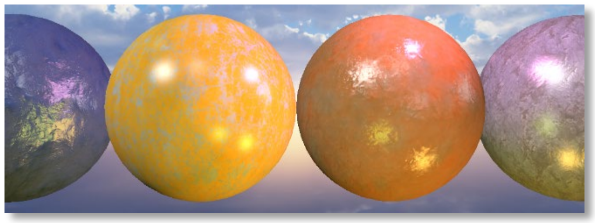
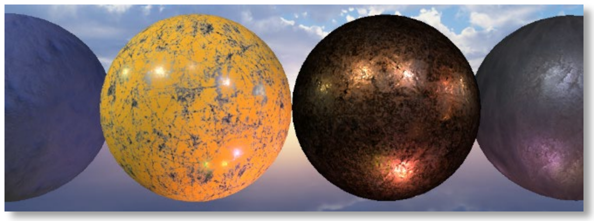
Physically Based Rendering

A more physically-accurate lighting model

What is Physically Based Rendering?

Physically Based Rendering (PBR), sometimes called Physically Based Shading (PBS), is a class of lighting model that more accurately approximates interactions between lights and materials. It makes rendered surfaces look more realistic than classic Lambert (diffuse) + Phong (specular) lighting models.



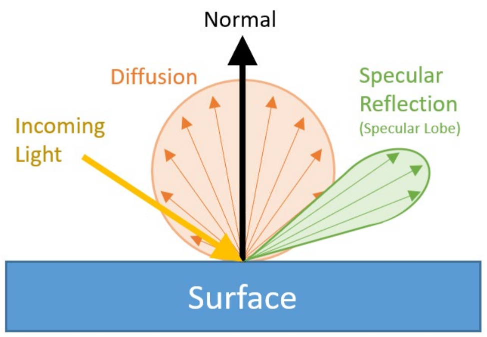


To the right, you'll see two sets of spheres. The top spheres are using the Lambert + Phong lighting model. The bottom spheres are using a physically based lighting model. Both sets of spheres are using the exact same surface color textures.

Notice how the bottom metal spheres, including the "scratched" areas of the yellow sphere, look drastically different than the Lambert + Phong spheres. PBR lighting models take into account whether a surface is metallic (conductor) or non-metallic (dielectric), as metals and non-metals diffuse and reflect light differently.

Additionally, the strength of specular highlights is greatly diminished on rougher surfaces, like the left-most blue sphere and right-most iron sphere. This is another important part of physically based models.

Lighting Revisited: Diffuse & Specular

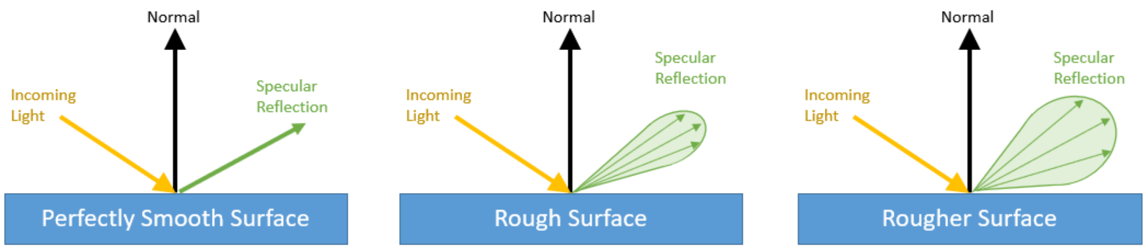


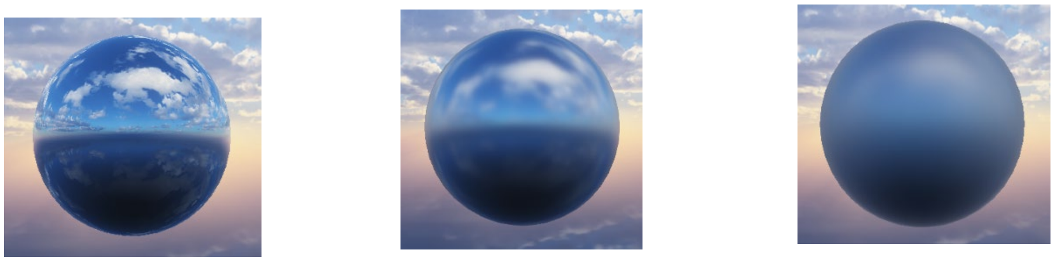
Before we get into the details of PBR, it's important to understand exactly what it is we're trying to approximate with our lighting equations. The diagram to the right shows the direction of diffusion and specular reflection on a surface, given incoming light at a particular angle. Some amount of that light is reflected and the rest is diffused.

Notice that diffusion spreads out in all directions while the specular reflection's direction is dependent upon the direction of the light. If our viewpoint in the 3D scene (our virtual camera) is within the direction of the *specular lobe*, we'll see the reflection of this particular light source. The closer to the center of the lobe, the stronger the reflection appears. For diffusion, it doesn't matter where our virtual camera is placed; the camera will always receive the same amount of diffusion. In other words, the diffuse component looks the same from any angle while the reflection can only be seen at certain angles.

Roughness & Reflections

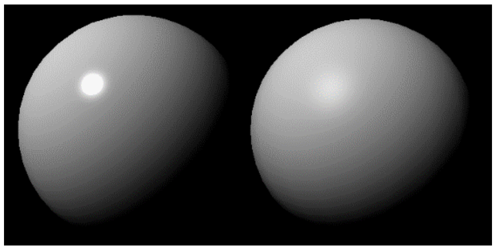
As the roughness of a surface increases, so too does the size of the specular lobe, as shown below. A perfectly smooth surface, like a mirror, has incredibly precise reflections. Rougher surfaces have reflections that are more spread out, appearing blurrier to the viewer.





In fact, on a surface that is extremely rough, the specular lobe would be so large it would reflect in all directions, which is essentially diffusion! A perfect diffuse surface would have no observable specular reflections because the light is bouncing in all directions, though there are no truly perfect diffuse materials in the real world.

Issues with Phong Reflection

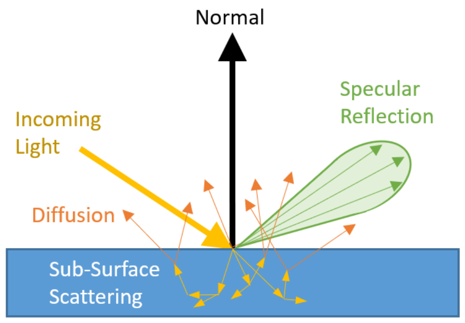


In the Phong reflection model, the roughness of a surface, and therefore the size of the specular reflection, is controlled through a *specular power* parameter. A high specular power results in more precise reflections, as if the surface is very shiny, meaning the actual highlights (the bright spots) are smaller. A lower specular power results in larger highlights, corresponding to a surface that is more matte (less shiny).

This *specular power* parameter is essentially modelling the roughness of the surface. Unfortunately, the numbers used for specular power for Phong reflectance don't correspond to any real-world units. The value is simply a knob for artists to tweak until things look good. On the high end, there is no upper bound to specular power that results in a perfect reflection. On the low end, Phong does not work correctly for surfaces that are perfectly diffuse, as a specular power of zero breaks the equation. This becomes problematic if we want a single equation that can work for all surfaces.

This is another area that physically based lighting models aim to improve. PBR uses a normalized roughness value, where a roughness of 0 corresponds to a perfectly smooth surface and a roughness of 1 corresponds to a perfectly diffuse surface.

Diffusion Details



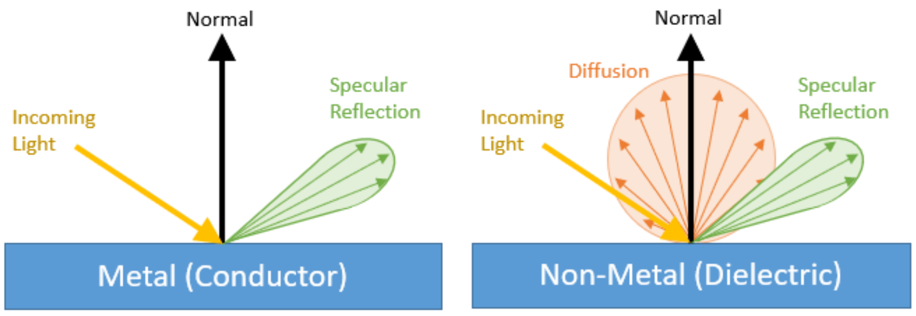
As you might suspect, actual light is much more complex than our simple model. What we're modeling as diffusion is actually light entering a surface, bouncing around and exiting in different places, at a microscopic level. Some of that light gets fully absorbed (turning into heat) before leaving the surface.

While our greatly simplified diffusion model is meant to approximate the sub-surface scattering of light, some games will specifically model this interaction for very translucent materials like skin, marble and wax.

Conductors vs. Dielectrics

Another distinction physically based lighting models make is whether a surface is metal (a conductor) or non-metal (a dielectric, like plastic, wood or glass). While this might seem arbitrary at first, it's required for physically based modelling of diffusion.

Simply put: dielectric materials exhibit diffusion while metals do not. As discussed above, diffusion is the result of sub-surface scattering. In conductors, however, any and all light that enters the surface is absorbed, and thus, none of it ever leaves.



Energy Conservation

An important aspect of light that our simple Lambert + Phong lighting model doesn't include is *conservation of energy*: a surface cannot reflect more light than it receives. However, our current lighting model calculates Lambert and Phong separately and simply adds the results together. This means we might have full diffusion and full reflection at a single point on a surface (a single pixel in our case), which is physically impossible. If a portion of the incoming light is reflecting, that portion cannot be diffusing, and vice versa.

A physically based lighting model will generally ensure that energy is conserved by limiting diffusion based on the amount of reflection. A basic energy conservation function simply reduces the amount of diffusion by the amount of specular:

$$diffuseBalanceFactor = 1.0 - specularReflection$$

Considerations for a Physically Based Lighting Model

A quick recap of what we've covered so far:

- Light striking a dielectric (non-metallic surface) will partially reflect and partially diffuse
- Light striking a conductor (metallic surface) will only reflect
- Regardless of surface, the size and strength of the reflection depends on the surface's roughness
- Outgoing reflection + outgoing diffusion must never exceed the incoming energy

A physically based lighting model needs to account for each of these. To that end, we'll need:

- A representation of the surface's *roughness* within the [0,1] range
- A representation of the surface's *metalness*, generally 0 (non-metal) or 1 (metal)
    - Note: values between 0 and 1 are possible due to interpolation and must be handled
- A physically based *specular BRDF* with *roughness* and *metalness* parameters
- A physically based *diffuse BRDF*
- An *energy conservation function* to balance *diffuse*, taking *metalness* into account
    - Where metal is 0: return a portion of *diffuse* based on *specular*
    - Where metal is 1: return zero

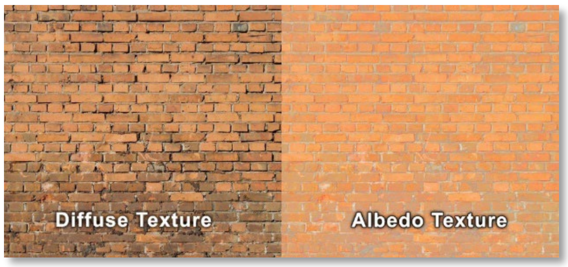
Changes for PBR

We'll still be calculating a diffuse component and a specular component for each light. Those will be balanced to conserve energy before being added together. This represents the light from a single light source at a point on the surface (the current pixel). The results from multiple lights will be added together to compute the final shading for each pixel. However, our existing ambient term is not physically based and should be *removed entirely* or our surfaces will be abnormally bright.

Most of our existing inputs to lighting, like the surface normal and color, will remain largely the same. We'll also need new inputs for roughness and metalness, both of which can come from textures (since each pixel on a surface can have different values for each).

If you were using a specular map previously, it will be replaced entirely by a roughness map.

We'll also be replacing some of our lighting calculations with physically based ones. If you previously had helper functions to calculate diffuse and specular, you can create new functions for our physically based BRDFs. Then, before adding the results to the running light total for the current pixel, you'll need to balance those diffuse & specular results.



We'll need to ensure our existing surface color textures are actually *albedo textures*. What's albedo? In Latin, it means "whiteness". Mathematically, the term describes the amount of light leaving a surface in relation to the amount of incoming light. For us, an *albedo texture* contains the colors of a surface under 100% even, white light – with no shadows, ambient occlusion or other shading "baked in". All of those phenomena should be handled by our lighting calculations.

Let’s Get Physical: Diffuse BRDF

We’re finally ready to dig into the guts of PBR. Let’s start with the diffuse BRDF. We previously used a Lambertian BRDF, which we implemented with a simple dot product between the surface normal and the vector towards the light (often referred to as “N dot L”).

Side note: We’ve actually been missing a proper part of the Lambert BRDF in our implementation that makes it energy conserving by itself. Since light spreads out in all directions during diffusion, even the brightest spot on a surface (where the normal faces the light) shouldn’t send 100% of its light towards the viewer. Only a portion of that light actually hits our eye.

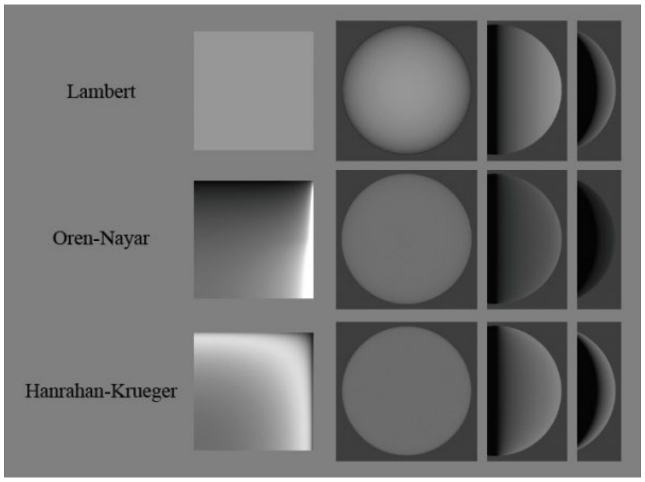
To compensate for that spread, we should be dividing by π.

$$diffuse = \frac{(N \cdot L)}{\pi}$$

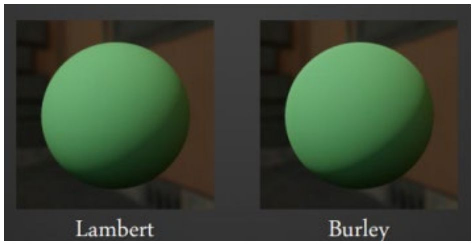
Why π? Without going into the full derivation, it’s due to the light scattering over a hemisphere.

While technically correct, this produces a darker result than our previous version. Some engines, like Unity, even skip the division by π entirely. Why? To give a more predictable result, especially for artists and designers who might expect a 100% white surface under 100% white light to appear 100% white. With the proper division by π, the result would instead be a grey color.

Lambert is great, but it’s not the only BRDF out there. Below you’ll see examples of several others.



Real Image
Lambertian Model
Oren-Nayar Model



With all of these options, which should we choose? As it turns out, many PBR implementations, including Unreal Engine 4 and Unity, still use Lambert! Why? Because, while Lambert is not perfect, it’s quite good and it’s really, really fast!

The Burley diffuse BRDF from the seminal paper Physically Based Shading at Disney, for instance, takes roughness into account but is *much* more complex than a single dot product. Epic Games found that the difference wasn’t worth the extra computation cost at the time of Unreal Engine 4 development. Godot Engine 3, a more recent release, uses Burley as its default diffuse BRDF, but keeps Lambert as an option.

We’ll stick with Lambert for our diffuse BRDF, which won’t require any changes (though we could add the division by π as noted above). It’s good enough for Unreal and Unity, so it’s good enough for us!

Let's Get Physical: Specular BRDF

Since we're still using Lambert as our diffuse BRDF, it means the bulk of our changes come from the specular BRDF. We've already discussed some of the issues with Phong, so we'll move right along to the most popular physically based specular BRDF: the Cook-Torrance microfacet specular BRDF.

$$specular(v, l) = \frac{D(n, h, \alpha)F(v, h, f_0)G(n, v, l, \alpha)}{4(n \cdot v)(n \cdot l)}$$

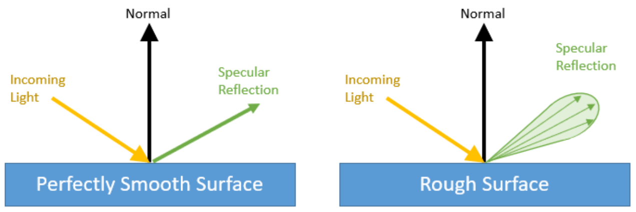
The terms found in this equation are as follows:

Term	Meaning
$\boldsymbol{v}$	View vector – from surface to camera
$\boldsymbol{l}$	Light vector – from surface to light
$\boldsymbol{n}$	Surface normal
$\boldsymbol{h}$	Half vector – halfway between v and l, calculated as $h = normalize(v + l)$
$\boldsymbol{\alpha}$	Roughness – in the range [0, 1]
$\boldsymbol{f_0}$	Specular reflectance at normal incidence (how reflective is the surface head-on?)
$\boldsymbol{D}(\,)$	Normal distribution function
$\boldsymbol{F}(\,)$	Fresnel term
$\boldsymbol{G}(\,)$	Geometric shadowing

Before we get into the details of our new specular BRDF, we need to discuss *roughness* and *microfacets*.

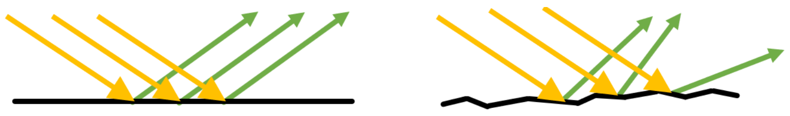
What Even is Roughness?

The following surfaces, which we might describe as being "flat", have normals that clearly point straight up, at least at a *macro level* (our zoomed-out view).



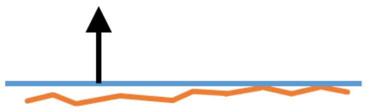
However, *something* is different about each surface, since the reflections on the surfaces are distinct. In this case, each surface has a different *roughness*. But what does that actually mean? To answer this, we'll need to look at the surfaces at a *micro level* (smaller than a single pixel).

If we zoomed *way* in, such that we're looking at a microscopic fraction of the surface, far less than the area covered by a single rendered pixel, we might see that the surfaces actually have different shapes.



Perfectly Smooth Surface Rough Surface

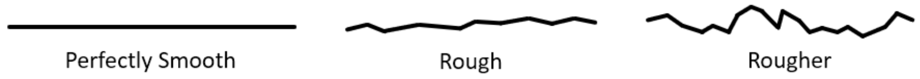
At the macro level (a single pixel), as we saw above, both surfaces have the same overall normal. But at the micro level (a tiny fraction of a pixel), the rough surface is incredibly uneven, resulting in the larger specular lobe.



Rough surface: Micro vs. Macro

Microfacets

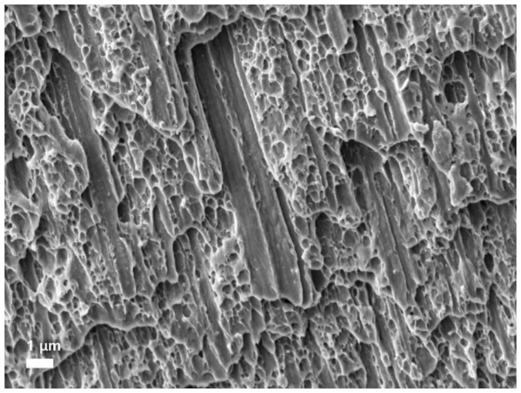
A microfacet BRDF, like Cook-Torrance, regards surfaces as being made up of a myriad of microscopic planes, or *microfacets*, each facing in random directions. The smoother the surface, the more aligned the microfacets are. The rougher the surface, the more disordered the microfacets become.



Each of these microfacets is effectively a perfect mirror, but due to their haphazard orientations, only a fraction of them will actually reflect light directly at the viewer. The rougher the surface, the fewer microfacets within a single pixel contribute to that visible reflection.

Note that there are no actual microfacets in our geometry. This is simply a mathematical model to approximate what is happening at a microscopic level.

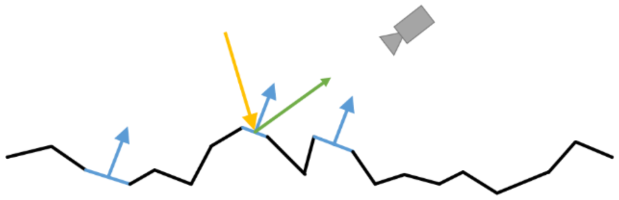
For example, the image to the right is stainless steel under an electron microscope. Notice the amount of detail that, while invisible to the naked eye, would change how light is reflected.



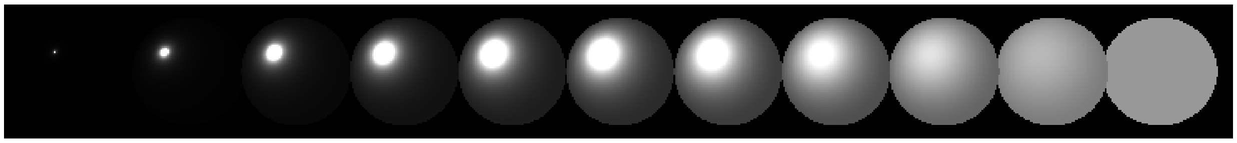
Normal Distribution Function

One component of a microfacet BRDF is modelling the percentage of microfacets whose normals reflect light towards the viewer, as a function of surface roughness. This is the *normal distribution function*.

For a microfacet to reflect light toward the viewer, its normal must be oriented halfway between the light vector (surface to light) and the view vector. As shown to the right, only a fraction of the normals of a rough surface meet this requirement.



How many microfacets within the area of a single pixel are oriented in this way? As mentioned, it depends on the surface's roughness. The rougher the surface, the fewer microfacets within a single pixel contribute to reflection. This also means that as the surface gets rougher, more and more pixels of the surface may reflect *some* light to the viewer, causing the reflection to appear "spread out".

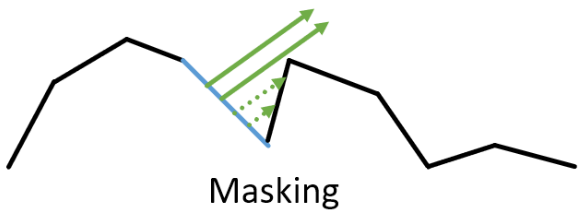


Above you can see the result of a normal distribution function across spheres of different roughness values. When the surface is smooth (left), the microfacet normals match the surface normal, so the "perfect" reflection is extremely tight. As the surface gets rougher, the reflection spreads across the surface but gets duller overall.

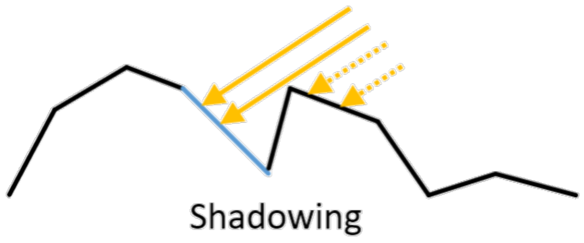
Geometric Shadowing

Our microfacet BRDF must also account for a phenomenon known as *geometric shadowing*, in which the geometry of the micro-surface blocks a fraction of the light, before and after reflection. This acts as a reduction to the overall specular result as a function of surface roughness.

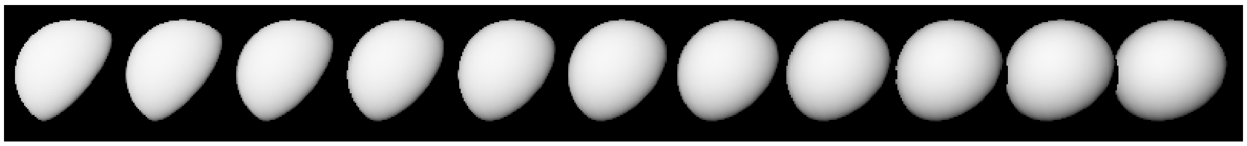
For instance, due to the arrangement of microfacets, light might strike the surface but get blocked before reaching the viewer. This is known as *masking*. A rough surface will inherently mask more light than a smooth surface.



In the opposite direction, light might get blocked by another area of the surface on its way to a microfacet. This is known as *shadowing*. As above, rougher surfaces exhibit more shadowing than smoother surfaces.



Below you'll see another set of spheres with roughness values ranging from 0.0 to 1.0, this time with the results of a geometric shadowing functions. The differences are subtle, but the results do get dimmer as roughness increases, especially at grazing angles.



To the right is the result of geometric shadowing on a normal-mapped sphere with varying roughness. The resulting scalar value acts a multiplier for overall reflection, reducing it where masking and shadowing are high.

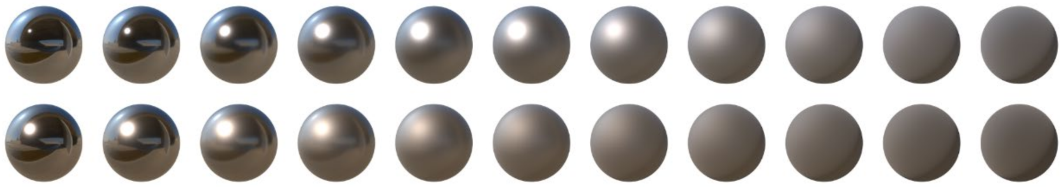
Implementing Cook-Torrance

Here is the Cook-Torrance microfacet BRDF again:

$$specular(v,l) = \frac{D(n,h,\alpha)F(v,h,f_0)G(n,v,l,\alpha)}{4(n \cdot v)(n \cdot l)}$$

While this seems like a lot, the input vectors are all either ones we already have in our shaders or, in the case of h, can be calculated from existing vectors.

Roughness, α, is new and is sampled directly from a single channel of a greyscale roughness texture (very similar to specular maps in that regard). Note that in many PBR implementations, roughness values are remapped as $\alpha = roughness^2$, which gives a more linear result (top row below). Since roughness is artist-driven, this remapping leads to more predictable results and easier asset creation.



The f_0 term is either a color from our albedo texture (for metals) or a constant scalar value (for non-metals). Since we'll need to make this determination per-pixel, we'll also sample a single channel of a grayscale metalness texture and use it to choose an appropriate f_0. The metalness value is also used when balancing diffuse & specular, since metals do not exhibit diffusion.

Some engines combine roughness & metalness into a single texture as an optimization, similar to embedding a specular map into the alpha channel of an albedo texture. However, it is common to find these as separate textures when searching online for PBR assets.

Functional Choices

The big unknowns are the three functions in the numerator: $D(\,)$, $F(\,)$ and $G(\,)$.

Function	Purpose
$\boldsymbol{D}(\,)$	*Normal distribution function* – what percent of microfacets reflect toward us?
$\boldsymbol{F}(\,)$	*Fresnel term* – reflection strength based on viewing angle
$\boldsymbol{G}(\,)$	*Geometric shadowing* – how much does the surface block light?

Here is where we have choices, as there are several options for each of these functions. Brian Karis, senior graphics programmer at Epic Games, documented all of the options analyzed when implementing Cook-Torrance in Unreal Engine 4. We'll more or less use the same functions that Unreal 4 went with, which are detailed in the notes to Real Shading in Unreal Engine 4 from SIGGRAPH 2013.

In the link above, you'll see Physically Based Shading at Disney from SIGGRAPH 2012 (mentioned earlier) had a big influence on PBR in Unreal 4 (among other engines). You might not think of Disney when writing shaders, but their animation and graphics research has far-reaching effects.

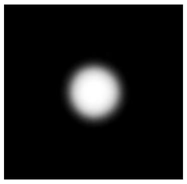
Let's dig into the details for each of the above functions. We'll look at $\boldsymbol{D}(\,)$ and $\boldsymbol{G}(\,)$ first.

***D*( )** - Normal Distribution Function – Trowbridge-Reitz (GGX)

The normal distribution function (NDF) describes the percentage of microfacets that are aligned to reflect light to the viewer. There are many NDFs to choose from, including variations of Phong.

In terms of matching real-world surfaces, one of the most commonly used NDFs for real-time microfacet models is Trowbridge-Reitz (GGX):

$$D(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$



To the right you'll see examples of the GGX distribution (top) and Blinn-Phong (bottom), a variation of Phong that is quicker to calculate. The longer "tail" on GGX more closely matches real-world observations of specular reflections.

> *What does GGX mean? The GG portion stands for "ground glass", which was an example of a rough surface used in the paper that introduced the GGX distribution function in 2007, by Bruce Walter et al.*
>
> *What about the Trowbridge-Reitz part? As it turns out, unknown to Walter et al., the GGX function described in their 2007 paper matches a distribution function from the 1975 paper by Trowbridge and Reitz! Graphics literature often uses both names when referring to the function.*

Here is the function in unoptimized HLSL. Note the remapping of roughness as $\alpha = roughness^2$ before being squared again as part of the function, and a minimum roughness being applied to ensure the denominator does not become 0 when roughness is 0 and $n \cdot h$ is 1. MIN_ROUGHNESS is 0.0000001.

```
float D_GGX(float3 n, float3 h, float roughness)
{
	// Pre-calculations
	float NdotH = saturate(dot(n, h));
	float NdotH2 = NdotH * NdotH;
	float a = roughness * roughness; // Remapping roughness
	float a2 = max(a * a, MIN_ROUGHNESS);

	// Denominator to be squared is ((n dot h)^2 * (a^2 - 1) + 1)
	float denomToSquare = NdotH2 * (a2 - 1) + 1;
	return a2 / (PI * denomToSquare * denomToSquare);
}
```

$\boldsymbol{G}(\,)$ – Geometric Shadowing – Schlick-GGX

Geometric shadowing describes how the microfacets of a rough surface block a portion of the incoming and reflected light. Many different functions exist for geometric shadowing, but most follow the Smith model to start:

$$G(n, v, l, \alpha) = G_1(n, l, \alpha) G_1(n, v, \alpha)$$

In this formulation, we run the same function, $G_1(\,)$, twice – once for the light vector (for *shadowing*) and once for the view vector (for *masking*) – and combine the results. This leads us to the next question: what do we use for $G_1(\,)$? Unreal Engine 4 uses an equation presented by Christophe Schlick in his 1994 paper, but tweaked slightly to better fit with the GGX normal distribution function:

$$G_1(n, v, \alpha) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

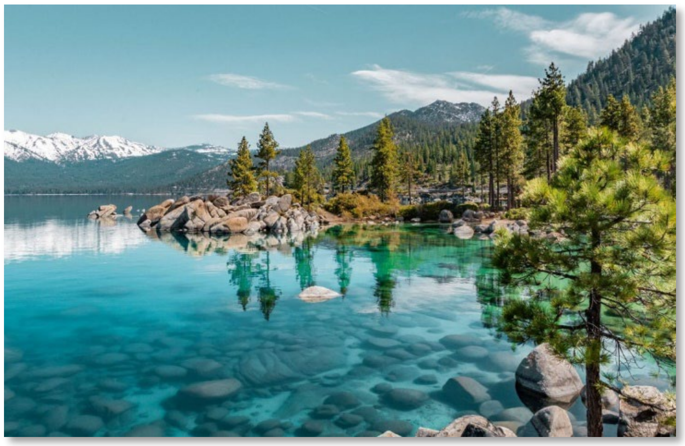
The tweak mentioned above comes into play with the new variable k. It is a remap of roughness, such that $k = \alpha/2$. Recall that α is already roughness remapped as $\alpha = roughness^2$. However, both Unreal Engine 4 and Disney chose to further remap roughness from the [0,1] range to the [0.5,1] range in this function *before squaring it* to reduce high specular values for very low roughness inputs. The end result:

$$k = \frac{(roughness + 1)^2}{8}$$

Here is the $G_1(\,)$ function in unoptimized HLSL. Note that it will need to be called twice, with slightly different parameters, as part of our final BRDF calculation. An optimized version might combine the two calls into one to avoid recalculating k and $N \cdot V$, among other things.

```
float G_SchlickGGX(float3 n, float3 v, float roughness)
{
	float k = pow(roughness + 1, 2) / 8.0f; // End result of remaps
	float NdotV = saturate(dot(n, v));
	return NdotV / (NdotV * (1 - k) + k);
}
```

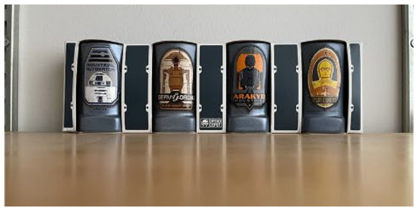
$\boldsymbol{F}(\,)$ – Fresnel – Schlick's Approximation



The Fresnel reflectance function models the ratio of reflected light and refracted light as our viewing angle changes.

This effect can be observed in the photograph of Lake Tahoe to the right. When looking down through the water, there is much less reflection and much more refraction. In contrast, when looking across the surface at a grazing angle, the light is almost entirely reflected.

Though this effect is very obvious on large bodies of water, it actually occurs on every surface we see. All surfaces become reflective at grazing angles, even ones that we might not consider to be “shiny” like wood, pavement or cardboard. Below you’ll see some knickknacks from three different angles: above (left), at an angle (middle) and head-on (right). Notice how the reflection on the wooden shelf becomes much more pronounced as we look *across* the surface (right image).



In the same 1994 paper linked above, Schlick presents a computationally inexpensive (and quite accurate) approximation of the full Fresnel term:

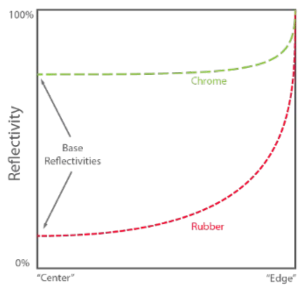
$$F(v,h,f_0,f_{90}) = f_0 + (f_{90} - f_0)(1 - v \cdot h)^5$$

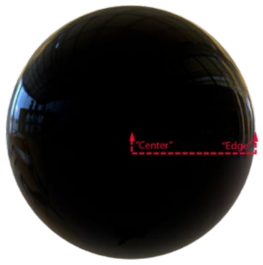
The f_0 parameter is specular reflectance at 0 degrees, also known as *normal incidence*. This means “the reflectance when light strikes the surface head on (matching the normal)”. This value is different for every real-world material.

The f_{90} parameter is specular reflectance at grazing angles (90 degrees), which is very, very close to 1.0 for most materials. So close, in fact, many implementations replace it with the constant 1.0 and drop the parameter entirely:

$$F(v,h,f_0) = f_0 + (1 - f_0)(1 - v \cdot h)^5$$

In the graph to the right (a simplified representation, not an exact scientific measurement), you can see that the two materials – chrome and rubber – have very different base reflectivities (f_0) but both become highly reflective at grazing angles. In other words, even if a surface has next to no reflection “head on”, it becomes more reflective at extreme angles, as shown on the sphere to the right.

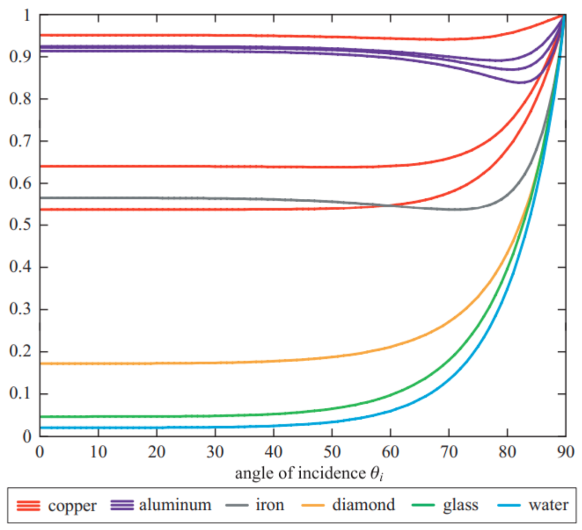




Here is the simplified $F(\,)$ function in HLSL:

```
float3 F_Schlick(float3 v, float3 h, float3 f0)
{
        float VdotH = saturate(dot(v, h));
        return f0 + (1 - f0) * pow(1 - VdotH, 5);
}
```

What About f_0?



The vector inputs to Fresnel have already been discussed, but what values do we use for f_0? For this, we'll need some real-world measurements.

To the right you'll see a graph of the reflectivity of various materials as the angle of incidence increases. f_0 corresponds to the left side of the graph. As the angle approaches 90 degrees, the reflectivity of these surfaces approaches 1.0 (hence the simplification presented above).

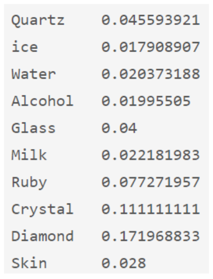
> *Not every surface truly tops out at 1.0 reflectivity at grazing angles, but it's an alright assumption to make for real-time.*

There are a few interesting observations here. First, metals like copper and aluminum have significantly different reflectivity across the color spectrum (the three lines for each metal in the graph correspond to R, G and B). This means that these metals reflect red, green and blue differently, leading to a tinted reflection. Second, most non-metals have extremely low reflectivity at normal incidence compared to metals. Glass and most liquids are some of the lowest, while gemstones are slightly higher but still nowhere near metals.

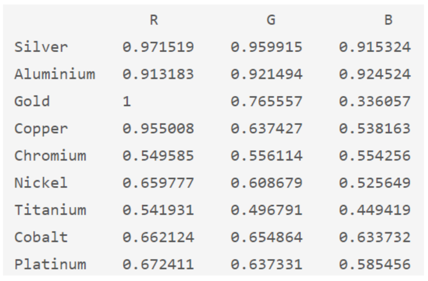
Finding f_0 Values

We need f_0 values to plug into our specular BRDF equation, and we know that different materials (glass, plastic, wood, bronze, gold, etc.) have different values. For dielectric surfaces, f_0 is a scalar value. For conductors, however, f_0 is chromatic (a 3-component color instead of a scalar). How do we find them?

Here are f_0 values of several real-world materials, from the article Feeding a Physically Based Lighting Model by Sébastien Lagarde, director of rendering research at Unity. The dielectric materials have a scalar value, while the metals have three values, corresponding to an RGB color.



Quartz	0.045593921
ice	0.017908907
Water	0.020373188
Alcohol	0.01995505
Glass	0.04
Milk	0.022181983
Ruby	0.077271957
Crystal	0.111111111
Diamond	0.171968833
Skin	0.028



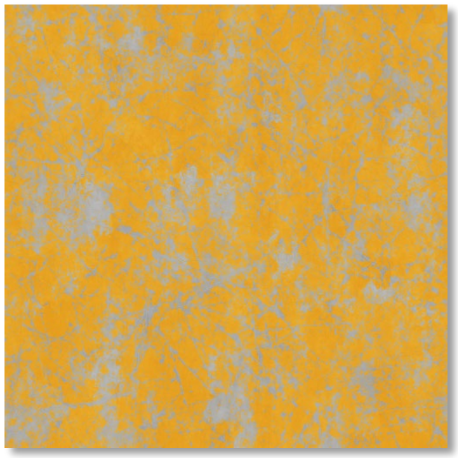
	R	G	B
Silver	0.971519	0.959915	0.915324
Aluminium	0.913183	0.921494	0.924524
Gold	1	0.765557	0.336057
Copper	0.955008	0.637427	0.538163
Chromium	0.549585	0.556114	0.554256
Nickel	0.659777	0.608679	0.525649
Titanium	0.541931	0.496791	0.449419
Cobalt	0.662124	0.654864	0.633732
Platinum	0.672411	0.637331	0.585456

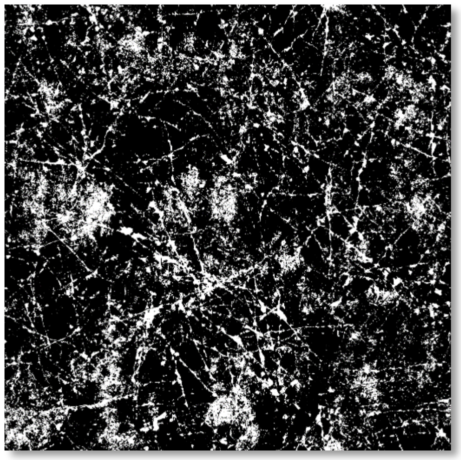
To simplify dielectrics, many renderers simply use 0.04 as the f_0 for all non-metals. This is the exact reflectivity of glass and very close to most plastics. We'll do the same (though it could be parameterized if we wanted). However, due to the f_0 of metals being so different, we must parameterize their values.

Using f_0 Values

We know we need an RGB color for the specular reflectivity of metals. Does this mean yet another texture? Fortunately, based on what we know about metals and light, it does not. Recall that metals do not exhibit diffusion: their final diffuse term is zero. This also means our albedo texture, which is multiplied by diffuse, is *irrelevant for pixels that represent metals*.

Three whole color channels going to waste when a pixel represents a metallic surface. Whatever shall we do? Let's fill those pixels with the metal's specular color instead! You can see an example of this in the albedo texture and corresponding metalness texture below. Notice that the grey areas in the albedo are marked as metal (a value of 1.0) in the metalness texture.





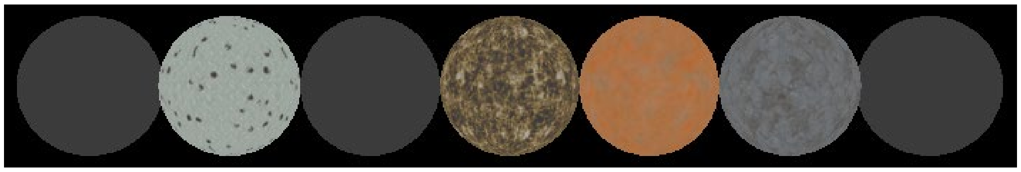
This is great for artists, as they can simply paint the metal's specular color directly into the albedo texture. This is also great for us, as we can sample this single texture and easily adjust how we use the resulting color, right in the shader, based on the pixel's metalness value.

Below is an example of the metal/non-metal f_0 interpolation in HLSL. F0_NON_METAL is the scalar value 0.04, which will be turned into the 3-component vector (0.04, 0.04, 0.04) for us by the compiler. Note that this must be an interpolation and not a binary choice due to texture filtering.

```
float3 f0 = lerp(F0_NON_METAL, surfaceColor, metalness);
```

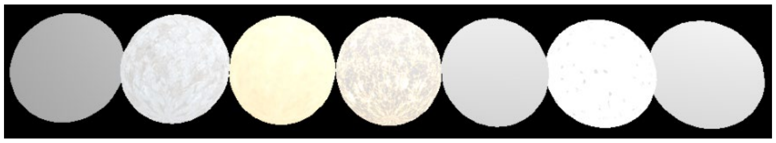
Fresnel Results

Below you can see the results of Fresnel. The first, third and last spheres are non-metals, hence the identical reflectance. The rest are metals, which have chromatic Fresnel results.



Note that microfacet Fresnel uses the view vector and the half-vector between view and light. Since the camera is facing a similar direction to the light in this screenshot, the results appear mostly uniform.

To the right are the results from the opposite side, where the Fresnel term approaches 1.0 (white).



Putting it All Together

We've got our three functions, so we're ready to translate the overall Cook-Torrance BRDF into code. This is rather straightforward other than perhaps the denominator, which is tweaked slightly to prevent a possible divide by zero when one of the dot products bottoms out.

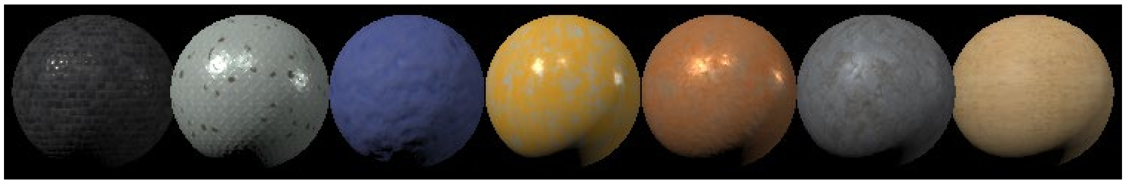
$$specular(v,l) = \frac{D(n,h,\alpha)F(v,h,f_0)G(n,v,l,\alpha)}{4(n \cdot v)(n \cdot l)}$$

Here is the function in unoptimized HLSL.

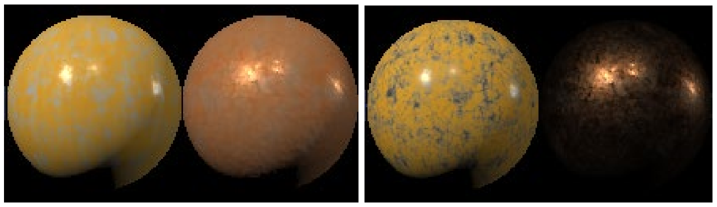
```
float3 MicrofacetBRDF(float3 n, float3 l, float3 v, float roughness, float3 f0)
{
    float3 h = normalize(v + l);

    // Run each function: D and G are scalars, F is a vector
    float  D = D_GGX(n, h, roughness);
    float3 F = F_Schlick(v, h, f0);
    float  G = G_SchlickGGX(n, v, roughness) * G_SchlickGGX(n, l, roughness);

    // Replacing dot product multiplication with max() to prevent// divide by 0
    return (D * F * G) / (4 * max(saturate(dot(n, v)), saturate(dot(n, l))));
}
```

There are many opportunities to optimize the overall specular BRDF code, both here and in the individual D/F/G functions. The code is presented in its current form to closely match the formula above, but could be restructured to reduce repetitive calculations (like dot products).

Now that we have a new specular BRDF, we can use it in place of our previous one (Phong). However, if all we do is swap one for the other, things will *not* look right just yet, as shown here:

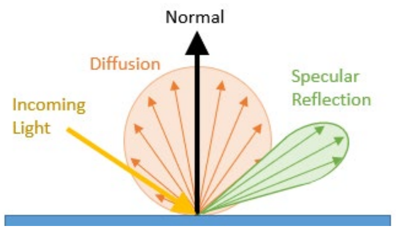


The specular reflections are looking good overall, but there's an issue with the metal spheres: they're still exhibiting diffusion! We haven't accounted for the lack of diffuse lighting on metallic surfaces yet. Here is a comparison of the incorrect versions (left) and correct versions (right):



The fix is to zero out the diffuse term on any pixel that represents a metallic surface. As discussed earlier, we also need to balance diffuse and specular to account for *energy conservation*. Since the goal is to (potentially) reduce the diffuse term, we can structure it to also cut diffuse entirely for metals.

Energy Conservation



Without energy conservation, our surfaces are most likely brighter than they should be. In the real world, the portion of light reflecting off a surface is not diffusing. However, we're calculating diffuse and specular independently and simply adding them together. Instead, we need to balance the two terms.

Recall that the Fresnel term describes the ratio of reflected (specular) and refracted (diffuse) light. Our microfacet BRDF already contains this term: the $F(\,)$ function. We'll need to apply the remainder of the ratio, $1 - F(\,)$, to our diffuse term before adding it to specular.

We might wrap this up into a helper function like so in HLSL:

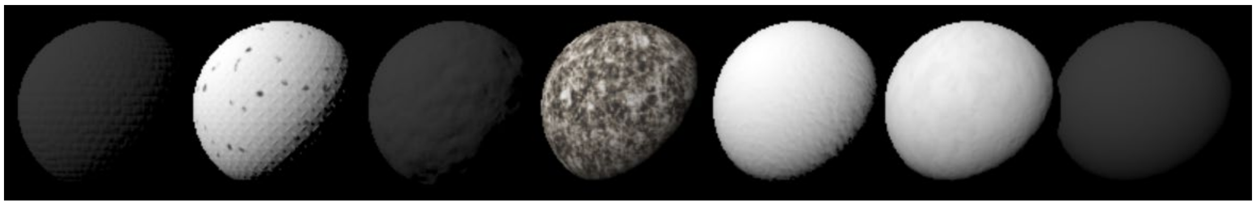
```
float3 DiffuseEnergyConserve(float3 diffuse, float3 F)
{
	return diffuse * (1 - F);
}
```

Since the purpose of this function is to adjust the diffuse term we ultimately add to our lighting results, it would be a great place to handle the lack of diffusion on metallic surfaces. We can add a metalness parameter to further adjust the result. When metalness is 0, we keep 100% of the result. When metalness is 1, we keep 0% of the result, effectively cutting the diffusion entirely when a pixel represents a metal.

```
float3 DiffuseEnergyConserve(float3 diffuse, float3 F, float metalness)
{
	return diffuse * (1 - F) * (1 - metalness);
}
```

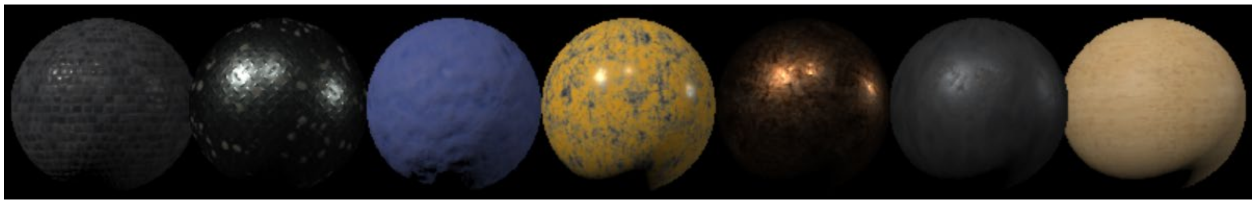
Note that this requires the results of our Fresnel function <u>outside</u> of our MicrofacetBRDF() function. There are a number of ways of handling this, each of which may require a slight refactor to the code presented here. (The least efficient option, of course, would be re-running the F() function a second time in the main() function of our shader.)

Presented below is the reduction in diffuse for each pixel, calculated as the difference between the original and "balanced" diffuse terms. Metals have the most reduction since the diffuse is cut almost entirely. Non-metals, on the other hand, have a slight, but noticeable, reduction in diffuse.



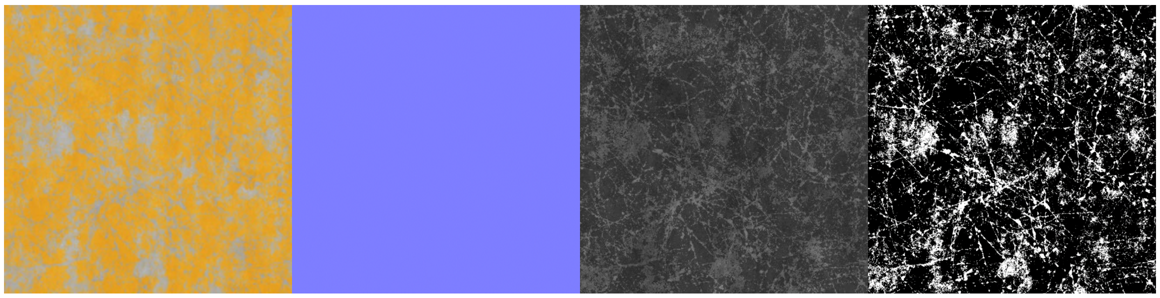
Final Results

Below are the results of seven spheres lit by three directional lights using our new specular BRDF!



Appendix: PBR Asset Reference

For your reference, here are the PBR-appropriate assets used for the center yellow/metal sphere above. It is meant to look like yellow paint partially scratched off of a metal surface. The textures below are, left to right, albedo, normal map, roughness, metal.



Note that the normal map is effectively “flat”, meaning it won’t alter the surface normals at all for this particular material.