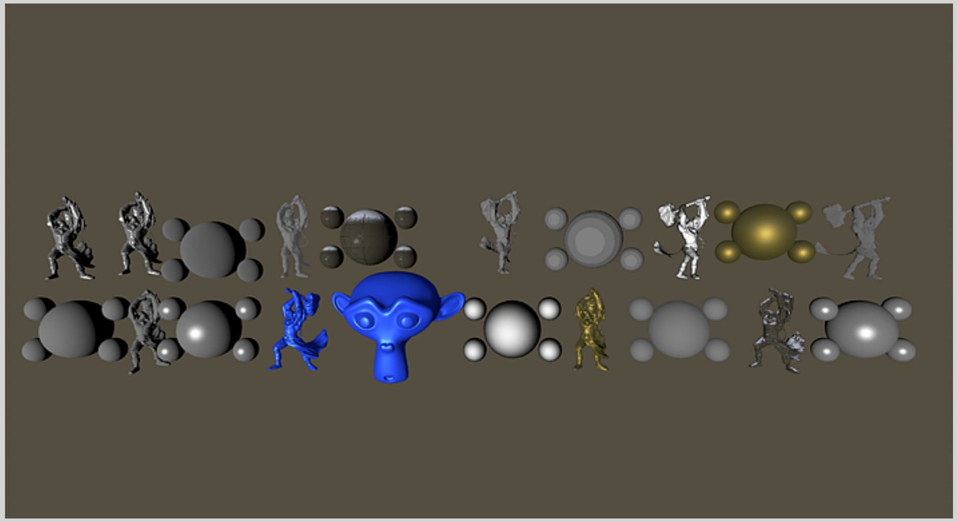


Lighting Models are methods/algorithms used to determine the light at the surface of a model. There are many varying methods of computing lighting with shaders. Artists and programmers choose them depending on the level of realism/stylization/performance implications of the model. Below we will cover a range of lighting and shading models.

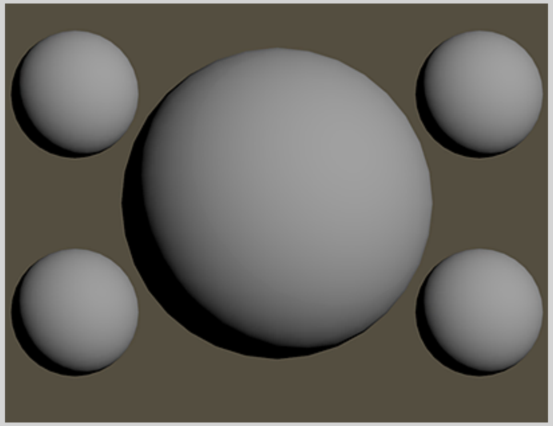
Lambert

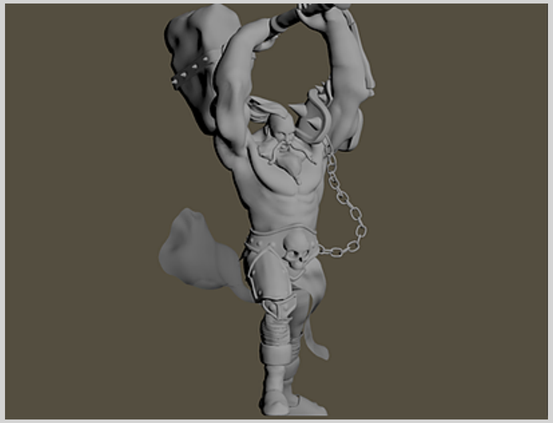
Lambertian Reflectance, or Diffuse Light, is one of the most common, if not the most common, lighting models used.

This lighting model is view independent, meaning the surface's apparent brightness should not change based on viewpoint. It is a very simple model and can be constucted in pseudocode as follows:

```
float3 SurfaceColor;          //objects color
float3 LightColor;            //lights color * intensity
float LightAttenuation;       //value of light at point (shadow/falloff)

float NdotL = max(0.0,dot(normalDirection,lightDirection));
float LambertDiffuse = NdotL * SurfaceColor;
float3 finalColor = LambertDiffuse * LightAttenuation * LightColor;
```





Half-Lambert (Diffuse Wrap)

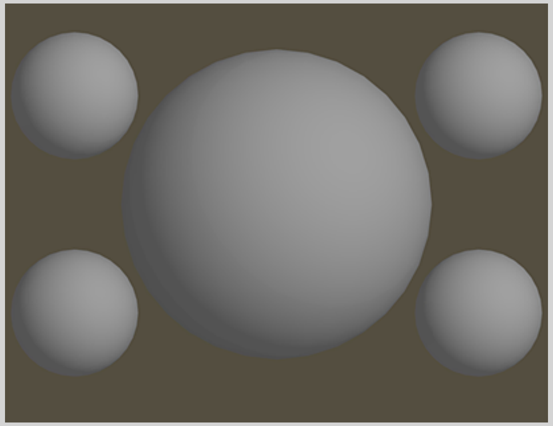
The Half-Lambert lighting model is a technique first developed in the original Half-Life by Valve. It was developed to prevent the rear of an object losing it's shape and thereby looking flat. This is an extremely forgiving lighting model, and as such is completely non-physical.

The key to the Half-Lambert lighting model is that the Lambert diffuse is halved, then half is added, then it is squared. A common implementation, called diffuse wrap, operates on the idea that instead of using halves, you can use any value between a half and a whole. ie. Instead of:

pow(NdotL * 0.5 + 0.5,2)

One could use:

pow(NdotL * wrapValue + (1-wrapValue),2)



Where wrap value is anything between 0.5 and 1.

Below is an implementation of the Half-Lambert lighting Model:

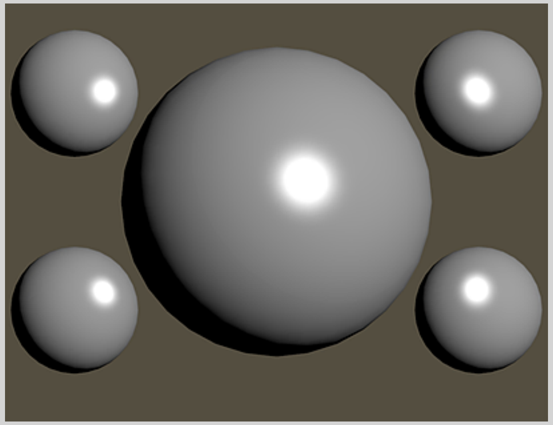
```
float3 SurfaceColor;         //objects color
float3 LightColor;           //lights color * intensity
float LightAttenuation;      //value of light at point (shadow/falloff)

float NdotL = max(0.0,dot(normalDirection,lightDirection));
float HalfLambertDiffuse = pow(NdotL * 0.5 + 0.5,2.0) * SurfaceColor;
float3 finalColor = HalfLambertDiffuse * LightAttenuation * LightColor;
```

Phong Lighting

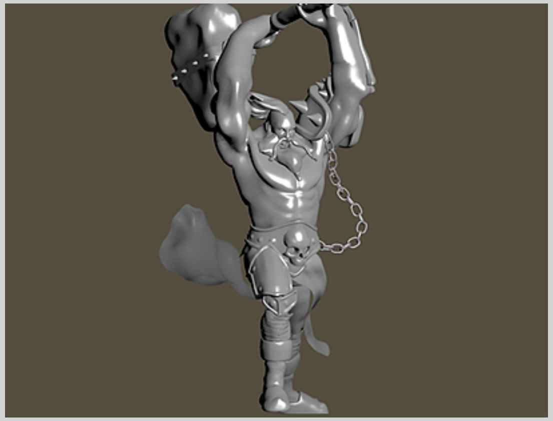
The Phong Shading Model is commonly used to create specular effects. It is based on the assumption that the way a surface reflects light is a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces.

Below is a common implementation of the Phong Model:

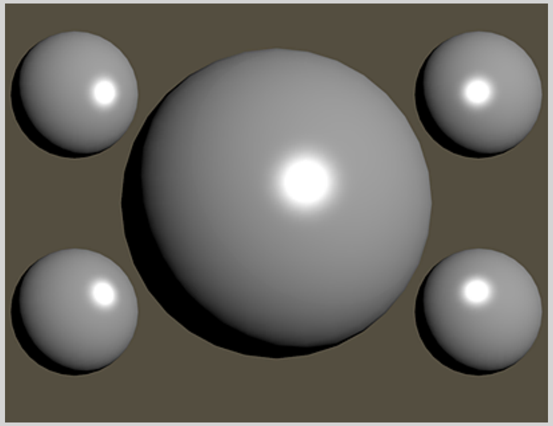


```
float3 viewDirection = normalize(_WorldSpaceCameraPos.xyz - i.posWorld.xyz);
float3 lightReflectDirection = reflect( -lightDirection, normalDirection );
float NdotL = max(0, dot( normalDirection, lightDirection ));
float RdotV = max(0, dot( lightReflectDirection, viewDirection ));
//Specular calculations
float3 specularity = pow(RdotV,_SpecularGloss/4)*_SpecularPower
*_SpecularColor.rgb ;

float3 lightingModel = NdotL * diffuseColor + specularity;
float attenuation = LIGHT_ATTENUATION(i);
float3 attenColor = attenuation * _LightColor0.rgb;
float4 finalDiffuse = float4(lightingModel* attenColor,1);
```



Blinn-Phong Lighting



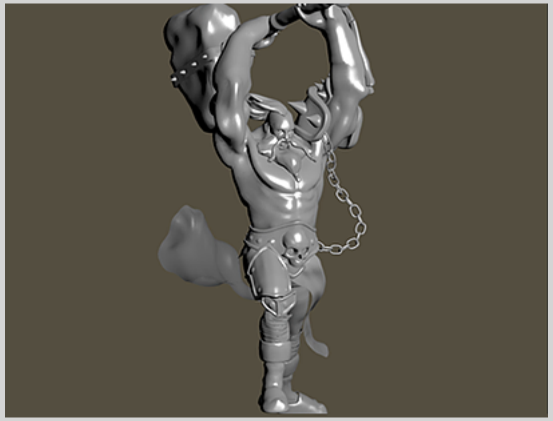
The Blinn-Phong shading model (also called the modified Phong) is a modification to the Phong reflection model, and was developed by Jim Blinn.

The Blinn-Phong model is an optimization to the Phong model, which requires one to calculate the light reflection vector every frame. Instead, Blinn calculates what is referred to as the "Half Direction" which is simply the halfway angle between the light direction and the view direction. The benefit to this is a much smoother specular reflectance shading model.

Below is a common implementation of the Blinn-Phong Lighting Model:

```
float3 viewDirection = normalize(_WorldSpaceCameraPos.xyz - i.posWorld.xyz);
float3 halfDirection = normalize(viewDirection+lightDirection);
float NdotL = max(0, dot( normalDirection, lightDirection ));
float NdotV = max(0, dot( normalDirection, halfDirection ));
//Specular calculations
float3 specularity = pow(NdotV ,_SpecularGloss)*_SpecularPower
 *_SpecularColor.rgb ;
float3 lightingModel = NdotL * diffuseColor + specularity;

float attenuation = LIGHT_ATTENUATION(i);
float3 attenColor = attenuation * _LightColor0.rgb;
float4 finalDiffuse = float4(lightingModel* attenColor,1);
```

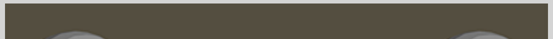


Banded-Lighting

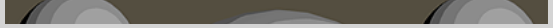
This is less of a lighting model, and more of a lighting tweak to show what you can do with simple mathematical operations on standard lighting models. This method works by breaking the lighting direction up into bands. This method can then be passed into any standard lighting model by replacing NdotL with the banded NdotL as shown below:

```
    float NdotL = max(0.0, dot( normalDirection, lightDirection ));

    float lightBandsMultiplier = _LightSteps/256;
    float lightBandsAdditive = _LightSteps/2;
    fixed bandedNdotL = (floor((NdotL*256+lightBandsAdditive)/_LightSteps))
* lightBandsMultiplier;
```



```
    float3 lightingModel = bandedNdotL * diffuseColor;
    float attenuation = LIGHT_ATTENUATION(i);
    float3 attenColor = attenuation * _LightColor0.rgb;
    float4 finalDiffuse = float4(lightingModel * attenColor,1);
    return finalDiffuse;
```

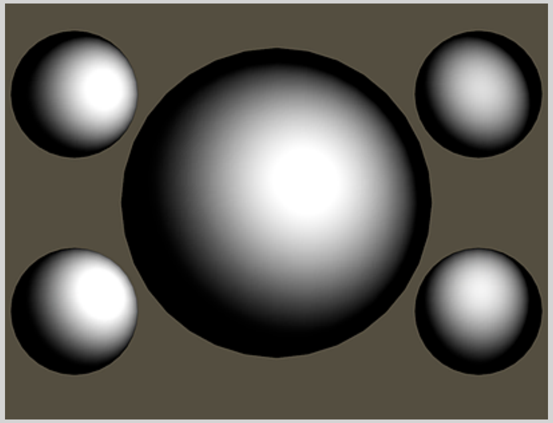


Minnaert Lighting

The Minnaert Lighting Model was originally designed to replicate the shading of the moon, so it's often called the moon shader. Minnaert's good for simulating porous or fibrous surfaces, such as the moon or velvet. These surfaces can cause a lot of light to back-scatter. This is especially apparent where the fibers tend to be mainly perpendicular to the surface like velvet, velour, or even carpets.

This simulation provides results that are pretty close to Oren-Nayar, which is also frequently called a velvet or moon shader.

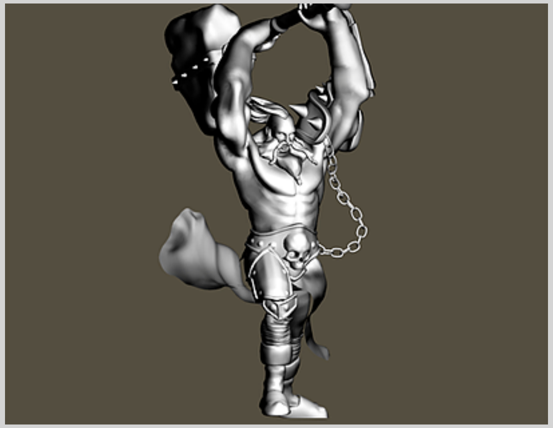
Below is an example approximation of Minnaert Lighting:



```
float3 viewDirection = normalize(_WorldSpaceCameraPos.xyz - i.posWorld.xyz);
float NdotL = max(0, dot( normalDirection, lightDirection ));
float NdotV = max(0, dot( normalDirection, viewDirection ));

float3 minnaert = saturate(NdotL * pow(NdotL*NdotV,_Roughness));

float3 lightingModel =  minnaert * diffuseColor;
float attenuation = LIGHT_ATTENUATION(i);
float3 attenColor = attenuation * _LightColor0.rgb;
float4 finalDiffuse = float4(lightingModel * attenColor,1);
     UNITY_APPLY_FOG(i.fogCoord, finalDiffuse);
     return finalDiffuse;
```



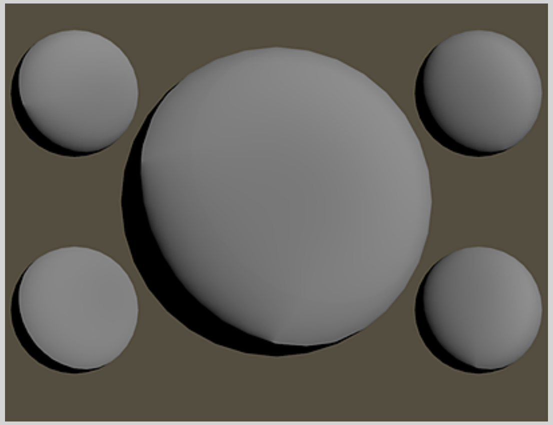
Oren-Nayer Lighting

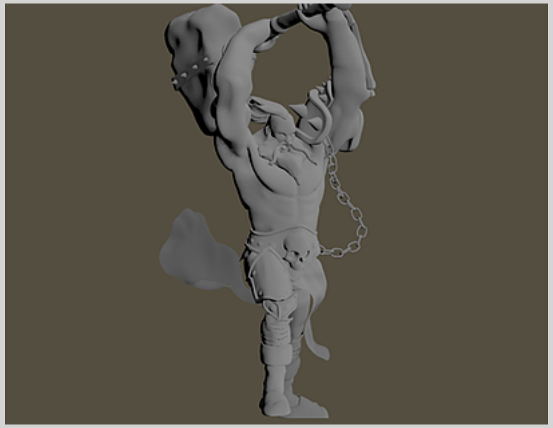
The Oren–Nayar reflectance model is a reflectivity model for diffuse reflection on rough surfaces. This model is a simple way to approximate the effect of light on a rough, yet still lambertian, surface.

Below is a simple implementation of Oren-Nayer:

```
     float roughness = _Roughness;
     float roughnessSqr = roughness * roughness;
     float3 o_n_fraction = roughnessSqr / (roughnessSqr
+ float3(0.33, 0.13, 0.09));
     float3 oren_nayar = float3(1, 0, 0) + float3(-0.5, 0.17, 0.45)
* o_n_fraction;
     float cos_ndotl = saturate(dot(normalDirection, lightDirection));
     float cos_ndotv = saturate(dot(normalDirection, viewDirection));
     float oren_nayar_s = saturate(dot(lightDirection, viewDirection))
- cos_ndotl * cos_ndotv;
     oren_nayar_s /= lerp(max(cos_ndotl, cos_ndotv), 1,
step(oren_nayar_s, 0));


     //lighting and final diffuse
     float attenuation = LIGHT_ATTENUATION(i);
     float3 lightingModel = diffuseColor * cos_ndotl * (oren_nayar.x
+ diffuseColor * oren_nayar.y + oren_nayar.z * oren_nayar_s);
     float3 attenColor = attenuation * _LightColor0.rgb;
     float4 finalDiffuse = float4(lightingModel * attenColor,1);
```





view
nte
click
reloa

Leave a message...

Wow, I just stumbled upon your blog and must say it is such an incredibly useful resource. The psuedocode is super useful. I only wish I had known about this resource when I started learning to write my first shaders in Unity! Now I'll get to experiment with different approaches than just Blinn-Phong.
Stefan · 7 months ago

amazing! please do more unity shader stuff, your are best on the internet. Really nice to have pseudocode algorithms for the lightning models pulled out from main info (which lets face it is main reason we are here and meat of the code)
Isaac Surfraz · a year ago

Not sure if you still check this or how long ago this was posted, but I don't know much about this type of programming, but I'm trying to alter UE4's source code to implement banded lighting. I understand all the concepts you present in that section, but when you use the variable _LightSteps, what is that referring to exactly? I can't find an equivalent variable in the source code by name alone, I need to understand what concept that represents.
Seth · a year ago

Hi Seth,
_LightSteps is a custom variable you can use to define the number of bands you want to create. I should have called this out specifically in my article.

Apologies,
Jordan
Jordan Stevens · a year ago