Introduction to Computer Graphics & Lighting

Petridis Konstantinos

Computer Graphics @ ECE AUTh, June 2022

Abstract

Having already discussed about **Triangle Filling**, **Affine Transformations** and **Projection**, it is time to provide some insights on the **Lighting** models used by a graphics system. This paper includes detailed explanation on the proper manipulation of lighting, particularly the **Phong Reflection Model**, in order to achieve the desired visual effect.

1. Introduction

In simple terms, the purpose of this project is to demonstrate the process of calculating the colors of every pixel, which were taken as granted in the preceding projects. The light model utilized to do that is the **Phong Light Model**, which takes into consideration three light components:

- **Ambient light**, which is basically relates to the light inside the surrounding environment, creating a uniform light level.

  $$I_{ambient} = k_a I_a \quad (1)$$

  with k_a, I_a being the ambient coefficient and the intensity of the light source for the ambient component respectively.

- **Diffuse light**, which corresponds to light, scattered inside the environment and is dependent on the position of the light source.

  $$I_{diffuse} = k_d I_d \cos a \quad (2)$$

  $$\cos a = < \hat{N}, \hat{L} >$$

  $$\hat{L} = \frac{\boldsymbol{CP}}{|\boldsymbol{CP}|}$$

  C being the position on which the light is being calculated and P the position of the light source. $\hat{N}$ is the **normal vector** as computed on the position of interest.

- **Specular light**, which expresses the light reflected by the object back to the camera.

  $$I_{specular} = k_s I_s \cos\theta^n \quad (3)$$

  $$\cos\theta = < 2\hat{N} < \hat{N}, \hat{L} > - \hat{L}, \hat{V} >$$

  with N, L as above and V being the vector from the camera to the point of interest. n is the Phong parameter.

  For an arbitrary pixel, and an initial RGB color value, the final color c is given by the **Phong** light formula:

  $$c = I_{ambient} + cI_{diffuse} + cI_{specular} \quad (4)$$

  The basic difference between the **Phong Model** and regular **Gouraud Shading** techniques is that, in **Phong Light Model** we only use the normal vectors and not the actual colors to compute a new color. For every point of the object, the normal vector is calculated and using formula (4) we obtain the RGB value for the current pixel.

2. Problem Definition

Given the matrices below:

- **verts3d**: $L \times 3$ matrix containing the 3D coordinates of every vertex. L vertices in total.
- **vcolors**: $L \times 3$ matrix containing the partially computed RGB colors of every vertex. L vertices in total.
- **faces**: $K \times 3$ matrix containing the indices from the vertices of every triangle. K triangles in total.

and the **coordinates** and **pointing axis** of the **camera**, use the **Phong Light Model** to render the object and illustrate the visual effect of each individual light component, and finally all of them combined.

3. Problem Decomposition

3.1. Normal Vector Calculation

Firstly, the **normal vectors** on every point of the object should be obtained. The idea is that, initially we need the normal vectors on the center of every triangle. This can easily be computed for an arbitrary triangle t with vertices $\hat{ABC}$ as

$$\hat{N}_{center,t} = \frac{\boldsymbol{AB} \times \boldsymbol{AC}}{|\boldsymbol{AB} \times \boldsymbol{AC}|}$$

Given the normal vectors of every triangle, we can now compute the normal vector on every point i of the object as:

$$\hat{N}_{vertex,i} = \sum_k \hat{N}_{center,k}$$

where k iterates over all the neighboring triangles, which are by default, incident to vertex i. To avoid redundant computations, instead of searching the incident triangles of every vertex, we perform the calculations incrementally. Specifically, after getting the normal of a triangle center, we just add it to all three vertices of the triangle, to obtain the partially computed normal of its vertices. In the end all vertices will be holding the sum of the normals, from their incident triangles. Finally, we have to normalize once again, since the addition of normal vectors do not result in normal vectors. The described process is given in *Algorithm 1.*

Algorithm 1 Normal Vector Calculation

```
1:  procedure CALCULATENORMALS
2:      input : $verts$, $faces$
3:      $N$ = zeros()
4:      for indices in faces do
5:          $v\_tr \leftarrow verts[indices]$
6:          $N\_tr \leftarrow cross(v\_tr[1] - v\_tr[0], v\_tr[2] - v\_tr[0])$
7:          $N\_tr \leftarrow N\_tr / norm(N\_tr)$
8:          $N[indices] \leftarrow N[indices] + N\_tr$
9:      for normal in N do
10:         $normal \leftarrow normal / norm(normal)$
11:     return $N$
```

3.2. Intermediate Steps

From the point of calculating the normal vectors, up until the triangle filling, the steps followed are identical to those from the previous projects. In more detail, the points are projected onto the 2D space using the **Projection** formula presented in our previous paper, and rasterization is applied to obtain the object, as it appears through the camera. Of course this presupposes the change of the coordinate system so that it matches the one formed by the camera axes.

3.3. Gouraud Shading

If the shader setting is set to **Gouraud**, then the same function from the previous project is called to render the object. The process is the exact same, however some pre-processing is required because in this version we don't have the colors from the start. So in order to use the gouraud shading method, we first have to compute the RGB colors of every point as the sum of three **Phong** light components and then proceed to call the gouraud renderer. The necessary steps to use gouraud shading are summarized in *Algorithm 2*

Algorithm 2 Preprocessing for Gouraud Shading

```
1: procedure SHADEGOURAUD
2:     input : $verts3d$
3:     for p = 0:1:len(verts3d) do
4:         $vcolors[p] \leftarrow Phong()$
5:     $img \leftarrow$ RENDERGOURAUD()
```

where $Phong$ calculates (4) formula.

3.4. Phong Shading

For **Phong shading** we don't need to compute the RGB colors at the start, but we will rather use the **normal vectors** as given by *Algorithm 1*, as follows. The process is at first similar to a regular gouraud shading, specifically we iterate over every triangle, a vertical scan across y axis is performed and for every scan-line y, the points belonging inside the triangle are determined, using the active points and edges. When the point that should be colored is located, instead of calculating the color as interpolation of the two active vertices' colors, we get the **interpolation** of the **normal vectors** of the two active vertices, which resulted from interpolating the normal vectors of the triangle vertices, incident to the active edges. We use the result normal vector to get the color as given by (4) formula.

3.5. Shading with Phong Light Model

This section presents the high level description of the entire process from start to finish. *Algorithm 4* combines all individual functionalities to form the final result, which was used for the experimentation.

Algorithm 3 Shading Using Phong Light Model

```
1: procedure RENDEROBJECT
2:     $normals \leftarrow$ CALCULATENORMALS()    ▷ Algorithm 1
3:     $P, D \leftarrow$ PROJECTCAMERA()
4:     $verts2d \leftarrow$ RASTERIZE()
5:     if shader == 'gouraud' then
6:         return SHADEGOURAUD()              ▷ Algorithm 2
7:     return SHADEPHONG()
```

4. Results

Now let's examine each individual light component's effect on the object's appearance.

4.1. Ambient Light

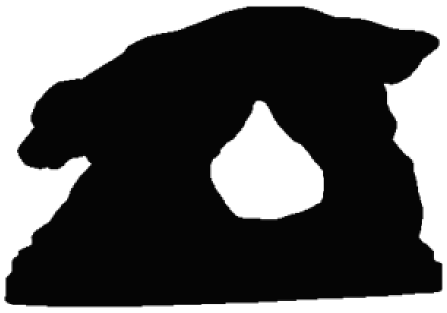

Figure 1: *Gouraud Shading*

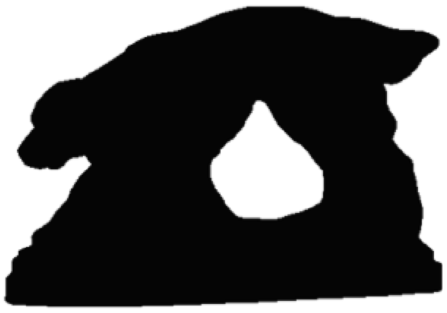

Figure 2: *Phong Shading*

4.2. Diffuse Light

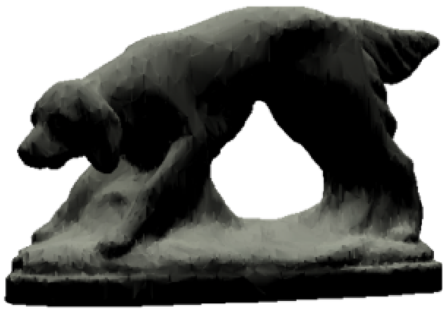

Figure 3: *Gouraud Shading*

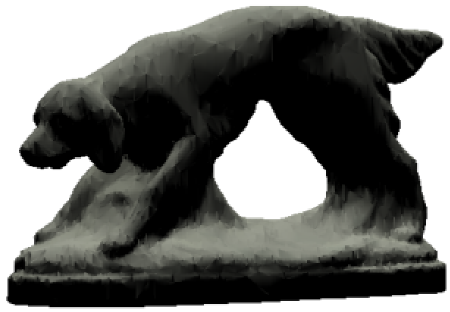
Figure 4: *Phong Shading*

4.3. Specular Light

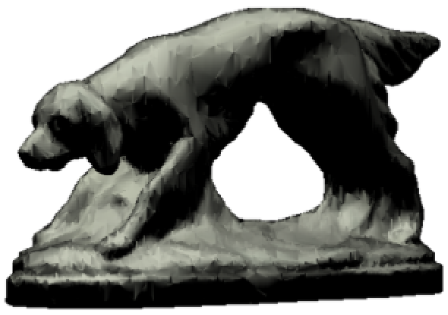
Figure 5: *Gouraud Shading*

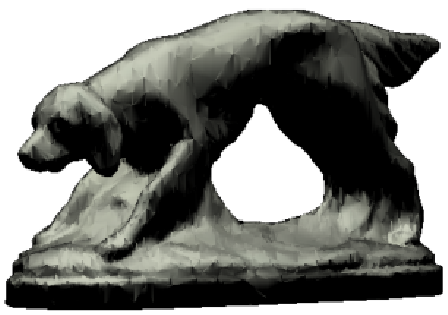
Figure 6: *Phong Shading*

4.4. Ambient & Diffuse & Specular

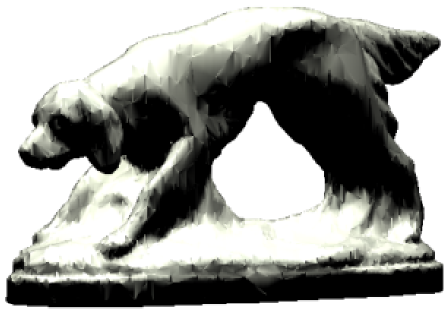
Figure 7: *Gouraud Shading*

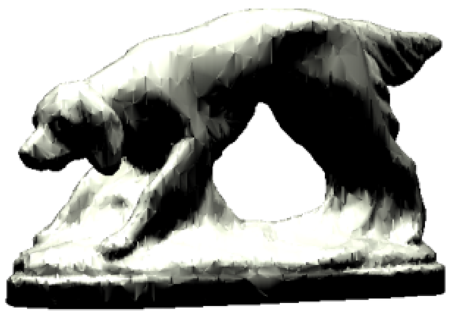
Figure 8: *Phong Shading*

5. Code Instructions

There are two **demo** files called, named **demo_phong.py** and **demo_gouraud.py**, under the 'src' directory. All helper functions are located inside the 'inc' directory. Finally, the file containing all the necessary data is placed under the 'data' directory. Everything is set to run automatically and it's not a necessity for the user to interfere. Be sure to check out the **README** for direct instructions as well. The main library used is **Numpy** for the computations and data structures that it provides. **Attention**: the object is rendered and displayed normally on screen, however there is an issue while saving it locally and it appears modified.