What is Filament?

https://github.com/google/filament

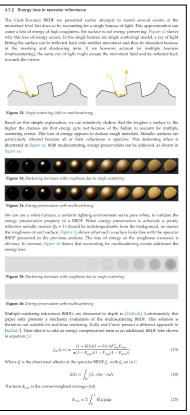
Filament is a physically based real-time renderer for Android, iOS, macOS, Linux and Windows
It supports OpenGL ES 3.0, OpenGL 4.1, Vulkan, and Metal
It is JUST a rendering engine, not a game engine
Used in AR applications at Google for instance

Design goals

Embeddable
aarch64 first
Small binary size (aarch64 is 480 KiB compressed)
High rendering quality: advanced material models (anisotropy, clear coat, cloth…) and effects (refraction)
Stay physical as much as possible: camera, light units, etc.

Physically based rendering in Filament

https://google.github.io/filament/Filament.html



You may know Filament because of its documentation on PBR
It's a long document explaining all the theory, math and choices behind our PBR implementation

---

Optimizing for tilers

There are TWO MAJOR optimizations you MUST implement before you do ANYTHING else in your engine
The first one of them is handling tiling GPUs properly

---

resolve/unresolve

Bandwidth is at a premium on mobile, it's improving at a glacial pace
And to help with bandwidth management on tilers you must RESOLVE and UNRESOLVE tiles properly
A resolve — or store — is writing the GPU tile to memory
An unresolve — or load — is reading a GPU file from memory
Explicit on Metal and Vulkan, and GLES drivers try to do this using heuristics
For instance if you do a full-surface

glClear, the driver will likely skip the unresolve

---

glInvalidateFramebuffer()

before draw calls → prevent unresolves

after draw calls → prevent resolves

Resolves/unresolves can be controlled explicitly in GLES using glinvalidateFrameBuffer()
It is however tricky to do manually, you must know whether you need the content to be resolved/ unresolved, which becomes complicated when you have many features in your rendering pipeline that can be enabled/disabled
We let our FRAMEGRAPH do this for us

Framegraph

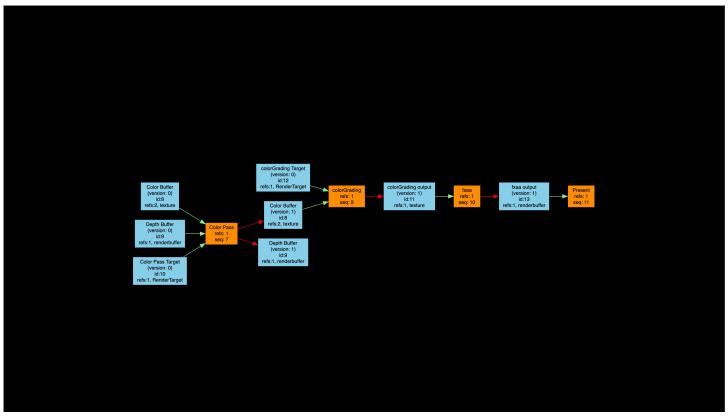
knows all buffer dependencies

manages buffers lifetime

computes discard flags

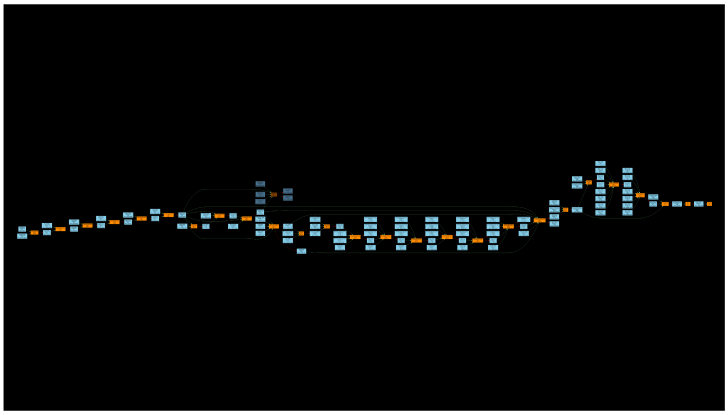
The discard flags are the flags passed to glInvalidateFramebuffer()
They are computed per buffer, and per pass:

- Buffer is never read after current pass: discarded at the end of the current pass
- Buffer never written to before the current pass: discarded at the beginning of the current pass
- Buffer cleared by the pass: discarded, no matter what happened before

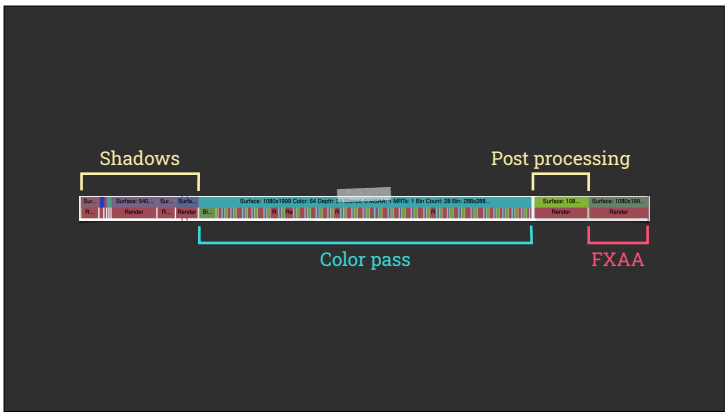


Here is a frame graph for a color pass + a post process pass (color grading/tone mapping/dithering) + a pass of FXAA
It's fairly simple and managing resolves/unresolves by hand here is easy

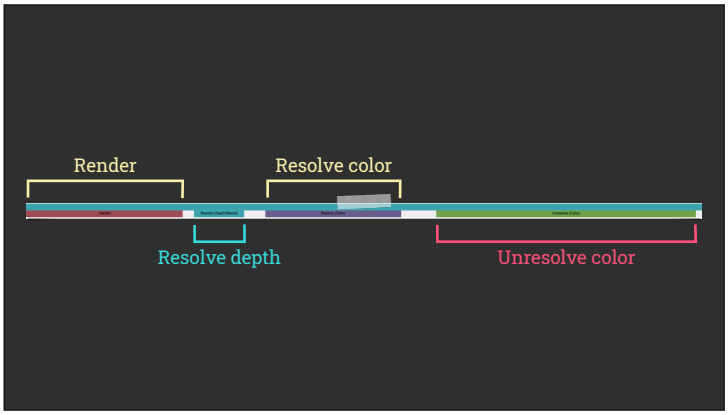
---



And now, here is a frame graph with more passes:
directional shadows, spot shadows, screen-space shadows, screen-space refraction, SSAO, bloom, color grading, FXAA and MSAA.
This gets even more complicated when we enable depth-of-field
Code that resolves/unresolves buffers quickly becomes unmaintainable when you have so many passes and dependencies between the buffer. The problem is made worse by the fact that everything can be changed dynamically at runtime.

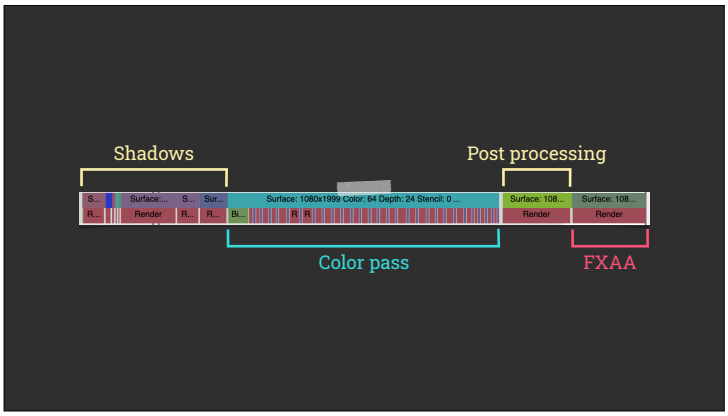


But let's take a look at a concrete example.
This is the anatomy of a frame on a Pixel 4 where we don't both with glInvalidateFramebuffer().
First the shadow passes. Then the color pass: RGBA16F, MSAA, in 2000x1080 pixels. We're going to zoom in on that pass in a minute, but look at the pattern of colored blocks. Each repetition is one of the GPU tiles. Then, we have post processing. And finally, FXAA.

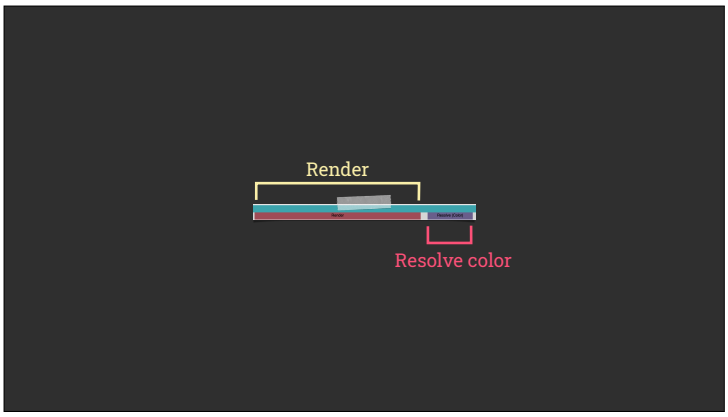


Let's zoom in one of those tiles for the color pass. First, we render.
Then, the depth buffer is resolved. That's unfortunate since after the color pass we never need the depth!
The, the color buffer is resolved. That makes sense, we need is for post-processing/output
Finally, the color buffer is unresolved for *the next tile*. That's silly because we always render the entire screen and we don't need the previous content

---



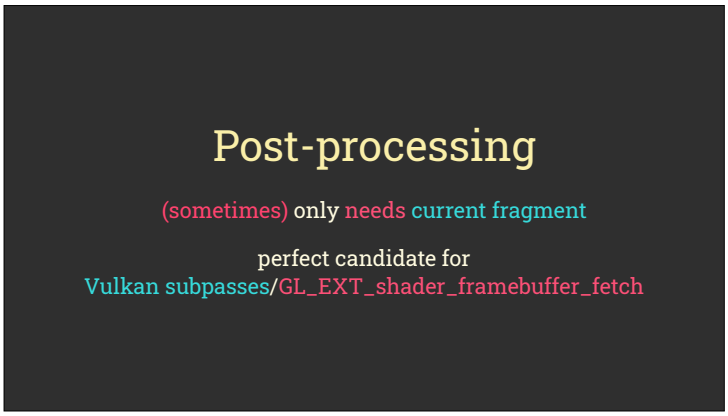
Now here is the same frame with tile management using glInvalidateFramebuffer()
We still have the same render pass but you may be able to notice a difference about the color pass
It's much smaller relative to the other pass. Let's zoom on that…



And indeed it is much simpler! We render then resolve the color buffer.
We don't do any extra work!
On a Pixel 4, at 1080p all those extra resolves and unresolves can add up to 1ms per frame just in the color pass

---

There is one more thing we can do. Recall that our color buffer is RGBA16F, which means each resolve writes 64 bits per fragment. We need this data for the color grading/tone mapping/dithering pass. Or… do we?



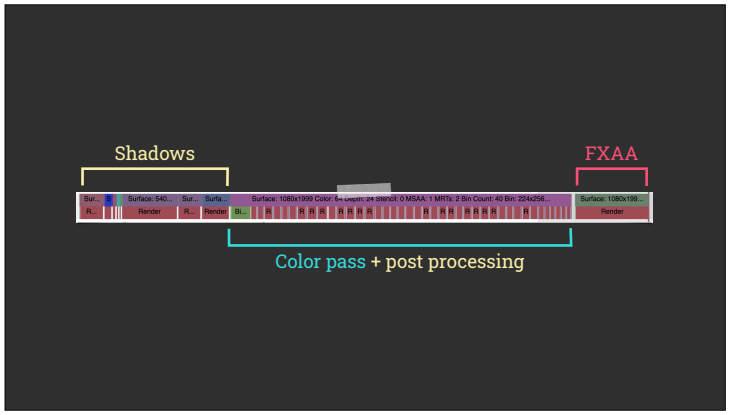
When effects like bloom, screen-space refraction, etc. aren't needed, we go directly from the color pass to the post-process pass. And that pass only needs the *current* fragment
This makes it a perfect candidate for Vulkan subpass, or in the GLES world, the frame buffer fetch extension

Recipe

1. Use 2 MRTs: 1 HDR (RGB16F), 1 LDR (RGB8)
2. Render color pass into HDR target
3. Render full-screen triangle into LDR target
   3.1. Shader fetches from HDR target
   3.2. Apply color grading, tone mapping, dithering, etc.
   3.3. Outputs to LDR target

Here's how this solution works, it's really simple
We start with 2 render targets, both bound at the same time (it's key to staying in the tile)
Note that the color pass never writes to or reads from the LDR target

---



Using frame buffer fetch, the post processing pass is completely gone. It is now merged with the color pass. There are still resolves happening, but we resolve to 8 bit (RGB8) instead of RGBA16F. There are more tiles than before (40 vs 28) because of the increase memory requirement per tile.

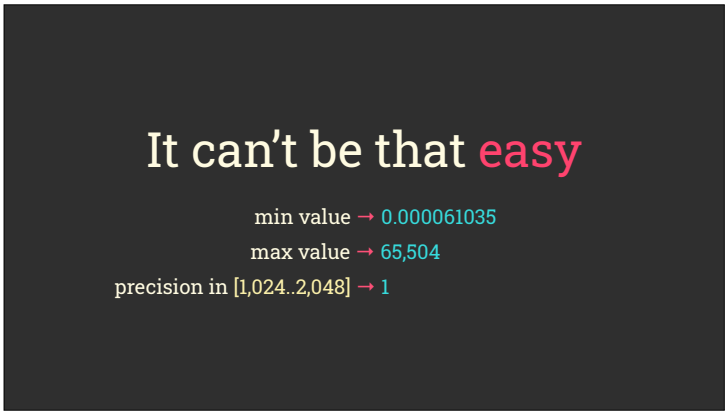
---

fp16

The second major optimization you can implement is to use half floats everywhere in your fragment shaders
You can run 2x as many instructions per cycle
Which means either better performance, or using less power/ generating less heat

```
precision mediump float;

out vec4 fragColor;

void main() {
    // Do work
}
```

With GLSL it's trivial, just set your default precision to mediump and you're done!



Well… almost
We must deal with 3 problems: first the min value is not that small. It's quite easy to reach when you start squaring small values in BRDFs. Note that we assume that GPUs don't support denormals, or at least that we don't want denormals. The second issue is that the large value is only 65k or so. We'll see why that's an issue very soon
Finally we lose all fractional precision above 1024. This makes half float unsuited to store texture sizes or any kind of UVs, especially in screen space since on mobile resolutions are much higher. Obvious this is also an issue for vertex positions. For instance this would mean a resolution of 1 meter 1km away from the origin. Not good.

```
// we use x / (roughness^4) in our code
// min value in fp16 → 2^-14 = 0.000061035
// we want roughness^4 >= 2^-14
// so min roughness = 2^(-14/4) ~= 0.089
roughness = clamp(roughness, 0.089, 1.0);
```

So let's take a look at how to use half precision floats in your shaders successfully
First make sure to clamp values so that you don't end up with divisions by 0
For example here is how we clamp roughness to a minimum value.
This number 0.089 comes from understanding the math we execute in the shaders…

```
float D_GGX_Anisotropic(float at, float ab, float ToH, float BoH, float NoH) {
    // at & ab → roughness^2, a2 → roughness^4
    // The dot product below computes roughness^8
    // → won't fit in fp16 without clamping the roughness to 0.298
    // → perform the dot product and the division in fp32
    float a2 = at * ab;

    highp vec3 d = vec3(ab * ToH, at * BoH, a2 * NoH);
    highp float d2 = dot(d, d);

    float b2 = a2 / d2;
    return a2 * b2 * b2 * (1.0 / PI);
}
```

Sometimes however you don't have a choice
Here is for instance the anisotropic NDF we use in Filament
Notice how we compute "d" and "d2" at high precision, with single precision floats. Why is that?

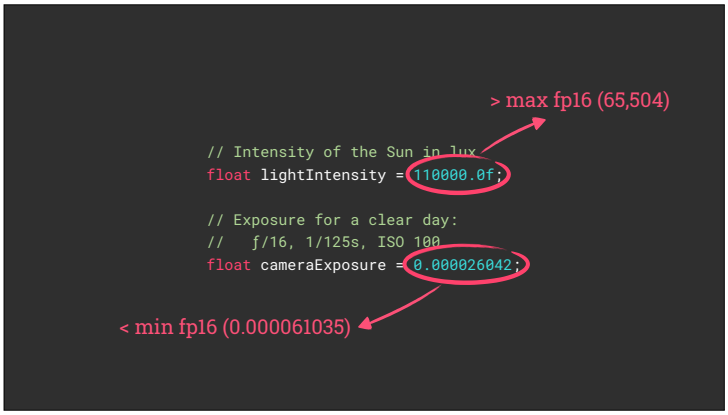
```
#define MEDIUMP_FLT_MAX      65504.0
#define saturateMediump(x)   min(x, MEDIUMP_FLT_MAX)

float V_SmithGGXCorrelated(float roughness, float NoV, float NoL) {
    float v = // ...
    return saturateMediump(v);
}
```

Here is another interesting example. Sometimes computations can tend to infinity, as is the case with this visibility term.
Unfortunately on some mobile GPUs infinity leads to a black pixel.
To avoid this we clamp the result to 65,504

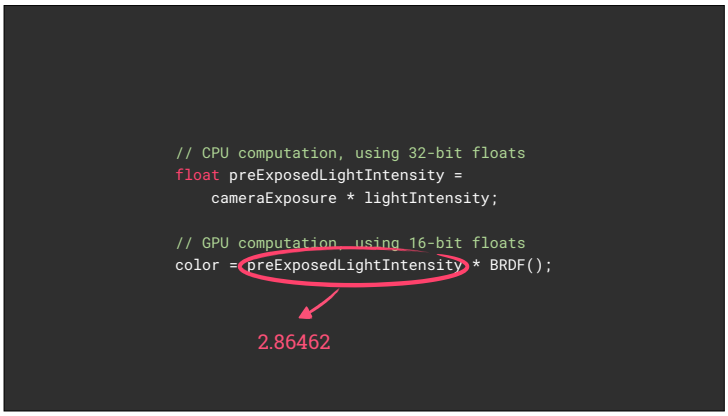
```
// GPU computation
color = lightIntensity * cameraExposure * BRDF();
```

Here is one last problem. This is a simplified implementation of our lighting. Nothing crazy here
Unfortunately things are a little more complicated when we look at the possible values at play

---



This code represents a typical setup with our physical light units
And there are two issues with those values
- the sun intensity is too large to fit in fp16
- the exposure is too small to fit in fp16



The solution is quite simple, we simply pre-multiply light intensities on the CPU
In our example this yields a value that fits neatly in fp16 and our computations work just fine
In practice we store our lights pre-exposed

Be careful

Using these techniques we've been able to write our shaders in fp16. It can be tricky to chase artifacts created by a computation in fp16 but it's worth the efforts. Falling back to fp32 is sometimes unavoidable, mostly when you deal with UVs/texture sizes and vertex positions, etc.

Faster* BRDFs

* Faster but more "wrong" :)

$$V(v,l,\alpha) = \frac{0.5}{n \cdot v\sqrt{(n \cdot v)^2(1-\alpha^2)+\alpha^2} + n \cdot v\sqrt{(n \cdot l)^2(1-\alpha^2)+\alpha^2}}$$

$$\sqrt{a^2b^2+c^2} \approx ab + c$$

Here is the height-correlated Smith NDF described by Eric Heitz in "Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs". It's the term we use in our BRDF… at least on desktop. The two square roots are not ideal on mobile.
Let's focus on one of those square roots
It's the square root of a mad of squares, and all the terms are in the [0..1] range, so if we squint hard enough… we can remove the square roots :)

$$V(v,l,\alpha) = \frac{0.5}{n \cdot l(n \cdot v(1-\alpha)+\alpha) + n \cdot v(n \cdot l(1-\alpha)+\alpha)}$$

$$V(v,l,\alpha) = \frac{0.5}{2(n \cdot l)(n \cdot v)(1-\alpha) + (n \cdot v + n \cdot l)\alpha}$$

And look, no more square roots, no more squares!

```
float V_SmithGGXCorrelated_Fast(float roughness, float NoV, float NoL) {
    // Hammon 2017, "PBR Diffuse Lighting for GGX+Smith Microsurfaces"
    return 0.5 / mix(2.0 * NoL * NoV, NoL + NoV, roughness);
}
```

After simplification we end up with a simple lerp
This horror was described independently by Earl Hammon at GDC 2017
Note that we only use this approximation on mobile

Let’s take a quick look at clear coat
Clear coat is implemented as a second specular lobe on top of the base layer
It’s great to crate beautiful materials like this lacquered wood or coated carbon fiber, and it’s easy to implement too
But it’s a second BRDF to compute per light…

$$V_{clearCoat}(l,h) = \frac{1}{4(l \cdot h)^2}$$

For clear coat we replace the visibility term by another one, described by Kelemen
in the paper “A Microfacet Based Coupled Specular-Matte BRDF”
Eric Heitz has shown in his own paper, “Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs”, that this term is not physically based
But it’s *cheap* and it still looks great, so we adopted it
We may revisit this decision since the simplified Visibility term shown previously isn’t that much more expensive

---

Tone mapping

Krzysztof Narkowicz has a popular fit of ACES

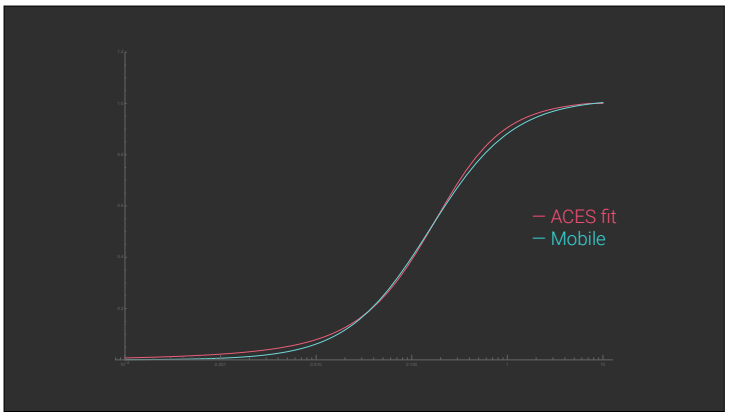
A few years ago Krzysztof Narkowicz popularized a fit of the ACES RRT+ODT for sRGB output

```
vec3 Tonemap_ACES(const vec3 x) {
    // Narkowicz 2015, "ACES Filmic Tone Mapping Curve"
    const float a = 2.51;
    const float b = 0.03;
    const float c = 2.43;
    const float d = 0.59;
    const float e = 0.14;
    return (x * (a * x + b)) / (x * (c * x + d) + e);
}
```

Here is the implementation. It's simple and straightforward, which probably explains its popularity
Especially compared to how complicated the ACES RRT+ODT are…

```
vec3 Tonemap_Mobile(const vec3 x) {
    // Transfer function baked in,
    // don't use with sRGB OETF!
    return x / (x + 0.155) * 1.019;
}
```

We propose a simple, extremely fast tone map operator that also approximates the sRGB transfer function.
It's all kinds of "wrong" but it's good enough for mobile
And it's cheap



Here is a comparison of the tone mappers with gamma 2.2

And a quick comparison of the ACES fit…

And the approximation for mobile
NOTE: As of Filament 1.8 we don’t use this anymore. We use a LUT for both tone mapping and color grading, with different tone mappers to choose from

Image-based lighting

Spherical harmonics

SH 2 or 3 bands for diffuse lighting

easy to compute offline & cheap to apply

looks great for low/medium dynamic range environments

We started, not surprisingly, with spherical harmonics.
We use either 2 or 3 bands for the diffuse component of indirect lighting
These SH are easy to compute and are dirt cheap to apply, especially with 2 bands
And importantly they yield great looking results for low to medium dynamic range

Here is an example of a scene with a rough, dielectric gray sphere lit by 2 bands SH
The environment is medium dynamic range, ~9 EVs
It looks great, it's fast, we love it

Unfortunately spherical harmonics don't fare so well in high dynamic range environments, here 21 EVs (the center of the sun has a value > 300,000). This is not something you'd see in a game, or at least you'd want to remove the sun and use an analytical light instead (for shadows, etc.)
We can see 2 issues with 2 bands SH here:

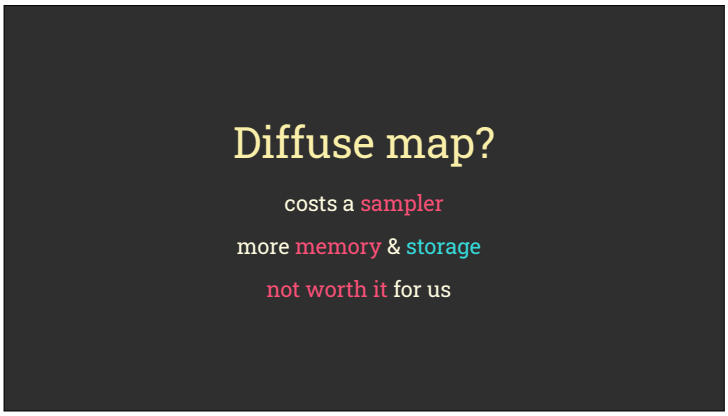
- ringing
- and the back side of the sphere is completely wrong. It's too dark but the light also wraps around the sphere

---

There are ways to improve the results with SH. First we can use more bands, and we can apply a low-pass filter to the spherical harmonics ("windowing"). Our implementation follows "Deringing Spherical Harmonics" by Peter-Pike Sloan, and you can see the result above. The ringing is gone but the back side of the sphere is still wrong; it looks like we use a wrap diffuse term.
This is is not good enough for us because some of our clients want to use such high dynamic range

environments.

So Filament gives you a choice: you can use SH, or you can use a diffuse map. Here is the result of the same scene with diffuse lighting coming from a texture. The result is much better and much closer to that of an analytical directional light.



So clearly a diffuse map gives better results, but is it what we want?
First, an extra texture means another sampler. We have a limit of 16 samplers on mobile and because we use a forward renderer any sampler we use cannot be used by the user's materials.
Of course, this also means more memory and storage for the app, although those are less of an issue as the diffuse map can be small (128x128 for instance). It's still an extra texture to carry around

Ultimately we decided it wasn't worth it

Specular map!

diffuse lighting ~ specular lighting at roughness=1

$$\sum L(l) \approx \sum \langle n \cdot l \rangle L(h)$$

roughness 1.0 → cannot be 1x1 LOD

looks good enough

So instead we use the specular map, or the preconvolved environment map
If you squint hard enough at the equation, the integration of the specular BRDF at roughness=1 is almost diffuse lighting. Besides our IBL already relies on the split-sum approximation, we can do funky stuff with the math :)
Of course this means we cannot map roughness=1 to the 1x1 LOD, otherwise we'd lose all details. In practice we use the 16x16 LOD

In practice, it looks like this. Now if you look at the terminator you'll notice aliasing that appears in high dynamic range environments.

On desktop we use a tent filter when sampling the “diffuse” map, to smooth the aliasing artifacts. It works well but it requires multiple samples so we don’t enable this feature on mobile. At high pixel densities and with textured objects, this is not an issue in practice. You can also use Spherical Harmonics instead if you want (but change your env map!)

Roughness mapping

256x256 to 16x16 → 5 mip levels

$log_2(roughness) + roughnessOneLOD$

$roughnessOneLOD * roughness * (2 - roughness)$

I mentioned we want to use the last specular level, or 16x16 px, as our diffuse map. Since our first mip is 256x256, we need to find a good mapping for the roughness that doesn’t waste high resolution mips.
Starting at 256x256 all the way to 16x16, we have 5 mip levels
So we looked for a mapping that would give glossy surfaces the lowest mips
This is our empirical solution.
But in practice we use a quadratic fit instead to avoid a log2
This mapping works well for a good range of mip levels, up to 9

Roughness mapping

0 → 0.000

1 → 0.018

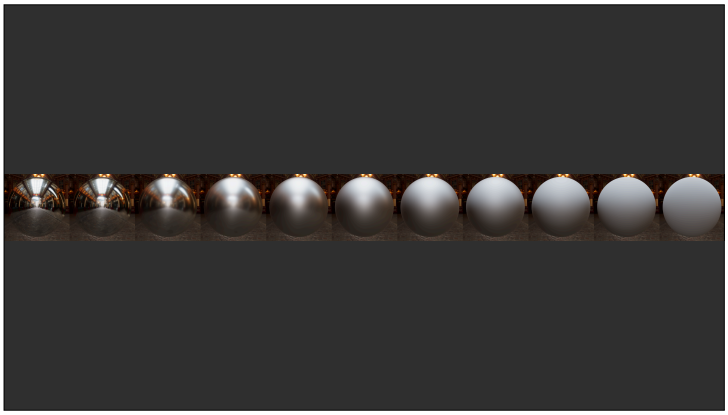
2 → 0.086

3 → 0.250

4 → 1.000

So here is our mip map to roughness in practice
The first 4 levels are used for roughness < 0.25
It works really well in practice since above roughness 0.25 lighting becomes very blurry

---



Here is a metallic ball with roughness varying from 0 to 1 showcasing this remapping

---

Storing IBLs

RGBM → quality issues

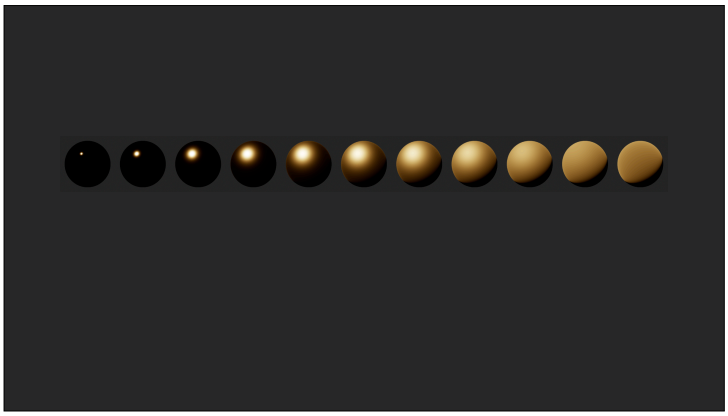
R11G11B10F → swizzled as RGBA8

Filament doesn't impose the format of the IBL. You can use RGB16F if you wish, or compressed texture formats. Unfortunately when targeting OpenGL ES 3.0 devices there's no easy way to compress HDR textures.
Originally we were using RGBM, but it proved difficult to pick a shared multiplier that would work in all environments, and making it customizable would just create confusion in the asset pipelines
So instead we use R11G11B10F, and we simply swizzle the data as

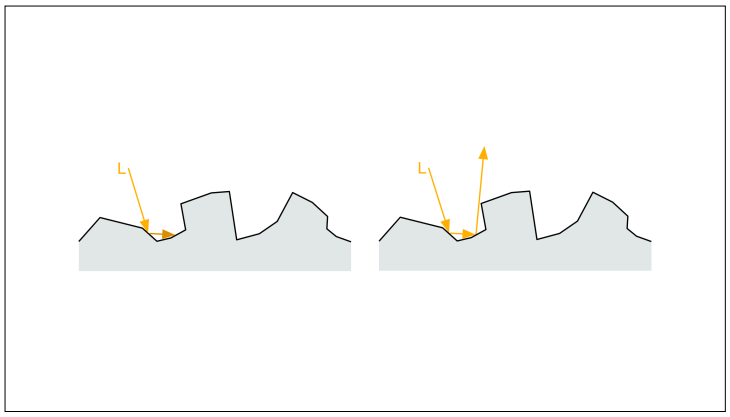
RGBA8 for storage, typically PNG files. You should of course not run any compression on that swizzled data, all the channels are correlated

---

Fixing metals

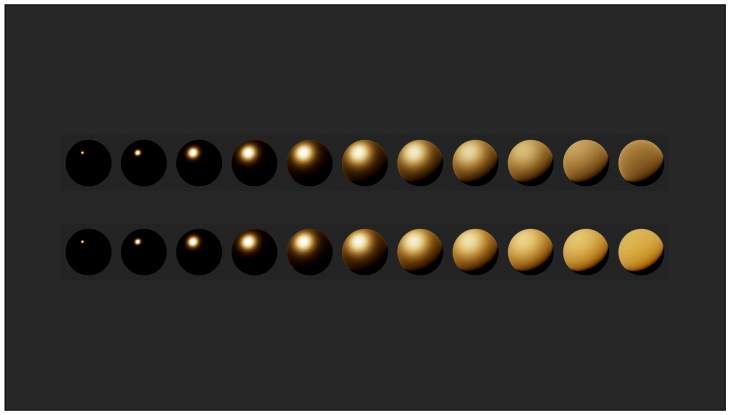


The Cook-Torrance BRDF we (and pretty much everybody) use models only a single bounce of light in the micro facets field.
This lead to a loss of energy particularly visible at high roughness. This cause metals in particular to get darker.



It makes sense intuitively. With a single bounce (left), a light ray is unlikely to espace the micro facets field.
If instead we model multiple scattering events (right), the light ray might be reflected toward the viewer

---



If we properly account for multiple scattering events, we can recover the lost energy and get metallic surfaces that behave a lot better
See the second row of spheres

The difference is even more impressive in a white furnace
There *are* two rows of spheres.
The top one, with single scattering, shows how much loss we suffer
The second row reflects all incoming energy, as expected

We use a solution from
Lagarde & Golubev

Solution presented at SIGGRAPH 2018

$$f_r(l,v) = f_{ss}(l,v) + f_{ms}(l,v)$$

$$f_r(l,v) = f_{ss}(l,v) + f_0\left(\frac{1}{r} - 1\right) f_{ss}(l,v)$$

$$r = \int_\Omega D(l,v)V(l,v)\langle n \cdot l \rangle \, dl$$

The idea is very simple, the specular lobe is the single scattering lobe (from your BRDF) + a multi scattering lobe
And the multi scatter lobe is itself a scaled version of the single scattering lobe
The term “r” happens to be what we compute in the DFG LUT for image-based lighting!

```
const float V = Visibility(…) * NoL * (VoH / NoH);
const float F = pow5(1.0f - VoH);
r.x += V * (1.0f - F);
r.y += V * F;
```

When you compute your DFG LUT, your code will look similar to this
We store the importance sampled visibility term multiplied by Fresnel

```
const float V = Visibility(…) * NoL * (VoH / NoH);
const float F = pow5(1.0f - VoH);
r.x += V * F;
r.y += V;
```

The change is very simple, we store the visibility term directly

---

```
vec2 dfg = textureLod(dfgLut, vec2(NoV, roughness), 0.0).xy;
// For image-based lighting
vec3 iblSpecularColor = mix(dfg.xxx, dfg.yyy, f0);

// For other lights
vec3 energyCompensation = 1.0 + f0 * (1.0 / dfg.y - 1.0);
Fr *= pixel.energyCompensation;
```

Then we can simply apply the formula from earlier and scale our specular lobe
It's cheap, easy to implement and produce beautiful results
There are other solutions that treat dielectrics and conductors separately but they are more expensive
We like this one for its simplicity and low cost

Rendering cloth

Filament offers a material model to create cloth & fabric, including satin, velvet, etc.
The cloth models lets you control the sheen color directly and the roughness works very differently than it does for regular materials

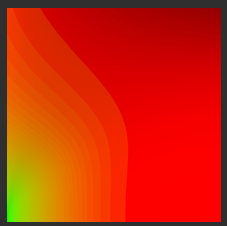
---

$$f_t(v,h,\alpha) = \frac{D_{velvet}(v,h,\alpha)}{4(n \cdot l + n \cdot v - (n \cdot v)(n \cdot l))}$$

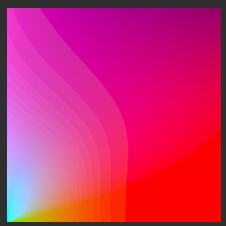
$$D_{velvet}(m) = \frac{(2 + \frac{1}{\alpha})sin(\theta)^{\frac{1}{\alpha}}}{2\pi}$$

For cloth we use the Ashikhmin BRDF modified by Neubelt & Pettineo for The Order 1886
However, our NDF is neither Ashikhmin's nor the one from The Order (normalized Ashikhmin)
Instead we use the "Charlie" sheen BRDF from Estevez and Kulla presented at SIGGRAPH 2017
It is softer, cheaper and provides a more natural control over the roughness

---



DFG LUT



DFG LUT with cloth BRDF

However, because we use a different BRDF, we cannot use the DFG LUT as it
That LUT is computed for the regular Cook-Torrance BRDF with GGX NDF and Fresnel term, which looks completely different
—> !!NOTE: The DFG LUT shown above is the single-scattering LUT because you may be more familiar with it
If you recall the previous slide, the cloth BRDF does not have a Fresnel term, so we only need 1 channel to store the cloth BRDF in the LUT

We use the green channel for that purpose
We found, just like Estevez & Kulla, that uniform sampling works well to generate that LUT

```
static float DFV_Charlie_Uniform(float NoV, float roughness, size_t numSamples) {
    float r = 0.0f;
    const float3 V(std::sqrt(1.0f - NoV * NoV), 0.0f, NoV);

    for (size_t i = 0; i < numSamples; i++) {
        const float2 u = hammersley(uint32_t(i), 1.0f / numSamples);
        const float3 H = hemisphereUniformSample(u);
        const float3 L = 2 * dot(V, H) * H - V;
        const float VoH = saturate(dot(V, H));
        const float NoL = saturate(L.z);
        const float NoH = saturate(H.z);

        if (NoL > 0.0f) {
            const float V = VisibilityAshikhmin(NoV, NoL, roughness);
            const float D = DistributionCharlie(NoH, roughness);
            r += V * D * NoL * VoH;
        }
    }

    return r * (4.0f * 2.0f * (float) F_PI / numSamples);
}
```

Just for reference, here is the code to generate the 3rd channel in the LUT

One thing left is the convolution of the environment maps…
Technically we should have a separate convolution of each IBL because the BRDF is different
This is however not practical in terms of storage, memory and performance on mobile so we reuse the same convolution
It is wrong but good enough

Multi-bounce AO

The way we render ambient occlusion suffers from a similar issue we saw earlier for metals that darken at high roughness: we only take direct visibility into account. Thankfully Jimenez offers a great solution in Practical Real-Time Strategies for Accurate Indirect Occlusion, from SIGGRAPH 2016 (GTAO)

First, let's see what it does
Here is a scene with regular ambient occlusion (note we are not using skin shading in this example)

And here is the same scene with multiple light bounces modeled in the ambient occlusion term. Note the area around the eyes or the ears. This solution offers both a more accurate AO term but is also introduces colored indirect illumination which provides a more natural and less dull appearance

```
vec3 gtaoMultiBounce(float visibility, const vec3 albedo) {
    // Jimenez et al. 2016,
    // "Practical Realtime Strategies for Accurate Indirect Occlusion"
    vec3 a =  2.0404 * albedo - 0.3324;
    vec3 b = -4.7951 * albedo + 0.6417;
    vec3 c =  2.7552 * albedo + 0.6903;

    return max(
        vec3(visibility), ((visibility * a + b) * visibility + c) * visibility
    );
}

diffuseLobe *= gtaoMultiBounce(ao, diffuseColor);
```

The best part is that the implementation is cheap and simple. It's a polynomial fit found by the Activision team and it's trivial to integrate in an engine. This feature is optional per material in Filament and is well worth turning on for some assets

---

Specular anti-aliasing

Our main AA techniques are MSAA and FXAA. It is however sometimes not enough, and for those situations we offer an optional material feature to enable specular anti-aliasing

---

Here is an example of a glossy dielectric material
I used a low albedo to make the specular highlights a little easier to see

If we zoom out, the highlights get lost due to aliasing
If we go back and forth… we can that we lose the same of some highlights, like the one on the chest

---

Now here is the same model with specular antialiasing turned on

---

And if we go back and forth…
<FLIP> we properly retain all the highlights

```
float normalFiltering(float perceptualRoughness, const vec3 worldNormal) {
    // Kaplanyan 2016, "Stable specular highlights"
    // Tokuyoshi 2017, "Error Reduction and Simplification for Shading Anti-Aliasing"
    // Tokuyoshi and Kaplanyan 2019, "Improved Geometric Specular Antialiasing"
    vec3 du = dFdx(worldNormal);
    vec3 dv = dFdy(worldNormal);

    float variance = specularAntiAliasingVariance * (dot(du, du) + dot(dv, dv));

    float roughness = perceptualRoughnessToRoughness(perceptualRoughness);
    float kernelRoughness = min(2.0 * variance, specularAntiAliasingThreshold);
    float squareRoughness = saturate(roughness * roughness + kernelRoughness);

    return roughnessToPerceptualRoughness(sqrt(squareRoughness));
}

materialRoughness = normalFiltering(
        materialRoughness, getWorldGeometricNormalVector());
```

To implement this we use a solution originally devised by Anton Kaplanyan in 2016, later improved by Yusuke Toyushi in 2017, and improved again by both of them in 2019

The main idea is to vary the roughness based on local curvature, or variance of the geometric normal, to filter the NDF. The papers cited above present two techniques: one for DEFERRED, and one for FORWARD. The forward technique is expensive, with a multiplication by a tangent frame for every light. Instead we use the deferred variant in forward, modifying the material roughness once. It works really well and the cost is sometimes worth it.

// specularAntiAliasingVariance and specularAntiAliasingThreshold are two uniforms, our defaults are respectively 0.15 and 0.2

Ambient occlusion

Scalable Ambient Obscurance

excellent quality → only 7 samples

great performance → 2.3ms @ 1080p on Pixel 4

We use an algorithm called Scalable Ambient Obscurance by Morgan McGuire
We chose it because it delivers great quality even with a limited number of samples
But it also offers great performance on mobile
Given how much it can improve the visuals, this effect is well worth the cost
NOTE: during this pass we create a mip chain of depth buffers, which can be reused for other effects like depth of field

---

Here is an example of a scene without SSAO…

And with SSAO
This is our lowest quality — 7 samples — and it gives very good results

Our changes

interleaved gradient noise → cheaper

constant spiral angle → avoid sin/cos

face normals from depth → without derivatives

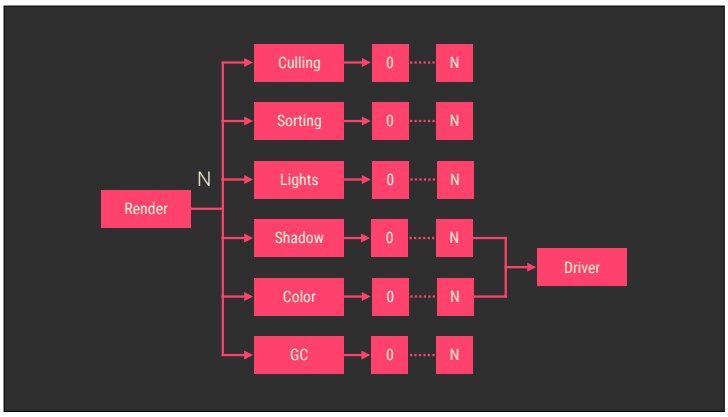
fp16 friendly → cheaper

We change the noise pattern for a simpler, cheaper one
the sampling pattern is a spiral, we made the angle constant to avoid sin/cos. These are EFU instructions on Qualcomm GPUs which have higher latency and/or cost more power
We don't reconstruct normals using derivatives. We can only count on partial derivatives on mobile and this helps us compute normals at full resolution
And finally our code tries to do everything in fp16 of course

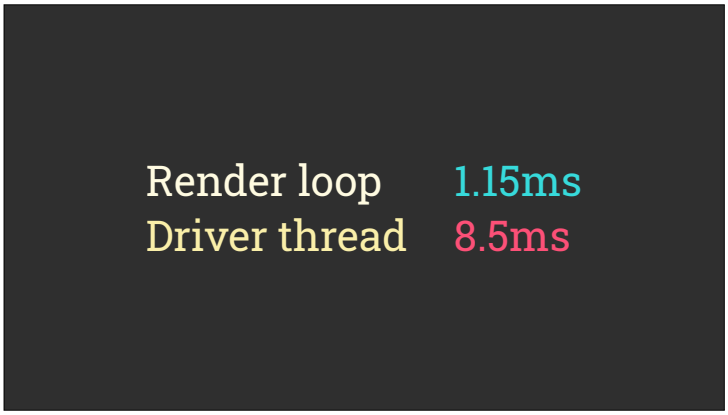
---

Job system

We're going to use this scene as an example, on a Pixel 4 (screenshot from desktop viewer)
It's the Amazon Lumberyard Bistro scene (https://developer.nvidia.com/orca/amazon-lumberyard-bistro)
Multiple lights, some cast shadows, some don't



Here is a simplified version of the job system used in Filament
Every block represents a new parented Job, which itself can spawn N jobs
Effectively every loop in this system is multi-threaded and jobified
It scales naturally to the number of cores in the machine
We use it for EVERYTHING
Our implementation is HEAVILY influenced by Stingray's



The render loop includes: culling, sorting per pass (shadow/color)
It also includes froxelization for the lights
In this scene we end up with 200 visible objects (200 draw calls) and multiple render passes (shadow maps, color pass, post process)

Here you can see the job system in action when we load the scene
It maximizes CPU occupancy (8 cores in this case, 4 big and 4 little)
Note it pins the driver thread (blue blocks at the bottom) to a big core

```
do {
    if (!execute(*state)) {
        std::unique_lock<Mutex> lock(mWaiterLock);
        while (!exitRequested() && !hasActiveJobs()) {
            wait(lock);
            setThreadAffinityById(state->id);
        }
    }
} while (!exitRequested());
```

Here is code at the core of our JobSystem, the loop run by every thread

---

Every time a thread wakes up we reset its affinity
We do this because CPUs can be unplugged on Android which resets the affinity
By setting the affinity we can pin a thread to a given core, thus avoiding hops between cores which are costly

---

```
// Highest affinity bit, assuming this is a big core
// This core is not used by the JobSystem
uint32_t id = std::thread::hardware_concurrency() - 1;

while (true) {
    JobSystem::setThreadAffinityById(id);
    if (!execute()) {
        break;
    }
}
```

The JobSystem always leaves a core available, and we guess it's a big core using the higher affinity bit
Again we do this every time we wake up the driver thread to reset the affinity if we lost it

```
void JobSystem::setThreadAffinityById(size_t id) noexcept {
#if defined(__linux__)
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(id, &set);
    sched_setaffinity(gettid(), sizeof(set), &set);
#endif
}
```

For reference, here is the implementation of setThreadAffinity()

Syntax highlighter (https://romannurik.github.io/SlidesCodeHighlighter/): {“bgColor":"#2e2e2e","textColor":"#f9f9f9","punctuationColor":"#d7b67c","stringAndValueColor":"#da7f5e","keywordTagColor":"#ff436e","commentColor":"#afd695","typeColor":"#faf0a8","numberColor":"#3adbda","declarationColor":"#f9f9f9"}

Physically based rendering in Filament, Guy & Agopian
https://google.github.io/filament/Filament.html

Scalable Ambient Obscurance, McGuire, 2012
https://casual-effects.com/research/McGuire2012SAO/index.html

PBR Diffuse Lighting for GGX, Hammon, 2017
https://www.gdcvault.com/play/1024478/PBR-Diffuse-Lighting-for-GGX

The road toward unified rendering with Unity’s high definition rendering pipeline, Lagarde & Golubev, 2018
http://advances.realtimerendering.com/s2018/index.htm

Crafting a Next-Gen Material Pipeline for The Order: 1886, Neubelt & Pettineo, 2014
https://www.gdcvault.com/play/1020162/Crafting-a-Next-Gen-Material

Production Friendly Microfacet Sheen BRDF, Estevez & Kulla, 2017
http://www.aconty.com/pdf/s2017_pbs_imageworks_sheen.pdf

Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs, Heitz, 2014
http://jcgt.org/published/0003/02/03/paper.pdf

Practical Real-Time Strategies for Accurate Indirect Occlusion, Jimenez, 2016
https://blog.selfshadow.com/publications/s2016-shading-course/

Stable Specular Highlights, Kaplanyan, 2016
http://developer.download.nvidia.com/gameworks/events/GDC2016/akaplanyan_specular_aa.pdf

Error Reduction and Simplification for Shading Anti-Aliasing, Tokuyoshi, 2017
http://www.jp.square-enix.com/tech/library/pdf/Error%20Reduction%20and%20Simplification%20for%20Shading%20Anti-Aliasing.pdf

Improved Geometric Specular Antialiasing, Kaplanyan & Tokuyoshi, 2019
http://www.jp.square-enix.com/tech/library/pdf/ImprovedGeometricSpecularAA.pdf

---

HDRIs/environments
https://hdrihaven.com/

Textures
https://texturehaven.com/ & https://www.cgbookcase.com/

glTF Samples
https://github.com/KhronosGroup/glTF-Sample-Models

Amazon Lumberyard Bistro
https://developer.nvidia.com/orca/amazon-lumberyard-bistro

Lee Perry-Smith Head Scan
http://graphics.cs.williams.edu/data/

Statue (“Lucy”) by Stanford Computer Graphics Laboratory
http://www.graphics.stanford.edu/data/3Dscanrep/#uses