



SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

THE PREMIER CONFERENCE
& EXHIBITION ON
COMPUTER GRAPHICS &
INTERACTIVE TECHNIQUES

Transitioning HypeHype to Physically Based Rendering

Sebastian Aaltonen, HypeHype



© 2024 SIGGRAPH. ALL RIGHTS RESERVED.

Hi, I am Sebastian Aaltonen. Today I am going to be talking about the work we have done in the past year to improve the visual fidelity of HypeHype's renderer.

What is HypeHype?



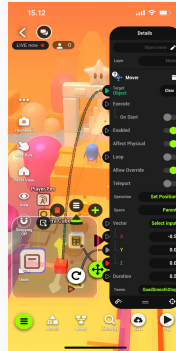
Feed

Instant loading games,
<10MB, replays



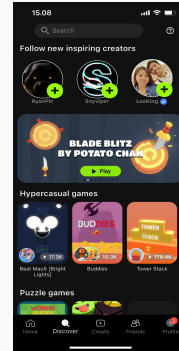
Multiplayer

8 players



Editor

Visual scripting, collaborative edit,
spectators, instant test play,
kitbashed content!



Social

Chat, replays,
leaderboards...

→ Real time (indirect) lighting, innovate on mesh/material storage

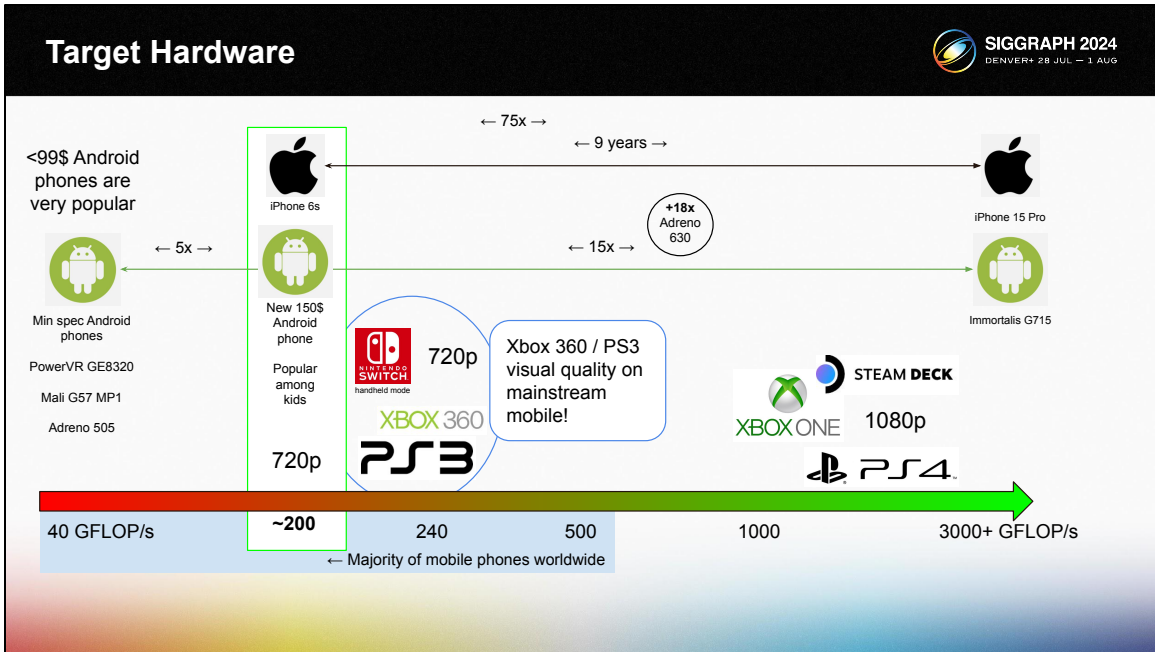
HypeHype is a social mobile game development platform. Games are created on the phone using a touchscreen user interface.

HypeHype has a feed similar to Tiktok for browsing though all the user created games. Games load instantly when you swipe the feed and a replay starts playing immediately. This is a familiar interface for social media video app users, and it gives you instant information about the game. When you swipe over a multiplayer game, you see an ongoing game session and you can instantly join it. There's no waiting.

HypeHype has a full game editor inside the application. Game logic is implemented using a node based visual scripting system. Game creation is also social. Spectators can watch creators and in multiplayer games, spectators immediately become players when the creator enters test play mode. Allowing very fast iteration time for multiplayer creation. Games are built from small objects, combining them. This is called kitbashing and results in dense soup of objects, which we must render efficiently. The most complex games have over 100,000 objects in them.

Instant loading games and limited 10MB storage make it impossible to store baked lighting to the game binary. We must employ a fully real time lighting solution. It must handle both direct and indirect lighting. Limited storage also means that we need to innovate on the mesh and material storage to make high fidelity content fit to the

budget.



HypeHype is targeting mainly touchscreen devices, such as tablets and phones. PC and Mac versions exist too, but most of our customers are using touchscreen devices.

I have a console background at Ubisoft. Let's take a look at popular phones today and compare them to previous generation consoles. Xbox 360, PS3 and Nintendo Switch are today roughly equal to mainstream Android phones. This was the first generation of consoles capable of running physically based lighting and a full post processing stack. This was a massive fidelity improvement back in the day and is today achievable on a cheap 150\$ Android phone. This is a really important market for a mobile gaming platform like HypeHype, since young audiences generally use cheap phones.

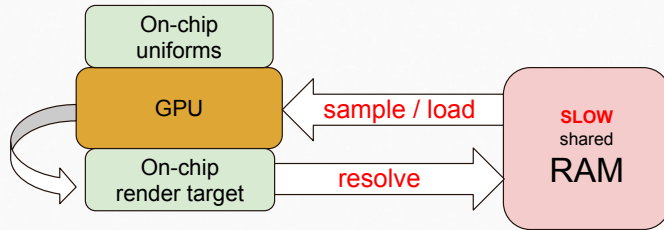
High end phones today already surpass PS4 theoretical ALU performance, but in practice can't sustain that kind of compute and bandwidth heavy workloads. It's better to run a lighter workload to avoid making users phones hot and draining their battery too fast.

HypeHype also needs to support older <99\$ Android phones. These phones are still super popular in Philippines and similar countries.

Mainstream Mobile GPUs Compared to Xbox 360

Similarities:

Z-buffer
Blend
Overdraw
Uniforms



BOTH

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

MOBILE

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

Differences:

VS→PS varyings

Xbox

mobile

VS→PS varyings

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

Still not great: Leans on preloaded uniforms. Indexed loads are slow. Compute is lacking: no DCC for compute writes, limited wave ops, possibly emulated groupshared memory, no 64 bit atomics, etc...

→ Traditional pipeline is optimal for performance

We are targeting Xbox 360 generation visuals, so let's compare a mainstream mobile phone and Xbox 360 to see how they compare to each other.

Xbox 360 used a unified memory design with a small EDRAM scratchpad for the GPU. This design is quite similar to mainstream mobile phones. We have slow shared main memory, and fast tiny on-chip storage for render target and Z-buffer. Blending and overdraw doesn't consume memory bandwidth. Render target must be resolved to main memory before it can be sampled in further passes. This resolve operation is expensive. Loading resolved uncompressed textures from memory is also expensive. To achieve best possible performance, you want to combine as many render passes as possible to avoid memory roundtrips.

Since memory loads are slow, these GPUs lean on uniform buffers for loading big data such as matrices. Xbox 360 and all mainstream mobile phones have some hardware mechanism for preloading uniform data to on-chip memory for fast access. These data paths are optimized for uniform address loads. Indexed memory loads with per-lane address are significantly slower and should be avoided as much as possible.

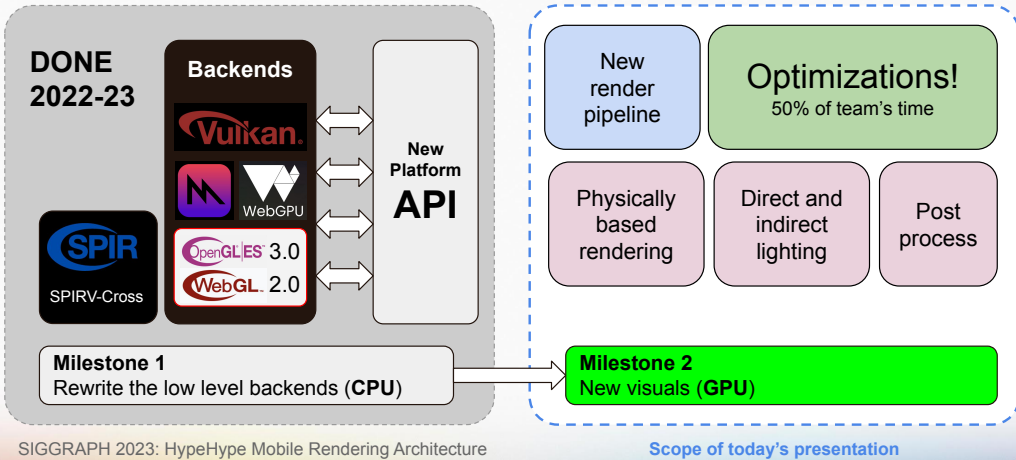
The biggest difference between mobile GPUs and Xbox 360 is tiled deferred rendering. Mobile GPUs have more efficient hidden surface removal, but need to write

vertex shader varyings to memory. Optimizing varying size is crucial for performance. 16 bit floating point helps a lot.

Today all mainstream mobile GPUs support ASTC texture compression, which provides better quality and data compression than DXT5 back in the day. And we have framebuffer compression, reducing the bandwidth cost of memory roundtrips between passes. And nowadays we also have robust support for 32 bit packed floating point formats for HDR render targets. These are all very nice improvements.

But modern PC and console workloads are still not a perfect fit for mobile: Preloaded uniforms are fine for classic workloads, but modern workloads need indexed loads and those are slow on mainstream Android GPUs. Compute shader performance is also lacking: There's no framebuffer compression for compute shader outputs, wave ops have poor coverage, some GPUs emulate groupshared memory and 64 bit atomics are missing. GPU-driven rendering is not a great for mainstream phones today. Traditional Xbox 360 techniques perform better.

Scope of this Presentation



SIGGRAPH 2023: HypeHype Mobile Rendering Architecture

Scope of today's presentation

Last year at SIGGRAPH I was talking about our graphics backend rewrite. All platform specific code was replaced with new code. We built an extremely efficient thin platform API which wraps Metal, Vulkan and WebGPU and built a new shader system leaning on SPIRV-Cross.

Today I am going to talk about our visual side. The focus will be on GPU side. I am going to be talking about our new render pipeline, implementing physically based rendering and modern post processing.

The biggest topic I want to discuss, again this year, is performance. Our team spent around 50% of our time optimizing code.

- Optimize for the right target
 - 80% of our customers have <200\$ phones. Kids have old/cheap phones
 - **Target:** 150\$ Android (~Xbox 360) needs to run full pipeline at **60 fps** (playability is the king)
 - Compromise on old <99\$ phones: 30 fps + 2x2 lower res rendering → 8x reduction in pixels
 - Creators also have mainstream phones! They can't author special high end content!
- Performance dictates the feature set
 - Can't do all we want: Learn to say no!
 - Time boxing helps during planning: Everything must fit in 16.6ms budget!
 - Prefer techniques with fixed cost. UGC kitbashed content is unpredictable
 - Optimize to fit more features
 - **Preferred:** Shader code optimization = free performance!
 - Simplify techniques, trade-off quality, add limitations
 - **Last resort:** Cut features if they cost too much
- 50% of rendering team's time spent optimizing + profiling!

When you are targeting mainstream mobile phones, performance is a crucial aspect.

We want to ensure that we are optimizing for the right target. To ensure this, we gather a lot of analytics data. 80% of our customers use cheap Android phones. We want to make sure that HypeHype runs flawlessly on these phones. Running at 60 fps is important for us, since game play is the king. Visuals should never make game play worse.

Since there's over 5x GPU performance difference between current 150\$ mainstream Android phone and older <99\$ phones, we have to compromise to improve our reach on some markets. We detect phone GPU performance at startup. If GPU is slow, we drop internal rendering resolution to 2x2 lower and lock frame rate to 30 fps.

It's important to notice that most creators are also using mainstream mobile phones. They can't author or test special high end content. Content has to scale automatically to high end phones.

The performance of our code dictates how many graphics features we can implement. Important to learn to say no when artists and game designers ask for new features. Time boxing helped during initial planning. We have already implemented Xbox 360 and PS3 rendering technology back in the day. This knowledge helps in time boxing

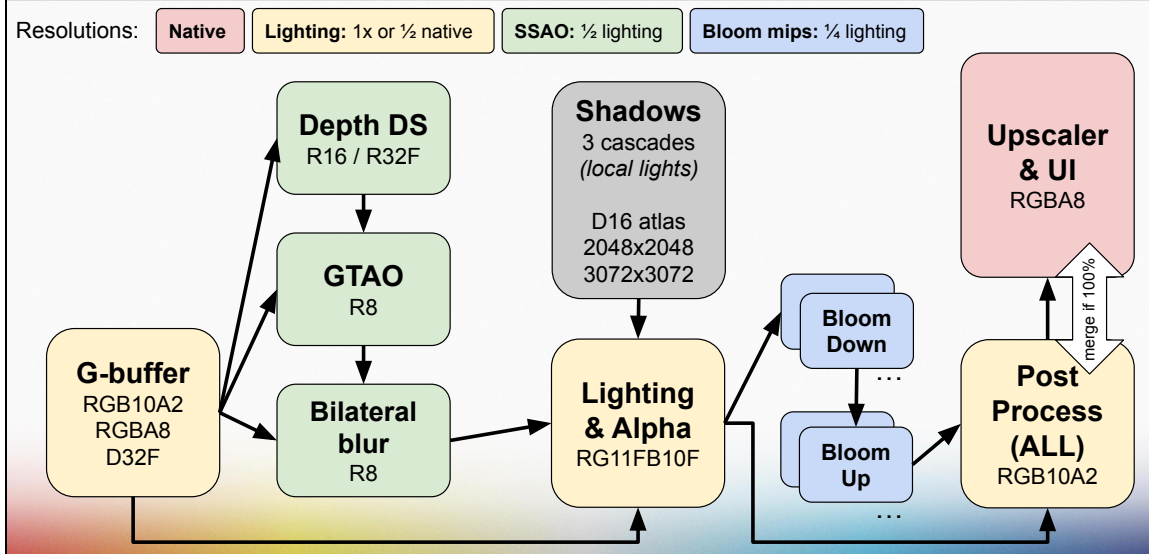
our initial feature set.

UGC kitbashed content is highly unpredictable and we can't test it or quality control it. It's preferable to aim for techniques that have fixed cost independent of the scene complexity. When choosing between two techniques, prefer techniques with slightly higher base cost, but lower added cost based on scene complexity.

When adding new features, we need to free performance to fit the new feature. Preferred way is to optimize existing code. This is free performance. Today our existing code is already very well optimized, so we have to trade-off with quality and add limitations to techniques to keep their cost lower. New features that cost too much can't be shipped. The last resort is to cut the feature, preferably early in the production.

Our rendering team has spent over 50% of our time during this year analyzing, profiling and optimizing.

The New Render Pipeline



Let's talk about our new rendering pipeline.

Each box in this diagram represents one render pass. We have tried to minimize the number of render passes to minimize the memory roundtrips which are expensive on mobile.

We have a deferred rendering pipeline, so we start by rendering a compact 64 bit G-buffer.

Then we downsample and linearize the scene Z-buffer. Linear Z buffer is 16 bit to save bandwidth. SSAO and bilateral blur passes consume linear Z and produce 8 bit SSAO contribution.

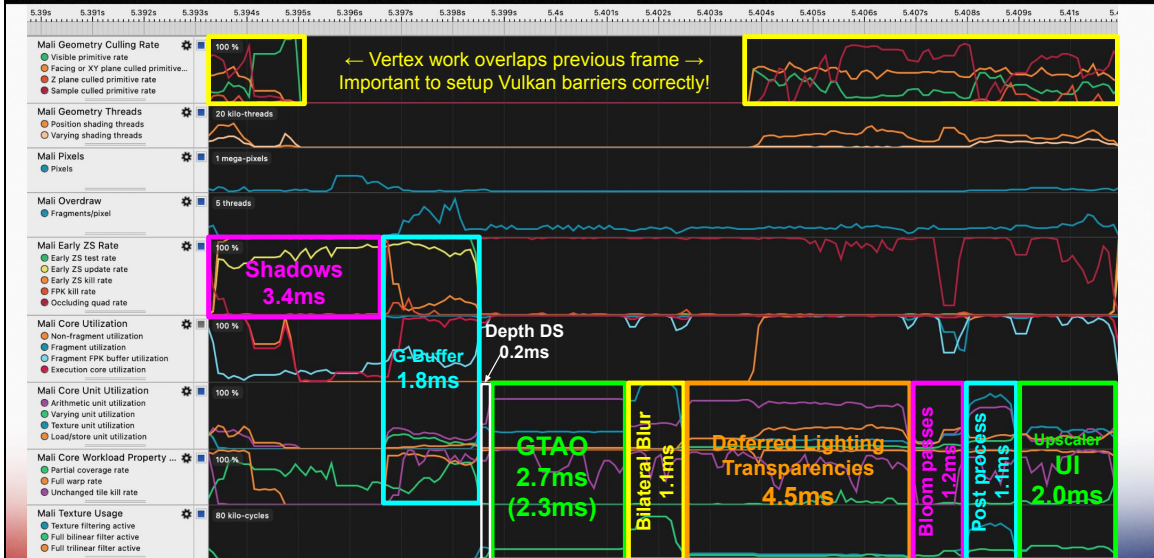
Shadow map pass follows. This is a single pass cascaded shadow approach. All shadow maps are in the same atlas, we change the viewport rectangle for each shadow cascade to fill the atlas. This way we only need one render pass, and there's no stalls between the cascades. Our shadow map is 16 bit to save bandwidth.

Deferred lighting pass consumes the shadow map, SSAO and G-buffer to provide a lit HDR image in 32 bit packed float format. Transparencies and particles are rendered on top of the lit scene in the same render pass.

Bloom passes start at quarter resolution, and go down the mip chain to blur recursively and up again to combine the results. This classic approach is a great fit for mainstream phones, but it starts to have occupancy issues already on high end phones.

Post processing uses a single pass approach to reduce memory traffic. All post effects are implemented in a single shader. Post processed image is then upscaled to native resolution and UI is rendered on top in the same render pass. On high end phones at 100% render scale, we render post process directly at start of the UI pass at native resolution to avoid having an extra render pass.

3 year old 99\$ phone timings (Mali G57 MP2) - 60 fps!



This is a profiler capture (ARM Streamline) on a 3 year old 99\$ Android phone. This is our min spec phone and it has a GPU that's around 5x slower than iPhone 6s (9 year old flagship).

As you can see, we manage to (barely) hit 60 fps on this device with the full render pipeline.

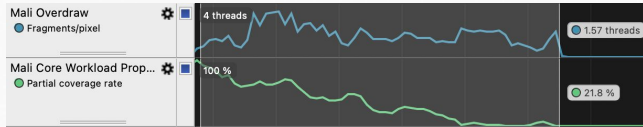
The first thing you see is vertex processing in the top. Our vertex processing overlaps the previous frame to avoid stalling the pixel pipeline. On low end phone the vertex processing can be active for around 50% of the frame, making overlap super important. In Vulkan it's crucial to setup barriers to allow vertex work overlapping previous render passes. You will lose around 15% of GPU performance if you fail to do that.

The most important thing to see here is that Shadows and G-buffer passes. These are the only passes that have varying cost. Remaining frame is fixed cost full screen render passes. This is super important for kitbashed user generated content. We want to avoid frame time fluctuation at all costs.

Deferred Rendering: G-buffer

- Motivation

- SSAO requires depth buffer
- Dense kitbashed geometry
 - Z-prepass doubles draw call count and triangle count 😞
 - Poor quad utilization in forward PBR lighting pass (+60% pixel waves) → **more fluctuation!**
 - **Solution:** Deferred minimizes geometry based fluctuation. Fixed pixel wave count in lighting!
- Future: Optimized light & shadow culling
 - Z-buffer = all visible surfaces = all shadow and light receivers



- Compact G-buffer

- **G-buffer A: RGB10A2**
 - Octahedral normal: 10+10 bits
 - Material ID: 10 bits – 1024 material array (lighting pass) → deferred texturing
 - Mode: 2 bits
- **G-buffer B: RGBA8** – Mode specific data (often 0,0,0,0 → DCC)

0.5ms faster!

G-buffer
RGB10A2
RGBA8
D32F

I will now talk about all our render passes: Deferred rendering will be the first topic.

Mobile games tend to use forward rendering to save memory bandwidth. But forward rendering has issues with modern rendering. We need depth buffer before lighting for SSAO. Z-prepass is extremely expensive with dense kitbashed geometry. It would double our draw call count and double our triangle count. We simply can't afford that. Additionally, Z-prepass doesn't help mobile TBDR renderers much at all, due to their TBDR architecture.

We also noticed that our geometry tends to be quite dense. LODs don't work that well with kitbashed content, as we can't optimize across object boundaries. We measured on average 60% extra pixel shader waves due to quad overdraw. This pixel shader is heavy as it implements our new physically based shading, lighting and shadow sampling techniques. With deferred lighting, we have fixed amount of lighting waves, one thread per pixel. Independent of the scene complexity. We don't want to pay for lighting overdraw.

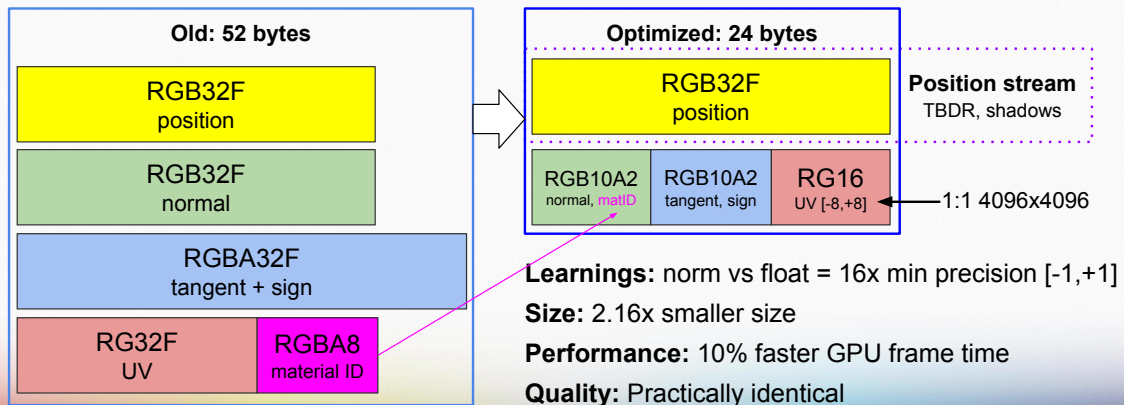
Next we need to solve the G-buffer bandwidth cost. We really wanted to fit the G-buffer two 32 bit render targets. The first G-buffer uses a 10 bit RGB format, we store octahedral normal in the two first channels and 10 bit material ID in the third. This way we can support up to 1024 materials, which is fine for small mobile games.

The 2 bit alpha channel serves as mode selector (4 modes). Our secondary G-buffer stores mode specific data. Mode 0 does deferred texturing and doesn't even need the secondary G-buffer. This compresses very well with framebuffer compression.

Packed vertex format

Motivation: Saves storage and bandwidth

Goals: Cheap decode, fp32 vec3 position data for editor and physics, needs to handle 4 material slots



HypeHype had a 52 byte fp32 vertex format. We wanted to save storage and bandwidth cost, so we designed a new format. The main goal was to make the format as small as possible, but without significant ALU cost. We also needed unpacked fp32 vec3 position data for physics and editor.

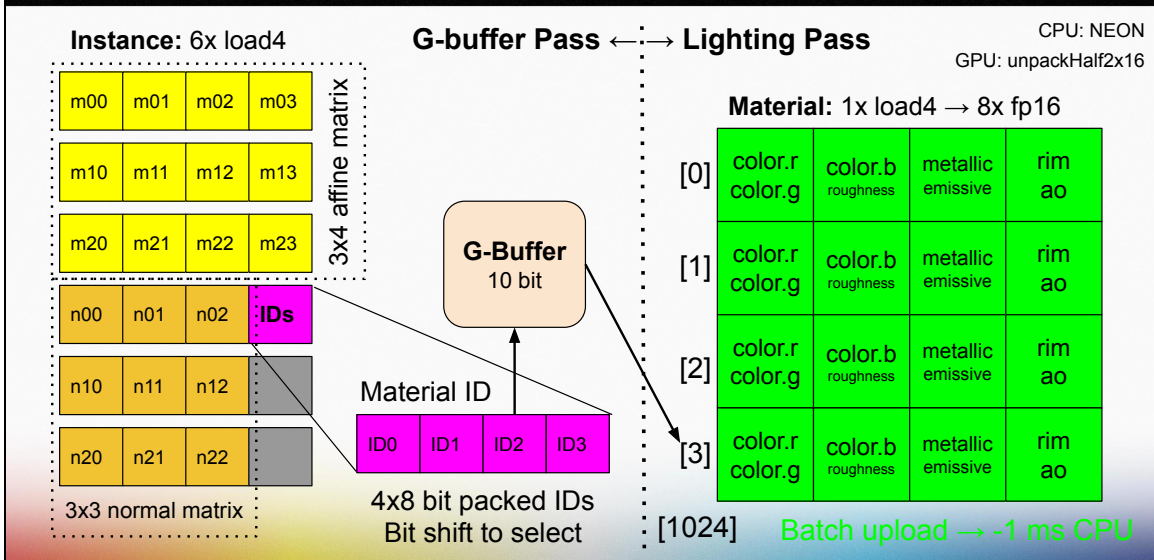
HypeHype's meshes support up to 4 materials. Each triangle belongs to one material. This was implemented using RGBA8 vertex colors. Color was decoded in vertex shader and we got a number between 0 and 3 as result. In the new format we store the local material index directly as a 2 bit value. 10 bit RGB values are used to store normal and tangent and their 2 bit alpha channels are used to store material ID and bitangent sign. Bitangent is calculated using cross product as usual. We store the UV as 16 bit normalized value scaled to [-8,+8] range. This supports up to 16x UV repeat without any tricks. If UVs repeat more, our mesh preprocessor can add extra UV seam to support unlimited tiling.

Position data is stored in a separate position stream. This is beneficial on mobile TBDR architectures as they run a simplified position-only vertex shader for binning. Separate position data improved cache utilization there. Similarly separate position stream helps with shadow map rendering.

Packed vertex format provided us on average 10% GPU frame time improvement with

practically identical quality. 2.16x smaller vertex size saves memory and game binary size, which are both crucial for our platform.

Instancing and Materials – 0.5ms per indexed load!



The next topic is geometry instancing and material data.

I noticed quite early during prototyping that indexed memory loads are very slow on older low end mobile phones. On Mali G57 MP2, I measured that loading a single 128 bit float4 from programmable per-lane address costs 0.5ms in a native resolution full screen pixel shader pass. HypeHype's original main forward render pixel shader was doing five of them. That's 2.5ms. Of course we can hide this memory cost by adding ALU, but it can easily become the bottleneck.

Geometry instancing is a nice way to reduce CPU and driver cost, but it always adds some GPU cost. Majority of this extra cost is caused by fetching matrix and material data using indexed loads instead of constant address loads. Each indexed load has to be executed per pixel, there's no preloading to on-chip storage.

Our instancing vertex shader data layout is packed in 6x float4s. 6 memory loads give us the object matrix, normal matrix and material IDs required for shading. Object matrix is a 4x3 matrix, which saves 25% loads (and registers) compared to unoptimized 4x4 matrices. Normal matrix is a 3x3 matrix. We store it as 4x3 due to alignment, but reuse the padding for various purposes, including material IDs.


In one 32 bit integer we pack four 8 bit material IDs. This maps our local mesh

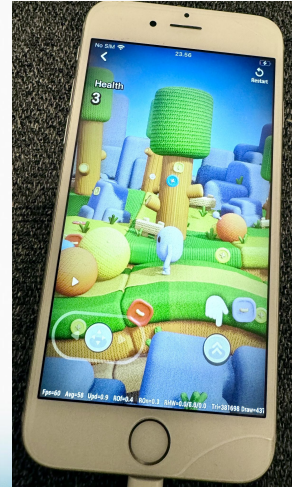
specific 2 bit material index to a global material index. Selecting the correct material is simply implemented as bit shift and mask. Which is extremely fast. We don't need an additional indexed load to select the material index. The resulting material index is stored in the G-buffer.

The lighting pass reads the material index in the G-buffer and indexes the global material data array accordingly. We bitpack eight 16 bit floats inside a single 128 bit uint4. This way we can load the whole material using a single indexed load. GLSL has an intrinsic for unpacking two half floats from one integer. This is free on many GPUs, as they store 16 bit values in upper/lower half of a 32 bit register.

On CPU side we use NEON vector instructions to convert our material data to fp16 and write it to GPU once a frame. Our draw calls don't do any material processing anymore. This saved around 1ms of CPU time in complex scenes.

Physically Based Rendering (PBR)

- Improved visual fidelity
 - An iterative path towards Pixar quality visuals
 - Easy to validate against DCC tools and ground truth
- Asset library (cloud server)
 - Old per-object lighting/ambient worked only in one environment
 - User had to tweak each material by hand to make it fit to their game
 - PBR assets work in all lighting conditions. Drag and drop
- Visual styles and remixing 
 - Remixing games is a core feature of HypeHype.
 - Visual remixing should be fun and easy!
- Future: Content portability across platforms
 - Internet asset libraries, GLTF2, AI generation...



Our initial main goal for Physically Based Rendering (PBR) pipeline was to improve the visual fidelity. People wanted to start making content that looked more realistic and could bring more mature audiences to the platform. But it's important to note that PBR is not just for photorealistic graphics, it also works fine for cartoon-style graphics, as Pixar and Disney and others have proven multiple times in the past decade.

PBR helps artist workflows and iteration time. DCC tools render the same PBR image as our engine. Artists no longer need to import their assets to engine to see their appearance in games. We did several validation passes against popular DCC tools and managed to match the visuals very closely. Having a ground truth comparison point during development was very useful.

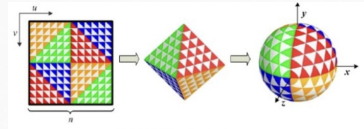
HypeHype has an asset library in our cloud server. This library is shared between all games, and users can import new assets to it. In our old ad-hoc renderer the light and ambient colors were defined in each material. For example if you wanted to make a sunset scene, you would make the material ambient colors yellow. This made it cumbersome to load these same assets into another scene. PBR solves the content portability issue, because objects, materials and the lighting environment are fully separated concepts. Objects and materials simply look correct in all lighting environment. Which made it much easier to share content between games.

We also wanted to add quick visual remixing features similar to video platforms such as Tiktok. Their video filters are super popular. Our visual styles include post process effects, direct lighting, ambient lighting, fog and background settings. Users can easily remix existing content by changing the visual style or they can edit a theme and modify each separate setting manually.

In the future we want to improve content portability across platforms. There's lots of assets in the internet and AI will generate more quickly. GLTF2 and similar formats are important. All third party tools use PBR so we need to be compatible.

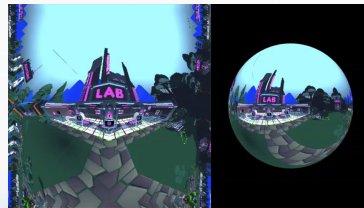
- PBR lighting

- Basic stuff: D = GGX [Walter], G = Smith, F = Schlick
- 100% 16 bit float (2x rate)
 - GGX with Lagrange's identity [Filament]
 - Clamp roughness (underflow), clamp output (overflow)
 - 16 bit float issues randomly surface → fix



- Indirect lighting: Oct-maps [Praun]

- Similar to cubemap but 8 faces, HW filter with borders
- Advantages
 - PowerVR = ¼ rate cubemap (explicit mip) 🤦
 - Process all faces in single pass, cubemap needs 6 passes
 - Supports standard texture atlasing (mix sizes)



Our PBR lighting pipeline is built from standard primitives on purpose. This is to ensure compatibility. D = GGX, G = Smith, F = Schlick.

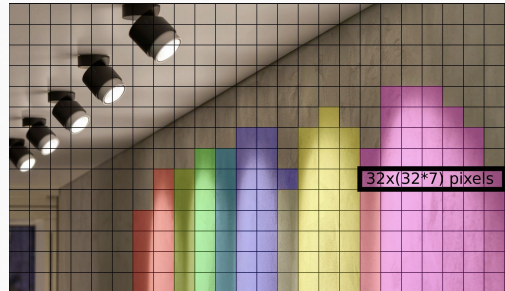
Since the PBR lighting math is complex, we spent a lot of time optimizing it. Running all PBR math as fp16 is important on mobile as fp16 runs at double rate. Porting the lighting math to fp16 was surprisingly easy. The biggest issue was the GGX formula, which had precision issues at highlights. We borrowed Filament's idea of using Lagrange's identity in the GGX formula to make it more stable for low precision floating point execution. This added a few instructions (cross product), but allowed us to keep all lighting math in fp16 without any conversions. We had to clamp roughness and HDR output to avoid underflows and overflows, but this wasn't a big issue. fp32 lighting with fp16 render target already requires clamping.

During the production we ran into various fp16 precision issues. fp16 pipeline requires more testing and more tweaking during the whole project to stay in good shape.

For indirect lighting we use oct-maps. These are similar to cube maps, but can be packed in NxN textures. HW filtering is possible if you add a seam on the outer edges. Advantage of this is faster execution on old PowerVR GPUs. Those had ¼ rate cubemap filtering with explicit mip level. Since we run a whole game editor in the device and we generate these oct-maps at level load, we also cared about oct-map

generation, lighting and convolution performance. With oct-map we can run all these steps in a single render pass. Cubemap would require 6 passes, one per face. It's also possible to atlas oct-maps in standard 2d texture atlases, which matters in WebGL2 (it doesn't support cubemap arrays).

- **CPU:** Bin lights to tiles
 - 32x32 tiles
 - 64 bit visible lights bitfield per tile
 - 600x720 → 50x23 tiles → 9KB (<L1\$)
- **Rendering**
 - Render all tiles with same lights in single draw call
 - Bump allocate light data to a big UBO (offset binding)
- **Shader**
 - Different shader permutations for each light count & type
 - Fastest possible UBO access (no dynamic indexing)
 - Multipass if light count > limit (on-chip TBDR). Limits register bloat and permutations
 - Early out: Light source Z min/max test in shader



Let's talk about our local light system, which is shipping later in the fall.

We are binning lights to tiles on CPU side to 32x32 tiles. 64 bit light mask per tile. CPU is doing software raster to fill the tile map. The whole tile map data fits in CPU L1\$ so this is highly efficiently done using a single CPU thread.

Rendering: Render all tiles with the same light set (same bitmask) at once in a single draw call. Bump allocate new tile set lights to UBO between draw calls, and change UBO binding start offset.

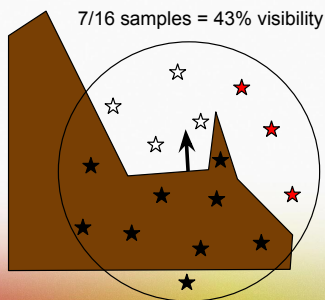
There's one shader permutation for each supported light count & type permutation. This way we can read light data from fixed UBO address, avoiding slow indexed load paths. Let's say we compile lighting shader permutations for up to 8 point and 8 spot light sources. If there's more lights than 8 in a tile, then we run remaining lights in the next draw call using additive blending. TBDR architectures to blending in tile memory, so this doesn't cost any extra memory bandwidth for render target writes & reads. This also limits the register bloat in the unrolled lighting shader. Complex shaders can be problematic for low end devices.

There's a Z-early out min/max stored for each light. If the pixel is not in these bounds, then the light is skipped. It's common that all threads coherently skip the same light

sources (behind a wall for example). In the future we do compute pass to cull lights based on low res min/max Z-buffer. This is one of the reasons we want Z-buffer in memory before the lighting.

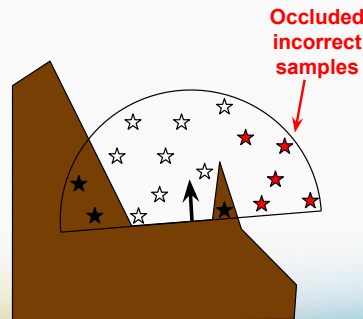
Crytek [Mittring]

- Doesn't use normal
- Uniform sampling pattern
- **1 bit result per sample**



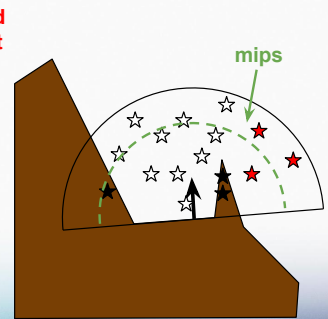
Use Normal

- Mirror samples below normal
- Doubles sample efficiency
- Removes halo on outer edges



Better Sampling + Mips

- Take cosine lobe into account
- Similar to importance sampling
- Z-buffer mips (longer distance)



Since we can't bake indirect lighting, we need a runtime solution for it. Ambient occlusion is a great starting point.

I ended up implementing many different SSAO algorithms before I found a solution that suited our needs. Let's go though a brief history of SSAO algorithms next.

Crytek popularized SSAO in video games. Their original technique was sampling N 3d points randomly distributed around the surface. For each sample they projected it in NDC and compared the point Z value and the Z-buffer value at the point location. If the sampling point was closer, that sample was considered visible, otherwise it was considered hidden. This gives us a single bit of data per each texture sampling operation. Then we count the number of visible bits and divide by sample count to get our visibility approximation.

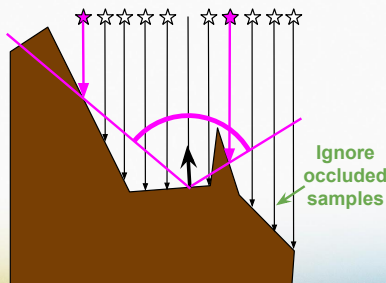
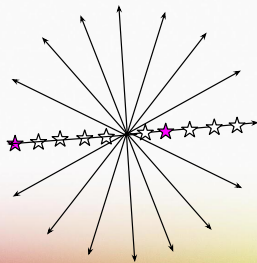
People quickly noticed issues in this original algorithm. The first improvement was to use pixel's normal vector. G-buffer already contains the normal vector, so we can simply use that. Samples below the normal vector (negative dot product) are mirrored on the top hemisphere. This both improves the quality and performance. Halos caused by sampling outer edges beyond the surface was fixed and we got double efficiency per sample.

Later various authors further improved this technique by improving sampling pattern. More samples closer to normal direction resemble cosine lobe importance sampling, and provide better approximation. Mip maps were added to Z-buffer to improve memory access pattern for far away samples.

Still none of these improvements could fix the issue of counting visible samples that are occluded from the surface's point of view. Narrow surfaces leaked light through them.

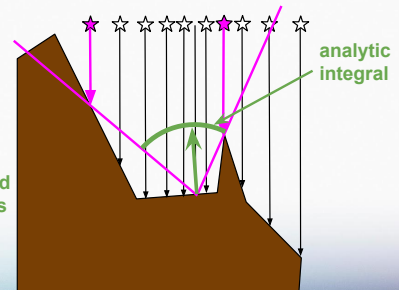
HBAO [Bavoil]

- Trace N horizon lines
- Step along line (linear steps)
- **Sample = height**
- Take max horizon angle
- AO = opening angle / directions



GTAO [Jimenez]

- x^2 line sample distribution
- AO = analytic integral (cosine lobe)
- Optimization: Both line sides at the same time (negate direction)
- Z-buffer mips

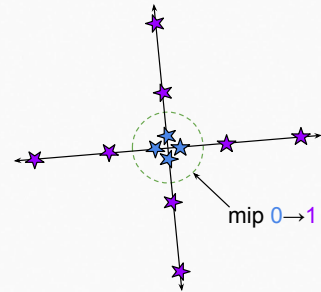


Horizon based ambient occlusion (HBAO) solved the light leaking issues. It traces N horizon lines in screen space instead of relying on random sampling. Each line is stepped with N linear steps. For each step the Z-buffer is sampled. Opening angle is calculated using the Z-buffer depth. And minimum (most restrictive) angle is used. This is a big efficiency improvement over the 1 bit per sample in the previous slide. Now we get a full angle from each sample. Once the line trace is done, we simply use the opening angle as our visibility estimate.

The most advanced HBAO-style algorithm is called GTAO (ground truth ambient occlusion). They improved the algorithm in many ways. Fixed size line sampling steps were replaced with x^2 distribution, which sampled more steps closer to the point, where the depth difference has higher impact on the angle. Visibility estimate is calculated by projecting the normal vector to the min/max angle space, where analytic cosine integral is used to calculate properly weighted integral. GTAO also introduced two optimizations over HBAO. They used Z-buffer mip levels just like the most advanced point sample algorithms (last slide) and they traced each line segment to both directions at the same time. Negative direction simply requires negating the screen space sample offset, which is free to calculate. This way we reduce various line setup overhead costs to half.

Details

- Optimization: Trace “+” shape at once: (-y, x)
- Share samples with dFdxFine inside 2x2 quads → GLES driver issues 😞
- Micro-optimized 100% fp16 inner loop
- ALU based noise, staggered sampling (samplers slower on mobile)
- SSAO runs at ½ resolution
- Only 3 samples per line segment → 12 samples per cross
- 2 mip Z-buffer: Closest sample = mip 0, others = mip 1
- 4x4 spatial noise (angles/offsets) → 64 directions
- 4x4 bilateral box blur + bilateral upscale



Future

- Top down Z-buffer for large scale AO (sky visibility)
- Screen space / top down specular occlusion and approx GI

Bilateral upscale on off



Our ambient occlusion algorithm is based on GTA0. I wrote our algorithm from scratch since I wanted to get full knowledge in order to micro-optimize it.

The first optimization we did was tracing a full cross in the inner loop (4 directions) instead of 1 (HBAO) or 2 (GTA0). This further reduces the line setup costs. Calculating perpendicular line is free (-y, x).

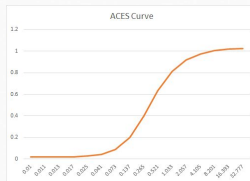
I implemented 2x2 quad neighborhood data sharing and it was great for reducing the noise. I implemented this using chained fine derivatives, which worked fine in Vulkan and Metal. However GLES3 (we still have WebGL2 backend for web) doesn't support fine derivatives. GLES has a hint for requesting fine derivatives, but ARM and Qualcomm drivers simply ignore this hint. As a result, we still haven't shipped this quality and performance optimization.

Our GTA0 is 100% fp16 code and we use ALU based noise instead of a texture lookup. This is because texture samplers are slower on mobile. SSAO is running at half resolution and we do only 3 samples per line segment. This is possible because each sample gives us an angle (instead of 1 bit result). In total we have 12 samples for the cross and we run one cross per pixel. 4x4 spatial noise for angles and offsets provides good enough 64 direction sample coverage. 4x4 bilateral box blur is used to blur the result. Since the 4x4 noise region is repeating, box blur at any location

contains all sampling directions.

Bilateral upscale is very important for quality when upscaling the SSAO. The images show edges looking half res without bilateral upscale. Our bilateral upscaler simply does one gather4 at the direction of the 2x2 lower resolution edge, resulting in always 4 different values. We pick pixel's own value if available and fallback to the closest one of the neighbors.

In the future we want to add a top-down Z-buffer based long range stable AO to approximate sky visibility better. We will also be experimenting using our GTA0 data to implement specular occlusion and some screen space GI approximation.



Bloom

- Combine bloom mips
- Physically based blend

Tonemap

- ACES and linear
- fp16 optimized

Color grading

- 16x16x16 LUT
- Brightness, contrast, saturation, tint, shadows, midtones...
- Presets: sin city, sepia...

Vignette

- Darken / brighten edges

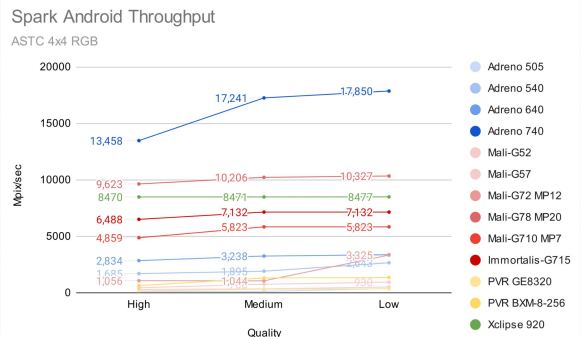
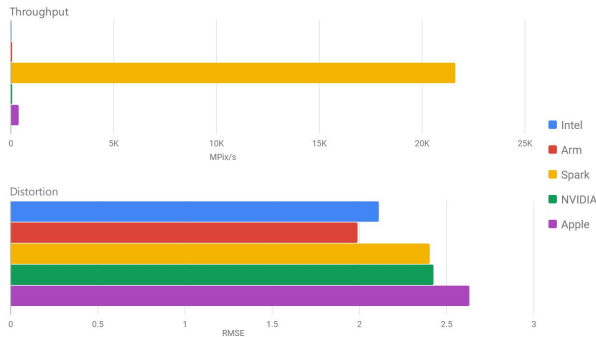
Post processing is a single pass design to avoid memory roundtrips. It's 100% fp16 code.

Bloom uses physically based blend (instead of additive to remain energy conserving). Tonemap is based on optimized ACES curve.

Color grading uses a 16x16x16 LUT. We generate the LUT on CPU side whenever post process sliders change. We support brightness, contrast, saturation, tint, shadows/midtones/highlight, etc, etc settings. And we also have presets for users such as Sepia and Sin City.

Vignette is added on top at the end of the shader.

Spark: Real-time GPGPU ASTC/BC7 Compressor



Compress 1000 ASTC textures per second on low end!

- 10MB game binary size limit
- **Storage:** JXL/AVIF at 2bpp (Vulkan AVIF HW decode?)
- **Runtime:** Spark GPGPU compressor → ASTC4x4 and BC7 (8bpp)
- **Future - Material composer:** Blend layers and decals at material load time
 - “Mini Substance Painter”, terrain heightmap, KTX2/Basis aren’t suitable for runtime composing

HypeHype is still lacking textures. I would have loved to talk about our texturing pipeline, but let’s talk about what we have now and some future plans for next fall.

10MB game binary size limit makes traditional textures very hard to implement. We want to compress the texture data as tightly as possible and use techniques to expand data further during game loading.

We co-funded development of a real-time ASTC4x4 / BC7 block compressor. It’s developed by Ignacio Castano and is called Spark (<https://ludicon.com/spark/>). This encoder has quality only slightly bit lower than the best offline compressors, but it is so fast that we can compress up to thousand 1024x1024 ASTC4x4 textures in a single second on low end phones.

This allows us to use state-of-the-art lossy compressors for our texture assets in our cloud server. We have been experimenting with 2bpp AVIF and JXL and the quality looks very good. AVIF can also be HW decoded on modern phones (Vulkan Video APIs). Decoded RGBA8 is sent to Spark for real-time GPU compression.

Why don’t we used existing super-compressed formats such as KTX2/Basis? We have plans to introduce a material composer in HypeHype. This is basically a “Mini Substance Painter”, and it allows our users to create their own materials on the phone. Materials are composed from multiple texture layers and decals, allowing us to bring lots of texture variety and hide tiling even with very small texture storage budget.

Visual comparison - Fall 2023 → Summer 2024



And this is how it looks like. On left side we have a screenshot from last fall. This is using the old rendering pipeline.

On the right side you see the new PBR pipeline with all the new features: PBR lighting (direct & indirect), GTA0, cascaded shadow mapping, LODs, instanced rendering, PBR bloom (mip chain), color grading and ACES tonemapping. We managed to implement all of this in 8 months.

The game shown here is a 2 year old UGC game. I didn't change the game content at all. It's 100% the same scene. I did however spend 5 minutes remixing the lighting environment of this game to look it better, and improved the draw distance as our new technology has performance to run much larger scenes.

References



- **[Walter]** Bruce Walter et al. 2007. Microfacet Models for Refraction through Rough Surfaces. Proceedings of the Eurographics Symposium on Rendering.
- **[Heitz]** Eric Heitz et al. 2016. Multiple-Scattering Microfacet BSDFs with the Smith Model. ACM SIGGRAPH 2016.
- **[Schlick]** Christophe Schlick. 1994. An Inexpensive BRDF Model for Physically-Based Rendering. Computer Graphics Forum, 13 (3), 233–246.
- **[Filament]** Physically Based Rendering in Filament (<https://google.github.io/filament/Filament.html>)
- **[Praun]** Emil Praun, Hugues Hoppe. 2003. Spherical Parametrization and Remeshing. ACM transactions on graphics
- **[Mitting]** Martin Mitting. Finding next gen: Cryengine 2. In ACM SIG- GRAPH 2007 Courses, 2007.
- **[Bavoil]** Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In ACM SIGGRAPH 2008 Talks, 2008.
- **[Jimenez]** Jiménez, Jorge, et al. "Practical real-time strategies for accurate indirect occlusion." SIGGRAPH 2016 Courses: Physically Based Shading in Theory and Practice (2016).