Introduction to Physically Based Rendering
Walt O'Connor

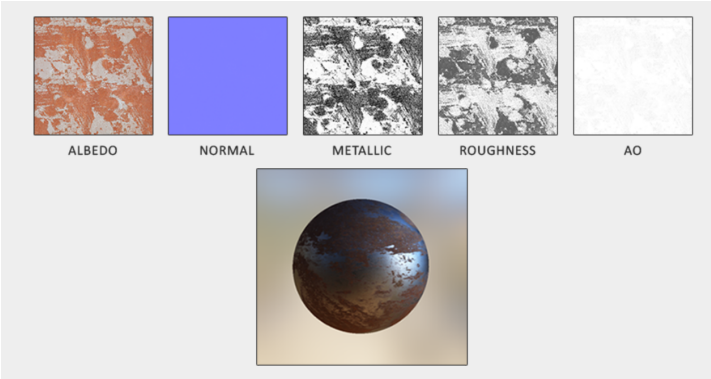
What is possible with a full PBR pipeline

Section 1: What is PBR (Physically Based Rendering)

Physically Based Rendering attempts to roughly model how light works in reality by simulating light and the material it is striking using physical properties about the material (compared to something like Phong shading, where a number of parameters with little connection to the physical properties of the material being simulated are tweaked until you have a somewhat convincing replica). Physically based rendering has existed as an idea since the 80s, however it only saw implementation in industry starting in the early 2010s. Nowadays it is ubiquitous in games and film, and most popular game engines ship with an extensive PBR pipeline by default these days (this is what you are using in Unity). PBR is not a specific set of equations and code used for rendering the way Phong shading is, rather it is a set of concepts and approaches that can be applied in different ways depending on the demands of the system being implemented. The techniques and approximations we will be implementing today are rooted primarily in work done at Epic Games on Unreal Engine, and Disney for their animated films.

A normal PBR workflow includes *Roughness*, *Metallicity*, *Normal Mapping*, *Ambient Occlusion* and *Image Based Lighting* (Cubemaps), however we will only be covering *Roughness* and *Metallicity,* as *Ambient Occlusion* is trivial to implement if you have baked output from your 3D

modeling program, *Normal Mapping* requires substantial modification to your input code to compute tangent spaces, and implementing *Cubemaps* would probably take longer than the rest of this project combined. Some advanced workflows also include a *Displacement Map* which gets fed to a tessellation shader to add triangles to the scene in order to simulate surface features, but that is far beyond the scope of this work.



Normal PBR input textures

PREREQUISITES:

You need to have a model on hand that has accurate UV/texture coordinate mapping. You need to be able to successfully load textures and provide them to your shader. The textures I provide are in a number of different formats (the albedo is 8 bit RGB while the metallic and roughness are 16 bit grayscale), and you need to be able to read these and make OpenGL understand what the bytes you are giving it mean (or just convert everything to 8 bit RGB before giving it to OpenGL like I did). In your fragment shader main function, I expect there to be:

```
vec3 albedo = texture(albedo_tex, tex_coords);
float roughness = texture(roughness_tex, tex_coords).r;
float metallic = texture(metallic_tex, tex_coords).r;
```

If you don't have this setup you might have to adapt what you are doing to match your inputs. We are making the incorrect assumption that every vertex in your model is the same distance from the source light (this is how you model a directional light), and the distance will be computed from the vector to your light, so you should pass the unnormalized vector to the light in, use it to compute intensity, and then normalize it.

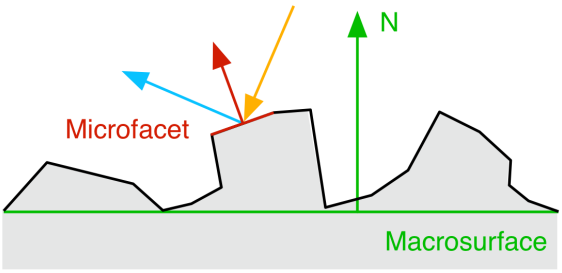
This assignment consists of two parts: getting the texturing working (part 1) and getting the shader working (part 2). These instructions only cover part 2. *THERE IS A TO DO LIST ON THE LAST PAGE COVERING EXACTLY WHAT YOU NEED TO DO FOR BOTH PARTS.*

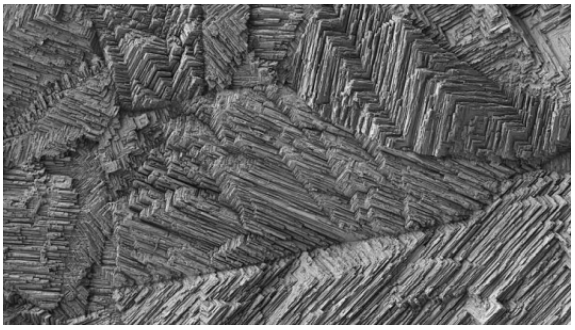
Section 2: Fundamental Ideas Behind PBR

At a foundational level, Physically Based Rendering typically entails two fundamental concepts:

1. **The microfacet surface model**
2. **Energy conservation**

The Microfacet Surface Model:



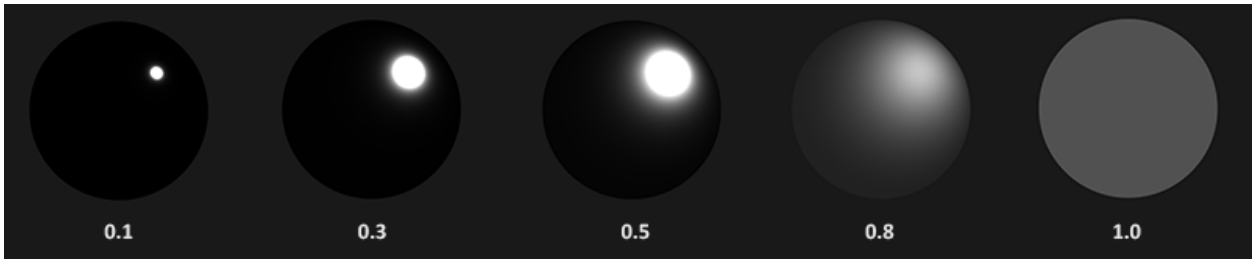


A "smooth" object under a microscope

For physically based rendering, surfaces are treated as a jagged collection of microscopic perfectly reflective mirrors. The *Roughness* parameter controls the bumpiness of the surface at a microscopic level. This would obviously be completely impractical to compute, so instead a statistical approximation is used to model how much a surface of a given roughness would reflect to a camera at a given position, given a light at another position. In general a smooth surface will tend to reflect light directly along the reflection vector, while a rough surface will scatter it in many directions. A perfectly rough surface has Lambertian properties.

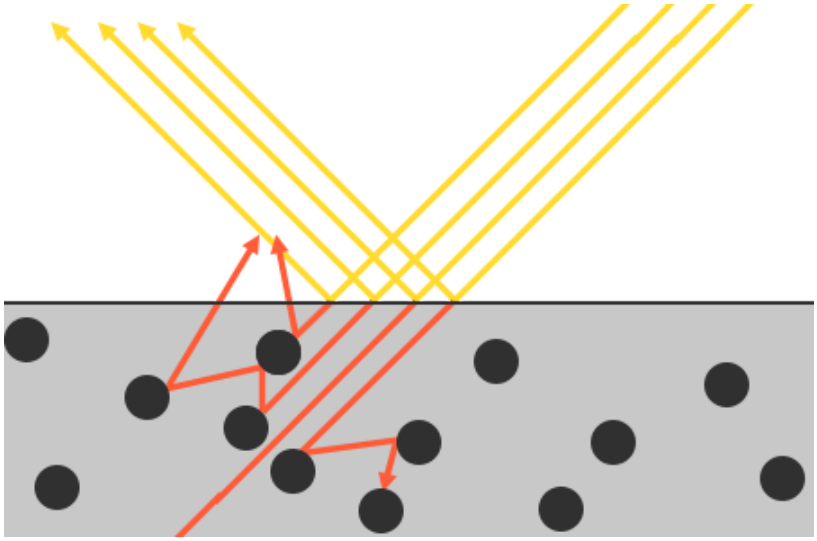
Fig. 3.2 Specular, diffuse, and spread reflection from a surface.

Specular reflection occurs at Roughness = 0, Diffuse at Roughness = 1, and spread anywhere in between.



Appearance of a gray sphere at different levels of roughness, notice that the size of the reflection increases with roughness.

Energy Conservation:
The above diagram gives rise to an interesting observation: *energy is conserved*. The specular highlight on the smooth sphere is very tight and bright, but the highlight on the rough sphere is very large and dim. Ultimately the same amount of light is making it to the camera for every sphere, but where it comes from is different. This is obviously consistent with how real light interacts with objects, but it is a large development over the Phong model where the specular highlights would trivially violate conservation of energy and you had to manually tune them down with the Ks parameter. At this point we have to start making a distinction between *diffuse refraction* and *specular reflection*: when light hits an object, part of it is reflected immediately (*specular*), and some of it enters in to the surface of the object where it is bounced around (*refracted*) until it exits the object some distance away from where it entered. For this project we are assuming opaque objects so we can make the assumption that light will exit the object fairly close to where it entered, but if you want to start modeling transparent or more complex materials (skin) then this assumption no longer holds and you need to use refractance equations and subsurface scattering to model the light. It is important to note that refraction and reflection must be mutually exclusive in order to maintain conservation of energy (i.e. light_refracted + light_reflected = light_in)



Yellow light is reflected, red light is refracted, some is absorbed by the object

Aside 1: Metallicity
Everyone knows metals interact with light differently than dielectrics (non-electrically conductive/non metallic objects), especially graphics artists trying to recreate the appearance of metal, but far fewer people know what makes metal's interaction with light so unique. It's actually quite simple: any light that is not immediately reflected off a metal is entirely absorbed (all the red arrows disappear in the above diagram). The reason this happens is also quite simple: electromagnetic waves can not travel through electrical conductors, instead they are absorbed and induce an electrical current. This is also why we call non-metallic materials dielectrics in PBR--they look the way they do because they don't conduct electricity. This means that most metals have little to no diffuse component, and instead they just reflect light (and their color comes from which wavelengths of light they are prone to absorbing and reflecting).

Generally materials are either fully metallic or fully dielectric, but metallicity is represented as a range between 0 and 1 rather than a binary because many materials are composites of metallic and nonmetallic components, and the range lets you describe the composition of the material.

The Reflectance Equation

If you want to model the reflection and refraction of light, you need something called the reflectance equation, which looks like this:

$$L_o(p,\omega_o) = \int_\Omega f_r(p,\omega_i,\omega_o) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

DON'T GET SCARED, WE ARE GOING TO COMPLETELY ELIMINATE THAT INTEGRAL IN ONE STEP

(If you do want to scare yourself, this is a specialized and simplified form of the render equation, https://en.wikipedia.org/wiki/Rendering_equation).

This equation tries to tell us the intensity of the light leaving a surface through a hemisphere Ω, given:

ω_i = direction to incoming light
ω_o = direction of the outgoing light
p = position in space
n = surface normal

Luckily, we only need to worry about the integral when our light sources are not point lights (you have to integrate the light coming in across the area of the surface the light is leaving from for non-point lights). If you assume just point lights, it becomes a sum over each point light where ω_i is direction to the current light. If we consider only a single point light, we don't even need to have the sum, we just have the equation:

$$L_0(p,\omega_o) = f_r(p,\omega_i,\omega_o) L_i(p,\omega_i) n \cdot \omega_i$$

Let's tackle the easiest part first (THIS IS WHERE YOU NEED TO START CODING YOUR FRAGMENT SHADER): $n \cdot \omega_i$

The provided starter code provides an UNNORMALIZED vector to the camera and to the light and assumes you are not rotating the model.

Next we do the middle term:

L_i is very simple, that is just the intensity of light i at point p, which can be trivially determined using the inverse square law:

$$L_i(p, position_i) = \frac{intensity_i}{(p - position_i)^2} \text{ where}$$

$p = \text{current point (fragment) position}$
$position_i = \text{current light position}$
$intensity_i = \text{current light intensity}$

The starter code passes in an un-normalized vector to the light, and this is the same for every vertex since we are modeling a directional light rather than a point light, so you should use that as the distance. I used a light intensity of 30 and that seemed to work well, if your model is all white or all black, try varying this before thinking you made a mistake.

Finally we get to that f_r equation. This is known as the Bidirectional Reflective Distribution Function. We are using the *Cook-Torrance* approximation for the BRDF:

$$f_r = \frac{k_d * albedo}{\pi} + k_s f_{cook-torrance-specular}$$

The value for albedo should be provided for you already if you are using my starter code, otherwise it is just the color of the diffuse texture at that point.

Cook and Torrance were kind enough to also give us an approximation for the specular part of the *Cook-Torrance BRDF:*

$$f_{cook-torrance-specular} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

Now things are getting fun. The denominator is fairly self explanatory, just make sure to normalize all your vectors before dotting them. For ALL equations with fractions, ensure to bound the denominators below at like 0.0001 so that you don't divide by zero. I found that I had to do $-1 * \omega_i$ to get it to look right, if your model is entirely black on the shaded side, the denominator **might** be the problem.
The numerator has three parts, **D, F,** and **G,** which we will cover in the next few sections.

D: This is the Normal Distribution Function, which does the math that actually approximates the microfacets given surface roughness. We will use the *Trowbridge-Reitz GGX* approximation:

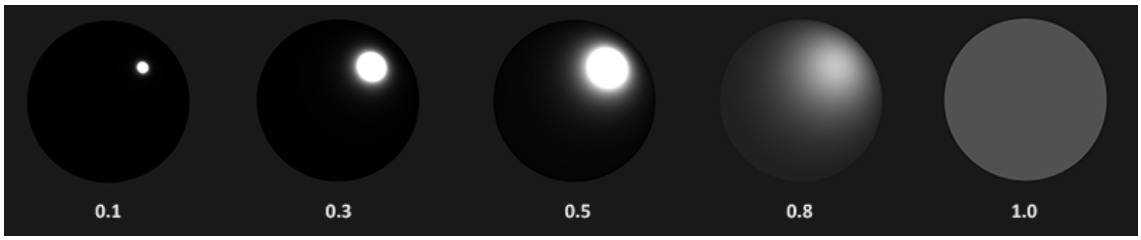
$$NDF_{TR-GGX}(n, h, roughness) = \frac{roughness^2}{\pi((n \cdot h)^2(roughness^2 - 1) + 1)^2}$$

The only term you don't have immediately available here is h, which is the halfway vector:

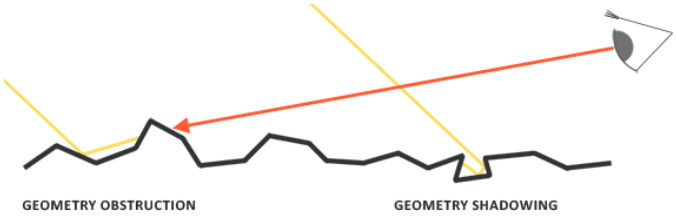
$$h = \frac{l + v}{\|l + v\|}$$

It is just a vector halfway between the light and view vector, very useful for speeding up certain lighting computations. The function stub for this should already be prepared, so just fill it out and you should be in business.

This is the equation that generates the chart from earlier:



We have a problem now, notice how in the above the perfectly rough surface is still illuminated even where it is in shadows. This is because we have no model of the microsurfaces getting in the way of each other when reflecting light. Enter…
G: The Geometry function.



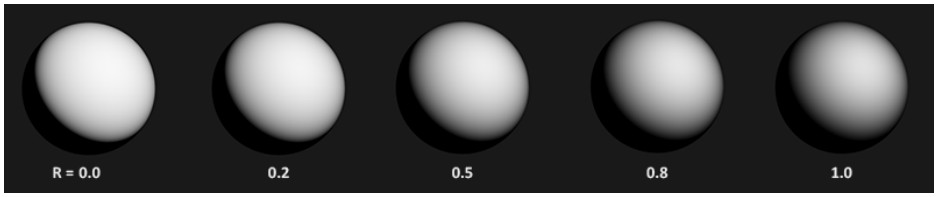
We need a statistical model of microsurfaces blocking each other from reflecting light back at the camera, and this function will provide it. We use the *Schlick-GGX* approximation:

$$G_{Schlick-GGX}(n, v, roughness) = \frac{n \cdot v}{(n \cdot v)(1 - \frac{(roughness+1)^2}{8}) + \frac{(roughness+1)^2}{8}}$$

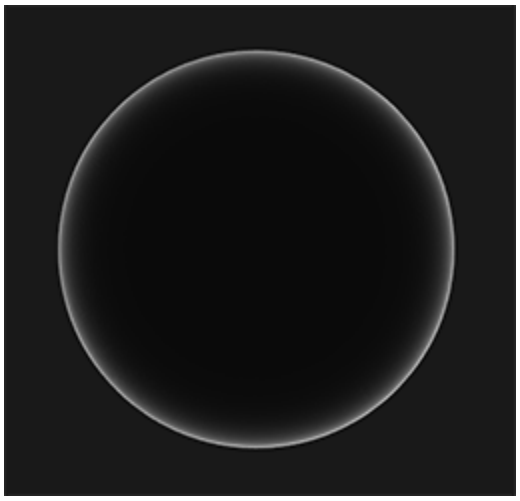
Where v is a vector to the camera. This works to model geometry obstruction, but we still need to model geometry shadowing. Remember to bound the denominator below at some small value (I used 0.0001) so that you don't end up dividing by 0. Luckily we can use the same equation again but replace the vector to the camera with the vector towards the light. If we do both of these and multiply them, we get a more accurate model of the surface geometry obstructing the light. This is known as *Smith's Method*:

$$G_{smiths-method} = (n, v, l, roughness) = G_{Schlick-GGX}(n, v, roughness) * G_{Schlick-GGX}(n, l, roughness)$$

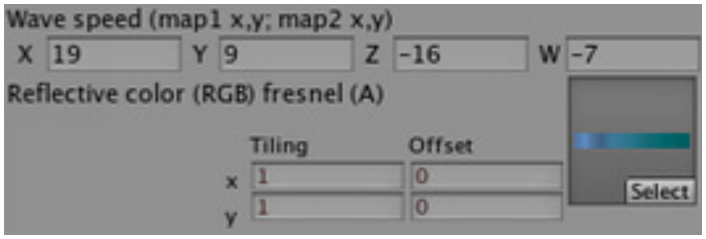
Again, these all have function stubs, so just fill the function contents out and you should be good to go (*Smith's Method* becomes **G** in your *Cook-Torrance* equation). This equation models how shadows are harder or softer against a surface depending on how rough it is:



Finally we have…
F: The *Fresnel* equation.
As the angle between the camera and a surface approaches 90 degrees, the surface becomes more and more reflective until it is perfectly reflective at 90 degrees *regardless of roughness*. This is modeled using the Fresnel equation. In the past (up through 2013ish by my own recollection), artists would provide a texture describing the Fresnel effect for each somewhat reflective material (particularly for water where the changing color of waves depending on height was modeled using the Fresnel effect)



Sphere with all lighting other than the Fresnel effect disabled



The preset Fresnel color for water in Unity3D version 3

About a decade ago, it was determined that the Fresnel effect for all dielectric materials was more or less the same and could be easily computed using the *Fresnel-Schlick* approximation:

$$F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

F_0 is typically computed using a surface's index of refraction, but since metals don't really refract you can't compute it like that. Instead people get really cheap and compute F_0 differently depending on the metallicity: for very metallic surfaces, they replace the pixel values in the albedo texture with the F_0value since metals don't really have albedo values (you can see this in a lot of PBR metal materials that have very strange colors in the albedo texture, it's because they are feeding in the F_0 via the albedo texture). For dielectric materials, $F_0$is typically really low, like 0.04-0.08. To implement this in code, lerp between (0.04, 0.04, 0.04) and (albedo_r, albedo_g, albedo_b) based on the metallic value (such that metallic=0 will return (0.4,0.4,0.4), and metallic=1 will return the albedo) and use that as F_0.

Material	F_0 (Linear)	F_0 (sRGB)	Color
Water	(0.02, 0.02, 0.02)	(0.15, 0.15, 0.15)	
Plastic / Glass (Low)	(0.03, 0.03, 0.03)	(0.21, 0.21, 0.21)	
Plastic High	(0.05, 0.05, 0.05)	(0.24, 0.24, 0.24)	
Glass (high) / Ruby	(0.08, 0.08, 0.08)	(0.31, 0.31, 0.31)	
Diamond	(0.17, 0.17, 0.17)	(0.45, 0.45, 0.45)	
Iron	(0.56, 0.57, 0.58)	(0.77, 0.78, 0.78)	
Copper	(0.95, 0.64, 0.54)	(0.98, 0.82, 0.76)	
Gold	(1.00, 0.71, 0.29)	(1.00, 0.86, 0.57)	
Aluminium	(0.91, 0.92, 0.92)	(0.96, 0.96, 0.97)	
Silver	(0.95, 0.93, 0.88)	(0.98, 0.97, 0.95)	

Table of precomputed F_0 values

At this point you should have all the parts of your *Cook-Torrance* specular equation finished, and thus your overall *Cook-Torrance BRDF* equation is finished. You aren't *quite* done yet as you still haven't determined k_s and k_d, luckily k_s can just be set to the output from the Fresnel approximation, and k_d is just $1 - k_s$. Remember that metallic objects don't refract light thus they don't have diffuse components, so you also need to multiply k_d by $1 - metallic$

Finally, you need to do some tone mapping as currently there is no upper limit on the value your *Cook-Torrance BRDF* produces, this is easily solved by using the abasic tone mapping equation

```
color = color / (color + vec3(1.0));
```

which will bound your output between 0 and 1 (`color` is just the output of your BRDF), and you might be inclined to gamma correct your output (we are working in a linear color space, but they eye sees light differently, so we map the colors in to eyeball color space):

```
color = pow(color, vec3(1.0/2.2));
```

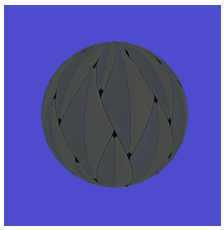
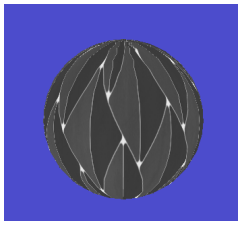
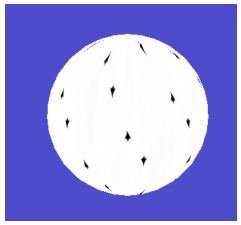
You should have a fairly realistic looking (though somewhat flat surface). I have provided a number of materials to check out, and you can find more at https://ambientcg.com/ if you want to play around more.

Concrete textures I provided mapped on to a ball with normal mapping

Your objects will not look super realistic yet because they have nothing to reflect (you need cube mapping for that), and they aren't very bumpy (you need normal mapping for that). Normal mapping can be implemented using only a shader, if you google around for a bit you might be able to find some code to copy paste in so you can get the full effect (it should perturb the view vector at the beginning of your fragment shader and then propagate that through the rest of the shader). I have provided normal maps for you so you can do this if you want.

TODO LIST:
(images are generated by setting the fragment color to the output of their respective functions, using MetalPlates006_1K_*.png as texture inputs)

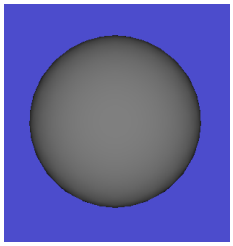
1. **GET TEXTURE LOADING WORKING**
   (ctrl-f "TODO" in pbr.cxx and fill all of them in, they should look like this when drawn on the sphere with no shading, check the note in "uint8_t* load_images(...)" to see how to handle grayscale inputs)

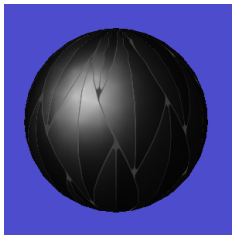
Albedo *Roughness* *Metallic*

2. **Set up LightIntensity**
3. **Set up HalfwayVec**

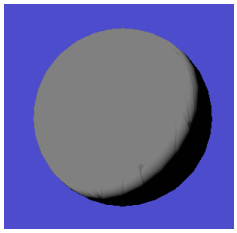
4. **Compute NdotL**



5. **Compute D (TrowbridgeReitzGGX)**



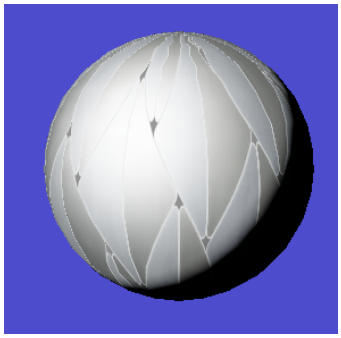
6. **Set up GeometrySchlickGGX**
7. **Compute G (Smith's Method)**



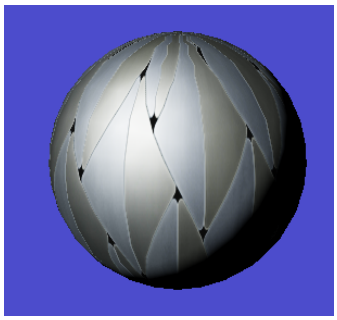
8. **Set up lerp between F0 and Albedo**

9. **Compute F (Fresnel Schlick)**

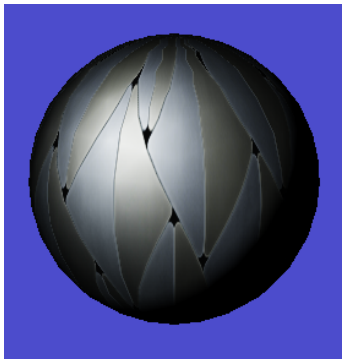
10. **Compute the full specular (DGF/(4*CdotN*NdotL))**



11. Compute full BDRF (CookTorrance)



12. Compute full reflectance equation



DONE