SIGGRAPH 2023
LOS ANGELES+ 6-10 AUG

THE PREMIER CONFERENCE & EXHIBITION ON COMPUTER GRAPHICS & INTERACTIVE TECHNIQUES

HypeHype

Mobile Rendering Architecture

Advances in Real-Time Rendering in Games *course*

What is HypeHype?

Feed

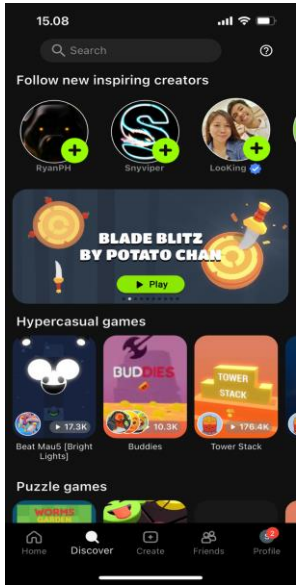
Instant loading games!

Multiplayer

8 players

Editor

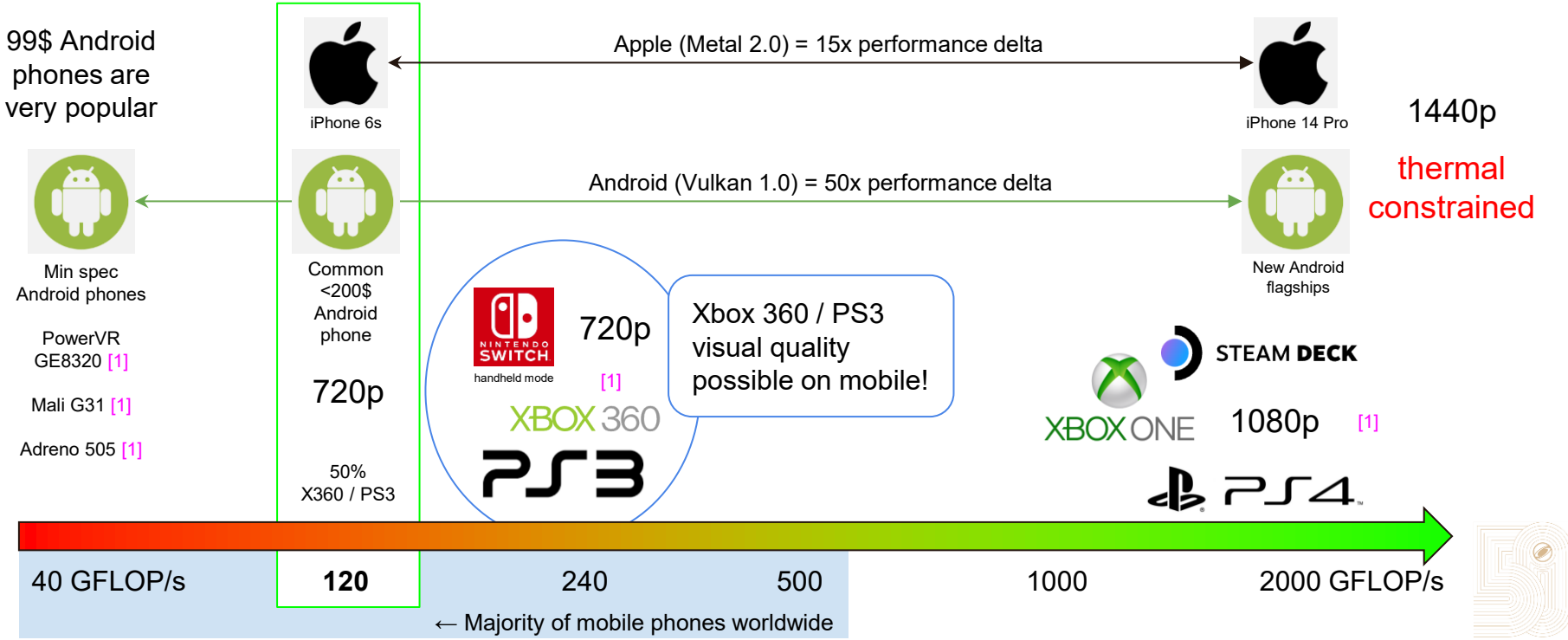
Visual scripting, collaborative edit, spectators, instant test play, no offline data cooking!



Social

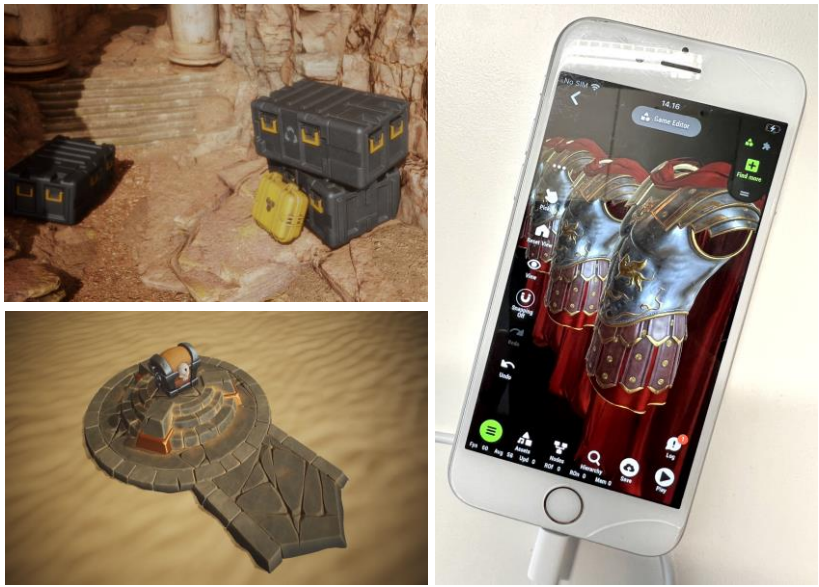
Chat, replays, leaderboards…

Target Hardware



Visual Target: Xbox 360 / PS3

Untextured, simple gamma space lighting, tiny scenes, no LODs

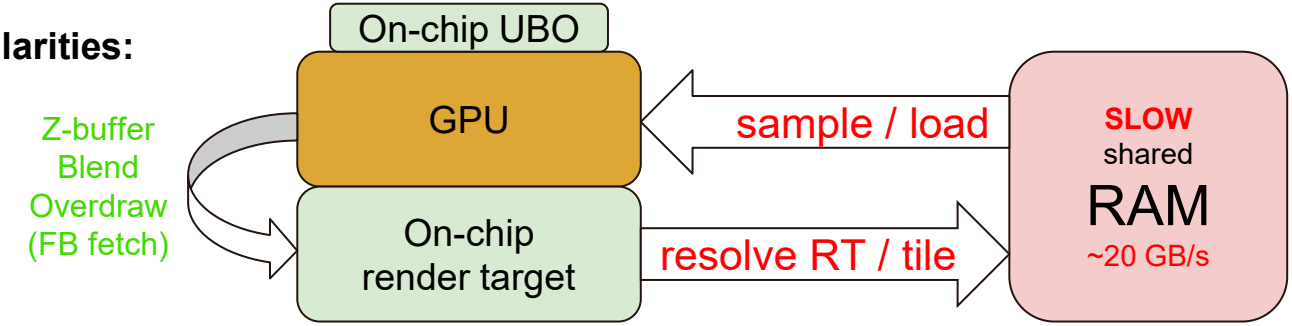


PBR materials, modern lighting and shadows, larger scenes, post…

@ 60 fps on iPhone 6s!

Mainstream Phones vs Xbox 360 (and Xbox One)

Similarities:



Differences:

Xbox	mobile
VS→PS varyings	VS→PS varyings

BOTH
Prefer uniform data
Pack data tightly
Combine passes
VS + PS over compute

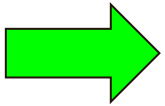
MOBILE
Minimize varyings
ALU over lookups (fp16)
Utilize (lossy) DCC
ASTC

Improvements: Framebuffer compression, ASTC textures (100% coverage), double rate fp16, 32 bit HDR render target formats.

Still not great: SSBO loads vs UBO (uniform access fast path). Compute is not perfect yet on mainstream phones (no DCC, no wave ops)

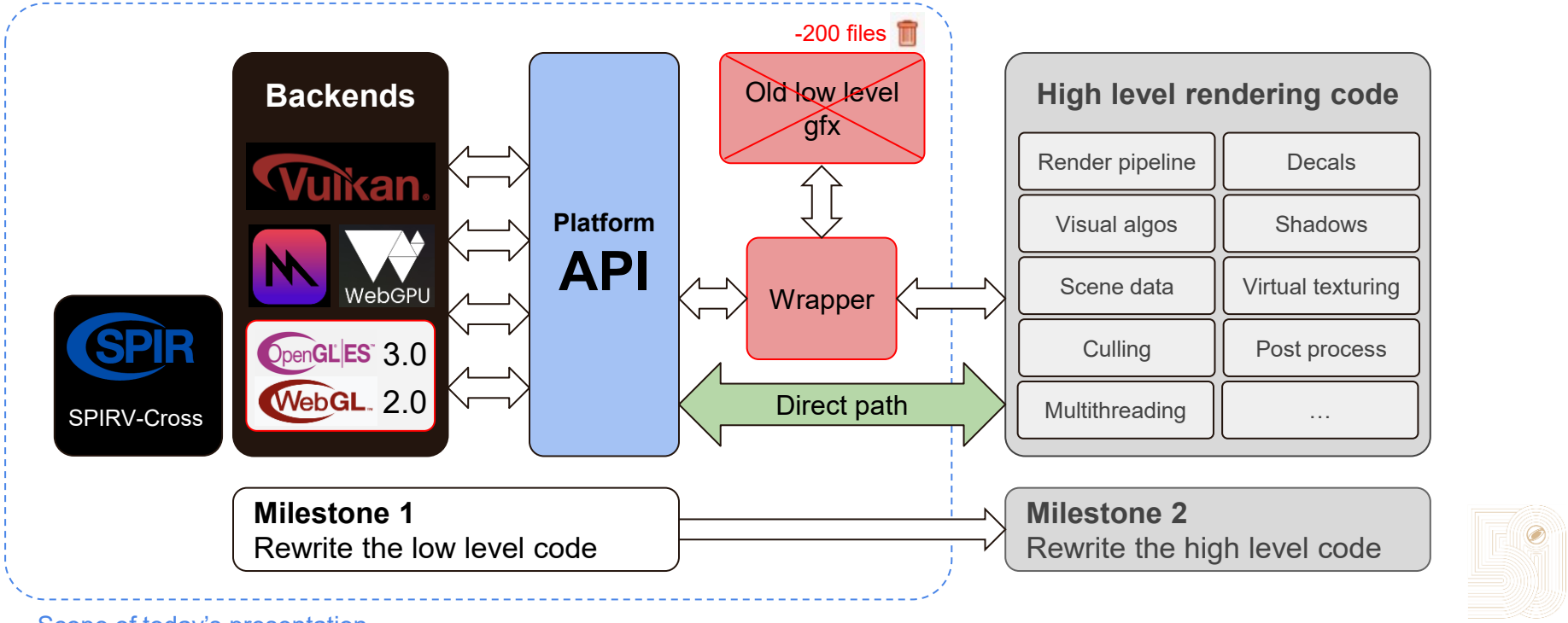
GPU-Driven Rendering on Mainstream Phones?

- GPU-Driven Rendering (Ubisoft), SIGGRAPH 2015 [2]: Proprietary console engines
  - Core ideas: GPU-driven 2-pass occlusion culling, fixed size clusters (meshlets), deferred texturing…
- Nanite (Epic), SIGGRAPH 2021 [3]: Mainstream availability
  - Combines: V-buffer, material classification, compute material passes, analytic derivatives, SW raster…
- Performance issues on mobile:
  - Heavy SSBO usage in pixel shader: load instance data (incl matrices), material data and 3x vertices (all attribs)
  - Full screen compute passes don't utilize framebuffer compression, cause pipeline bubbles
  - 64 bit atomics not available on mobile GPUs
  - SampleGrad is slow. ⅛ rate or even slower
  - Poor coverage of wave intrinsics, emulated groupshared memory
- New ray-tracing phones improve the situation!



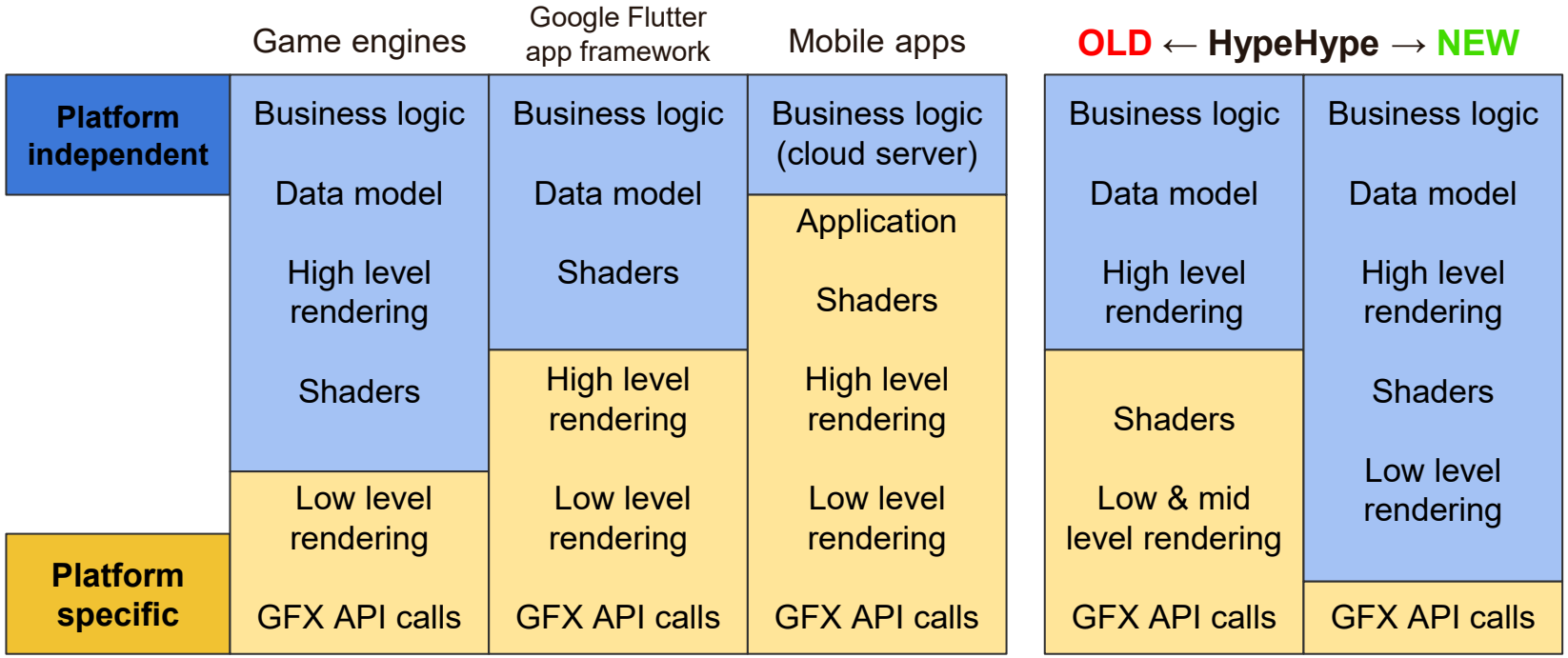
Fast traditional CPU draw call code is still relevant for mainstream mobile!

Roadmap for Renderer Rewrite



Scope of today's presentation

The Correct Platform Abstraction Level?



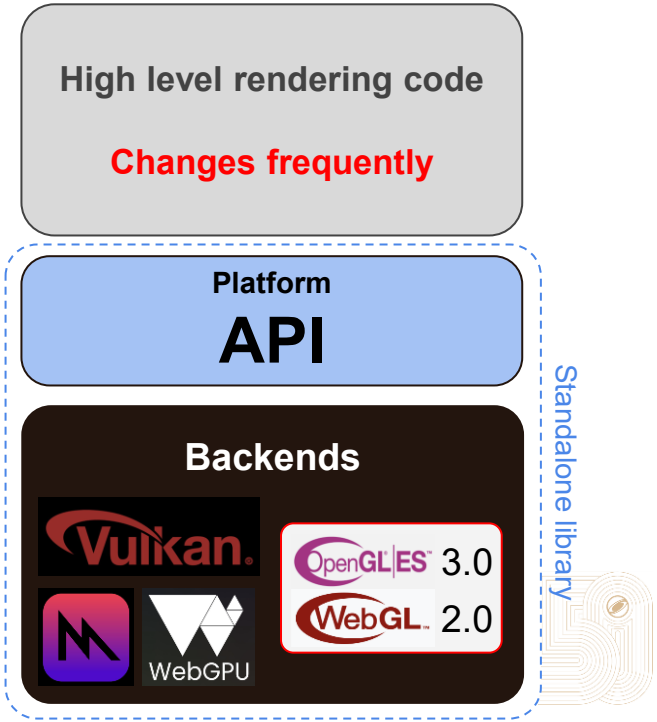
Issues: API bloat, “fast paths”, maintenance, testing matrix, parity issues, sim ship?

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
    - MoltenVK for debugging. Our Metal 2.0 backend is ~40% faster (CPU)
- Trim deprecated stuff
  - Transform feedback, strips, fans
  - Geometry shaders, HW tessellation
  - Vertex buffers?
    - Some mobile devices still benefit (shader codegen) and WebGL2!
- Single set of shaders
  - GLSL and use SPIRV-Cross to cross compile [4]

Platform API Design Goals

- Standalone library
  - Independently designed and maintained. Stable API
- Avoid higher level concepts creeping into the API
  - **No** mesh & material: Can be represented as IB+VBs and bind groups
  - **No** automatic data setup or forced data layout
  - **No** fixed draw algorithm: Traditional, instancing. Future = GPU-driven
  - **User land code responsible for setting up all the data!**
- "Zero" extra API overhead
  - **Design core pillar:** As easy as DX11, as fast as optimized DX12
  - **Wrong solution:** Implement DX11 driver in your code base
    - Fine-grained inputs, render state, shadow state, copies
    - PSO + render state caching, bind group caching (hash tables)
    - Software command buffers



Best Process for High-Performance 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

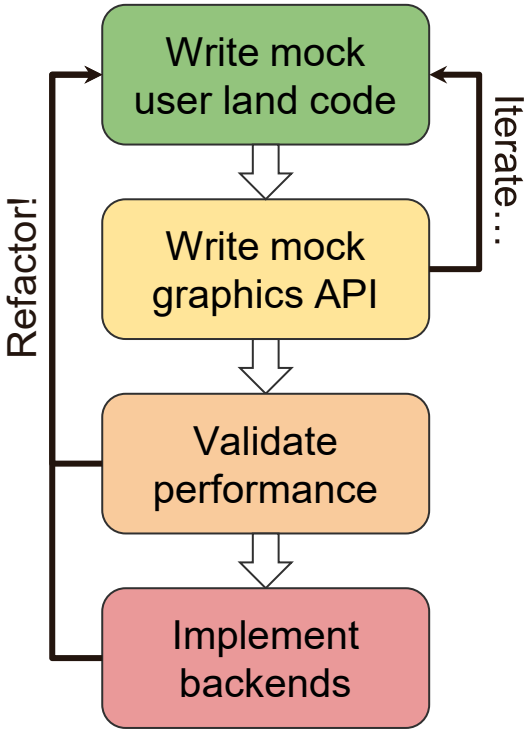
- What we need in the next sprints?
- Implement small pieces of tested production ready modular code

Issues

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

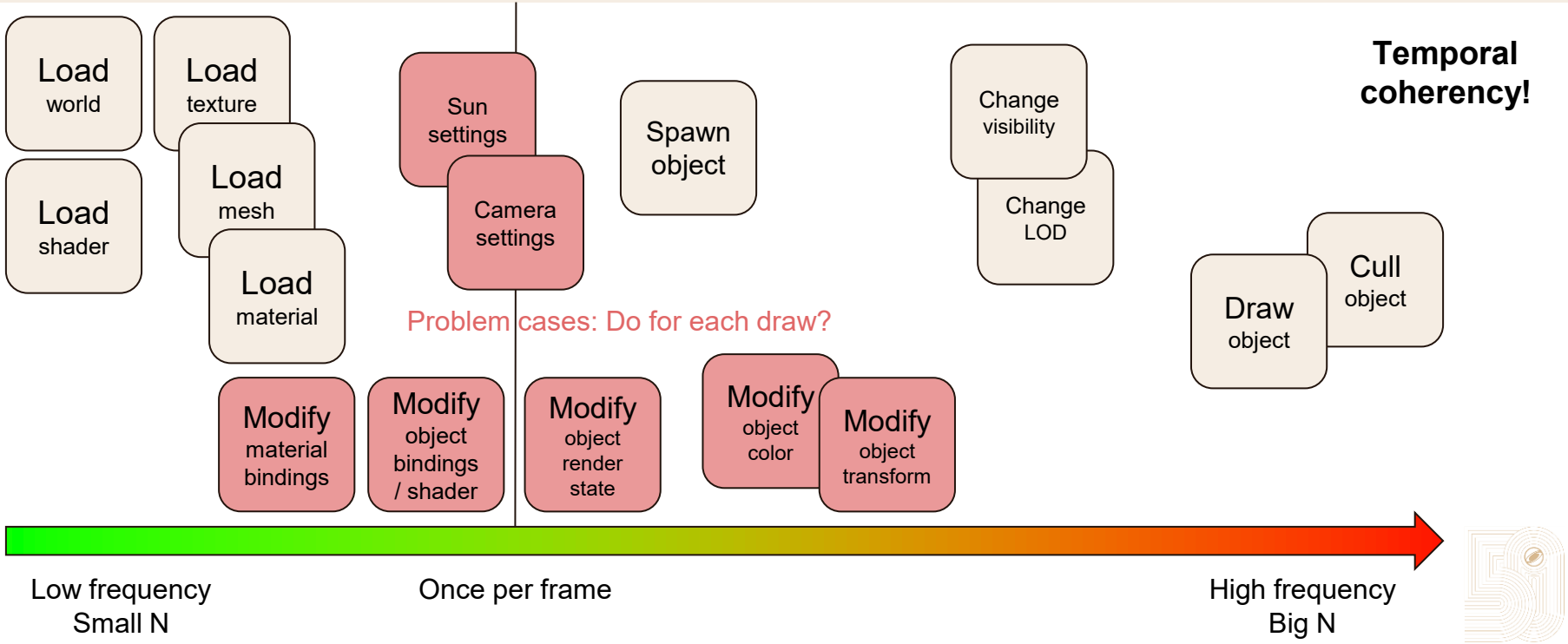
? Traditional ? Agile ?

Our Solution: Iterative API Design Process



- Write mock user land code
  - Create resources: textures, shaders, buffers, passes, etc
  - Render a full frame using the resources (+animate)
- Write mock gfx platform API
  - No backend implementation yet
  - Use compiler for syntax checking (no link / run yet)
- Iterate until happy
  - Add missing use cases & refactor whenever design feels sub-optimal
  - Is the performance optimal: Extra allocs, map lookups, etc?
- Finally: Implement the platform backends
  - Refactor ASAP when issues are found
  - Vulkan and Metal API validation layers = Thousands of free tests!

Process Things at the Right Frequency

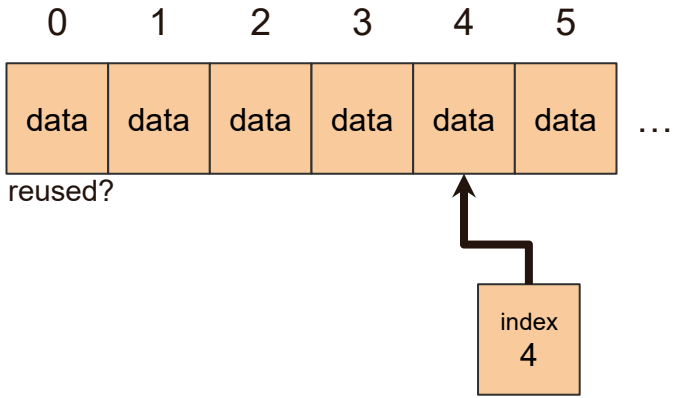


Our Solution: Separate Data Modification from Drawing

- PSOs
  - Build all pipelines (all render state combinations) at application startup. Doable since our PSO count is low
  - Store the PSO handle to each objects visual component
- Bind groups (descriptor sets)
  - 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, WebGPU 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

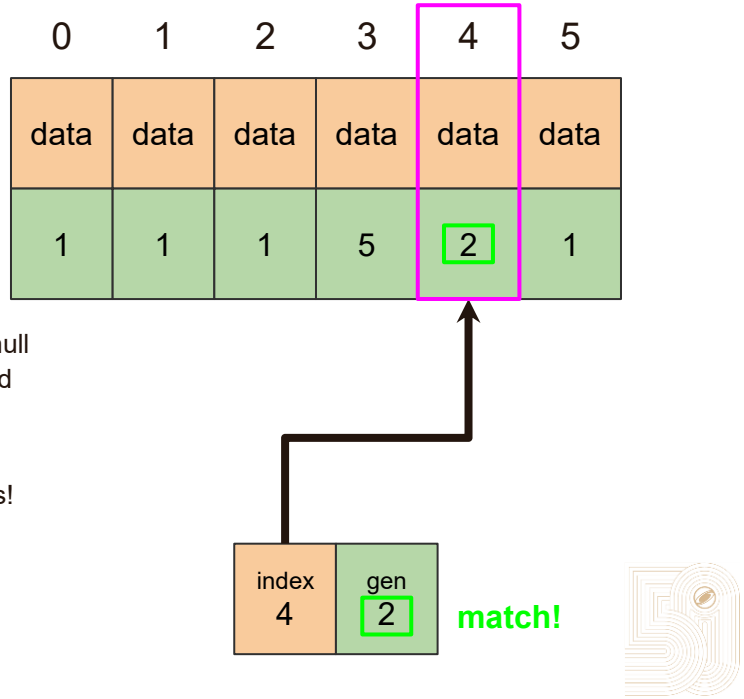
Fast & Safe Object Storage & Lifetime Tracking

- **Modern practices:** Smart pointers, ref counting and RAII [6]?
  - **Too slow:** Memory alloc per object. Scatters data around the memory (cache misses). Copy pointer is 2x atomics
  - **Safety issues:** Ref count runs out → RAII side effect → invalidates iterator (another thread). **Mutex is expensive!**
- **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?



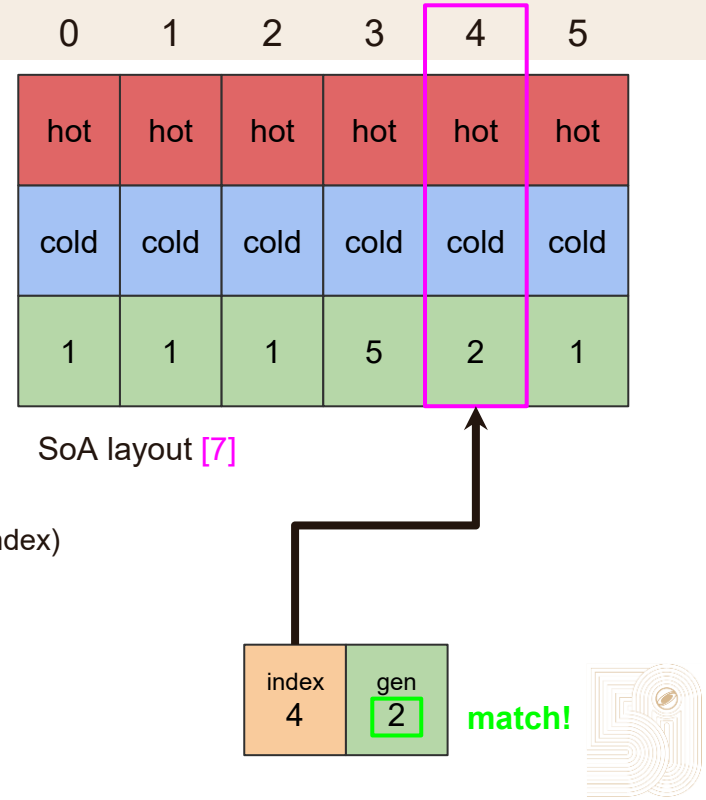
Generational Pools and Handles

- 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 → critical draw loop L1$ suffers
  - 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 Resource Construction

- 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: Use C++20 designated struct initializers [8]
  - 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
    } });
```

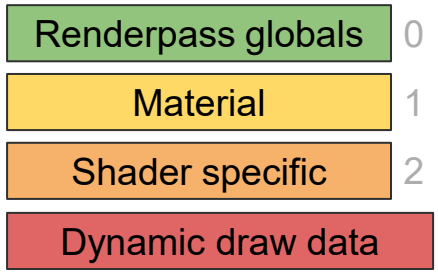
Efficient GPU Memory Allocation

- **Temp:** High frequency
  - Bump allocate 128MB memory blocks (stored in a ring)
    - Backend heap object contains a full sized GPU buffer: Buffer = offset + heap index
  - Backend provides a concrete bump allocator object
    - Allocation function bumps a pointer. Inlines to caller
    - `if offset >= 128MB` → call backend to obtain the next block
    - **WebGL2:** 8MB CPU memory blocks, glBufferSubData call per render pass
- **Persistent:** Only when needed!
  - Two-level segregated fit algorithm [9]
    - 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) [10]

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/WebGPU 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 some mobile GPUs :(

HypeHype bind groups



Not hardcoded!

Software Command Buffer, but 10x+ Faster…

- Initial design: Array of draw structs
  - Only contains the “metadata”
  - 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
- Can we do even better?
  - 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;
};
```

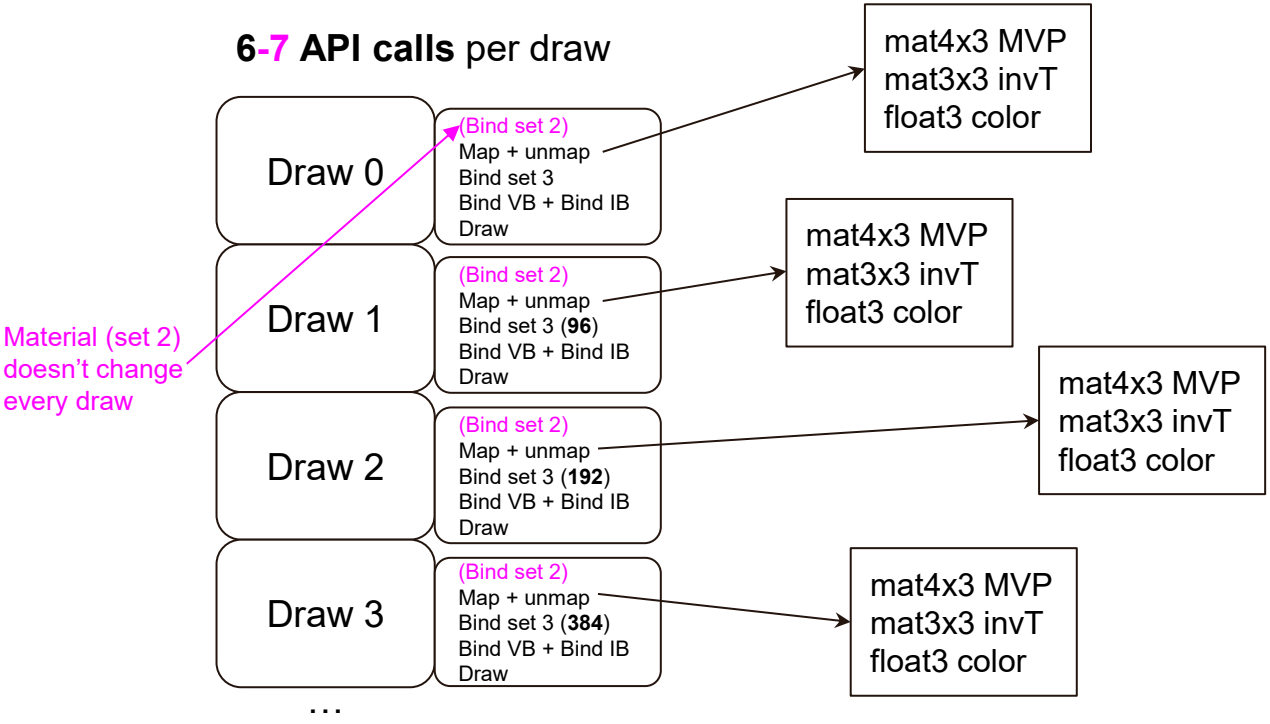
PSO with all render state

Draw Stream: Data Interface for Draw Calls

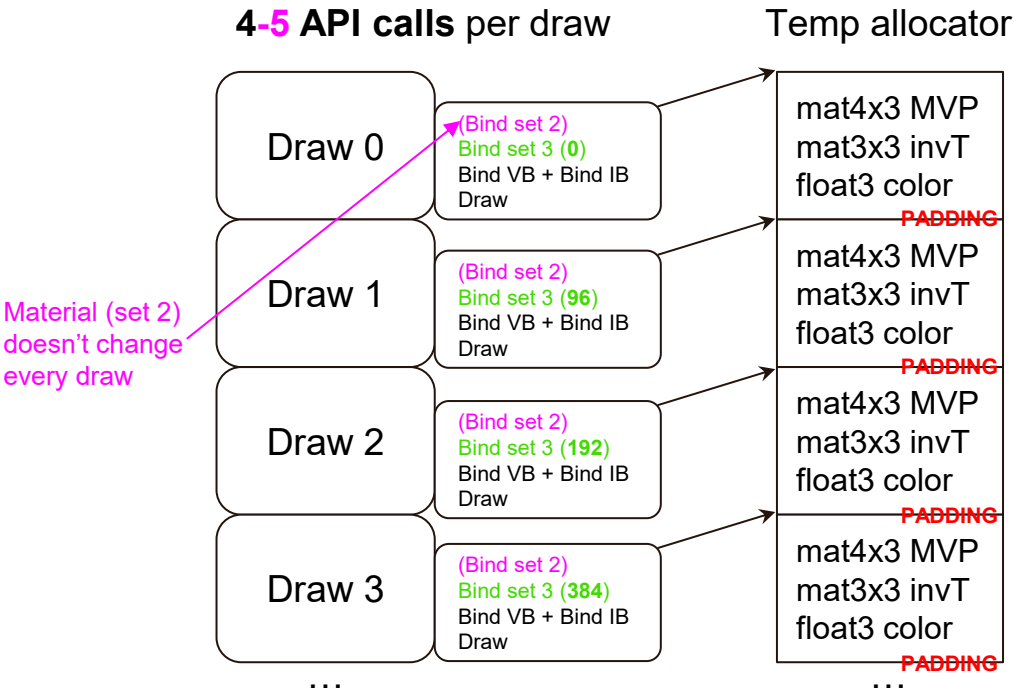

```
struct Draw
{
    Handle<Shader> shader;
    Handle<BindGroup> bindGroups[3];
    Handle<DynamicBuffers> dynamicBuffers;
    Handle<Buffer> indexBuffer;
    Handle<Buffer> vertexBuffers[3];
    uint32 indexOffset = 0;
    uint32 vertexOffset = 0;
    uint32 instanceOffset = 0;
    uint32 instanceCount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

Draw Call Performance: Baseline (Worst Case)



Draw Call Performance: Bump Alloc + Offset Bind

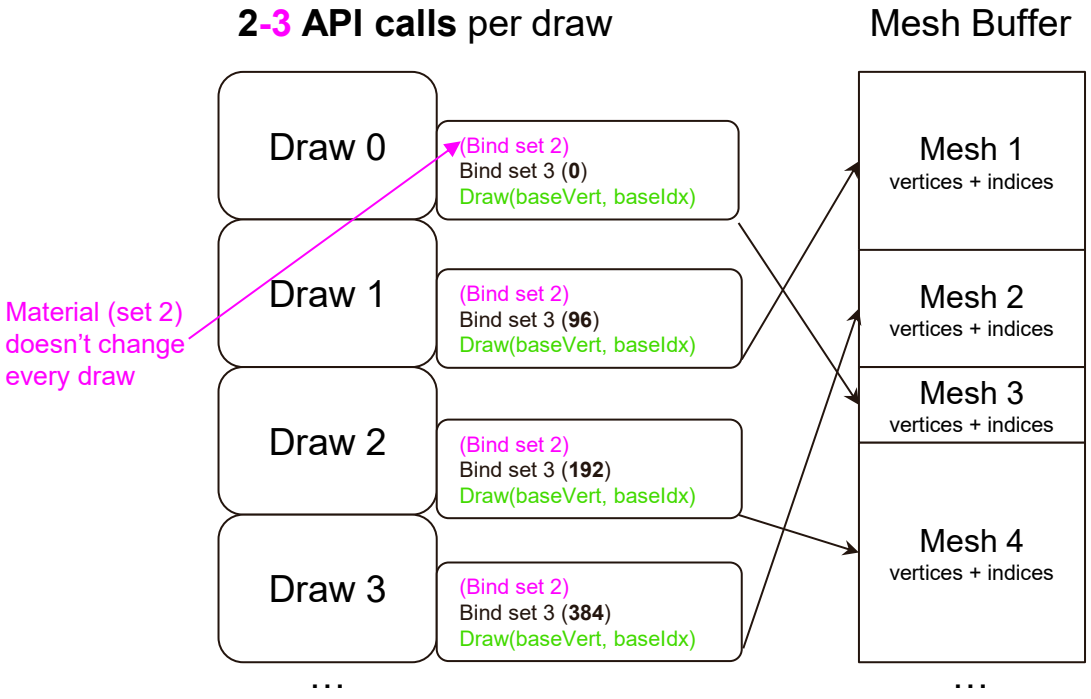


Throughput (CPU):

~3x versus old WebGL / GLES backend

No Vulkan results :(

Draw Call Performance: Pack Meshes

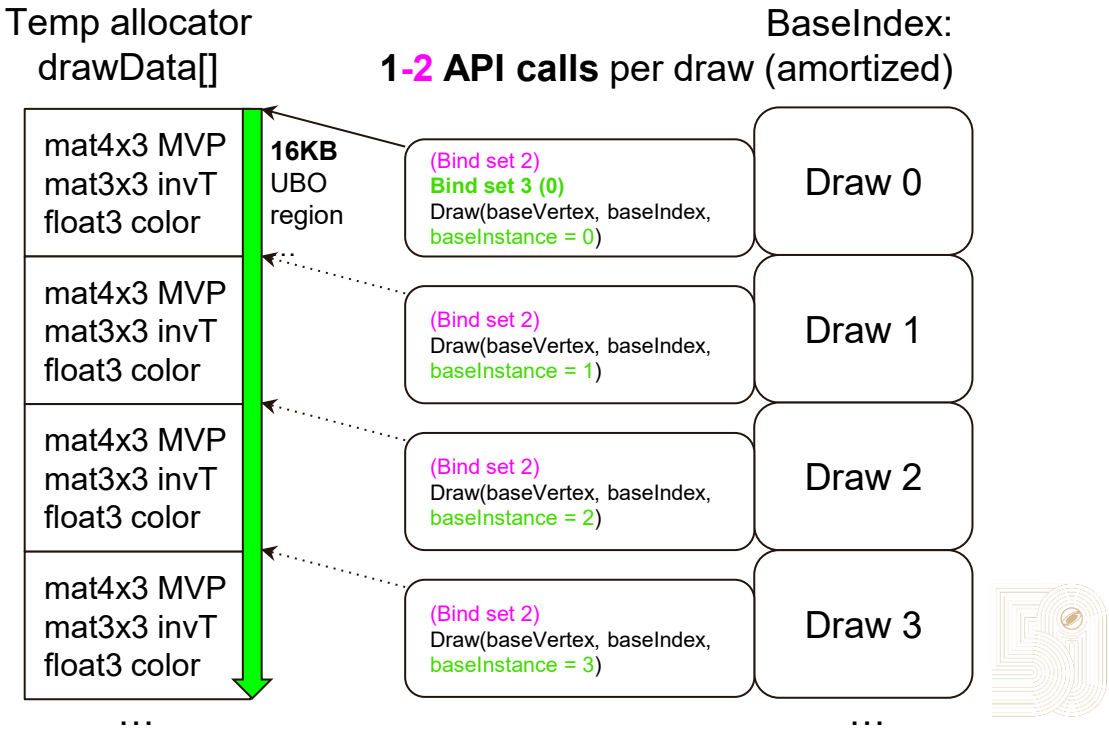


Throughput (CPU):

Nvidia: 2.17x
AMD: 1.75x
Adreno: 1.29x
Mali: 1.40x

Draw Call Performance: BaseInstance (FAILED)

- Instancing data layout (no padding)
  - 16KB UBO binding limit
  - Change offset every 16KB
    - Amortized over >100 draws
- Better shader codegen vs instancing
  - gl_InstanceID = dynamic offset
  - gl_BaseInstance = static offset
  - Scalar loads / fast UBO path
  - Saves vector registers / loads
- **Not supported in web / DX12 :(**
- **Mobile shader codegen issues**
  - **Often slower for the GPU :(**



10,000 draw calls (CPU time)

10,000 unique materials, 10,000 unique meshes

AMD	RDNA2 iGPU + 6800HS 4.7GHz	0.85ms
7y 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
Actual draw calls (no instancing)
No batching: 10,000 mesh and material changes
No GPU persistent scene data
~90% time spent in driver

References

[1] Various hardware performance numbers (Wikipedia):

- ARM Mali: https://en.wikipedia.org/wiki/Mali_(processor)
- Qualcomm Adreno: https://en.wikipedia.org/wiki/Adreno
- PowerVR:https://en.wikipedia.org/wiki/PowerVR
- Nintendo Switch: https://en.wikipedia.org/wiki/Nintendo_Switch
- Microsoft Xbox 360: https://en.wikipedia.org/wiki/Xbox_360
- Microsoft Xbox One: https://en.wikipedia.org/wiki/Xbox_One
- Sony PS3: https://en.wikipedia.org/wiki/PlayStation_3
- Sony PS4: https://en.wikipedia.org/wiki/PlayStation_4

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

[3] Karis, Stubbe, Wihlidal: Nanite, A Deep Dive, SIGGRAPH 2021: Advances in Real-Time Rendering in Games: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

References

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

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

[6] RAII, Wikipedia: https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

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

[8] C++20 designated initializers, cppreference.com: https://en.cppreference.com/w/cpp/language/aggregate_initialization

[9] Masmano, Ripoll, Crespo, Real: TLSF: A new memory allocator for real-time systems: http://www.gii.upv.es/tlsf/files/ecrts04_tlsf.pdf

[10] Aaltonen: OffsetAllocator, GitHub: https://github.com/sebbbi/OffsetAllocator