SIGGRAPH 2026
Los Angeles 19–23 JUL

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

Reducing graphics API complexity:

A clean slate API design for modern GPUs

Sebastian Aaltonen - 30 years of 3d graphics experience:

Graphics APIs:

OpenGL 1.1 - 4.5

DirectX 5.0 - 12.X

Vulkan 1.0 - 1.4

Metal 1.0 - 4.0 [1]

WebGL

WebGPU

Career – Principal Engineer at:

hypehype

UBISOFT

unity

Console graphics APIs:

N-GAGE NOKIA

Nintendo

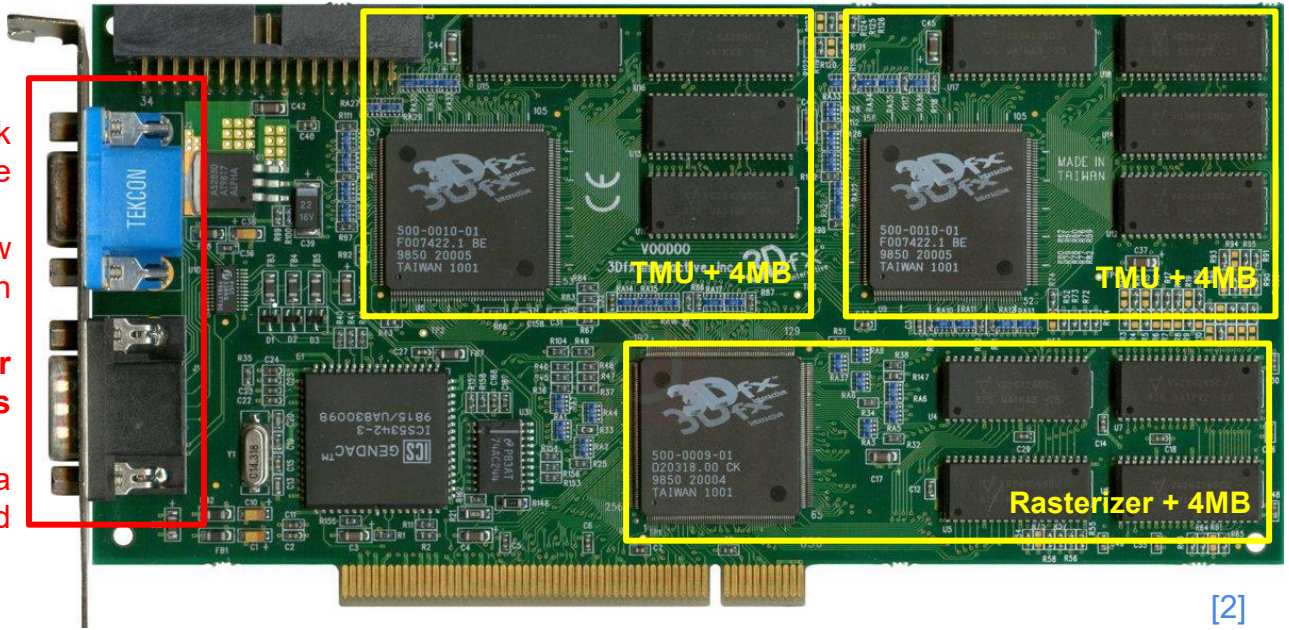
XBOX

NVIDIA CUDA

iOS

MacOS

GPU history: 3dFX Voodoo popularizes 3d hardware



Loopback cable

No window composition

No render target textures

Required a separate 2d card

No shaders

No geometry processing

Simple blending: 2x textures + vertex color

Separate memories

[2]

DirectX and OpenGL were designed for these GPUs

30 YEARS AGO

GPU history: The first programmable GPUs

VLIW, vec4, vec4+1, narrow SIMD → **abstraction is required**

Framework-style 1:1 element transform

- System provides inputs: vertices, varyings, uniforms
- User provides **element transform instructions** (8 → 64 → 512 → 65536)
- Read-only texture sampler
- No programmable memory load or store → No need for generic R&W caches
- **Huge** influence on shader language design → shader framework has **16** entry points today!

HLSL and GLSL were designed for these GPUs

20 YEARS AGO

GPU history: Unified shaders + compute

CUDA

SIMT / SIMD
R&W buffers
R&W caches

Microsoft® DirectX® 11

"The Buffer Zoo"

Sampler →
Buffer (SRV) – homogeneous array
RWBuffer (UAV) – TypedUAVLoad :(
ByteAddressBuffer (SRV) – 4-byte align:(
RWByteAddressBuffer (UAV)
StructuredBuffer (SRV) – AoSoA :(
RWStructuredBuffer (UAV)
AppendStructuredBuffer (UAV)
ConsumeStructuredBuffer (UAV)
ConstantBuffer – 16-64KB, 16-byte align arrays :(
VertexBuffer – no dynamic indexing
IndexBuffer

R&W caches/coherency add a lot of complexity

15 YEARS AGO

GPU history: “Modern” Graphics APIs

Pressure: Consoles with MT rendering + fast draw calls…

- Designed for **12-16** year old hardware
  - GTX 580, AMD GCN1, Intel HD 4200…
  - Adreno 400, Mali Midgard, PVR Series 6
- Custom (limited) binding models
  - Bindless not universally supported
- Cache coherency & barriers & layouts
- 256MB CPU aperture (driver)

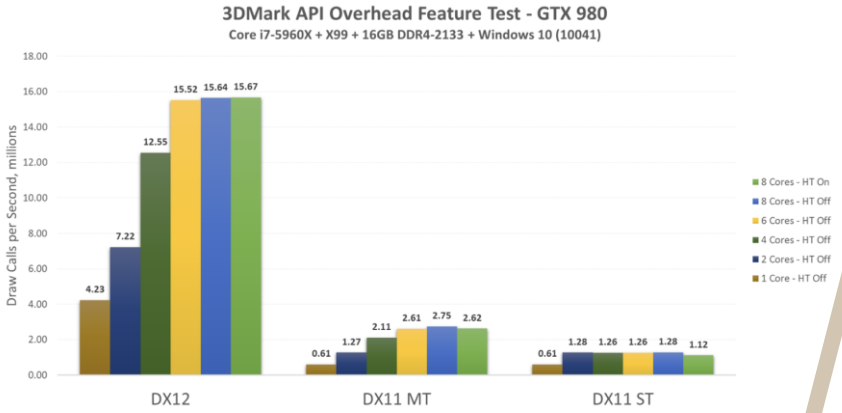
DX12, Vulkan and Metal (→WebGPU) designed for 14-16 year old HW

10 YEARS AGO

Modern APIs: Promise vs Reality

NEWS

Tested: DirectX 12's potential performance leap is insane



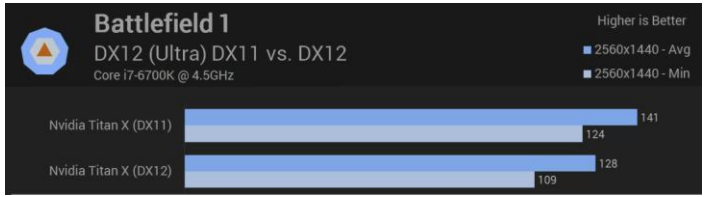
[3]

3DMARK®
The Gamer's Benchmark

ARK: Survival Evolved – DX12 Renderer Delayed, Performs Worse Than DX11

[4]

JOHN2 | 11 YEARS AGO | ARK: SURVIVAL EVOLVED



[5]

On Unity 2018.2.17, still the same poor performance on Android Vulkan. Switching from GL ES 3.1 to Vulkan cuts the fps to less than half!

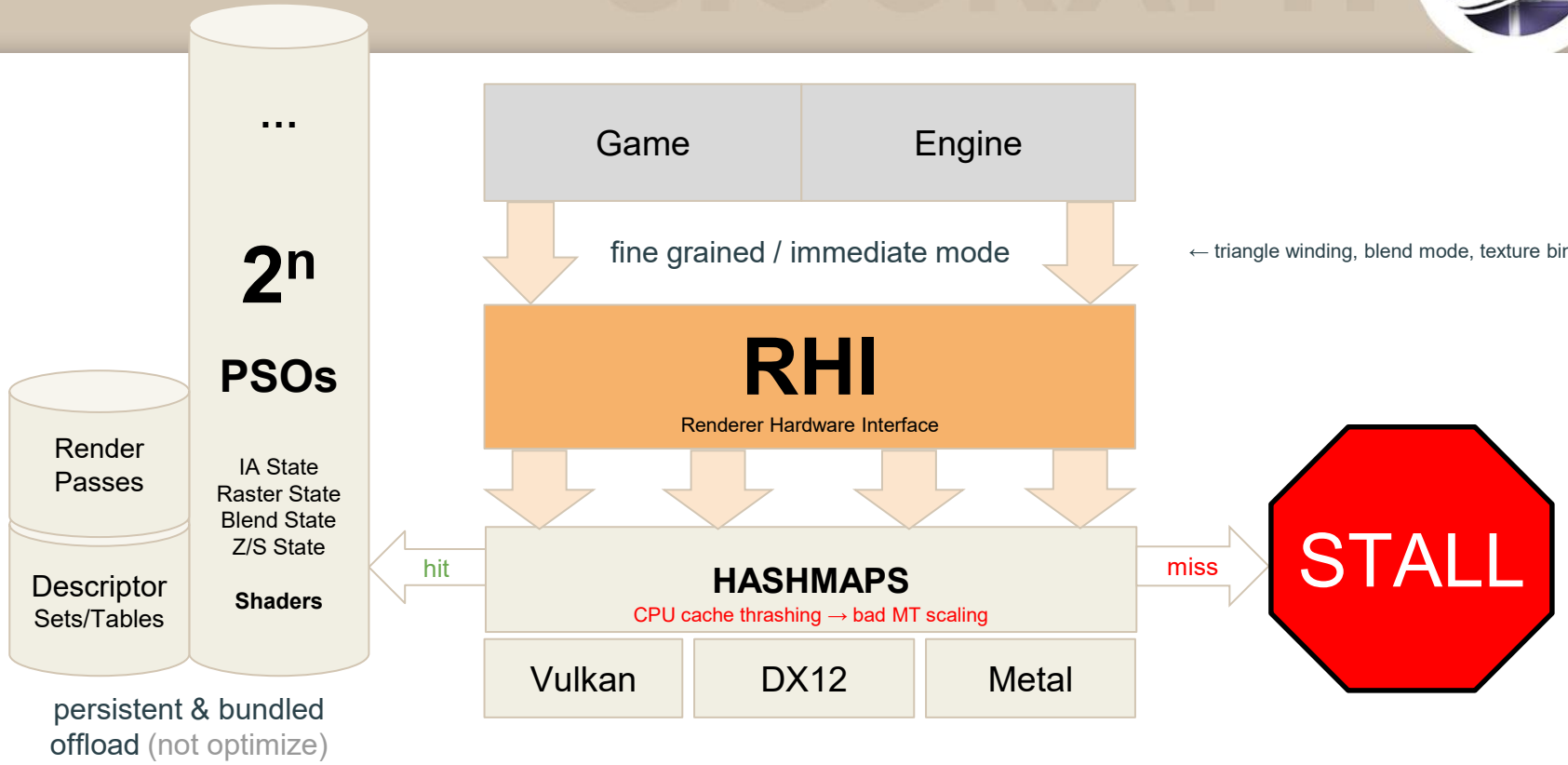
[6]

Unity Founder: DirectX 12 API Alone Doesn't Give A Significant Performance Boost

[7]

David Helgason also talks about the use of Enlighten in Unity.

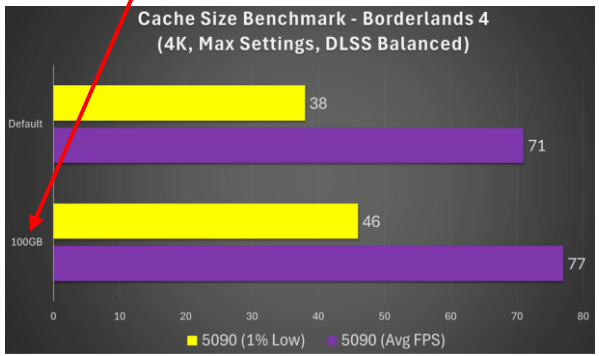
RHI vs API design mismatch



PSO explosion: Instead of fixing the root cause…

~100GB PSO Cache

100 TB+ Cloud Server



[8]

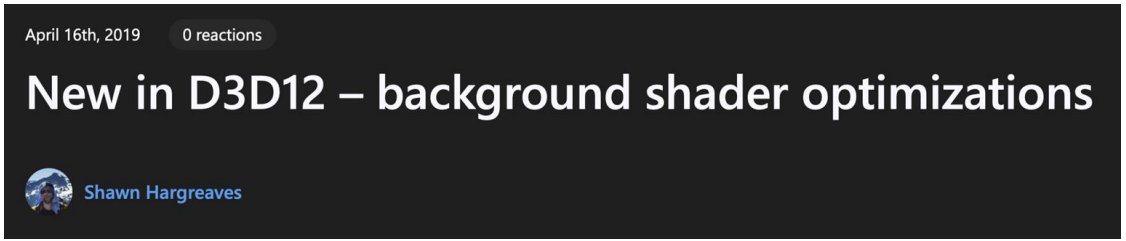
Driver update → 30 minutes waiting for shaders to recompile :(

Forza Horizon 6 boots up in just 4 seconds instead of 90 with new Advanced Shader Delivery tech and AMD GPUs — Microsoft claims 95% reduction in gaming load times

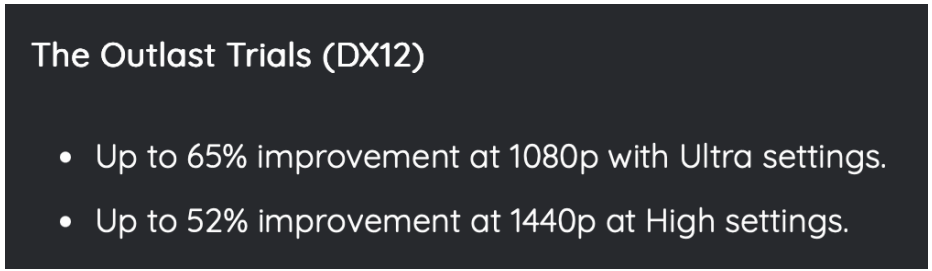
News By Hassam Nasir published May 17, 2026

Only available with games downloaded via the Xbox PC app. [9]

More developer responsibility, simpler driver?



[10]



[11]

[12]

GPU history: Hardware convergence

Hardware converged rapidly in 10 years

- 100% SIMD: SIMD32 (AMD, Nvidia, Apple), SIMD16+ (ARM, Intel, Qualcomm)
- Coherent LLC: No need to flush to VRAM
- No texture layouts: Samplers & display support DCC
- Bindless texture samplers
- Virtual memory and 64-bit GPU pointers
- Full CPU→GPU access: PCIE-Rebar + UMA → persistently mapped pointers
- Dynamic register alloc (NV/AMD intrinsics, Apple Dynamic Caching) and SER

Modern workloads

- GPU-driven rendering, V-buffer, ray-tracing
- C/C++ pointer based shading languages: Metal and CUDA (+tensors)
- Dynamic memory access: Uniform/vertex buffers → fast raw loads

DirectX® 12

- Dynamic Resources
- ByteBuffer.Load<T>
- 16-bit Types
- Wave Intrinsics

Vulkan®

- Dynamic Rendering
- Buffer Device Address
- Descriptor Buffer/Heap
- Unified Image Layouts

…

TODAY

→ Don’t need a complex API abstraction anymore








The Premier Conference & Exhibition on Computer Graphics & Interactive Techniques

SIGGRAPH 2026
Los Angeles 19–23 JUL

The Clean API

Memory Management

vkCmdPushDataEXT [13]

UMA or **PCIE-ReBAR** = Persistent CPU→GPU pointers

```
void* gpuMalloc(size_t bytes, MEMORY memory = MEMORY_DEFAULT);
void* gpuMalloc(size_t bytes, size_t align, MEMORY memory = MEMORY_DEFAULT);
void* gpuTempMalloc(GpuCommandBuffer cb, size_t bytes, size_t align);
void gpuMemCpy(GpuCommandBuffer cb, void* dest, void* src);
void gpuFree(void *ptr);
```

- Returns either a shared virtual memory (SVM) pointer or CPU+GPU pointer pair
  - CPU+GPU pointer pair is supported by all modern GPUs (already used by DX12, Vulkan and Metal)
  - Single SVM pointer is cleaner, but not supported by all GPUs yet. Using SVM in my slides for simplicity.
- Default memory visible to both CPU and GPU: Easy to write directly from C/C++ code to GPU memory.
- Command buffer temp alloc similar to vkCmdPushDataEXT and Metal setBytes

No Binding Layout

SYNOPSIS top

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *_Nullable restrict attr,
                   typeof(void *(void * Nullable)) *start_routine,
                   void *_Nullable restrict arg);
```

OS doesn't care about data layout
Graphics API should not either!

```
GpuPipeline gpuCreateComputePipeline(ByteSpan computeIR);
GpuPipeline gpuCreateGraphicsPipeline(ByteSpan vertexIR, ByteSpan pixelIR, GpuRasterDesc desc);
GpuPipeline gpuCreateGraphicsMeshPipeline(ByteSpan meshIR, ByteSpan pixelIR, GpuRasterDesc desc);
void gpuFreePipeline(GpuPipeline pipeline);
```

No root/push argument APIs. No descriptor set/table APIs. No buffer/texture binding APIs

Root Data is a 64-bit GPU Pointer

Vulkan BufferDeviceAddress [14]

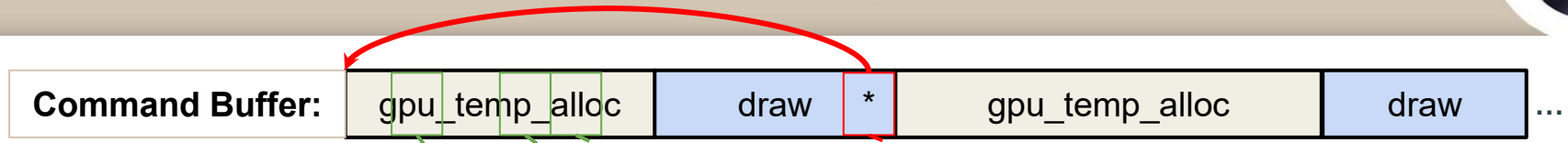
```
// Common header...
struct alignas(16) Data
{
    // Uniform data
    float16x4 light_hdr;
    int16x2 offset;
    uint8x4 color;

    // Pointers to in/out data arrays
    const uint32* input;
    uint32* output;
};
```

= 2x 128-bit wide loads

```
// CPU code...
gpuSetPipeline(commandBuffer, computePipeline);

Data* data = gpuTempMalloc<Data>(commandBuffer);
data->light_hdr = {1.0f, 0.0f, 0.0f, 1.0f};
data->offset = {16, 0};
data->color = { 255, 255, 255, 0}
data->input = input;
data->output = output;

gpuDispatch(commandBuffer, data, { 128, 1, 1 });
```

```
// GPU kernel...
[groupsize = (64, 1, 1)]
void main(uint32x3 threadId : SV_ThreadID, const Data* data)
{
    uint32 value = data->input[threadId.x + data->offset.x];
    data->output[threadId.x] = value;
}
```

Root Optimizations and Uniformity Analysis



Uniformity analysis

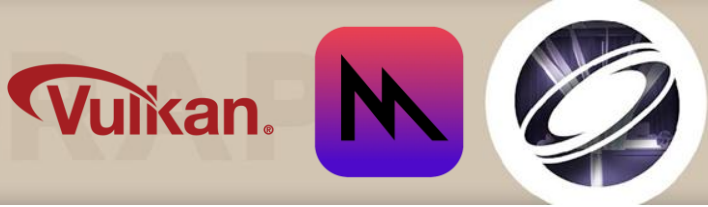
- Preload to uniforms
- Scalarization
- Wave broadcast
- Arm Mali pilot shaders

```
// GPU kernel...
[groupsize = (64, 1, 1)]
void main(uint32x3 threadId : SV_ThreadID, const Data* data)
{
    uint32 value = data->input[threadId.x + data->offset.x];
    data->output[threadId.x] = value;
}
```

Data access

- root_ptr + struct field offset
- A virtual memory pointer
  - GPU memory, CPU memory, command buffer…

Static (specialization) Constants



```
// Common header...
struct alignas(16) Constants
{
    int32 qualityLevel;
    float16* blueNoiseLUT;
};
```

```
// CPU code...
Constants constants { .qualityLevel = 2, blueNoiseLUT = blueNoiseLUT.gpu };
GpuPipeline computePipeline = gpuCreateComputePipeline(shaderIR, &constants);
```

```
// GPU kernel...
[groupsize = (8, 8, 1)]
void main(uint32x3 threadId : SV_ThreadID, const Data* data, const Constants constants)
{
    float16 noise = constants.blueNoiseLUT[threadID.x & 0xff];
    if (constants.qualityLevel == 3)
    {
        // Dead code eliminated
    }
}
```

Flexible “bindings”. Can bake pointers. API bloat is gone

(No) Texture Bindings

Microsoft® [15] DirectX® Shader Model 6.6

Two competing binding models (modern HW):

- 32-bit index to descriptor heap → TMU caches descriptors
- 256-bit descriptors in memory → loaded to scalar registers

} **UNIFIED API**

ptr64* (uniform) + u32 (lane?)

```
// Pixel shader...
const Texture textureHeap[];

PixelOut main(const VertexIn &vertex, const DataPixel* data)
{
    Texture texture = textureHeap[data->textureIndex];
    Sampler sampler = {.minFilter = LINEAR, .magFilter = LINEAR};

    float32x4 color = sample(texture, sampler, vertex.uv);
    return { .color = color };
}
```

Descriptor cache

Scalar registers

Descriptor heap (256-bit array)

```
[0] metal5_albedo
[1] metal5_normal
[2] metal5_pbr
[3] plastic7_albedo
[4] plastic7_normal
[5] plastic7_pbr
[6] icon_banner
[7] ui_final_v3_fix2
[8] g_buffer_A
[9] g_buffer_B
[10] g_buffer_C
[11] particle_atlasA
…
```

Texture Descriptors

VK_EXT_descriptor_heap
[16]

```
GpuTextureDescriptor *textureHeap = gpuMalloc<GpuTextureDescriptor>(65536);

textureHeap[0] = gpuTextureViewDescriptor(texture, {
    .ptr = catImage.gpuPtr,
    .dimensions = catImage.dimensions,
    .format = FORMAT_RGBA8_UNORM,
    .usage = USAGE_SAMPLED
});

gpuSetActiveTextureHeapPtr(commandBuffer, textureHeap);
```

Minimal API. Descriptors are pure memory. Can memcpy & compute write

Texture Data Upload

```
GpuTextureSizeAlign sizeAlign = gpuTextureSizeAlign({
    .dimensions = catImage.dimensions,
    .format = FORMAT_RGBA8_UNORM,
    .usage = USAGE_SAMPLED
});

void *texturePtr = gpuMalloc(sizeAlign.size, sizeAlign.align, MEMORY_GPU);

void *uploadPtr = pngLoader.uploadToGPU(...);

gpuCopyToTexture(uploadCommandBuffer, texturePtr, uploadPtr);
```

Texture data upload still needs to be abstracted due to HW morton order and DCC

Draw Calls: Vertex Shader

```
// Common header...
struct Vertex
{
    float32x4 position;
    uint32 tangentFramePacked;
    float16x2 uv;
};

struct alignas(16) Data
{
    float32x4x4 matrixMVP;
    const Vertex *vertices;
    uint32 textureBaseIndex;
};
```

```
// CPU code...
gpuSetPipeline(commandBuffer, graphicsPipeline);

Data* data = gpuTempMalloc<Data>(commandBuffer);
data->matrixMVP = camera.viewProjection * modelMatrix;
data->vertices = mesh.vertices;
data->textureBaseIndex = material.textureBaseIndex;

gpuDrawIndexed(commandBuffer, data, mesh.indices, mesh.indexCount);
```

```
VertexOut main(uint32 vertexIndex : SV_VertexID, const Data* data)
{
    Vertex vertex = data->vertices[vertexIndex];
    float32x4 position = data->matrixMVP * vertex.position;
    return { .position = position, .tangent = tangent, .uv = vertex.uv };
}
```

Vertices are just memory. Dynamic vertex layout/stride doesn't require multiple PSOs

Draw Calls: Pixel Shader

```
// Common header...
struct Vertex
{
    float32x4 position;
    uint32 tangentFramePacked;
    float16x2 uv;
};

struct alignas(16) Data
{
    float32x4x4 matrixMVP;
    const Vertex *vertices;
    uint32 textureBaseIndex;
};
```

```
gpuDrawIndexed(commandBuffer, data, mesh.indices, mesh.indexCount);
```

```
// Pixel shader...
const Texture textureHeap[];

PixelOut main(const VertexIn &vertex, const Data* data)
{
    Texture textureAlbedo = textureHeap[data->textureBaseIndex];
    Texture textureNormal = textureHeap[data->textureBaseIndex + 1];
    Texture texturePBR = textureHeap[data->textureBaseIndex + 2];

    Sampler sampler = {.minFilter = LINEAR, .magFilter = LINEAR};

    float32x4 albedo = sample(textureAlbedo, sampler, vertex.uv);
    return { .gBufferAlbedo = albedo, ... };
}
```

No texture bindings. Heap index calculated in the shader

Indirect Draw and Dispatch

GPU-pointer → Can write to it using **compute shader!**

```
// GPU kernel...
[groupsize = (64, 1, 1)]
void main(uint32x3 threadId : SV_ThreadID, const Data* data)
{
    uint32 value = data->input[threadId.x + data->offset.x];
    data->output[threadId.x] = value;
}
```

GPU pointers!

Multidraw: **`data[SV_DrawID]`** or **`data->array[SV_DrawID]`**

```
gpuDispatchIndirect(commandBuffer, data, arguments);
gpuDrawIndexedInstancedIndirect(commandBuffer, data, arguments, drawCount);
```

Barriers 😡😰🤯

VK_KHR_unified_image_layouts [17]

Massive mismatch between API and hardware

- **API:** User has to bookkeep individual resources and their layout transitions
  - Lots of bookkeeping: Lots of CPU cycles wasted
  - Awkward in multithreaded renderer: Need to know previous layout ahead of time → **HACKS!**
  - Awkward with 100% bindless: CPU needs to know which resources are written
- **Reality:** GPU doesn't care about individual resources at all
  - Samplers and display can read DCC today: No layout transitions for decompress
  - Coherent LLCs on all modern GPUs: Only a few special cases where $L1 → LLC flush is needed
- **Modern driver implementation**
  - Loops though your barrier resource list (looks for special usage & stage transitions)
    - Populates a bitfield (u32) denoting which $L1s to flush to LLC
  - Wasted work on both user land and in the driver
- **Fix:** User provides the (special-case) cache-flush bitmask directly. No resources!

Barriers 😀 😊 😌

API:

```
void gpuBarrier(GpuCommandBuffer cb, STAGE before, STAGE after, HAZARD_FLAGS hazards = 0);
```

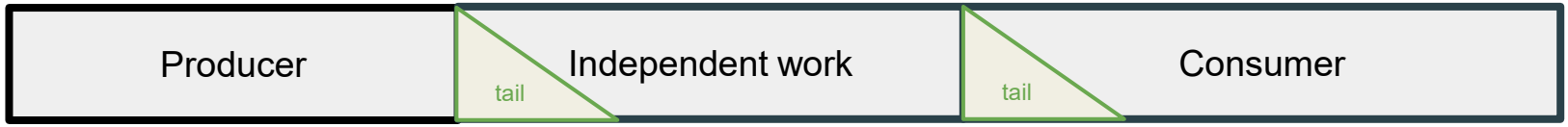
Examples:

```
// Compute -> compute barrier
gpuBarrier(commandBuffer, STAGE_COMPUTE, STAGE_COMPUTE);

// Hazard flag: Compute write to descriptor heap
gpuBarrier(commandBuffer, STAGE_COMPUTE, STAGE_COMPUTE, HAZARD_DESCRIPTORS);

// Raster pass -> pass (allow vertex overlap)
gpuBarrier(commandBuffer, STAGE_RASTER_COLOR_OUT | STAGE_RASTER_DEPTH_OUT, STAGE_PIXEL_SHADER);
```

Cache flush bitmask

No resource lists in barriers. User doesn’t need to track resource layouts

Split Barriers: 😡😰🤯 → 😀😊😌



NO STALL - TAILS OVERLAP

API:

```
void gpuSignalAfter(GpuCommandBuffer cb, STAGE before, void *ptr, uint64 value, SIGNAL signal);
void gpuWaitBefore(GpuCommandBuffer cb, STAGE after, void *ptr, uint64 value, OP op, HAZARD_FLAGS hazards
```

Example (GPU timeline semaphore):

```
gpuSignalAfter(commandBuffer, STAGE_RASTER_COLOR_OUT, gpuPtr, counter, SIGNAL_ATOMIC_MAX);
// TODO: Add independent work here
gpuWaitBefore(commandBuffer, STAGE_PIXEL_SHADER, gpuPtr, counter++, OP_GREATER_EQUAL);
```

No need for persistent GPU synchronization objects

Significantly cleaner and more flexible API (vs Vulkan and DX12)

Making the PSO thinner…

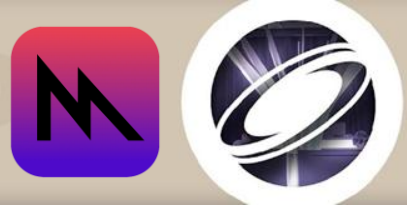
```
GpuRasterDesc rasterDesc =
{
    .topology = TOPOLOGY_TRIANGLE_LIST,
    .alphaToCoverage = false,
    .supportDualSourceBlending = false,
    .sampleCount = 1,
    .depthFormat = FORMAT_D32_FLOAT,
    .stencilFormat = FORMAT_NONE,
    .colorTargets =
    {
        { .format = FORMAT_RG11B10_FLOAT },
        { .format = FORMAT_RGB10_A2_UNORM },
        { .format = FORMAT_RGBA8_UNORM }
    },
    .blendstate = GpuBlendDesc { ... } // optional
};
```

Move as much state out of PSO as possible…

- Less PSOs = better performance + less stalls
- GPU hardware doesn't have monolithic PSO design
- Fixed function units still have their configuration packets
  - Rasterizer
  - Depth-stencil
  - Blender
  - (Input assembler is mostly gone)

```
GpuPipeline graphicsPipeline = gpuCreateGraphicsPipeline(vertexIR, pixelIR, rasterDesc);
```

Depth-Stencil State



```
GpuDepthStencilDesc depthStencilDesc =
{
    .depthMode = DEPTH_READ | DEPTH_WRITE,
    .depthTest = OP_LESS_EQUAL,
    .depthBias = 0.0f,
    .depthBiasSlopeFactor = 0.0f,
    .depthBiasClamp = 0.0f,
    .stencilReadMask = 0xff,
    .stencilWriteMask = 0xff,
    .stencilFront =
    {
        .test = OP_ALWAYS,
        .failOp = OP_KEEP,
        .passOp = OP_KEEP,
        .depthFailOp = OP_KEEP,
        .reference = 0
    },
    .stencilBack = { ... }
};
```

Separate depth-stencil state

- Metal already separates depth-stencil state
- Depth-stencil state has low frequency of change
  - Once per pass is common
  - N subsequent draws
  - No good reason to bake it in the PSO (*)

```
GpuDepthStencilState depthStencilState = gpuCreateDepthStencilState(depthStencilDesc);
```

Blend State

```
GpuBlendDesc blendDesc =
{
    .colorOp = OP_ONE,
    .srcColorFactor = FACTOR_SRC_ALPHA,
    .dstColorFactor = FACTOR_ONE_MINUS_SRC_ALPHA,
    .alphaOp = OP_ONE,
    .srcAlphaFactor = FACTOR_SRC_ALPHA,
    .dstAlphaFactor = FACTOR_ONE_MINUS_SRC_ALPHA,
    .colorWriteMask = 0xf
};
```

Has blending hardware?

- **Yes:** AMD, Nvidia, Intel, Qualcomm, ARM
- **No:** Apple, PowerVR

Options for user:

- Burn blend state into PSO
- Separate blend state
- Framebuffer fetch (mobile)

```
GpuBlendState blendState = gpuCreateBlendState(blendDesc);
```

Dynamic Render State

```
void gpuSetCull(GpuCommandBuffer cb, CULL cull); // CCW, CW, ALL, NONE
void gpuSetScissor(GpuCommandBuffer cb, Span<Rect2> scissors);
void gpuSetViewport(GpuCommandBuffer cb, Span<Rect3F> viewports);
```

- Separate depth/stencil + blend state + above dynamic state matches Vulkan 1.3
  - Input assembler state is implicitly fully dynamic (no vertex buffers)
  - No possibility to burn viewport, scissor, cull (winding) or depth/stencil into PSO.
    - Single code path simplifies driver state management.
- Limited dynamic state is a MASSIVE issue today
  - Android = Vulkan 1.1 (good coverage). No dynamic render state.
  - DirectX 12 is only slightly bit better than Vulkan 1.1 (no Vulkan 1.3 equivalent patch)

Comparison of APIs

	CPU→GPU ptr access	Pointers in shader	Descriptor heap	Root/data	Barriers
Proposed	data, root descriptors	YES	YES	const ptr = root	Dependency Only
DirectX 12	data	NO (HLSL)	YES	Messy APIs	Resource list
Mobile today* — Vulkan 1.4 +BDA +pushData +descriptorHeap +unifiedImageLayouts	data descriptors	u64 (GLSL) bad syntax	YES (extension)	Push struct (extension)	List, no layouts
Mobile today* — Metal 4	data	YES	TextureView Pool**	const ptr = root [N]	Dependency Only
WebGPU	NO	NO	NO	Immutable sets	Automatic (slow)

Latest 2-3 generations of mobile GPUs from all vendors support everything we need!

More Info: 50-page Blog Post

https://www.sebastianaaltonen.com/blog/no-graphics-api [18]

References

[1] Logos: Copyright: Khronos (OpenGL, WebGL, WebGPU, Vulkan), Apple (Metal, MacOS, iOS), Microsoft (DirectX, Xbox, Windows), Google (Android), Nvidia (CUDA), Sony (PS), Nintendo, Unity, Ubisoft, HypeHype

[2] Image: 3dFX Voodoo 2 (TechPowerUp, 1998). https://www.techpowerup.com/gpu-specs/voodoo2-12-mb.c3560

[3] News headline: Tested: DirectX 12 potential performance leap is insane (PC World, 2015). https://www.pcworld.com/article/432594/tested-directx-12s-potential-performance-leap-is-insane.html

[4] News headline: ARK Survival Evolved DX12 rendered delayed, performs worse than DX11 (dsogaming, 2015). https://www.dsogaming.com/news/ark-survival-evolved-dx12-renderer-delayed-performs-worse-than-dx11/

[5] Chart: Battlefield DX12 Ultra: DX11 vs DX12 (TechSpot, 2015). https://www.techspot.com/review/1267-battlefield-1-benchmarks/page5.html

[6] Forum post: Unity 2018.2.17 poor Vulkan performance (2017). https://discussions.unity.com/t/vulkan-api-poor-performance/680636

[7] News headline: Unity founder: DirectX 12 alone doesn't give a significant performance boost (wccftech, 2015). https://wccftech.com/unity-founder-dx12-doesnt-give-significant-performance-boost/

[8] Chart: Cache size benchmark, Borderlands 4 (TechPowerUp, 2025). https://www.techpowerup.com/review/borderlands-4-performance-benchmark/

[9] News headline: Forza Horizon boots up in just 4 seconds instead of 90 with new Advanced Shader Delivery tech (Toms Hardware, 2026). https://www.tomshardware.com/pc-components/gpu-drivers/forza-horizon-6-boots-up-in-just-4-seconds-instead-of-90-with-new-advanced-shader-delivery-tech-and-amd-gpus-microsoft-claims-95-percent-reduction-in-gaming-load-times

[10] News headline: New in DX12 - shader background optimization (Microsoft DevBlog, 2019). https://devblogs.microsoft.com/directx/background-shader-optimizations/

[11] News headline: Nvidia's 378.78 driver delivers giant gains (PC Gamer, 2017). https://www.pcgamer.com/nvidias-37878-driver-delivers-giant-gains-in-several-directx-12-games/

[12] The Outlast Trials DX12 (HardwareTimes, 2023). https://hardwaretimes.com/latest-intel-arc-gpu-drivers-boost-gaming-performance-by-up-to-65-in-outlast-trials-and-minecraft/

[13] vkCmdPushDataEXT: https://docs.vulkan.org/refpages/latest/refpages/source/vkCmdPushDataEXT.html

[14] BufferDeviceAddress (BDA): https://docs.vulkan.org/samples/latest/samples/extensions/buffer_device_address/README.html

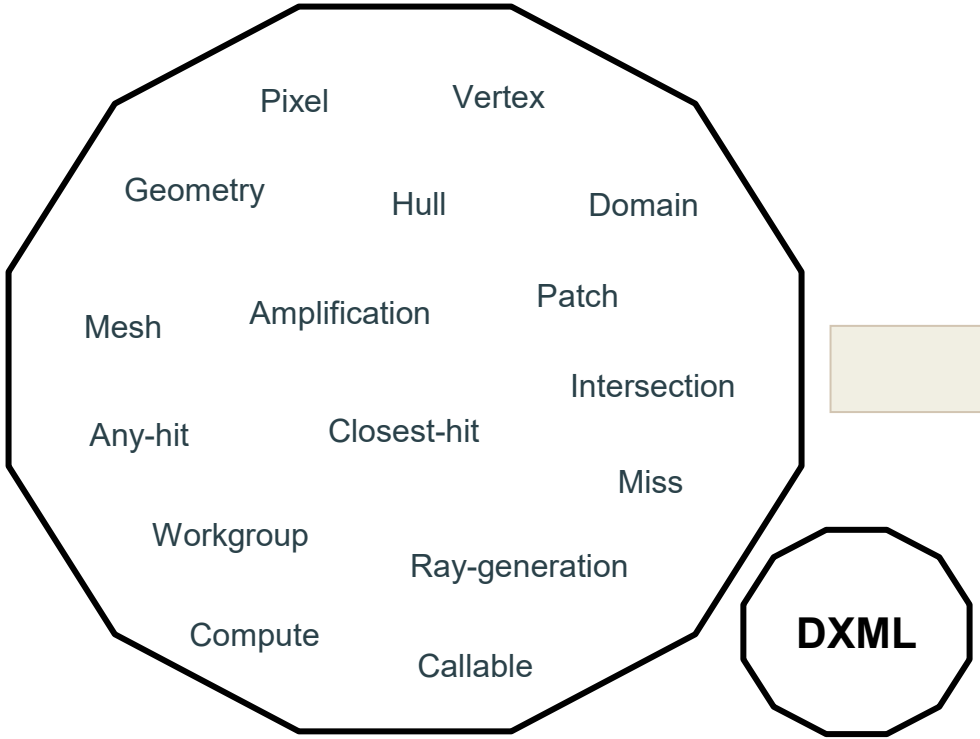
[15] Shader Model 6.6: https://devblogs.microsoft.com/directx/hlsl-shader-model-6-6/

[16] VK_EXT_descriptor_heap: https://docs.vulkan.org/features/latest/features/proposals/VK_EXT_descriptor_heap.html

[17] VK_KHR_unified_image_layouts: https://docs.vulkan.org/refpages/latest/refpages/source/VK_KHR_unified_image_layouts.html

[18] No Graphics API (Sebastian Aaltonen, 2025). https://www.sebastianaaltonen.com/blog/no-graphics-api

EXTRA: Shader Framework



One kernel with intrinsics:

- Sampler (bindless)
- Ray (BHV + triangle)
- Tensor
- Spawn wave (data = ptr)
- Meshlet (→rasterizer)

A proper library ecosystem needs composable constructs. CUDA is a $5T library ecosystem. HLSL and GLSL frameworks lack libraries.