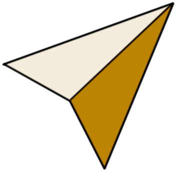


COMP4610/COMP6461

Week 8 - Introduction to Lighting

<Print version>

The Phong Lighting Model

[Lighting Models]

Early lighting models were all about **feasability** rather than **realism**. These lighting models are still common today, and can be tricky to get to look right.

In contrast, modern lighting models look great by default, are easier to work with, but have higher computational costs.

An early (1972) ‘halftone’ lighting model on limited hardware by Edwin Catmull, who went on to found Pixar.

Phong Reflection Model

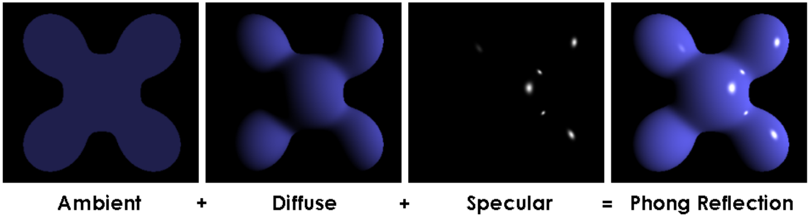


Image Brad Smith 2006 ShareALike 3.0 obtained from wikipedia

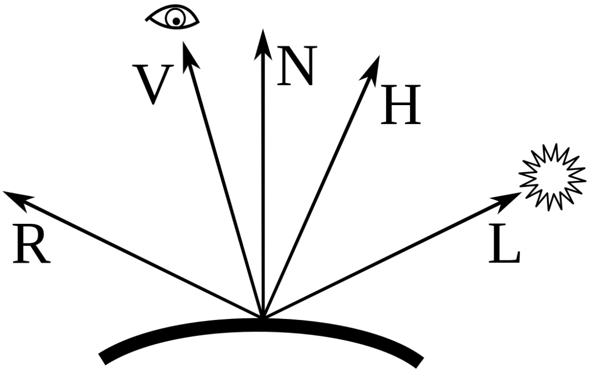
Phong Reflection Model

The Phong reflection model combines: **ambient**, **diffuse**, and **specular** lighting effects. All vectors are normalized.

$$I_p = \sum_{m \in \text{lights}} \underbrace{k_a i_a^m}_{\text{ambient}} + \overbrace{k_d (L^m \cdot N) i_d^m}^{\text{diffuse}} + \underbrace{k_s i_s^m (R^m \cdot V)^\alpha}_{\text{specular}} \tag{1}$$

I_p	Intensity for the surface patch.
L^m	Direction vector from surface toward light m.
R^m	L^m perfectly reflected off the surface.
V	Direction vector pointing towards viewer.
N	Surface normal.
k_a, k_d, k_s	Material constants.
i_a^m, i_d^m, i_s^m	Lighting constants for light m.

Phong Reflection Model



Wikipedia https://en.wikipedia.org/wiki/Phong_reflection_model#/media/File:Blinn_Vectors.svg

Blinn-Phong Reflection Model

It's quite common to drop the reflected vector, and instead use the normalized 'half' vector.

$$I_p = \sum_{m \in \text{lights}} \underbrace{k_a i_a^m}_{\text{ambient}} + \overbrace{k_d (L^m \cdot N) i_d^m}^{\text{diffuse}} + \underbrace{k_s i_s^m ({\color{red}N \cdot H})^\alpha}_{\text{specular}} \qquad (2)$$

where

$$H = \frac{L + V}{|L + V|} \qquad (3)$$

In cases where the surface is far away from the viewer, H can be considered constant, and unlike R, need not be recalculated per-pixel. Surprisingly, the Blinn-Phong simplification produces a better approximation to **BRDF** than Phong (more on BRDF later).

Phong Clipping

When calculating Phong the diffuse and specular intensity may sometimes be negative. Because of this we need to clip the intensity to using $\max(\cdot, 0)$.

For diffuse light this will be more apparent when there are multiple lights. In this case 'negative' light from one light will cancel out the others light.

Phong and Colour

The Phong lighting model only specifies the **light intensity** for a particular patch. For this reason, it is often necessary to apply the formula three times for each of the primary colours.

In most cases, vectorized math can be used to perform these three calculations simultaneously. In that case, I_p would be a three-vector containing the RGB values, as would the material and lighting constants.

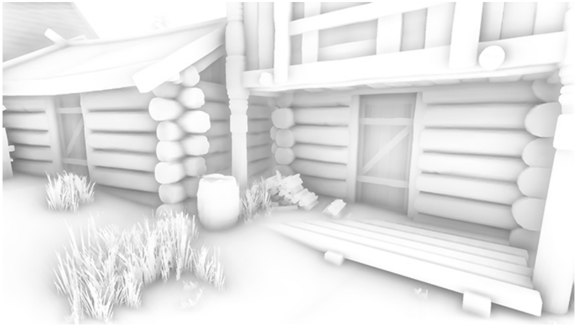
Ambient Light

- Areas in shadow still receive some ‘ambient’ light.
- Lighting is independent of both viewer angle, and surface normal $\rightarrow$ flat looking images.
- Ambient lighting is usually applied to the diffuse material properties of a surface’s material.

Shadowed areas are not completely black.

Ambient Occlusion

In reality ambient light is not uniform, and can be occluded. This affect can be applied as a post processing affect (**SSAO**), and generally makes lighting look much better.



Credit `https://github.com/Unity-Technologies/PostProcessing`

Diffuse Lighting

- Often called **Lambertian** reflectance.
- Assumes a rough surface that emits light evenly in all directions.
- Therefore lighting is independent of the viewing angle (but not the surface normal).
- Used extensively in early graphics as it just requires calculating $L \cdot N = \cos\alpha$ (for unit vectors).

Specular Lighting

- Lighting is dependant on the surface normal, and the viewing angle.
- What we are seeing is actually blurry refelection of the light source.
- Unlike diffuse lighting, this does not interpolate well, and so should be calculated per pixel.
- The Phong lighting model uses $(R^m \cdot V)^\alpha$. which is not physically plausible, and may give poor results. (**PRB** resolves this, more on that in future weeks...)

[Tips and Tricks]

Some common issues with lighting...

- Make sure you don't have a negative sign somewhere? Remember, L goes from surface $\rightarrow$ light, not light $\rightarrow$ surface etc.
- Remember to normalize your vectors. A common mistake is to forget this.
- When debugging, I like to add a sphere to a scene just to make sure that lighting is as expected.
- Normals can be tricky...

OpenGL

- OpenGL's fixed function pipeline implements the **Phong reflectance** model.
- Supports multiple lights, and control over material properties.
- Lighting is calculated **per-vertex**, using the **Blinn-Phong** reflectance model.
- Shaders are required for **per-pixel** lighting. If using shaders the entire model must be implemented by the user using shaders (predominantly in the fragment shader).

OpenGL - Lighting

First we enable lighting,

```
gl.glEnable(gl.GL_LIGHTING);
gl.glEnable(gl.GL_LIGHT0);
gl.glEnable(gl.GL_NORMALIZE); // not required, but a good idea...
```

then a light can be setup as follows,

```
float[] color = {0.3f,0.7f,0.1f,1};
gl.glLightfv(gl.GL_LIGHT0, gl.GL_AMBIENT, ambientColor, 0);
gl.glLightfv(gl.GL_LIGHT0, gl.GL_DIFFUSE, diffuseColor, 0);
gl.glLightfv(gl.GL_LIGHT0, gl.GL_SPECULAR, specularColor, 0);
// define the light location (x,y,z,0) for directional, (x,y,z,1) for point.
// note: position is defined according to the current model view matrix.
gl.glLightfv(gl.GL_LIGHT0, gl.GL_POSITION, lightPosition, 0);
```

and a material as follows.

```
gl.glMaterialfv(gl.GL_FRONT_AND_BACK, gl.GL_AMBIENT_AND_DIFFUSE, color, 0);
gl.glMaterialfv(gl.GL_FRONT_AND_BACK, gl.GL_SPECULAR, color, 0);
gl.glMaterialfv(gl.GL_FRONT_AND_BACK, gl.GL_EMISSION, color, 0);
gl.glMaterialfv(gl.GL_FRONT_AND_BACK, gl.GL_SHININESS, shine, 0);
```

OpenGL - Normals

For the lighting to work we also need to define normals at each vertex.

```
gl.glBegin(gl.GL_POLYGON);
gl.glNormal3d(nx, ny, nz); // normal will be shared for the entire triangle.
gl.glVertex3d(x1,y1,z1);
gl.glVertex3d(x2,y2,z2);
gl.glVertex3d(x3,y3,z3);
gl.glEnd();
```

also, you can temporarily turn lighting off using,

```
gl.glDisable(gl.GL_LIGHTING);
// draw something.
gl.glEnable(gl.GL_LIGHTING);
```

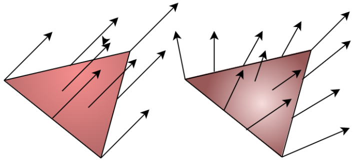
which can be useful for drawing UI elements, or objects that are unlit.

The Phong Lighting Model

Normals

Normals

- For a surface, the normal is a vector orthogonal to the surface.
- Normals are often calculated using the **cross-product**, however is is also common to generate normals when creating geometry.
- This method of ‘cheating’ the normal is quite useful, and used in both **bump mapping** and **normal mapping**.



Left, surface has identical normals, flat looking shading. Right, surface normals change over the surface giving smooth shading.

Transforming Normals

- Normals undergo a transformation that is based on the **model-view matrix**. These normals are important for the lighting calculations.
- If M is the upper left 3x3 matrix of the model view matrix then the transformation matrix on the normals is $(M^{-1})^T$. We can derive this by asserting that the new normal must have the same dot product between any vector and the original normal and that of the transformed vector and the new transformed normal. That is if n is the original normal, m is the new normal and u is any vector then we expect:

$$m \cdot Mu = n \cdot u \tag{4}$$

- The matrix $(M^{-1})^T$ may not preserve the length of the normal vector. As the reflection model may expect unit length normals these vectors may require normalization. This may be done within a shader. Or in the fixed pipeline approach this is enabled by: gl.glEnable(gl.GL_NORMALIZE)

Limitations of the Phong Model

- The Phong model is a combination of three different ‘wrong’ lighting models.
- Because of this the lighting model may break **conservation of energy**.
- Phong looks OK for plastic like surfaces, but is much worse for natural looking surfaces.
- However, with enough tweaking you can almost always get things to look right using Phong. This is especially true when combined with normal maps, and textures.
- It can be tricky for artists to express their ideas in terms of the parameters provided by Phong (specular color, shininess etc). Also, we could take measurements from real world objects, but how do these measurements map to the Phong lighting model?
- Different shaders are needed for special cases (metal shader, Fresnel, mirrored surfaces, etc).
- Is there a better way? Yes, we will cover this in week 10.

The Phong Lighting Model

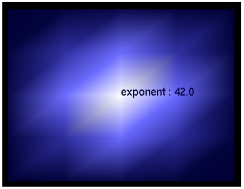
Shading

Flat Shading (Per surface)

- Single lighting calculation for each polygon.
- Minimal lighting calculations required.
- Can be used stylistically.

Gouraud Shading (Per Vertex)

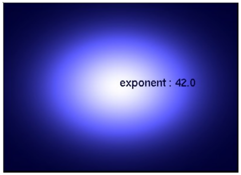
- Lighting is calculated at the vertices and then **interpolated** across the pixels of the polygon.
- Originally interpolated in pixel space, but Blinn later improved the algorithm to perform perspective correct interpolation.
- Works well if lighting does not change quickly.
- Sometimes need to subdivide large surfaces to improve lighting quality.



Spot light on a tiled surface using Gourad Shading.

Phong Shading (Per Pixel)

- The surface **normals** are interpolated across the pixels of the polygon and then the Phong reflection model is used to calculate the colour at each pixel.
- Requires a lot more computation at each pixel (in comparison to Gouraud shading).
- With modern GPUs and their programmable shaders we can simply and efficiently implement Phong shading.



Spot light on a tiled surface using Phong shading

The Phong Lighting Model

Lighting Sources

Introduction

- Photorealism requires:
  - accurate representation of the surfaces (and their properties) that make up scene, and
  - accurate modeling of light and its interaction within the scene.
- Both of these are complex, can use large amount of memory and are computationally intensive. Hence, simplification is required to reduce programming complexity, memory footprint, and computational overhead.

Types of light source

- In theory, everything is a light, but in practise we model light emitting objects separately.
- Common light types include
  - Ambient lights
  - Point lights
  - Directional lights
  - Area lights

Ambient Lights

- All objects receive some baseline light
- Common to set this to something low (0.1,0.1,0.1). But may also want to tint it blue, if outside.
- Points in shadows should use this lighting.
- Sometimes ambient light is baked into the model (either as vertex colors or as a texture). But **SSAA** is usually the way to go.

Point Lights

- A very common type of light source.
- Defined by a location, and the intensity.
- Calculate direction of light based on a vector from the surface to the light.
- Point lights have no size.
- **Area lights** can be approximated by generating multiple point lights. However, this is slow.
- Light should attenuate at $\frac{1}{d^2}$ but we don't typically use this. Instead we either apply no no drop off, or a linear drop off.

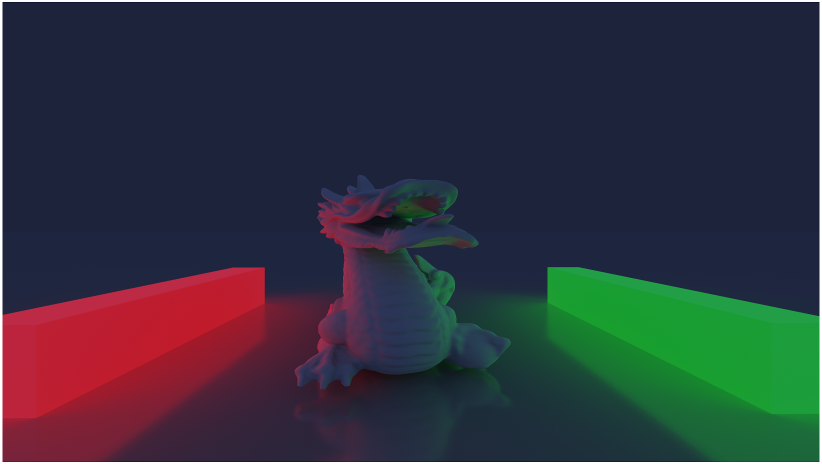
Directional Lights

- Defined a direction.
- Not that different from a very distant point-light.
- When using **shadow maps** (more on this later) directional lights require only a single shadow map (point lights need a cube, i.e. 6 maps).
- A good choice for for sunlight.

Global Illumination

- In reality every surface is emitting.
- This is called **Global Illumination**. In this model we can simply give ambient color to objects, then they become lights.
- This approach can also be approximated using lots of low-intensity point lights.
- More on this when we get to **radiosity** and **raytracing**.

Global Illumination



A GI lighting model. Scene is illuminated primarily from the two light-boxes.