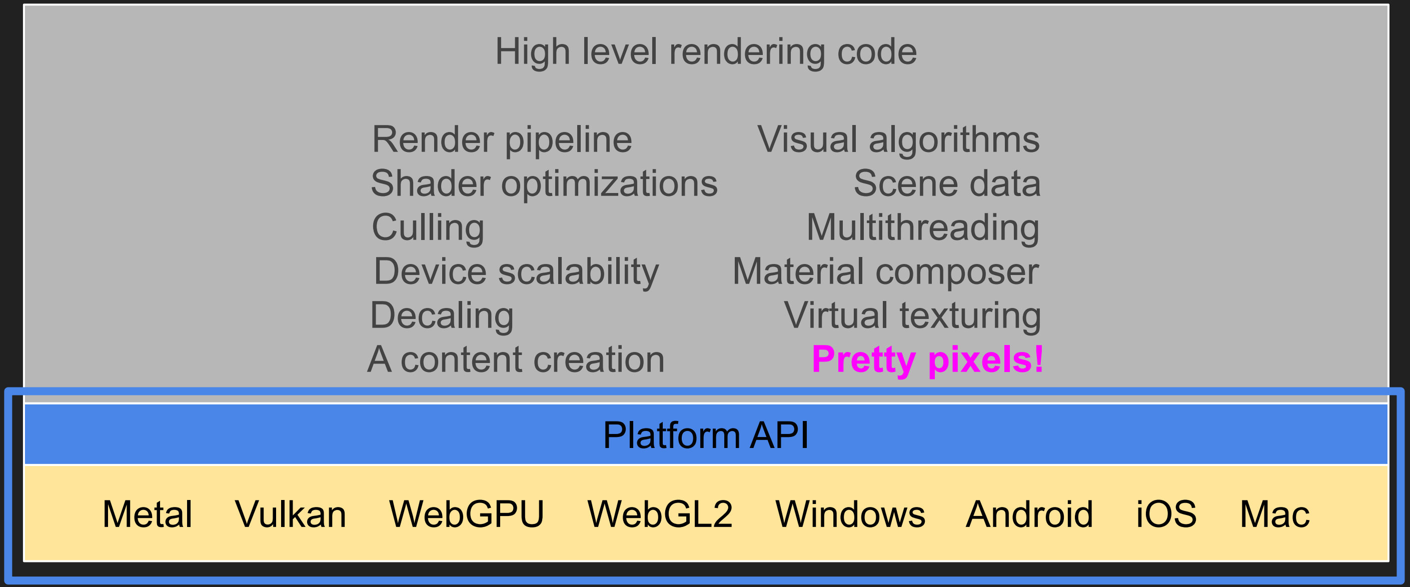
Modern Mobile Rendering @ hypehype

Sebastian Aaltonen (2023)

Modern Mobile Rendering @ HypeHype

- **Research**
  - Understand your target hardware (and audience)
  - Why can’t we have Nanite and Media Molecule: Dreams on a phone?
- **Design**
  - What is the correct platform abstraction level?
  - The iterative API design process
  - Do things at the right frequency and granularity
- **Implementation**
  - Fast & safe object lifetime tracking
  - Fast & clean C++20 API for constructing resources
  - Efficient GPU memory allocation
  - Bind groups, exposed to user land
  - A software command buffer, but an order of magnitude faster

Scope of today’s presentation



Today’s scope = bottom levels. Pretty pixels next time!

Research

Understanding your target audience

- Gather analytics data
  - GPU manufacturer, model, driver version
  - Amount of RAM, OS version
- Compare analytics data to older data
  - Which devices are soon leaving the market?
  - Extrapolate one year in the future = project ETA
  - Keep tracking the data to adjust plans during production
- What is the correct “min spec” hardware?
  - Android: 95% of our users have Vulkan 1.0 + Android 9
  - 2GB memory (1.4GB usable)
  - Have to cut bottom 5% of users
    - New tech improves bottom 50% experience a lot
    - Better user retention for 95% of users

Most popular Android devices

	DEVICE MODEL
1	vivo; vivo 1906
2	OPPO; CPH2083
3	realme; RMX3231
4	OPPO; CPH1909
5	samsung; SM-A125F
6	samsung; SM-T225
7	OPPO; CPH2269
8	samsung; SM-T295
9	samsung; SM-A115F
10	samsung; SM-A035F
11	realme; RMX3263
12	vivo; V2120
13	Xiaomi; M2006C3MG
14	HUAWEI; MRD-LX2
15	INFINIX MOBILITY LIMITED; Infinix X688B
16	samsung; SM-A127F
17	vivo; V2026
18	OPPO; CPH1937
19	realme; RMX3261
20	realme; RMX3195

HypeHype top 20 Androids

Understanding your target hardware

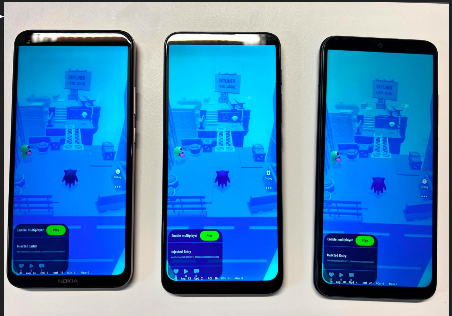
- Form contacts with mobile hardware vendors
  - ARM (Mali), Qualcomm (Adreno), PowerVR, Apple
  - Present your early design. Get feedback. Ask questions
- Read the best practice guides and API docs
  - Questions → Ask your IHV contacts
- Trick: Read the new hardware marketing material
  - Big improvements → That's still SLOW on 50%+ devices!
- Buy test devices
  - Min spec device of every GPU vendor
- Prototyping
  - Write a small test app to measure the most important gfx API features on each vendor min spec device
  - Confirm that the driver works for our use cases

Android

OS: Android 9
CPU: 32 bit + 64 bit
ARM: Mali-G series (Bifrost)
Qualcomm: Adreno 500 series
PowerVR: 8000 series (Rogue)
6 years old hardware

Apple

OS: iOS 13
CPU: 64 bit
iPhone 6s (A9) / iPad Air 2 (A8X)
7 years old hardware

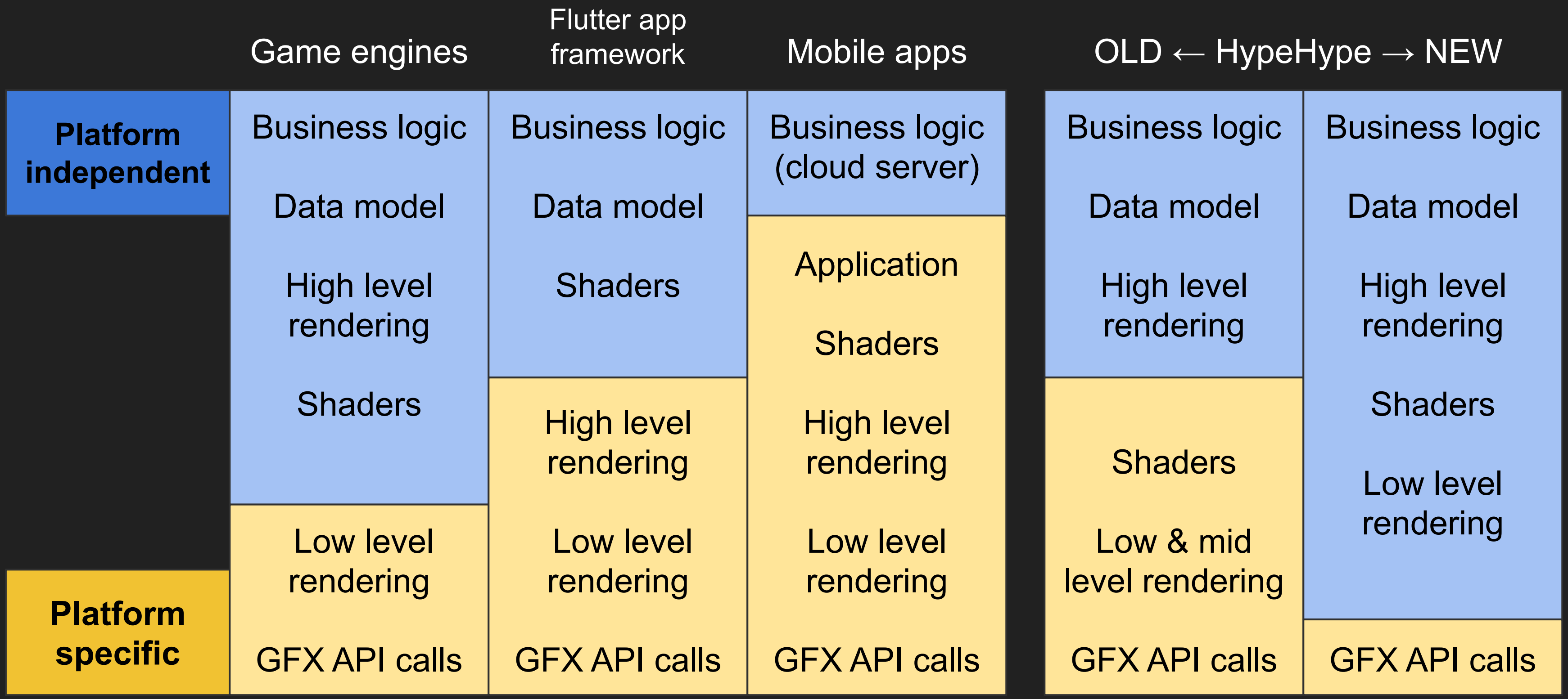


Why can’t we have Nanite or MM: Dreams on a phone?

- GPU-driven rendering: 8 years ago at SIGGRAPH 2015 [1]
- SDF ray-tracing (Claybook): 5 years ago at GDC 2018 [2]
- Nintendo Switch (handheld) versus bottom 50%+ mobile phones [3]
  - Peak flops (~200 GFLOP/s) and mem bandwidth (~20 GB/s) are in the same ballpark
  - Nvidia GPU architecture is designed for compute (CUDA, AZDO):
    - Fast generic memory load/store - Mobile: 16KB uniform buffers! SSBOs are slow!
    - Fast & big groupshared memory - Mobile: Small or emulated
    - Fast local/global atomics and wave intrinsics - Mobile: Wave intrinsic support <10%
    - Big register files and big generic caches - Mobile: Avoid complex shaders
    - 64 bit atomics - Mobile: No 64 bit integers at all!
  - Modern PC graphics: 3d tiling layout for volume textures. Big deal for SDF rendering
- 50%+ of mobile phones: Designed to run existing GLES 3.0 games efficiently

Design

What is the correct platform abstraction level?



Issues: new features, API bloat, fast paths, maintenance, dependencies, parity issues, sim ship?

Our solution: Minimal platform abstraction

- Thin low level gfx API wrapper
  - Cross reference Vulkan, Metal and WebGPU docs
  - Find the common set of features and differences
  - Design performance optimal way to abstract the differences
  - Metal 2.0: Placement heaps, argument buffers, fences
- Trim deprecated stuff
  - Transform feedback
  - Strips, fans
  - Geometry shaders, HW tessellation
  - Vertex buffers?
    - Some mobile devices still benefit
- Single set of shaders
  - GLSL and use SPIRV-Cross to cross compile [4]

Low level API design goals

- Avoid higher level concepts creeping into low level API
    - No mesh or material: Can be represented as VBs + IB and bind groups.
    - No automatic data setup or forced data layout
    - No fixed draw algorithm: Traditional, instancing, etc. Future = GPU-driven?
    - No data loading from disk
- "Zero" extra API overhead
    - Design core pillar: As easy to use as DX11, but as fast as hand optimized DX12
    - Wrong solution: Implement DX11 driver in your code base
    - Potential performance pitfalls:
        - Fine grained inputs, render state and data copies
        - Resource state tracking, shadow state
        - PSO + render state and bind group caching (hash tables)
        - Software command buffers

What is the correct process for API design?

Traditional

- Big technical design document
- Scheduled & split into tasks
- Design first, then code

Issues
- Plans locked too early
- Programmer notices architectural issues too late
- Refactor impacts production

Agile + Test Driven

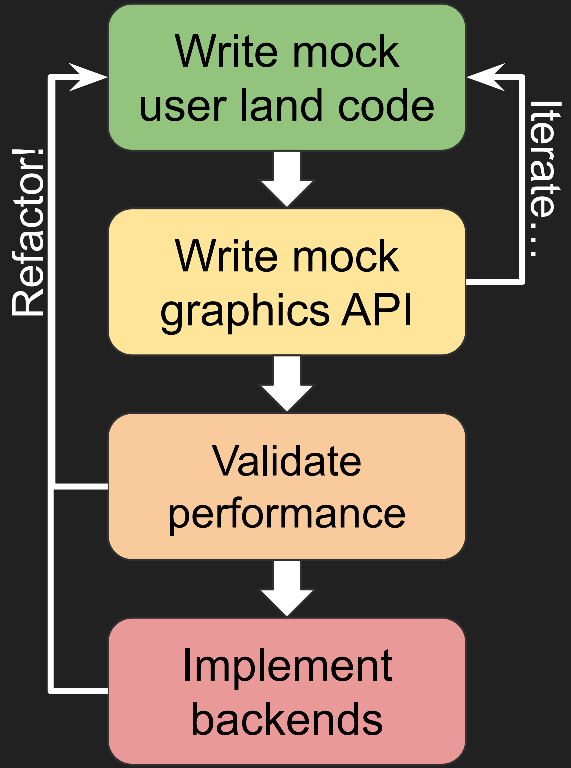
- What we need in the next sprints?
- Implement small pieces of tested production ready modular code

Issues
- Can’t see the forest from the trees
- Good pieces != good architecture
- Hard to throw away production ready code with 100% tests

? Traditional ? Agile ?

Our solution: Iterative API design process

- Write mock user land code
  - Create resources: textures, shaders, buffers, passes, etc
  - Setup resources with valid data
  - Render a full frame using the resources (+animate)
- Write mock gfx platform API
  - Don't write any backend implementation code yet
  - Compile it with the mock user land code to syntax check
  - Program doesn't yet link or run. It's 100% fine!
- Iterate until happy
  - Add mock use cases whenever needed to improve the coverage
  - Do big architecture refactorings immediately when issues surface
  - Do we cleanly implement all gfx APIs? Abstract differences optimally?
  - Is the performance good? No allocs, copies, map lookups, etc…
- Finally: Implement the platform backends
  - Refactor ASAP if issues are found!
  - Vulkan/Metal API validation layers == big pre-existing test suites



Do things at the right frequency and granularity

- Temporal coherency
  - We are rendering the same game world 60 times per second
  - The camera moves smoothly (most of the time)
  - ~90% of the data is unchanged from the previous frame
- Operation frequencies
  - Once: Load a game world (+ all baked data)
  - Low: Load a mesh, texture, material or shader
  - Low: Change material texture bindings
  - Low: Change objects mesh, material, shader variant or render state
  - Medium: Change camera and sun properties
  - Medium: Modify object/material color (<10% objects)
  - Medium: Modify objects transform (<20% objects)
  - Medium: Change object visibility or LOD level
  - High: Render the object

Do all of these inside the draw loop?

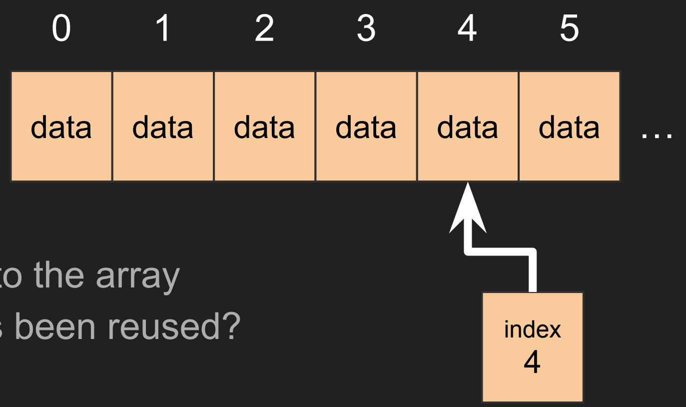
Our solution: Separate lower frequency ops from drawing

- PSOs
  - Build all pipelines (all render state combinations) at application startup
  - Store the PSO handle to each objects visual component
- Bind groups
  - Create a bind group per material at level load: Contains all texture and buffer bindings
  - Store the material bind group handle to each objects visual component
  - Changing the material = a single Vulkan/Metal command
- Data upload
  - Persistent data: Upload once at startup. Delta update when data changes. [5]
  - Dynamic Data:
    - Batch upload whole pass: No per-draw map & unmap
    - Separate by frequency: Per pass | per draw
- Resource synchronization
  - Render pass: RT texture transitioned to write and then read
  - No state tracking per draw call

Implementation

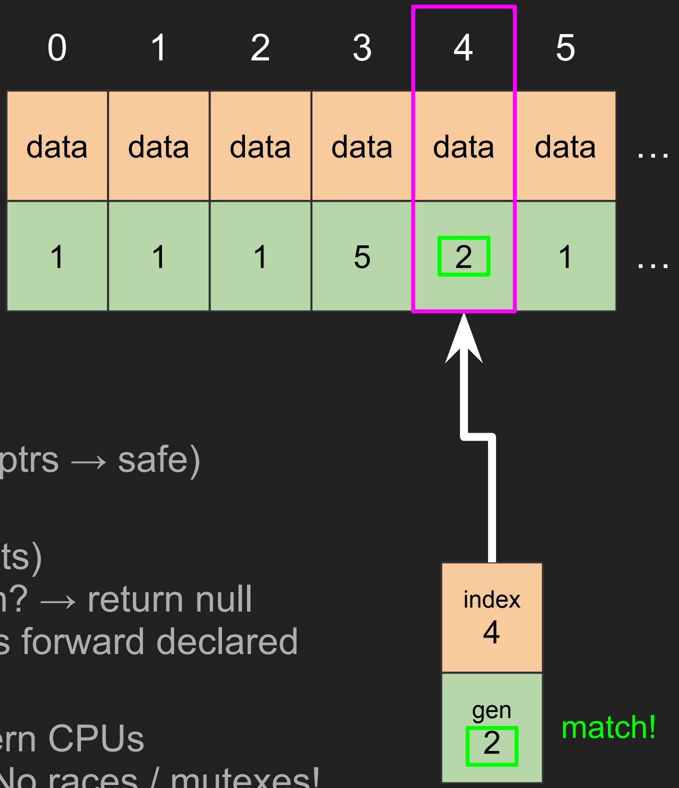
Fast & safe object lifetime tracking

- Modern practices: Smart pointers, ref counting and RAII?
  - Too slow: Memory allocation per object, scatters data around the memory causing cache misses, copy pointer = 2x atomics
  - Safety issues: Ref count runs out while iterating an array causing a destructor RAII side effect, maybe in another thread. Using a mutex kills performance
- Our solution: Arrays!
  - One big allocation for all objects of the same type
  - Array index is a nice data handle
    - POD. Trivial to copy and pass around
  - Safe to pass to worker threads
    - Can't dereference an array index. Needs access to the array
  - **PROBLEM:** Old handles referring an array slot that has been reused?



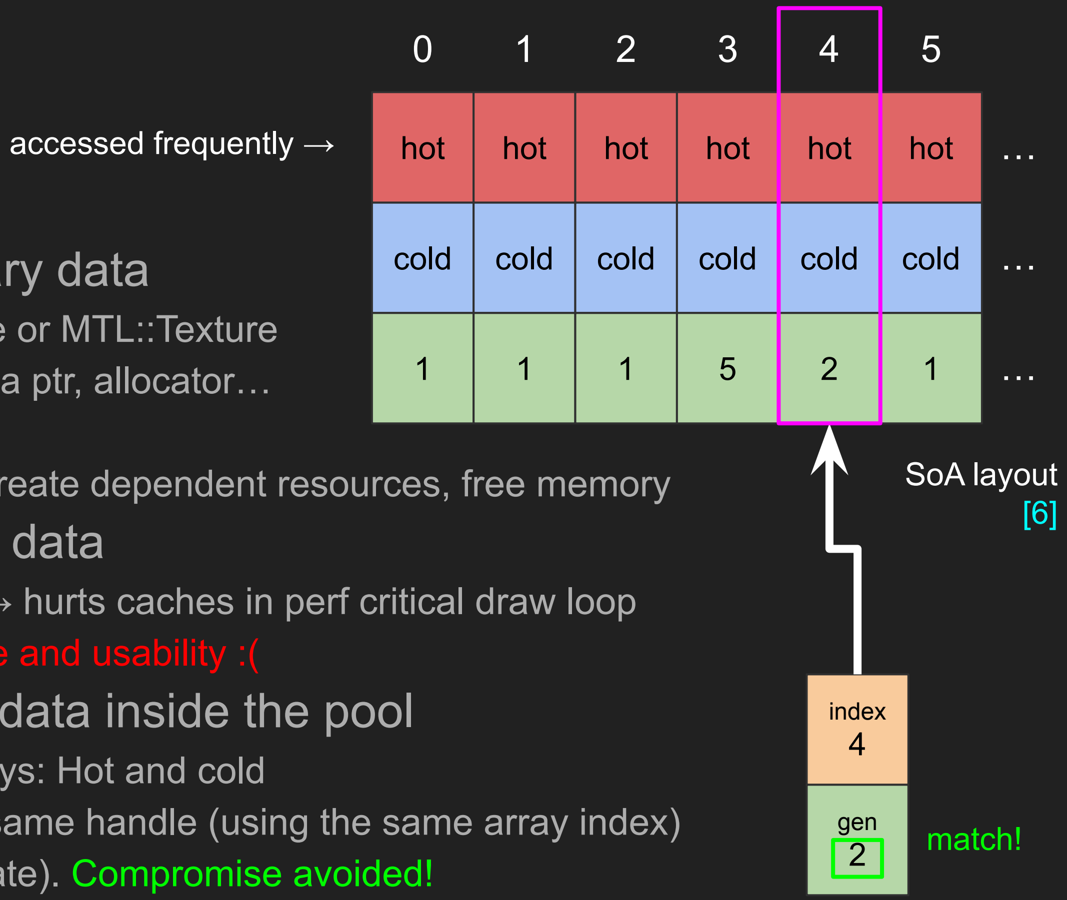
Pools and handles

- Pool
  - Typed array of objects
    - Every array slot has a generation counter
    - Counter is increased when the slot is freed
  - Freelist for slot reuse
    - An array (stack) of unused pool indices
    - Delete object = push index
    - Create object = pop index. Resize if needed (no ptrs → safe)
- Handle
  - POD struct: Array index + generation counter (32/64 bits)
  - pool.get<T>(handle): Compare generations. Not match? → return null
  - Typed Handle<T>. Pool has the same handle type. T is forward declared
- Weak reference semantics
  - Null check (predictable branch) is almost free on modern CPUs
  - Much better than callbacks in multithreaded systems. No races / mutexes!



Hot vs cold data

- Easy to use API needs auxiliary data
  - Texture can't be just a VkTexture or MTL::Texture
  - Additional data: size, format, data ptr, allocator…
  - Needed for low frequency tasks:
    - Update, readback, sync, create dependent resources, free memory
- Rendering needs only the hot data
  - Auxiliary data bloats the struct → hurts caches in perf critical draw loop
  - Hate compromising performance and usability :(
- Our solution: Split hot at cold data inside the pool
  - Pool has two types and two arrays: Hot and cold
  - Both can be accessed with the same handle (using the same array index)
  - Split hot and cold data (investigate). Compromise avoided!



Fast & clean C++20 API for constructing resources

- Vulkan and DirectX use big structs to initialize complex resources
  - Structs contain other structs and non-owning pointer references to arrays of structs
  - Code bloat. No default values. Lifetime of temporary objects causes bugs
- Existing solutions
  - Builder pattern: Debug perf is horrible. Release codegen not optimal either
- Our solution: C++20 designated struct initializers
  - The best C99 feature finally in C++. Waited 11 years!
  - Default values:
    - Provided by C++11 struct aggregate initialization
    - Extremely clean syntax. Best readability
  - Array data?
    - Custom span that supports initializer lists
    - Safety: “const &&” parameter forces temporaries

```
struct BufferDesc
{
    const char* debugName = nullptr;

    uint32 byteSize = 0;
    USAGE usage = USAGE_UNIFORM;
    MEMORY memory = MEMORY::CPU;
    f::Span<const uint8> initialData;
};
```

Resource construction examples

```
Handle<Buffer> vertexBuffer = rm->createBuffer({
    .debugName = "cube",
    .byteSize = vertexSize * vertexAmo,
    .usage = BufferDesc::USAGE_VERTEX,
    .memory = MEMORY::GPU_CPU });
```

```
Handle<Texture> texture = rm->createTexture({
    .debugName = "lion.png",
    .dimensions = Vector3I(256, 256, 1),
    .format = FORMAT::RGBA8_SRGB,
    .initialData = Span((uint8*)data, dataSize)
});
```

```
Handle<BindGroup> material = m_rm->createBindGroup({
    .debugName = "Car Paint",
    .layout = materialBindingsLayout,
    .textures = { albedo, normal, properties },
    .buffers = {{.buffer = uniforms, .byteOffset = 64}}
});
```

```
m_shader = rm->createShader({
    .debugName = "mesh_simple",
    .VS {.byteCode = shaderVS, .entryFunc = "main"},
    .PS {.byteCode = shaderPS, .entryFunc = "main"},
    .bindGroups = {
        { m_globalsBindingsLayout }, // Globals bind group (0)
        { materialBindingsLayout },  // Material bind group (1)
    },
    .dynamicBuffers = dynamicBindings.getLayout(),
    .graphicsState = {
        .depthTest = COMPARE::GREATER_OR_EQUAL, // inverse Z
        .vertexBufferBindings {
            {
                // Position vertex buffer (0)
                .byteStride = 12, .attributes = {
                    {.byteOffset = 0,.format = FORMAT::RGB32_FLOAT}
                }
            },
            {
                // 2nd vertex buffer: tangent, normal, color, texcoord
                .byteStride = 24, .attributes = {
                    {.byteOffset = 0,.format = FORMAT::RGBA16_FLOAT},
                    {.byteOffset = 8,.format = FORMAT::RGBA16_FLOAT},
                    {.byteOffset = 16,.format = FORMAT::RGBA8_UNORM},
                    {.byteOffset = 20,.format = FORMAT::RG16_FLOAT}
                }
            },
        },
        .renderPassLayout = m_renderPassLayout
    } });
```

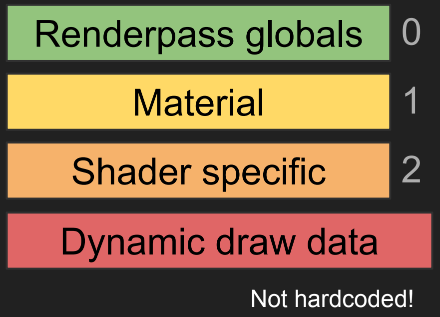
Efficient GPU memory allocation

- **Temp:** high frequency
  - Must be **extremely** fast (million calls per frame)
  - Bump allocate 128MB memory blocks (stored in a ring)
    - Backend heap object contains 1 full sized GPU buffer: Buffer = offset + heap index
  - Backend provides a concrete bump allocator object
    - Allocation function bumps a pointer. Inlines to caller
    - Checks ***offset >= 128MB*** → calls backend to provide the next block
    - **WebGL2:** 32MB CPU memory blocks, glBufferSubData call per render pass
- **Persistent:** only when needed!
  - Two-level segregated fit algorithm
    - O(1) hard real time alloc/free. Uses two level bitfield + 2x lzcnt to find the bin
    - Delete: Merge neighbor blocks on both sides, if they are free
  - Same allocator on Metal (placement heaps) and Vulkan!
  - I open sourced the allocator in Github (MIT license) [7]

Bind groups, exposed to user land

- Traditional way: Separate bindings
  - Backend creates new bind groups on demand
  - Problem: Creating new groups is expensive
  - Workaround: Store bind groups in hashmap → SLOW!
- Our solution: User land bind groups
  - User constructs an immutable persistent bind group from a set of bindings
    - Example: Material (5 textures + uniform buffer with value data)
  - Draw calls have three bind group slots: 0, 1, 2 (Vulkan Android min spec = 4)
    - Matching the GLSL shader descriptor set slots
    - Group data by bind frequency
- **Abstraction:** Dynamic bindings group
  - A flexible way to provide draw data. Only supports buffer bindings (with offset).
  - Vulkan/WebGPU: set 3 (dynamic offset). Metal: setBuffer + setOffset
  - Push constants? Emulated on many mobile GPUs :(

HypeHype bind groups



A software command buffer, but an order of magnitude faster

- Initial design: Array of draw structs
  - Only contains “metadata”
  - Super simple and fast
    - 64 bytes = 1 cache line per draw
  - Actual data inside buffers (inside groups)
    - Write temp data from N threads directly into GPU memory
- Let’s analyze the data
  - All fields are 32 bit integers
  - Most data doesn’t change between draw calls when rendering binned content
  - On average 4.5 fields change (~18 bytes)

```
struct Draw
{
    Handle<Shader> shader;
    Handle<BindGroup> bindGroups[3];
    Handle<DynamicBuffers> dynamicBuffers;
    Handle<Buffer> indexBuffer;
    Handle<Buffer> vertexBuffers[3];
    uint32 indexOffset = 0;
    uint32 vertexOffset = 0;
    uint32 instanceOffset = 0;
    uint32 instanceCount = 1;
    uint32 dynamicBufferOffsets[2] = {0};
    uint32 triangleCount = 0;
};
```

PSO with all render state

Draw stream: Our data interface for draw calls


```
struct Draw
{
    Handle<Shader> shader;
    Handle<BindGroup> bindGroups[3];
    Handle<DynamicBuffers> dynamicBuffers;
    Handle<Buffer> indexBuffer;
    Handle<Buffer> vertexBuffers[3];
    uint32 indexOffset = 0;
    uint32 vertexOffset = 0;
    uint32 instanceOffset = 0;
    uint32 instanceCount = 1;
    uint32 dynamicBufferOffsets[2] = {0};
    uint32 triangleCount = 0;
};
```

- Store only the modified fields of the draw struct
  - uint32 dirty mask tells which fields have modified
- User land: Draw stream writer class
  - Contains a draw struct (current state) + dirty mask
  - Setter for each field: if changed → set dirty bit + write field to stream
  - Draw: write dirty mask in front of the draw (stored offset)
- Backend: Stream decoding
  - For each draw: Read the dirty field bitmask
    - For each set bit: Read field and emit a gfx API call
  - **Advantages:** No change tracking in the backend. ~3x reduced BW

Example: Simple draw loop

```
// Per pass bindings
drawStream.setBindGroup(0, m_globalBindGroup);

// Draw all objects
for (const SceneObject& sceneObject : sceneObjects)
{
    // Bump allocate uniforms (in GPU memory)
    DynamicBinding drawData = tmpAlloc.allocate(sizeof(DrawData));
    DrawUniforms* uniforms = (DrawUniforms*)drawData.data;
    Matrix3x4 mat = Matrix3x4::translate(sceneObject.position);
    uniforms->model = mat;
    uniforms->modelInv = Matrix3x3::inverse(mat);

    // Draw
    drawStream.setShader[redacted](sceneObject.shader);
    drawStream.setBindGroup[redacted](1, sceneObject.material);
    drawStream.setDynamicBuffers[redacted](drawData.buffers);
    drawStream.setDynamicBufferOffset(0, drawData.byteOffset);
    drawStream.setMesh(meshes[sceneObject.meshIndex]);
    drawStream.draw();
}
```

← Per pass bindings (once)

← GPU temp bump allocator

← Write uniforms directly to GPU

← DrawStream setters

← Note: User land mesh

← Write draw dirty bitmask

Thank you!

10,000 draw calls (CPU time)

iGPU	AMD RDNA2 + 6800HS 4.7GHz	0.85ms
7 year old	Apple iPhone 6s + 1.85GHz	11.27ms
99€	PowerVR GE8320 + A53 2.3GHz	20.93ms
99€	ARM Mali G57 MP1 + A75 1.6GHz	15.01ms
149€	QC Adreno 610 + Kryo 260 2GHz	13.69ms

Single CPU thread
Standard non-instanced draws
No GPU persistent scene data
No batching: 10,000 mesh and material changes
>80% time spent in driver
All devices run cool over long period of time

iPhone 6s PBR
@ 60 fps

Art by Daniel Palmi

References

[1] Haar, Aaltonen: GPU-Driven Rendering Pipelines, SIGGRAPH 2015: Advances in Real-Time Rendering in Games, 2015
https://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf

[2] Aaltonen: GPU-Based Clay Simulation and Ray-Tracing Tech in 'Claybook', GDC, 2018
https://www.gdcvault.com/play/1025316/Advanced-Graphics-Techniques-Tutorial-GPU

[3] Various IHV documents and hardware statistics

- Wikipedia: Nintendo Switch (ALU & bandwidth): https://en.wikipedia.org/wiki/Nintendo_Switch
- Nvidia CUDA: https://en.wikipedia.org/wiki/CUDA
- Google Android Baseline Vulkan Profile (Android limits): https://github.com/KhronosGroup/Vulkan-Profiles/blob/main/profiles/VP_ANDROID_baseline_2021.json#L54
- Subgroup operation support in Vulkan 1.0 on Android (not supported devices): https://vulkan.gpuinfo.org/listdevicescoverage.php?extension=VK_EXT_shader_subgroup_ballot&platform=android&option=not
- 64 bit uint support in shaders on Android (not supported devices): https://vulkan.gpuinfo.org/listdevicescoverage.php?feature=shaderInt64&platform=android&option=not
- ARM lack of SSBO support in vertex shaders in GLES: https://community.arm.com/support-forums/f/graphics-gaming-and-vr-forum/52064/do-you-have-a-plan-to-support-ssbo-at-vertex-shader-on-opengl-es

[4] SPIRV-Cross: https://github.com/KhronosGroup/SPIRV-Cross

[5] Tatarchuk, Cooper, Aaltonen: Unity Rendering Architecture, Rendering Engine Architecture Conference (REAC), 2023
http://enginearchitecture.realtimerendering.com/downloads/reac2021_unity_rendering_engine_architecture.pdf

[6] SoA layout, Wikipedia: https://en.wikipedia.org/wiki/AoS_and_SoA

[7] Aaltonen: OffsetAllocator: https://github.com/sebbbi/OffsetAllocator