Hi, I am Sebastian Aaltonen. I have over 20 years of experience in graphics programming. In the past have been working at Ubisoft and Unity building their cross platform rendering technologies.

I joined HypeHype one year ago with a mission to rewrite their mobile rendering technology. Today I am going to be talking about the first milestone of that project: rewriting the low level graphics API and the platform backends.
HypeHype is a mobile game development platform. You create games directly on the touchscreen and upload them to our cloud server.

Gamers use a Tik Tok-style feed to browse the games. The games are instant loading. This is a big technical challenge. Both the game binary size and the loading code have to be highly optimized. To make the initial binary smaller, we store data in highly compressed form and also lean on streaming.

HypeHype has up to 8 player multiplayer. Multiplayer features and the player count will increase in the future, once our cloud game server infrastructure is deployed.

We have a full blown game editor inside the mobile app. Visual scripting system is used for writing the game logic. Players can spectate creators creating games, and multiple creators can collaboratively create games together at real time. It’s a bit like Google Docs, but for game creation. Test play is instant and all the spectators join multiplayer test session as players. This improves the iteration time drastically.

We of course have a full set of social features, including chat, leaderboards, replays and similar.
HypeHype is mainly targeting mobile devices and tablets. But we also have a web client and native PC and Mac applications.

I have a console development background at Ubisoft, so I like to compare mobile devices to the past console generations to understand them better.

Xbox 360 and PS3 are nowadays equal to low-mid tier mobile devices in GPU performance. This is excellent news, since these consoles offered the greatest visual jump we have seen between console generations so far: We got HD output resolution, and were able to implement proper HDR lighting pipelines, physically based material models and image post processing for the first time. All of that is possible today on mainstream mobile devices. And we can scale that down to bottom tier devices at 30 fps with upscaling.

When you look at the high end, you see latest 1000$+ phones reaching Xbox One and PS4 levels of performance already. However these phones run at higher native resolution and are thermally constrained, thus in reality, we can’t quite reach that generation of visual fidelity yet on mobile devices in real games. And we don’t even want to, since that would make the devices hot and drain their battery in couple of hours.
HypeHype games have been limited to simple visuals: Stylized untextured objects, simple gamma space lighting and tiny scenes with short view distance. This has been fine for simple hyper-casual games.

This is however a big limitation to the platform, so we started building a new renderer from the scratch one year ago. The visual fidelity target for the new renderer is to match the best looking Xbox 360 and PS3 games. We will be bringing full PBR pipeline with modern lighting, shadowing and post processing techniques. We will be targeting larger game worlds and longer draw distances to allow more game genres to be built properly on the platform.

This is all nice of course, but we have to be really careful about the performance cost of all these new improvements. We still want to be running HypeHype games at locked 60 fps on mid tier mobile phones without throttling the devices. This is a big concern for us and is the main reason why we are focusing heavily on performance in our new rendering architecture.
If you compare current mainstream phones with Xbox 360, you notice a lot of similarities.

Both designs have slow shared main memory. Bandwidth is the main limiting factor. Both designs also employ techniques to reduce the memory bandwidth usage. The most important one being on-chip storage for render targets. On Xbox 360 you had a 10MB EDRAM buffer for your whole render target. On mobile phones, you have a smaller on-chip tile memory. Both technologies solve similar problems. Overdraw doesn’t require extra memory bandwidth and Z-buffering and blending happens fully on chip. On mobile phones, you also have framebuffer fetch, allowing you to load back the previous pixel from the same render target location without a memory round trip. The newer Xbox One console also was equipped with read-write ESRAM allowing similar optimizations.

Since the main memory is slow, you want to avoid resolving render targets as much as possible. You want to minimize the amount of render passes. Doing multiple things at once is the key to good performance. Modern mobile phones also have framebuffer compression, to reduce the render target resolve and sampling bandwidth cost. This is a good addition, but doesn’t fully solve the problem. ASTC texture compression also helps. It offers better quality and smaller footprint than DXT5 back in the day.
Mobile phones also have double rate fp16 math. This helps, since you don’t want to
lean on memory lookups on bandwidth starved devices. And there’s now better lower
precision HDR framebuffer formats available.

But some old limitations still remain: Mobile GPUs are still designed around uniform
buffers. SSBO loads from dynamic addresses are still slow. If you are able to
scalarize your memory access patterns you hitting the performance sweet spot. This
limits the algorithms we are able to implement efficiently. Many mobile phones also
write vertex varyings to main memory, which costs significant amount of precious
bandwidth. Optimizing the size of the varyings is key for good performance on these
devices.
I was talking about GPU-driven renderer already 8 years ago at SIGGRAPH, we presented the core ideas such as the clustered rendering and the 2-pass occlusion culling which have become a de-facto standard nowadays.

Recently Nanite by Epic made GPU-driven rendering available for mainstream. They combine V-buffer, material classification, analytic derivatives and software rasterizer to make GPU-driven rendering robust enough for generic engine.

However, there’s still a lot of unsolved performance problems with GPU-driven rendering on mainstream mobile GPUs.

Mobile GPUs are not optimized yet for SSBO loads. AMD and Nvidia optimized their data paths couple of generations ago when they added ray-tracing. Ray-tracing access patterns are dynamic and you can’t lean on tiny on-chip buffers for vertex attributes anymore. We still need to wait for mobile GPUs with similar optimizations to become mainstream.

V-buffer requires you to run the vertex shader 3 times per pixel, and this includes fetching all the vertex attributes of these 3 vertices. You also need to fetch all the instance data and material data from dynamic location. This is over 20 non-uniform memory loads in the pixel shader. Mobile chips aren’t simply designed for this kinds of memory heavy workloads.
No current mobile GPU supports framebuffer compression for compute shader writes. Compute shader is the most efficient way to implement full screen material passes in deferred V-buffer shading. If you do that on mobile, you waste a lot of bandwidth.

64 bit atomics are commonly used in software rasterizers. You pack the Z value in high bits and payload in low bits and let the atomic resolve the closest surface. There’s no 64 bit atomic support in mobile GPUs. SampleGrad is also slow. ⅛ rate or even slower. Which makes deferred texturing with analytic gradients quite costly. And there support for wave intrinsics is spotty and you even have emulated groupshared memory on some low end devices.

As a result, traditional CPU-based rendering is still the best be for mainstream mobile phones today. We could do 10,000 draw calls on Xbox 360 back in the day at 60 fps. To reach that goal on mobile devices today, we need to write very well optimized rendering code.
Let’s talk about our roadmap.

We split the renderer rewrite into two stages. First we rewrote the low level gfx API and all the platform specific backend code. In order to run both the old and the new backends in tandem, we introduce a minimal wrapper with ifdefs, so that we can keep shipping the old rendering code and switch between the new and the old to compare them. We have already deleted 200 files of old rendering code and recently started tearing down the wrapper and replacing it with direct calls to the new platform API.

This presentation will focus on the low level platform API and the backends. I will be later talking about our new high level rendering code. Our design allows us to refactor these pieces completely independently of each other. I will come back to that topic later in the presentation.
The first thing we need to decide is the platform abstraction level. What code is platform specific and what code is platform independent.

Game engines generally limit platform specific code to the lowest levels of the stack. This minimizes the amount of code that is platform specific and reduces the implementation and maintenance cost. However some engine and renderer specifics tend to leak to the lowest level platform code.

If you look at mobile apps, the platforms specific code tends to reach a bit higher levels in the stack. For example the popular Google Flutter app framework is developed by multiple platform specific teams. They usually ship new features first on mobiles and later to desktops. Android and iOS don’t have full feature parity either. Their high level rendering code is different on desktop and mobile platforms, including Mac and iOS, even though they both use the same Metal API.

Many mobile apps bring the code separation even higher. There’s often completely separate iOS and Android team with their dedicated code bases. Most of the business logic in these apps tends to be running in a cloud server, which is of course shared and maintained by a third team.
HypeHype is a real time game engine, so we of course need to have all of our world state locally. Games must run identically on all devices and cross play must work across all devices. The old HypeHype gfx code base had duplicated shaders for Metal and some duplicated higher level code as well. This bloated the test matrix, added maintenance cost and made adding new features slow. This was the first thing I wanted to solve. The goal was to bring the platform API to even lower level compared to existing game engines.
We want as small amount of platform specific code as possible. This leads to a design where we tightly wrap the existing low level gfx APIs.

The design work started by cross referencing Vulkan, Metal and WebGPU docs. I was already familiar with all of these APIs, which made the work easier and less error prone.

When writing a wrapper, you first want to find the common set of features. These are often straightforward to wrap. The difficulties arise when there’s differences in the API design. Care must be taken in order to abstract these differences in a way that is performance optimal. We chose to use Metal 2.0 because it’s closer to Vulkan and WebGPU and it provides placement heaps, argument buffers and manual fences, allowing us to extract a bit more performance out of Apple devices too. We also support MoltenVK to make cross platform development easier, but we don’t ship it since our Metal 2.0 backend is roughly 40% faster for CPU.

In order to make the API more compact, we trim all deprecated stuff that nobody uses anymore. These things were failed experiments that never lived up to our expectations regarding performance. Vertex buffers are an interesting topic. At Ubisoft, we deprecated vertex buffers in our GPU-driven renderer already 8 years
ago. But at HypeHype we still support vertex buffers, since some mobile GPU shader compilers generate better code for them. Also we are still using WebGL2 in our web client, since WebGPU coverage is not yet good enough. I will likely be removing vertex buffers from the API in a few years.

Single set of shaders is crucial for tech artist productivity. We use modern open source tools such as SPIRV-Cross to cross compile our shaders to all target platforms.
Let’s talk about the design goals of this new platform API.

First, we want it to be a standalone library. Designed and maintained independently from the HypeHype engine. It needs to have a stable API that doesn’t change often.

I have seen a lot of graphics platform abstractions during my career and the problem in most abstractions is that user land concepts creep into the hardware API. Having a mesh and material in the platform code is the most common issue. This is problematic since mesh and material both have change pressure. Meshlets and bindless textures are the future. We don’t want to commit to a certain way of presenting them. Mesh can be simply represented as an index buffer binding + N vertex buffer bindings, and material can be represented as a bind group containing multiple texture descriptors and a buffer for value data.

Automatic the uniform handling might feel like a good idea in the beginning, but eventually you want to add stuff like geometry instancing and now you need to refactor your backend code to change the data layout. Or even worse, add a new fast path to complicate the API. And eventually you add a new fast path for GPU-driven rendering too, bloating the API further. In our design, the user land code is responsible for setting up all the data!
Zero extra API overhead is another crucial design core pillar for us. The platform interface should not add significant cost. It should be as easy to use as DX11, but always as efficient as hand written optimized DX12. A wrong solution is to copy the DX11 API as is. This way you end up emulating the DX11 driver in your code base, and rest assured that Nvidia and AMD does this better than your team. Thus your modern backends are slower than DX11. The reason for this is that you have too fine grained inputs, too fine grained render state, lots of shadow state and data copies. PSO and render state tracking and caching is a big performance drain, and slow software command buffer design usually adds to the cost.
So, we have very strict performance standards for our API, but at the same time we want it to be as easy to use as DX11. How can we achieve this?

We need a good process for designing the API.

The traditional way would be to spend months researching the API documentations and writing a big technical design document describing the new API in detail, splitting in into tasks and estimating the implementation time for each task for each backend.

The issue with this approach is that you lock in the design too early and it’s hard to change later. Small nitty gritty details matter a lot in platform specific graphics code. You can’t really understand the performance impact of all the corner cases without writing any code. Now you will notice those issues when there’s a lot of production ready code written. It’s too hard to justify a full rewrite of plans and code at this point.

Agile test driven development has the opposite problem. You are focusing on what you need in the next sprints. You implement small independent pieces of code that have full test coverage. The assumption is that once you put these pieces together, you have good architecture. But you didn’t even do any architecture design. More pieces equals more interfaces, equals more communication overhead. It is difficult to
reach optimal performance with this kind of programming practice. And it’s even more difficult to throw away lots of production ready code with full test coverage and lots of story points spent once you notice that the architecture needs a big overhaul to meet the performance goals.
Our solution for this problem is to use a highly iterative design process.

I start by writing mock user land code. I use my prior expertise and start writing my dream graphics code, assuming I have the perfect API. That API doesn’t exist yet, but I keep writing mock code until I am happy with it. I write code for creating all the resources I need for rendering, textures, shaders, buffers and so and then I write a small draw loop using these resources. The draw loop is called multiple times, with some resources mutated to implement animation. It’s important to design both dynamic and static data paths early.

Once I am happy with the first iteration of the user land code, I write a mock platform API for it. This is just a hollow API at this point. There’s no backend implementation. But it allows me to start using the compiler to do syntax checking and autocomplete. Now I can really start experimenting with the API to see how good it feels to use. I will of course refactor all the time when I find even slightest need for it. I add missing mock use cases and go through the Vulkan, Metal and WebGPU API docs to ensure that I have not missed anything important.

Then I will do a performance check for all the user land code. I have a good understanding how all the platform APIs work, so I think what kind of implementation
each API call would require in Vulkan and Metal and WebGPU backends. If the implementation is trivial then it's fine. If the implementation requires extra data copies, hash map lookups, memory allocations or other expensive operations, then I scrap that design and rewrite that part of the API to be more efficient. As you remember, our goal is to be as fast as hand optimized DX12 in every single case. We can't do that if our API doesn't map perfectly to the underlying hardware API.

Once we are happy with the performance of our mock code, we start implementing the backends. We of course notice nitty gritty details that we missed during this process and immediately refactor the mock code and the mock API when this happens. We don't write heavy test suite just yet, as that would slow down our iteration time. Instead we are leaning on Vulkan and Metal validation layers to provide us thousands of test cases for free. We hook the validation layer error callback to our automated tests to ensure our code keeps functioning when we refactor it.
The last topic about API design I wanted to discuss today is doing things at right frequency and granularity.

A big problem in rendering code tends to be that expensive operations are done at too high frequency. This also tends to add tracking cost to the hot draw loop.

Games have a lot of temporal coherency. You load the game world and you slowly mutate it every frame. Most of the data stays the same. Also the camera is most of the time moving slowly. Human brain needs temporal coherency between the frames to see smooth movement. This is great for us! We want to exploit it!

Let's take a look at all the stuff happening: Loading the game world and all the shader PSOs happens in the beginning. If you have a larger level, you also load textures, meshes and materials when you move around. Most of the objects are spawned in the beginning, but sections of the level might be spawned during streaming, enemies, loot, projectiles, etc are generally spawned throughout the game, but not that many every frame. The only really high frequency operation is culling all the objects and drawing the visible objects. The culling and draw loops are the most time sensitive loops in your whole code base.
I have highlighted problem cases with red. People tend to do processing related to these inside their hot draw loop.

Modifying material bindings is not common. How often do you replace the normal map of an already loaded material? How often do you change the shader that’s used to render an object? How often do you change the render state that is used to render an object? Pretty much never, except in some special effects. Animating object color and object transform are more common operations. A small subset of objects animate every frame. We only want to pay for these things when they happen. Not for every draw call.
Our solution for this problem is to fully separate all data modification from drawing. All the data is ready before the draw loop.

Pipeline state objects (PSO) should be built at application startup or at level load time. Building PSOs at runtime causes stuttering. In our philosophy shader variants are authored by coders and tech artists and hand optimized. There’s only a limited amount of them. This is similar to that id-Software does and provides very good performance.

We store the PSO handle directly to each object’s visual component. We don’t need hash map lookups to obtain it every frame.

We precreate all the bind groups (descriptor sets). Material descriptor sets contain all their textures and buffer for value data. We store the material bind group handle to objects visual component. This avoids a hashmap lookup and makes it possible to efficiently change the material bindings with a single Vulkan, Metal and WebGPU command.

Separating persistent and dynamic data is important. Persistent data is uploaded at startup and delta updated when changed. I had a talk about this topic two years ago
at REAC 2011. You can refer to that presentation if you want more information about that topic.

Dynamic data should be batch uploaded once per pass instead of using map/unmap per draw call. Global data should be separated from per-draw data to minimize the wasted bandwidth cost.

Resource synchronization is a big CPU cost in many engines. Our current solution is simple: When a render pass begins, we transition the render target to writable layout. At the end of the render pass, we transition it back to sampled texture layout. This way all the textures (both static and dynamic) are always in sampler readable layout. We never need to do per-draw call resource tracking at all. This saves significant amount of CPU cycles.
And now we are ready to talk about the implementation details.

A renderer needs textures, buffers, shaders and several other resource objects. We need a good way to store these objects and ensure they are safe to use.

The modern C++ practice would be to use smart pointers, reference counting and RAII (resource acquisition is initialization).

Frankly, these are too slow for us. Reference counted smart pointers tie the life time of a reference and the backing memory together. This results in a lot of small memory allocations. Memory allocations are expensive in current highly multithreaded systems. Allocations are also randomly scattered around the system memory, making data access patterns worse and increasing the cache misses. Copying reference counted smart pointers requires two atomics (add, sub), since we are in a multithreaded system. Ownership can be shared between threads.

There’s also safety issues. Ref counting makes object lifetime vague. Hard to reason about. It might die in any thread. RAII objects such as listeners cause destructors to have side effects. Example: Object ref count runs in another thread, destructor of listener de-registers it from an array. Another thread is just iterating that array.
CRASH! To avoid this crash, you have to protect part of the destructor with a mutex. This means that every time you delete an object, you need a mutex lock. This is very expensive. HypeHype is loading and unloading games rapidly in the feed. We can’t afford slow loading and tear down code!

Our solution for this problem (and most other problems too) is to use arrays!

There’s one big allocation containing all objects of the same type. Array index is a surprisingly nice data handle. If I have an array of textures, I can simply ask texture at index 4. Index is POD data. It’s trivial to copy around. I can pass it to worker threads safely too. As long as that thread doesn’t have access to the array, they can do nothing dangerous with the indices. This allows us to write culling and draw stream generation tasks with no safety concerns. These threads simply take array indices from one place and combine them to form draw calls. No access to the data arrays is needed at all.

But there’s a critical flaw: An array index doesn’t guarantee object lifetime. We of course reuse slots in our data arrays. Data could have died and slot reused…
To solve the problem, we need to replace our array indices with generational handles. Let’s discuss what this means.

A pool is similar to our data array. It is an typed array of objects, but now it also has an additional generation counter array. Generation counter tells how many times the slot has been reused. Counter is increased when the current data in that slot is freed.

We also have a freelist in the pool. A freelist is simply a linear array containing the indices of each free slots. It has stack semantics. When you allocate a new object, you pop a free index from the top. When you delete an object you push the index of the freed slot on top of the free list. These are both fast O(1) operations. When the freelist runs out, we double the size of the pool. This is safe as nobody is allowed to have direct pointer references to the data in the array. All references are done using handles.

The handle is just a POD struct. It contains the array index just like in the previous slide, but now we also have the generation counter next to it. This is in total 32 bit (for example 16+16 bit split) or 64 bits (32 + 32 bit), depending how many simultaneously active resources you require and how short are the lifetimes of the objects. In HypeHype we use 32 bit (16+16) handles for all graphics resources, this is enough for...
65536 resources of each particular type.

Pool offers a getter API, which takes the handle as a parameter. It reads pool’s generation counter array at handle index and compares it to the handle's generation counter. If they match, you get the data. If they don’t you get a null pointer.

This results in weak reference semantics. It’s completely safe to use stale handles. You just get a null pointer back. A null check is a predictable branch, and is almost free on modern CPUs. The branch prediction fails once when the handle is deleted. At that point you clean up yourself too. Weak references result in coding practice that doesn’t required callbacks. Callbacks require buffering or mutexes to avoid race conditions in multithreaded systems.
One of our main goals was to make the API as easy to use as DX11. This requires us to bundle auxiliary data in our data structs in the pools. In Vulkan the VkTexture handle doesn’t know anything about itself, which is annoying if you try to write your rendering code in pure Vulkan. We want our texture struct to know it’s size, format, data pointer for writing, allocator for deleting it and so forth.

This auxiliary data is required for low frequency tasks such as modifying the resource and deleting the resource. Since our design principle is to separate resource modifications from drawing, we are accessing this data only when the resource is modified or deleted. This means that putting the auxiliary data in the same struct as the data required in the hot draw loop is not cache efficient. The draw loop will load data to L1$ that is not used. I hate trade-offs between performance and usability.

Our solution for this problem is to use SoA layout inside the pools. We identify which data is required every frame in the hot draw loops and put that data in one struct and the remaining low frequency auxiliary data in another struct. The pool now has two data arrays instead of one. We can use the same array index in the handle to access either of the data arrays (or both). This way we only need to load the hot data to caches in the performance critical draw loop. The auxiliary data struct is only loaded at low frequency, solving our performance issue with L1$ cache utilization.
Now we have a good way to store and refer to graphics resources. The next topic is creating the resources.

Creating graphics resources in Vulkan and DX12 is cumbersome. You need to fill big structs that contain other big structs. Some of these structs also contain pointers to arrays of structs too. This makes it possible to shoot yourself in the foot with temporary object life times.

The most common existing solution for this problem is using builder pattern for resource descriptors: The builder object contains good default state for the descriptor. It offers an API to mutate itself to set all the fields you want to change. Once you are ready, you call build function to get the final descriptor struct. This is easy to use, but the codegen, especially in debug mode is far from perfect. At HypeHype we use debug mode a lot during development, so we want it to be fast too.

Our solution for this problem is to use C++20 designated struct initializers in combination with C++11 struct aggregate initialization. These two features in combination allow us to set default values to each struct in a trivial way. Look at the code example box below. If you want to override one of these defaults, you use the designated struct initializer syntax to override the values of named fields. The syntax
is super clean and codegen is perfect.

To solve the array data cleanly, we have to write our own span class. C++20 built-in span class doesn’t support initializer lists, because initializer lists have very short life time. They die immediately after the statement. It was too dangerous to allow putting initializer list inside a span in the generic case. However we use this only in a special case, and we have a solution for it: C++ const && function parameter only accepts temporary unnamed objects. C++ guarantees that temporary objects in function parameter list live long enough to finish the function call. This gives us enough guarantees to safely store initializer lists inside spans in our resource descriptor structs.
And this is how it looks in practice.

Let’s start with the left side: First we are creating a vertex buffer and a texture. The syntax here is nice and we are only declaring fields that differ from the struct default values.

If you look at the bottom left, you see us declaring a material. This is a bind group. The bind group has an array of textures: albedo, normal and properties. We are using initializer list here to provide the array. This makes the syntax super clean. And it’s worth noting that this array doesn’t require any heap allocations. The initializer list and the whole descriptor struct lives in the stack. It is never copied. We just pass a reference to it in the resource creation function call. This is as fast as raw DX12 or raw Vulkan.

One the right side you see us initializing a more complex resource. This looks a bit like json. We have named fields, arrays and fields and arrays inside each other with proper indentation. This is much more easier to write and read compared to raw DX12 and Vulkan. Yet still we pay no runtime cost. There’s still no memory allocations or data copies. Everything is pure stack data.
Now that we have a good way to create and store resources, we need to allocate GPU memory for them.

I prefer to use temporary memory whenever possible. Temporary memory doesn’t fragment your memory pools and allocating it is as simple as adding a number to a counter.

We use 128MB memory heaps in our bump allocator. The heaps are stored in a ring. If the bump allocator reaches the tail, we allocate a new heap block. Once we reach stable state, there’s no heap allocations happening at all. We create a platform specific buffer handle for each GPU heap we create. This buffer handle maps the whole heap. This way we don’t need to create platform specific buffer objects at runtime. Our buffer struct simply contains an heap index and an offset. It’s super efficient to construct them at runtime and pass to user.

As an extra optimization, we provide the user land a concrete bump allocator object. This has a function to allocate N bytes. This function inlines perfectly to the caller. It simply increments a counter and then tests whether the counter is over the heap block boundary. This check is a predictable branch. When the block runs out, we call a virtual function in th gfx API to obtain the new temp allocator block. This happens
only once for 128MB of data, making it highly efficient.

Since WebGPU doesn't yet have 100% coverage, we had to add WebGL2 support during the project. We use the same temp allocator abstraction for WebGL2. User land code doesn't need to know whether the returned pointer is a CPU pointer or a GPU pointer. In WebGL2 we use 8MB CPU side temp buffers and we copy these buffers using a single glBufferSubData at beginning of each render pass. This amortizes the cost of data updates, and is a big performance win over calling map/unmap per draw call.

We do persistent allocations only when needed, since persistent allocation is always much slower than temporary.

I implemented a two-level segregated fit algorithm. This is O(1) hard real time allocator. It uses a two level bitfield and two lzcnt instructions to find the bin. Bin size classes follow floating point distribution. This guarantees that overhead percentage is always small, independent of the size class. Delete operation is similar to allocate. But in addition you are checking neighbor pointers on both sides and merging empty memory regions. This is also O(1).

We use the same allocator for both Vulkan and Metal 2.0 (placement heaps). I open sourced the offset allocator. It can be used for sub-allocating GPU heaps or buffers, and generally anything that requires a contiguous range of elements (and doesn’t require CPU memory backing for embedded metadata).
One of the biggest differences our design has towards other renderers is user land bind groups (descriptor sets in Vulkan terminology).

The traditional way is to have separate bindings for each texture and buffer. Before drawing you set all bindings separately. The gfx backend has to combine these bindings in shader specific layout and create respective bind groups (WebGPU), descriptor sets (Vulkan), argument buffers (Metal) or descriptor tables (DX12). These bind group objects are GPU objects and are expensive to create. IHVs recommend you to precreate all GPU objects to avoid stalls and memory fragmentation issues.

The common workaround is to cache bind groups in a hash map in the backend. All bindings are hashed and a lookup is made. If the bind group exists, then it is reused instead of created. The problem with this approach is that hashing is expensive and hash map lookups randomize your memory access pattern. If you are rendering from multiple threads, you might even need to protect your bind group hash with a mutex, making it even more expensive.

Our solution is to bring bind groups directly to user land: User creates immutable bind groups ahead of time. For example a material bind group contains 5 textures and one uniform buffer (filled with value data). You get a handle, which you use to bind the
material.

Our draw call API exposes three bind group slots to the user land. Vulkan on Android and WebGPU mandate minimum of four bind group slots. Three first groups are exposed directly to use land code, matching the GLSL set=X semantics. This is easy for gfx programmers and tech artists to understand.

HypeHype higher level rendering code uses an convention to split data to bind groups by binding frequency. The first group has render pass global bindings (sun light, camera matrices, shadow maps, etc), the second slot has material bindings, third slot has shader specific bindings and the last slot is special.

We use the last slot in Vulkan and WebGPU for dynamic offset bound buffers. This is important for bump allocated temporary data, such as uniform buffers. Metal API doesn’t have similar offset update API for argument buffer buffer bindings. Instead we use Metal setBuffer API to set these dynamic buffers separately, and use setOffset API to change their offset. This provides an abstraction that uses the most efficient code paths on all platform APIs.

Push constants are emulated on some mobile GPUs. It’s faster to bump allocate your uniforms and just change the offset.
I already said that software command buffers are slow, yet we have one :)

This software command buffer is entirely different to the ones most people are familiar with. We don’t have any data in the software command buffer. We only have metadata pointing to already uploaded data. The metadata is also grouped, making it much smaller than individual bindings and individual state. This allows us to represent a draw call with only 64 bytes of data, which is just a single CPU cache line.

Our initial design was to use an array of draw structs. The draw struct contains handles to the shader (this is a resolved PSO variant including all render state), 3 user land bind groups, dynamic buffers (for temp allocated offset bound data), index and vertex buffers and some offsets. Offsets are needed because sub-allocating resources is usually a big performance win.

This 64 byte struct is already pretty good, but I wanted to improve it further. I analyzed the data and noticed that all fields are 32 bits. Optimized rendering uses sorted order to minimize the costly PSO and render state transitions. When rendering binned content we notice that most fields don’t change between draw calls. On average only 18 bytes change between the draws. We want to take advantage of this.
The idea is to store only the fields that change. This leads to a draw stream design.

We store a 32 bit bitmask in front of each draw call. This bit mask tells which fields in the draw struct have changed.

It’s the responsibility of the user land code to write data according to the stream data API contract. For this we have user land draw stream writer class. It contains a single draw struct describing the current state and a dirty mask. The draw stream writer provides an function for setting each field in the struct. These functions check whether the data value was changed. If yes, then set the corresponding dirty bit and write that field to the stream. After writing all fields the user calls draw, which simply writes the dirty bitmask in front of the data values.

The backend is simple: For each draw call it reads the dirty bitmask. Then it reads one uint32 from the stream for each bit and calls the corresponding platform API call to set that binding/state/value. The advantage of this design is that the backend doesn’t need any state filtering. We have already done that in the user land code. This is handy on platforms where secondary command buffers are not available or are slow (some Qualcomm GPUs disable optimizations with secondary command buffers). We can still generate draw stream using multiple worker threads and offload
the state filtering cost there. The render thread is as fast as possible, which is a big win since the platform API calls are slow on mobile devices. We also save roughly 3x bandwidth versus full blown 64 byte structs.
Let’s talk about draw call performance.

This slide represents a quite traditional DX11 and OpenGL style draw loop. For each draw call we call map/unmap and write uniforms separately. We also bind vertex buffer and index buffer and we bind our textures and buffers. Here I am simply binding set 2 (material) and set 3 following the conventions we have.

In total this is 6 to 7 API calls per draw call. 6 calls when the material doesn’t change and 7 otherwise. If we bin by material, then we can assume that the number is closer to 6 than 7.
This is using the temp allocator to bump allocate uniforms (and other dynamic data). Now we don’t need to call map/unmap per draw call. This reduces the API call count to 4-5 per draw call.

Map/unmap are surprisingly expensive calls. Our old GLES backend was uploading uniforms per draw call. The biggest difference in our new GLES3 backend (WebGL2) was the lack of map/unmap per draw and this change alone got around 3x CPU performance gain for us.

We didn’t implement per draw map/unmap to our new Vulkan backend (Vulkan supports persistent mapping), so I can’t unfortunately show you Vulkan numbers here.
The next optimization with big impact was packing meshes. We allocate big 128MB heap blocks and have one platform buffer handle for each. This makes it easy for us to sub-allocate meshes and simply change the base vertex and base index in each draw call to change the mesh.

This way we get rid of two API calls: set vertex buffer and set index buffer. We are down to 2-3 API calls per draw, which is very nice!

This optimization improved the CPU throughput on all devices. We saw biggest gains on desktop GPUs (close to 2x), but mobile GPUs also showed notable gains (30%-40%).
The last optimization I want to discuss is base instance.

Base instance drawing uses identical data layout as instancing uses. You use tightly packed array of draw data. On mobile uniform buffers have 16KB binding size limitation. The idea is to change the binding offset once per 16KB, amortizing the cost of rebinding the temp allocator buffer with a different offset. This cuts our API call count by 1 and we now have optimal amount of API calls: just the draw itself and the possible material bind group change. The draw call has base instance parameter, which we change to point to different slot in the uniform buffer data array.

So why not use instancing instead? Base instance results in better shader codegen on many platforms. The reason is that instance ID is dynamic offset. GPUs pack multiple instances in the same vertex wave, meaning that all data indexed by instance ID must use vector registers and vector loads. This is a lot of extra register bloat for loading 4x4 matrices and similar. Base instance on the other hand is a static per-draw offset. Every lane loads from the same location. This means that compilers can scalar code paths and/or use fast constant buffer hardware.

In practice however, we run into various issues. While the base instance codegen is perfect on PC, on mobile GPUs it's a mixed bag. Some drivers simply don’t optimize
this properly. Also this feature has poor coverage. DX12 doesn’t support base instance at all and WebGL and WebGPU also have no support. So I wouldn’t recommend this optimization, unless you are shipping only on desktop. Not worth it for mobiles.
Let’s take a look at the performance numbers.

This is using a single render thread. Ten thousand actual draw calls without any instancing tricks. Each draw call using an unique mesh and unique material. With bind groups and packed meshes it’s fast to change the material and the mesh.

I didn’t have time to implement GPU-persistent scene data yet in HypeHype. These numbers are with per-draw bump allocated uniforms, as described in the previous slides.

We are targeting 10k draw calls because that’s what we managed to push 15 years ago with Xbox 360 at 60 fps. And the results are impressive. Even the low end 99$ Android phones are close to hitting 60 fps in this stress test. In a real kit bashed UGC game scene we will have lots of repeated meshes and materials, allowing batching and reduction in gfx API call counts. We also intend to multithread the rendering.

On AMD’s modern integrated GPUs (found also in the Steam Deck and ROG Ally handheld) our renderer can push 10k draws in less than one millisecond. When multithreading is used, our renderer could push up to 1 million draw calls at 60 fps on modern AMD and Nvidia GPUs.
References

- 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


