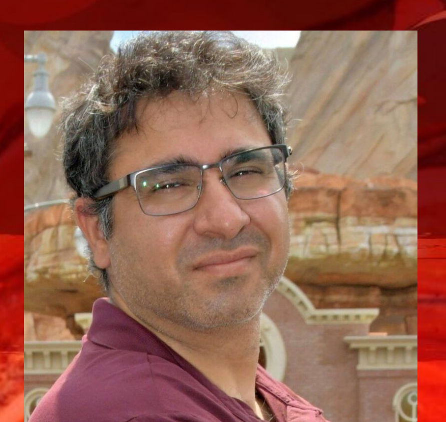
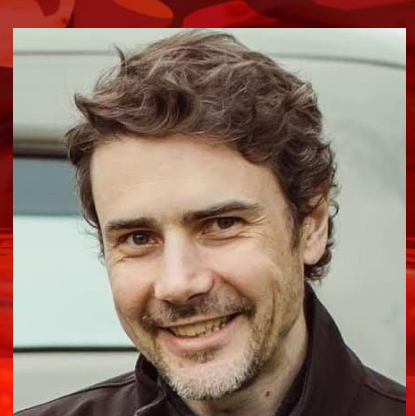
The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

# From Vulkan 1.1 to 1.4: Streamlining Your Code with New Features

Preetish Kakkar, Senior Staff Engineer, Qualcomm Technologies Inc.
Mauricio Maurer, Independent





# Agenda

- Problem statement (why migrate?)
- Pain Points
- Migration Philosophy
- What’s new
- Our code
- Dynamic rendering local read
- Push descriptors
- Shader Objects
- Conclusion

# Problem Statement

- Simple Vulkan wrapper + multiple fully contained samples
  - Sample code that accompany our introductory Vulkan book
- Goal was to modernize both *wrapper* and *samples*
- No intent to publish it (yet)
  - Current intent is just to modernize the codebase and present newer Vulkan code to readers

# Pain Points (RenderPass complexity)

- Requires rigid upfront description of entire render pass
- RenderPasses were designed with tile-based GPUs, however it leaks hardware details into the API design
- Can't change attachments dynamically
- Framebuffer tied to specific image views
- Very easy to make mistakes during setup of the Render Pass
- Spaghetti Vulkan
  - Daunting for beginners, especially in a book format

# Pain Points (RenderPass complexity)

## RENDER PASS: ATTACHMENT DESCRIPTIONS

```
// Must describe EVERY attachment upfront
VkAttachmentDescription offscreenColorAttachment{};
offscreenColorAttachment.format = swapchainImageFormat;
offscreenColorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
offscreenColorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
offscreenColorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
offscreenColorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
offscreenColorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
offscreenColorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
offscreenColorAttachment.finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkAttachmentDescription depthAttachment{};
// ... another 10 lines ...

VkAttachmentDescription swapchainAttachment{};
// ... another 10 lines ...
```

## RENDER PASS: SUBPASS SETUP

```
// Subpass 0: Geometry rendering
VkAttachmentReference colorAttachmentRef{};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkAttachmentReference depthAttachmentRef{};
depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

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

// Subpass 1: Post-processing... another 15 lines
```

## RENDER PASS: DEPENDENCIES

```
// Must manually specify synchronization between subpasses
std::array<VkSubpassDependency, 3> dependencies{};

// External -> Subpass 0
dependencies[0].srcSubpass = VK_SUBPASS_EXTERNAL;
dependencies[0].dstSubpass = 0;
dependencies[0].srcStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
dependencies[0].dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
                               VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
dependencies[0].srcAccessMask = VK_ACCESS_MEMORY_READ_BIT;
dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                                VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

// Subpass 0 -> Subpass 1... another dependency
// Subpass 1 -> External... another dependency
```

## RENDER PASS: FRAMEBUFFERS

```
// Need a framebuffer for EACH swapchain image
framebuffers.resize(swapchainImageViews.size());

for (size_t i = 0; i < swapchainImageViews.size(); i++) {
  std::array<VkImageView, 3> attachments = {
      offscreenColorImageView,  // Attachment 0
      depthImageView,           // Attachment 1
      swapchainImageViews[i]    // Attachment 2
  };

  VkFramebufferCreateInfo framebufferInfo{};
  framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
  framebufferInfo.renderPass = renderPass;
  framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
  framebufferInfo.pAttachments = attachments.data();
  framebufferInfo.width = swapchainExtent.width;
  framebufferInfo.height = swapchainExtent.height;
  framebufferInfo.layers = 1;

  vkCreateFramebuffer(device, &framebufferInfo, nullptr, &framebuffers[i]);
}
```

# Pain Points (Descriptor Set)

- Requires pre allocating pools (can exhaust at runtime if you didn’t get it right)
- Complex lifetime management
- In case of streaming, requires reallocation all the time.
- Biggest hurdle to understanding the connection between code/resources and shaders for beginners

# Pain Points (Descriptor Set)

## DESCRIPTORS: POOL CREATION

```
// Must know EXACT counts upfront!
void Renderer::createDescriptorPool() {
  std::array<VkDescriptorPoolSize, 3> poolSizes{};

  poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  poolSizes[0].descriptorCount = MAX_FRAMES_IN_FLIGHT;  // Hope this is enough!

  poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
  poolSizes[1].descriptorCount = MAX_FRAMES_IN_FLIGHT;  // Guessing again...

  poolSizes[2].type = VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT;
  poolSizes[2].descriptorCount = 1;

  VkDescriptorPoolCreateInfo poolInfo{};
  poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
  poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
  poolInfo.pPoolSizes = poolSizes.data();
  poolInfo.maxSets = MAX_FRAMES_IN_FLIGHT + 1;  // Don't forget +1!

  vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);
}
```

## DESCRIPTORS: LAYOUT CREATION

```
void Renderer::createDescriptorSetLayouts() {
  // Cube descriptor set layout
  VkDescriptorSetLayoutBinding uboLayoutBinding{};
  uboLayoutBinding.binding = 0;
  uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
  uboLayoutBinding.descriptorCount = 1;
  uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

  VkDescriptorSetLayoutBinding samplerLayoutBinding{};
  samplerLayoutBinding.binding = 1;
  samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
  samplerLayoutBinding.descriptorCount = 1;
  samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

  std::array bindings = {uboLayoutBinding, samplerLayoutBinding};

  VkDescriptorSetLayoutCreateInfo layoutInfo{};
  layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
  layoutInfo.pBindings = bindings.data();

  vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &cubeDescriptorSetLayout);

  // Post-process layout... another 20 lines
}
```

## DESCRIPTORS: ALLOCATION & UPDATE

```
void Renderer::createDescriptorSets() {
  // Allocate from pool
  std::vector<VkDescriptorSetLayout> layouts(MAX_FRAMES_IN_FLIGHT, cubeDescriptorSetLayout);
  VkDescriptorSetAllocateInfo allocInfo{};
  allocInfo.descriptorPool = descriptorPool;
  allocInfo.descriptorSetCount = MAX_FRAMES_IN_FLIGHT;
  allocInfo.pSetLayouts = layouts.data();

  cubeDescriptorSets.resize(MAX_FRAMES_IN_FLIGHT);
  vkAllocateDescriptorSets(device, &allocInfo, cubeDescriptorSets.data());

  // Update EACH set for EACH frame
  for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
    VkDescriptorBufferInfo bufferInfo{};
    bufferInfo.buffer = uniformBuffers[i];
    bufferInfo.offset = 0;
    bufferInfo.range = sizeof(UniformBufferObject);

    VkDescriptorImageInfo imageInfo{};
    imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    imageInfo.imageView = textureImageView;
    imageInfo.sampler = textureSampler;

    std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
    // ... setup both writes, 20 more lines ...

    vkUpdateDescriptorSets(device, descriptorWrites.size(), descriptorWrites.data(), 0, nullptr);
  }
}
```

# Pain Points (Pipelines)

- Pipeline creation is convoluted/lengthy
  - Overwhelming for new Vulkan users
- Shader modules aren't of any use after pipeline is created
- Pipeline caching is complex
  - And necessary (but not in our case)
- Pipeline switching can cause hitching when created during runtime
- Every state combination require a new pipeline
  - Exponential explosion

| State | Options |
| --- | --- |
| Polygon Mode | Fill, Line, Point = 3 |
| Cull Mode | None, Front, Back, Both = 4 |
| Depth Test | On, Off = 2 |
| Depth Write | On, Off = 2 |
| Blend Mode | Opaque, Alpha, Additive = 3 |

**3 × 4 × 2 × 2 × 3 = 144 pipelines!**

# Pain Points (Pipelines)

## PIPELINES: STATE IS BAKED IN

```
// ALL FIXED at pipeline creation
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.polygonMode = VK_POLYGON_MODE_FILL; // Can't change!
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;    // Can't change!
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizer.lineWidth = 1.0f;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.depthBiasEnable = VK_FALSE;

VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.depthTestEnable = VK_TRUE;         // Can't change!
depthStencil.depthWriteEnable = VK_TRUE;        // Can't change!
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS; // Fixed!
```

## PIPELINES EXPLOSION

```
// Want wireframe? NEW PIPELINE!
VkPipelineRasterizationStateCreateInfo wireframe = rasterizer;
wireframe.polygonMode = VK_POLYGON_MODE_LINE;

vkCreateGraphicsPipelines(device, cache, 1, &solidInfo,
                          nullptr, &solidPipeline);
vkCreateGraphicsPipelines(device, cache, 1, &wireframeInfo,
                          nullptr, &wireframePipeline);

// At runtime - pick correct pipeline
VkPipeline p = useWireframe ? wireframePipeline : solidPipeline;
vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, p);
```

# Migration Philosophy

- “Delete code, don’t just add features” – Less code means easier maintaince
- Don’t rewrite everything at once, migrate in chunks, use mandatory features first that require no fallbacks.
- Profile old vs new implementation. Use validation layer extensively during migration.
- Vulkan 1.4 provides ability for more dynamic/runtime decision, carefully identify static vs dynamic requirements.
- Switch to buffer device address and timeline semaphores if you haven't already.
- Prioritization
  - Which features offer the biggest bang for the buck
  - In our case that meant ease of understanding, not just performance gains
- Backwards compatibility considerations
  - Platform (desktop, mobile, HMDs)
  - Available GPUs (architecture, driver version)
  - In short: “everything that affects feature availability”

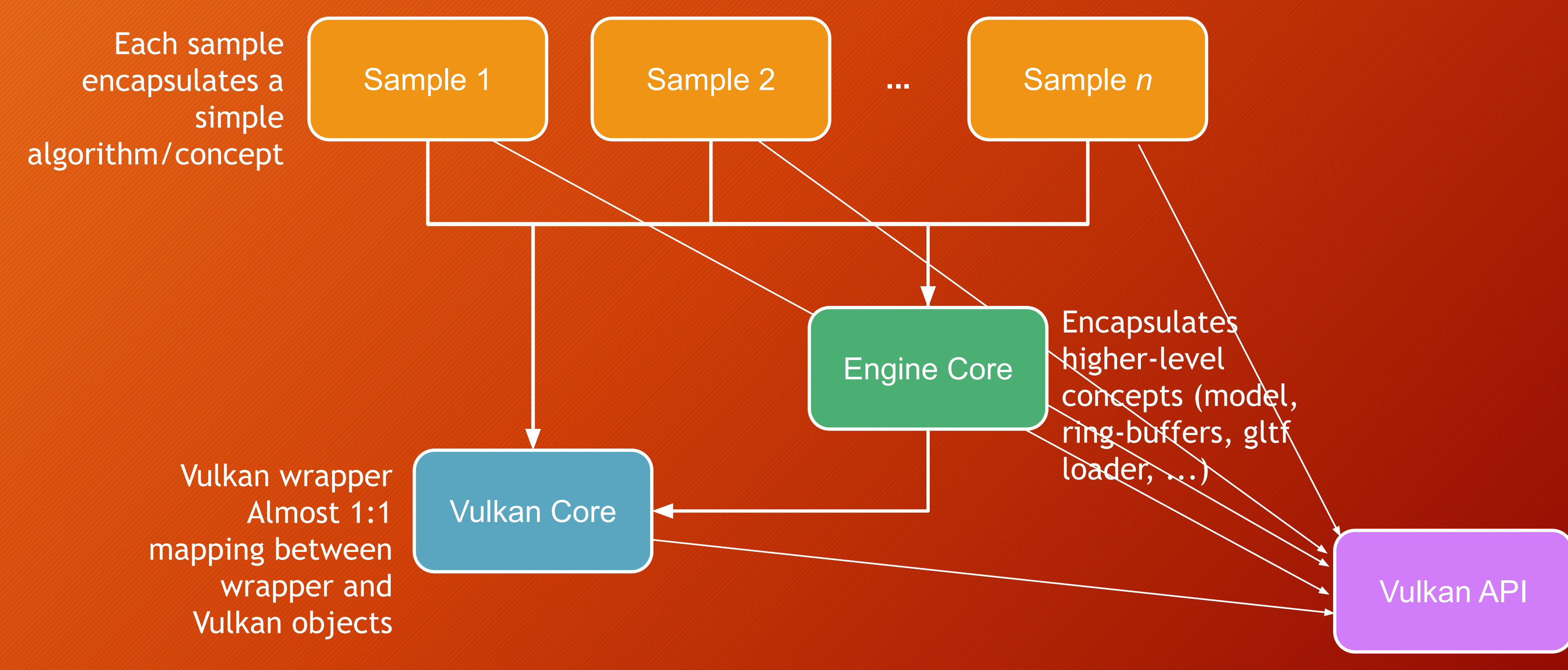
# What’s new

- Subgroup Operations (1.1 Core, enhanced in 1.3/1.4)
- Buffer Device Address (1.2 Core)
- Host Image Copy (1.2 Core)
- Scalar Block Layout (1.2 Core)
- Timeline Semaphores (1.2 Core)
- Dynamic Rendering (1.3 Core)
- Extended Dynamic States (1.3 Core)
- Synchronization2 (1.3 Core)
- Dynamic Rendering Local Read (1.4 Core)
- Maintenance 5 & 6 (1.4 Core)
- Push Descriptors (1.4 Core)
- Pipeline Binary (Extension - VK_KHR_pipeline_binary)
- Pipeline Robustness (Extension - VK_EXT_pipeline_robustness)
- Shader Objects (Extension – VK_EXT_shader_object)

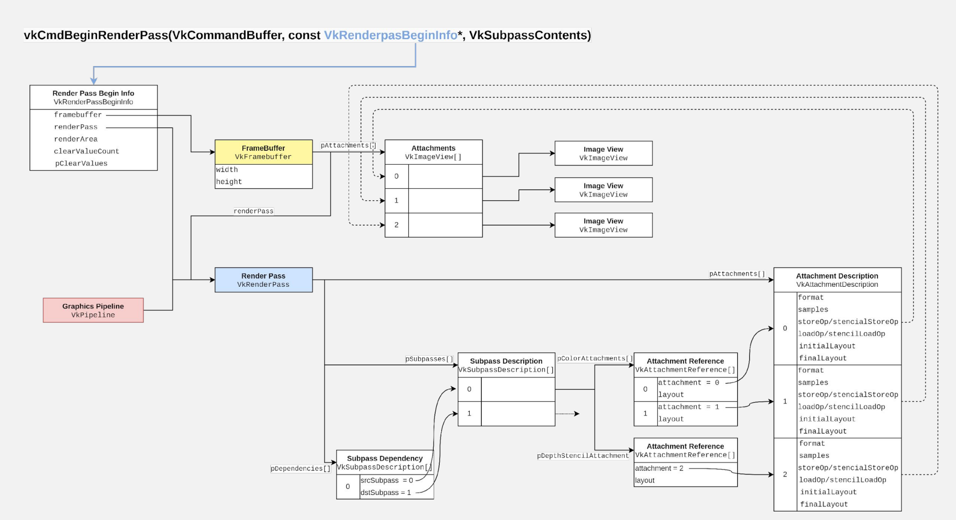
# What’s new

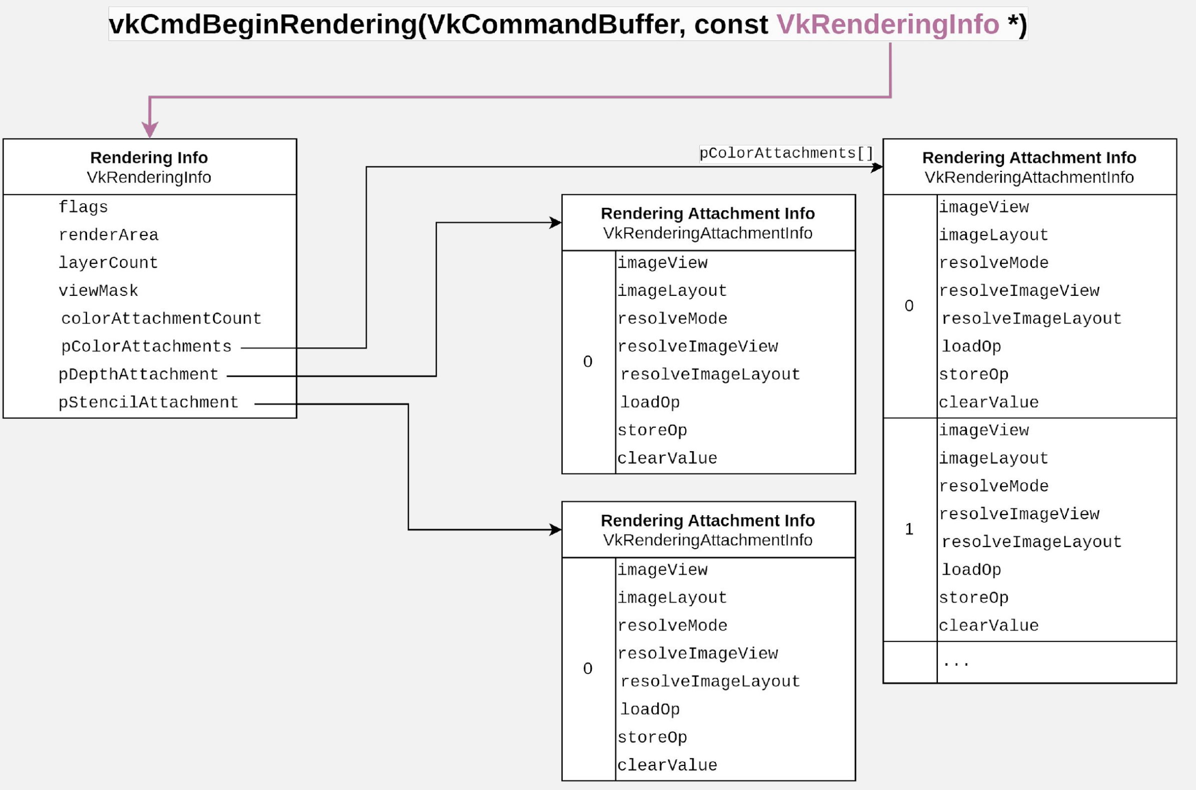
- Subgroup Operations (1.1 Core, enhanced in 1.3/1.4)
- Buffer Device Address (1.2 Core)
- Host Image Copy - VkMemoryToImageCopy (1.4 Core)
- Scalar Block Layout (1.2 Core)
- Timeline Semaphores (1.2 Core)
- **Dynamic Rendering (1.3 Core)**
- Extended Dynamic States (1.3 Core)
- Synchronization2 (1.3 Core)
- **Dynamic Rendering Local Read (1.4 Core)**
- Maintenance 5 & 6 (1.4 Core)
- **Push Descriptors (1.4 Core)**
- Pipeline Binary (Extension - VK_KHR_pipeline_binary)
- Pipeline Robustness (Extension - VK_EXT_pipeline_robustness)
- **Shader Objects (Extension - VK_EXT_shader_object)**
- **Device Generated Commands (Extension - VK_EXT_device_generated_commands)**

# Our Code



# Dynamic Rendering




In the olden days...

Now

# Dynamic Rendering

## VULKAN 1.1: RENDER PASS SETUP

**118 lines** of configuration

```
// Attachment 0: Offscreen color (10 lines)
VkAttachmentDescription offscreen{};
offscreen.format = swapchainImageFormat;
offscreen.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
offscreen.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
offscreen.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
offscreen.finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

// Attachment 1: Depth (10 lines)
// Attachment 2: Swapchain (10 lines)
// Subpass 0: Geometry (15 lines)
// Subpass 1: Post-process (15 lines)
// Dependencies: 3 × 10 lines each (30 lines)

vkCreateRenderPass(device, &info, nullptr, &renderPass);
```

## VULKAN 1.4: INLINE AT DRAW TIME

**~15 lines** - defined when needed

```
VkRenderingAttachmentInfo colorAttachment{
  .sType = VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO,
  .imageView = swapchainImageViews[imageIndex],
  .imageLayout = VK_IMAGE_LAYOUT_RENDERING_LOCAL_READ,
  .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR,
  .storeOp = VK_ATTACHMENT_STORE_OP_STORE
};

VkRenderingAttachmentInfo depthAttachment{
  .sType = VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO,
  .imageView = depthImageView,
  .imageLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
};

VkRenderingInfo renderingInfo{
  .sType = VK_STRUCTURE_TYPE_RENDERING_INFO,
  .renderArea = {.extent = swapchainExtent},
  .layerCount = 1,
  .colorAttachmentCount = 1,
  .pColorAttachments = &colorAttachment,
  .pDepthAttachment = &depthAttachment
};
vkCmdBeginRendering(cmd, &renderingInfo);
```

# Dynamic Rendering

- Focusing on code right now, so no plans to re-write any chapters
  - The simplification is quite considerable in the Vulkan Core (wrapper) part of the code
- RenderPass and Framebuffer classes were "removed"
  - ~550 LOC down to ~130 (utilities to create VkRenderingInfo and VkRenderingAttachmentInfo)
  - Less setup code in each Sample (~40 LOC)

# Dynamic Rendering Local Read

- Help simplify the code slightly in each Sample
- No performance benefits (mostly)
  - No subpasses were used in the Samples

## SUBPASS VS LOCAL READ

### VULKAN 1.1 - SUBPASS

```
vkCmdDrawIndexed(cmd, indexCount, ...);

// Transition to subpass 1
vkCmdNextSubpass(cmd,
    VK_SUBPASS_CONTENTS_INLINE);

// Bind post-process pipeline
vkCmdBindPipeline(cmd,
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    postProcessPipeline);
vkCmdBindDescriptorSets(cmd, ...);

vkCmdDraw(cmd, 3, 1, 0, 0);
vkCmdEndRenderPass(cmd);
```

### VULKAN 1.4 - LOCAL READ

```
vkCmdDrawIndexed(cmd, indexCount, ...);

// Explicit barrier (Synchronization2)
VkMemoryBarrier2 barrier{
  .sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER_2,
  .srcStageMask =
    VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT,
  .srcAccessMask = VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT,
  .dstStageMask = VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT,
  .dstAccessMask = VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT
};
VkDependencyInfo depInfo{
  .sType = VK_STRUCTURE_TYPE_DEPENDENCY_INFO,
  .dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT,
  .memoryBarrierCount = 1,
  .pMemoryBarriers = &barrier
};
vkCmdPipelineBarrier2(cmd, &depInfo);

vkCmdDraw(cmd, 3, 1, 0, 0);
vkCmdEndRendering(cmd);
```

# Push Descriptors

- 400 LOC in the Vulkan wrapper module (not counting code to create pipeline layout )
  - Reduced to ~80 LOC for each sample; Vulkan wrapper code was "removed"
  - Significantly easier to understand and parse
- Complexity moved from Vulkan wrapper to samples
  - More code to type for each sample...
  - ... but code is more mentally manageable
    - You can see the connection between resources and shaders happening right there!

# Shader Objects

- Supported on older GPUs via the Emulation Layer
- Another step towards pipeline-less Vulkan
  - They help avoid all those pipeline issues (combinatorial explosion, long creation times, and inflexible state management)
- Runtime benefits
  - Dynamic shader switching
  - Hot reloading is far easier
  - No need to pre cache thousands of pipeline permutations
- Shaders and states are independent, mix and match can be done at runtime.

# Shader Objects

## VULKAN 1.1: PIPELINE EXPLOSION

```
// ~266 lines of setup...

// Rasterization state BAKED IN
VkPipelineRasterizationStateCreateInfo rast{};
rast.polygonMode = VK_POLYGON_MODE_FILL; // Fixed!
rast.cullMode = VK_CULL_MODE_BACK_BIT;   // Fixed!

// SEPARATE pipeline for wireframe!
VkPipelineRasterizationStateCreateInfo wireframe = rast;
wireframe.polygonMode = VK_POLYGON_MODE_LINE;

vkCreateGraphicsPipelines(..., &solidPipeline);
vkCreateGraphicsPipelines(..., &wireframePipeline);

// Destroy modules (can't reuse easily)
vkDestroyShaderModule(device, cubeVertModule, nullptr);
```

## VULKAN 1.4: SHADER OBJECTS

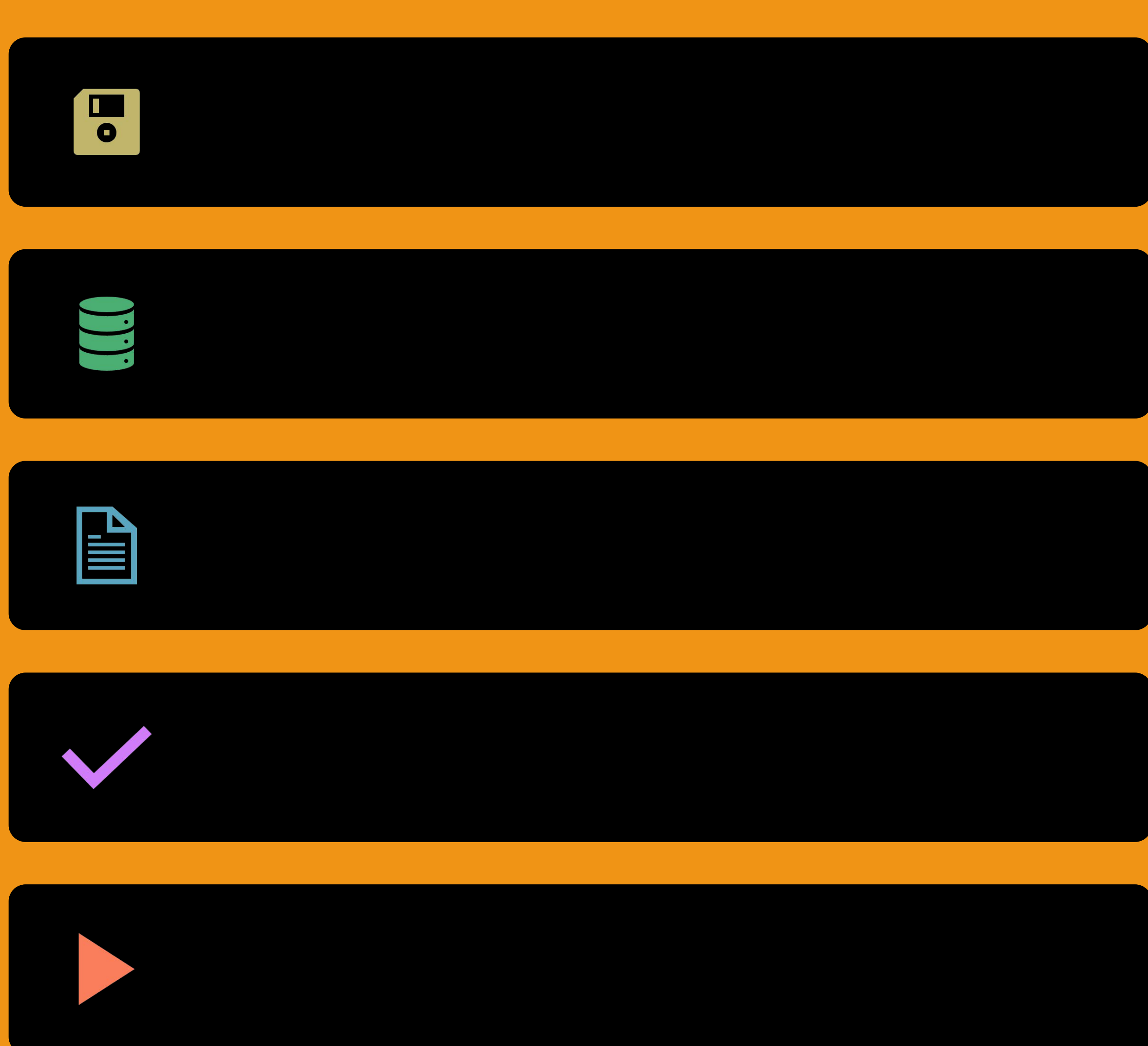
```
// Create shader objects (persist for reuse)
VkShaderCreateInfoEXT shaderInfo{
  .sType = VK_STRUCTURE_TYPE_SHADER_CREATE_INFO_EXT,
  .stage = VK_SHADER_STAGE_VERTEX_BIT,
  .nextStage = VK_SHADER_STAGE_FRAGMENT_BIT,
  .codeType = VK_SHADER_CODE_TYPE_SPIRV_EXT,
  .codeSize = spirvCode.size(),
  .pCode = spirvCode.data(),
  .pName = "main",
  .setLayoutCount = 1,
  .pSetLayouts = &descriptorSetLayout
};
vkCreateShadersEXT(device, 1, &shaderInfo, nullptr, &vertShader);
```

## DYNAMIC STATE AT DRAW TIME

```
// Bind shaders
VkShaderStageFlagBits stages[] = {VK_SHADER_STAGE_VERTEX_BIT,
                                  VK_SHADER_STAGE_FRAGMENT_BIT};
VkShaderEXT shaders[] = {vertShader, fragShader};
vkCmdBindShadersEXT(cmd, 2, stages, shaders);

// NO PIPELINE PERMUTATIONS!
vkCmdSetPolygonModeEXT(cmd, useWireframe ?
  VK_POLYGON_MODE_LINE : VK_POLYGON_MODE_FILL);
vkCmdSetCullMode(cmd, VK_CULL_MODE_BACK_BIT);
vkCmdSetFrontFace(cmd, VK_FRONT_FACE_COUNTER_CLOCKWISE);
vkCmdSetDepthTestEnable(cmd, VK_TRUE);
vkCmdSetDepthWriteEnable(cmd, VK_TRUE);
vkCmdSetPrimitiveTopology(cmd, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST);
```

# Shader Objects Caching



# Shader Objects Caching

## CREATING WITH CAPTURE FLAG

```
// Create WITH capture flag to enable caching
VkShaderCreateInfoEXT shaderInfo{
  .sType = VK_STRUCTURE_TYPE_SHADER_CREATE_INFO_EXT,
  .flags = VK_SHADER_CREATE_CAPTURE_DATA_BIT_EXT, // Enable!
  .stage = VK_SHADER_STAGE_VERTEX_BIT,
  .nextStage = VK_SHADER_STAGE_FRAGMENT_BIT,
  .codeType = VK_SHADER_CODE_TYPE_SPIRV_EXT,
  .codeSize = spirvCode.size(),
  .pCode = spirvCode.data(),
  .pName = "main"
};
VkShaderEXT shader;
vkCreateShadersEXT(device, 1, &shaderInfo, nullptr, &shader);

// Extract driver-compiled binary
size_t dataSize;
vkGetShaderBinaryDataEXT(device, shader, &dataSize, nullptr);
std::vector<uint8_t> binary(dataSize);
vkGetShaderBinaryDataEXT(device, shader, &dataSize, binary.data());

// Save: shaders/cube.vert.cache
saveToDisk("shaders/cube.vert.cache", binary);
```

## LOADING FROM CACHE

```
// Next run: load binary directly (faster!)
auto binary = loadFromDisk("shaders/cube.vert.cache");

VkShaderCreateInfoEXT shaderInfo{
  .sType = VK_STRUCTURE_TYPE_SHADER_CREATE_INFO_EXT,
  .stage = VK_SHADER_STAGE_VERTEX_BIT,
  .nextStage = VK_SHADER_STAGE_FRAGMENT_BIT,
  .codeType = VK_SHADER_CODE_TYPE_BINARY_EXT, // Binary!
  .codeSize = binary.size(),
  .pCode = binary.data(),
  .pName = "main"
};

// Skip driver compilation!
vkCreateShadersEXT(device, 1, &shaderInfo, nullptr, &shader);
```

# VK_EXT_device_generated_commands

- Basic idea is that drivers read command sequences from storage buffers instead of command buffer
- This implies buffer can be generated directly on GPU and hence eliminates CPU round trip (Fully GPU driven rendering)
- Can be used to implement Dynamic LOD, Frustum and Occlusion culling, particle systems etc.
- DGC buffer is divided into sequences, each sequence must follow a template (i.e. Indirect command layout), each sequence must dispatch/draw work exactly once.
- Think of it as next level to indirect draw/dispatch

# Device generated commands

**Traditional Indirect commands**

- Only single command type (such as **vkCmdDrawIndirect)**
- **Can't update push const per draw**
- **Can't change vertex/index buffers**
- **Just varies draw count, instance count, offset etc, limited control.**

**DGC**

- Multiple commands types per sequence
- Can update push const, index/vertex buffers etc
- *DGC - sequence of commands*

*Sequence 1: [push constants] → [bind buffers] → [draw]*

*Sequence 2: [push constants] → [bind buffers] → [draw]*

*// Each sequence can have different data, commands/layout must be same!*

# DGC workflow

# Device generated commands

```
typedef struct VkIndirectCommandsLayoutCreateInfoEXT {
    VkStructureType                       sType;
    const void*                           pNext;
    VkIndirectCommandsLayoutUsageFlagsEXT  flags;
    VkShaderStageFlags                    shaderStages;      // Active shader stages during execution
    uint32_t                              indirectStride;    // Bytes from one sequence to the next
    VkPipelineLayout                      pipelineLayout;    // Pipeline layout for the commands
    uint32_t                              tokenCount;
    const VkIndirectCommandsLayoutTokenEXT* pTokens;         // Command tokens array defining the sequence
} VkIndirectCommandsLayoutCreateInfoEXT;
```

```
typedef struct VkIndirectCommandsLayoutTokenEXT {
    VkStructureType                 sType;
    const void*                     pNext;
    VkIndirectCommandsTokenTypeEXT  type;
    VkIndirectCommandsTokenDataEXT  data;
    uint32_t                        offset;
} VkIndirectCommandsLayoutTokenEXT;
```

# Migration checklist

## Phase 1: Foundation

1. Update to Vulkan 1.4 SDK (headers, loader, validation layers)
2. Enable VK LAYER KHRONOS validation during development
3. Verify existing code still compiles and runs
4. Migrate to Synchronization2 (vkCmdPipelineBarrier2)
5. Switch to Buffer Device Address (if not already using)
6. Switch to Timeline Semaphores (if not already using)

## Phase 2: Core Simplifications

7. Replace VkRenderPass with vkCmdBeginRendering (Dynamic Rendering)
8. Delete all VkFramebuffer objects (~140 lines saved)
9. Replace descriptor pools/sets with vkCmdPushDescriptorSet (~80 lines saved)
10. Test on multiple GPUs (NVIDIA, AMD, Intel)
11. Profile frame times (CPU & GPU) - compare old vs new
12. Run validation layers extensively

## Phase 3: Advanced Features

1. Query VK EXT shader object support at runtime
2. Migrate pipelines to shader objects (with fallback path)
3. Implement dynamic state (polygon mode, depth/stencil, blend)
4. Add Dynamic Rendering Local Read for post-processing (mobile priority)
5. Test hot-reloading workflow with shader objects
6. Profile bandwidth savings on mobile devices

# Migration checklist

## Critical Rules

1. Don't rewrite everything at once - migrate one subsystem at a time
2. Always query feature support - never assume availability
3. Keep old code paths during transition for fallback
4. Profile old vs new - measure everything (frame time, memory, bandwidth)
5. Test each step before moving to the next
6. Use validation layers - catch sync errors early
7. Identify static vs dynamic - don't make everything dynamic (overhead!)
8. Prioritize understanding over raw performance gains

## Platform Specific Checks

9. Desktop: Verify shader object support (widely available)
10. Mobile: Test local read bandwidth savings (huge win for tile-based GPUs)
11. VR/XR: Check per-device feature support (may lag behind desktop)
12. Check driver versions for known bugs/workarounds

## Success Metrics

1. Code reduction achieved (~37% target)
2. Frame times maintained or improved
3. Memory bandwidth reduced (especially mobile)
4. Developer velocity increased (faster iteration)
5. Bug count reduced (simpler code = fewer bugs)
6. Maintainability (in our case “readability”)

# Conclusion

- Vulkan 1.4 simplifies some very important features
  - Pipeline in general (creation and management)
  - Number and types of objects lifecycle (render passes, framebuffers, ...)
- Makes Vulkan more introduction-friendly
  - More manageable for newcomers
  - Some features offset the complexity to the driver
  - New features are more opaque to performance tweaking
- Performance is comparable
  - No obvious or easy gains (desktop); mobile is WIP
- Complexity of Vulkan wrapper is reduced
- LOC is reduced, considerably in some cases (due to cherry-picking)
- Caveat: performance may be compromised with new features
  - And an opportunity to show how to write performant Vulkan is lost

# Thank you!

- Questions?