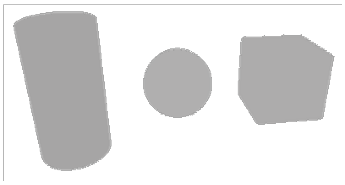
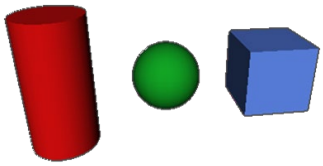
Lighting & Shading

Applying virtual lights to our 3D meshes



Now that we have the ability to render fully 3D meshes, you'll notice they look rather flat without any lighting or shading information applied to them.



When you think of lighting and shading on a 3D models, you probably picture the meshes being brighter on one side and darker on the other, with a gradient in between. That might look like the image to the right.

How do we achieve this? What kinds of calculation must we perform to get an accurate, or at least plausible, result? What information do we need before we can perform those calculations? And how do we combine the results of these calculations in a physically plausible way? Let's first define the terms *lighting* and *shading*.

Lighting & Shading

The terms *lighting* and *shading* refer to different aspects of our rendering engines. That being said, the terms are sometimes used interchangeably since they're so similar.

Lighting generally refers to the process of determining how a virtual light interacts with a virtual surface: the mathematical equations that approximate light in the physical world. *Shading*, on the other hand, is what we're doing when applying those lighting results to the rasterized pixels of a mesh. *Shading* encompasses running the lighting equations, combining the results with surface material data and getting the final color on the screen.

Simulation vs. Approximation

Note the use of the term *approximation* here. This means we're running equations that, while based on the physics of light, give us a close but not 100% physically correct result. In a real-time application like a game, this is extremely important: *fast and close* is better than *perfect but slow*. Any shortcuts we take here mean we have more time within a single frame to do other work, such as texturing, shadows and post-process effects.

Unless you're writing a specialized form of a ray tracer known as a *path tracer*, you're not really simulating the light in a 3D scene. And even in a path tracer, light is far too complex to simulate *all of it*; care is taken to simulate as much as necessary while allowing the final image to be created in a reasonable amount of time. Recent hardware and API advances allow us to do more in real-time, but there are still trade-offs.

Colors

Let's take a quick look at how colors are stored in a computer and how they're used in shaders, as the eventual output of all of this work is a color. Color systems for modern displays are often either 24-bit or 32-bit. These terms refer to the number of bits used to store a single color, enough for one pixel.

In both systems, the first 3 bytes (24 bits) are divided up into three 8-bit color channels: red, green and blue. With 8 bits (or one byte), a single channel can represent one of 256 unique values. With three channels, the total number of unique colors becomes 256*256*256, or approximately 16.7 million. This matches how most standard digital display devices (monitors, cell phone screens, etc.) display colors.

Red	(1.0, 0.0, 0.0)	White	(1.0, 1.0, 1.0)
Blue	(0.0, 0.0, 1.0)	Black	(0.0, 0.0, 0.0)
Purple	(1.0, 0.0, 1.0)	Grey	(0.5, 0.5, 0.5)

But what about the last 8 bits of 32-bit colors? Those bits are devoted to an *alpha* channel that is used most often for transparency. Instead of describing even more unique colors, the alpha value describes how to blend this color with another one. For instance, an alpha value of 50% means "use 50% of this color and 50% of the existing color when layering these pixels on top of one another". An alpha value of 10% would mean "10% of this color and 90% of the existing color", resulting in the rendered pixel appearing to be extremely transparent.

> *Note that, while these are certainly the most common, plenty of other color representations exist, some of which are necessary for more advanced graphics effects such as High Dynamic Range (HDR) rendering. For instance, color systems exist where each individual color channel is an entire 32-bit float value, meaning you can represent colors brighter than white.*

Storing Colors

There are several ways we could store a color in a computer. Basic color representation systems like 24-bit and 32-bit color are generally stored as unsigned integer values: each color channel is a standard 8-bit unsigned integer, and all three are packed next to each other in memory.

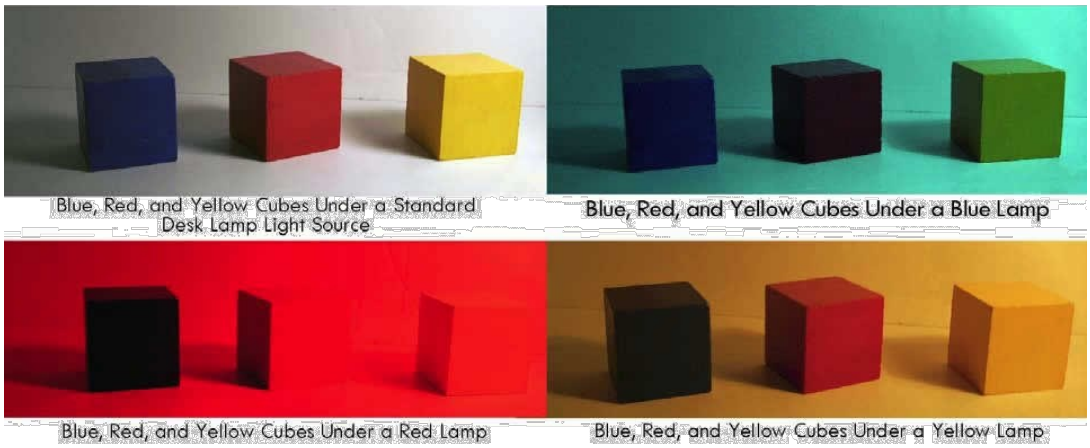
If you've ever done any front-end web development, you might have written colors instead as hexadecimal values like #FFFFFF or #FF0033. Hex colors like this have the same goal: represent 256 unique values per color channel. Since each digit in a hexadecimal representation has 16 possible values, two digits is 16*16, or 256 unique values. Since most front-end web work is stored as plain text, hexadecimal ends up being a very compact way of representing 16.7 million possible values.

In either case above, we're representing unsigned integer values for each color channel. However, once you get to a shader, we actually want the values as floats instead of integers. This is not a simple cast, however, as we also need to change the range of the values. As integers, the range is 0 - 255, but as floats we want values within 0.0 – 1.0. This greatly simplifies how we work with colors, as operations like multiplication and division become much easier. *Whenever a shader reads from **textures** or writes to **render targets**, it automatically converts from int to float or float to int and also adjusts the ranges.*

However, do note: Any custom color data we want to store in C++ that will eventually make its way down to a shader, such as the color of a light or a color tint of a game entity, should be stored in our C++ code as three float values (in the 0.0 – 1.0 range). Since we directly copy our data to constant buffers, no automatic conversion occurs; the shader doesn't know it's actually a color, so its easiest to represent it that way from the start.

Combining Colors

There are several situations in which we'll need to combine colors together. When two lights shine on the same surface, that surface should get brighter. Therefore, the colors of each light need to be added together. However, when light hits a surface, it is also tinted by the color of that surface. For instance, if purple light hits a perfectly blue surface, only the blue portion of the light is reflected. This requires us to multiply the light color and the surface color to get the final perceptual color. If we want to make a color twice as bright, we could multiply it by 2. To make it half as bright, we'd divide by 2.

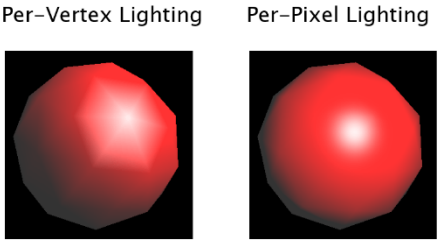


The example photographs above show three cubes under various lighting conditions. Take note of the red cube under a blue lamp and the blue cube under a red lamp: they appear black!

We'll be doing our color work in a shader and, as stated above, the shader works with colors as vectors of float values between 0.0 – 1.0. This means that multiplying two colors together is as simple as multiplying the two vectors together. Adding two colors is simple vector addition. Brightening and darkening requires multiplying or dividing the vector by a scalar. Luckily for us, all of these operations are baked into HLSL's syntax, leading to very concise code.

Which Shader(s) Perform Lighting & Shading?

While we could perform lighting equations in either the vertex or pixel shader, the results will look very different. Any work we do in the vertex shader gets linearly interpolated across the face of each triangle. However, most of our lighting results should not have a linear result over the surface of the 3D model! This means we need lighting equations to happen at the pixel level.



Requirements for Shading

We need several pieces of information before we can run lighting equations and apply the results to our meshes. Some of this data comes from the mesh itself, some from the material and some we need to define within our engine.

1. Surface Information – Data about the surface itself
2. Light Sources – Representations of virtual lights in our 3D scene
3. Lighting Equations – The actual math to approximate the physical light/surface interaction

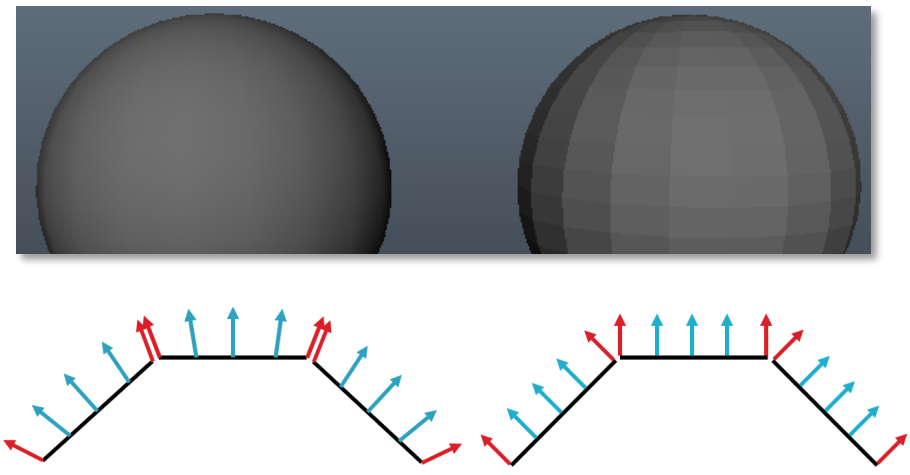
1. Surface Information

There are several pieces of information that define the surfaces we're rendering and shading. Some of this data, such as **surface normals**, comes directly from our meshes. Other data, such as **material properties** like roughness or color tint, are often defined by artists but are not part of the geometry definition. In our engines, we'll be defining materials through code, though we could load them from external sources if they matched up with our material definitions.

Surface Normals

The normal of a point on a surface is a vector perpendicular to the tangent plane of that surface at that point. In other words, it's the direction pointing perfectly away from the surface. As discussed in Reading 7 – 3D Models & Materials, we need to store this information at each vertex, though it gets interpolated by the rasterizer to give us a normal for each pixel.

If the normals of all three vertices are exactly the same, then the normal of each rasterized pixel will also all be the same, resulting in a faceted look (sometimes called *hard shading*). If, however, each vertex's normal is different, then each pixel will end up with a slightly different normal, resulting in a smoother-looking surface (*smooth shading*).



Smooth shading vs. hard shading

Material Properties

Exact material properties vary from engine to engine, but basic properties include an overall **color tint** for the surface and a **shininess** (or its inverse, **roughness**) value for reflections. Eventually we'll be using textures for surface colors, but even then, an overall color tint value can still be applied. These values will be defined in our C++ material system and fed into shaders when rendering. This means our shaders and material systems must both be written to store and use the same kinds of data.

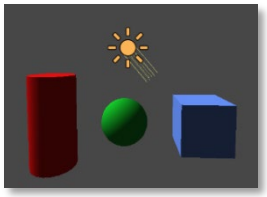
2. Light Sources

Next, we need to define the *sources of light* in our virtual 3D world. We want to mimic real-world light sources while keeping the overall light contribution easy to calculate, which means we'll need to simplify the way we define a light's *position* and/or *direction*.

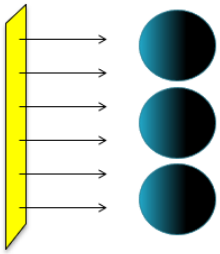
We'll also generally store a *color* for each light, as well as an *intensity* value that acts as a multiplier for the color. In this way, we can separate the color from the overall brightness, and allow a light to contribute more or less light to a scene irrespective of its color.

The goal of our lighting equations is to compare the *direction the light is traveling* with the *direction the surface is facing*. Ultimately, we need to know the direction from the light source to a point on the surface we're rendering (a pixel, in this case). As most real-world lights have area (like a bright computer monitor) or volume (a light bulb or the sun), we would need to employ calculus to properly approximate the amount of light coming from an infinite number of points on those surfaces. That is generally too much work for a real-time application. So, instead, we use simplified representations of light sources that are mathematically easy to work with. Several of these are *punctual* light sources, meaning they represent a single point in space instead of an area or volume.

Directional Lights



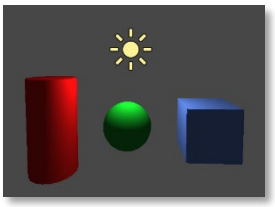
Since we ultimately need a direction for a light, we could just pick a direction and always use that for all of our calculations. This is the simplest light definition, as there is no per-pixel direction calculation: we always know which direction the light is traveling. This is known as a *directional light*.



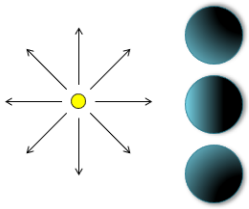
Directional lights have a few interesting properties. First, they do not need a position, since it wouldn't affect their direction. Second, all of the light for a directional light is effectively parallel. Third, while being a computationally easy light source, directional lights do not have an actual physical analog: if all of the light rays are parallel (and there is no starting position), a directional light effectively represents an infinitely large wall of light infinitely far away. No initial position means that directional lights cannot attenuate over distance.

While that last fact might make directional lights seem useless, they are actually well-suited to approximate light from the sun. The sun is so much larger than the earth that the light coming from it might as well be parallel for our simple approximations.

Point Lights



While directional lights are great for a large outdoor light source (like the sun), we also probably want to represent light sources akin to light bulbs. For this, we use *point lights*. As the name suggests, point lights are defined as a single point in space from which all of the light emanates. Because of this, they can be defined with just a position instead of a direction.

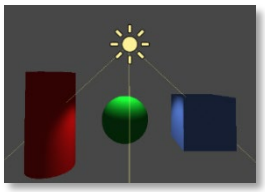


However, we need to know which direction the light is traveling when it strikes any given point on a surface. This means, for each pixel, we need to calculate the light's direction; while the position of the light is fixed during rendering, each pixel has a different position in space. This can be achieved using simple vector subtraction: `surfacePosition - lightPosition`.

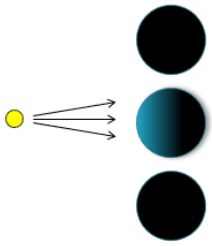
> *Implementation note: we don't inherently have a pixel's final world position in a pixel shader, so we'll need to specifically calculate that in the vertex shader and pass it through the pipeline.*

Point lights also do not have exact physical analogs since they represent infinitely small sources of light. They still make great virtual light bulbs because they emit light in all directions and are fast enough for real-time. To mimic real light, the intensity of a point light should attenuate (lessen) as the distance between the light and the surface increases. This is an extra calculation we must perform.

Spot Lights



The last of the three basic sources is a *spot light*, often used to approximate flashlights, car headlights or any other conical light source. To define a spot light, we need both *a position and a direction*. The position specifies where the light is, and the direction specifies which way the spot light's cone is facing. In addition, a spot light will need a falloff value that defines how its light dims (or *falls off*) as it spreads out from the light's main direction.



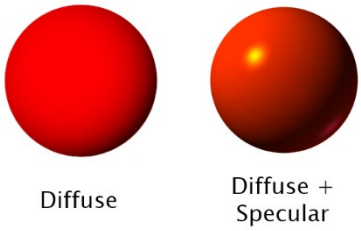
Implementation-wise, a spot light is much like a point light. The same calculations are performed for the direction of the light. An extra calculation is used to dim the light's contribution as a surface is farther from the center of the spot light's cone. In other words, a spot light is just a point light with an extra restriction on the direction of the light. Also, like point lights, spot lights should attenuate with distance.

3. Lighting Equations

Now that we have a surface and a light, we need to use them in conjunction to produce a value for shading. This is where actual lighting equations come into play. The big question is: which one(s) do we actually need to use? Many mathematical functions exist to approximate how light interacts with surfaces. Some are used to approximate light bouncing (or reflecting). Others approximate light transmitting through surfaces. Others approximate general light scattering (entering a surface, bouncing around and exiting elsewhere). Within each of these types, individual functions exist for

handling different properties of light/surface interactions, such as diffusion and reflection (also known as specular reflection). So where do we begin?

For a basic real-time application, we want to handle the most obvious lighting situations; the ones that make the most impact on the final color of a surface with the smallest computational cost. For this, we'll use a category of lighting functions called *Bidirectional Reflectance Distribution Functions*, or BRDFs. BRDFs approximate how light reflects off of a surface given the direction the light is traveling, the normal of the surface and, potentially, where our eye (camera) is in relation to the surface.



Diffuse | Diffuse + Specular

> *BRDFs are only one category of lighting functions. Others include Bidirectional Transmittance Distribution Functions (BTDFs) and Bidirectional Scattering Distribution Functions (BSDFs). These are often used in raytracing to calculate where a ray might go after striking a surface.*

There are many functions that are considered BRDFs, each of which approximates a single property of light. We'll be looking a few classic computer graphics BRDFs later in this document. In a later reading, we'll look at a physically-based alternative to one of our basic BRDFs. But first, a more general lighting equation.

The Rendering Equation

Developed in 1986, the Rendering Equation is a general equation describing the radiance (outgoing light) leaving a point on a surface as the sum of the light emitted by the surface and the reflection of all incoming light in a hemisphere oriented on the given point's normal.

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) \; + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

Whew, that's a mouthful! What does it mean? Let's break it down:

$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$

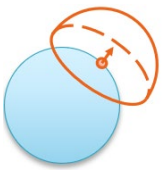
Outgoing Light: Light we *actually see* coming off the surface.

$L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$

Emitted Light: Light the surface *emits* ("creates"), like a light bulb or computer monitor.

$\int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$

Reflected Light: Total amount of light that hits the surface and *bounces off* towards us. We need to account for all light striking the surface. However, light cannot strike a (solid) surface from behind, so we only need to handle light in a hemisphere above the surface. To orient the hemisphere correctly, we rotate it such that the hemisphere's center matches the surface's normal.



$f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$

Our BRDF: The result of our lighting function, which is often just a scalar value describing how much of the reflected light we actually see.

$L_i(\mathbf{x}, \omega_i, \lambda, t)$

Incoming light: A single "ray" of light (its color and intensity)

$(\omega_i \cdot \mathbf{n})$

Dot product: The dot product between the surface normal and the direction back towards the light, which ensures that light striking the surface "head on" is brighter than light simply glancing off the surface. This is defined by Lambert's Cosine Law.

While the Rendering Equation does not describe every possible light/surface interaction, such as subsurface scattering or refraction, it does give a general form for what our basic lighting equations should approximate. However, as you can probably tell, this equation is a bit much for a real-time application. The integral in particular is probably too costly for a pixel shader in a game. Let's look at the individual pieces and then see how we can simplify the equation for games.

First up, definitions:

- $\mathbf{x}$ - Our geometry at a point in space
- $\mathbf{n}$ - Normal at $\mathbf{x}$
- ω_o - Direction of outgoing light (towards our virtual camera)
- ω_i - Negative direction of incoming light (meaning: the *direction to the light*)
- λ - Wavelength of light
- t - Point in time
- Ω - Unit hemisphere centered around $\mathbf{n}$ encompassing directions of *all possible* ω_i
- $\int_\Omega \dots \mathrm{d}\,\omega_i$ - Integral over the hemisphere Ω

So, where can we simplify? We'll go one by one:

- $\mathbf{x}$ - Necessary and simple, since it's just mesh data
- $\mathbf{n}$ - Necessary and simple, also from our mesh
- ω_o - May be necessary for some BRDFs, and easy to calculate: `cameraPos - surfacePos`
- ω_i - Necessary and simple. Direction from surface back towards light: `-lightDirection`
- λ - We'll assume R, G and B act similarly, so we can remove this
- t - Our "point in time" is a single frame, so we can remove this
- Ω - We're using simplified light sources, so no need for a full hemisphere
- $\int_\Omega \dots \mathrm{d}\,\omega_i$ - We'll have a finite number of light sources we can loop through, no integral necessary
- $L_e(\mathbf{x}, \omega_o, \lambda, t)$ - We'll skip emitted light for now, though it's easy to add back in later

Here is the Rendering Equation once we simplify it:

$$L_o(\mathbf{x}, \omega_o, \cancel{\lambda}, \cancel{t}) = \cancel{L_e(\mathbf{x}, \omega_o, \lambda, t)} + \cancel{\int_\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \cancel{\lambda}, \cancel{t}) L_i(\mathbf{x}, \omega_i, \cancel{\lambda}, \cancel{t})(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\cancel{\omega_i}$$

$$L_o(\mathbf{x}, \omega_o) = f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

This would work for a single light source, but since we'll have multiple lights in our scene, we need to combine the results of each together. Luckily this is straightforward: we simply add them, as multiple lights will linearly increase the brightness of a surface. We replace the integral over the entire hemisphere with a simple sum of all lights (basically a *for loop* through an array of lights):

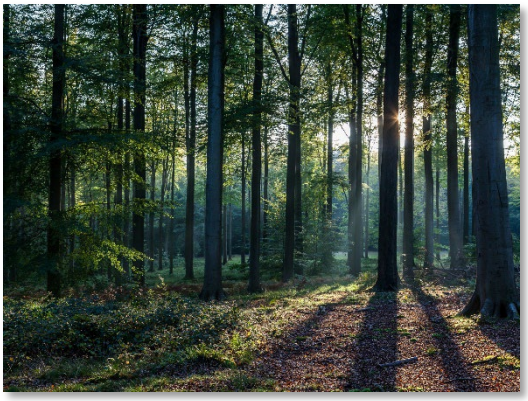
$$L_o(\mathbf{x}, \omega_o) = \sum_{light=1}^{numLights} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

This gives us an overall set of calculations for shading, which might look something like the following HLSL pseudocode. Note that some of the structures and functions below are *examples of things we might write ourselves*, not built-in language features!

```
float3 total = float3(0,0,0);
for (int i = 0; i < numLights; i++)
{
   Light light = lights[i];
   total += BRDF(light, surface) * LightColor(light) * dot(-light.Direction, normal);
}
return total;
```

As we'll see, some of this will be simplified further and other parts will be expanded, but hopefully this gives you a sense of how we can begin translating the rendering equation into code!

Lighting Approximations & Their BRDFs



As we've discussed, actual light is incredibly complex and far too costly to simulate or even fully approximate. Take this photograph of a forest for instance:

In this photo, each "object" (trees, leaves, logs, grass, etc.) has light striking it and bouncing towards the camera. That light is what we capture as the photograph. However, there is light-related phenomena here that our engines cannot handle quite yet, like shadows, translucent materials (leaves), volumetric light shafts (the light rays poking through the trees) and even lens flare.

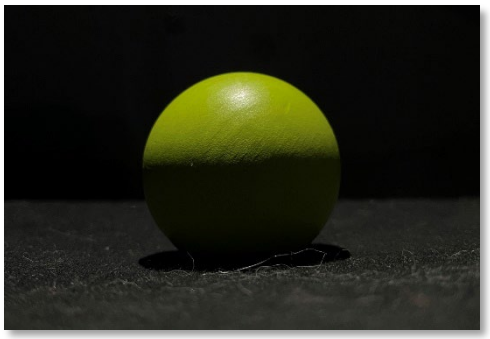
> *These effects aren't impossible with rasterization, but take a lot more work. They often require pre-rendering scene data (for shadows) or changing pixel colors once the entire frame has been rendered (post-processing).*

Direct Light vs. Indirect Light

Remember: we're using a rendering pipeline where each mesh is rendered in isolation; it has no knowledge of any other objects in the scene. We can really only focus on what's happening to the pixels of that single mesh, not how it might affect lighting on other objects. This means our lighting approximations are for **direct light** – light coming directly from a light source, bouncing and hitting our eye – and not **indirect light** – light that bounces around multiple times in a scene, picking up different colors from various surfaces and eventually hitting our eye.

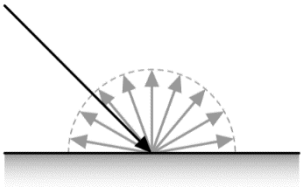
What Should We Approximate?

Here's a much simpler photograph: a ball in a dark room with a single light source overhead. (Don't mind the cat hair; that's not part of this, it's just everywhere in my house!) What do we see here?



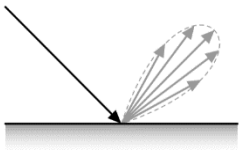
Diffuse

The most obvious piece is that the top half of the sphere is lit, while the bottom half is dark. There is a gradient from the light to dark areas. The light is striking each point on the surface and *diffusing* (spreading out) in all directions. In that way, these areas of the ball will look the same from any viewing angle. When the light strikes head on, its much brighter. When it just glances off, as on the very edges, its darker. Note that the transition from bright to dark is a gradient, though it is not linear!



Specular

The next rather obvious phenomenon is the bright highlight near the top of the ball. This is a *specular reflection* of the light source itself. Though it's a reflection, it is not perfectly mirror-like since the ball itself isn't perfectly smooth. The rougher a surface, the more subtle and "spread out" its reflections are. Some surfaces are so rough their specular reflections are almost imperceptible, but since every real-world object has some sort of reflectivity, it's a property of light we should aim to include.

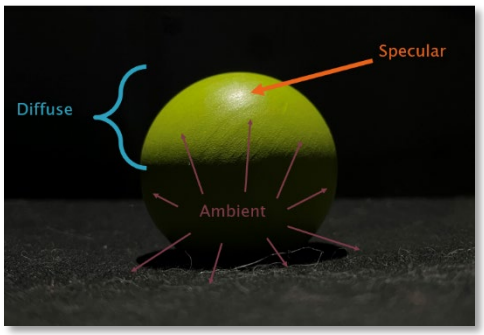


Ambient

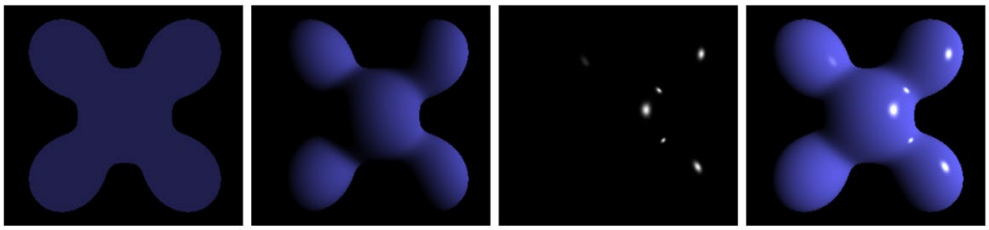
Ambient light is light that is *everywhere*: light bouncing around a scene. In the photo above, the underside of the ball is *dark* but not *pitch black*. This is due to light hitting the surface below it and bouncing back up. True global illumination, tracing the path of every single light ray, isn't feasible, but we *could* come up with a very simplistic approximation that prevents the backsides of our objects from becoming pitch black in an otherwise bright scene.

The Big Three: Ambient, Diffuse & Specular

Ambient, diffuse and specular are the big three computer graphics lighting approximations. We'll begin our journey into photorealistic rendering by looking at some basic values and equations for each of these. A future reading will replace some of these with more physically-based versions (at the cost of performance).



Let's take a look at these three in the context of a game engine.



Ambient + Diffuse + Specular = Combined

Each of these is chosen (ambient) or calculated (diffuse & specular) separately and added together. Note how the specular calculation literally just gives us the bright highlights meant to approximate reflections of a light source. The object doesn't look shiny until those are applied! Next, we'll look at the implementation of each phenomenon.

Ambient Term – Implementation

Ambient is an approximation of light bouncing around a scene and is meant to represent the minimum light level in a scene. Essentially, if no light directly hits a pixel, what color should it reasonably be? Note that this color will be applied *everywhere*, even on pixels that *do* receive direct light! Why? Because it's meant to be an extremely simplistic representation of *global illumination*: light that is everywhere.

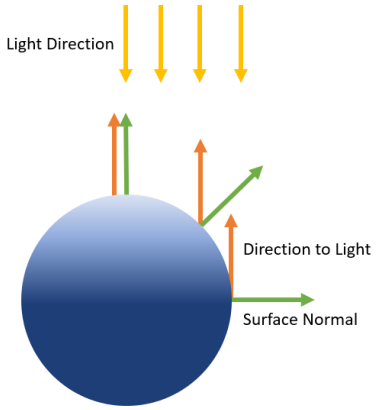
How do we get the ambient color for a scene? Pick one! If it's an outdoor scene under a blue sky, maybe its blue. In a desert? Tan or brown. In a forest? Green!

Really? Just pick one? Yup. Like I said: extremely simplistic. It's scene dependent, so choose a suitable color for the scene. Whatever color you choose, it needs to be ***subtle***. If you pick "white", every pixel will be white, which is as bright as possible. Remember: this should be the minimum light level in the scene, so choose a color that is as dark as the darkest area of your scene.

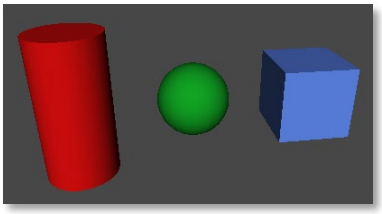
HLSL Shader Code

```
// - ambientColor is from a constant buffer
// - surfaceColor is the material's color tint (eventually will come from a texture)
float3 ambientTerm = ambientColor * surfaceColor;
```

Diffuse Term – Implementation



The intensity of diffusion – light striking a surface and bouncing in all directions – is dependent upon the light's direction and the surface normal. This means we need a BRDF to calculate the *diffuse term*. While there are several possible diffuse BRDFs, one of the simplest (and still fairly accurate) ones is Lambertian reflectance. This approximates light striking a perfectly matte surface, one where no specular reflection would be noticeable, such as unfinished wood. While most real-world surfaces exhibit both diffuse and reflection, the simplicity of Lambertian reflectance makes it an ideal choice.



The goal is to determine how the intensity of the light leaving the surface changes as the angle of impact changes. As stated earlier, this is described by Lambert's Cosine Law. We need to compute the cosine of the angle between two vectors: the *normal* of the surface and the direction from that surface *to the light*. If our vectors are normalized (having a length of 1.0), then the *dot product* gives us exactly this answer as, geometrically, the dot product is equivalent to the cosine of the angle between the vectors multiplied by their magnitudes.

$$\mathbf{L} \cdot \mathbf{N} = |N||L| \cos \alpha = \cos \alpha$$

Because of its simplicity, Lambertian reflectance is sometimes referred to as "N dot L" lighting.

In fact, this term is already baked into the Rendering Equation from earlier: $(\omega_i \cdot \mathbf{n})$. We simply fold it into our BRDF for diffusion. This will return a scalar value which denotes the intensity of the light leaving the surface. However, we still need to take into account the color and original intensity of the light, as well as the color of the surface. This makes our overall diffuse term:

$$I_D = \mathbf{L} \cdot \mathbf{N} C I_L$$

I_D is the intensity of the outgoing light, $\mathbf{L}$ is the direction to the light, $\mathbf{N}$ is the surface normal, C is the surface color and I_L is the light's overall color (taking into account its initial intensity). While not explicitly shown in the formula above, we must ensure the result of the dot product doesn't go negative on the back side of our objects, so we need to clamp it to the 0-1 range.

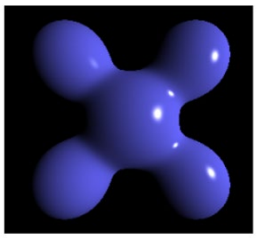
Unlike the ambient term, this must be calculated for each light in the scene! If you have more than one light, you'll need a loop to calculate the diffuse term of each and add them together.

HLSL Shader Code

```
// - input.normal is from vertex shader (interpolated by rasterizer)
// - direction, color & intensity of the light is from a cbuffer
// - surfaceColor is the material's color tint (eventually will come from a texture)
float3 diffuseTerm =
   saturate(dot(input.normal, -lightDirection)) * // Diffuse intensity, clamped to 0-1
   lightColor * lightIntensity *                  // Light's overall color
   surfaceColor;                                  // Tinted by the surface color
```

Specular Term – Implementation

Using just Lambertian reflectance, we can render pretty accurate matte surfaces – ones without any reflections. But most surfaces have some sort of obvious reflection, even if it's subtle (or not so subtle in the photo to the right).

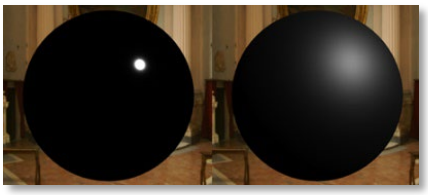


Recall that we're dealing with only *direct light* in our engines. The only information we have available when rendering is the current pixel and the lights in our scene. This means we cannot have objects reflect other objects or even themselves. But what we *can* do is approximate a reflection of the *light source itself.* This can give the appearance of a very reflective surface even though its not reflecting the full environment.

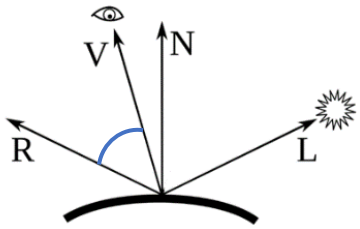
> *One thing to note about reflections: they change with viewing angle. If everything in a scene remains stationary, but the viewer moves, these shiny spots should appear to travel across the surface. This means that not only do we need information about the light and the surface, we also need to know the position of the virtual camera itself.*

We need a BRDF that approximates the "shiny spots" on a surface, also known as *specular highlights*. The classic choice for specularity is the Phong reflection model. This is a computationally simple, but somewhat physically-inaccurate, way of approximating reflections of light sources.

> *You may have also heard of Phong shading. This is a technique for interpolating the normals of a surface so we can perform smooth shading across a triangle. In effect, this is per-pixel lighting, which we've been gearing up to do this whole time. Both Phong shading and the Phong reflection model were created by Bui Tuong Phong in the mid-1970s.*



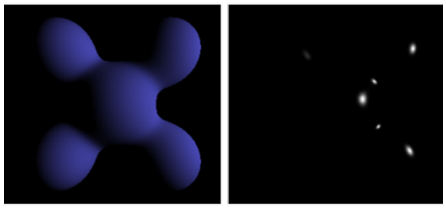
So, how does Phong work? For a very shiny surface, we want our specular highlights to be strongest where we have a perfect mirror reflection and have a rather sharp falloff after that. For a less shiny surface, the overall highlight should be duller and fall off slower. Both of these can be seen to the right.



As is the case with all BRDFs, we'll be dealing with vectors again. Rather than comparing a light direction and a normal, we need to compare the *perfect reflection vector* of the light across the normal (R in the diagram to the right) and our *view vector* (a vector from the surface to the camera – V in the diagram). If R and V are very close, then a perfect reflection is hitting the camera and that pixel should be very bright. As R and V spread apart, the camera is farther from a perfect reflection, so our specular value should diminish quickly.

> *Comparing the angle between two vectors? Sounds like a job for the dot product! However, while that is part of the BRDF, we also want the value to drop to zero quickly. Since the result of the dot product will be between 0 and 1 (after clamping), raising it to a power will make it very small very quickly (especially a large power, like 64 or 128).*

The BRDF for Phong is: $k_s(\hat{R}_m \cdot \hat{V})^{\alpha} i_m i_s$



Diffuse + Specular

The terms are as follows:

- k_s is a specular constant (usually just 1.0)
- R_m is the normalized light reflection vector
- V is the normalized view vector (surface to camera)
- α is the material's shininess constant, used as a power here
- i_m is the light's color (including its intensity)
- i_s is a specular intensity used to adjust the overall result

The trickiest part is calculating the correct vectors. The view vector is easy as long as we know the surface's position (passed in from the vertex shader) and the camera's position (passed in through a constant buffer). For the reflection vector, most shading languages have a reflect() function built in! The rest is a pow() function and some multiplies.

HLSL Shader Code

```
// Calculate the reflection of the INCOMING light dir
// using the surface normal (ensure both are normalized first!)
float3 refl = reflect( lightDir, input.normal );

// Calculate cosine of the reflection and camera vector
// - saturate() ensures it's between 0 and 1
float RdotV = saturate( dot( refl, dirToCamera ) );

// Raise the that to a power equal to some "shininess"
// factor – try large numbers here (64, 200, etc.)
float3 specularTerm =
   pow(RdotV, shininess) *      // Initial specular term
   lightColor * lightIntesity * // Colored by light's color
   surfaceColor;                // Tinted by surface color
```

One quick note about that very last line: should the reflection actually be tinted by the surface color? Technically, it depends on several factors, none of which we have parameters for. Is the surface metallic or dielectric (non-conductive, like rubber or wood)? Does the surface have a clearcoat or gloss layer on it, like car paint or varnish?

For now, feel free to experiment with and without the surface color tinting. This will be handled in a later reading when we replace Phong with a more physically-based specular term.

Putting It All Together

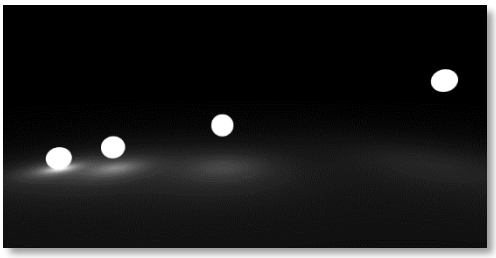
You have ambient. You have diffuse. You have specular. How does it all fit together?

$$I_{\mathrm{p}} = i_{\mathrm{a}} + \sum_{m \in \text{ lights}} ((\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}})$$

- I_{p}: Final light intensity for a pixel
- i_{a}: Ambient term
- $\sum_{m \in \text{ lights}}$: Add up the following for each light
- $(\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}}$: Diffuse (Lambert)
- $(\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}}$: Specular (Phong)

This is effectively a for loop in which you calculate the diffuse and specular for a single light and add it to an ongoing sum. If you don't have an array of lights, then simply perform the calculations for each individual set of light data and add them up!

Attenuation



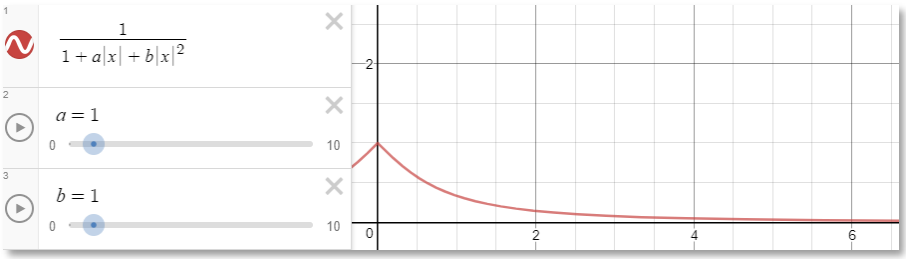
One quick addendum to our lighting calculations. If we're trying to mimic real-world lights, then a light's intensity should attenuation (or lessen) with distance traveled. See the image to the right for an example.

We can handle attenuation as long as we can calculate the distance between a light source and a surface. Since directional lights have no position, they cannot attenuate. Point lights and spot lights, on the other hand, *can* and *should* attenuate with distance. For that, we need is an attenuation function.

If you were to search for an attenuation function online, you'd probably find the following:

```
attenuationFactor = 1.0 / (1.0 + a * dist + b * dist * dist))
```

This is a classic computer graphics attenuation function. It works. It's tweakable. And it kind of sucks. It's designed to be flexible, as it can represent both linear attenuation (fake) or exponential attenuation (how light actually works), but its parameters don't correspond to world units. In other words, if you want a light to attenuate to zero at an exact range in your 3D scene, this function won't help! It doesn't hit zero at a finite distance!



Instead, it's often much more useful to have a smooth attenuation function that hits zero at an exact distance. While it might not be perfectly exponential, it's often good enough and very predictable. Here is an example of one such function:

$$attenuationFactor = \max(0, 1.0 - dist^2 / range^2)^2$$

If we graph this out, we can see it hits zero at exactly the desired range and still has a smooth falloff.

HLSL Shader Code

```
// Apply this to the diffuse and specular terms of any
// point or spot lights in your shaders.
// dist is the distance between the light source and the pixel
// range is the maximum range of the point or spot light
attenuationFactor = saturate(1.0f - (dist * dist) / (range * range));
attenuationFactor *= attenuationFactor;
```