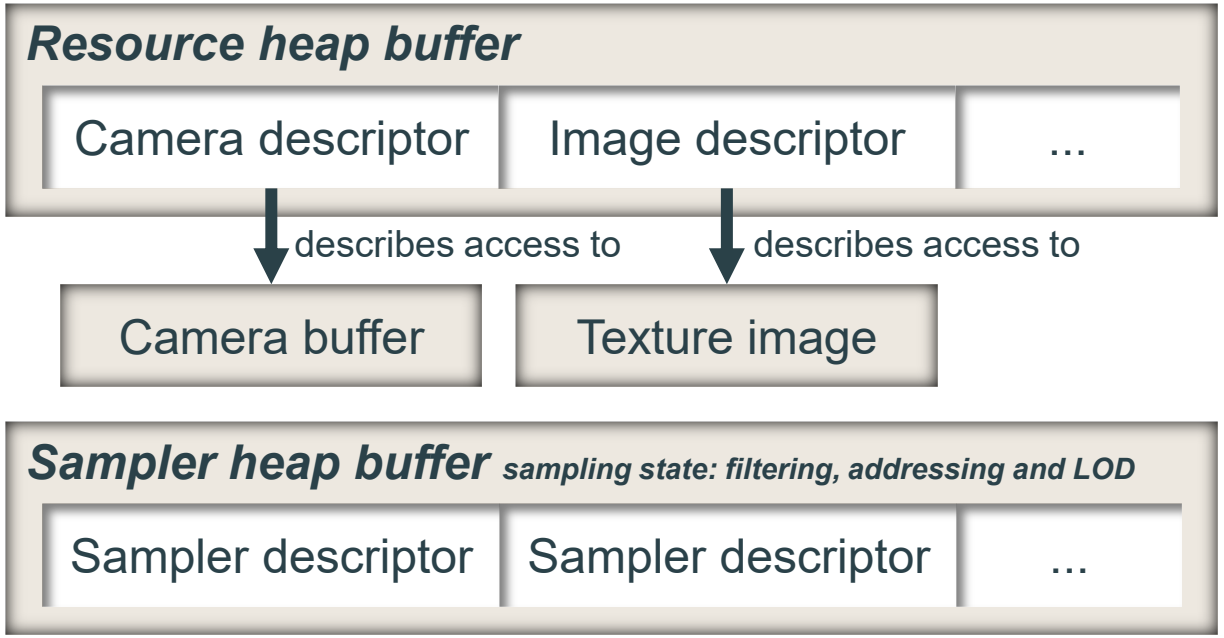
Slang: `vk::binding(binding, set)`

```
[[vk::binding(0, 0)]]
ConstantBuffer<CameraData> g_camera;
```

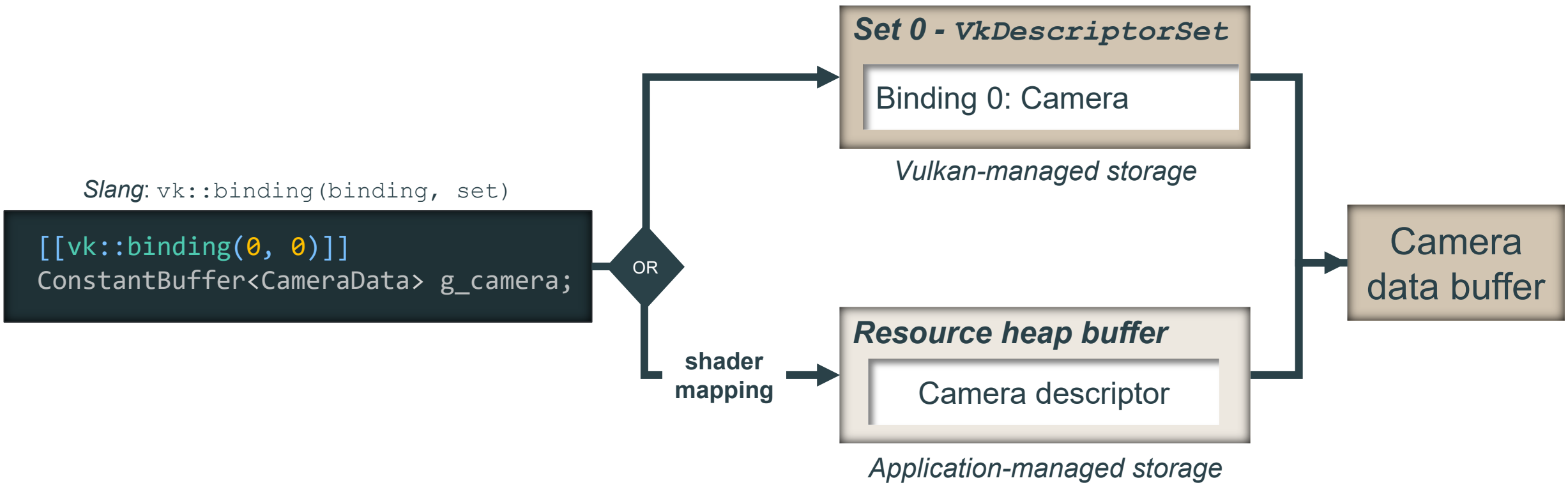
- A binding identifies a resource slot
- A descriptor set groups bindings that are updated and bound together
- Descriptor sets remain valid and widely used

Descriptor heaps

- A heap is a GPU-addressable buffer range containing descriptor bytes
- The application places descriptors at explicit byte offsets
- Resource data remains in separate buffers and images



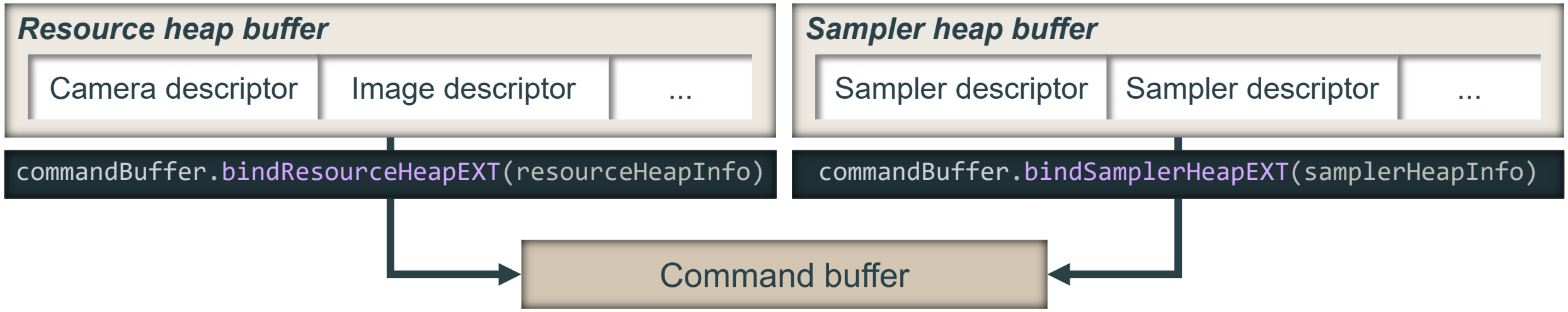
Descriptor heaps: same binding, different storage



- Fewer descriptor objects and direct control over placement and reuse
- Application manages allocation, lifetime and synchronization

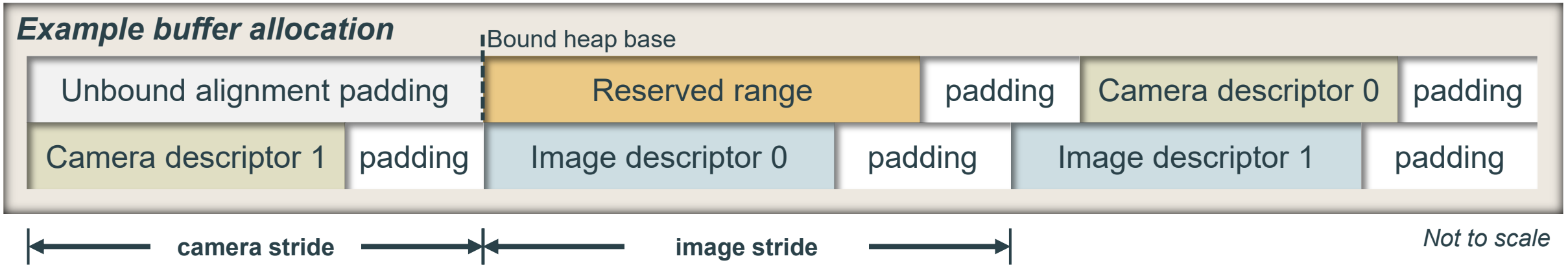
Create and bind the descriptor heaps

```
const vk::BufferUsageFlags heapUsage{
  vk::BufferUsageFlagBits::eDescriptorHeapEXT | vk::BufferUsageFlagBits::eShaderDeviceAddress };
// Create resource and sampler heap buffers using heapUsage
```



- Heap buffers use descriptor-heap and shader-device-address usage
- Record one resource-heap binding and one sampler-heap binding
- The same bound heaps can serve many draws

Anatomy of a descriptor heap



- Descriptor offsets are measured from the bound heap base

Descriptor heaps: query requirements and allocate slots

Query device requirements

```
vk::PhysicalDeviceDescriptorHeapPropertiesEXT
  heapProps{};
vk::PhysicalDeviceProperties2 properties{
  .pNext = &heapProps};
physicalDevice.getProperties2(&properties);

const auto reservedRangeSize =
  heapProps.minResourceHeapReservedRange;

const auto bufferAlignment =
  heapProps.bufferDescriptorAlignment;

const auto cameraDescriptorSize =
  physicalDevice.getDescriptorSizeEXT(
      vk::DescriptorType::eUniformBuffer);
```

Allocate descriptor slots

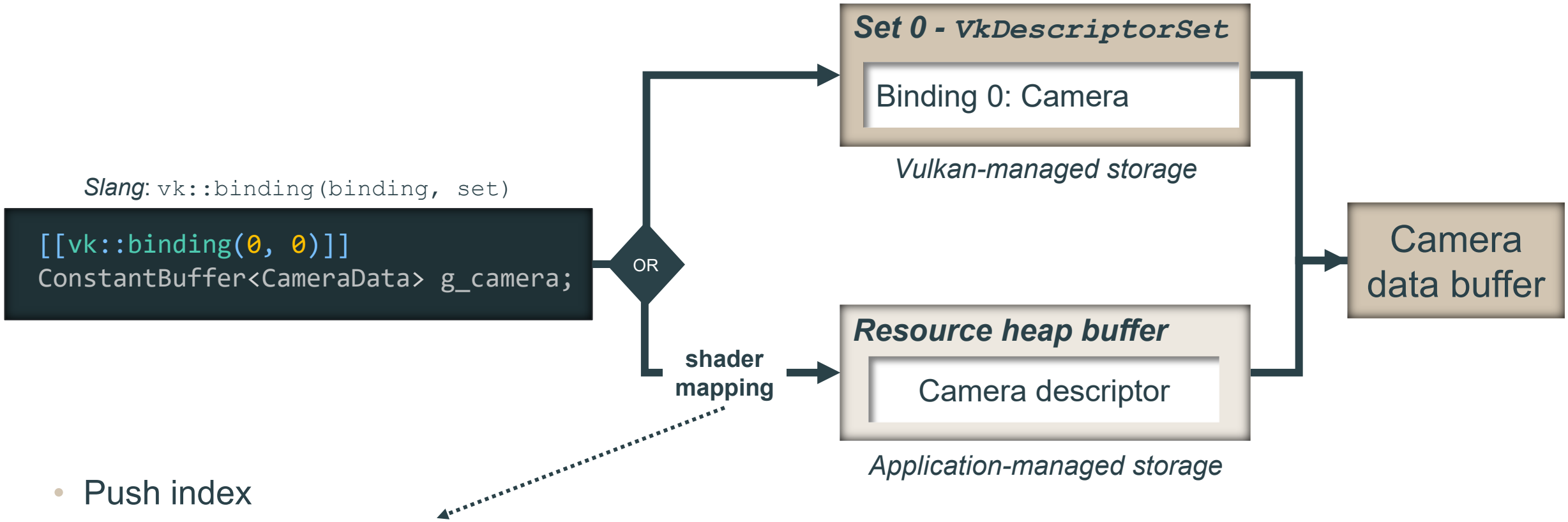
```
cursor = reservedRangeSize;

auto allocateSlot = [&](VkDeviceSize size,
                        VkDeviceSize alignment){
  cursor = alignUp(cursor, alignment);
  const auto offset = cursor;
  cursor += size;
  return offset;
};

cameraOffset =
  allocateSlot(cameraDescriptorSize,
               bufferAlignment);
```

Descriptor heaps: write descriptor bytes

1. Describe the resource
2. Select the destination slot
3. Write the opaque descriptor bytes

```
vk::DeviceAddressRangeEXT cameraRange{
  .address = cameraBuffer.addressGPU,
  .size    = sizeof(CameraData)
};

vk::ResourceDescriptorInfoEXT descriptorInfo{
  .type = vk::DescriptorType::eUniformBuffer
};
descriptorInfo.data.pAddressRange = &cameraRange;

// Slot allocated on the previous slide
vk::HostAddressRangeEXT destination{
  .address = mappedHeapBase + cameraOffset,
  .size    = cameraDescriptorSize
};

device.writeResourceDescriptorsEXT(
  1, &descriptorInfo, &destination);
```

Descriptor heap mapping



- Push index
- Constant offset

Descriptor heap mapping: push index

Created with the shader object

Shader declaration

```
[[vk::binding(0, 0)]]
ConstantBuffer<CameraData> g_camera;
```

Binding mapping

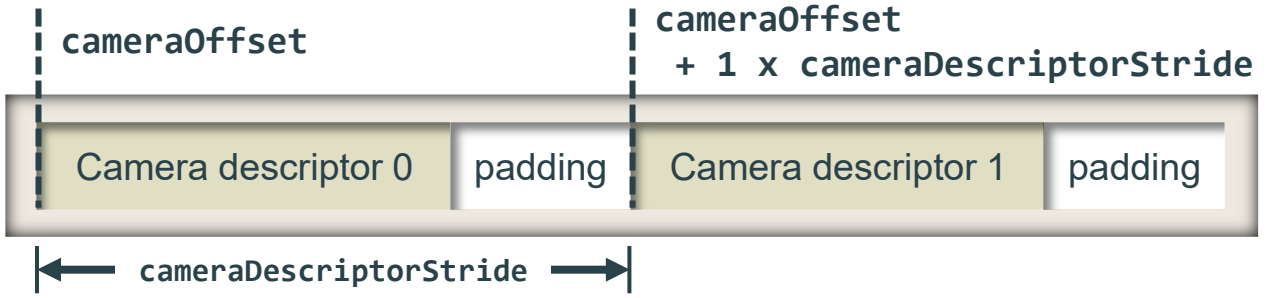
```
vk::DescriptorSetAndBindingMappingEXT
 cameraMapping{
  .descriptorSet = 0,
  .firstBinding  = 0,
  .source        = eHeapWithPushIndex,
  .sourceData    = cameraSourceData};
```

Heap-selection rule: base + index x stride

```
vk::DescriptorMappingSourceDataEXT
 cameraSourceData{
  vk::DescriptorMappingSourcePushIndexEXT{
   .heapOffset = cameraOffset,
   .pushOffset = offsetof(PushData, cameraIndex),
   .heapIndexStride = cameraDescriptorStride}};
```

Push data layout and per-draw value

```
struct PushData {
 std::uint32_t cameraIndex; // byte offset 0
};

PushData pushData{
  .cameraIndex = frameIndex; // example: 1
};

commandBuffer.pushDataEXT(...);
```



Descriptor heap mapping: constant offset

Created with the shader object

Shader declaration

```
[[vk::binding(2, 0)]]
 ConstantBuffer<LightData> g_pointLight;
```

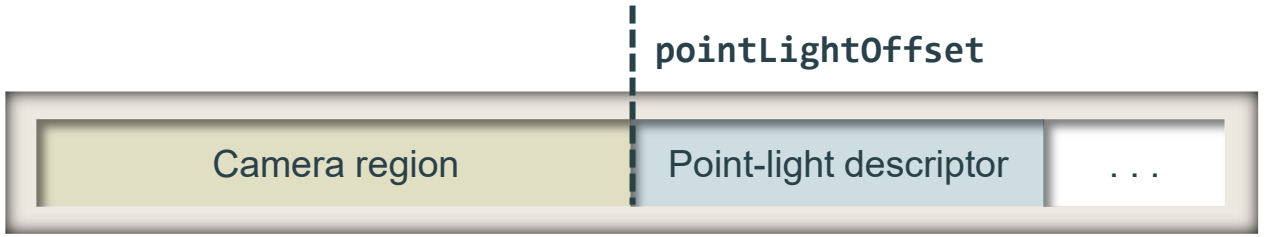
Binding mapping

Heap-selection rule: base

```
vk::DescriptorSetAndBindingMappingEXT
 pointLightMapping{
  .descriptorSet = 0,
  .firstBinding  = 2,
  .source     = eHeapWithConstantOffset,
  .sourceData = pointLightSourceData};
```

```
vk::DescriptorMappingSourceDataEXT
 pointLightSourceData{
  vk::DescriptorMappingSourceConstantOffsetEXT {
   .heapOffset = pointLightOffset
}};
```

Fixed descriptor selection



Descriptor heap mappings and alternatives

MAPPING TOOLBOX

- Heap selection
  - ✓ Pushed index
  - ✓ Constant offset
  - Indirect index or index array
- Inline resources
  - Heap data
  - Push data
  - Pushed or indirect addresses
- Shader records
  - Ray-tracing mappings
- Direct heap access
  - Untyped pointers, no binding mapping

CHOOSE ACCORDING TO SUPPORT

- Descriptor heaps
  - Full explicit memory and mapping control
- Descriptor indexing
  - Bindless-style arrays while retaining descriptor sets
- Descriptor buffer
  - Older explicit-memory model, relevant on existing implementations
- Classic descriptor sets
  - Portable and still entirely valid

From descriptor objects to descriptor memory

CLASSIC DESCRIPTOR SETS

- Predeclare layouts and grouping
- Create pools and allocate sets
- Update and bind set objects
- Implementation manages descriptor storage

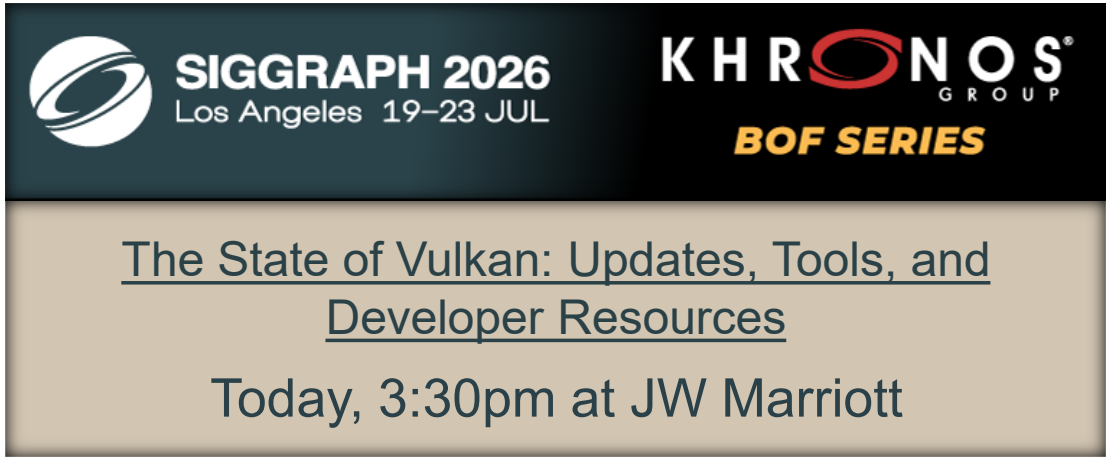
DESCRIPTOR HEAPS

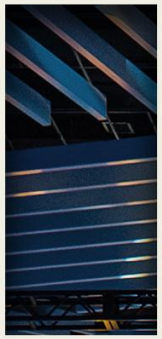
- Allocate and align heap memory
- Write and copy descriptor bytes
- Map shader bindings to heap locations
- Application manages descriptor storage

More explicit setup, but fewer descriptor objects and
greater freedom to organize, stream and select resources

Descriptor heaps: further reading

- Vulkan Docs: Descriptor Heap
- Vulkan Spec Proposal for VK_EXT_descriptor_heap
- Nvidia: Streamlining Resource Binding with End-to-End Support for Vulkan Descriptor Heaps
- Sascha Willems: New Vulkan samples for using descriptor heaps (includes a sample using untyped pointers)
- XDC 2025: Descriptors are hard







SIGGRAPH 2026
Los Angeles 19–23 JUL

Synchronization

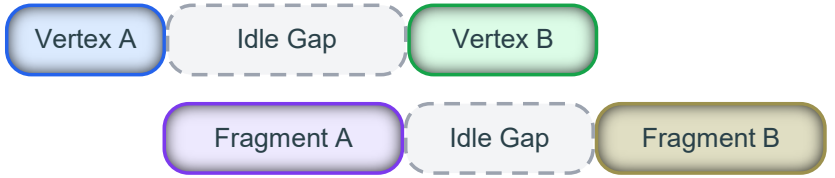
What synchronization establishes

1. Has the GPU finished work the CPU wants to reuse?
2. Which GPU operation must happen before another?
3. When are memory writes visible to later accesses?

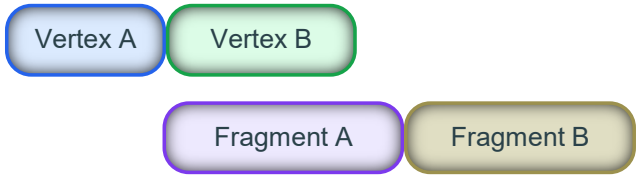
Synchronization must be correct and precise

- Too little → incorrect or undefined results
- Too much → correct, but independent work waits
- Precise → correct while preserving overlap

Excessive synchronization



Precise synchronization



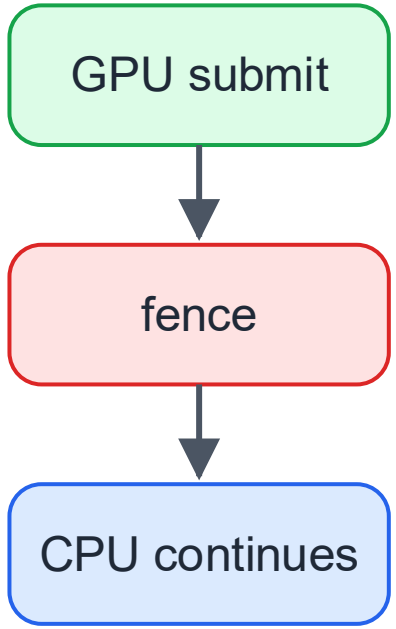
Synchronization primitives

Need	Primitive	Boundary crossed
CPU needs GPU completion	Fence	Queue submission → CPU
Later submission must wait	Semaphore	Submission or queue boundary
Later commands depend on earlier accesses	Pipeline barrier	Recorded queue command stream

Also available: events split an in-command-stream dependency into separate set and wait points

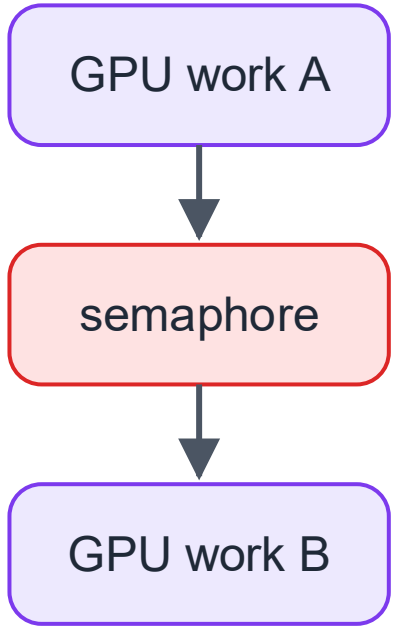
Fences

- CPU waits for finished GPU work
- Useful for safe reuse of per-frame resources
- Also useful before destroying GPU-used resources
- A fence does not order later GPU work by itself



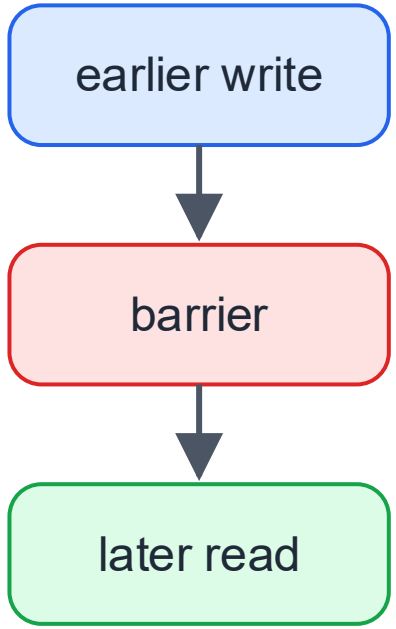
Binary semaphores

- Later submission waits for an earlier GPU operation
- Useful for cross-submit ordering
- Useful for acquire then render and render then present
- Binary semaphores are GPU side, not CPU waits



Pipeline barriers

- Later GPU commands depend on earlier accesses
- Recorded in the command buffer
- For images, the barrier may also change the image layout
- Source and destination stage/access masks describe the dependency

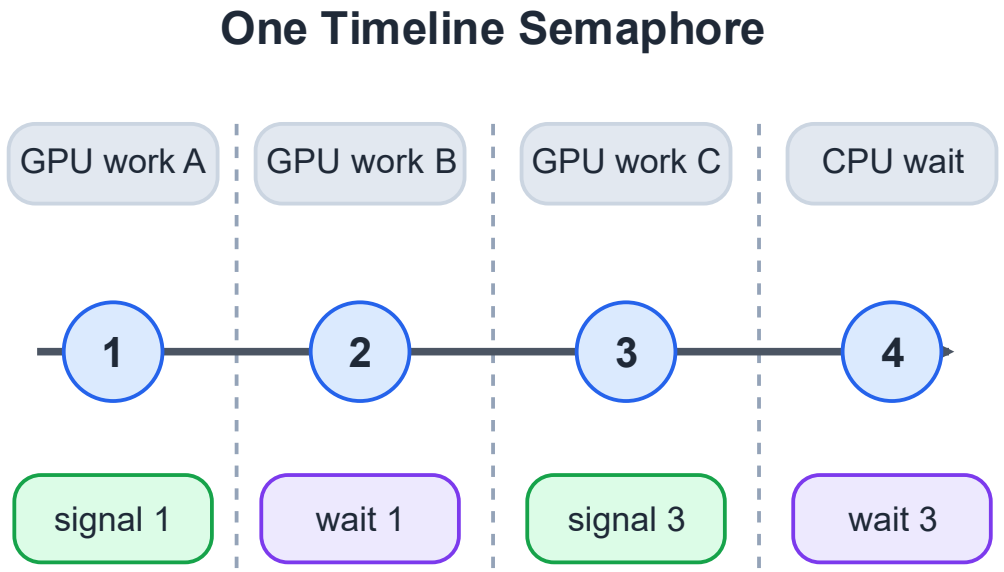


Reducing synchronization bookkeeping

TIMELINE SEMAPHORES	SYNCHRONIZATION2	UNIFIED IMAGE LAYOUTS
Many ordered milestones in one semaphore	Clearer dependency and submission structures	Fewer layout transitions

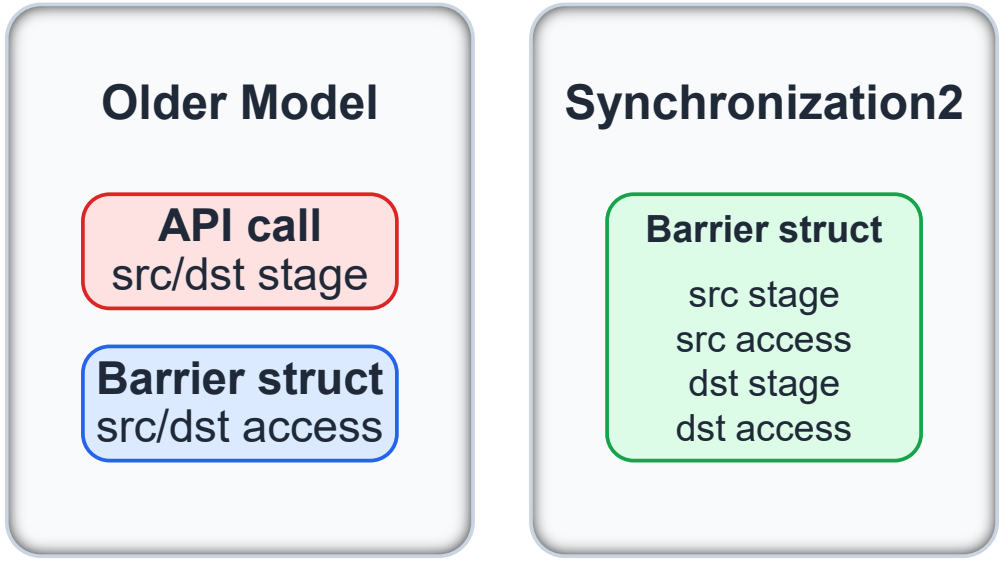
Timeline semaphores

- One semaphore represents many increasing milestone values
- CPU and GPU operations can wait for specific values
- Useful for long dependency chains and queue coordination
- Swapchain acquire and present still use binary semaphores



Synchronization2

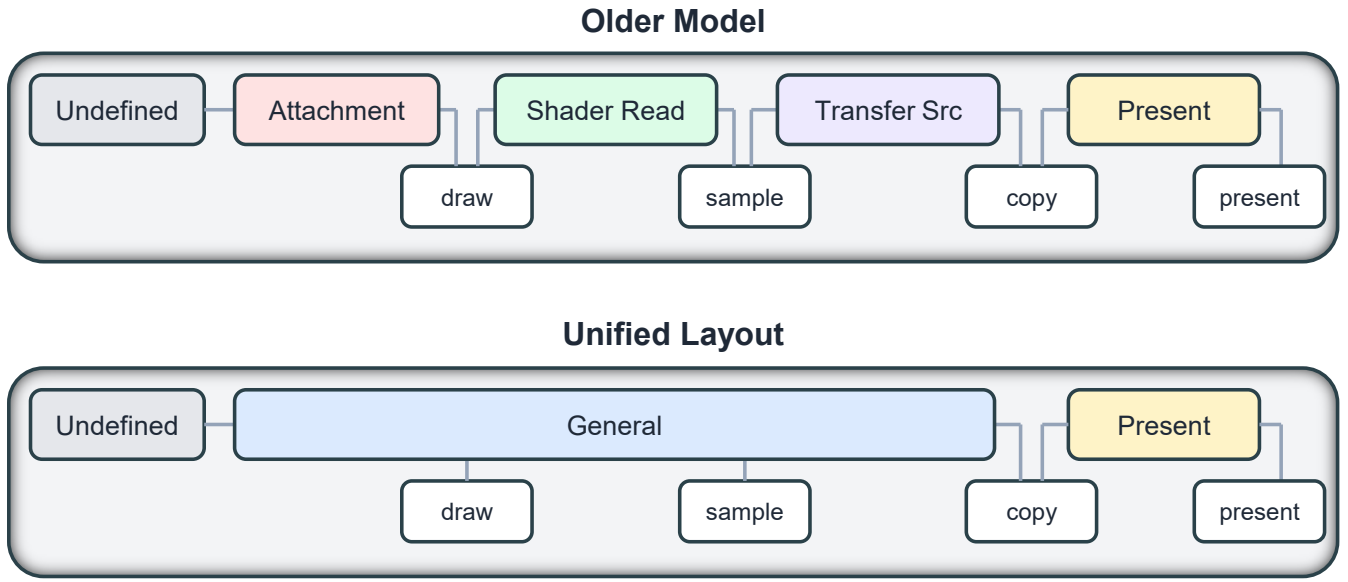
- `VK_KHR_synchronization2`, core in Vulkan 1.3
- Older barriers split details across calls and structs
- Synchronization2 keeps the full barrier dependency in one struct
- `submit2` waits and signals also use clearer per-entry structs
- Same semantics, cleaner expression



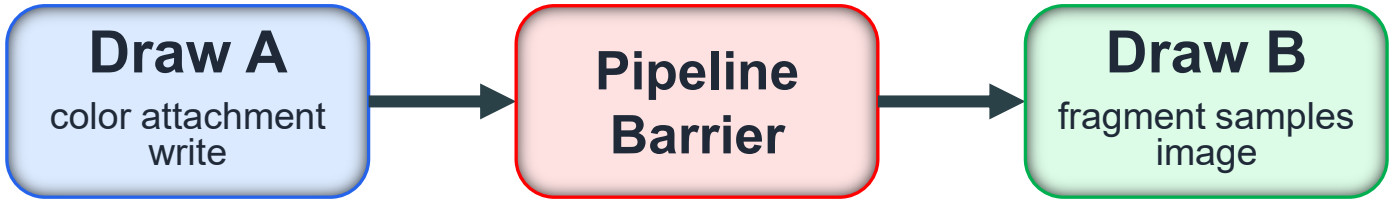
Unified image layouts

- With the feature enabled, `GENERAL` is efficient for most image usages
- Fewer layout transitions need to be tracked
- `UNDEFINED` and `PRESENT_SRC_KHR` remain distinct
- Memory-dependency requirements are unchanged

Same image, simpler layout story



Example: draw output to later read




```
const vk::ImageMemoryBarrier2 barrier{
  .srcStageMask  = vk::PipelineStageFlagBits2::eColorAttachmentOutput,
  .srcAccessMask = vk::AccessFlagBits2::eColorAttachmentWrite,

  .dstStageMask  = vk::PipelineStageFlagBits2::eFragmentShader,
  .dstAccessMask = vk::AccessFlagBits2::eShaderSampledRead,

  .oldLayout = vk::ImageLayout::eGeneral,
  .newLayout = vk::ImageLayout::eGeneral,
  // image and range omitted
};

// dependencyInfo contains barrier
commandBuffer.pipelineBarrier2(dependencyInfo);
```

Broad scopes can block unrelated work


```
const vk::ImageMemoryBarrier2 barrier{
  .srcStageMask  = vk::PipelineStageFlagBits2::eAllCommands,

  .dstStageMask  = vk::PipelineStageFlagBits2::eAllCommands,
  // ...
};
```

- A broad dependency may be correct, but it prevents unrelated work from progressing

Precise scopes preserve pipeline overlap

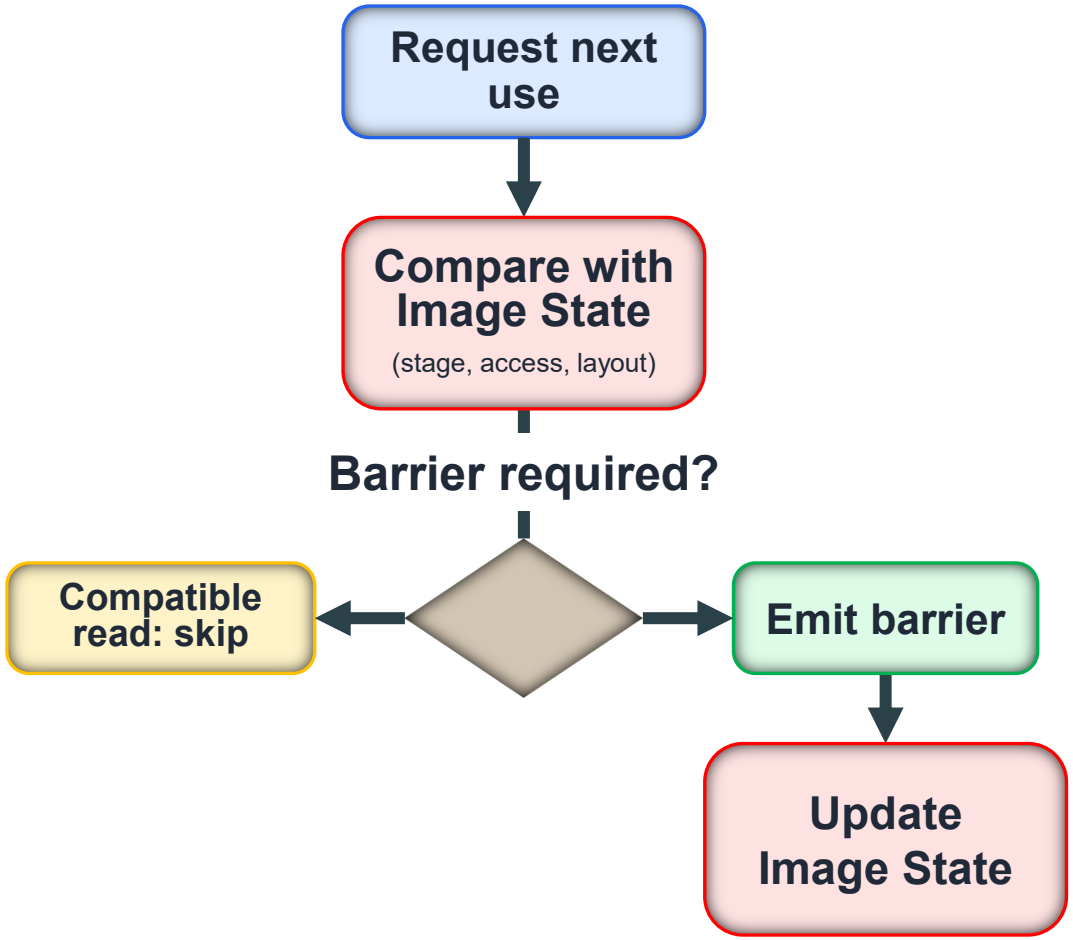

```
const vk::ImageMemoryBarrier2 barrier{
  .srcStageMask  = vk::PipelineStageFlagBits2::eColorAttachmentOutput,

  .dstStageMask  = vk::PipelineStageFlagBits2::eFragmentShader,
  // ...
};
```

- Correctness first, then use the narrowest scopes that cover the real hazard

Why manual synchronization does not scale

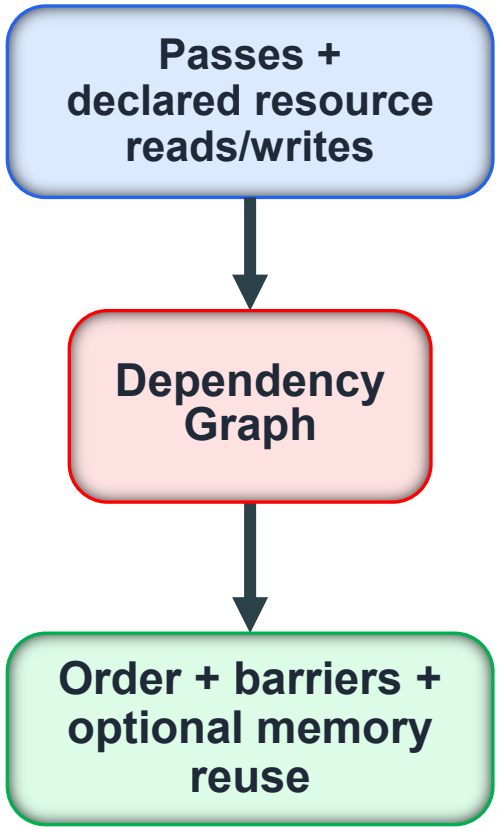
- Too much state is easy to lose track of by hand
- It is easy to insert too much or too little synchronization
- Barrier placement affects both correctness and performance
- Larger projects usually build a higher-level system

Simple resource-usage tracking



- Track the last stage, access and layout for each image
- Compatible read-after-read accesses may skip a barrier
- Any hazard involving a write requires synchronization

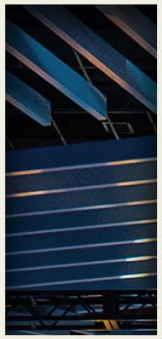
Render graphs generalize resource tracking



- Passes declare resource reads and writes centrally
- The graph derives execution order and synchronization
- Known resource lifetimes may also enable temporary-memory reuse

Synchronization: further reading

- Vulkanised 2024: Using Vulkan Synchronization Validation Effectively
- Vulkanised 2026: Solving All Synchronization Problems with Timeline Semaphores
- Arm: Vulkan synchronization on Mali GPUs (vertex fragment overlap)
- AMD's: Leveraging Asynchronous Queues for Concurrent Execution (async compute)
- Khronos' Vulkan Tutorial: Using async compute to saturate GPU (async compute)
- Resource-usage tracking
  - Vulkanised 2024: Vulkan synchronization for WebGPU
  - Vulkanised 2024: Vulkan Synchronization Made Easy (without rendergraphs)
- Render graphs
  - Khronos Vulkan Tutorial: render graphs and synchronization
  - REC 2023: Task Graph Renderer at Activision

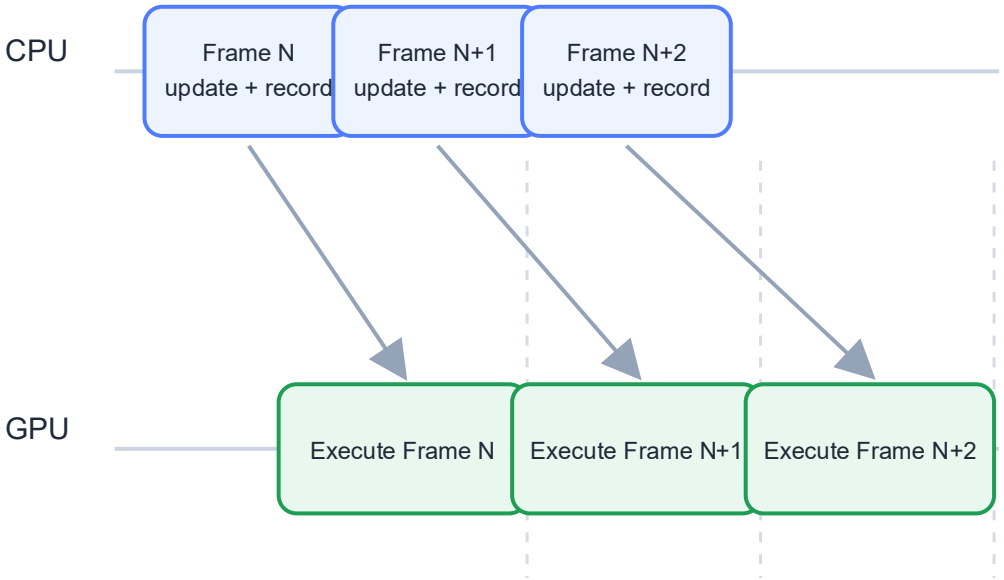




SIGGRAPH 2026
Los Angeles 19–23 JUL

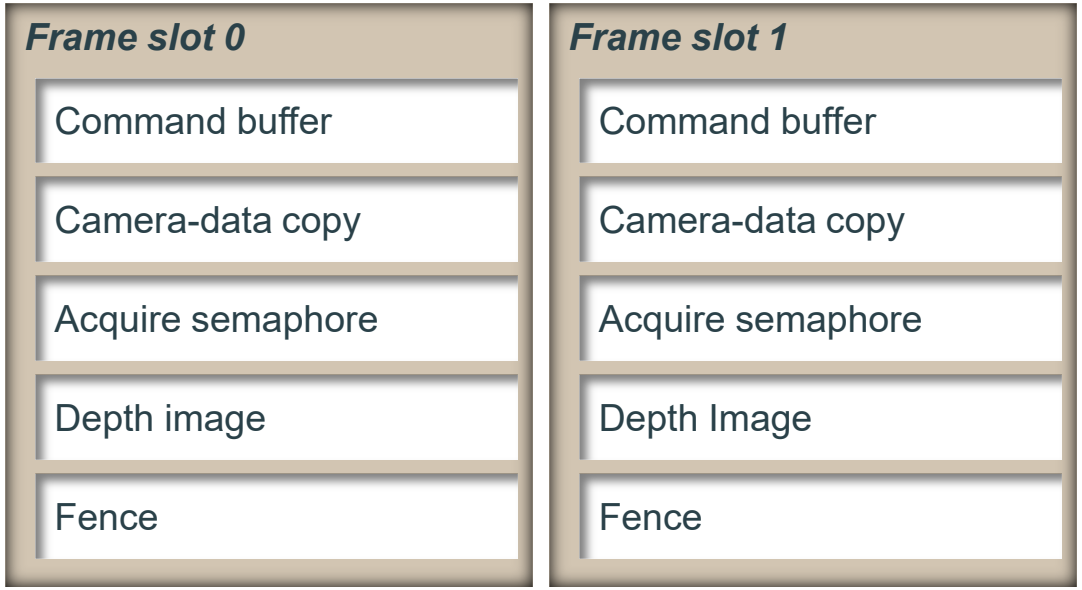
CPU and GPU communication

Frames in flight let CPU and GPU overlap



- CPU prepares frame N+1 while the GPU executes frame N
- A small ring of frame slots avoids waiting after every frame
- More frames can increase memory use and latency
- Two frames in flight is a common starting point

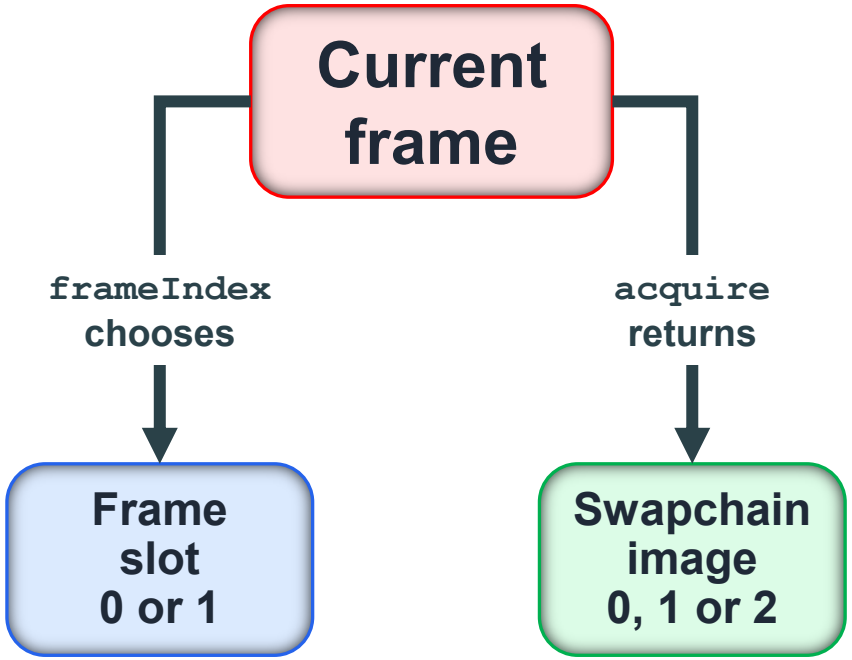
Frame slots are reusable submission bundles




```
frameIndex = (frameIndex + 1) % framesInFlight.size();
```

- Each slot owns resources that must remain valid for one in-flight submission
- Wait for the slot's fence before reusing those resources
- Duplicate only resources that are independently updated or concurrently in use

Frame slots and swapchain images are independent



- `frameIndex` selects a reusable frame-in-flight slot
- `swapchainImageIndex` is returned by image acquisition
- The two counts and indices are independent
- A frame slot may render to different swapchain images over time

Frame flow

1. Wait for and reuse the frame slot
2. Acquire a swapchain image
3. Update data and record commands
4. Submit:
    - Wait semaphores
    - Command buffer
    - Signal semaphores
    - Fence
5. Present

Acquire and reuse safely

```
auto& frame = m_framesInFlight[frameIndex];

// Wait for the frame slot to become reusable
m_logicalDevice.waitForFences(frame.m_inFlightFence,
  vk::True, UINT64_MAX);

// Acquire the next swapchain image
const auto acquireResult = m_logicalDevice.acquireNextImageKHR(
  vk::SwapchainKHR{m_vkbData.m_swapchain.swapchain},
  UINT64_MAX, *frame.m_imageAvailableSemaphore, nullptr);

const uint32_t swapchainImageIndex = acquireResult.value;

frame.m_commandBuffer.reset();
```

- `frameIndex` selects the frame slot
- Wait for its fence before reusing slot resources
- Acquisition returns a separate `swapchainImageIndex`
- Reset the command buffer before recording

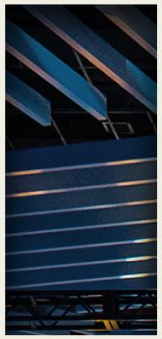
Submit and present

```
// Wait until the acquired image is available
const std::array waitInfos = { vk::SemaphoreSubmitInfo {
  .semaphore = *frame.m_imageAvailableSemaphore,
  .stageMask = vk::PipelineStageFlagBits2::eColorAttachmentOutput
}};

// Signal when rendering is done for this image
const std::array signalInfos = { vk::SemaphoreSubmitInfo{
      .semaphore = *swapchainImage.m_renderFinishedSemaphore,
      .stageMask = vk::PipelineStageFlagBits2::eAllCommands} };

m_logicalDevice.resetFences({ *frame.m_inFlightFence });

m_graphicsQueue.submit2(
  submitInfo, // waitInfos + command buffer + signalInfos
  *frame.m_inFlightFence);
m_presentQueue.presentKHR(
  presentInfo); // waits on the image's render-finished semaphore
```

- Submission waits on the frame slot's image-available semaphore
- Reset the frame fence, then submit the slot's command buffer
- Completion signals the image's render-finished semaphore and the frame fence
- Presentation waits on the image's render-finished semaphore

CPU and GPU communication: further reading

- Vulkanised 2026: Frames in Flight Demystified





SIGGRAPH 2026
Los Angeles 19–23 JUL

Debugging

Debugging And Validation

GOALS

- Provide some practical guidance on how to debug your application, and the tools you should be aware of

Debugging And Validation

VULKAN VALIDATION LAYERS

- By design, the core Vulkan API provides only minimal error checking
- The specification is filled with Valid Usage statements describing the limits on defined behavior
- Vulkan Validation Layers will warn you if you violate this
- Turn them on ASAP for development builds and leave them on
- Fix your validation errors as soon as you see them

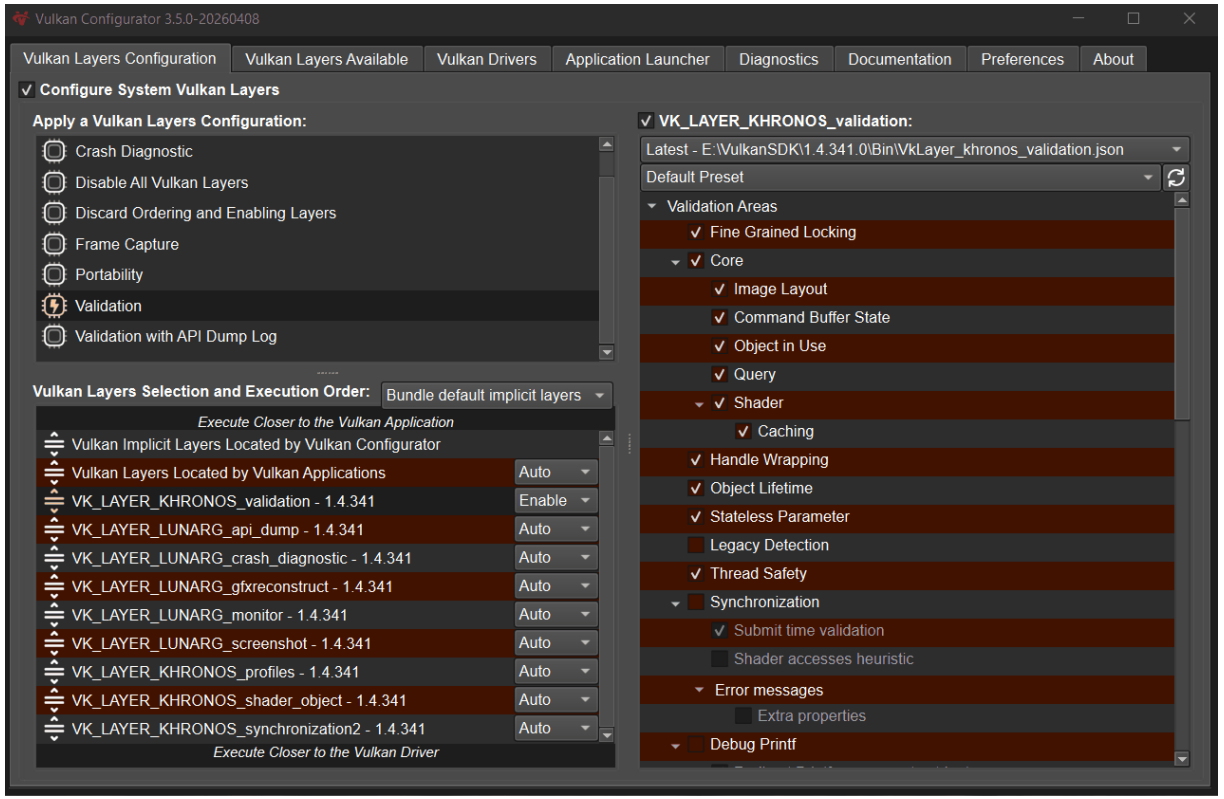
Valid Usage (Implicit)

- VUID-vkCreateImage-device-parameter
  `device` **must** be a valid VkDevice handle
- VUID-vkCreateImage-pCreateInfo-parameter
  `pCreateInfo` **must** be a valid pointer to a valid VkImageCreateInfo structure
- VUID-vkCreateImage-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreateImage-pImage-parameter
  `pImage` **must** be a valid pointer to a VkImage handle
- VUID-vkCreateImage-device-queuecount
  The device **must** have been created with at least `1` queue

Debugging And Validation

CONFIGURING VULKAN VALIDATION LAYERS

- There are a lot of tools and options in the validation layers
  - Some are disabled by default, but valuable for specific use cases (GPU-AV, sync-val)
- Older resources may recommend configuring via environment variables or programmatically
- We recommend using vkconfig from the Vulkan SDK instead

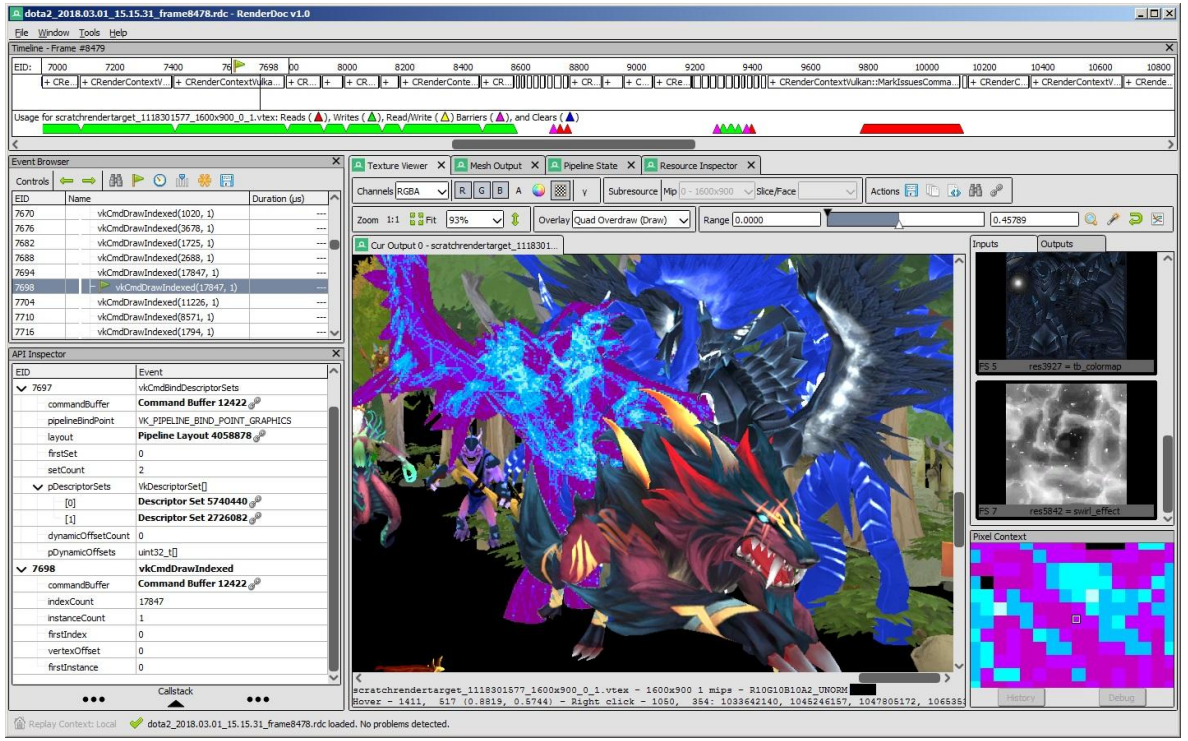


https://vulkan.lunarg.com/doc/view/1.4.350.0/windows/vkconfig.html

RenderDoc

FRAME DEBUGGER

- Inspect the state of frame, and it's associated resources, command by command
- Naming your resources via vkSetDebugUtilsObjectNameEXT (VK_EXT_debug_utils) is strongly recommended



https://renderdoc.org/

Vendor Tools

VENDOR TOOLS

NVIDIA: https://developer.nvidia.com/nsight-graphics

AMD: https://gpuopen.com/tools/

Arm: http://developer.arm.com/mobile-studio

Android: https://developer.android.com/android-performance-analyzer

Samsung: https://github.com/sarc-acl/sokatoa (multi-vendor)

Device Loss / Device Fault Diagnostics

CRASH DIAGNOSTIC LAYER

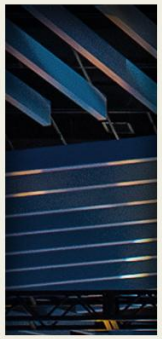
- Tells you what range of commands were executing when a GPU crashed
- Can warn you about invalid memory accesses
- Available in the Vulkan SDK (enable via vkconfig)

```
-  # Command:
  id: 17
  checkpointValue: 0x00000012
  name: vkCmdBeginDebugUtilsLabelEXT
  state: COMPLETED
  Labels:
    - Render Mesh
  Parameters: (...)
  message: "'>>>>>>>>>>>>>>> LAST COMPLETE COMMAND <<<<<<<<<<<<<'"

(more commands)

-  # Command:
  id: 24
  checkpointValue: 0x00000019
  name: vkCmdDrawIndexed
  state: INCOMPLETE
  labels:
    - Render Mesh
  parameters:
    indexCount: 8511627
    instanceCount: 1
    firstIndex: 0
    vertexOffset: 0
    firstInstance: 0
    internalState:
      pipeline: {}
      descriptorSets: []
    message: "'^^^^^^^^^^^^^^^ LAST STARTED COMMAND ^^^^^^^^^^^^^^'"
```

https://www.khronos.org/assets/uploads/developers/presentations/Siggraph_BoF_-_Crash_Diagnostic_Layer.pdf

Rolling Your Own Debug Tooling

Extension	Purpose
VK_EXT_debug_utils	Object/region labelling, debug messages
VK_KHR_device_fault	Page/instruction faults, crash dumps
VK_KHR_shader_abort	Enables writing assert functions in shaders
VK_EXT_device_address_binding_report	Maps GPU addresses to Vulkan objects





SIGGRAPH 2026
Los Angeles 19–23 JUL

Conclusion

What we covered today

- Getting started: device creation, API baseline, resources and memory
- Recording work: command buffers, pools and queues
- Shaders and pipelines: authoring shaders, pipelines and shader objects
- Drawing and rendering: geometry, dynamic rendering and attachments
- Binding data: descriptor heaps
- Correctness and performance: synchronization and frames in flight
- Tooling: debugging and validation

Prefer modern functionality over legacy

LEGACY APPROACH

- Render passes and framebuffers
- Fixed-state pipeline objects
- Descriptor set layouts and pools
- Binary semaphores and fences
- Manual layout transitions, legacy barriers
- Subpass input attachments

USE INSTEAD — COVERED IN THIS TALK

- Dynamic rendering (VK_KHR_dynamic_rendering / 1.3)
- Shader objects (VK_EXT_shader_object)
- Descriptor heaps (VK_EXT_descriptor_heap)
- Timeline semaphores (VK_KHR_timeline_semaphore / 1.2)
- Synchronization2 and unified image layouts
- Dynamic rendering local read (1.4)

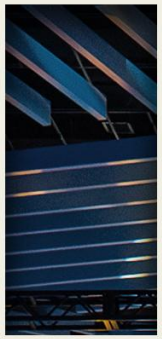
Takeaways

- Vulkan has evolved significantly over the last decade
- Recent extensions have significantly improved usability and developer experience:
  - VK_EXT_descriptor_heap
  - VK_EXT_shader_object
  - VK_KHR_unified_image_layouts
  - VK_KHR_dynamic_rendering (or Vulkan 1.3)
  - VK_KHR_dynamic_rendering_local_read (or Vulkan 1.4)
  - VK_KHR_timeline_semaphore (or Vulkan 1.2)
- Libraries can help (vk-bootstrap, VMA, Vulkan-HPP)
- Using Validation Layers is essential for correctness

Questions?

- Code, materials, and slides are available at: tinyurl.com/24fsppy9

2026

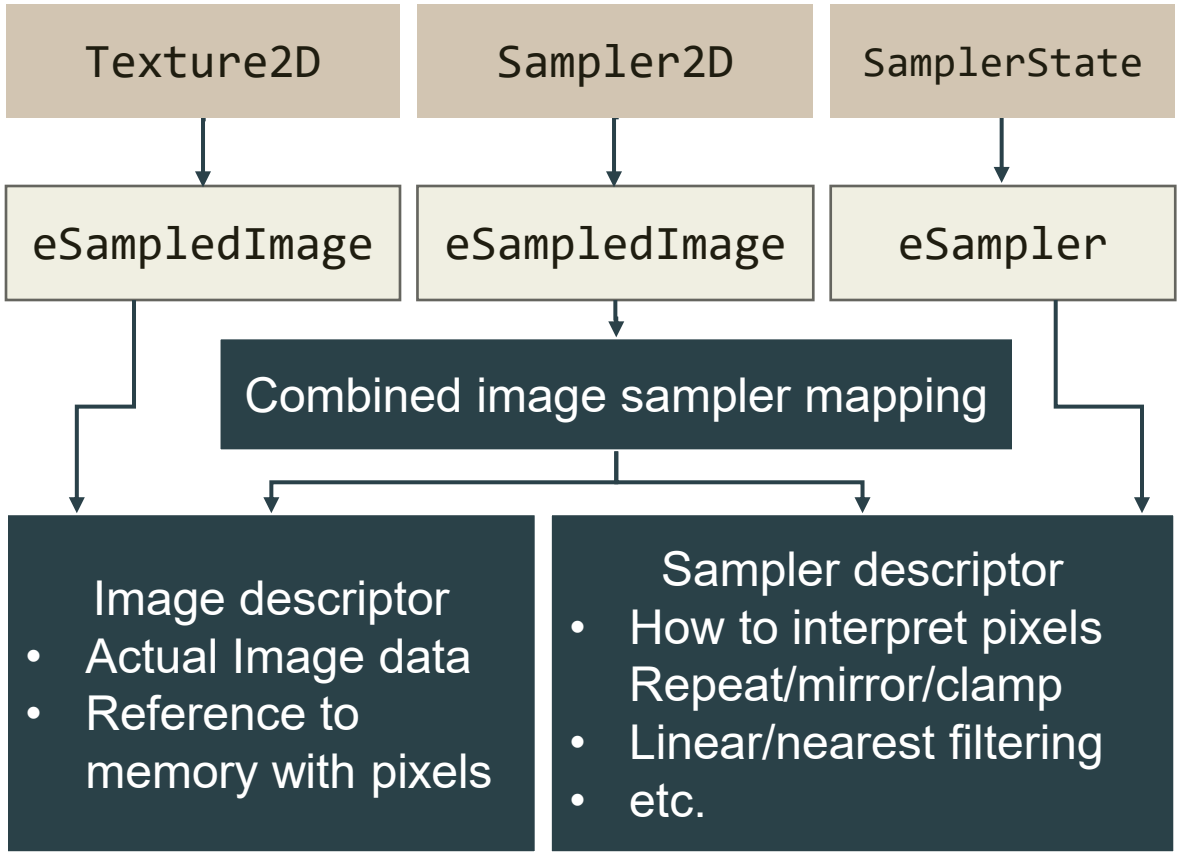




SIGGRAPH 2026
Los Angeles 19–23 JUL

Annex – Sampler Mip Map levels

Descriptor Heap



SAMPLERS

- A sampler defines how to read an image in Vulkan
  - Texture: Actual image data / texels
  - Sampler: How to interpret the data
    - Repeat at border, interpolate, etc.
- Separate descriptor for samplers and images `SamplerState` and `Texture2D`
- Combined descriptor for image and sampler `Sampler2D`
- Descriptor heap has a separate heap for samplers
  - Descriptor heap support separate and combined mappings

Descriptor Heap

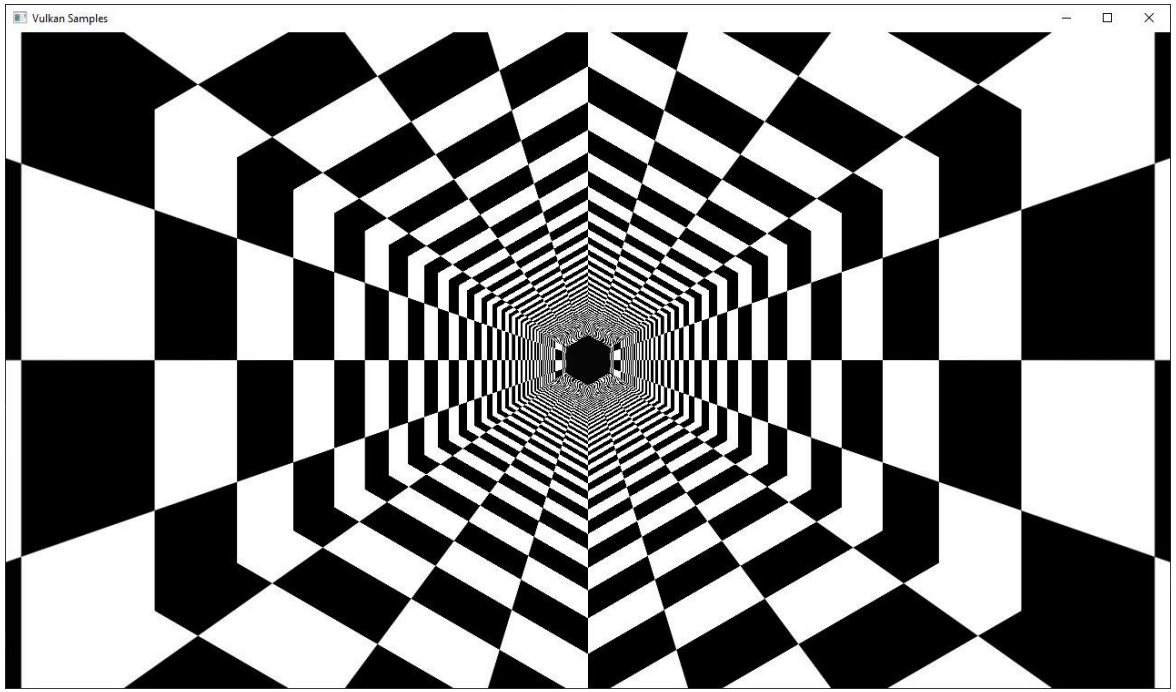
```
Application::GpuImage Application::createTexture(
        std::string_view path, std::string_view debugName) {
  // Create the image object.
  const vk::ImageCreateInfo imageInfo{
    .mipLevels =
   static_cast<uint32_t>(std::floor(std::log2(std::max(sourceImage.width,
   sourceImage.height))) + 1.0F),
    /* ... */ };

  //...

  // Upload the image data from the CPU to the GPU
  GpuBuffer stagingBuffer = uploadToNewStagingBuffer(sourceImage.pixels);

  // Copy the initial image to mip level 0 (biggest one)
  const vk::CopyBufferToImageInfo2 copyInfo{ /* Select Mip level 0 */};
  commandBuffer.copyBufferToImage2(copyInfo);

  for (uint32_t mipLevel = 1; mipLevel < imageInfo.mipLevels; ++mipLevel) {
    const vk::BlitImageInfo2 blitInfo{
        //Calculate blit from mipLevel - 1 to mipLevel
    };
    commandBuffer.blitImage2(blitInfo);

    // Create memory barriers to transition each mip level
  }
  // ..
}
```

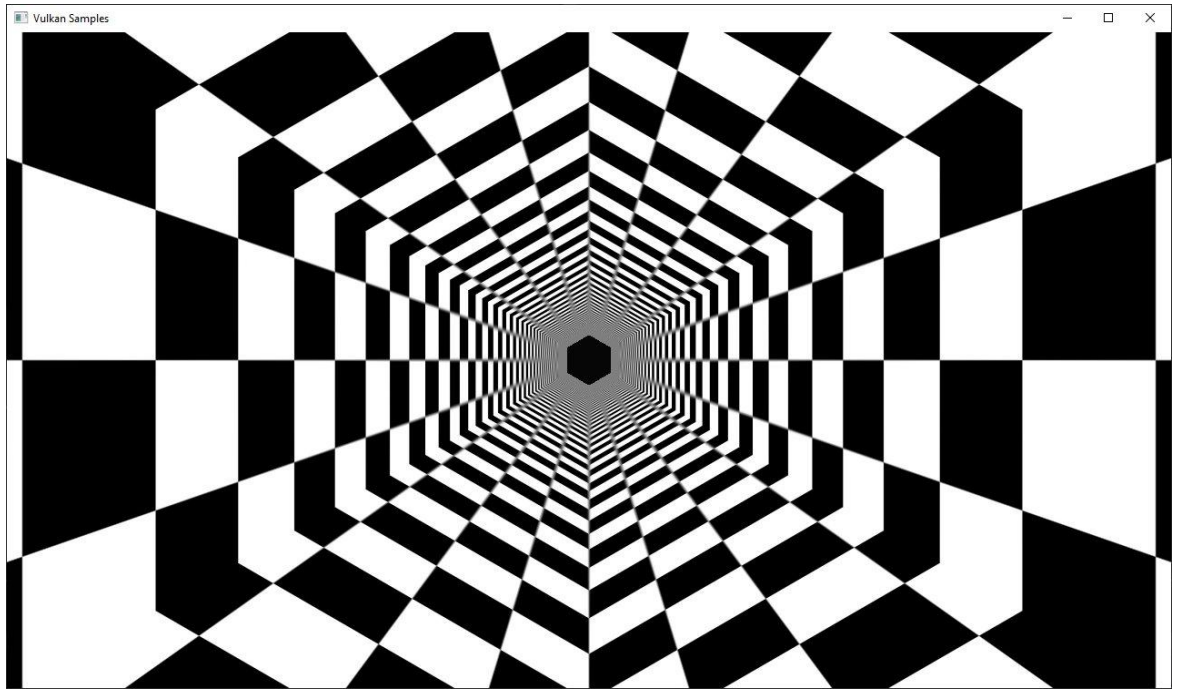
MIP LEVELS

- Samplers use mip levels allow efficient sampling of textures at different distances
  - Improves visual quality and performance when sampling textures
- Need to generate them:
  - Use blit operations to generate mip levels on the GPU
  - Copy initial image to mip level 0 (biggest one)
- You can also pre-generate mip levels and upload them
  - KTX texture format supports storing multiple mip levels

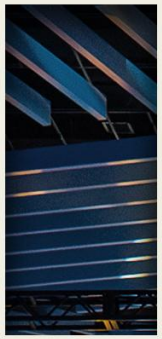
Vulkan Resources

MIP LEVELS





- Comparison from https://docs.vulkan.org/samples/latest/samples/api/texture_mipmap_generation/README.html





SIGGRAPH 2026
Los Angeles 19–23 JUL

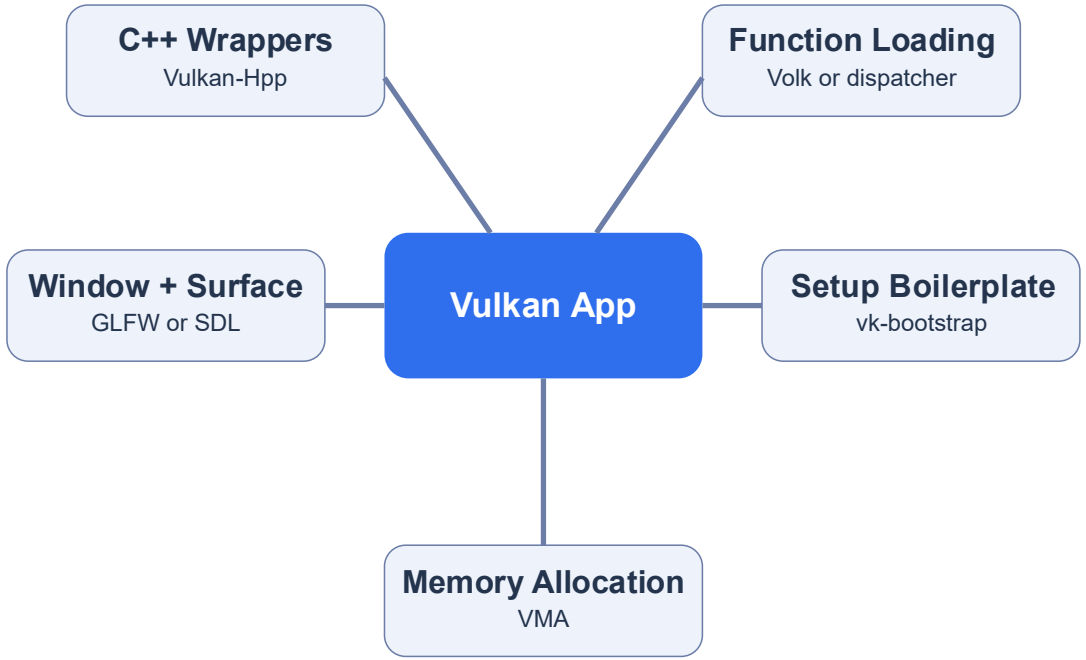
Annex – Libraries and helpers

Libraries And Helpers

INTRODUCTION

- Modern Vulkan apps usually rely on a few helper libraries
- Helpers remove boilerplate around loading, setup, windows, and memory
- They do not replace the Vulkan object model
- The goal is less repetitive code, not less understanding

Libraries And Helpers



WHAT PROBLEMS THESE HELPERS SOLVE

- Vulkan-Hpp gives typed C++ Vulkan wrappers
- Volk or a dispatcher solves Vulkan function loading
- GLFW or SDL creates windows and Vulkan surfaces
- vk-bootstrap shortens core Vulkan setup
- VMA shortens buffer and image memory allocation

Libraries And Helpers

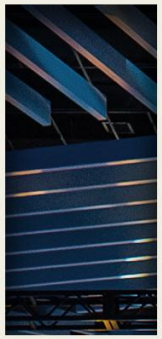
Helper	Solves	What you still manage
Vulkan-Hpp	Typed C++ wrappers (RAII, boilerplate, function pointers)	Vulkan objects and logic
Volk	Vulkan Function Loading	When and how loading happens
SDL / GLFW	Window and surface creation	Presentation behavior
VMA	Allocation boilerplate	Resource lifetime and synchronization

- Vulkan is a C API, Vulkan-Hpp official wrapper designed by Khronos for C++ style binding
- Vulkan can be used from most languages; third party binding exist for most popular language

Libraries And Helpers

USING VK-BOOTSTRAP

- vk-bootstrap shortens instance, device, and swapchain setup
- It is useful when setup boilerplate would otherwise dominate the code
- The app still chooses Vulkan version, features, and presentation policy
- It removes setup code, not Vulkan object dependencies





SIGGRAPH 2026
Los Angeles 19–23 JUL

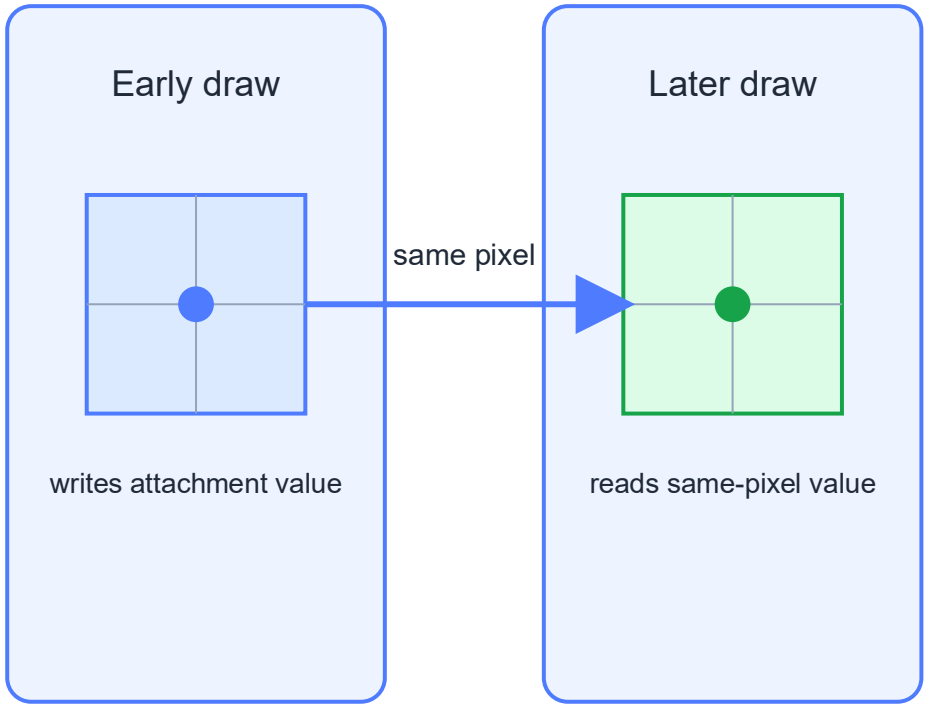
Annex – Dynamic rendering local read

Dynamic Rendering Local Read

INTRODUCTION

- Describes input attachments as shader reads from render attachments
- Reads the value at the current framebuffer location
- Avoids arbitrary texture sampling when the shader only needs the current pixel
- Fits tile-based GPUs well because the data can stay local
- Keeps input attachment reads tied to the current fragment location

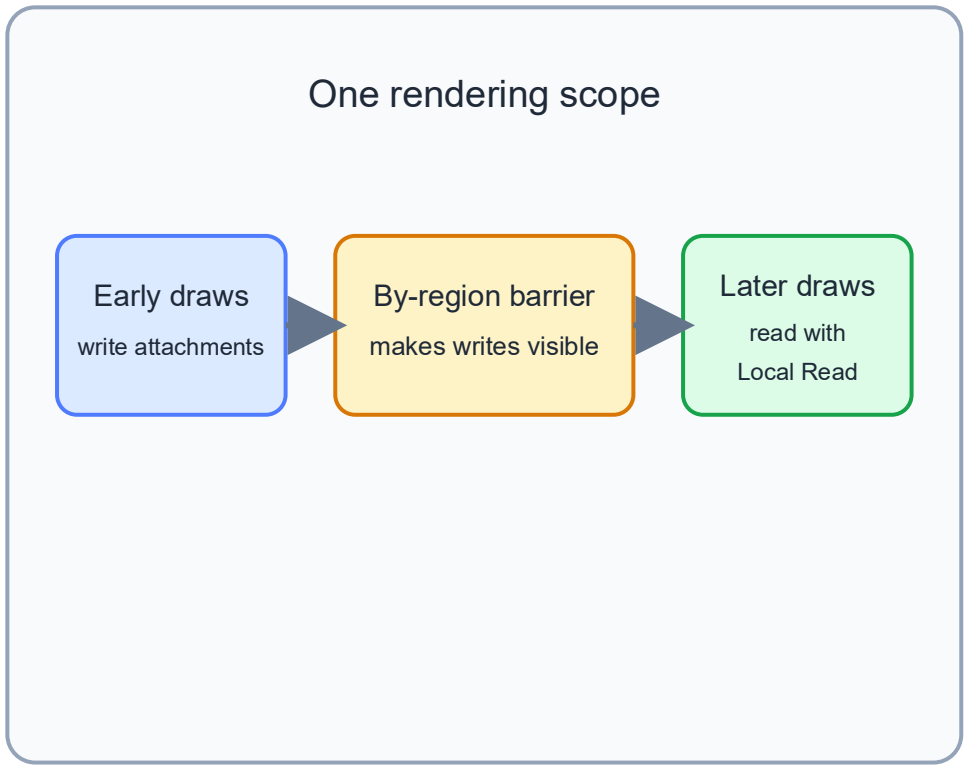
Dynamic Rendering Local Read



SAME-PIXEL READ

- First draws write attachment values
- A later draw reads the value at the same pixel
- The read follows fragment position, not arbitrary UVs
- This is the behavior Local Read keeps in dynamic rendering

Dynamic Rendering Local Read



LOCAL READ FLOW

- Begin one rendering scope
- Write attachments in early draws
- Insert an explicit by-region barrier
- Read those attachments in later draws
- End rendering after those later draws

Dynamic Rendering Local Read

```
// Set 0, binding 0, input attachment index 0
[[vk::binding(0, 0)]]
[[vk::input_attachment_index(0)]]
SubpassInput<float4> gbuffer_albedo;

[shader("fragment")]
float4 fragmentMain() : SV_Target
{
    // Read the input attachment at the current pixel
    return gbuffer_albedo.SubpassLoad();
}
```

SHADER VIEW

- Reads an input attachment
- Reads the value at the current framebuffer location
- Does not sample arbitrary UVs or neighboring pixels
- Connects one shader input to one input attachment

Dynamic Rendering Local Read

```
physicalDeviceSelector
  .add_required_extension_features(
    VkPhysicalDeviceSynchronization2FeaturesKHR{
      .sType = VK_..._SYNCHRONIZATION_2_FEATURES_KHR,
      .synchronization2 = VK_TRUE,
  })
  .add_required_extension_features(
     VkPhysicalDeviceDynamicRenderingFeaturesKHR{
       .sType = VK_..._DYNAMIC_RENDERING_FEATURES_KHR,
       .dynamicRendering = VK_TRUE,
  })
  .add_required_extension_features(
     VkPhysicalDeviceDynamicRenderingLocalReadFeaturesKHR{
       .sType = VK_..._DYNAMIC_RENDERING_LOCAL_READ_FEATURES_KHR,
       .dynamicRenderingLocalRead = VK_TRUE,
  });
```

REQUIRED SETUP

- Enables dynamicRendering
- Enables dynamicRenderingLocalRead
- Enables synchronization2 for the local-read barrier
- Chains those feature structs during device creation

Dynamic Rendering Local Read

```
// Begin rendering with local-read attachments
std::array<vk::RenderingAttachmentInfoKHR, 4>
   color_attachment_info{};

for (auto& color_attachment : color_attachment_info) {
    color_attachment = vk::RenderingAttachmentInfoKHR{
        .imageLayout = vk::ImageLayout::eRenderingLocalReadKHR,
        .loadOp      = vk::AttachmentLoadOp::eClear,
        .storeOp     = vk::AttachmentStoreOp::eStore,
    };
}

vk::RenderingInfoKHR render_info{
    .colorAttachmentCount = color_attachment_info.size(),
    .pColorAttachments    = color_attachment_info.data(),
    .pDepthAttachment     = &depth_attachment_info,
};

auto& inputIdx = rendering_attachment_index_info;

cmd.beginRenderingKHR(render_info);
cmd.setRenderingInputAttachmentIndicesKHR(inputIdx);
```

BEGIN RENDERING

- Builds attachments that will be written and later read
- Uses the local-read layout for those color attachments
- Begins one rendering scope for both the write and read passes
- Sets input attachment indices for the later shader reads

Dynamic Rendering Local Read

```
// Insert by-region barrier
vk::DependencyInfo dependencyInfo{
    .dependencyFlags    = vk::DependencyFlagBits::eByRegion,
    .memoryBarrierCount = 1,
    .pMemoryBarriers    = &memoryBarrier,
};

cmd.pipelineBarrier2(dependencyInfo);

// Run later draws that read those attachments
cmd.bindPipeline(
    vk::PipelineBindPoint::eGraphics, composition_pass.pipeline);

cmd.draw(3, 1, 0, 0);

cmd.bindPipeline(
    vk::PipelineBindPoint::eGraphics
    scene_transparent_pass.pipeline);
draw_scene(scenes.transparent, cmd,
           scene_transparent_pass.pipeline_layout);

// End rendering
cmd.endRendering();
```

BARRIER AND LATER DRAWS

- Early draws write the G-buffer attachments
- A by-region barrier makes those writes visible to later reads
- The composition draw reads with SubpassLoad
- The rendering scope stays open until those later draws finish

Dynamic Rendering Local Read

RENDERPASS + SUBPASSES VS DYNAMIC RENDERING LOCAL READ

- Render passes advance between subpasses with vkCmdNextSubpass
- Local Read uses an explicit by-region barrier
- Render pass dependencies live in render pass setup
- Local Read dependencies live next to the draw calls
- Shader-side input attachment reads stay the same
- Fallback is render passes and subpasses when support is missing

Before	Now
vkCmdNextSubpass	Explicit by-region barrier
Subpass dependency in render pass setup	Dependency recorded next to the later draw
Render pass with multiple subpasses	One dynamic rendering scope
SubpassInput and SubpassLoad	SubpassInput and SubpassLoad