Assignment 10 – Gamma & PBR

Overview

The goal of this assignment is to improve the lighting in your engine. To do this, you'll implement two features: gamma correction and a physically-based specular BRDF.

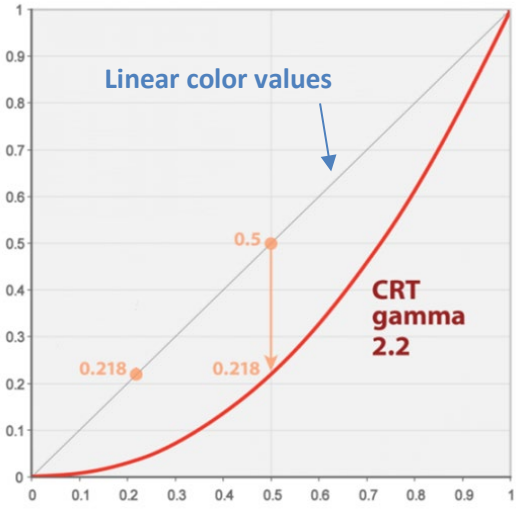
Task Overview

Here is the high-level overview of tasks, which are explained in more detail on the pages that follow:

- ❑ Update your pixel shaders to handle **gamma correction**
- ❑ Swap out the Phong specular calculations for the more physically-accurate **Cook-Torrence BRDF**
- ❑ Use some of the provided **PBR textures**, or find your own online
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

Task 1: Gamma Correction

While our lighting calculations are fine, the results we've been seeing on our screens are not accurate. As discussed in class, this is due to the way digital devices display colors. Our math is linear, meaning a value that should be twice as bright results a doubling of each color channel: (0.1, 0.1, 0.1) * 2 == (0.2, 0.2, 0.2). However, digital displays end up being darker than expected, meaning a color that *should* appear twice as bright does not. That discrepancy, when graphed, produces a curve known as a gamma curve. See the image to the right for an example of this curve.



If we want our results to not only be mathematically correct but also appear correct on the screen, we need to adjust our final pixel shader output to account for the darkening of colors by our monitors.

Additionally, since most digital images have had gamma correct already applied, we'll need to un-gamma-correct any texture data that represents surface colors before using it in any calculations.

> *If you're interested in reading more about gamma correction, check out The Importance of Being Linear from GPU Gems 3 and Linear Space Lighting from filmicworlds.com.*

Step 1: Gamma Correction

You'll need to complete the following step for any pixel shader that is performing a lighting calculation. You may have two shaders which fit that description at the moment: one that handles objects with normal maps and one for objects without normal maps.

> *Note that you do **not** need to do any gamma correction in your skybox pixel shader, as the image is already gamma corrected; you're simply sampling that texture and returning it.*

For each of the aforementioned light-calculating pixel shaders, find the line that returns the final pixel color (presumably the last line in each shader's main() function). Adjust the color (RGB) portion of the return value by raising it to the 1.0/2.2 power.

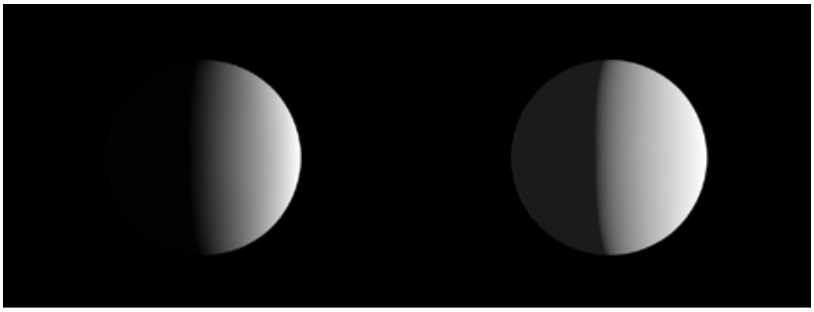
For example, let's say the final line in each pixel shader looks like this:

```
return float4(totalColor, 1);
```

To apply gamma correction, they should now look similar to this:

```
return float4(pow(totalColor, 1.0f / 2.2f), 1);
```

That's pretty much it for step 1. See the image below for examples of a sphere with a single, white directional light shining on it.



Diffuse lighting **without** gamma correction

Diffuse lighting **with** gamma correction

Step 2: Gamma & Textures

Since most digital images (those that represent photographs or surface colors) are already gamma corrected, using colors straight from these textures as part of our lighting calculations will also produce incorrect results. Our math is assuming surface colors are linear, but the colors we're sampling from our textures are not. To account for this, you'll need to un-correct the colors you sample from any texture that represents a *surface color* (like your diffuse texture), essentially re-applying the gamma curve to get it the colors back into linear space

> *Note: you should NOT do anything to material textures such as normal maps or specular maps.*

In the same shaders you edited in step 1, find the lines where you sample the *surface texture*. Raise the color (RGB) portion of that result to the 2.2 power.

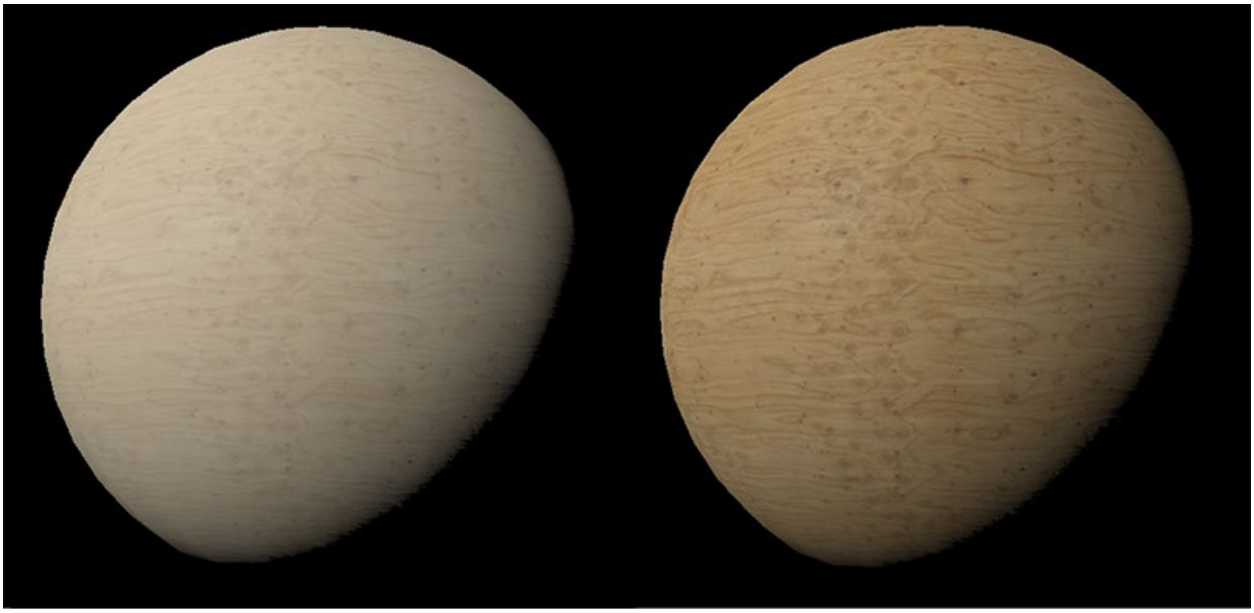
For example, if your texture sample code looks like this:

```
float3 color = surfaceTexture.Sample(sampOptions, input.uv).rgb;
```

Your "un-corrected" version might look like this:

```
float3 color = pow( surfaceTexture.Sample(sampOptions, input.uv).rgb, 2.2f );
```

Step 2 complete! Remember: do NOT apply this "un-correction" to normal maps, specular maps, etc. See the image below for before and after examples of this step.



Texture sample has **not** been properly "uncorrected"

Texture sample **has** been properly "uncorrected"

Texture used in both examples above

Task 2: Physically-Based Rendering

Your second major task is to implement physically-based rendering (also known as physically-based shading, as we're mostly just swapping out some of our lighting equations for more-sophisticated versions). While we discussed this math at a high level in class, you do not need to derive or convert it to code yourself; I'm providing the exact functions you'll need to use in your code at the end of this document, in an external text file on MyCourses and within the associated PBR demo shown in class.

Since the specific lighting functions are being provided, your main goals here are to decide how many lighting models you want to support, integrate the PBR lighting functions into your existing shaders, and choose which PBR assets you want to use. You'll also need to remove your *ambient term entirely*! Let's break each of those steps down.

Step 1: Classic Lighting & PBR? Or PBR only?

The first decision you need to make is whether you want to support the "old school" lighting we've been doing so far *in addition* to PBR, or just convert all of your shaders to PBR only.

It's totally fine if you'd like to simply *replace* your existing lighting with PBR. Unless you have a specific reason for handling both lighting models, I'd suggest going this route. This means that, going forward, all of your texture assets will need to be PBR-appropriate. I'll be providing some for you on MyCourses along with this assignment. These types of textures can also be found for free in a lot of places across the internet (I'll have some links later in this document).

If, instead, you'd like your engine to be able to handle both PBR and non-PBR materials, that's fine, too. It will, however, be a little bit more work, as you will need both a PBR and non-PBR version of each light-calculating pixel shader (and the ability to represent both types of materials in C++). If you currently have a pixel shader that handles normal maps and one that doesn't, this means you'd need two versions of *each* of those pixel shaders. Note that this setup is ***not** a requirement* for this assignment.

It is also ok if you'd like to assume, going forward, that every material will also have a normal map. This can also cut down on the number of shader permutations you need to handle.

> *When would it make sense to handle multiple lighting models?*
>
> *If you were in charge of making a re-usable game engine that needs to work across multiple platforms and devices, it might make sense to handle many different lighting models. However, when working on a specific game, generally speaking, you'd choose one lighting model and stick with it; there's no need to handle every option if you won't be using them. Your artists will thank you, too!*
>
> *In other words: It's ok to just build a game; you don't need to recreate Unity or Unreal first.*

Step 2: Implementing PBR for Direct Lighting

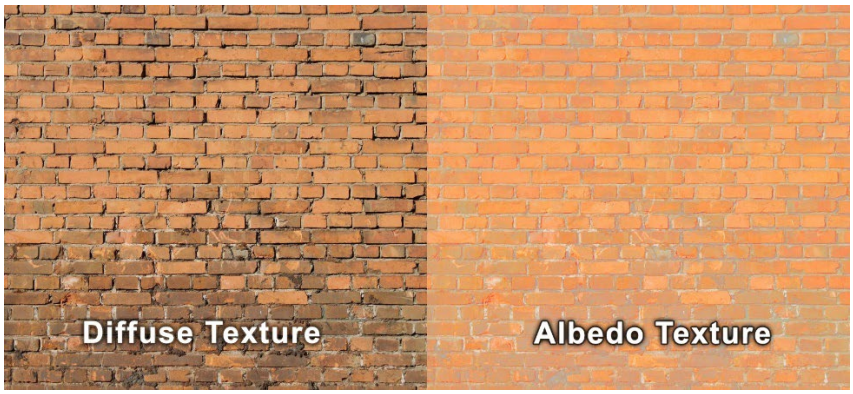
Implementing PBR means replacing some of our lighting equations with more-sophisticated versions. Recall that PBR still uses Lambert for its diffuse component. That code can remain exactly the same in your shaders. What primarily changes at this point is the specular portion: you'll be replacing the Phong BRDF with the Cook-Torrence BRDF. Once that is done, the last step is to ensure the diffuse and specular portions are balanced to properly conserve energy: as more light is reflecting, there is less light diffusing.

At this point, I'd suggest **reviewing** the PBR constants and functions at the end of this document and **adding** the functions to your shader include file(s). You are free to use the functions as-is. While neither necessary nor expected, you could adjust the function parameters to better suit your implementations and/or optimize the functions if you'd like.

PBR Textures

PBR requires several textures worth of data for each material: the surface color (which we call *albedo* rather than diffuse), the *roughness* of each pixel and the *metalness* of each pixel. While not explicitly required, almost all PBR materials will also have a *normal map*. Ensure that your shader(s) define a separate texture for each of these.

> *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 shaders, and will be affected by the lighting in our scenes. See the examples below.*



Example texture & sampler setup (feel free to adjust the names):

```
Texture2D Albedo            : register(t0);
Texture2D NormalMap         : register(t1);
Texture2D RoughnessMap      : register(t2);
Texture2D MetalnessMap      : register(t3);
SamplerState BasicSampler   : register(s0);
```

Sampling Textures in Preparation for PBR

Each texture holds a different piece of the PBR puzzle, and some values (such as metalness) will influence the usage of others. See below for details on exactly how to handle each texture. Note that some (like albedo and normal maps) will match what you're already doing.

Albedo

No changes required for PBR (aside from potentially updating variable names). This is effectively surface color. Sample this as you normally would sample the diffuse texture. Don't forget to gamma correct!

Normal Map

No changes required for PBR. Sample, unpack, apply the TBN matrix, etc. No gamma correction.

Roughness Map

Roughness is a single float value, so just grab the red channel after sampling the roughness texture. (This texture it usually greyscale, meaning all 3 color channels hold the same value.) No gamma correction.

```
float roughness = RoughnessMap.Sample(SamplerOptions, input.uv).r;
```

Metalness Map

Metalness is also a single float value, so just grab the red channel again. No gamma correction.

```
float metalness = MetalnessMap.Sample(SamplerOptions, input.uv).r;
```

Specular Color

We'll need a *specular color* for each pixel, though we don't have a separate texture for this. Metals tint their reflections (imagine shiny surfaces of gold vs. iron vs. copper), meaning any metal surface needs a specific specular color, ideally measured from the real world. Non-metals, however, generally have an approximately equal specular color (0.04). In other words: sometimes we'll need a specular color and sometimes we'll just use a constant. We make that decision based on the "metalness" of the pixel.

In terms of *albedo*, non-metals use this as their surface color while metals have an effective albedo of 0.

What does this all mean for us? It means we can interpret a color stored in the albedo texture as either a true albedo (for non-metals) or a specular color (for metals) based on the pixel's metalness.

A note on this: metalness will generally be either 0 or 1, since a surface is either metal or it isn't. However, since we're most likely using linear filtering when sampling, there's a chance that we'll sometimes get a value *between* 0 and 1. This means that, instead of an either/or statement, we'll need to interpolate the specular color ourselves using metalness as the interpolation value.

```
// Specular color determination -----------------
// Assume albedo texture is actually holding specular color where metalness == 1
// Note the use of lerp here - metal is generally 0 or 1, but might be in between
// because of linear texture sampling, so we lerp the specular color to match
float3 specularColor = lerp(F0_NON_METAL, albedoColor.rgb, metalness);
```

Replacing Phong with Cook-Torrence

The next major change is to replace your Phong calculation with the Cook-Torrence BRDF. This is provided as a function you can call, as shown below. If your current specular result variable is a single float, you'll need to **change it to a float3** since the new result will be tinted when dealing with metals.

```
float3 MicrofacetBRDF(float3 n, float3 l, float3 v, float roughness, float3 f0, out float3 F_out)
```

By this point in your shader, you should have all of the parameters necessary for this function:

- `n`: the normal (after normal mapping)
- `l`: the light vector (normalized direction to the light)
- `v`: the view vector (normalized direction to the camera)
- `roughness`: from the roughness map
- `f0`: specular color for this pixel (calculated in previous step above)
- `F_out`: this is an extra value being returned (an *out* param), so no specific value coming in

The result of this function is your new specular value. The F_out is the Fresnel result, which we'll also need outside the function to properly conserve energy.

Conserving Energy & Combine

The final change in your shader is to properly conserve energy between diffusion and reflection (specularity). A function, called DiffuseEnergyConserve(), has been provided for this. After calculating diffuse and specular, call this function to adjust diffusion based on how much light is reflecting just before your final combine for the current light.

```
// Calculate the light amounts
float diff = DiffusePBR(normal, toLight);
float3 F;
float3 spec = MicrofacetBRDF(normal, toLight, toCam, roughness, specColor, F);

// Calculate diffuse with energy conservation, including cutting diffuse for metals
float3 balancedDiff = DiffuseEnergyConserve(diff, F, metalness);

// Combine the final diffuse and specular values for this light
float3 total = (balancedDiff * surfaceColor + spec) * light.Intensity * light.Color;
```

What about Ambient?

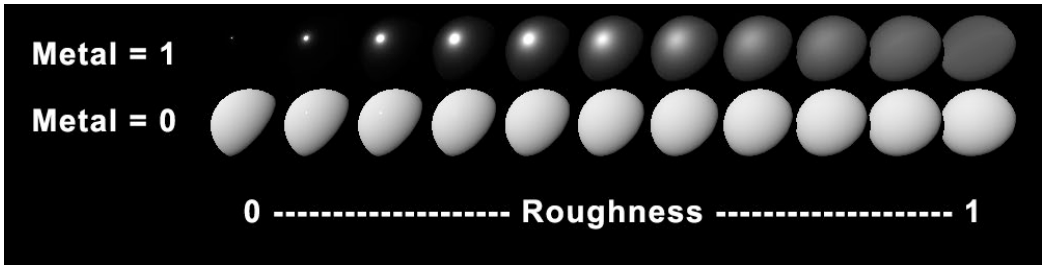
Since our extremely simple ambient approximation isn't physically-based, it ends up clashing with our PBR results. You should **remove it** from your PBR shaders or, at the very least, set your ambient color to black to remove its influence on any PBR-shaded objects.

Testing PBR

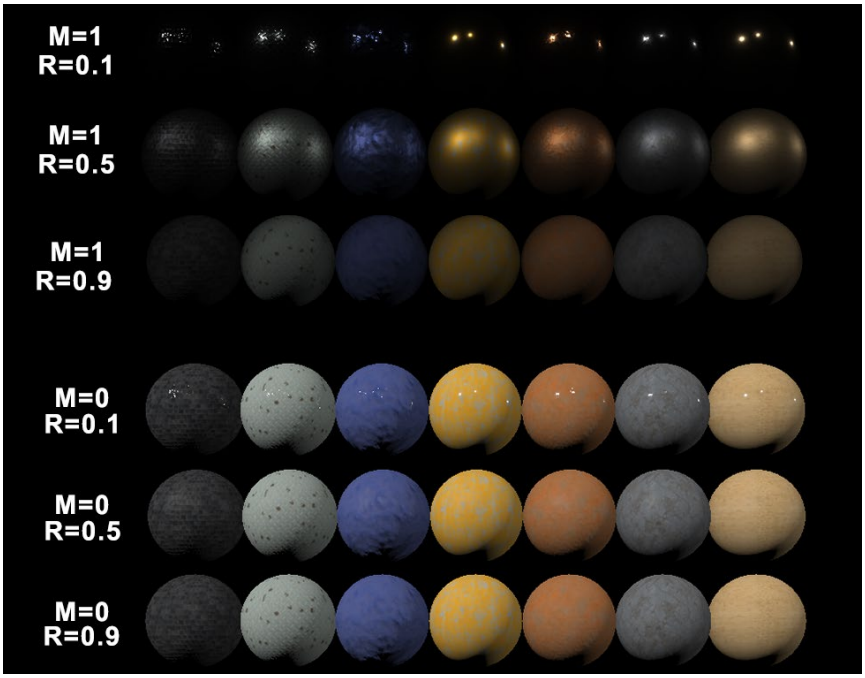
That's about it for the changes to your shaders. However, you probably don't have the required textures being loaded or bound to the pipeline to actually test this out quite yet.

If you'd like to test things out before moving on, I'd suggest temporarily commenting out the textures for roughness and metalness and replacing them with constants. The exact values are up to you, but roughness should be somewhere between `MIN_ROUGHNESS` and 1, and metalness should be either 0 or 1. We use a non-zero amount for the minimum roughness since our light sources are infinitely small, and a perfect reflection of an infinitely small point is infinitely small.

Below are examples of PBR surfaces with constant metalness and roughness values (instead of textures).



The surface color of the above spheres is white (1,1,1)



Task 3: PBR Assets

You'll find a zip file with several sets of PBR assets on MyCourses. These match the PBR materials present in my PBR demo. Feel free to use these textures for this assignment.

If, however, you'd like to see what else is out there, below are some places to find free PBR materials.

> *Note that many of these sites forego metalness textures if the material is either completely metal or completely non-metal. Your engine, however, still needs a texture to hold this information. The easiest solution is to create and use your own images that are 100% black or 100% white as your metalness textures. They can be any size, though smaller is more efficient. Since they're the same color throughout, they can even be as small as 2x2.*

For any particular material, you'll need the albedo, normal, roughness and metal maps. The others, like height and ambient occlusion, aren't useful to us right now.

Textures.com

This site has a whole PBR section, though it requires a free account registration. Once registered, you're given 15 "credits" per day for downloads.

I'd suggest grabbing the 1024x1024 versions of the textures, as they're 1 "credit" each for free accounts. Larger sizes are available for premium users.

Ambient CG

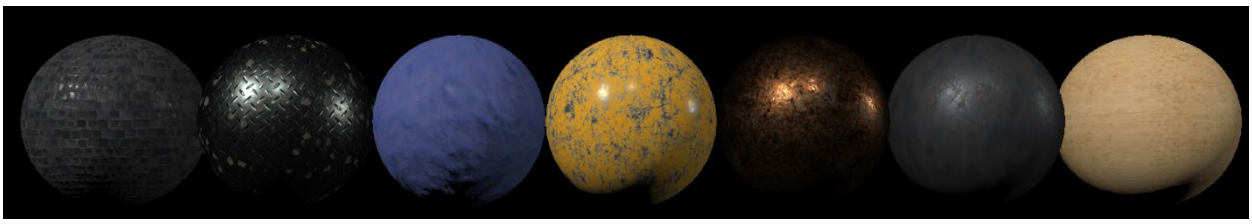
A great site dedicated to creative commons PBR materials which has hundreds of free textures.

Poly Haven

Another site with some great materials. The selection isn't huge, but what's there is quality.

Putting it All Together

You should have something that looks like the following (potentially with different meshes & materials).



Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. Remember to follow the steps in the "Preparing a Visual Studio project for Upload" PDF on MyCourses to shrink the file size.

Also, please **only include assets you're currently using**! Unused textures/skyboxes take up a lot of room.

Seriously, this is important! The DirectX Toolkit can really bloat your project size, so be sure to remove the files as outline in the PDF mentioned above; it will be redownloaded from NuGet automatically the next time you open and build your project.

Appendix: PBR Constants and Functions

You can either translate this code into your own project manually or copy/paste it. As this is a PDF, copy/pasting is a bit flaky, so I have also included a .txt file on MyCourses with this code. I highly recommend putting these into an external .hlsli file and including that in any shader that will be doing PBR. Note that there are several constants below that you'll also need to incorporate into your shaders (or the aforementioned include file).

Constants

```
// Make sure to place these at the top of your shader(s) or shader include file
// - You don't necessarily have to keep all the comments; they're here for your reference

// The fresnel value for non-metals (dielectrics)
// Page 9: "F0 of nonmetals is now a constant 0.04"
// http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
static const float F0_NON_METAL = 0.04f;

// Minimum roughness for when spec distribution function denominator goes to zero
static const float MIN_ROUGHNESS = 0.0000001f; // 6 zeros after decimal

// Handy to have this as a constant
static const float PI = 3.14159265359f;
```

PBR Functions

```
// Lambert diffuse BRDF - Same as the basic lighting diffuse calculation!
// - NOTE: this function assumes the vectors are already NORMALIZED!
float DiffusePBR(float3 normal, float3 dirToLight)
{
	return saturate(dot(normal, dirToLight));
}


// Calculates diffuse amount based on energy conservation
//
// diffuse   - Diffuse amount
// F         - Fresnel result from microfacet BRDF
// metalness - surface metalness amount
float3 DiffuseEnergyConserve(float3 diffuse, float3 F, float metalness)
{
	return diffuse * (1 - F) * (1 - metalness);
}
```

```
// Normal Distribution Function: GGX (Trowbridge-Reitz)
//
// a - Roughness
// h - Half vector: (V + L)/2
// n - Normal
//
// D(h, n, a) = a^2 / pi * ((n dot h)^2 * (a^2 - 1) + 1)^2
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;
	float a2 = max(a * a, MIN_ROUGHNESS); // Applied after remap!

	// ((n dot h)^2 * (a^2 - 1) + 1)
	// Can go to zero if roughness is 0 and NdotH is 1; MIN_ROUGHNESS helps here
	float denomToSquare = NdotH2 * (a2 - 1) + 1;

	// Final value
	return a2 / (PI * denomToSquare * denomToSquare);
}



// Fresnel term - Schlick approx.
//
// v - View vector
// h - Half vector
// f0 - Value when l = n
//
// F(v,h,f0) = f0 + (1-f0)(1 - (v dot h))^5
float3 F_Schlick(float3 v, float3 h, float3 f0)
{
	// Pre-calculations
	float VdotH = saturate(dot(v, h));

	// Final value
	return f0 + (1 - f0) * pow(1 - VdotH, 5);
}


// Geometric Shadowing - Schlick-GGX
// - k is remapped to a / 2, roughness remapped to (r+1)/2 before squaring!
//
// n - Normal
// v - View vector
//
// G_Schlick(n,v,a) = (n dot v) / ((n dot v) * (1 - k) * k)
//
// Full G(n,v,l,a) term = G_SchlickGGX(n,v,a) * G_SchlickGGX(n,l,a)
float G_SchlickGGX(float3 n, float3 v, float roughness)
{
	// End result of remapping:
	float k = pow(roughness + 1, 2) / 8.0f;
	float NdotV = saturate(dot(n, v));

	// Final value
	// Note: Numerator should be NdotV (or NdotL depending on parameters).
	// However, these are also in the BRDF's denominator, so they'll cancel!
	// We're leaving them out here AND in the BRDF function as the
	// dot products can get VERY small and cause rounding errors.
	return 1 / (NdotV * (1 - k) + k);
}
```

```
// Cook-Torrance Microfacet BRDF (Specular)
//
// f(l,v) = D(h)F(v,h)G(l,v,h) / 4(n dot l)(n dot v)
// - parts of the denominator are canceled out by numerator (see below)
//
// D() - Normal Distribution Function - Trowbridge-Reitz (GGX)
// F() - Fresnel - Schlick approx
// G() - Geometric Shadowing - Schlick-GGX
float3 MicrofacetBRDF(float3 n, float3 l, float3 v, float roughness, float3 specColor, float3 out F_out)
{
        // Other vectors
        float3 h = normalize(v + l); // That’s an L, not a 1! Careful copy/pasting from a PDF!

        // Run numerator functions
        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);

        // Pass F out of the function for diffuse balance
        F_out = F;

        // Final specular formula
        // Note: The denominator SHOULD contain (NdotV)(NdotL), but they'd be
        // canceled out by our G() term.  As such, they have been removed
        // from BOTH places to prevent floating point rounding errors.
        float3 specularResult = (D * F * G) / 4;

        // One last non-obvious requirement: According to the rendering equation,
        // specular must have the same NdotL applied as diffuse!  We'll apply
        // that here so that minimal changes are required elsewhere.
        return specularResult * max(dot(n, l), 0);
}
```