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

# A Data-Driven Approach to Modularity in Vulkan Renderers

Kerem Tuncer, University of Vienna

# Introduction

- JS developer for 8 years
- Doing Master's in University of Vienna
- Started learning Vulkan in 2024
- Vulkan is huge, with tons of power… But how?
- API and compatibility evolved a lot since then

- Just my opinions

# The 4 Hard Problems of Vulkan Renderers

**1. Control**

Who runs when?

Who owns which resource?

**2. Scale**

How do we submit ten thousand objects efficiently?

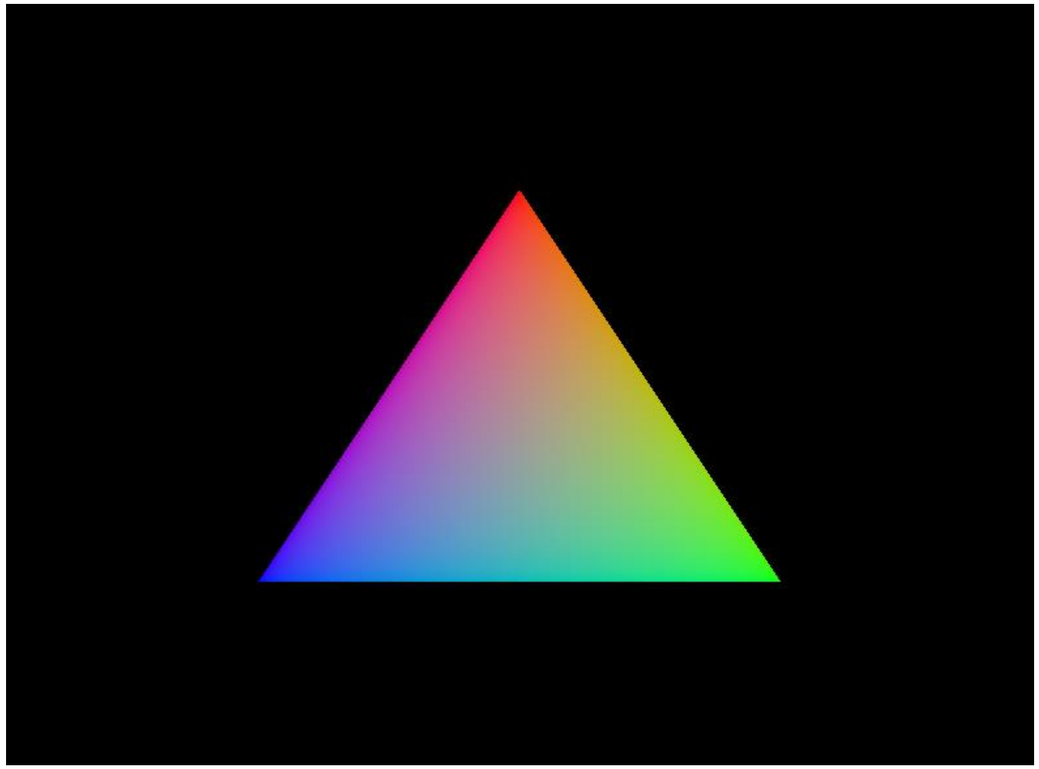
**3. Binding**

How does data actually reach shaders without killing the CPU?

**4. Trust**

How do CPU and GPU agree on memory layout without human error?

# Stage 1 // Hello World - Vulkan Triangle

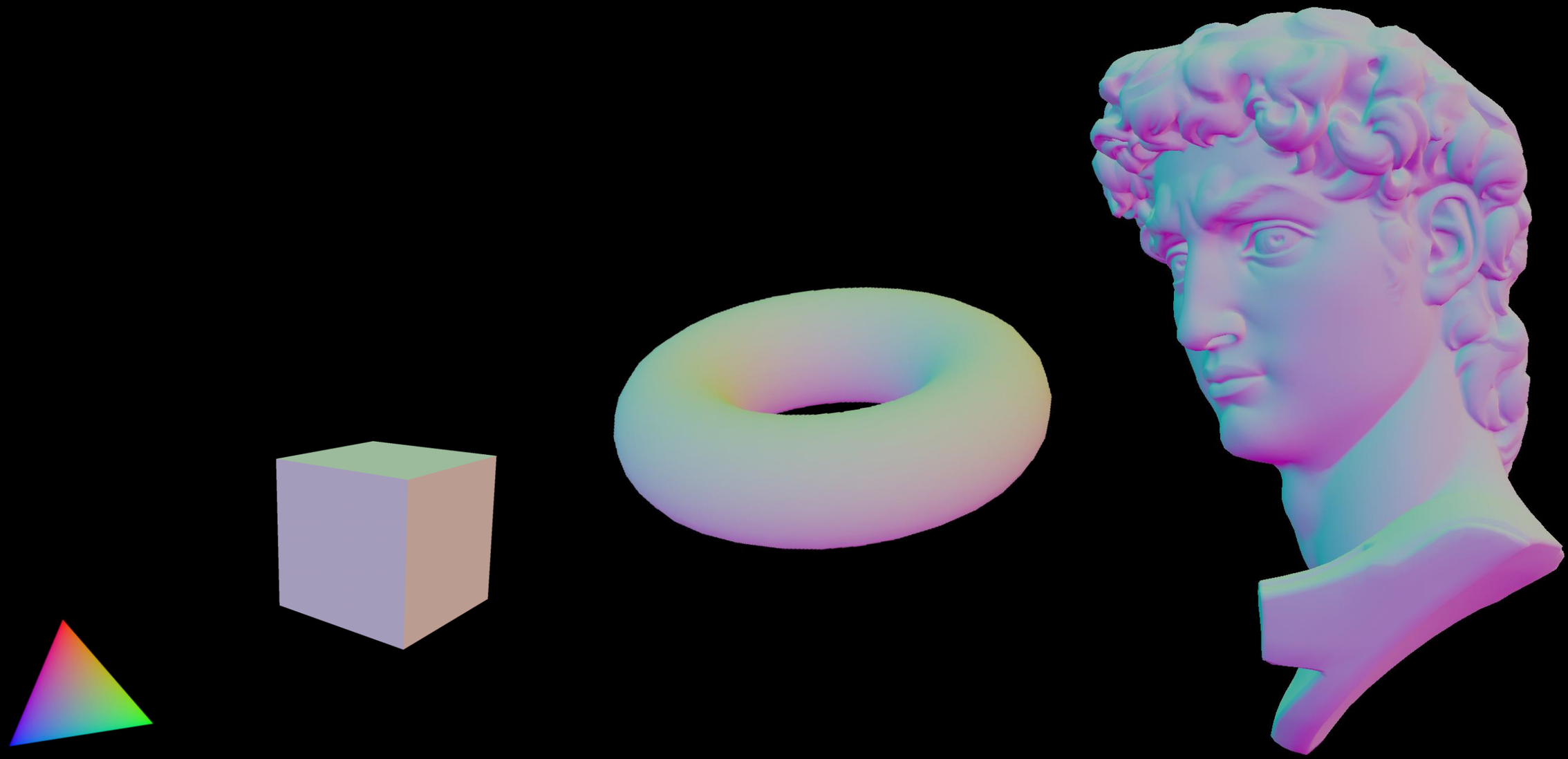


- One Vertex Buffer (hardcoded)
- One Pipeline
- One Draw Call
- Architecture is implicit and trivial at this stage.

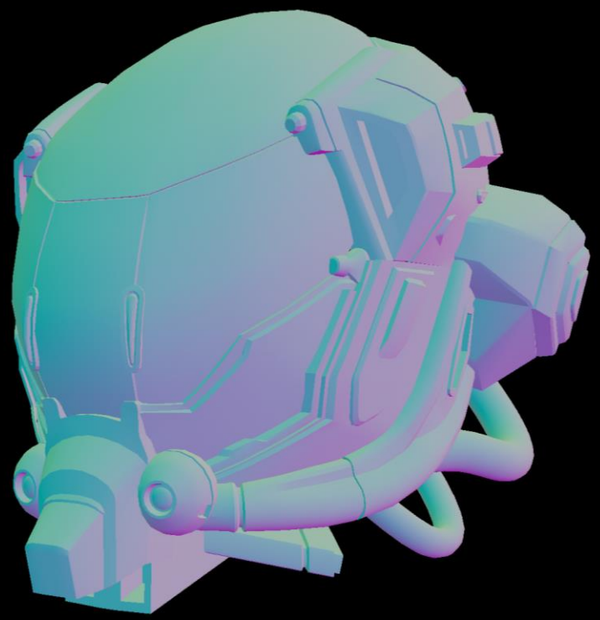
main.cpp

```
void Render() {
  // Setup: 1,500 lines of init code
  // (Instance, Device, Swapchain...)

  vkCmdBeginRenderPass(cmd, &rpInfo);
  vkCmdBindPipeline(cmd, ..., pipeline);
  vkCmdBindVertexBuffers(cmd, 0, 1, &vbo);

  // Hardcoded triangle vertices in buffer
  vkCmdDraw(cmd, 3, 1, 0, 0);

  vkCmdEndRenderPass(cmd);
}
```

# Stage 1.5 // From Triangle to Mesh - More Data -> Same Architecture



# Stage 2 // Introducing Data - First Texture Binding



# Stage 2 // Data Dependencies - Descriptor Sets Appear

**The First Real Jump**

- Descriptor set layouts
- Descriptor pools
- Descriptor sets
- Samplers + Image views
- Layout transitions
- Resources must be explicitly described

main.cpp

```
// Uniform buffer (light pos,color,intensity)
VkBuffer lightingUBO =
createBuffer(sizeof(LightData));
memcpy(lightingUBO.data, &lightData, ...);

// Create descriptor set (binding 0: texture,
binding 1: UBO)
VkDescriptorSet set0;
vkUpdateDescriptorSets(..., texView, lightingUBO);

void Render() {
  vkCmdBindPipeline(cmd, ..., pipeline);
  vkCmdBindDescriptorSets(cmd, ..., set0);
  vkCmdDraw(cmd, 6, 1, 0, 0);
}
```

# Stage 3 // Implicit State - The First Real Problem

- Multiple rendering passes (Shadow, Lighting, Post)
- Each pass owns GPU resources
- Implicit dependencies: Who owns shadowMap?
- Unclear lifetime: When is colorTex valid?
- Missing barriers: Who inserts them?

```
// Multiple pass rendering
struct ShadowPass {
  VkImage shadowMap;  // Who allocates?
  void Render(VkCommandBuffer cmd);
};

struct LightingPass {
  VkImage* shadowMap; // Borrowed? Owned?
  VkImage colorOutput;
  void Render(VkCommandBuffer cmd);
};

void RenderFrame() {
  shadowPass.Render(cmd);
  // ⚠ Missing barrier?
  lightingPass.Render(cmd);
  // ⚠ Who manages lifetime?
}
```

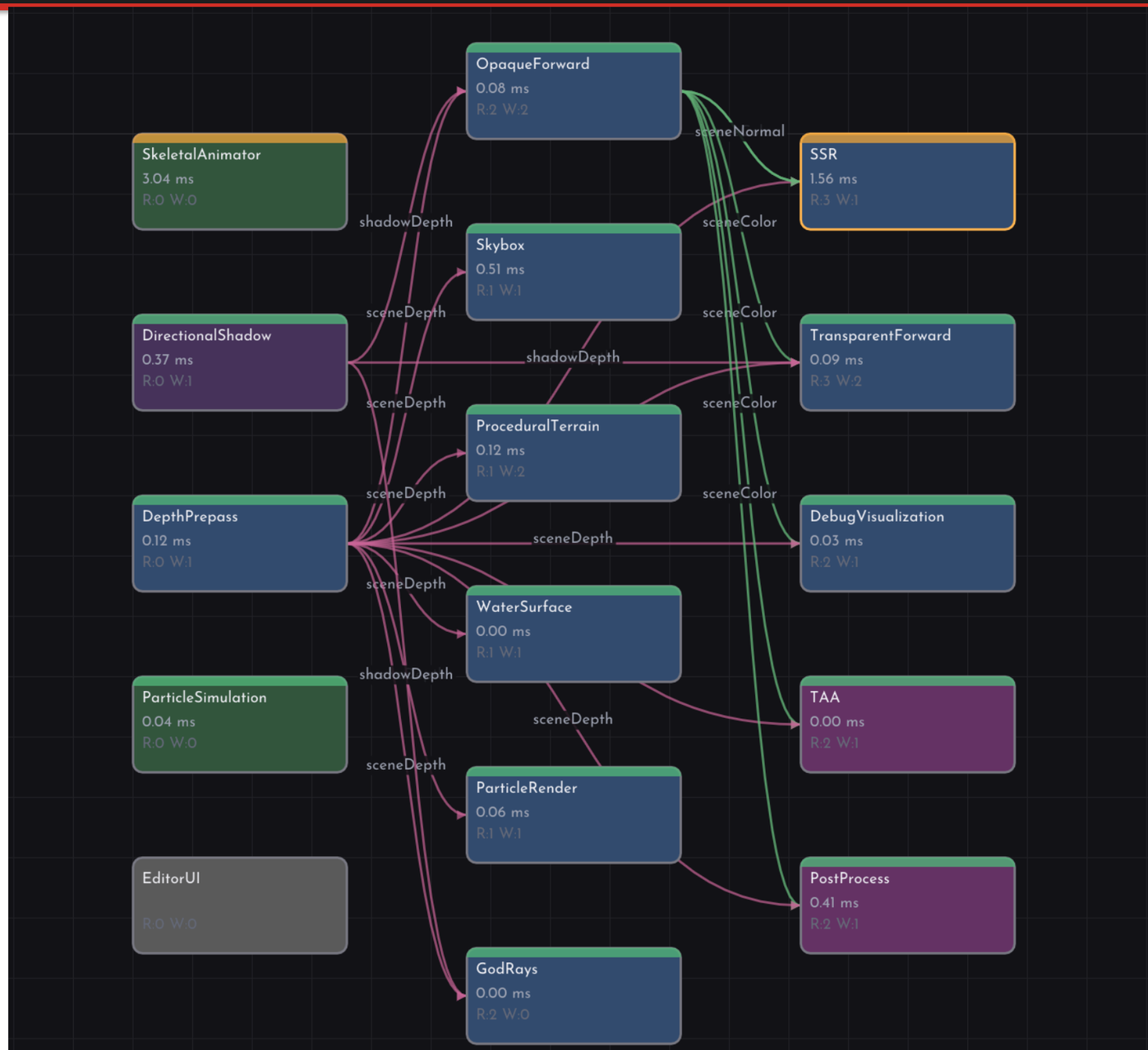
# Stage 4 - Solution // Frame Graph - Data-Driven Modularity

**What FrameGraph solves:**

- Execution order
- Resource lifetime
- Barriers & synchronization

**What it does NOT solve:**

- Draw submission cost
- Descriptor binding model
- Per-object CPU overhead



# Stage 4.5 - Solution // Module Contracts - Declare Inputs/Outputs

- Code doesn't call code
- Modules declare Data needs
- Graph computes execution order
- Graph inserts barriers automatically
- Resource lifetimes managed
- **Dynamic Rendering** - No VkRenderPass!

modules.json

```
// Shadow Module declares what it writes
{
  "name": "DirectionalShadow",
  "type": "DirectionalShadow",
  "enabled": true,
  "config": {
    "contract": {
      "pass": { "name": "DirectionalShadow" },
      "resources": [{
        "semantic": "shadowDepth",
        "type": "attachment",
        "usage": "write/clear"
      }]
    }
  }
}
```

## Stage 5 - Problem // CPU Bottleneck - 10,000 Objects Breaking Performance

# Stage 5 - Problem // 10k Draw Calls Kill Performance

**The CPU Bottleneck**

- 10,000 Objects = 10,000 Draw Calls
- Cache misses on Object list
- Driver overhead (validation)
- Descriptor set bindings
- CPU-side culling

```
void RenderScene() {
  // The Loop of Death
  for (int i = 0; i < 10000; i++) {
    auto& obj = scene.objects[i];

    // CPU-side frustum culling
    if (!isInFrustum(obj.bounds)) {
       continue;
    }

    // Bind material (descriptor sets)
    vkCmdBindDescriptorSets(..., obj.material);

    // Draw (expensive validation!)
    vkCmdDrawIndexed(cmd, obj.mesh);
  }
  // 10,000 iterations x 3 API calls = 30k calls!
}
```

# Stage 6 - Solution // Indirect Drawing - One Call for Everything

**Kill the Loop: GPU-Driven**

- Upload all DrawData to GPU (once)
- Compute shader does culling
- Writes VkDrawIndexedIndirectCommand
- 1 Indirect Draw per pass for all objects
- CPU: 0.1ms, GPU: busy

```
// Upload object data (CPU -> GPU)
DrawData drawData[10000];
memcpy(gpuBuffer, drawData, sizeof(drawData));

void RenderScene() {
  // GPU culling (compute shader)
  vkCmdDispatch(cmd, cullShader, ...);

  // Barrier: compute write -> draw read
  vkCmdPipelineBarrier(...);

  // ONE draw call per pass for 10,000 objects!
  vkCmdDrawIndexedIndirectCount(
      cmd, indirectBuffer, countBuffer, 10000
  );
}
```

# Stage 7 - Problem // GPU-Driven… But Still Slot-Bound - Pool/Layout/Update Complexity

**The Hidden Cost: Descriptors**

Slot-based binding (Set 0, 1, 2...)

Pool allocation/deallocation

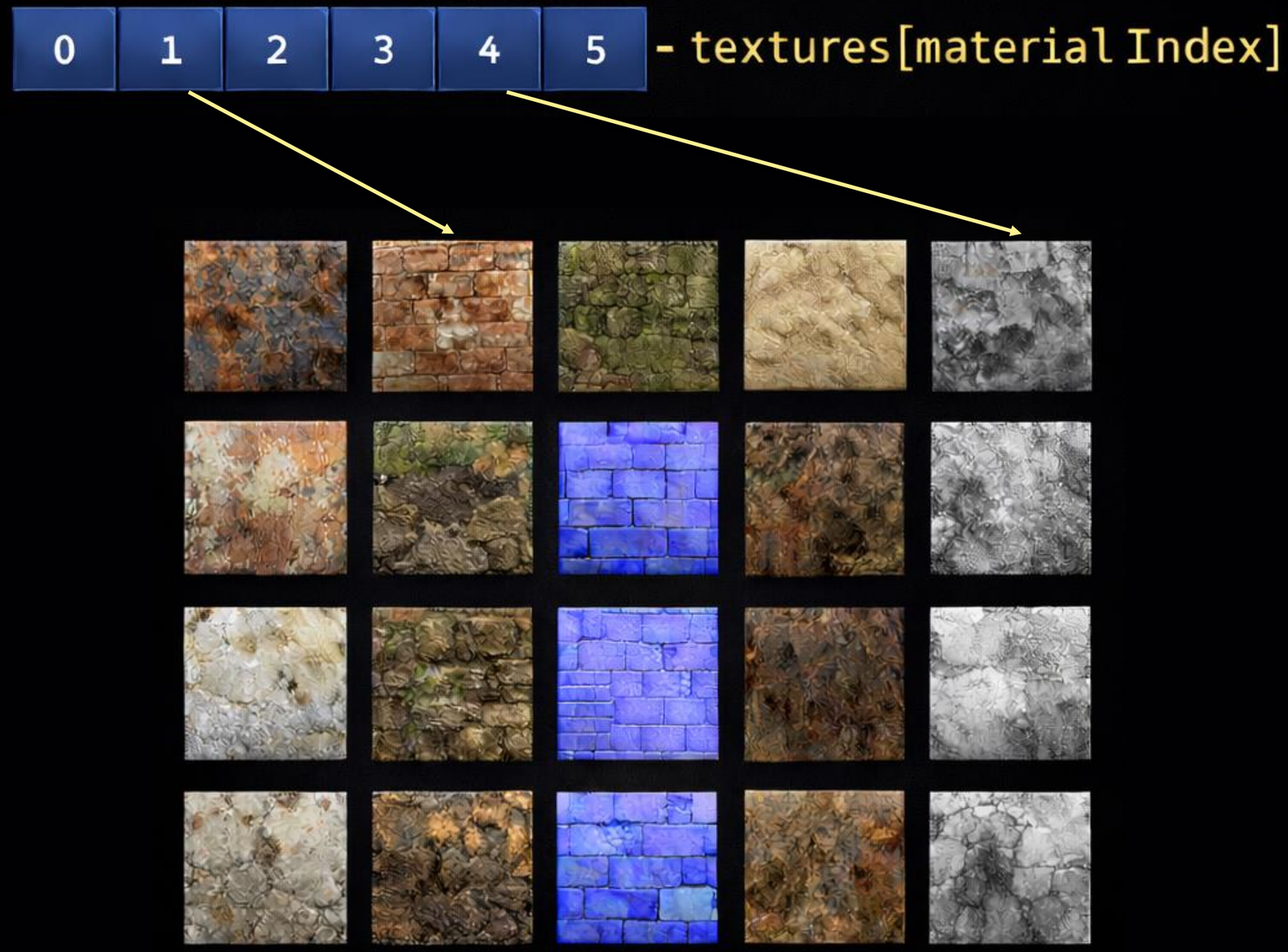
Update commands (VkWriteDescriptorSet)

Layout compatibility issues

Per-frame updates expensive

```
// Even with GPU-driven, still binding:
void RenderScene() {
  // Bind global scene data
  vkCmdBindDescriptorSets(cmd, ..., set0);

  // Bind material textures
  vkCmdBindDescriptorSets(cmd, ..., set1);

  // Bind shadow map
  vkCmdBindDescriptorSets(cmd, ..., set2);

  vkCmdDrawIndexedIndirect(...);
}

// 50+ lines to create these sets...
// Pool allocation, layout creation, updates...
```

# Stage 7 - Solution Part 1 // Bindless Textures via Descriptor Indexing



# Stage 7 - Solution Part 1 // Dynamic Indexing - texture[materialIdx]

**VK_EXT_descriptor_indexing**

- (Vulkan 1.2 core)
- Runtime descriptor arrays
- Dynamic indexing in shader
- Bindless texture access
- One descriptor set for all textures
- Bind once per frame, not per draw

shader.frag

```
// Slang with descriptor indexing
[[vk::binding(0, 0)]]
Sampler2D textures[16384];

float4 fragmentMain(float2 uv : TEXCOORD) : SV_Target {
  // Dynamic indexing!
  uint texIdx = material.baseColorIndex;
  float4 color = textures[texIdx].Sample(uv);
  return color;
}

// C++ side:
// Bind once per frame
vkCmdBindDescriptorSets(cmd, ..., textureArraySet);

// Problem: Still need descriptor updates
// when adding new textures
```

# Stage 8 - Full Solution // BDA + Descriptor Indexing - Zero Per-Draw Bindings

**BDA:** VK_KHR_buffer_device_address

- 64-bit GPU pointers for DrawData, Materials
- Pass pointers via push constants

**Result:** Eliminate per-object binding churn

OpaqueForward.cpp

```
// BDA, Obtain addresses: vkGetBufferDeviceAddress()
ShaderStructWriter::create("BDAPushConstants")
  .set("drawDataAddr", drawDataDeviceAddr)
  .set("materialAddr", materialDeviceAddr)
  .set("sceneGlobalsAddr", sceneGlobalsAddr)
  .push(cmd, layout, stages);

// Bindless: One descriptor set for all textures
vkCmdBindDescriptorSets(cmd, ..., bindlessSet);

// Indirect draw (GPU-driven)
vkCmdDrawIndexedIndirectCount(cmd, indirectBuf, ...);

// Shader accesses data via pointers:
// DrawData* drawData = (DrawData*)drawDataAddr;
// Material mat = materials[drawData.materialIndex];
// vec4 color = textures[mat.baseColorIdx].Sample(uv);
```

# Stage 8.5 // Data Flow - Model to Screen

**1. Initialization (Once)**

**Mesh data -> MegaBuffer (GPU):** All vertices/indices in one place

**Textures -> VkImage**, register to textures[1024] array

**Materials -> BDA Buffer** (persistently mapped)

**2. Per-Frame (60 times/sec)**

**CPU:** Update 10,000 DrawData -> single memcpy to mapped buffer

**GPU Compute:** Cull 10k objects -> 5k visible (frustum culling)

**GPU Graphics:** 1 indirect draw renders all visible objects

**3. Shader Execution (Per-pixel)**

**Vertex:** BDA read DrawData[instanceID], transform vertex

**Fragment:** BDA read Material[drawID], sample textures[materialIndex]

**Result:** Fully shaded pixel with PBR lighting

**Problem:** BDA requires exact struct layout

(1 byte off = crash)

**Solution:**

- Python script parses Slang shader structs
- Generates C++ with alignas(), static_assert()
- Calculates memory layout

  (BDAMemoryLayout.hpp)
- Zero manual synchronization needed

```
// Slang shader (single source of truth)
struct DrawData {
  float4x4 modelMatrix;  // 64 bytes
  uint32_t materialIndex;
  uint32_t boneOffset;
  uint32_t boneCount;
  ...
};

// Python script generates C++ struct:
struct DrawData {
  alignas(16) glm::mat4 modelMatrix;
  alignas(4) uint32_t materialIndex;
  alignas(4) uint32_t boneOffset;
  ...
};

// Compile-time validation:
static_assert(sizeof(DrawData) == 128);
```

# Stage 10 // Build Pipeline - Slang -> Reflection -> Runtime

**Reflection Pipeline**

**Edit:** shader.slang (single source)

**Compile:** slangc -reflection-json

**Run:** embed_reflection_data.py

**Output:** ReflectionData_Generated.hpp

**Runtime:** ShaderStructWriter (type-safe)

Runtime Usage (ShaderStructWriter)

```
// Zero C++ struct definition needed!
ShaderStructWriter::create("TAAPushConstants")
  .set("invViewProj", myMatrix)
  .set("screenWidth", extent.width)
  // uint32 → float32 (auto)
  .set("textureIndex", texIndex)
  // uint32 → uint32 (preserved)
  .push(cmd, layout, stages);

// Internally:
// 1. Lookup field in reflection cache
// 2. Read type: "float32" or "uint32"
// 3. Auto-convert based on shader type
// 4. memcpy to correct offset
// 5. vkCmdPushConstants()
```

# Use Case 1 // Adding Water Surface - Zero Touch to Existing Modules

**Step 1:** Create WaterSurface.cpp/.hpp

**Step 2:** Inherit from RenderModule

**Step 3:** Declare resource dependencies

**Step 4:** Add to modules.json

**Step 5:** Done! No other modules touched.

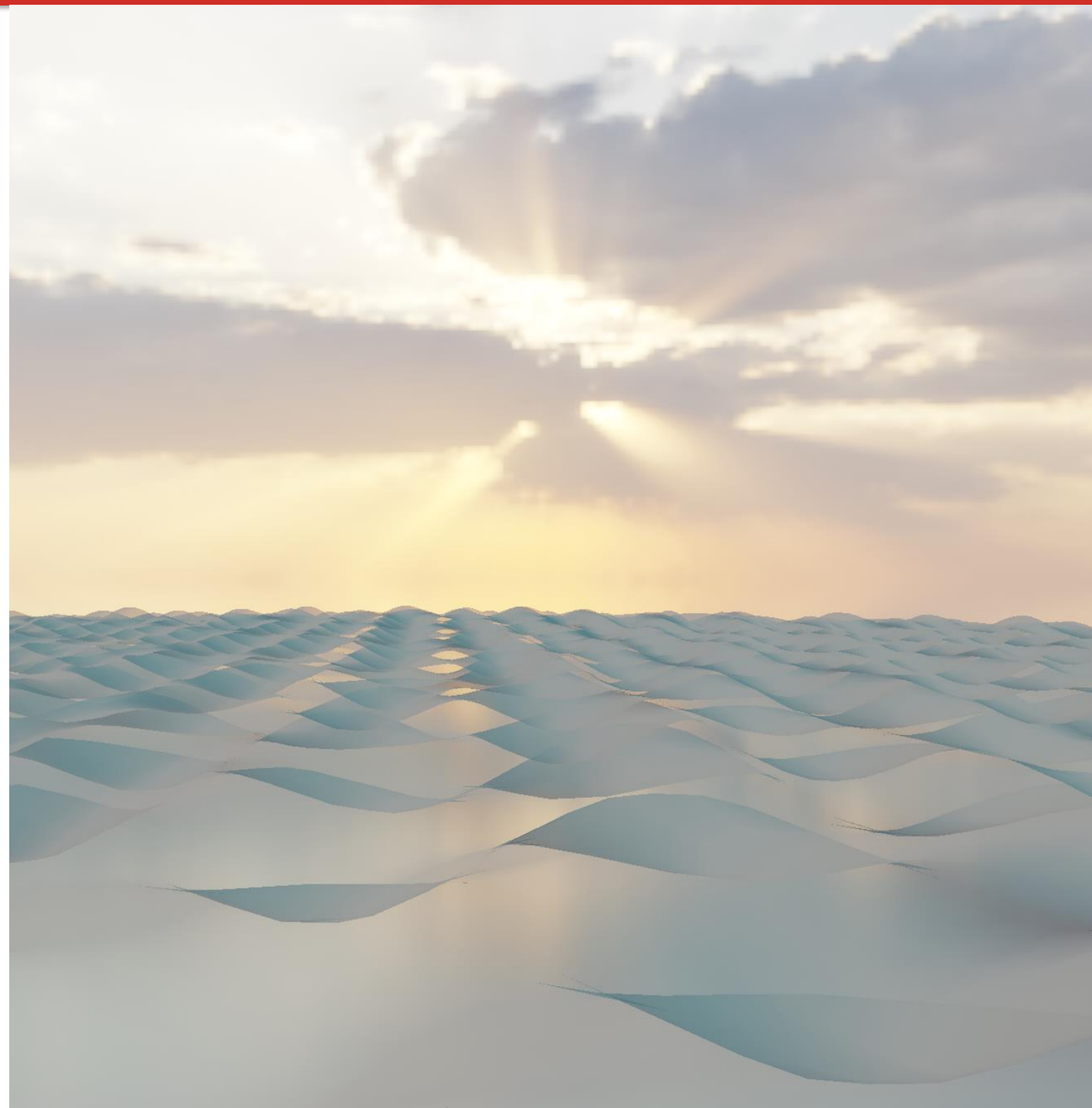
**Why Zero Touch?**

FrameGraph handles ordering

Barriers inserted automatically

Self-registering (REGISTER_MODULE)

JSON-driven configurationon



# Use Case 2 // Adding Post Processing - Compute Shader Pipeline

**Compute + Graphics Mix**

Create compute shader (reads: sceneColor)

**Apply effects:** Bloom, tone mapping, color grading

**Key:** FrameGraph handles compute -> graphics transitions

**Auto barriers:** COLOR_WRITE -> COMPUTE_READ

Scheduled after forward rendering automatically

Zero manual pipeline barriers needed

# Use Case 3 // Adding SSR - Screen Space Reflections

**Step 1:** Create SSR compute shader

**Step 2:** Declare reads:

sceneColor, sceneDepth

**Step 3:** Declare writes:

ssrOutput

**Step 4:**

Add to modules.json

**Step 5:**

FrameGraph schedules after forward pass

# Key Takeaways

## Architectural Principles

**1. Data-Driven Modularity:**

FrameGraph solves ordering & synchronization (orthogonal to GPU-driven)

**2. GPU-Driven Scale Move work to GPU:**

culling, indirect draws -> submission scale

**3. Zero Per-Object Bindings:**

Descriptor Indexing (textures) + BDA (buffers) = minimal CPU overhead

## Core Extensions

**VK_KHR_dynamic_rendering:**

No VkRenderPass, inline attachments

**VK_EXT_descriptor_indexing:** textures[1024], bind once per frame

**VK_KHR_buffer_device_address:**

64-bit pointers in push constants

# Thank You!

## Q&A

**Contact:** keremtuncer@yahoo.com
https://github.com/keremtuncer