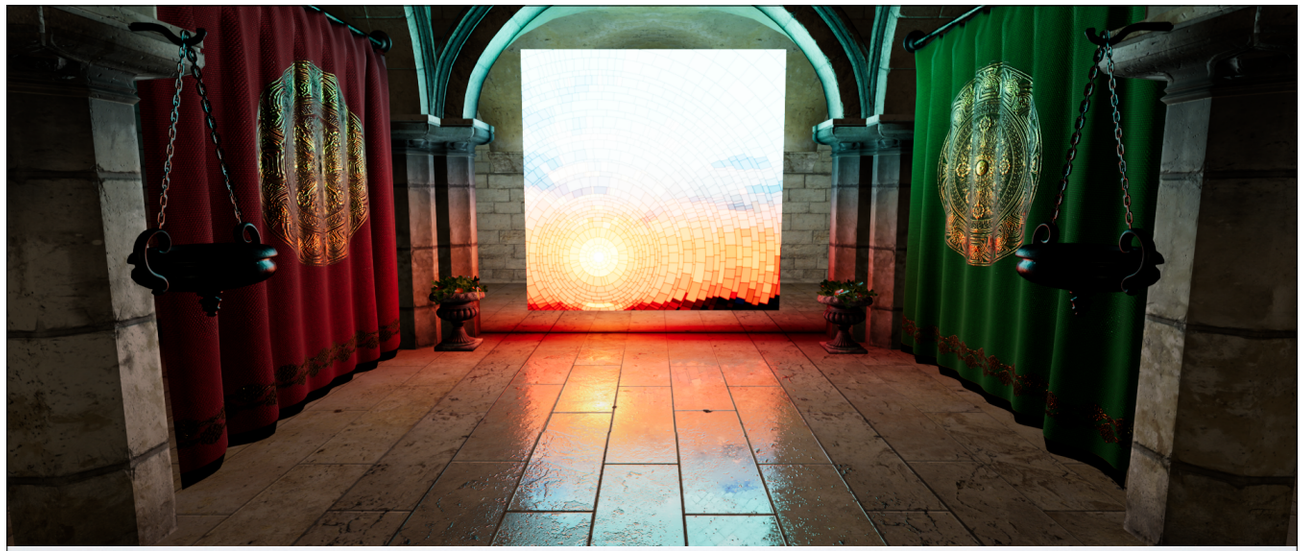
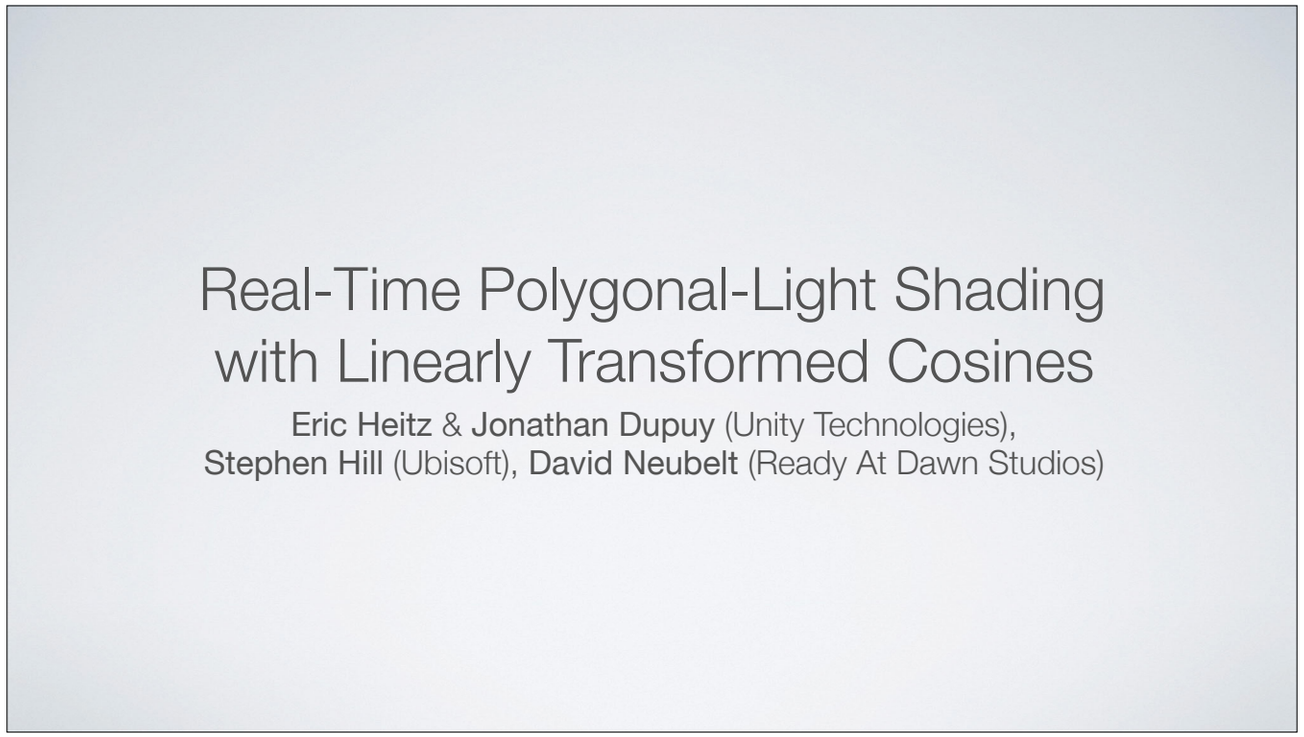


Real-Time Area Lighting: a Journey from Research to Production

Stephen Hill & Eric Heitz

In this talk, I’m going to cover joint work with Eric Heitz on streamlining the implementation…



…of this paper, which is about cracking the problem of doing area lighting in real time with a wide range of materials (not just diffuse).

As an aside, this paper was the result of a thoroughly enjoyable collaboration between Eric and Jonathan at Unity, Dave at Ready At Dawn, and myself at Ubisoft.

R & D

You could call it a real R&D effort…

Researchers
&
Developers

…researchers & developers! :)

Theory
&
Implementation

And this also reflects the content of this talk: a little bit of **theory** from the paper, plus some **implementation issues** to look out for.

Theory
&
Implementation

I’ll start, naturally, with the theory…

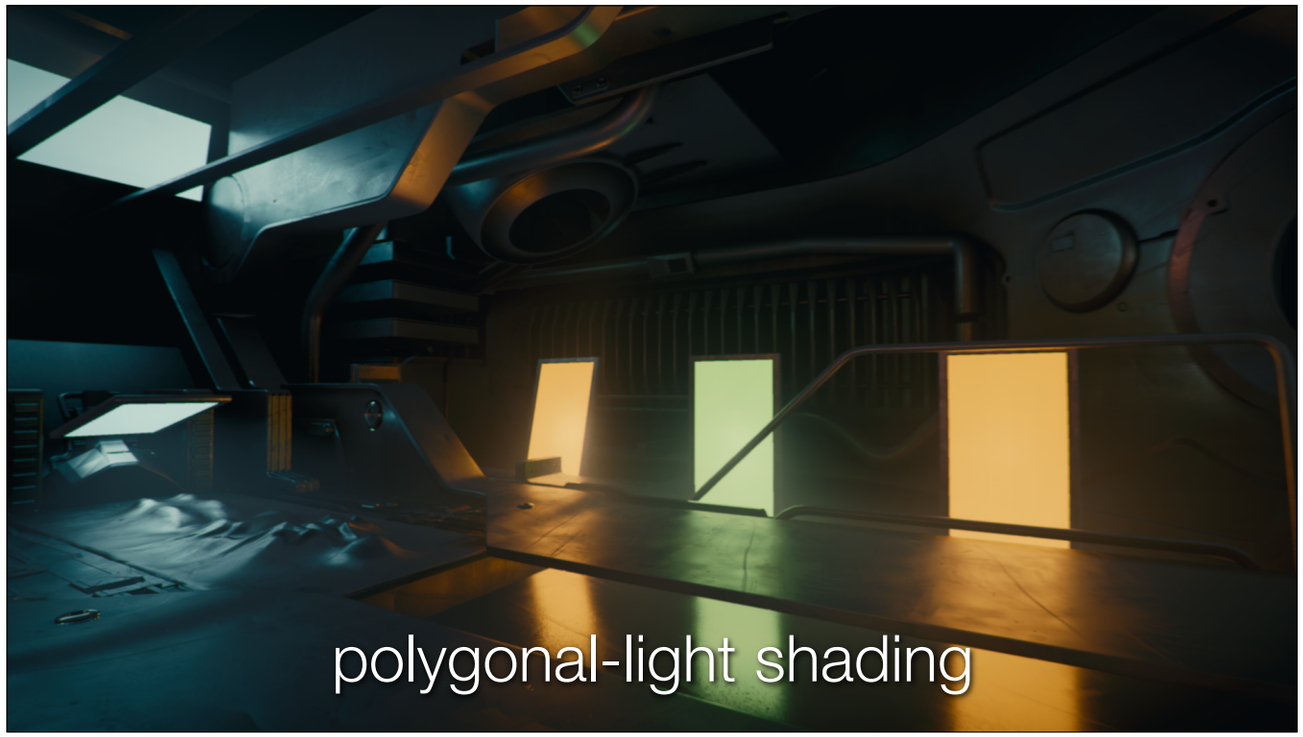
Real-Time Polygonal-Light Shading with Linearly Transformed Cosines

Eric Heitz & **Jonathan Dupuy** (Unity Technologies),
Stephen Hill (Ubisoft), **David Neubelt** (Ready At Dawn Studios)

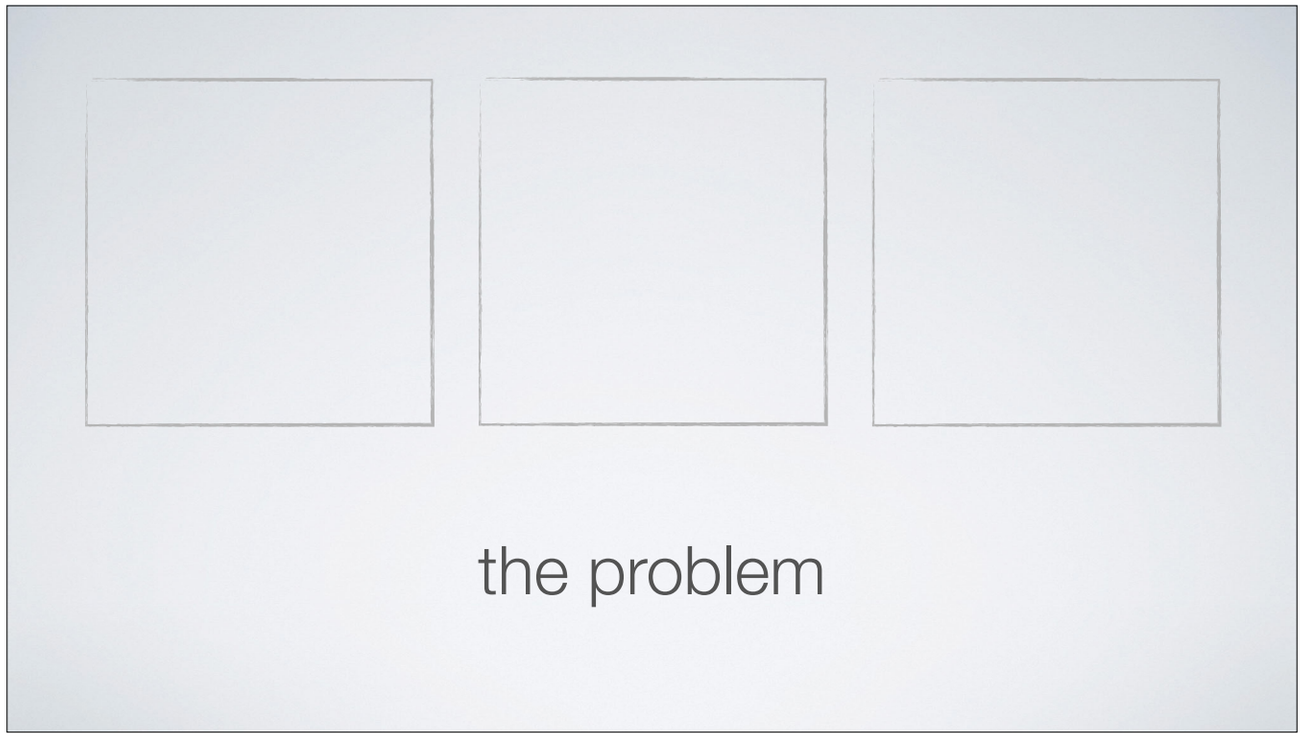
…but for a complete treatment, be sure to check out Eric's presentation of the paper:
https://eheitzresearch.wordpress.com/415-2/

Before I get into it, here’s a video of some results…

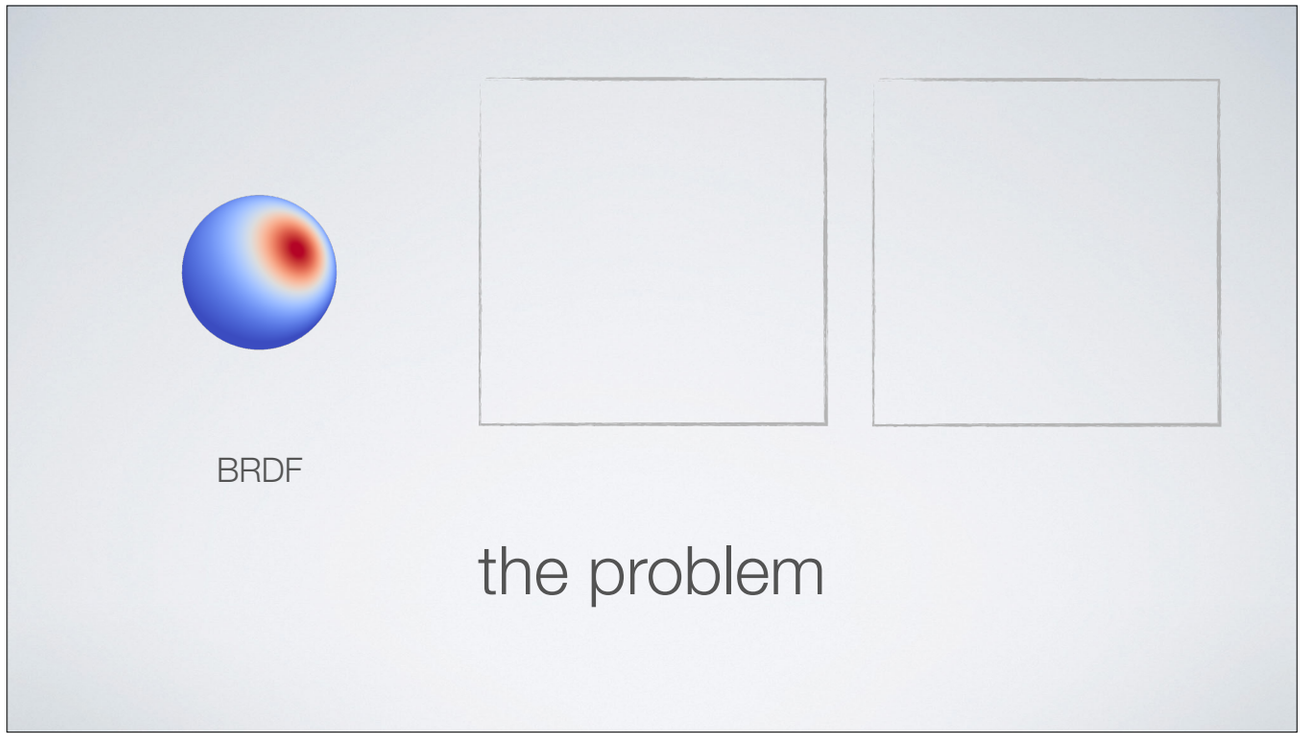
This is an early prototype that was together by Unity’s talented demo team. They picked up the technique and just ran with it, as you can see in the beautiful Adam demo: https://www.youtube.com/watch?v=GXI0l3yqBrA



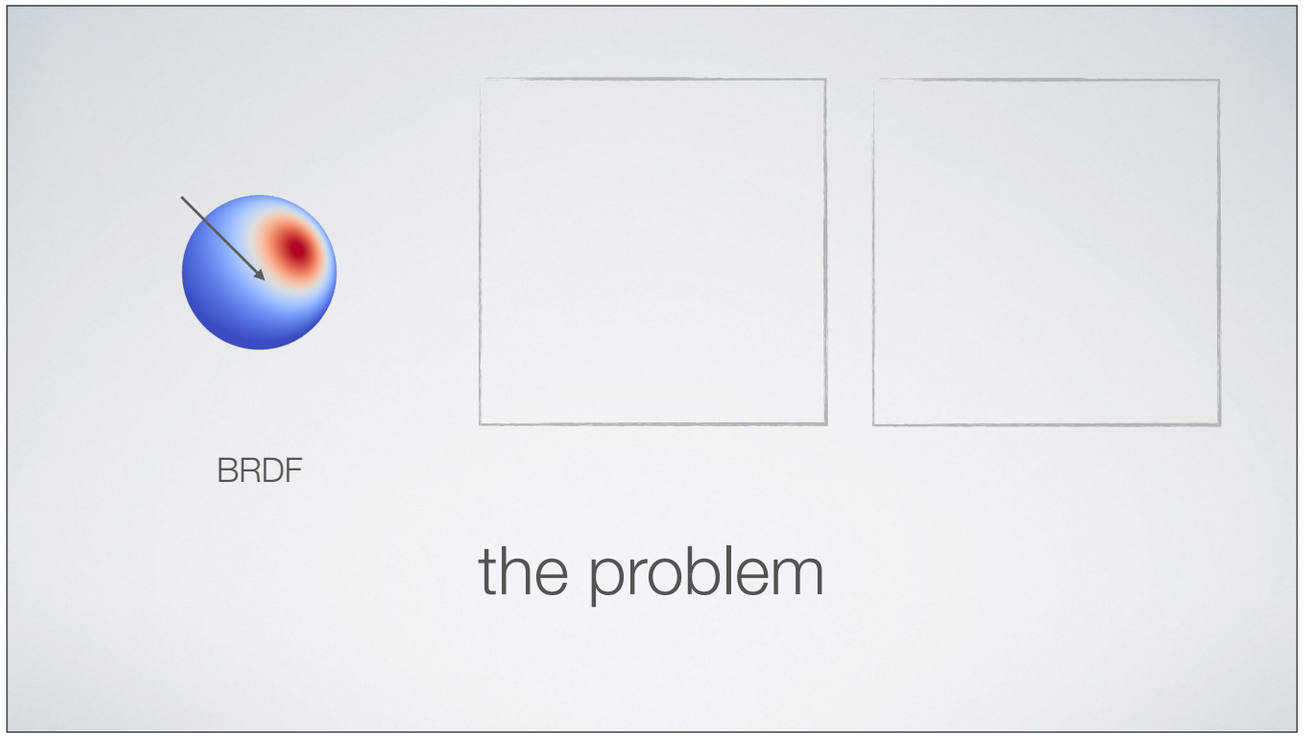
Okay, now down to the problem at hand: lighting with polygon sources.



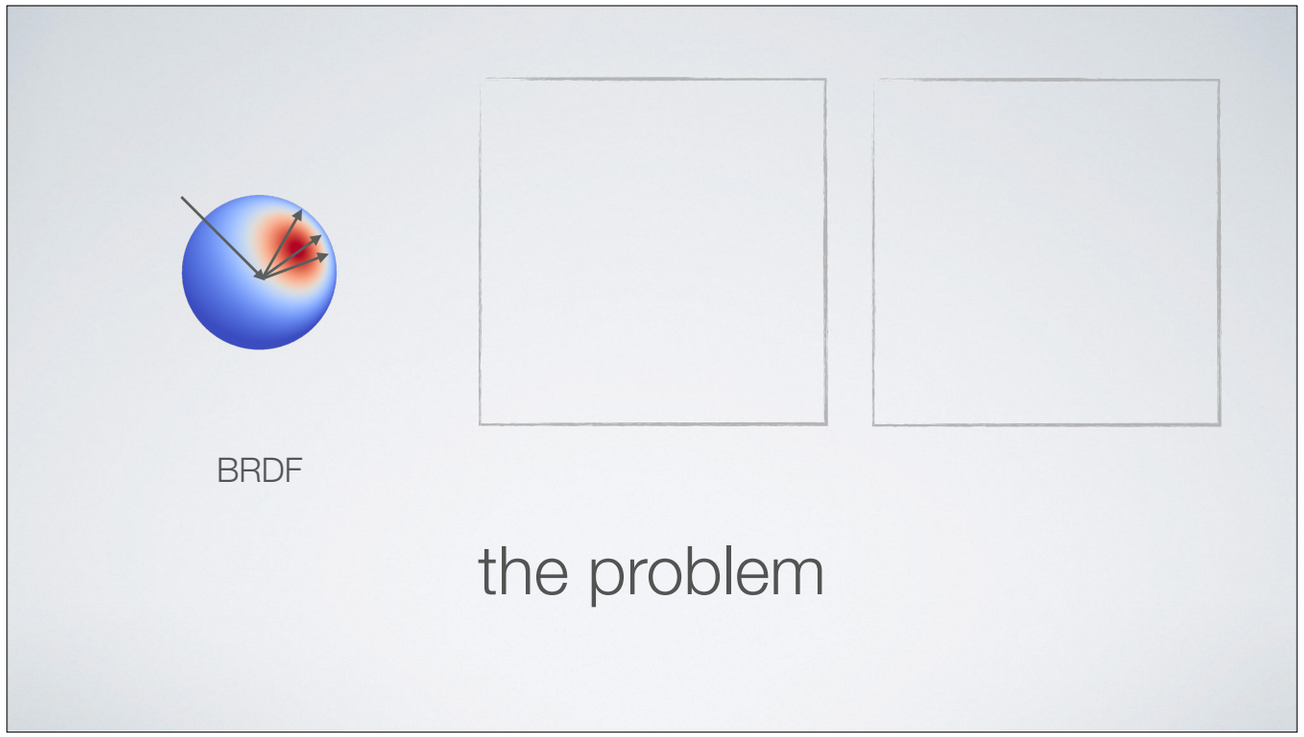
What are we really trying to solve here?



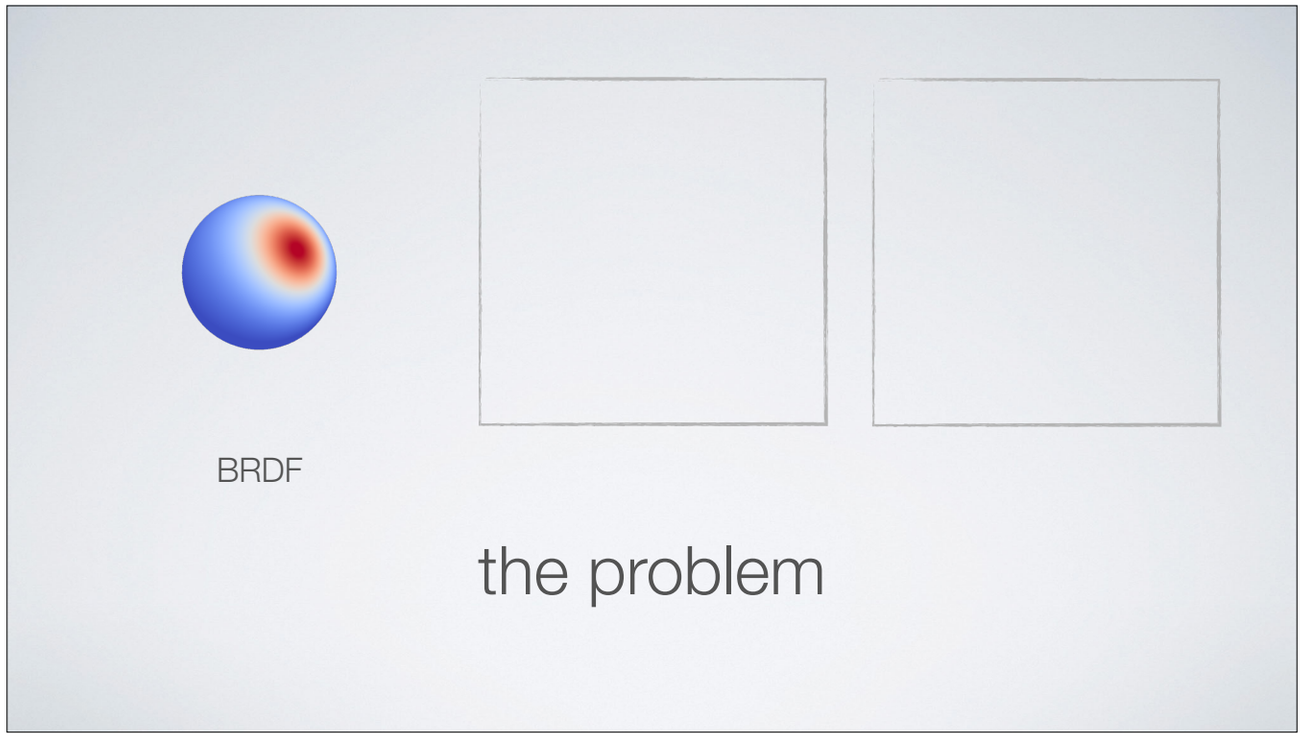
Firstly, we have the BRDF: a spherical function that describes how the material scatters light at a particular shading point.



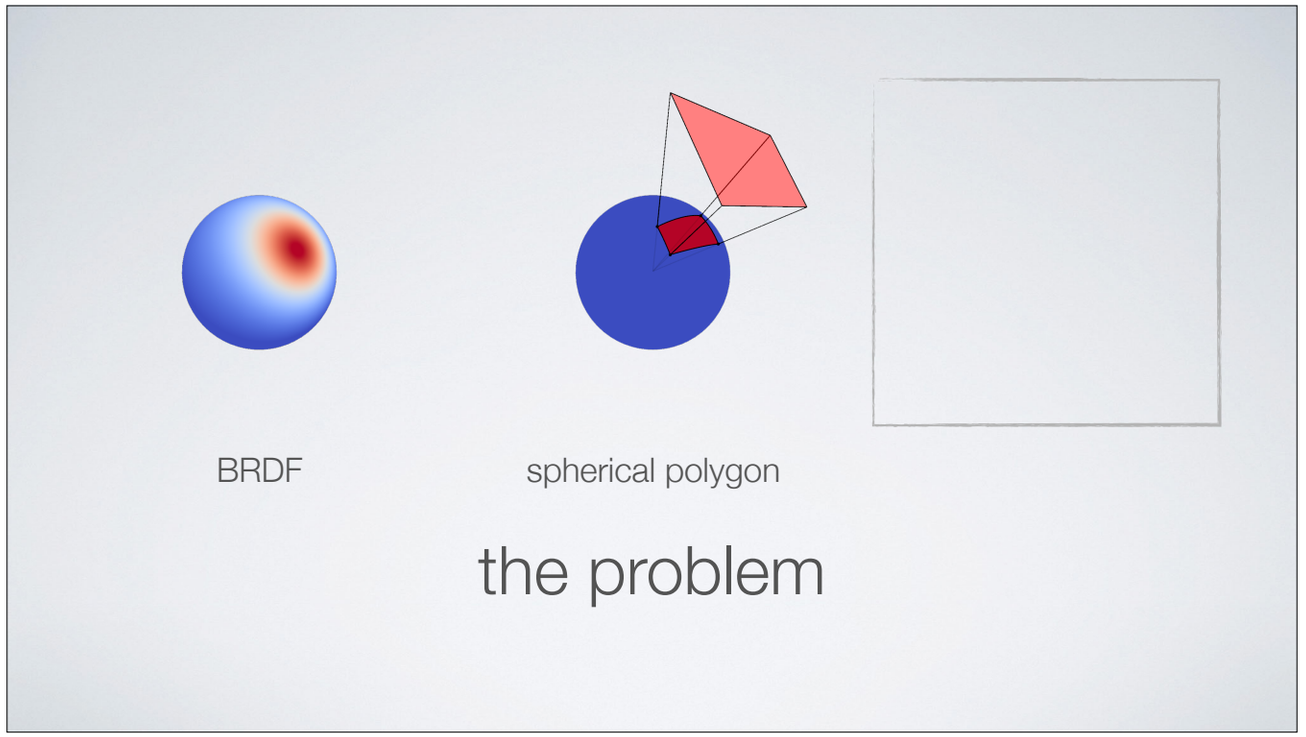
This plot is for a given view direction…
(or, reciprocally, a given light direction)



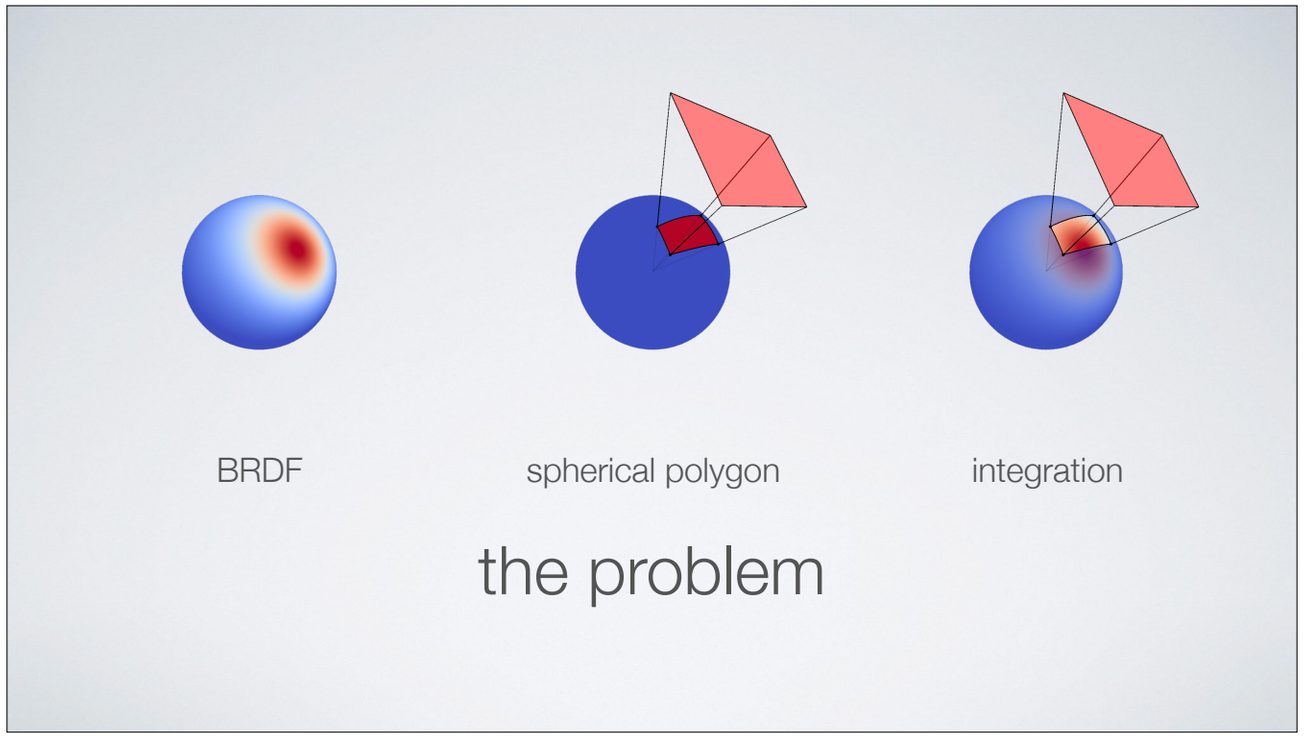
…and represents the directions from which light will scatter back to the eye.



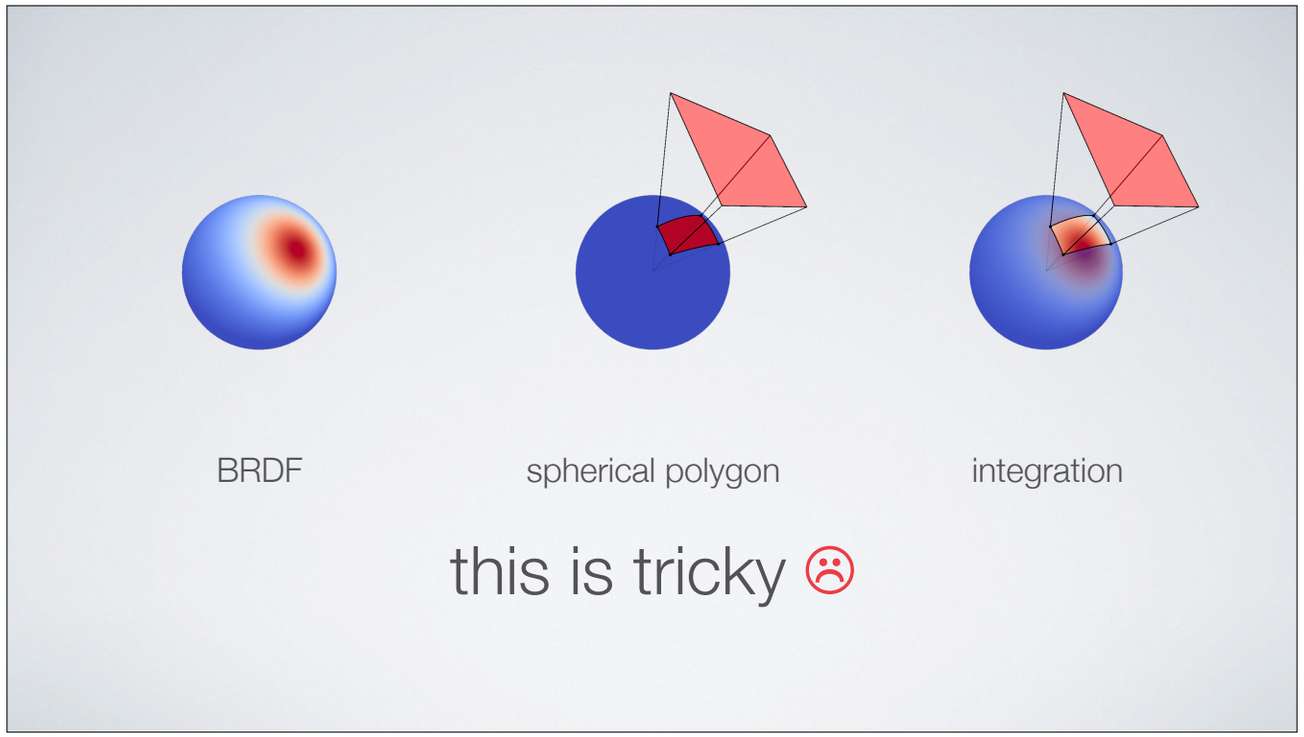
Note: while I'll be talking about BRDFs throughout the talk, this really could be any spherical function (the maths stays the same).



Secondly, we have lighting (incoming radiance) from a polygon that’s arriving at the shading point.



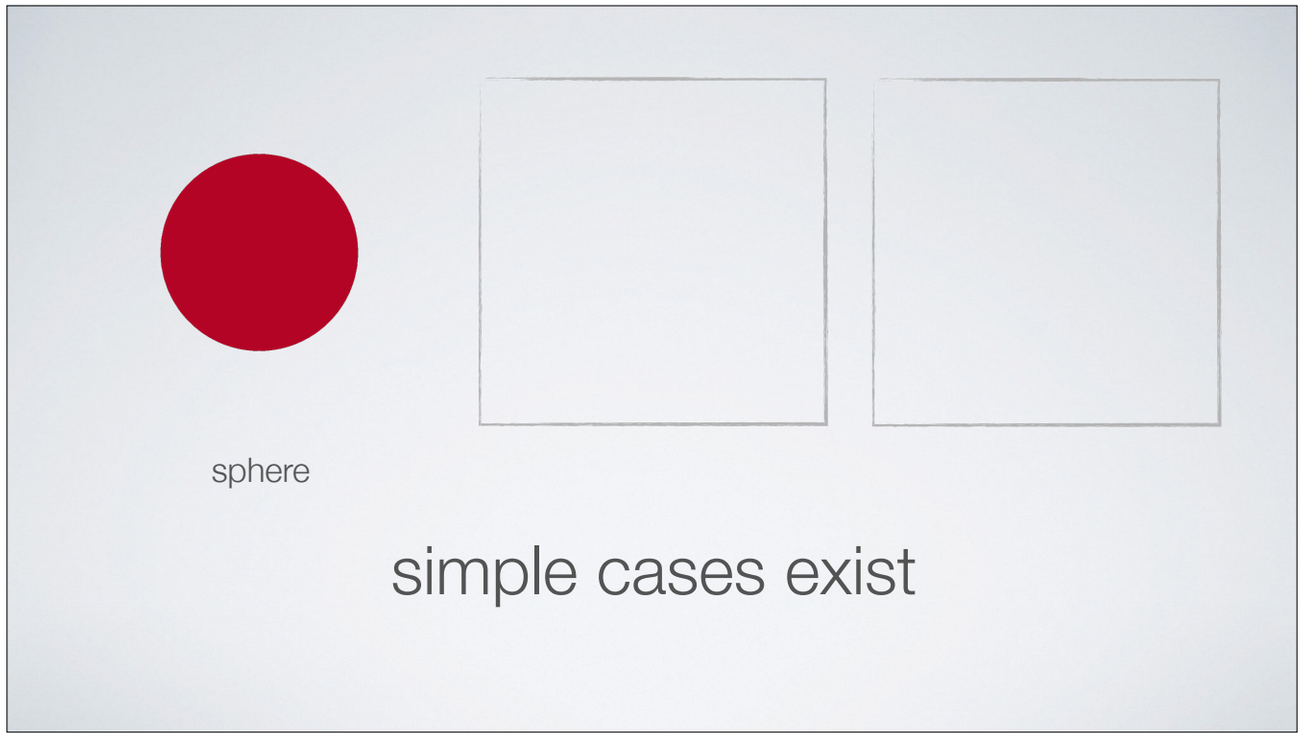
The shading result (outgoing radiance) is the integral of the BRDF over this spherical polygon.



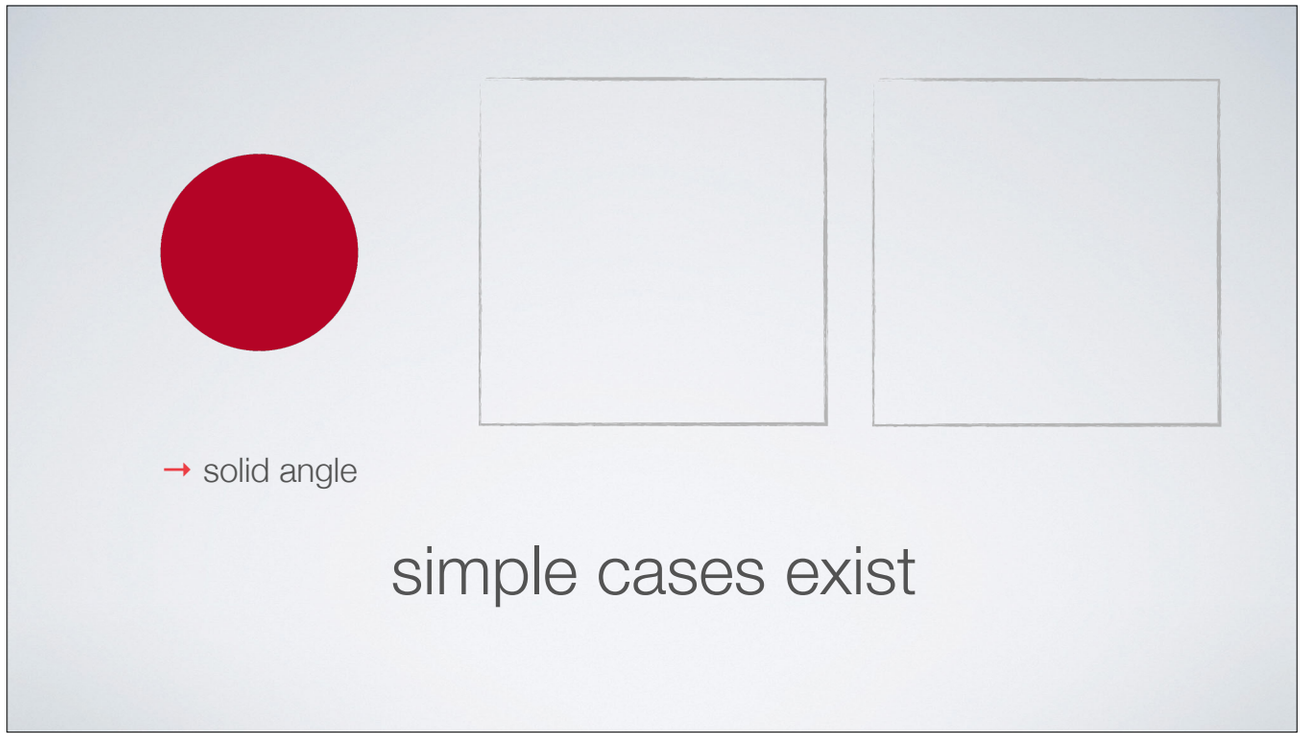
This is much harder than point lighting, where we only need to evaluate the BRDF for a single light direction. We now need to consider **many** directions.

In offline rendering, we might solve this with Monte Carlo sampling, but for real-time this isn't a viable option – it would either be too slow or too noisy.

We'd like to find a closed-form solution instead, i.e. an equation we can simply evaluate that will give us the right answer immediately without needing to sample.

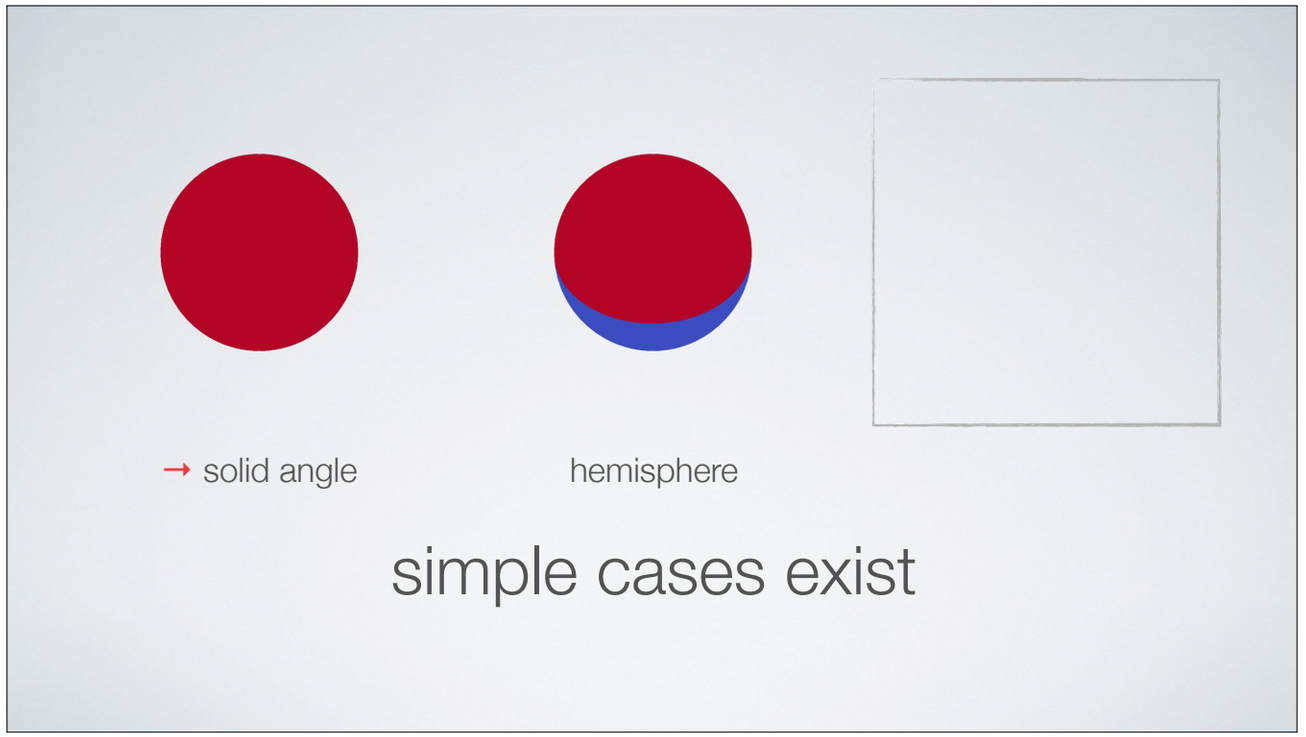


Simple distributions can be integrated over polygons in closed form. One example is the uniform spherical distribution.

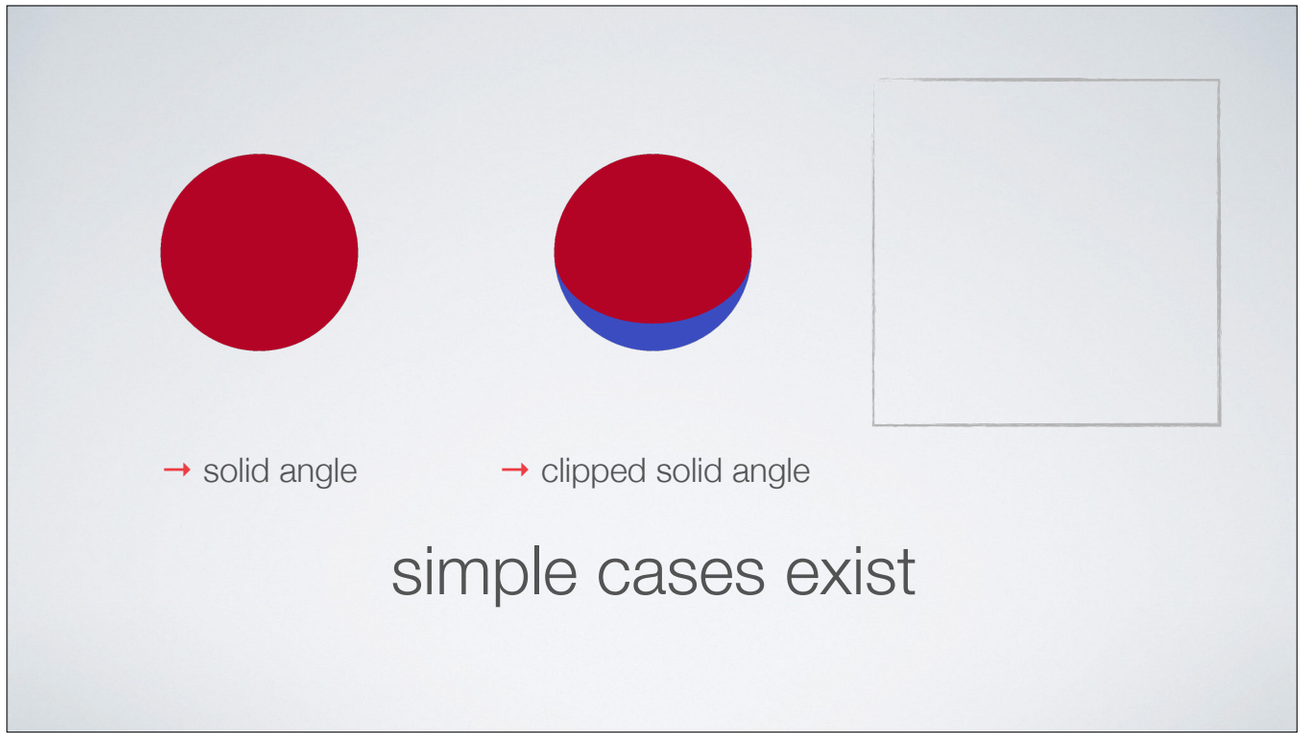


Its integral over a spherical polygon is equivalent to computing the solid angle of the polygon.

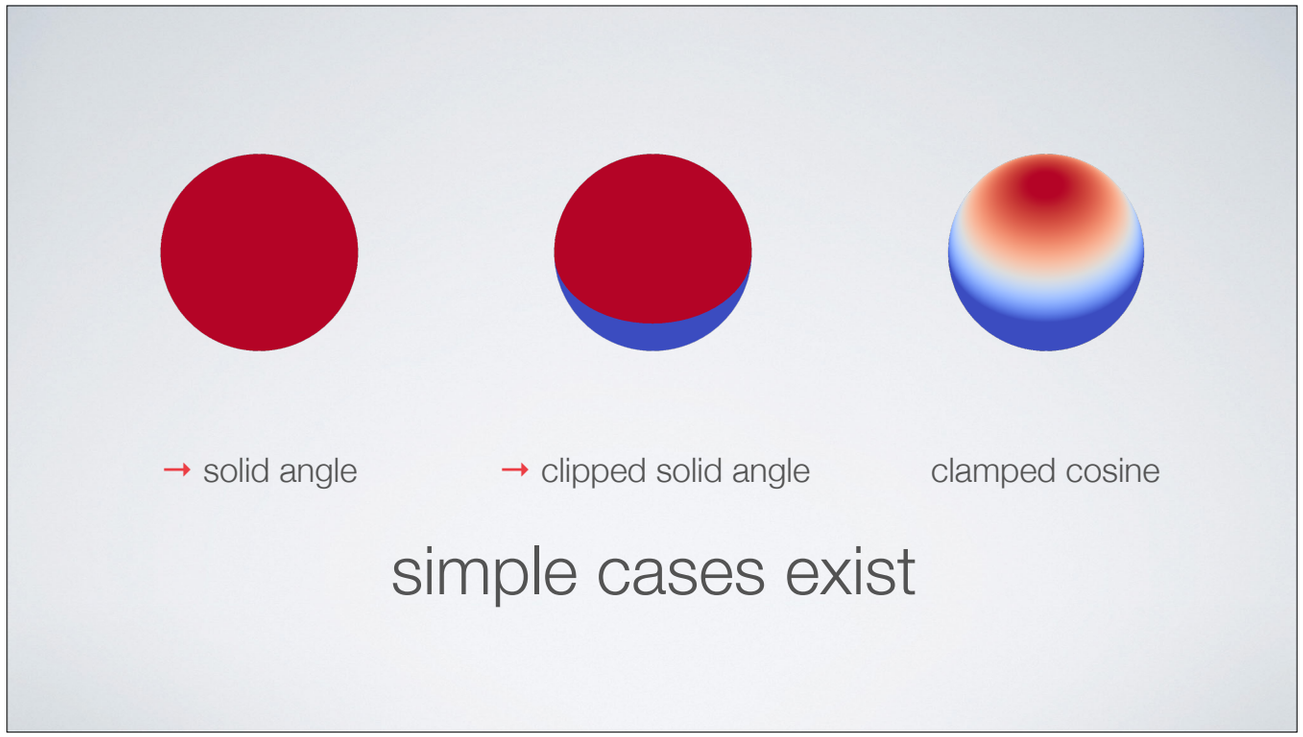
There is a closed-form expression for this: Girard’s theorem.



Another simple example is the uniform hemispherical distribution.

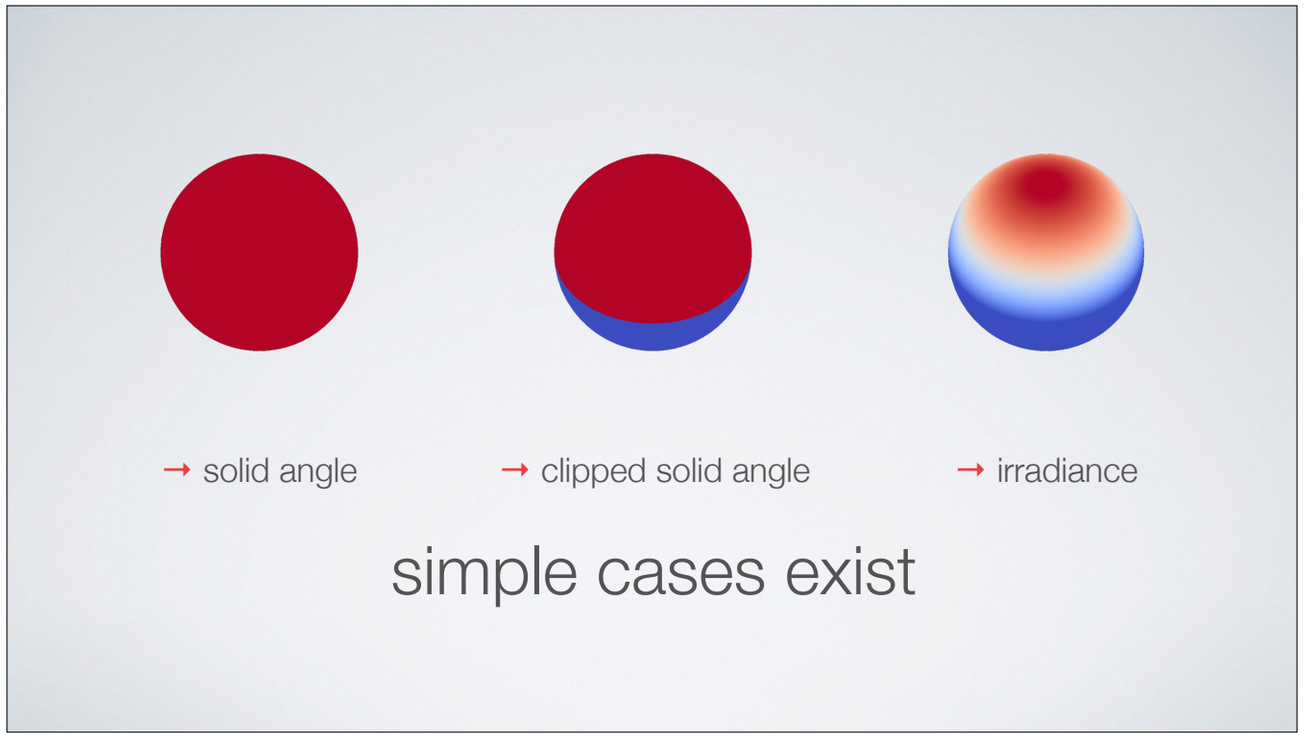


This is simply the solid angle of the polygon clipped to the hemisphere.



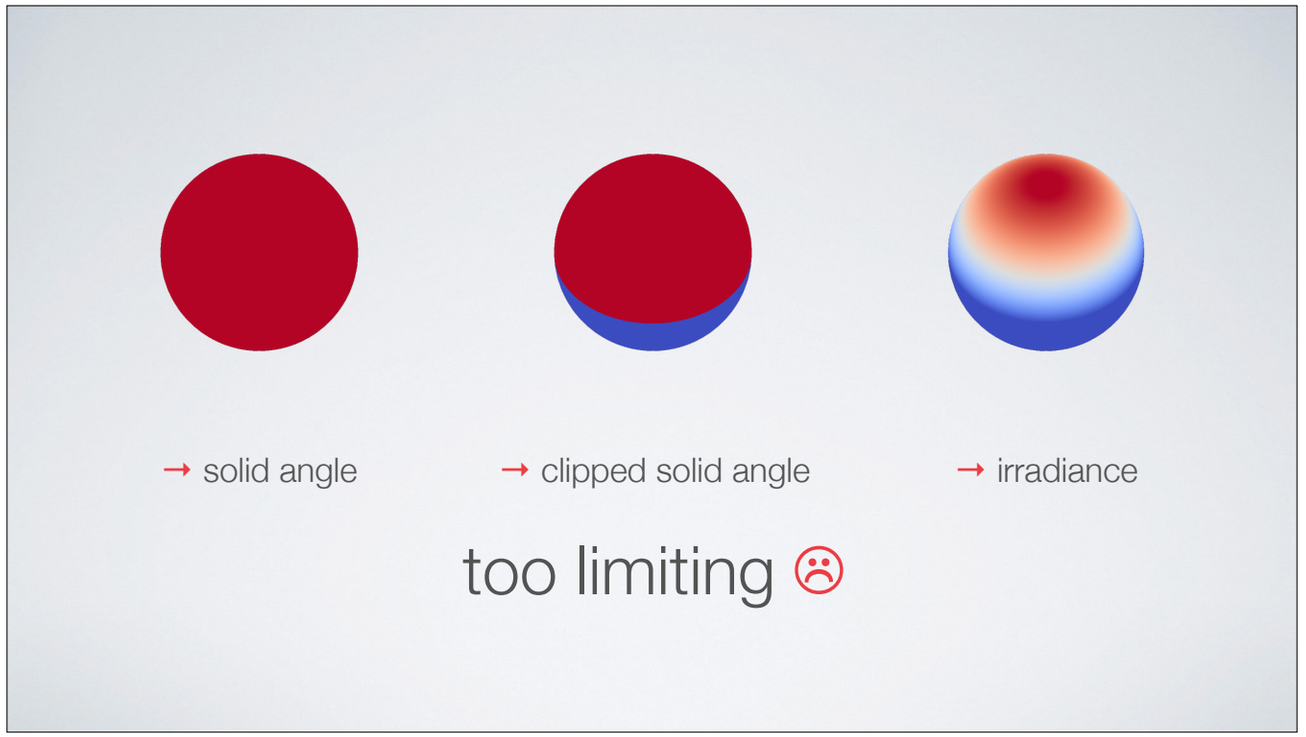
A more interesting example is the cosine distribution.

You can also call it *diffuse* or *Lambertian*.

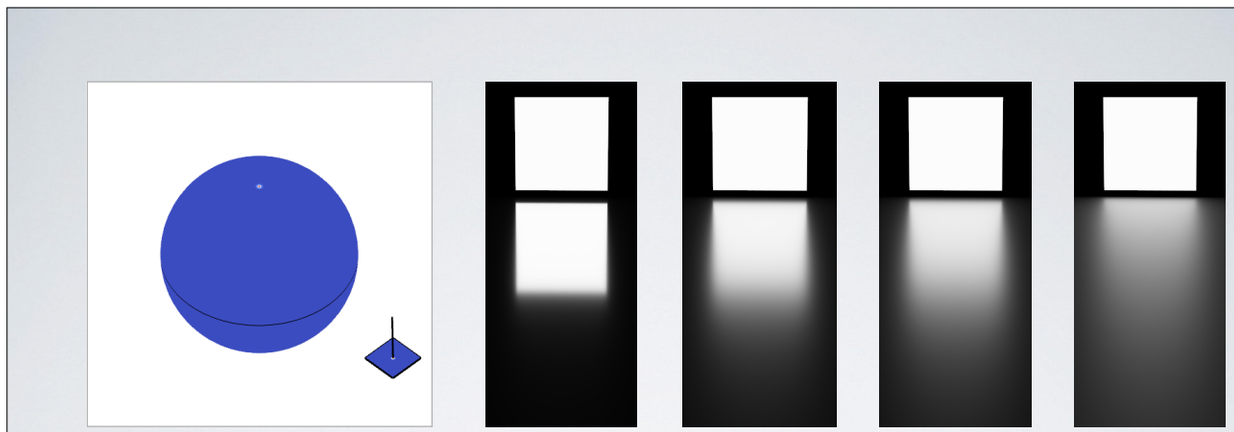


Computing its integral over the polygon gives the irradiance (or form factor).

Again, there is a closed-form expression for it, derived by Lambert in the 18th century! More on this later.

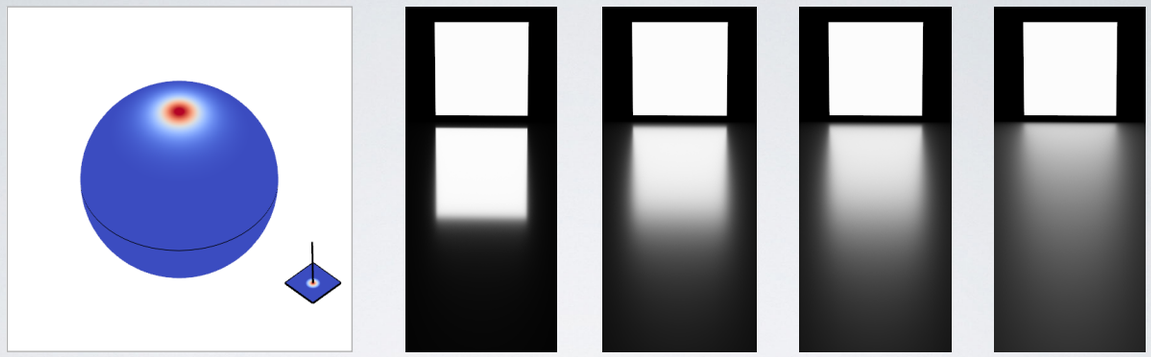


And that’s about it! Unfortunately these solutions are too limiting for what we need.



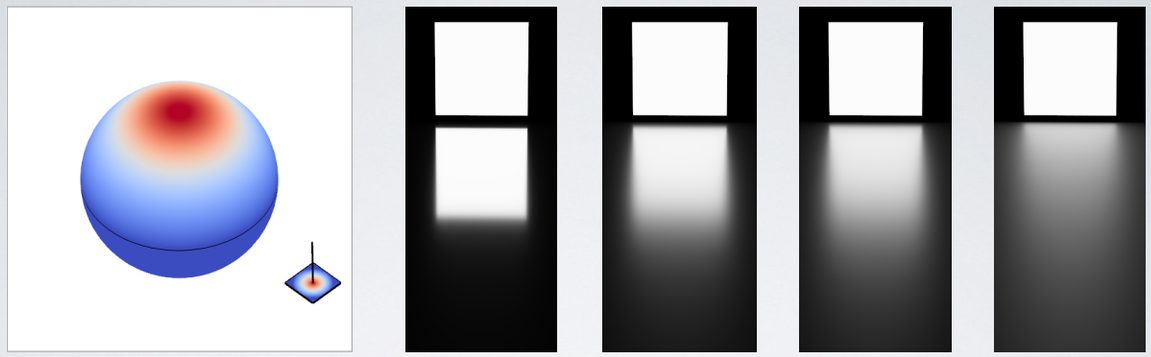
We’d really like to be able to represent a wider range of materials, from mirror-like, to semi-glossy, to rough.

This is a bare minimum of what we’d expect from a real-time shading model in terms of expressiveness. Giving this up in order to accommodate area lights doesn’t make sense.



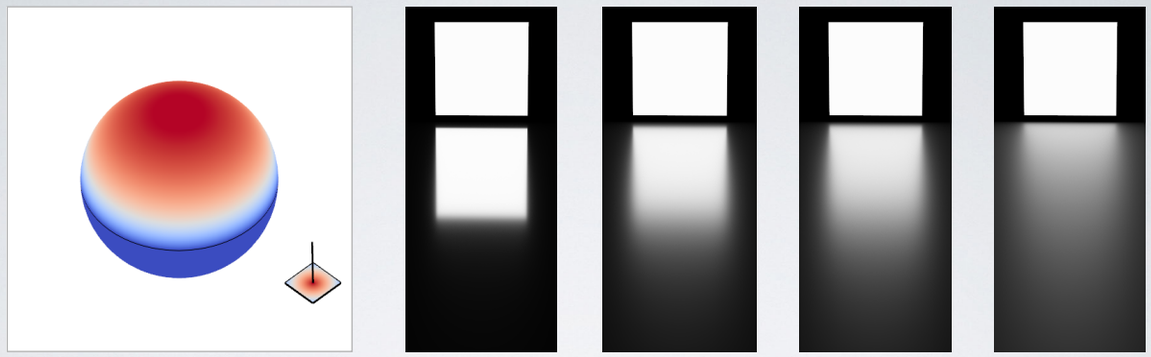
varying roughness

we want *all frequencies*



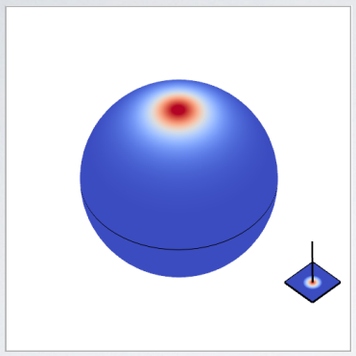
varying roughness

we want *all frequencies*



varying roughness

we want *all frequencies*

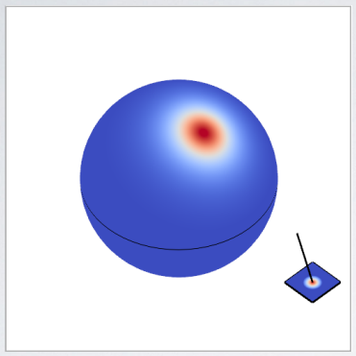


we want *anisotropy*

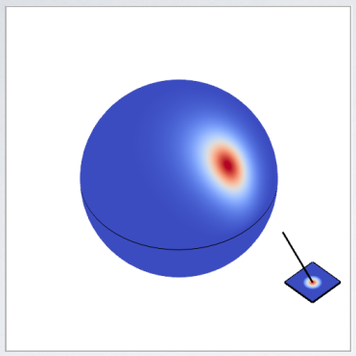
Real-world materials also exhibit strong anisotropy (or ‘stretched highlights’) at grazing angles.

Industry-standard microfacet models are able to reproduce this convincingly. It’s an effect that we’re used to seeing from real-time implementations with punctual light sources, so we’d love to be able to achieve the same behaviour with polygonal area lights.

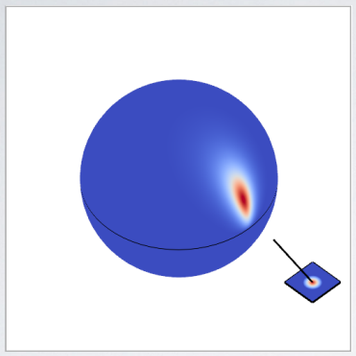
Image credit: Grand Rapids Press.



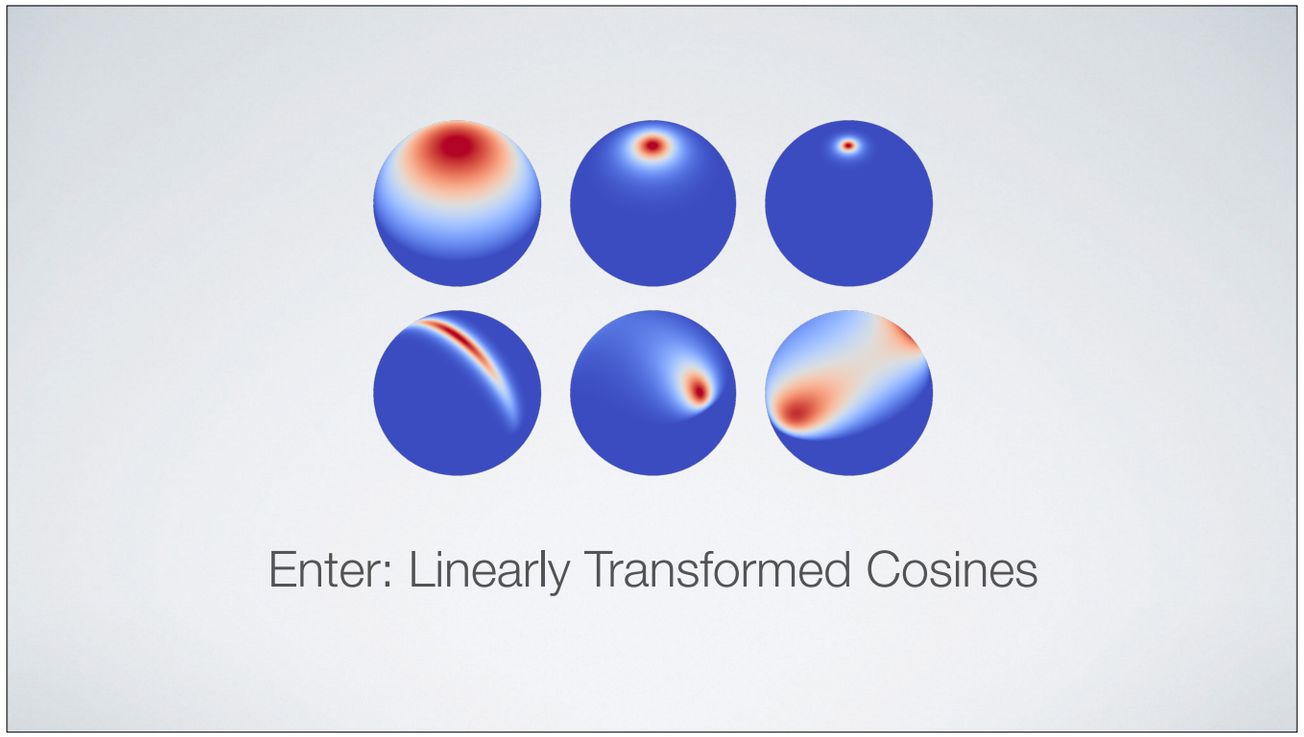
we want *anisotropy*



we want *anisotropy*



we want *anisotropy*

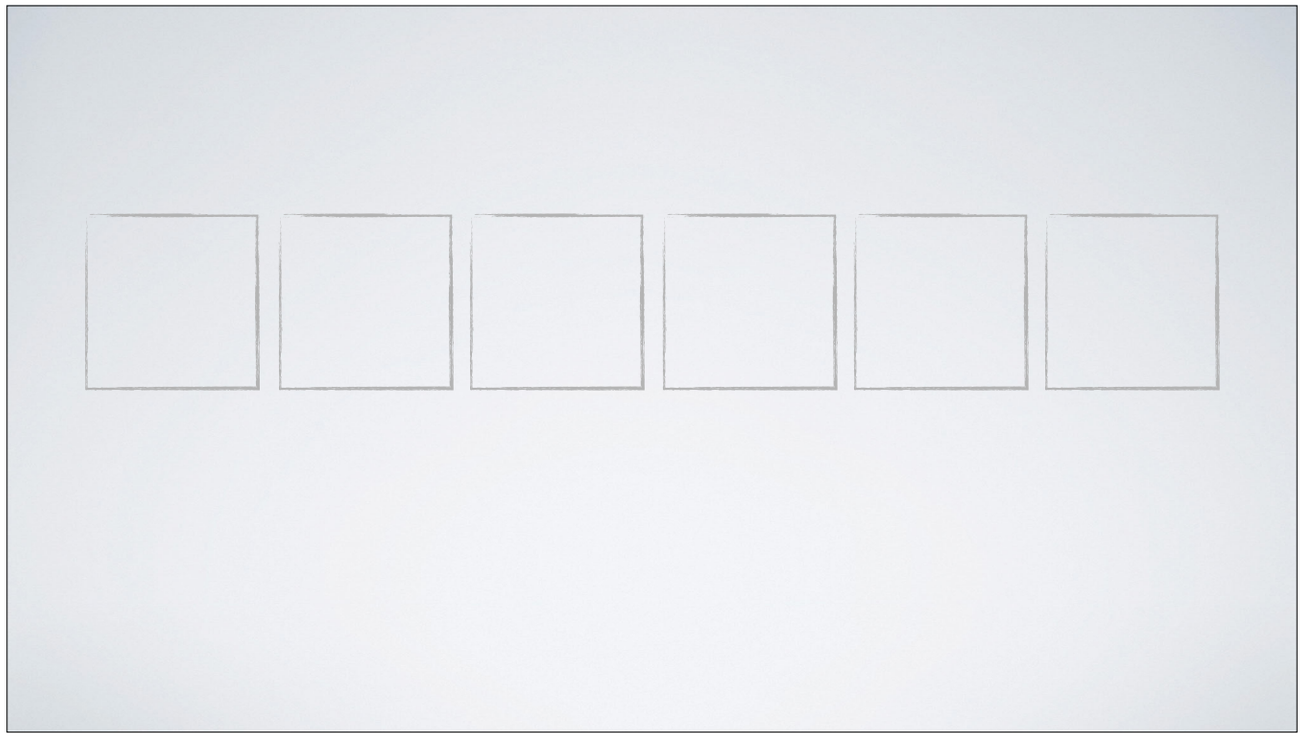


So to recap, we'd like to evaluate the product of a general BRDF and a polygonal light source in a fast and noise free way, but there's currently no way to do this.

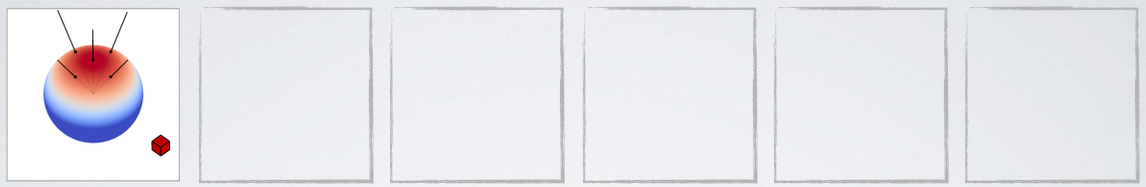
That's been a longstanding roadblock and it would be disappointing if we had to stop the presentation here. :)

Fortunately, we managed to find a solution to this problem: *Linearly Transformed Cosines* (LTCs).
This is the core contribution of our paper.

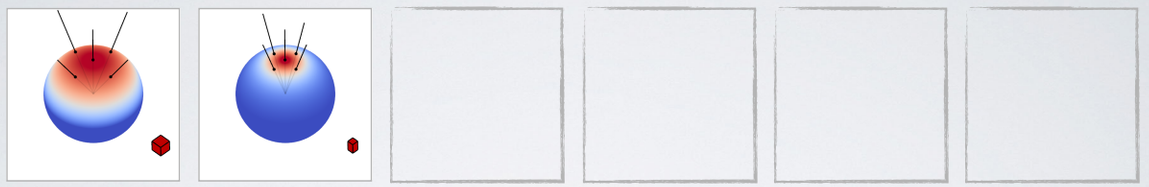
I'll now give you a high-level overview of LTCs, but (again) please refer to Eric's slides for more details.



Essentially, the main idea is to take a simple distribution and apply a linear transform to it. In doing so, we can create a wide range of more sophisticated ‘shapes’ (spherical functions).

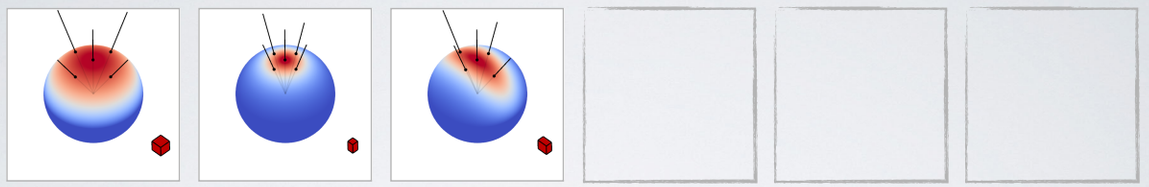


Let’s start with the clamped cosine distribution mentioned earlier…

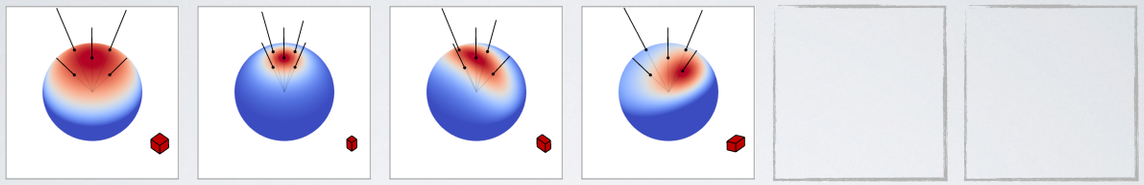


$$\begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

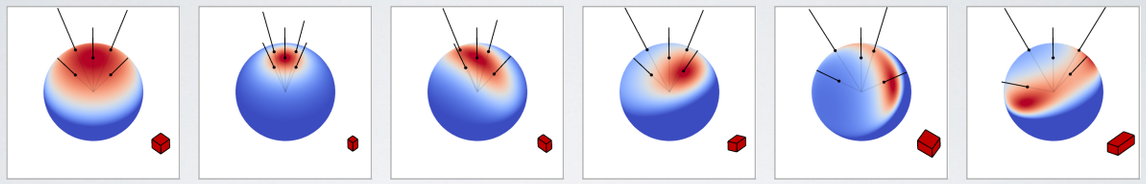
If we apply a uniform scale to x and y, we can vary the roughness of the distribution.



With a different factor for x and y we can create anisotropy.



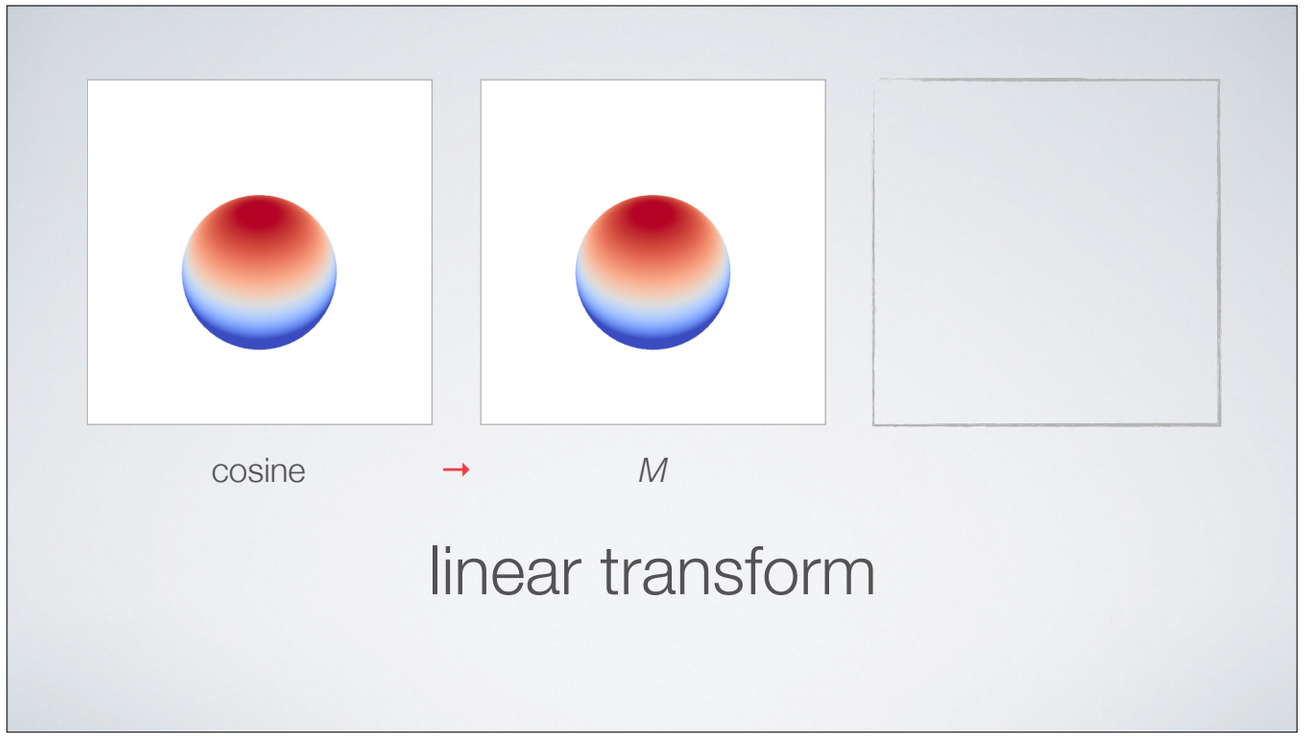
and through the bottom-left element of the transform we can introduce ‘skewness’.



$$\begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \lambda_x & 0 & 0 \\ 0 & \lambda_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & 0 & 1 \end{bmatrix}$$

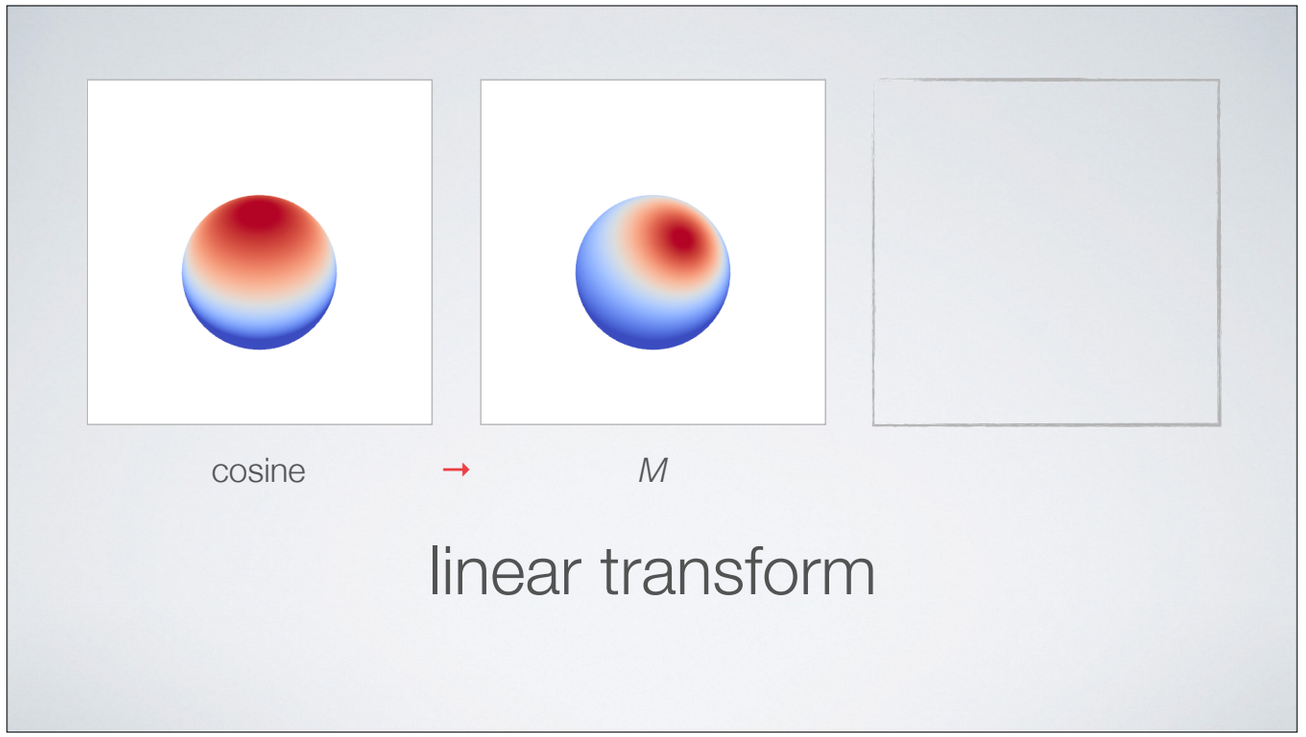
We can even create all sorts of wacky behaviour with random transforms, some of which even lead to bimodal distributions.

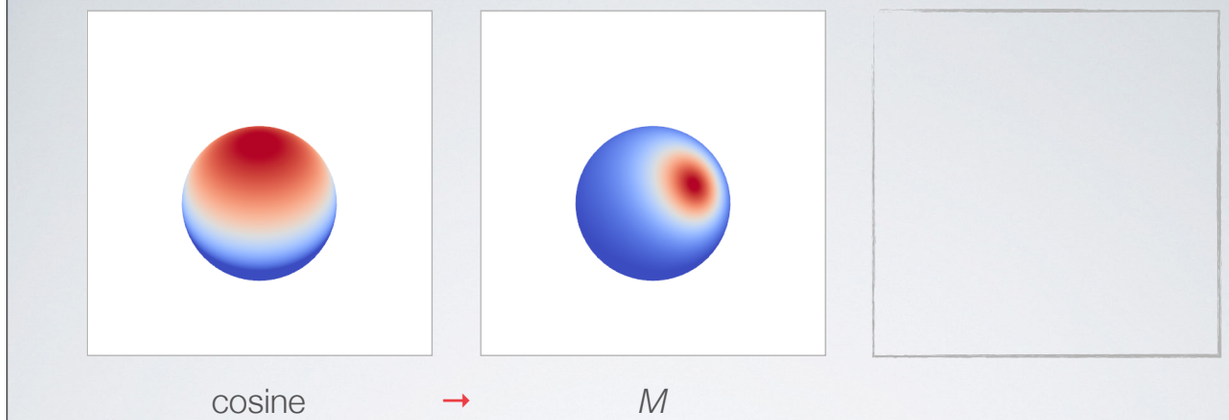
So, as you can see, this approach is very expressive.



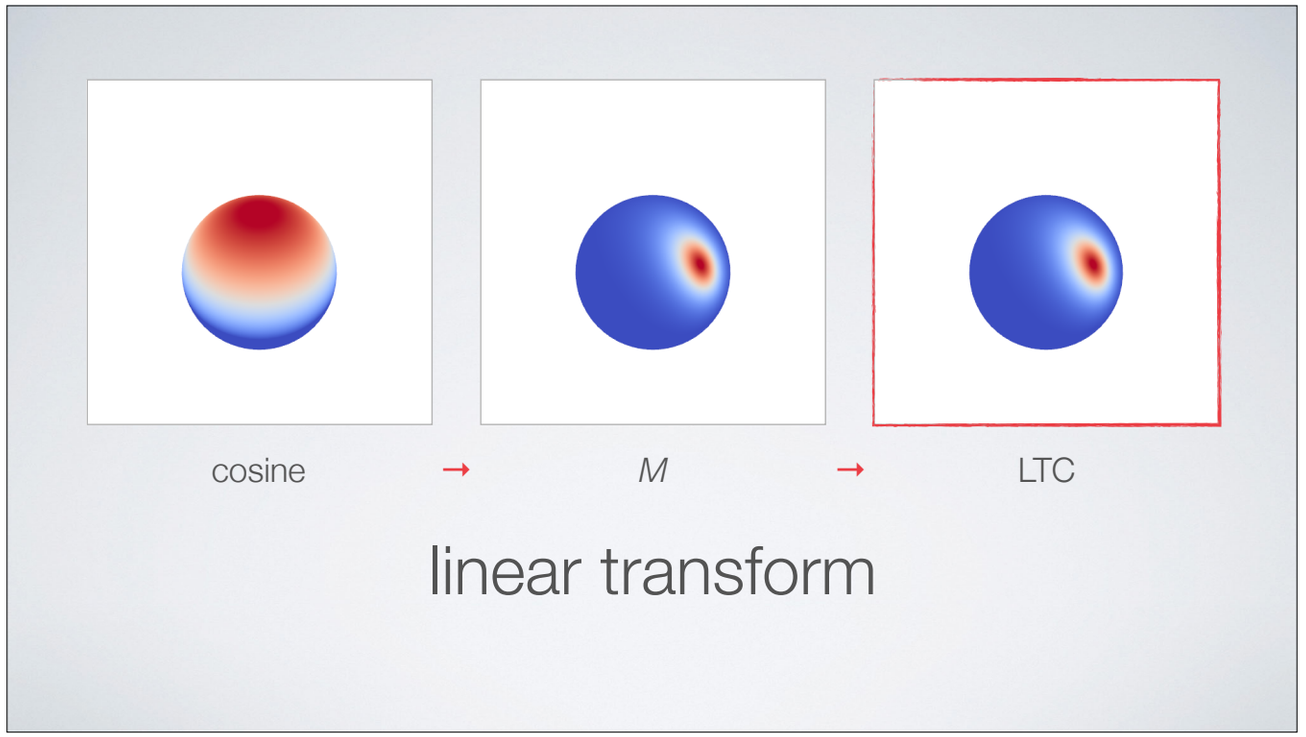
In general we can take a cosine distribution and apply an arbitrary 3x3 matrix, M, producing a new distribution.

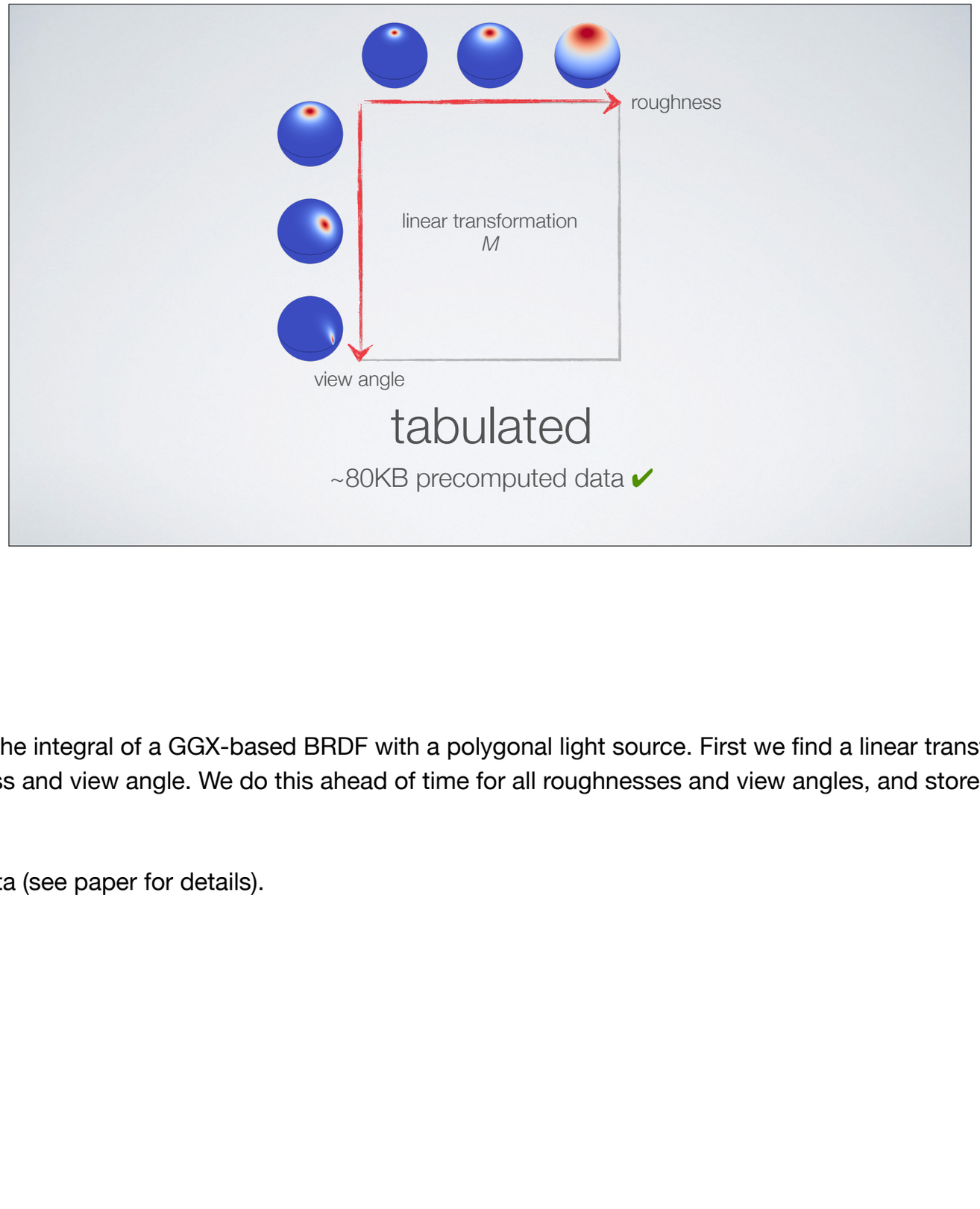
In the paper we refer to the **family** of distributions generated in this way as *Linearly Transformed Cosines*.


cosine
→
M
linear transform



linear transform

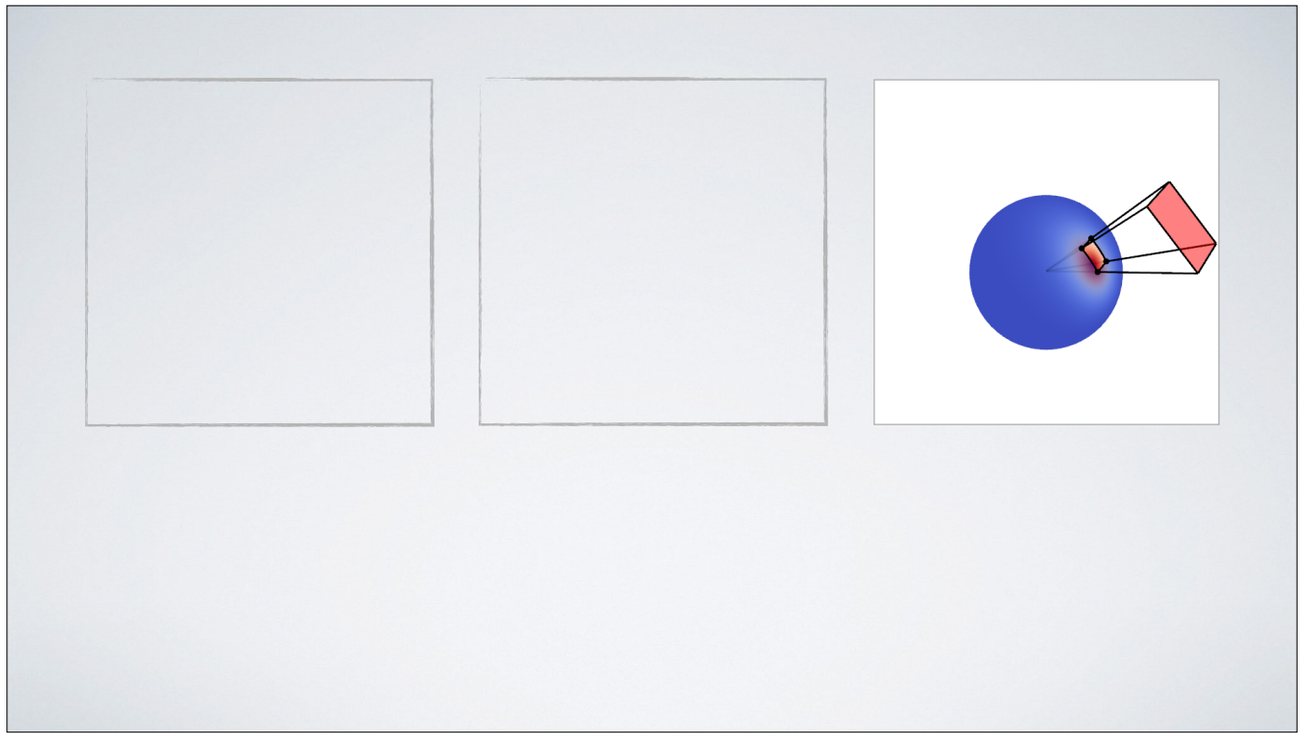

cosine
M
LTC
linear transform



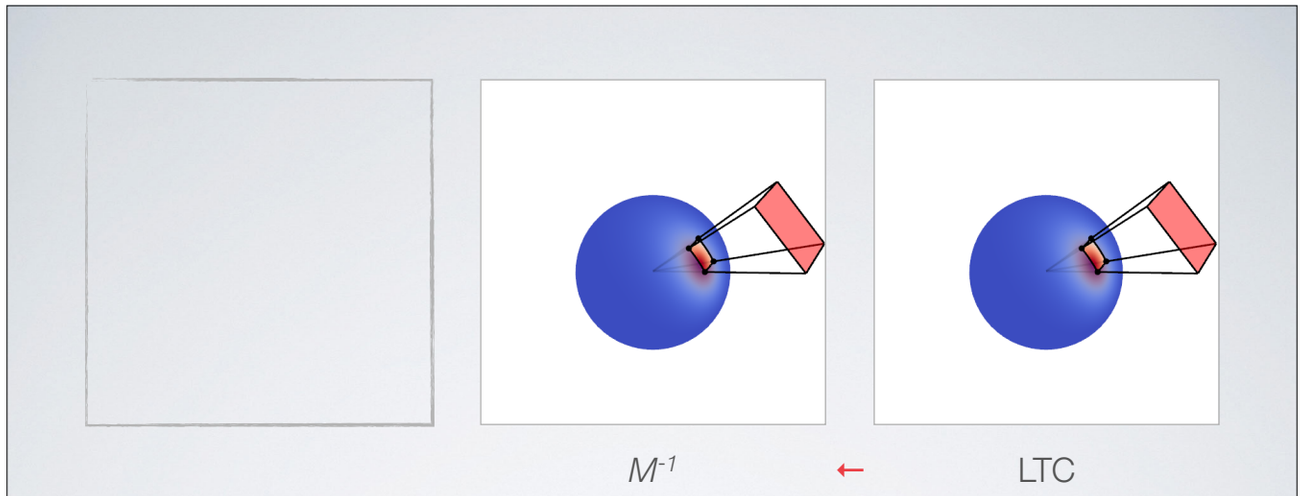
So how to we use this in practice?

Well let’s say that we want to compute the integral of a GGX-based BRDF with a polygonal light source. First we find a linear transform M that best approximates this BRDF with an LTC, for a given roughness and view angle. We do this ahead of time for all roughnesses and view angles, and store the resulting matrices in a table (= texture).

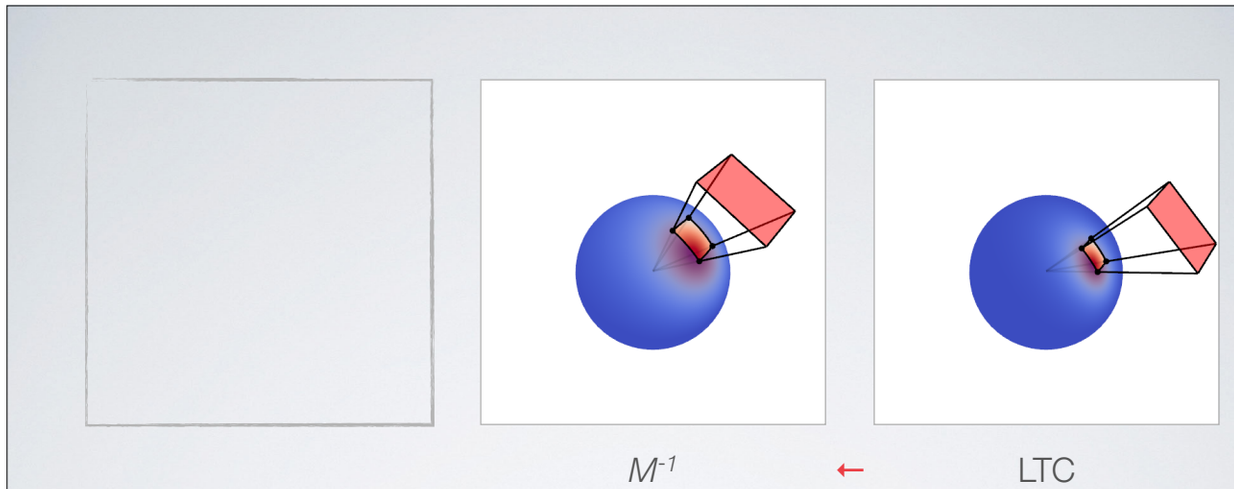
In practice we don’t need that much data (see paper for details).



Now for the ‘magic trick’. At runtime we take our BRDF-polygon configuration…

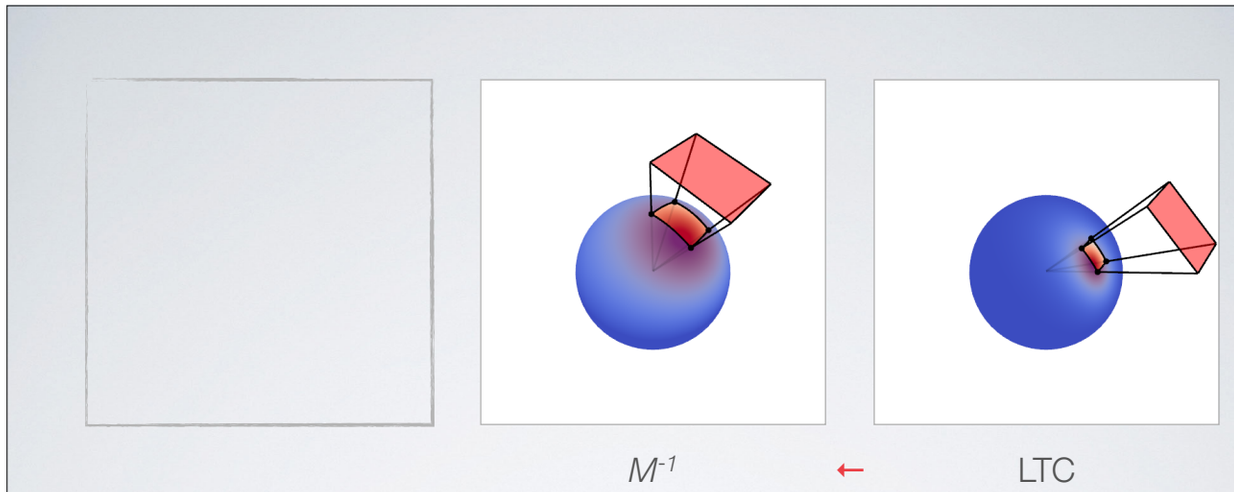


and apply the inverse transform to the vertices of the polygon, based on our LTC fitting of that BRDF (for the view angle and roughness at the current shading point).



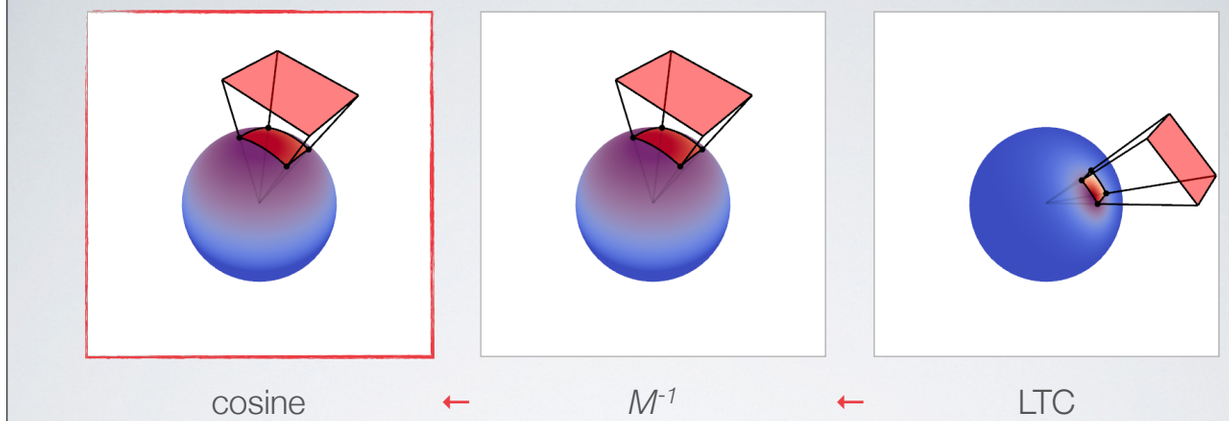
inverse transform

and apply the inverse transform to the vertices of the polygon, based on our LTC fitting of that BRDF (for the view angle and roughness at the current shading point).



inverse transform

and apply the inverse transform to the vertices of the polygon, based on our LTC fitting of that BRDF (for the view angle and roughness at the current shading point).

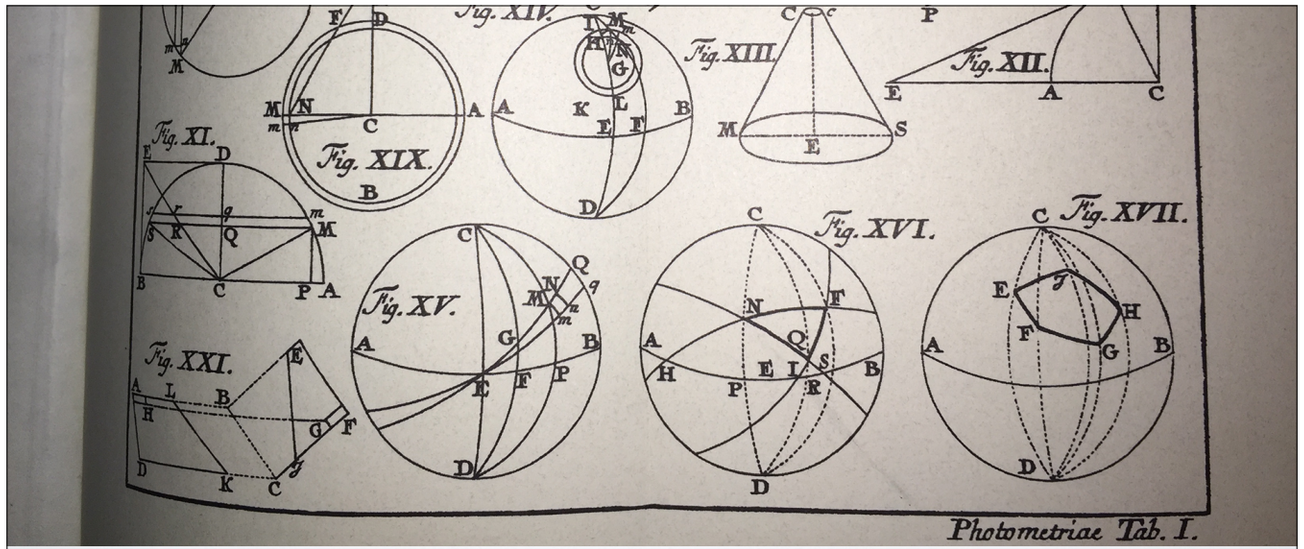


cosine integration

we know how to solve this ✔

This turns the configuration into an equivalent but simpler integration problem.

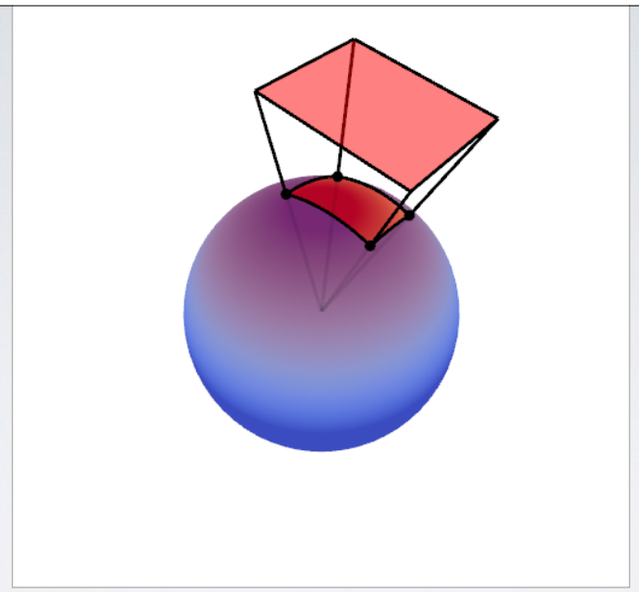
You can think of this as transforming back to ‘cosine space’.



Lambert, 1760!

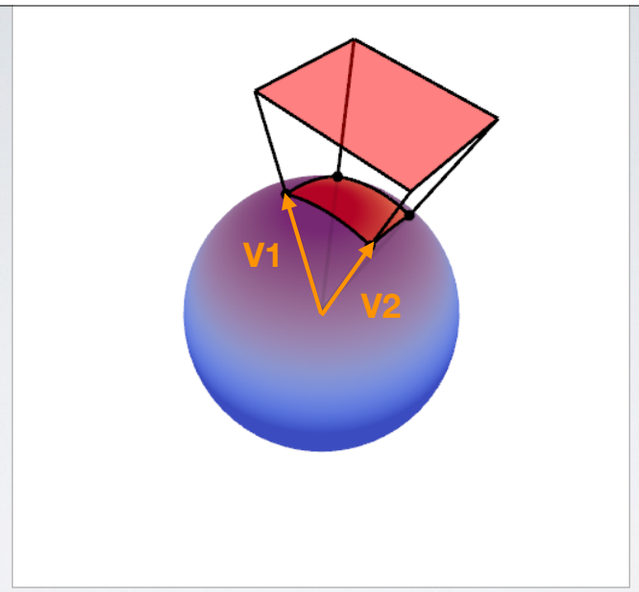
And we know how to solve this thanks to the work of Johann Heinrich Lambert (AKA ‘Mr NdotL’), from all the way back in the 18th century.

Image credit: Photometria, Lambert.



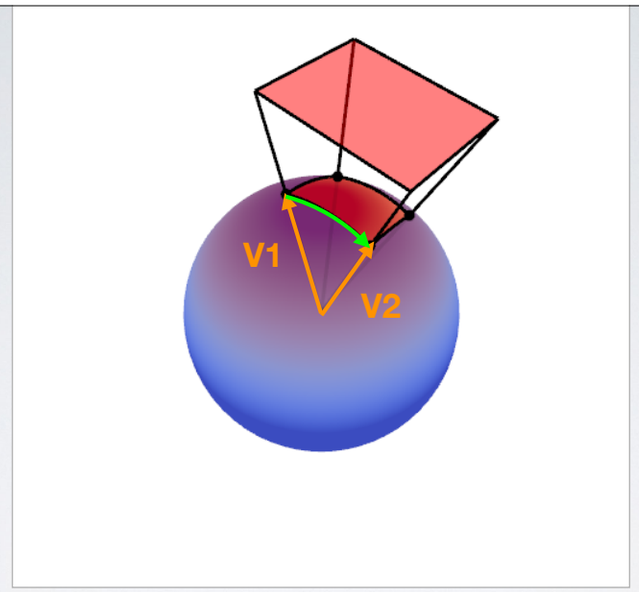
area integral → edge integrals

In practice we don’t calculate the area integral directly but instead evaluate a series of line/edge integrals over the boundary of the spherical polygon (one for each edge).



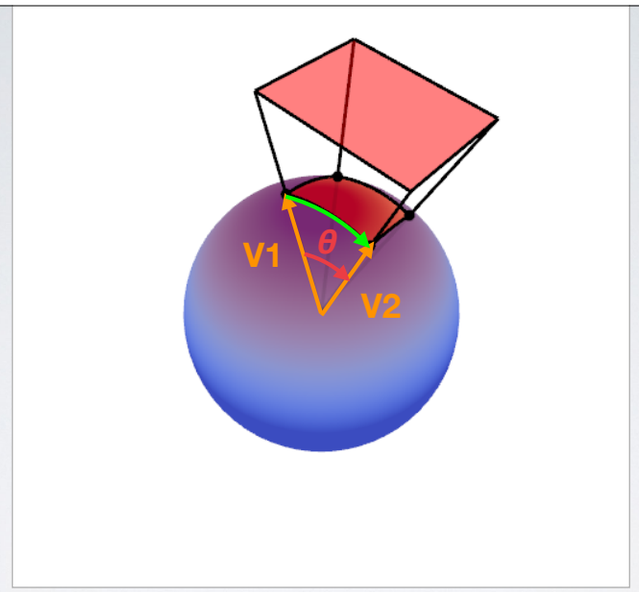
area integral → edge integrals

For a given edge, from two vertices v1 and v2…



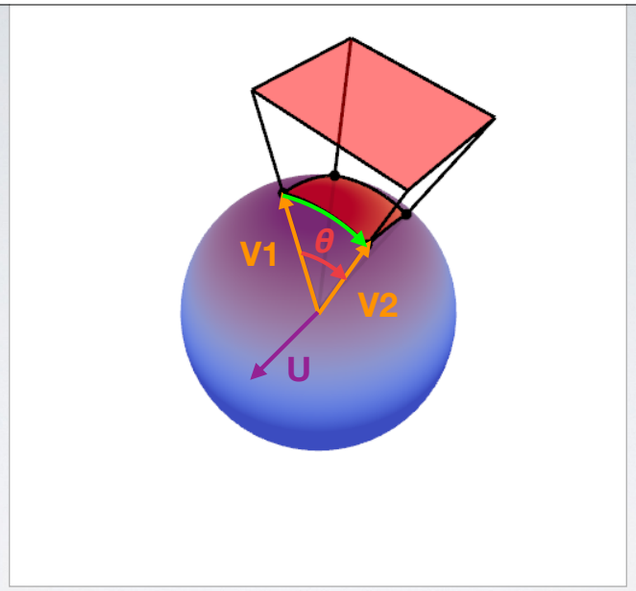
$$\operatorname{acos}\left(v_1 \cdot v_2\right) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

…we’re computing a 1D spherical line integral (green).



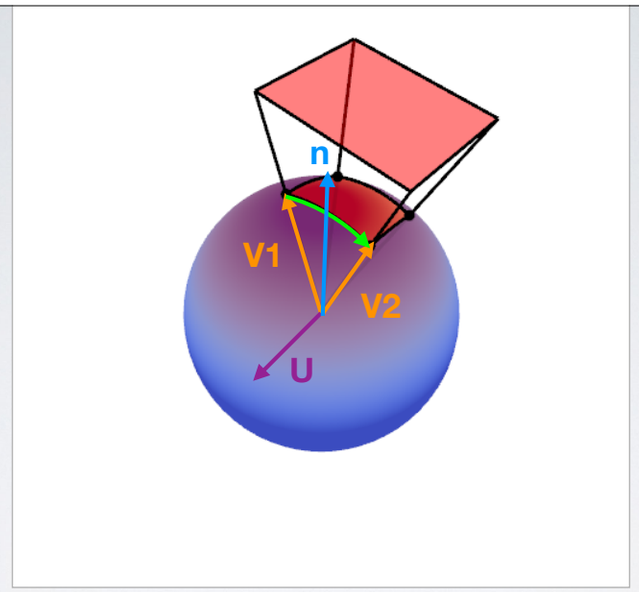
$$\operatorname{acos}(v_1 \cdot v_2) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

It involves the arc length in radians…



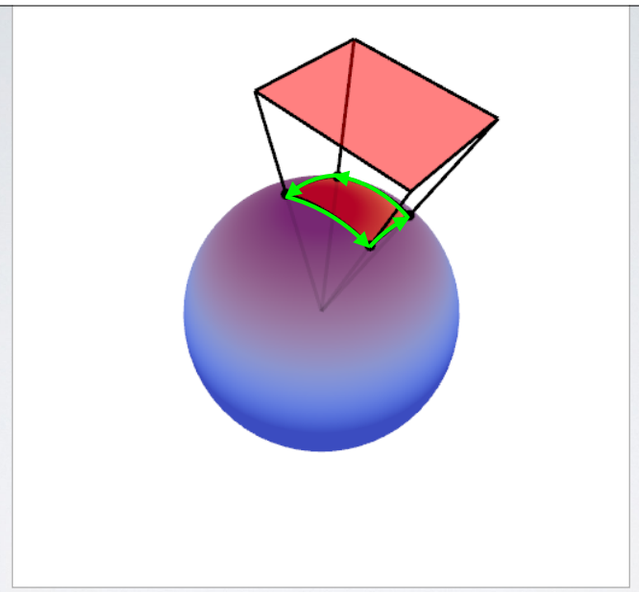
$$\operatorname{acos}(v_1 \cdot v_2) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

There’s also a perpendicular vector u (purple)…



$$\operatorname{acos}(v_1 \cdot v_2) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

…and this is ‘dotted’ with the local surface normal n.



$$\frac{1}{2\,\pi}\sum_{i=1}^{n}\operatorname{acos}\left(v_i\cdot v_j\right)\frac{v_i\times v_j}{\|v_i\times v_j\|}\cdot\mathbf{n}$$

And we simply repeat this process over all edges, summing the results up.

This gives the same result as the area integral.

I don’t have the time in this presentation to explain why this works. One way to think about it is as an application of Stokes’ theorem, which you might have encountered in other areas, such as fluid simulation. But this is a little bit abstract, so for a more intuitive treatment of this particular case, see Eric’s writeup here: https://hal.archives-ouvertes.fr/hal-01458129

```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float theta = acos(dot(v1, v2));
    float3 u = normalize(cross(v1, v2));
    return theta*dot(u, n);
}

float PolyIntegral(float3 v[4], float3 n)
{
    float sum;
    sum  = EdgeIntegral(v[0], v[1]);
    sum += EdgeIntegral(v[1], v[2]);
    sum += EdgeIntegral(v[2], v[3]);
    sum += EdgeIntegral(v[3], v[0]);
    return sum/(2.0*pi);
}
```

Anyway, this results in a really compact and elegant implementation, as you can see here.

It couldn’t be simpler, right?

peace out?

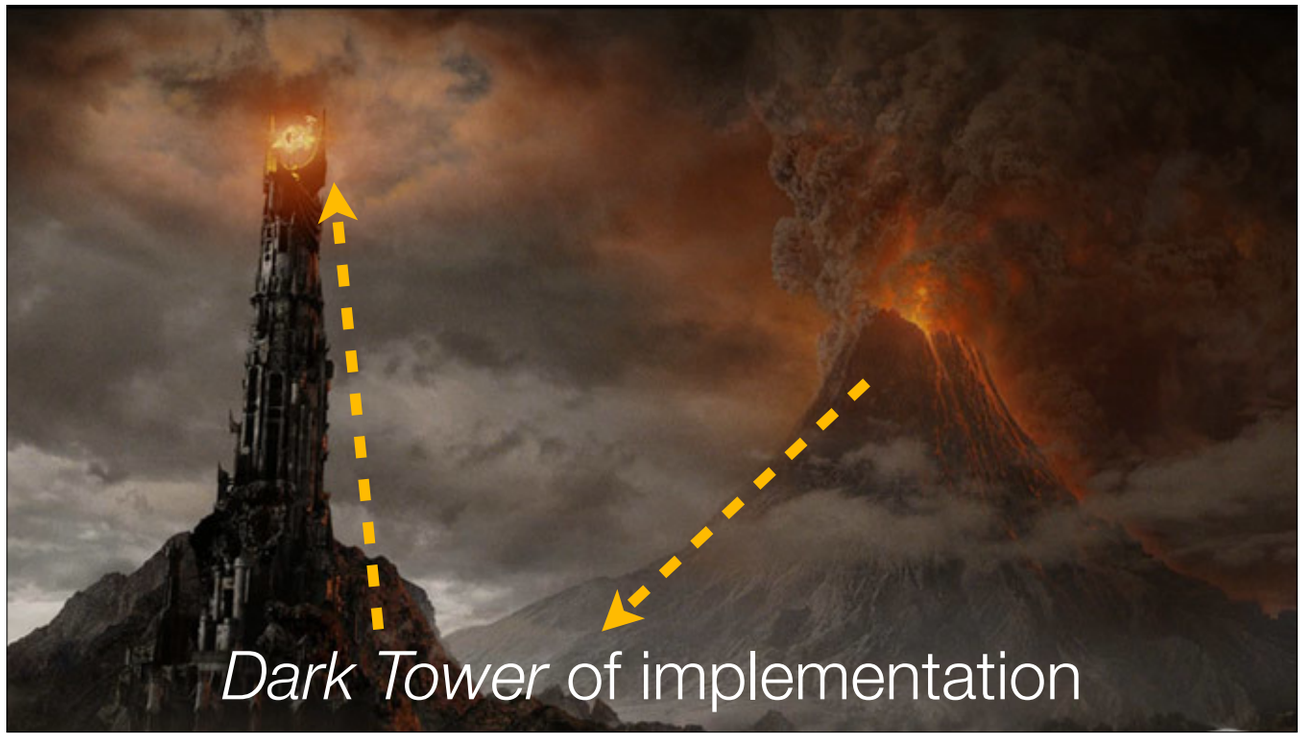
So is our job done? Of course not…

Image credit: White House Correspondent’s Dinner, C-Span.

No, far from it. To use an analogy:

we may have descended research mountain…

Image credit: Lord of the Rings, New Line Cinema.



…but we still need to scale the ‘dark tower’ of implementation.

Theory
&
Implementation

So, now for a few implementation tips and tricks.

There’s a lot more that we’ve gone through than I’ll cover here. We’ll be providing more detailed notes at a later date.

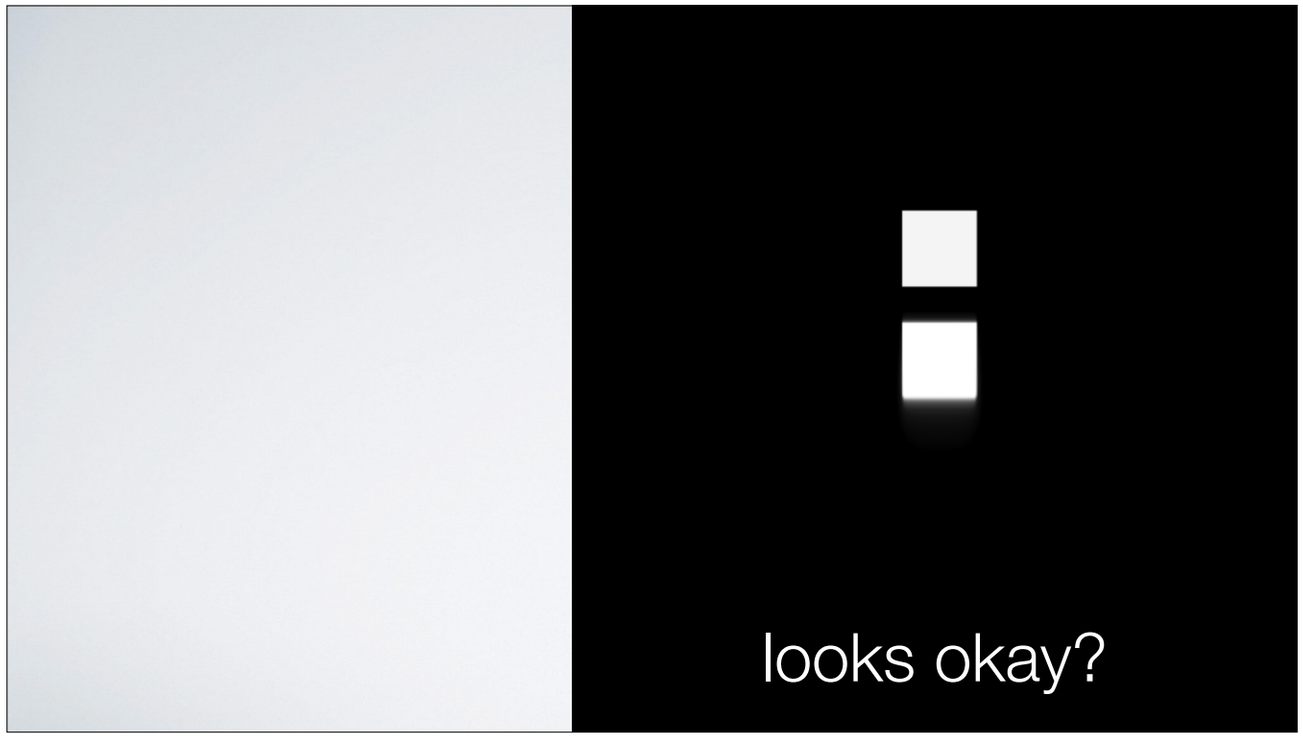
1. Lookup M^{-1}, based on roughness & v. angle
2. Transform polygon by M^{-1}
3. Clip polygon to upper hemisphere
4. Compute edge integrals

Elaborating a little, here are the basic steps (ignoring textures for now).

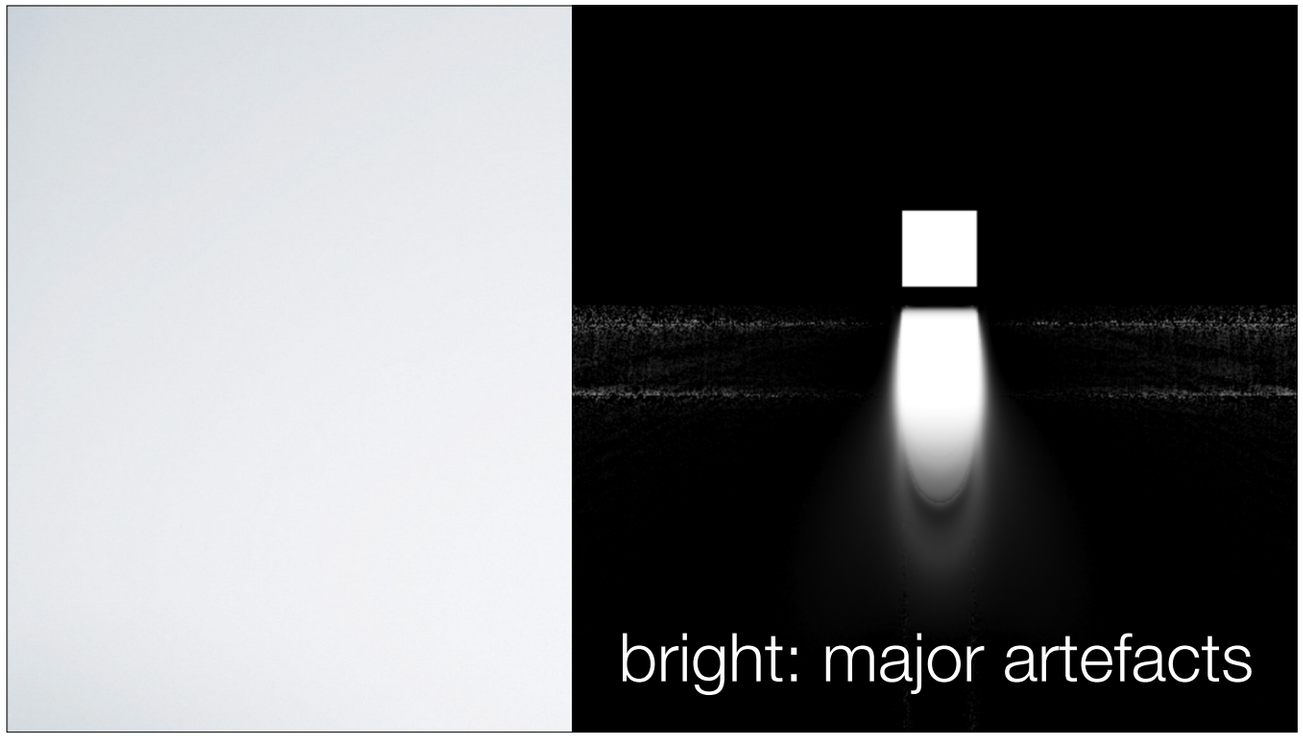
1. Lookup M^{-1}, based on roughness & v. angle
2. Transform polygon by M^{-1}
3. Clip polygon to upper hemisphere
4. **Compute edge integrals**

Going a little out of order, let’s look at those edge integrals first.

What could possible go wrong?

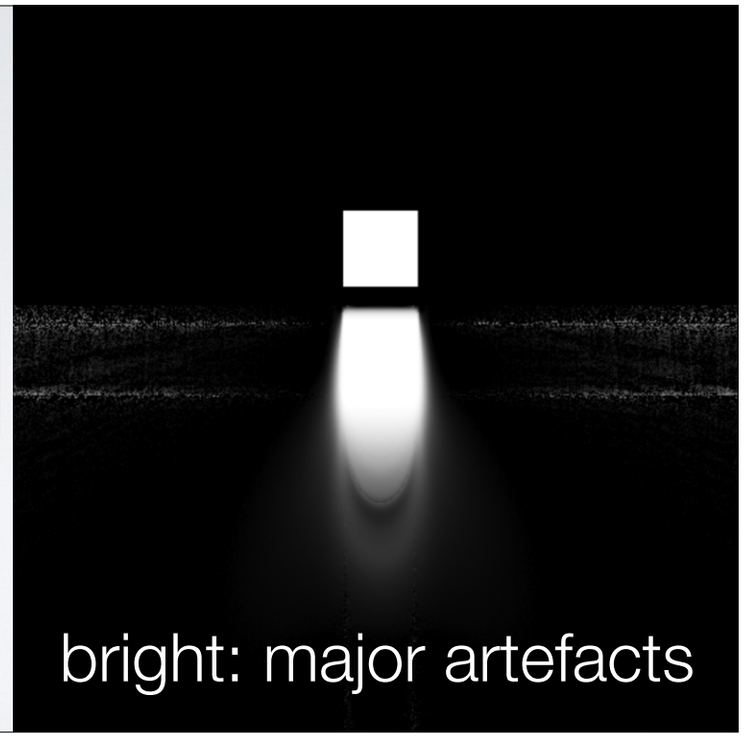


Everything looks okay here, right?



Unfortunately, if you crank up the light intensity, problems start to appear.

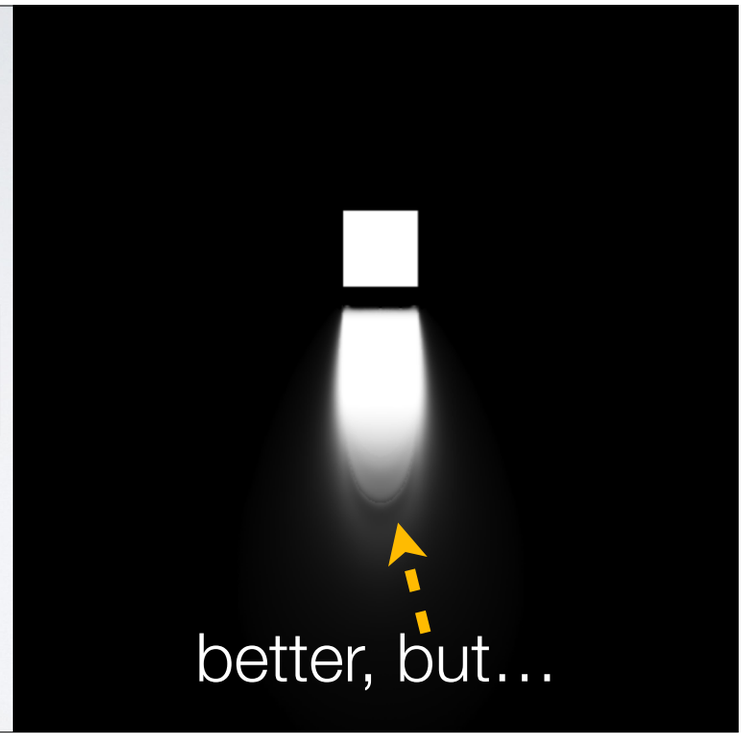
```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float theta = acos(dot(v1, v2));
    float3 u = normalize(cross(v1, v2));
    return theta*dot(u, n);
}
```



Here’s the code again.

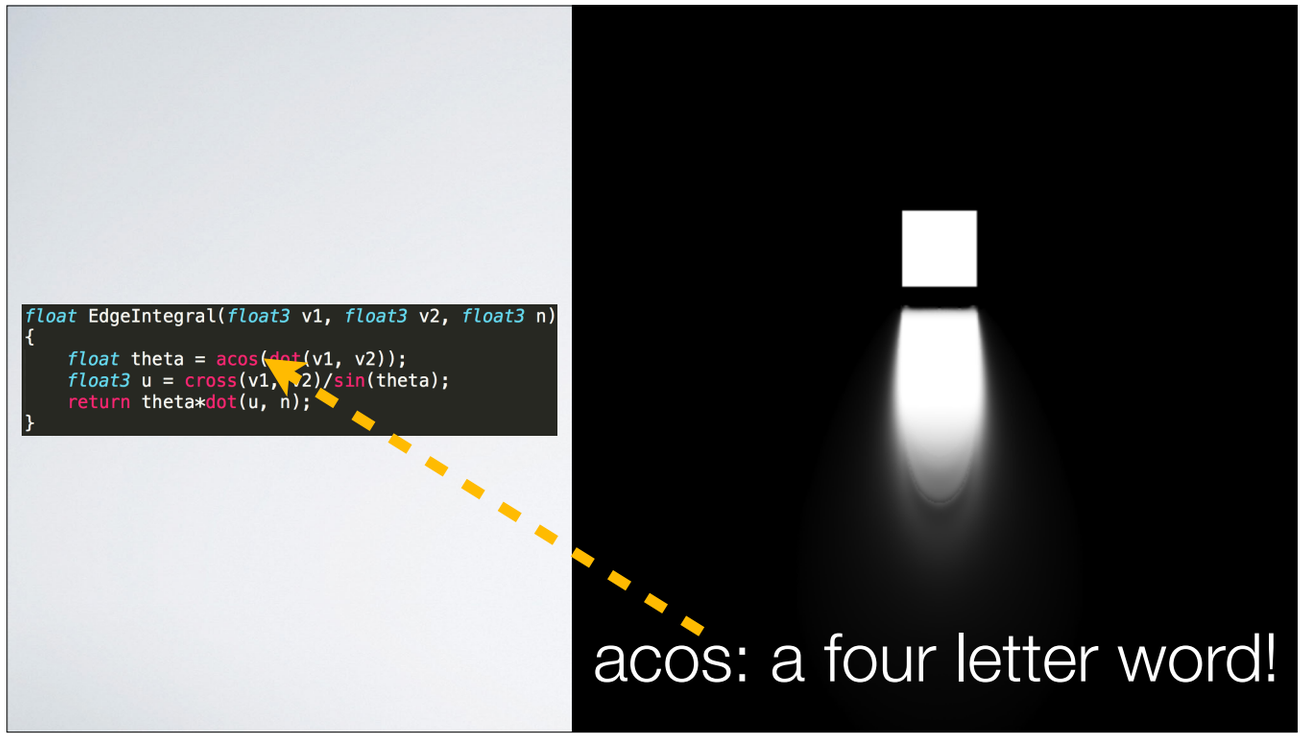
It’s mysterious. Shouldn’t this just work?

```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float theta = acos(dot(v1, v2));
    float3 u = cross(v1, v2)/sin(theta);
    return theta*dot(u, n);
}
```

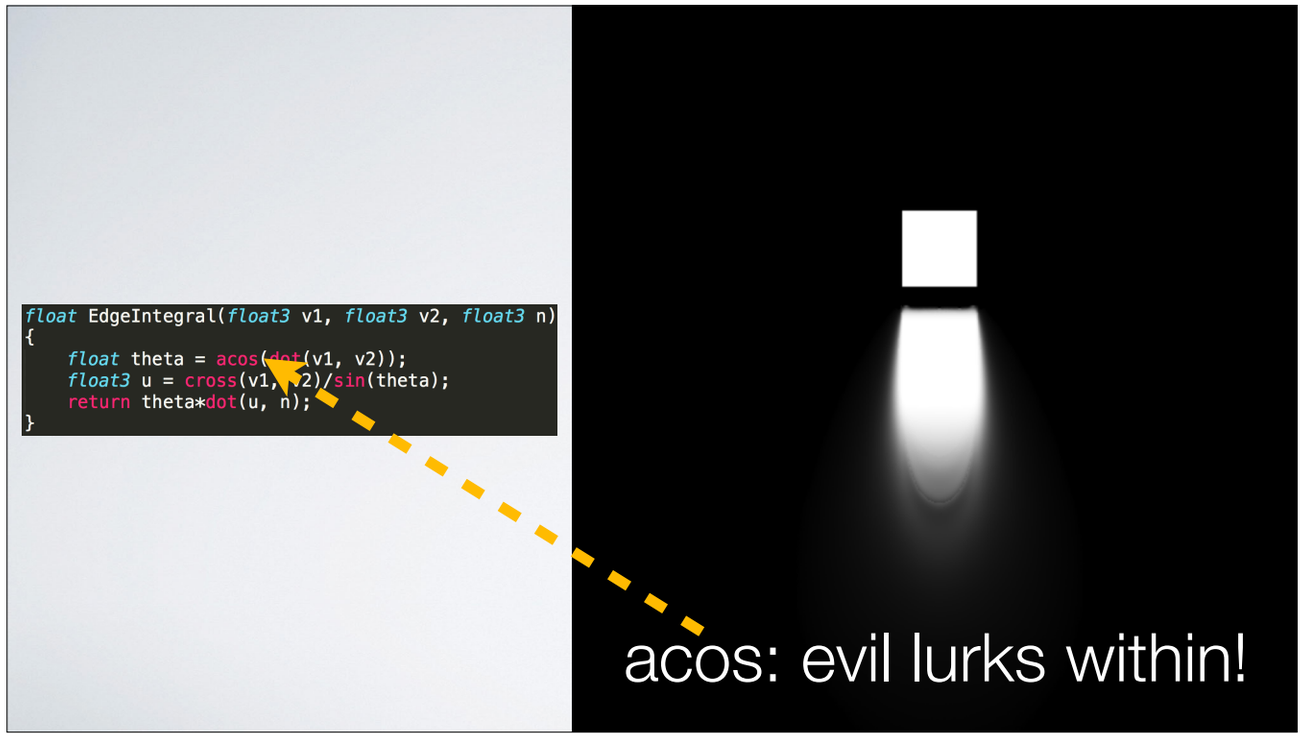


I’m ashamed to say that we worked around the problem by using a mathematically equivalent form that proved to be more numerically stable.

This partially fixed the problem and looked fine in demos, but surely there’s more going on here.



The culprit is acos!



As most of you probably know, this is not an intrinsic.

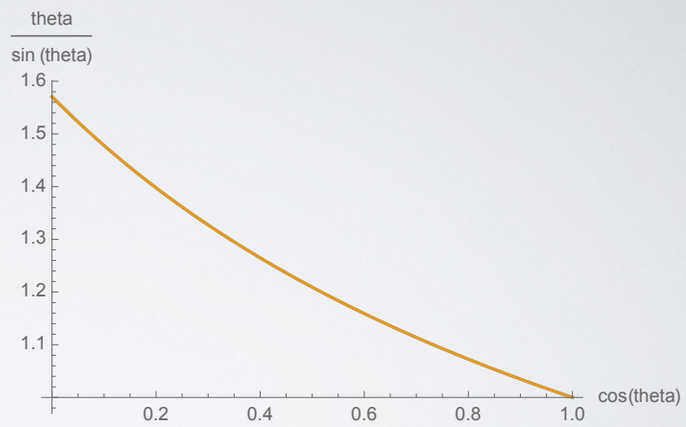
```glsl
float acos(float inX)
{
    float x1 = abs(inX);
    float x2 = x1 * x1;
    float x3 = x2 * x1;
    float s;

    s = -0.2121144 * x1 + 1.5707288;
    s =  0.0742610 * x2 + s;
    s = -0.0187293 * x3 + s;
    s =  sqrt(1.0 - x1) * s;

    return inX >= 0.0 ? s : pi - s;
}
```

Here's the standard implementation. I've seen equivalent code with HLSL, CUDA, OpenGL and on consoles. There's a `sqrt(1 - x)` which provides the basic shape, plus a cubic polynomial.

While this is accurate for most applications, it turns out that the edge integrals require a lot of precision. With the standard `acos` there can be ringing artefacts in some cases (high-intensity lighting and smooth receiver).

(See Handbook of Mathematical Functions, Abramowitz and Stegun.)

```glsl
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float theta = acos(dot(v1, v2));
    float3 u = cross(v1, v2)/sin(theta);
    return theta*dot(u, n);
}
```

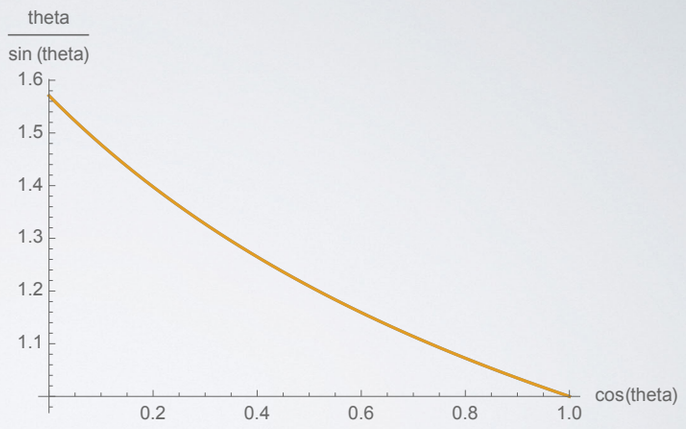


Long story short, after a bunch of trial and error, the solution became clear:
find a fit for `theta` over `sin(theta)`.

(This fit is computed offline, using a fully accurate CPU implementation of `acos` in the target function.)

```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float x = dot(v1, v2);
    float y = abs(x);

    float a = 5.42031 + (3.12829 + 0.0902326*y)*y;
    float b = 3.45068 + (4.18814 + y)*y;
    float theta_sintheta = a / b;

    if (x < 0.0)
        theta_sintheta = pi*rsqrt(1.0 - x*x) - theta_sintheta;

    float3 u = cross(v1, v2);

    return theta_sintheta*dot(u, n);
}
```

rational fit



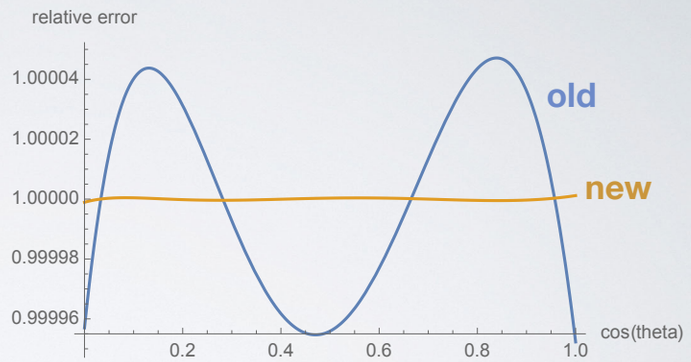
In order to obtain enough accuracy, this required a cubic rational fit.

This is not really much more expensive: we now have an additional division from the rational, but we've saved a `sin()` call.

Note: we only need to fit 0 to pi/2 as the remaining angular range can be calculated from this.

```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float x = dot(v1, v2);
    float y = abs(x);

    float a = 5.42031 + (3.12829 + 0.0902326*y)*y;
    float b = 3.45068 + (4.18814 + y)*y;
    float theta_sintheta = a / b;

    if (x < 0.0)
        theta_sintheta = pi*rsqrt(1.0 - x*x) - theta_sintheta;

    float3 u = cross(v1, v2);

    return theta_sintheta*dot(u, n);
}
```

rational fit



By doing this compound fit, the relative error* is much lower.

old: using the standard shading language implementation of `acos` to calculate theta
new: our fit of `theta/sin(theta)`.

(* This isn’t really the relative error, but the ratio of each approximation to the original function, which makes under- and over-shoot clear.)

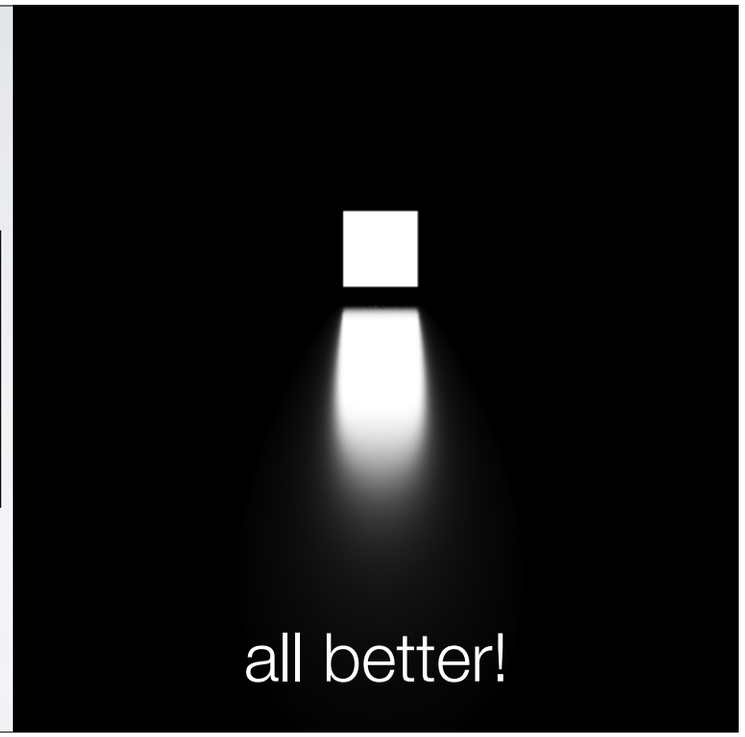
```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float x = dot(v1, v2);
    float y = abs(x);

    float a = 5.42031 + (3.12829 + 0.0902326*y)*y;
    float b = 3.45068 + (4.18814 + y)*y;
    float theta_sintheta = a / b;

    if (x < 0.0)
        theta_sintheta = pi*rsqrt(1.0 - x*x) - theta_sintheta;

    float3 u = cross(v1, v2);

    return theta_sintheta*dot(u, n);
}
```

rational fit

So, returning to our issue…

```
float EdgeIntegral(float3 v1, float3 v2, float3 n)
{
    float x = dot(v1, v2);
    float y = abs(x);

    float a = 5.42031 + (3.12829 + 0.0902326*y)*y;
    float b = 3.45068 + (4.18814 + y)*y;
    float theta_sintheta = a / b;

    if (x < 0.0)
        theta_sintheta = pi*rsqrt(1.0 - x*x) - theta_sintheta;

    float3 u = cross(v1, v2);

    return theta_sintheta*dot(u, n);
}
```

rational fit



Here’s the result, which looks a lot better.

```
float3 EdgeIntegralDiffuse(vec3 v1, vec3 v2, vec3 n)
{
    float x = dot(v1, v2);
    float y = abs(x);

    float theta_sintheta = 1.5708 + (-0.879406 + 0.308609*y)*y;

    if (x < 0.0)
        theta_sintheta = pi*rsqrt(1.0 - x*x) - theta_sintheta;

    float3 u = cross(v1, v2);

    return theta_sintheta*dot(u, n);
}
```

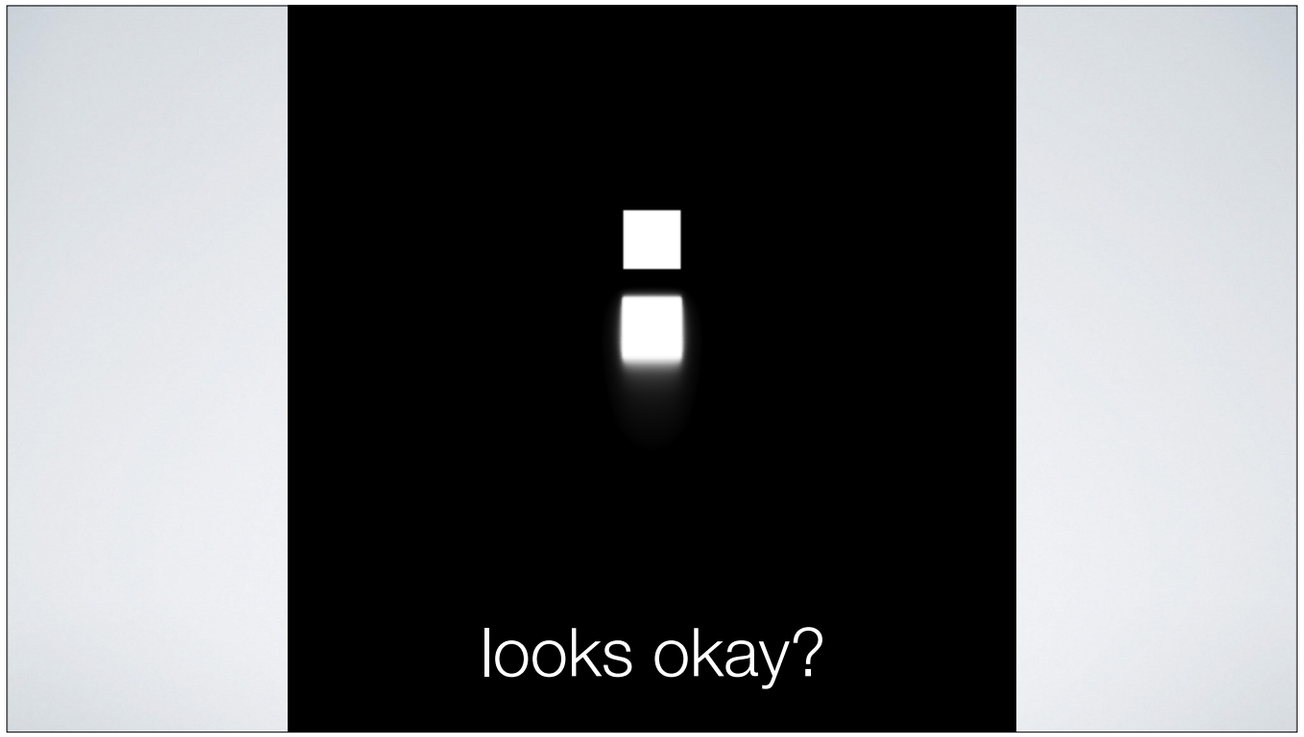
bonus: cheaper diffuse

As a bonus, we can use a cheaper version for diffuse, since it doesn’t need as much accuracy.

In this case we can get away with a quadratic, thereby saving some MADs and a division.

1. Lookup M^{-1}, based on roughness & v. angle
2. Transform polygon by M^{-1}
3. Clip polygon to upper hemisphere
4. Compute edge integrals

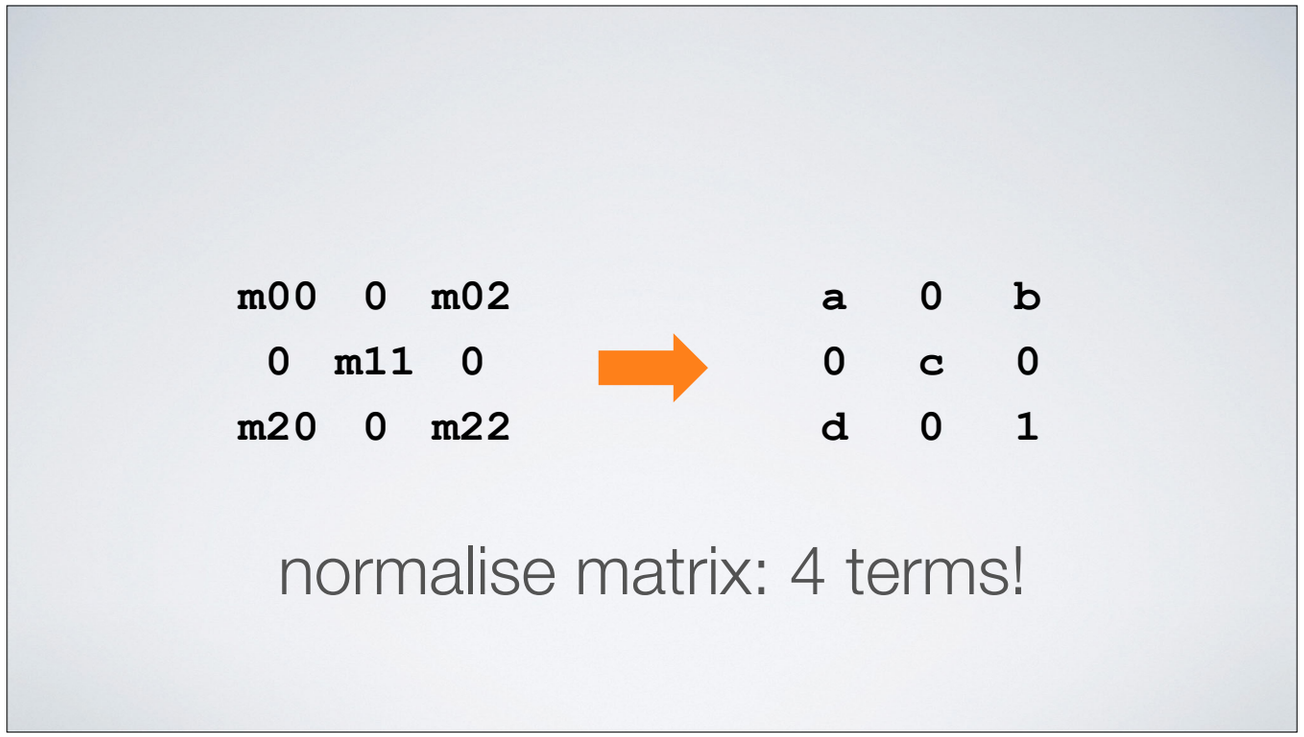
Right, now to the next gotcha.



Again, everything looks fine.

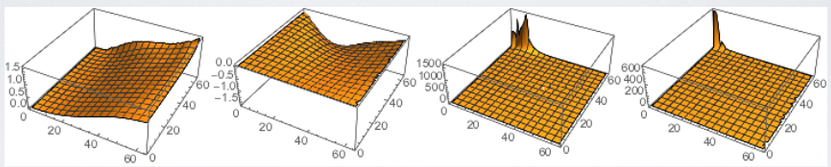
But at higher intensity the highlight shape isn’t correct.

I don’t have a video here, but when changing the view angle there’s some noticeable banding and interpolation issues, especially at grazing angles.

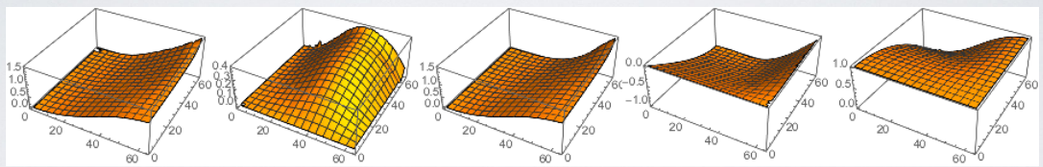


This actually came down to a trick we did, where we divided all of the components of each matrix by the bottom-right value, so it would always be 1 and therefore wouldn’t need to be stored. We did this to be able to fit the matrices into a four component texture. However, we still needed a fifth component for the magnitude of the BRDF, so a second texture fetch in the shader was unavoidable.

In the end, it was a bit of a false economy and by doing this division we introduced significant side effect…

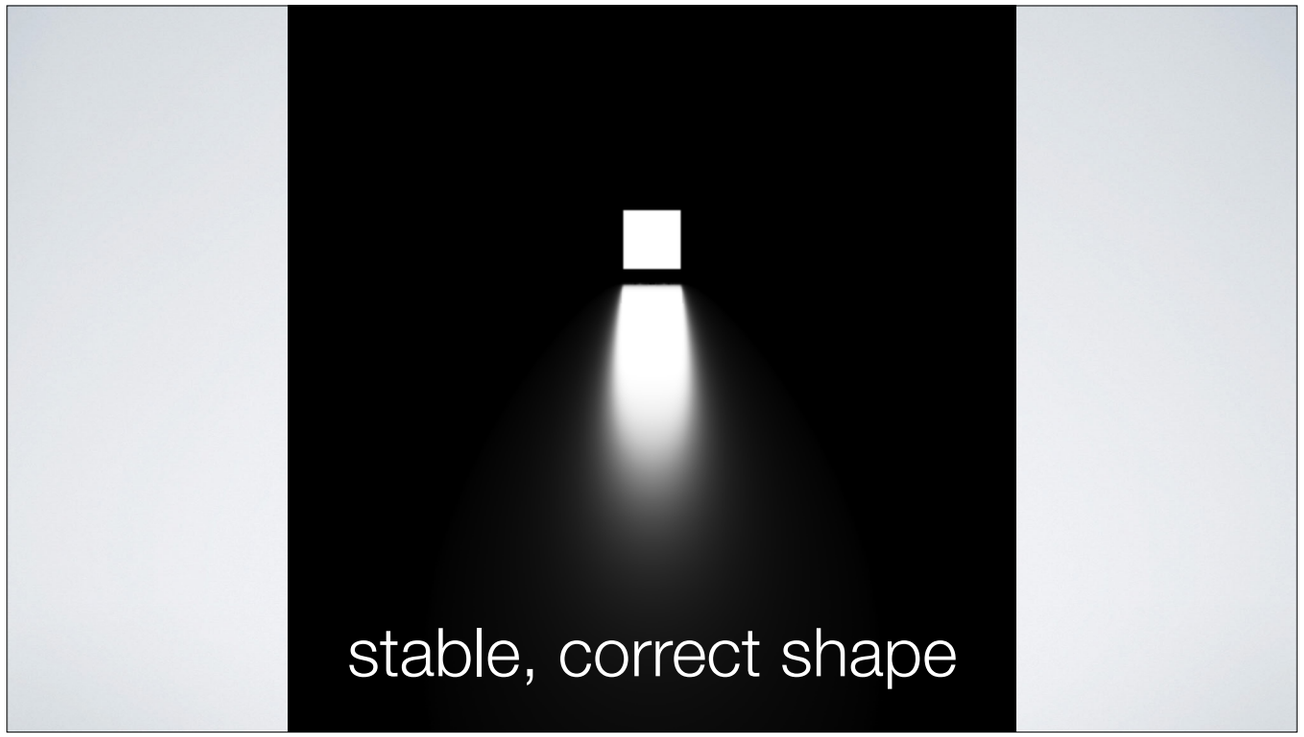


…which is that some of the matrix components ended up varying more wildly over the (roughness, view-angle) domain, so they don’t interpolate well.



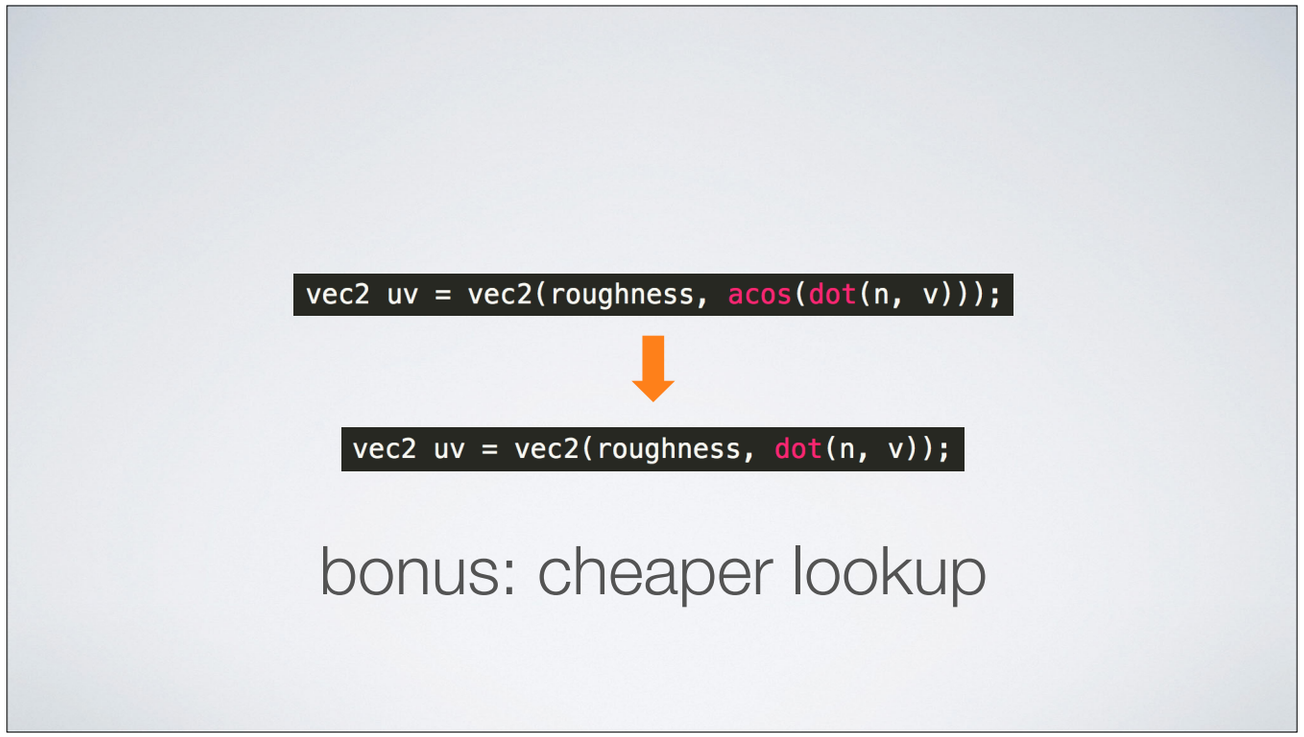
If we don’t do this rescaling, the original five components vary a lot more smoothly and have a lower dynamic range. (We might even be able to get away with < 16bits per component, though we haven’t tried this yet.)

Here’s the result we had before, with the matrix rescaling…



…and here’s after, without.

This is a good reminder that it’s helpful to visualise your data in different ways. While we had been carefully comparing the fitted LTC distributions versus the original BRDF during development, we didn’t take a close look at the tabulated values until later.



At the time, we used another workaround to reduce the artefacts: we parameterised the lookup tables by `theta` rather than `cos(theta)`. Now, with the correct fix, we no longer need to do this, which means we can avoid the expense of `acos`.

1. Lookup M^{-1}, based on roughness & v. angle
2. Transform polygon by M^{-1}
3. **Clip polygon to upper hemisphere**
4. Compute edge integrals

Okay, on to the third issue: clipping.

Something I'd glossed over up until now is that in order to get the correct result (form factor) of the polygon, we need to clip the polygon to the upper hemisphere.

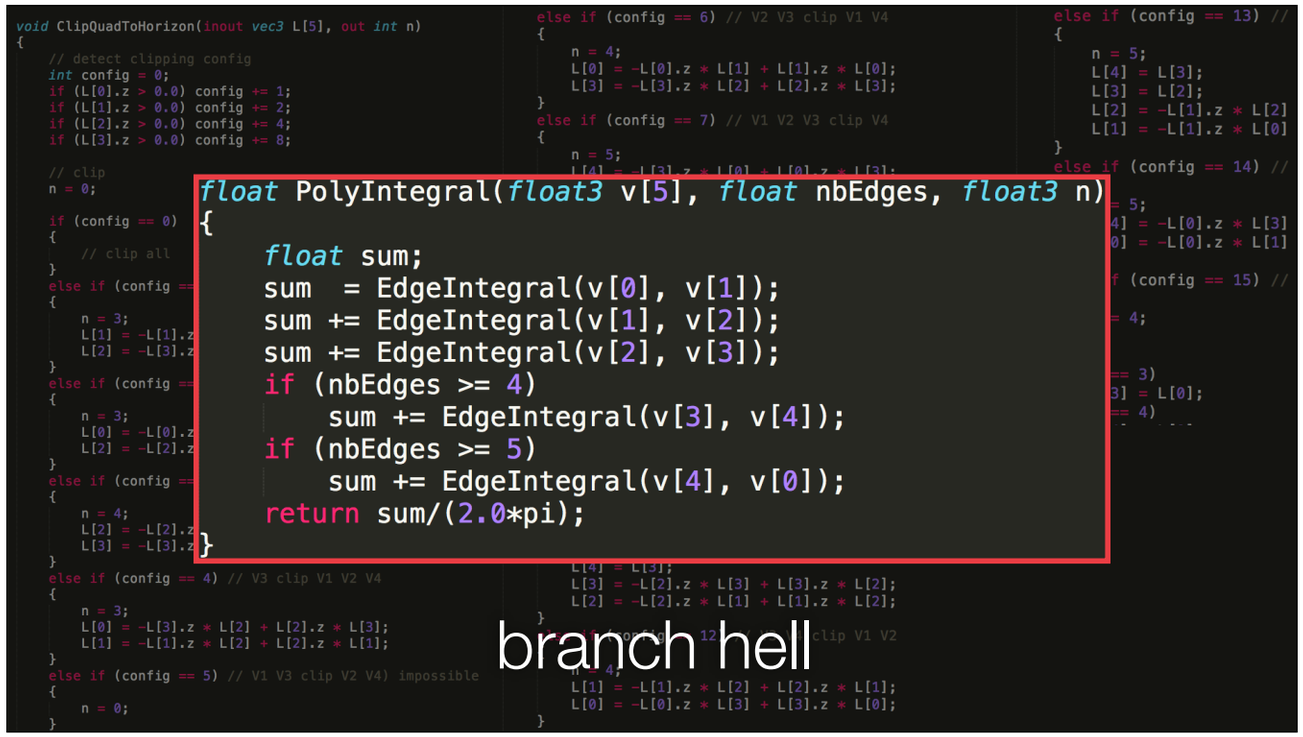
Polygon clipping isn't fun…

```
void ClipQuadToHorizon(inout vec3 L[5], out int n)
{
    // detect clipping config
    int config = 0;
    if (L[0].z > 0.0) config += 1;
    if (L[1].z > 0.0) config += 2;
    if (L[2].z > 0.0) config += 4;
    if (L[3].z > 0.0) config += 8;

    // clip
    n = 0;

    if (config == 0)
    {
        // clip all
    }
    else if (config == 1) // V1 clip V2 V3 V4
    {
        n = 3;
        L[1] = -L[1].z * L[0] + L[0].z * L[1];
        L[2] = -L[3].z * L[0] + L[0].z * L[3];
    }
    else if (config == 2) // V2 clip V1 V3 V4
    {
        n = 3;
        L[0] = -L[0].z * L[1] + L[1].z * L[0];
        L[2] = -L[2].z * L[1] + L[1].z * L[2];
    }
    else if (config == 3) // V1 V2 clip V3 V4
    {
        n = 4;
        L[2] = -L[2].z * L[1] + L[1].z * L[2];
        L[3] = -L[3].z * L[0] + L[0].z * L[3];
    }
    else if (config == 4) // V3 clip V1 V2 V4
    {
        n = 3;
        L[0] = -L[3].z * L[2] + L[2].z * L[3];
        L[1] = -L[1].z * L[2] + L[2].z * L[1];
    }
    else if (config == 5) // V1 V3 clip V2 V4) impossible
    {
        n = 0;
    }
    else if (config == 6) // V2 V3 clip V1 V4
    {
        n = 4;
        L[0] = -L[0].z * L[1] + L[1].z * L[0];
        L[3] = -L[3].z * L[2] + L[2].z * L[3];
    }
    else if (config == 7) // V1 V2 V3 clip V4
    {
        n = 5;
        L[4] = -L[3].z * L[0] + L[0].z * L[3];
        L[3] = -L[3].z * L[2] + L[2].z * L[3];
    }
    else if (config == 8) // V4 clip V1 V2 V3
    {
        n = 3;
        L[0] = -L[0].z * L[3] + L[3].z * L[0];
        L[1] = -L[2].z * L[3] + L[3].z * L[2];
        L[2] =  L[3];
    }
    else if (config == 9) // V1 V4 clip V2 V3
    {
        n = 4;
        L[1] = -L[1].z * L[0] + L[0].z * L[1];
        L[2] = -L[2].z * L[3] + L[3].z * L[2];
    }
    else if (config == 10) // V2 V4 clip V1 V3) impossible
    {
        n = 0;
    }
    else if (config == 11) // V1 V2 V4 clip V3
    {
        n = 5;
        L[4] = L[3];
        L[3] = -L[2].z * L[3] + L[3].z * L[2];
        L[2] = -L[2].z * L[1] + L[1].z * L[2];
    }
    else if (config == 12) // V3 V4 clip V1 V2
    {
        n = 4;
        L[1] = -L[1].z * L[2] + L[2].z * L[1];
        L[0] = -L[0].z * L[3] + L[3].z * L[0];
    }
    else if (config == 13) //
    {
        n = 5;
        L[4] = L[3];
        L[3] = L[2];
        L[2] = -L[1].z * L[2]
        L[1] = -L[1].z * L[0]
    }
    else if (config == 14) //
    {
        n = 5;
        L[4] = -L[0].z * L[3]
        L[0] = -L[0].z * L[1]
    }
    else if (config == 15) //
    {
        n = 4;
    }

    if (n == 3)
        L[3] = L[0];
    if (n == 4)
        L[4] = L[0];
}
```

We tried various flavours of this:

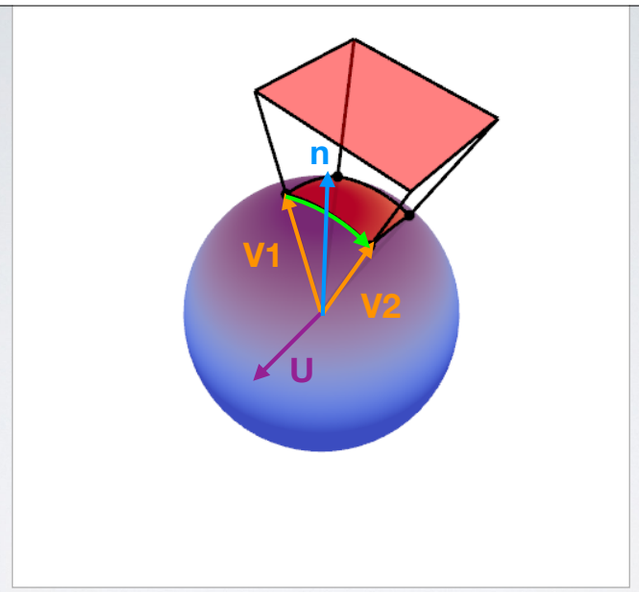
* ‘On the fly’ clipping during the edge integration
* Morgan McGuire’s quad clipping: https://casual-effects.com/research/McGuire2011Clipping/index.html. This minimises the number of branches, but involves quite a lot of data shuffling
* A big switch / if-else based on which points are above or below the horizon

The last one turned out to be the fastest (albeit close Morgan’s) on PS4, but all of them generated a large number of instructions or branches.



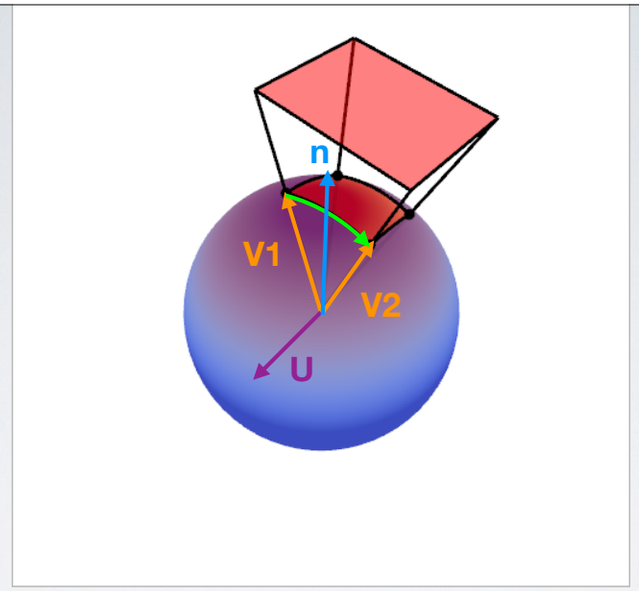
On top of that, the clipping process results in a variable number of edges: 3 to 5. So, yet more branches!

While there are probably some gains to be had by carefully examining and tuning the generated assembly for game consoles, it would be nice to avoid this complexity entirely.



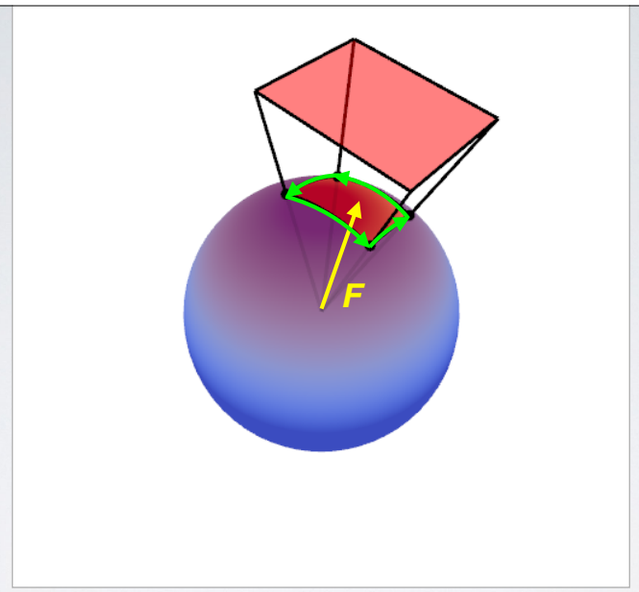
$$\operatorname{acos}(v_1 \cdot v_2) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

Let’s return to our edge integral.



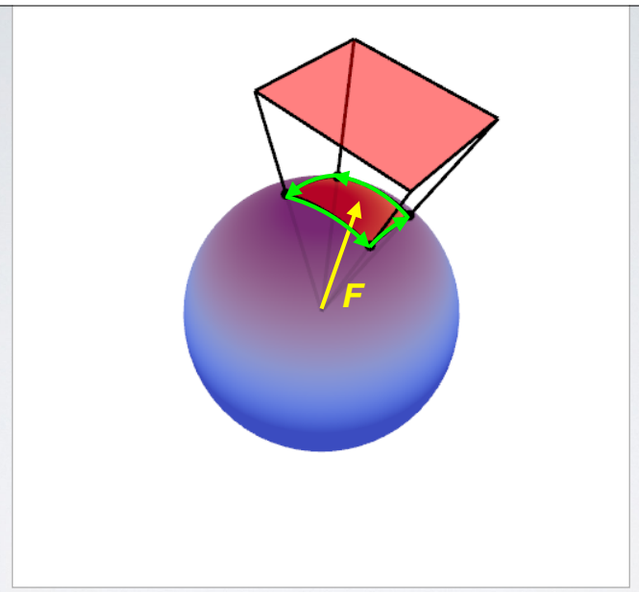
$$\operatorname{acos}\left(v_1 \cdot v_2\right) \frac{v_1 \times v_2}{\|v_1 \times v_2\|} \cdot \mathbf{n}$$

If we don’t project onto the plane (dot with the normal), we end up with a vector form.



vector form factor

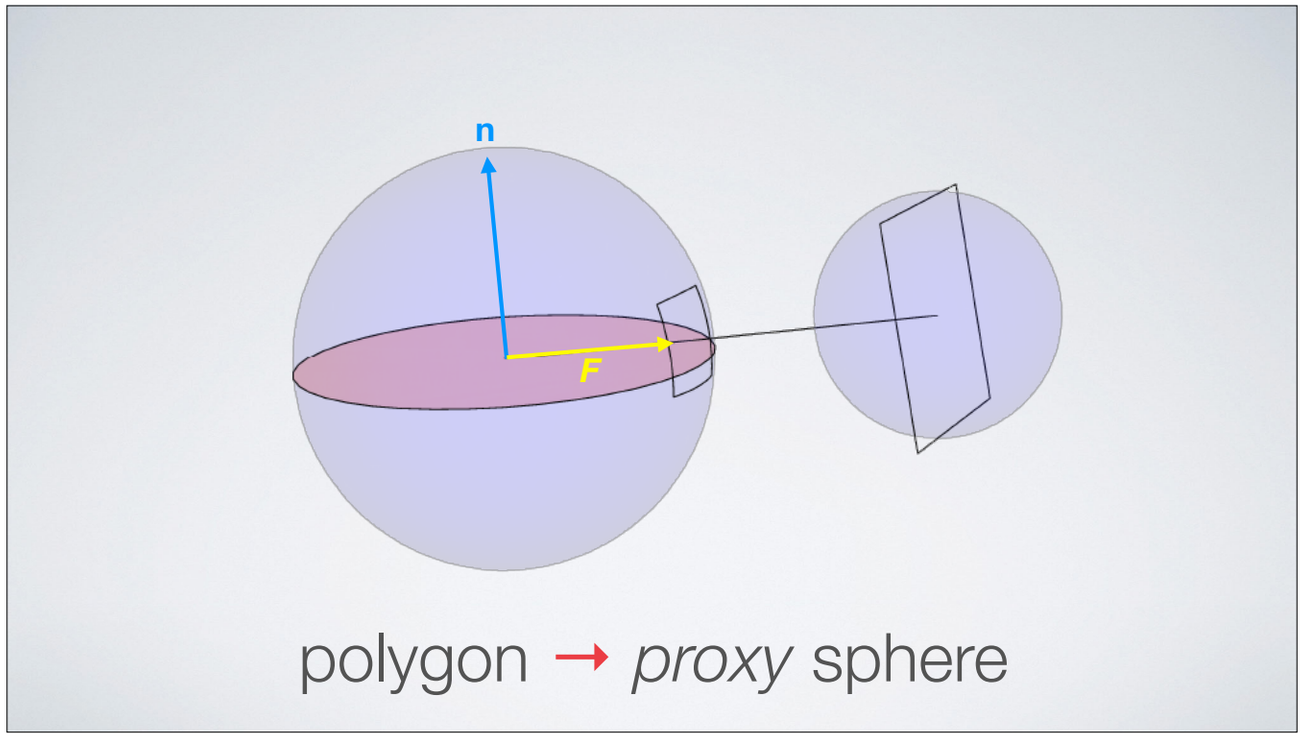
You can think of this as a *vector form factor*, or *vector irradiance*. Let’s call this vector F.



length(F) = form factor in direction of F

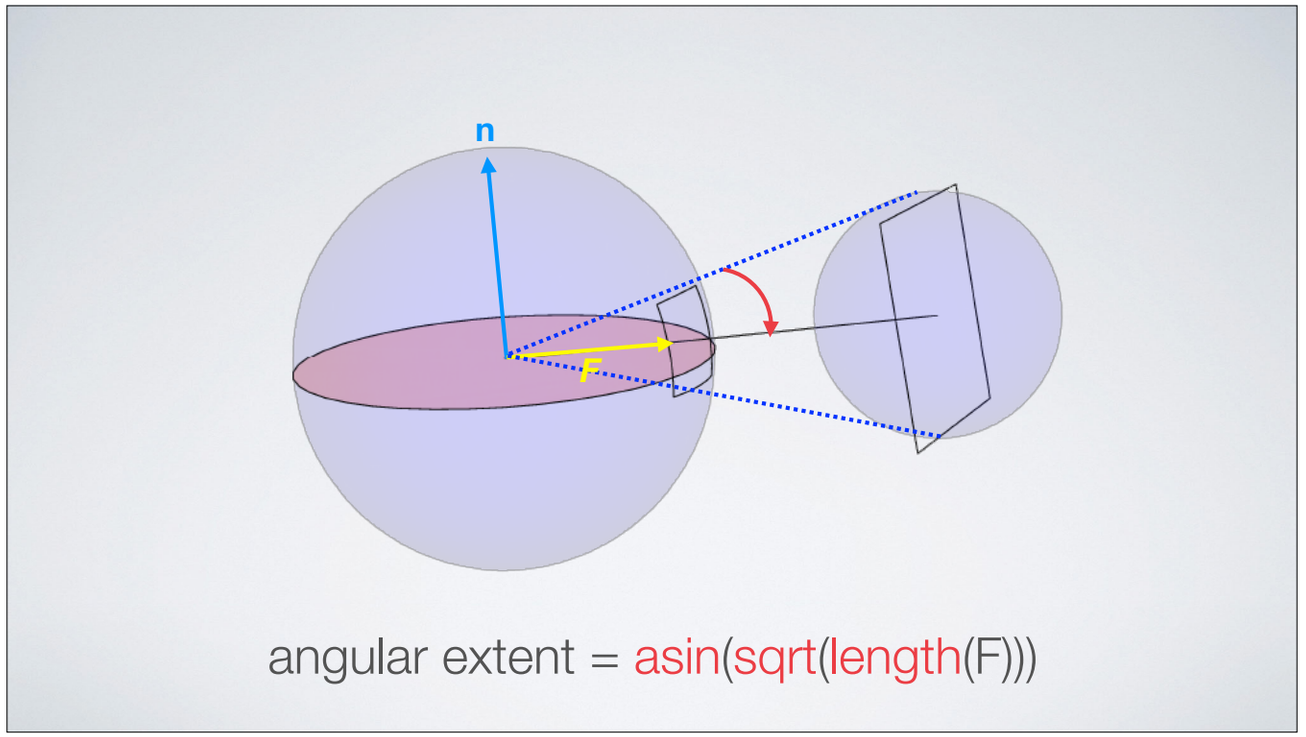
This has a very nice property: the length (norm) of F is the form factor of the polygon in the direction of F.

We can use this to approximate the form factor of the polygon as if it had been clipped to the horizon.



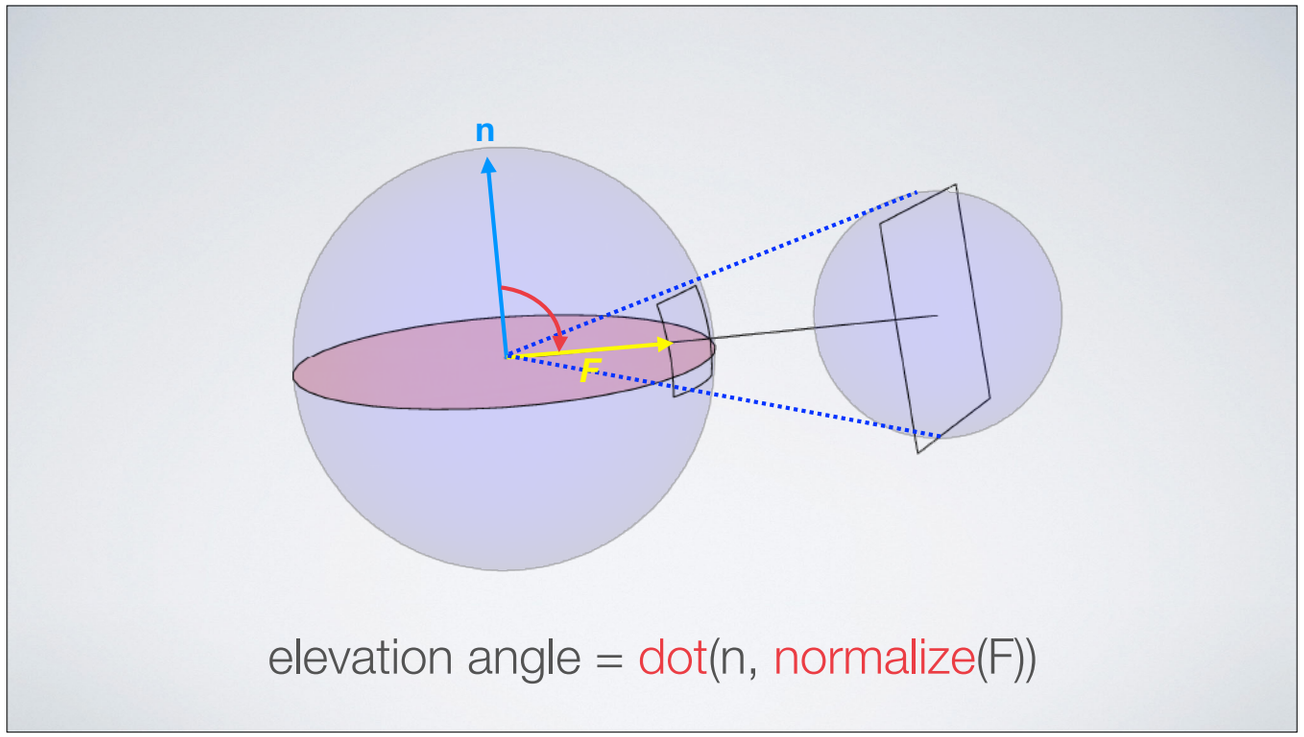
Instead of a polygon, we can use a sphere that has the same form factor.

Note: we'd like to thank Brian Karis for the inspiration for this approach. (In fact, he'd actually done this already and kindly shared his final approximation during a general email exchange on area lighting, but it wasn't clear at the time that it was based on the same idea. Essentially we re-derived it.)



We can compute the angular extent of the sphere from F…

(This comes from the fact that a sphere's form factor is $\sin(\text{angular\_extent})^2$.)



as well as its direction (or elevation angle).

$$I_{\text{hemi-sub}}(\omega,\sigma) \equiv \frac{1}{\pi}\begin{cases} \pi\cos\omega\sin^2\sigma, & \omega \in [0, \frac{\pi}{2}-\sigma] \\ \pi\cos\omega\sin^2\sigma + G(\omega,\sigma,\gamma) - H(\omega,\sigma,\gamma), & \omega \in [\frac{\pi}{2}-\sigma, \frac{\pi}{2}] \\ G(\omega,\sigma,\gamma) + H(\omega,\sigma,\gamma), & \omega \in [\frac{\pi}{2}, \frac{\pi}{2}+\sigma] \\ 0, & \omega \in [\frac{\pi}{2}+\sigma, \pi] \end{cases}$$

$$\gamma \equiv \sin^{-1}\left(\frac{\cos\sigma}{\sin\omega}\right),$$

$$G(\omega,\sigma,\gamma) \equiv -2\sin\omega\cos\sigma\cos\gamma + \frac{\pi}{2} - \gamma + \sin\gamma\cos\gamma,$$

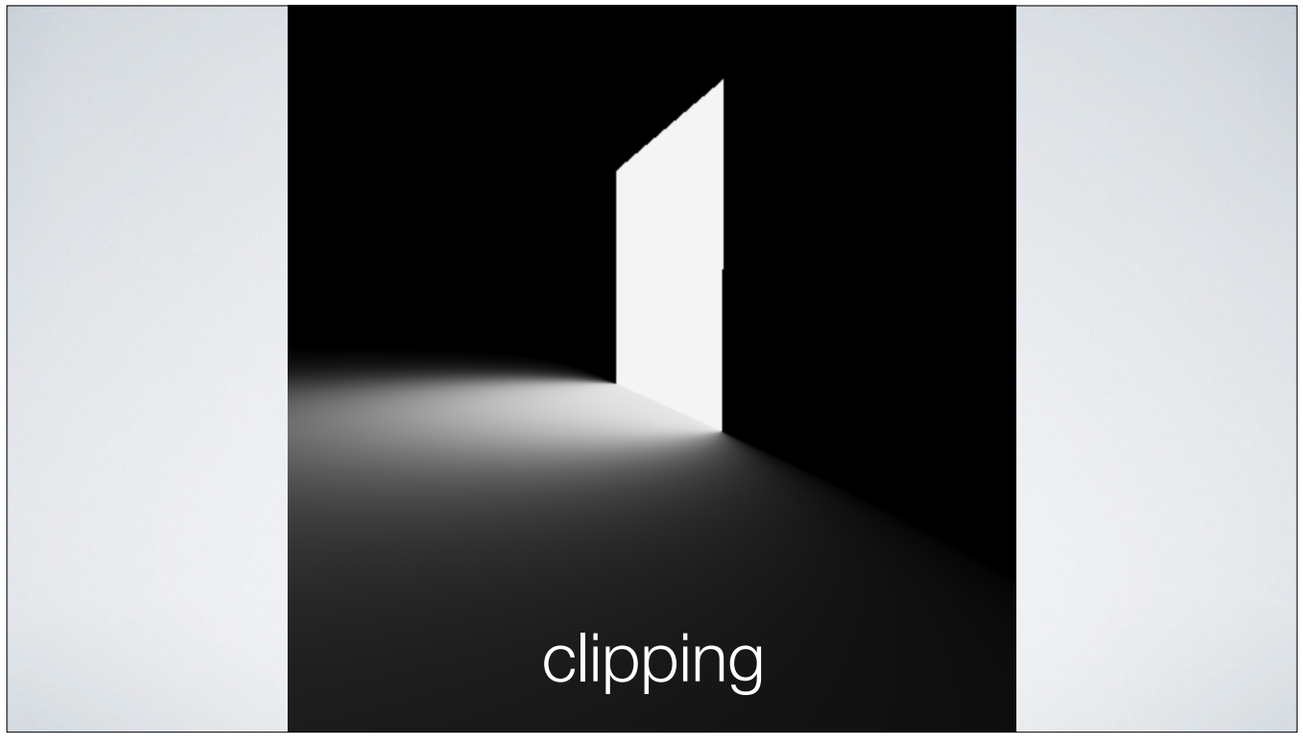
$$H(\omega,\sigma,\gamma) \equiv \cos\omega\left[\cos\gamma\sqrt{\sin^2\sigma - \cos^2\gamma} + \sin^2\sigma\sin^{-1}\left(\frac{\cos\gamma}{\sin\sigma}\right)\right].$$

sphere with horizon clipping [Snyder96]

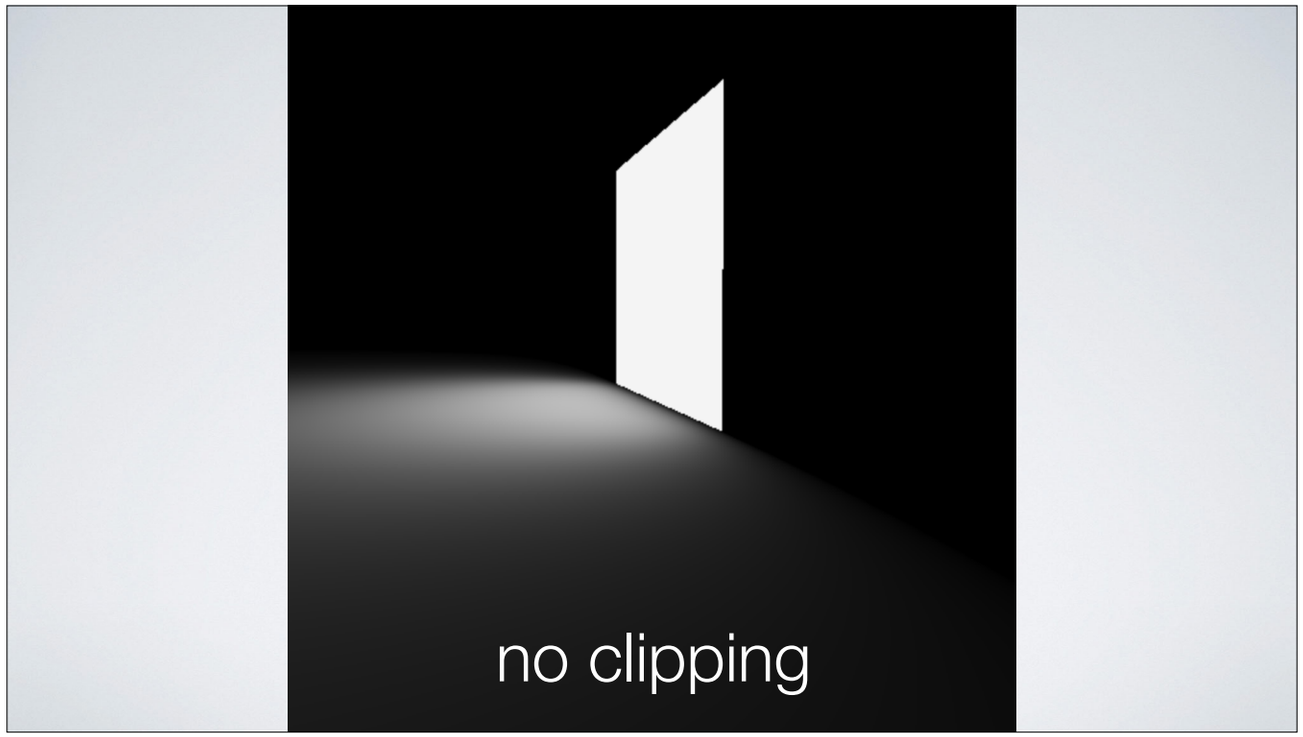
Why is this helpful? Well, there are a couple of (equivalent) analytical solutions for horizon-clipped sphere form factors. Here’s one.

Using this, we can precompute a 2D LUT that contains the clipped form factors for spheres of different angular extents (0 - pi/2) and elevation angles.

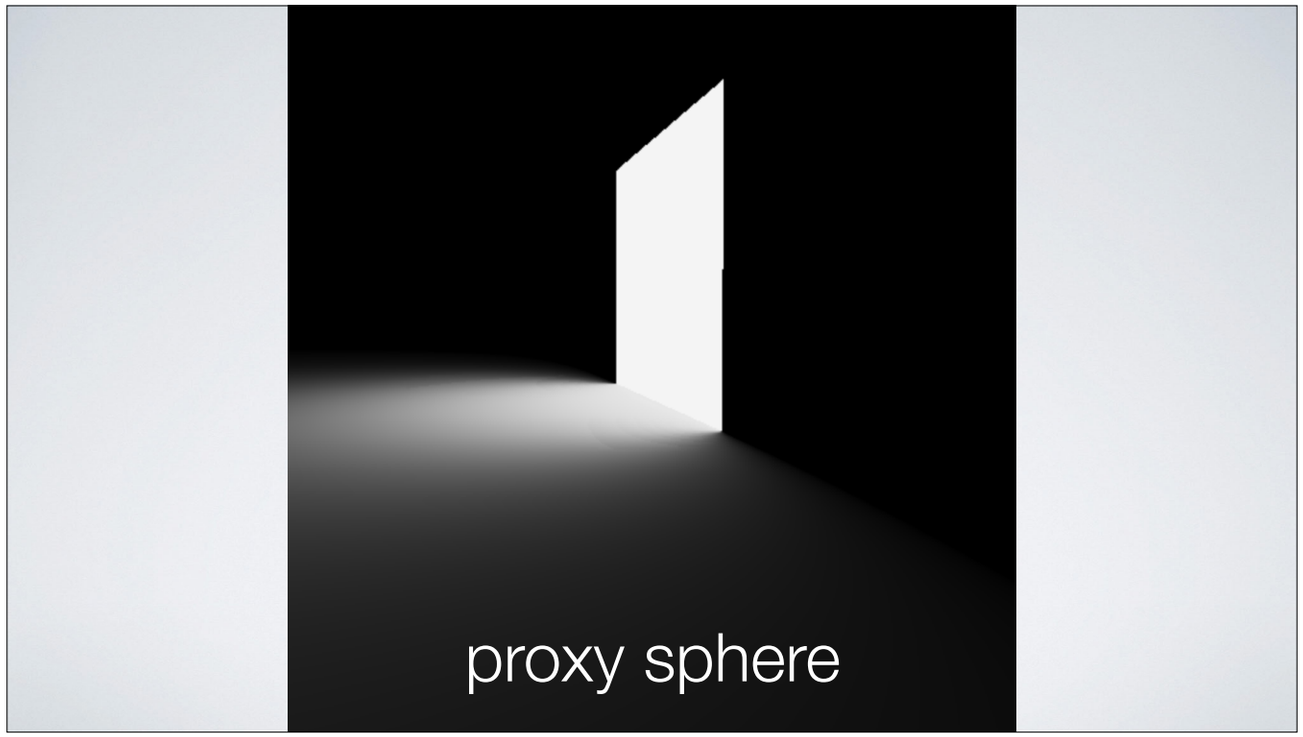
At runtime we can lookup into this texture with the extent and angle we’ve calculated from F, giving us an approximation of the clipped form factor of the polygon.



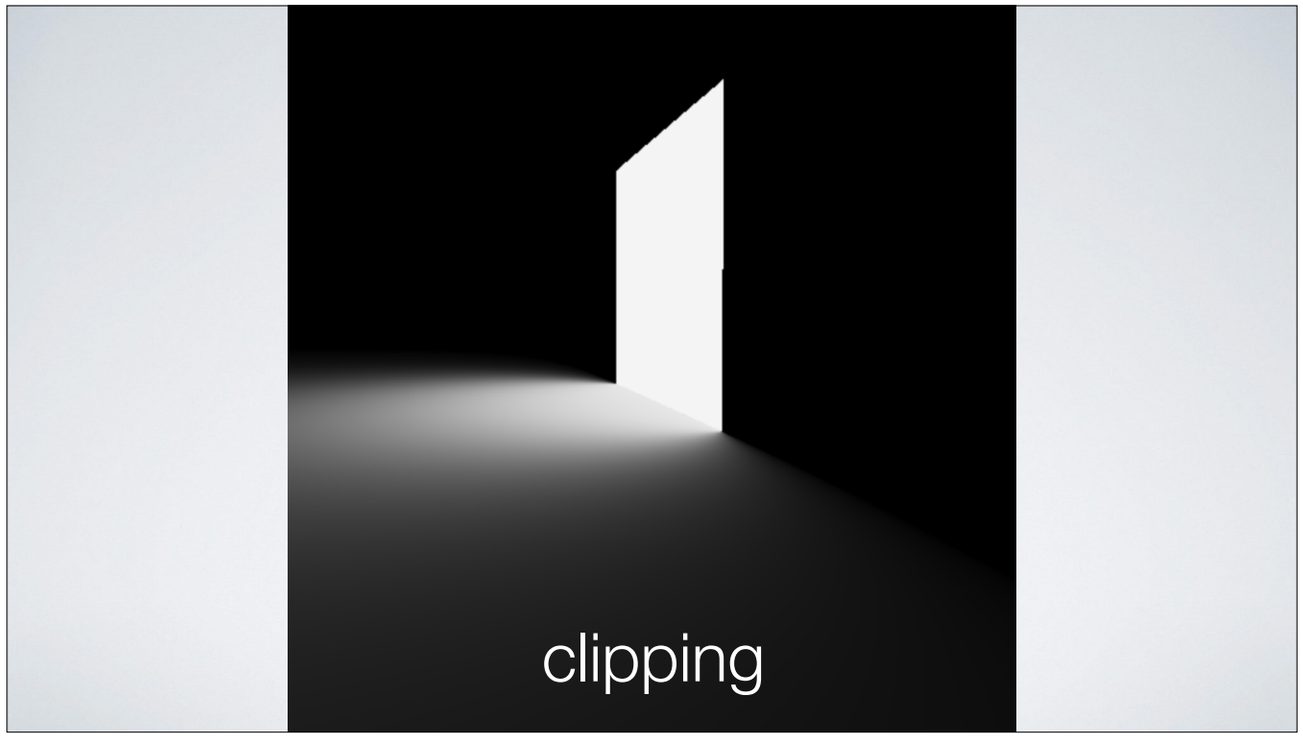
Here’s the result with the original, expensive clipping.



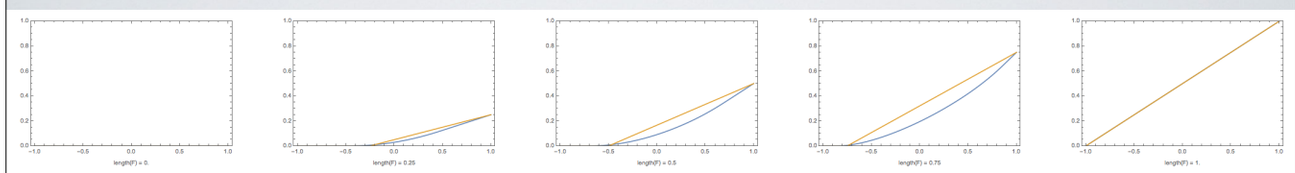
If we don’t do any clipping, we get darkening near to the light source (among other issues).



Here’s the result with the proxy sphere. It’s pretty close to the correct result.



Here’s the original again, for comparison.

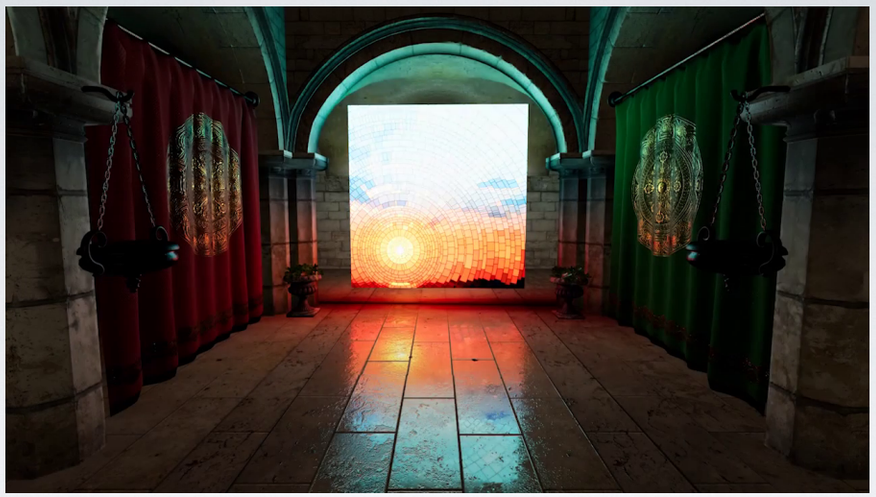


```
float SphereIntegral(float3 F)
{
    float l = length(F);
    return max((l*l + F.z)/(l + 1), 0);
}
```

very cheap approximation

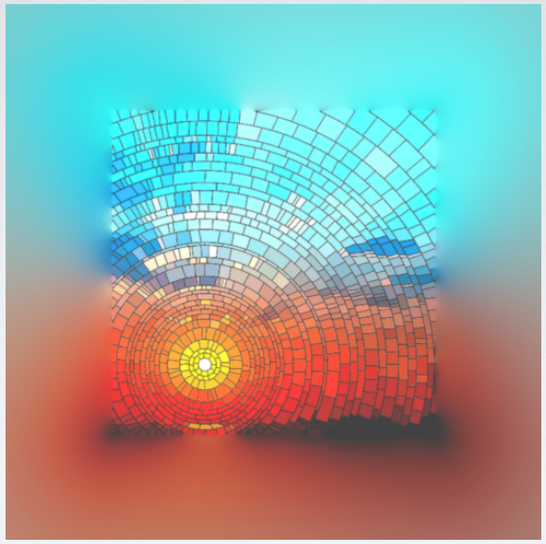
For precision reasons, it’s better to divide through by the original form factor and store a multiplier in the LUT instead.

It’s also possible to do away with the LUT entirely. John Snyder gives a couple of options involving cubic Hermite curves, but these are on the expensive side. The above function, which calculates an approximation of the clipped sphere form factor from F, is a somewhat crude but effective alternative. The name should really be `PolygonVectorFormFactorToHorizonClippedSphereFormFactor`, but there wasn’t enough space. :)



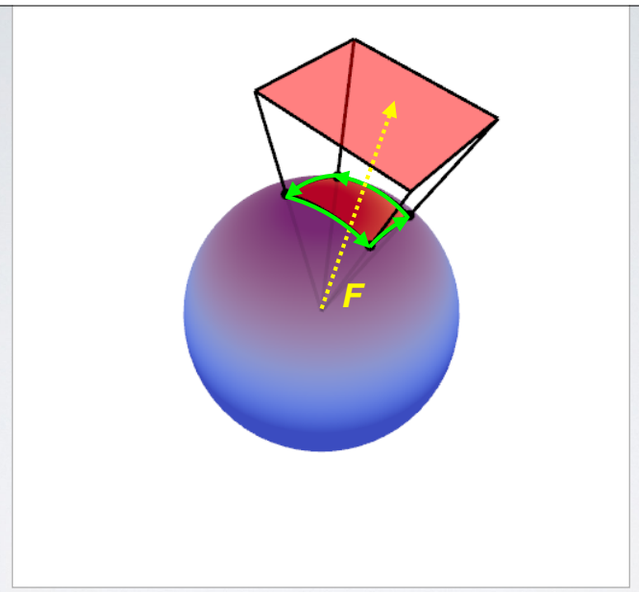
bonus: texturing

In the paper we also covered a way to do textured area lights with a prefiltered texture. For various reasons, the lookup direction we chose to use is the direction perpendicular to the transformed polygon. While this is well behaved, it doesn’t always give accurate results (see the paper for examples).



filtered border region

We also had to add a border region to the texture to handle cases where the lookup is outside of the original texture.



use *F* direction for texture fetch

An attractive alternative, suggested by Brian Karis, is to use F for the lookup direction.

This has the advantage that the F always intersects the polygon, so we no longer need to add a border to our prefiltered texture!

Summary

- Numerical issues vanquished
- Current performance for diffuse + specular: 0.9ms, PS4 @ 1080p
- Scaled the Dark Tower of implementation?
- Future: updated code (GitHub) and notes

Through all of these changes, we were able to combat visual quality issues and increase performance significantly.

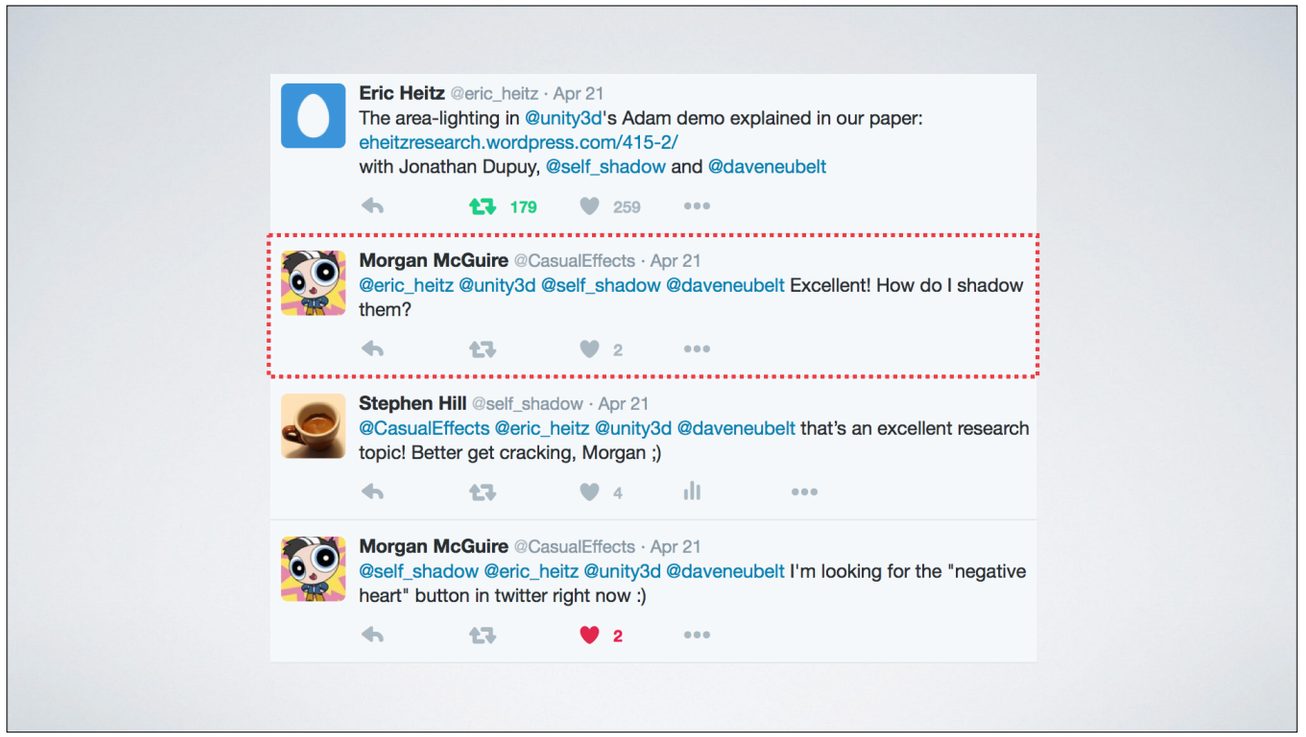
That said, area lighting is still relatively expensive compared to point lighting, so there's always room for further improvement.

We'll be following up on this talk with more detailed notes – containing a few additional topics (such as Fresnel) and optimisations – and updated source code.

elephant in the room

The focus of the paper was on area lighting, but there’s still another tower to climb…

Image credit: Elephant in the Room, Barely Legal show, Banksy. © REUTERS/Fred Prouser, 2006.



…area light shadows! It's an open research problem.

Someone should really solve this! :)

Thanks

- Jonathan Dupuy, David Neubelt
- Robert Cupisz + Unity demo team
- Brian Karis, Stephen McAuley
- Yves Jacquier, Alexandre Pichette, Olivier Pomarez
- All of you! :)

We would like to thank the following people.

References

- [Arvo95] *Applications of Irradiance Tensors to the Simulation of Non-Lambertian Phenomena*, SIGGRAPH'95
- [Baum89] *Improving Radiosity Solutions Through the Use of Analytically Determined Form-Factors*, SIGGRAPH'89
- [Lagarde & de Rousiers14] *Physically based shading in theory and practice: Moving Frostbite to PBR*, SIGGRAPH'14
- [Lambert1760] *Photometria, sive de mensura et gradibus luminus, colorum et umbrae*, 1760
- [Snyder96] *Area Light Sources for Real-Time Graphics*, Technical Report, 1996