Individual Assignment 4

# Introduction to Computer Graphics

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

last revision March 9, 2014

**Abstract**

We will discuss the principles behind light models by taking a look at a specific light model; namely, the Phong lighting model.

The reader is not expected to have any prior knowledge of the material presented. For the code the reader is expected to have some basic understanding of linear algebra and the C++ language.

## Contents

# 1 Theory

A lighting model describes the way in which we interpret how light bounces of object surfaces in a simulated environment. In this document we will discuss the Phong lighting model.

The Phong lighting model has three terms; an ambient term ($R_a$) which describes the minimal amount of emitted reflection bouncing of any surface, a diffuse term ($R_d$) which describes the surface scattered light bouncing of surfaces which get hit by a light source, and lastly a specular term ($R_s$) which describes the light emitted by surface, which in relation to the light source, reflects light directly at the viewer making the surfaces appear very bright and shiny.

$$I_p = \underbrace{k_a l_a}_{R_a} + \sum_{l \in \text{lights}} \underbrace{k_d l_d (L \cdot N)}_{R_d} + \sum_{l \in \text{lights}} \underbrace{k_s l_s (R \cdot V)^n}_{R_s} \tag{1}$$

Figure 1: The Phong lighting model.

The above equation describes the general Phong lighting model with an arbitrary number of lights in the scene — hence the summation. The variables $k$ denote the emissivity of a particular term. That is, *how much* a given term will reflect of the received light. These values are clamped at the interval $k \in [0; 1]$. The variables $l$ denote the light intensities of a given term.

## 1.1 Ambient

The ambient term of the Phong lighting model is very simple as it omits any concern of how the surface relates to the light source. By omitting such detail we regard the light reflection as being equal on any part of a surface.

$$R_a = k_a l_a \tag{2}$$

Figure 2: The phong ambient term.

Since ambient light is equally distributed among all surfaces we need not consider how the light hits the surface.

## 1.2 Diffuse

The diffuse term of the Phong lighting model has to take into account how the light hits the surface.

Consider a surface with normal $N$ getting hit by a light source at position $l_p$ with intensity $l_i$ from the direction unit vector $L$ and the direction unit vector $V$ toward the viewer. The light reflected off of the surface toward the viewer, by Lambert's law, is $k_d l_i \cos\theta = k_d l_i (L \cdot N)$

$$R_d = k_d l_d \cos\theta = k_d l_d (L \cdot N) \tag{3}$$

Figure 3: The phong diffuse term.

If there are multiple lights in the scene, one can extend this by a summation of the lights as done in the general equation (see figure 1)

### 1.3 Specular

The specular term of the Phong lighting model is slightly more involved as specular highlights are reflected in a small cone around the vector $V$. Consider such a reflection cone defined by the unit vector $R$ at an angle $\alpha$ from vector $V$. The emitted light at vector $R$ is the same as that going in from the light at vector $L$, the angle of which is $\theta$. The cone $R$ revolving around $V$ can then be expressed as $k_s l_s (R \cdot V)$. Adding to that the falloff, or *shininess*, we can take the $n$th power of the cosine, which reduces the cone angle $\alpha$, which translates to the same by putting $R \cdot V$ to the $n$th power.

$$R_s = k_s l_s \cos^n \alpha = k_s l_s (R \cdot V)^n \tag{4}$$

Figure 4: The Phong specular term.

### 1.4 Attenuation

From physics, we can describe a lights falloff rate by the square of the distance. That is, the attenuation falloff is given by $f_{att} = \frac{1}{d_L^2}$. We can add this to the general Phong lighting model as shown below.

$$I_p = \underbrace{k_a l_a}_{R_a} + \sum_{l \in \text{lights}} f_{att} \underbrace{k_d l_d (L \cdot N)}_{R_d} + \sum_{l \in \text{lights}} f_{att} \underbrace{k_s l_s (R \cdot V)^n}_{R_s} \tag{5}$$

Figure 5: The Phong lighting model with attenuation.

This will effectively make distant objects appear darker than those closer to a light source, which is a closer approximation of how real object surfaces reflect light.

### 1.5 Surface Colors

Now that we have a model for which we can describe how much light is reflected off of a surface, we can turn our attention toward what color the surface reflects and how to add it to our equation.

A surface reflection coefficients $k$ in conjunction with surface reflection colors $O$ form the notion *surface material*. Adding these to our equation, we get the following.

$$I_p = \underbrace{k_a O_a l_a}_{R_a} + \sum_{l \in \text{lights}} f_{att} \underbrace{k_d O_a l_d (L \cdot N)}_{R_d} + \sum_{l \in \text{lights}} f_{att} \underbrace{k_s O_s l_s (R \cdot V)^n}_{R_s} \tag{6}$$

Figure 6: The Phong lighting model with surface colors.

### 1.6 Remarks

The equation we've arrived at (6) can be reduced to form a more simple expression that is more easily readable. We can pull out anything not directly affected by the lights, such as coefficients and colors, etc. Also, we will employ a shorthand for the surface materials. We will express these by $k_{q,\lambda} = k_q O_q$, meaning the reflectiveness of surface property $q$, including the coefficient and color.

$$I_p = \underbrace{k_a O_a l_a}_{R_a} + f_{att} \left( k_{d,\lambda} \sum_{l \in \text{lights}} l_d (L_l \cdot N)_{R_d} + k_{s,\lambda} \sum_{l \in \text{lights}} l_s (R_l \cdot V)^n_{R_s} \right) \tag{7}$$

Figure 7: The Phong lighting model with surface colors.

Doing so, we get a much nicer equation that is easy to read. We could reduce it further by having a single sum, but this, I think, crams the equation.

## 2 Implementation

The implementation has a few bugs in it. I intended to write a per-fragment Phong shader, but it seems that the results are not as expected. I wasn't able to find the error in time for the deadline, but the shader does indeed produce Phong shading, just not per-fragment, I think — again, I'm not sure as to the source of why it looks like it produces per-vertex result.

### 2.1 Vertex Shader

The vertex shader is given the 3 attributes; `vtx`, `normal` and `tex`, being the vertex, normal and texture coord attributes. The main program has been altered since the last assignment to support `OBJ` file loading, which can have all those properties, hence they are present in the shader as well (lines 3–5).

Because of some issues with the shader, I had to split up the premultiplied model-view-projection matrix into their separate components and send each one down to the shader (lines 7–9).

On line 11 we have a uniform for the light position, and line 13 is a uniform for the shininess factor. Lines 15–16 are varying floats that are passed on to the fragment shader.

```
1  #version 110
2
3  attribute vec3 vtx;
4  attribute vec3 normal;
5  attribute vec3 tex;
6
7  uniform mat4 m;
8  uniform mat4 v;
9  uniform mat4 p;
10
11 uniform vec3 Lp;
12
13 uniform float n;
14
15 varying float dotNL;
16 varying float spec;
```

Figure 8: Code excerpt showing the global variables of the vertex shader.

On lines 20–21 we compute the eye-coordinate space vectors of the vertex and normal. On line 23 we compute the $L$ vector, and on the following line 24 we compute $L_l \cdot N$. On line 26 we compute the $R$ vector, using a helper function from GLSL, and the $V$ vector on line 27. Then on line 28 $R_l \cdot V$ is computed, and the specularity is computed on line 29.

Lastly, the position of the vertex in screen space is computed on line 31.

```
18 void main() {
19
20     vec3 p_eye = vec3(v * m * vec4(vtx, 1.0));
21     vec3 n_eye = vec3(v * m * vec4(normal, 1.0));
22
23     vec3 L = normalize(vec3(v * vec4(Lp, 1.0)) - p_eye);
24     dotNL = max(dot(normal, L), 0.0);
25
26     vec3 R = reflect(-L,normal);
27     vec3 V = normalize(-p_eye);
28     float dotRV = max(dot(R, V), 0.0);
29     spec = pow(dotRV, n);
30
31     gl_Position = p * v * m * vec4(vtx, 1.0);
32 }
```

Figure 9: Code excerpt showing the vertex shader progam.

### 2.2 Fragment Shader

In the fragment shader, since most of the work has been hoisted up the pipeline —improving the efficiency—, lines 3–14 are merely the uniforms we send to the shader, and lines 16–17 are the varying floats we passed from the vertex shader.

```
1  #version 110
2
3  uniform vec3 Ai;
4  uniform vec3 Li;
5
6  uniform vec3 Oa;
7  uniform vec3 Od;
8  uniform vec3 Os;
9
10 uniform float Ka;
11 uniform float Kd;
12 uniform float Ks;
13
14 uniform float Fatt;
15
16 varying float dotNL;
17 varying float spec;
18
19 void main() {
20
21   vec3 R_a = Ai * Oa * Ka;
22   vec3 R_d = Li * Od * Kd * dotNL;
23   vec3 R_s = Li * Os * Ks * spec;
24
25     gl_FragColor = vec4(R_a + R_d + R_s, 1.0);
26 }
```

Figure 10: Code excerpt showing the fragment shader.

The $R_a$, $R_d$ and $R_s$ are calculated on lines 21–23, and summed on line 25.

### 2.3 Tests

The program source code allows for moving and rotating a teapot object, which is being loaded from an `OBJ`-file. This should allow for easy verification of the test screenshots provided below.

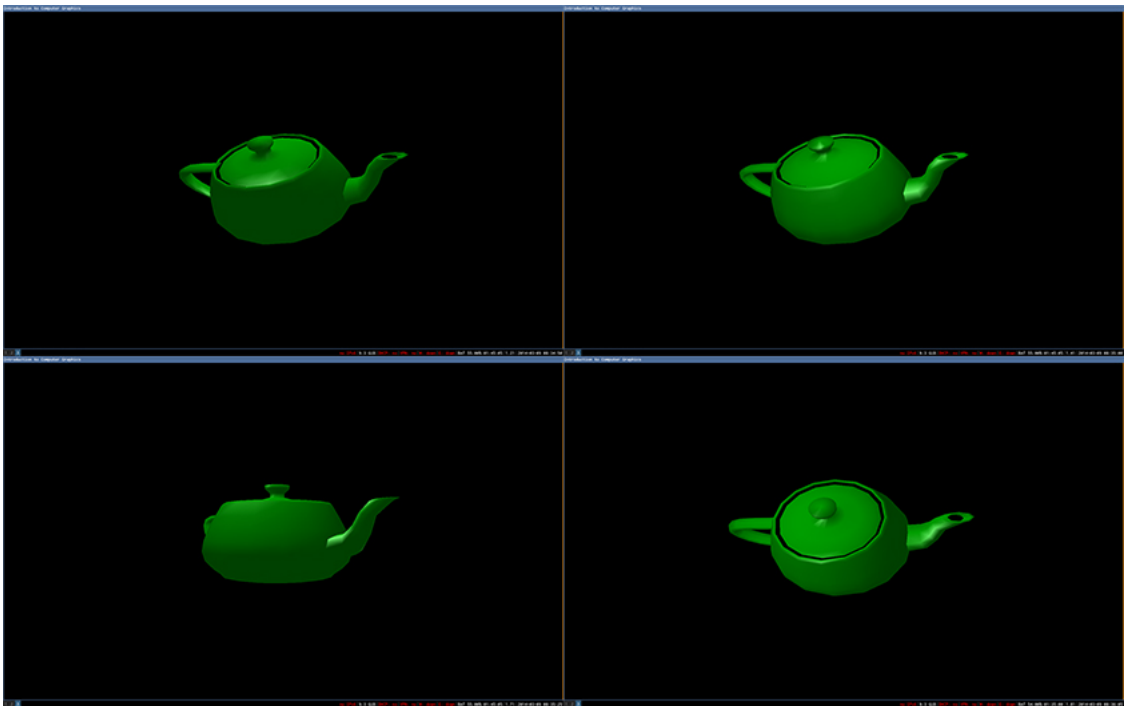

Figure 11: Screenshots showing the Phong shader on a teapot.