OpenGL Lighting

Notes for a Course in Computer Graphics
University of Minho
António Ramires
06-04-2017

1 Introduction

Lighting is an essential part of the rendering process. It improves 3D perception through the way tones vary on a surface. In here we shall provide a brief introduction to lighting, focusing mainly on real-time OpenGL lighting. We'll start by describing two illumination models, local and global, and their pros and cons. We then look at the lighting equation, and shading models, detailing a process similar to what compatibility OpenGL provides by default.

2 Illumination Models

When lighting a point in a surface, typically a pixel, we must consider at least the point being lit, and the light sources. Examples of the latter are the sun and light bulbs. When considering just these elements we are using what is called a **local model** and we are computing direct illumination.

This model is a gross simplification of the lighting phenomena. Light travels from the light sources and when it hits a surface a part is absorbed, some is reflected, and in case of transparent or translucent

objects, part is transmitted. The transmitted and reflected light keeps travelling until it hits another object, and the whole process is repeated while there is energy left. Each time a surface is hit part of the energy is absorbed, so the energy keeps decreasing with every bounce until it eventually does not contribute significantly to the final shading of the object. Since every time light hits an object some of its intensity may be reflected/transmitted light, we can consider every object in the scene as a potential light source. Furthermore, objects may occlude each other, i.e. shadows may exist. When we consider this broader notion of light source, as well as light occlusions (shadows), the illumination model is said to be **global**.

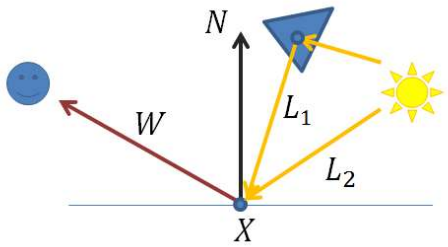

Figure 2.1 – Direct and indirect lighting

Considering Figure 2.1, the light contribution along L_2 is considered in both the local and global models, however, the contribution along L_1 is not computed under the local model.

In summary:

- **Local Lighting**: Lighting is computed based solely on the light sources, and the point being lit. This model only takes into account direct illumination.
- **Global Lighting**: Lighting computation takes into account indirect illumination, i.e. light that is reflected or transmitted from other surfaces, and occlusions.

Using global illumination we can add effects such as shadows, caustics, color bleeding, reflections and refractions, amongst others, that greatly contribute to the richness of the rendered image. On the other hand, this carries a performance penalty when compared to local illumination. As more and more effects are produced the longer it takes to render the final image, potentially turning the rendering process unsuitable for real-time graphics. Figure 2.2 below shows some of the effects mentioned before, such as shadows, diffuse reflections, specular reflections, transmission, and caustics.

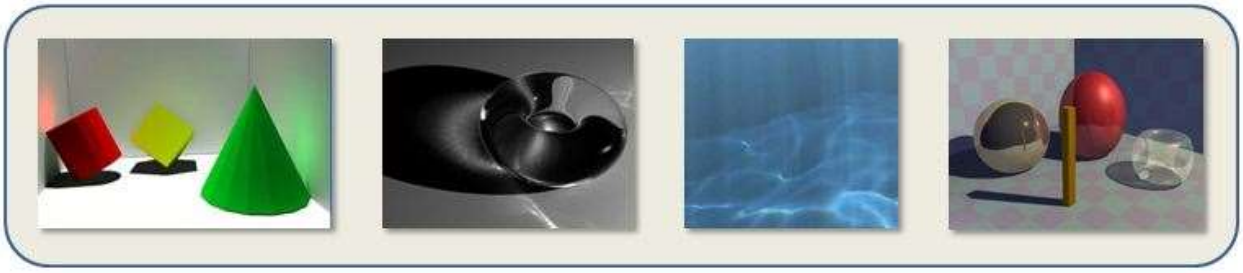

Figure 2.2- Illumination effects

3 The Lighting Equation

In here we will concentrate on the local lighting model, including the equation used by OpenGL fixed functionality.

The intensity perceived when looking at a lit object is a function of the surface's properties, the light it receives, and the camera position. The values for this function can be measured with special purpose hardware and/or encoded in a function called the Bi-directional Reflectance Distribution Function (BRDF).

Consider pure diffuse materials. These are materials that reflect light in a uniform way, regardless of the outgoing direction. The intensity of the reflected light is only a function of the intensity of the incoming light (I), the angle that the incoming light direction makes with the normal vector (a vector perpendicular to the surface), and a constant term that determines the diffuse color of the object (K_d).

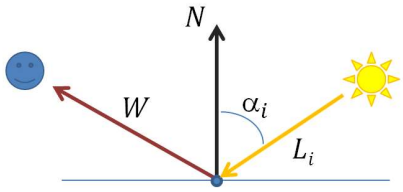

Figure 3.1) Diffuse lighting diagram

In this situation the lighting equation becomes:

$$I = \sum_i K_d \times I_i \times \cos(\alpha_i) \quad (1)$$

Equation 1) provides a local illumination solution to diffuse materials and is based on Lambert's law, which states that the intensity reflected by a purely diffuse material is proportional to the cosine of the angle between the surface normal and the incoming light direction. Note that only angles below 90 degrees are considered. For lights below the surface the result is 0.

The cosine computation can be performed with the dot product operation. The dot product between two vectors is defined as:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos(\alpha) \quad (2)$$

Hence, if both vectors are normalized, i.e. their length is 1.0, the dot product becomes an efficient way to compute the cosine of the angle between them.

Points on the surface facing away from the light will not be lit, producing black as the final color. In a global illumination model these points could receive indirect lighting. To simulate this effect in a local model, an ambient term is added to the equation. This term should add a small amount of light to every point on the surface, regardless of the lighting, thereby adding a very crude approximation, albeit an extremely fast one, to indirect illumination. Figure 3.2 shows the lighting produced with equation 1) (left), the ambient term (middle) and the final composition (right).

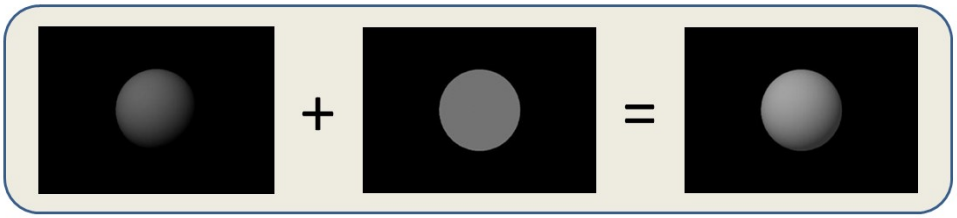

Figure 3.2 – Diffuse + ambient components

The new equation has two terms, for the diffuse and ambient components, and two new variables: K_a and I_{a_i}. The first variable relates to the ambient term of the material itself, whereas the second relates to light i contribution to the ambient illumination.

$$I = \sum_i (\boldsymbol{K_a} \times \boldsymbol{I_{a_i}} + K_d \times I_i \times \cos(\alpha_i)) \quad (3)$$

Phong introduced a term to represent specular highlights. This term is dependent not only on the incoming light's direction, but also on the viewer's position. The Phong term has its maximum in the direction R that is the light direction reflection vector regarding the surface normal N.

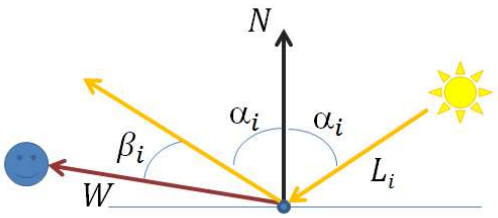

Figure 3.3- Specular component diagram

According to Phong's proposal the intensity of the light reflected on the viewer's direction (W) is computed as the intensity of the incoming light weighted with the cosine of β raised to a term called the shininess. The lighting equation becomes:

$$I = \sum_i (\mathrm{K_a} \times I_{a_i} + K_d \times I_i \times \cos(\alpha_i) + \boldsymbol{K_s} \times \boldsymbol{I_i} \times \mathbf{cos}(\boldsymbol{\beta_i})^{\boldsymbol{shininess}}) \quad (4)$$

The equation above implies the computation of the reflection vector R. Blinn proposed the usage of the half-vector, instead of the reflection vector. The half vector, is the vector that is halfway between the incoming's light direction (L) and the viewer's direction (W). It can easily be computed as the normalized sum of $-L$ and W, assuming that both vectors are normalized.

$$H = \frac{-L+W}{|-L+W|} \quad (5)$$

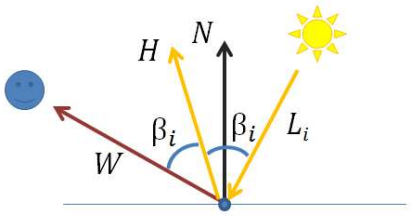

Figure 3.4 – The half-vector

With Blinn's half vector, instead of using the angle with the camera direction, the angle used is between the half vector and the normal.

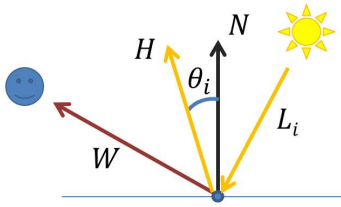

Figure 3.5 – The angle in Blinn's formula3.5

The formula, according to Blinn is:

$$I = \sum_i (\mathrm{K_a} \times Ia_i + K_d \times I_i \times \cos(\alpha_i) + \boldsymbol{K_s} \times \boldsymbol{I_i} \times \mathbf{cos}(\boldsymbol{\theta_i})^{\boldsymbol{shininess}}) \quad (6)$$

The shininess term determines the spread of the highlight. Shininess values smaller than 1.0 will flatten the curve, spreading the specular effect over a large area. Higher values will produce a more concentrated shiny spot, see Figure 3.6.

The above equation is similar to the one OpenGL uses in fixed functionality mode.

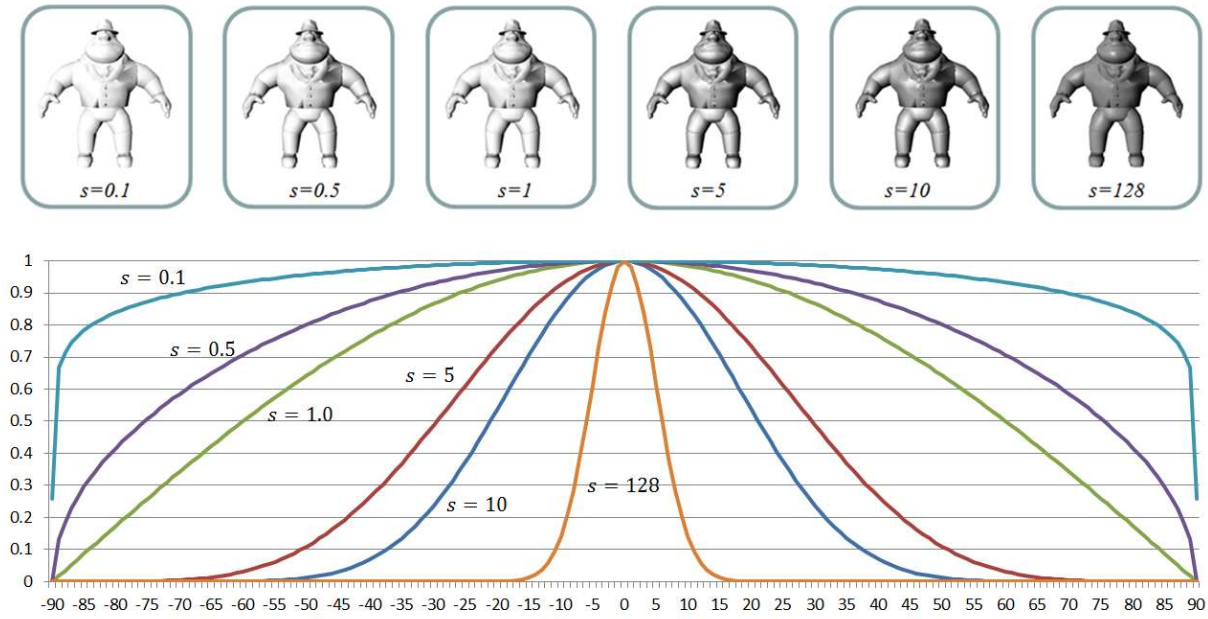

Figure 3.6 - Shininess effect

Combining all there components, ambient, diffuse, and specular, yields the following graphical result:

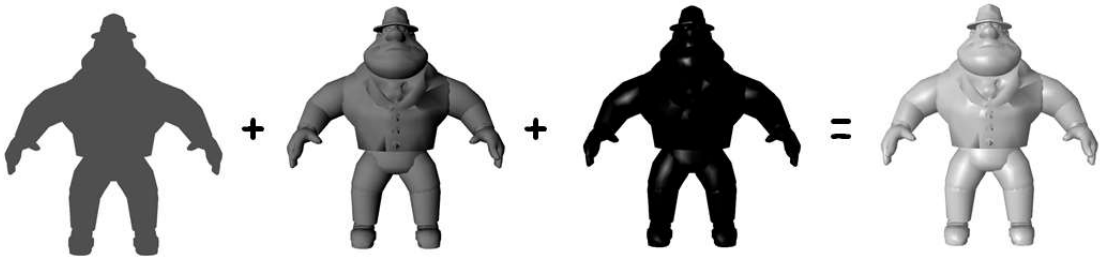

Figure 3.7 Ambient + diffuse + specular = final result

From left to right we can see the individual components applied: ambient, diffuse and specular. The end result, the last image on the right, is the sum of all components.

4 Normals and Shading Models

The triangles that make up a 3D model are projected on a 2D image plane, in a process which determines which pixels make up the projected triangle. Color must be computed for each of these pixels.

As can be seen from the lighting equation, the normal plays a crucial role in the final result. In this section, first we'll see how to determine the normal of a triangle, the basic building block of every 3D model. Afterwards, we discuss how the normal information is used according to several *shading models* to compute the color of points inside a triangle.

4.1 Computing the Normal of a Triangle

Consider the three vertices of a triangle: p_1, p_2 and p_3. Based on these points we can build two vectors as follows:

$$\overrightarrow{v_1} = p_2 - p_1 \qquad \overrightarrow{v_2} = p_3 - p_1 \tag{7}$$

The cross product of these two vectors provides a vector which is perpendicular to both, i.e. it is perpendicular to the triangle. This normal vector should be normalized, as mentioned before, to enable the efficient computation of the cosines in the lighting equation.

$$\vec{n} = \frac{\overrightarrow{v_1} \times \overrightarrow{v_2}}{|\overrightarrow{v_1} \times \overrightarrow{v_2}|} \tag{8}$$

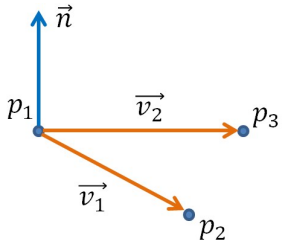

Figure 4.1 – Computing the normal vector

4.2 Flat Shading

In this shading model the lighting equation is run only once per triangle, and all its pixels are assigned the same color. The light direction is computed considering a particular point on the triangle, for instance one of its vertices, or its center. A single normal per triangle is required.

Due to its nature, triangles with different orientations can be clearly distinguished in the final rendered image, as seen in the rendering below.

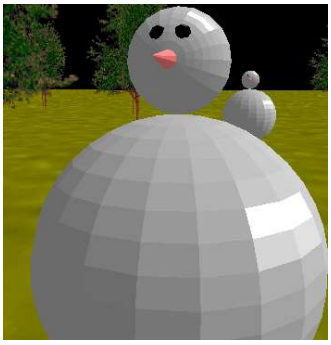

Figure 4.2 – Flat shading

This shading model only makes sense if:

- **The light is infinitely distant**, such that the light rays for every point inside the triangle are parallel, hence arrive with the same direction, implying that the cosine term is the same for every point in the triangle. This could be the case if we are considering the sun as the light source of our scene;
- **The viewer is also infinitely distant** (this could be a problem ☺), such that the specular highlight computation provides the same results for every point inside a triangle;
- The triangle geometric model is actually an **accurate representation** of the object we're trying to render (it may be the case that the snowman is actually a faceted sculpture!).

So how do we get a smoother look? A naïve solution would be to use smaller triangles in order to reduce the faceted look. However, this approach has two disadvantages:

- the higher number of triangles can have a negative impact on performance;
- the faceted look doesn't disappear unless we have a triangle per pixel, otherwise the issue can be enhanced due to the Mach band effect.

The Mach band illusion is due to the fact that our visual system accentuates changes in brightness, so that regions of constant brightness appear to have varying brightness near the edges.

Figure 4.3 – Mach band effect

As can be seen from the two strips, this effect is actually more prominent when using smaller areas.

4.3 Interpolation Shading

This shading model suggests that the light's intensity should be computed per vertex. The intensity, or color, of the points inside the triangle is obtained using interpolation. This addresses the first two assumptions of the flat shading model.

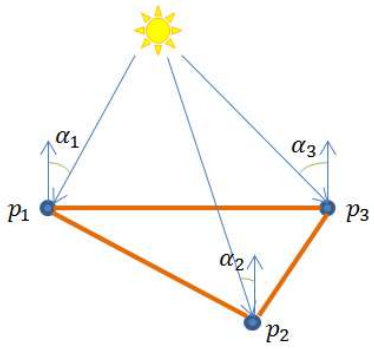

Figure 4.4 – Although the normal is common to all positions in the triangle, the angle with the light direction can be different for each vertex.

As can be seen from the picture above, each vertex has a different incoming light's direction. Hence, the intensity will be different for each vertex when considering a light that is not infinitely distant. Since the color for points inside the triangle is obtained through interpolation, this creates a gradient inside the triangle.

However, the rendered image will still show a discontinuity between adjacent triangles, i.e. triangles that share an edge, when these triangles have a different orientation. This is because the normals used at each vertex are the triangle's normal, and the two triangles will each have a different normal at the common vertices. For models with small triangles, this produces an image which is very close to flat shading when the light is relatively far from the model.

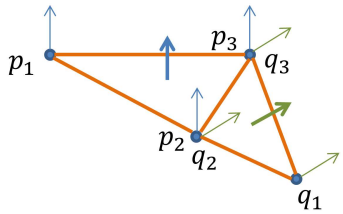

Figure 4.5 – Different normals used in the common edge can cause discontinuities

Figure 4.5 shows two triangles, $P(p_1, p_2, p_3)$ and $Q(q_1, q_2, q_3)$ sharing an edge. The shared vertices are $p_3 = q_3$ and $p_2 = q_2$. However, the normal at these vertices is different for each triangle (blue arrows represent the normal for triangle P, green for triangle Q). Therefore, adjacent pixels from different triangles will have different intensities in the general case.

4.4 Gouraud Shading

Gouraud proposed a method to represent smooth curved surfaces. The reasoning behind this proposal is based on the observation that the geometric modeling uses flat surfaces to approximate curved surfaces. Hence, if we could have at each vertex the normal that matches the normal of the surface we are trying to approximate, as opposed to the triangle's normal, the illumination would look as if we had a smooth curved model.

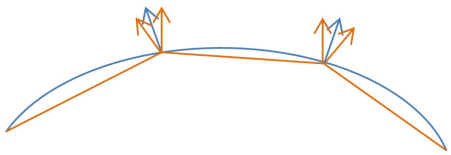

Figure 4.6 – Normals approximation the underlying surface

The figure above shows a 2D example of the concept. It shows the surface we're trying to approximate (blue) and its geometric representation (orange) using straight lines. The orange vectors represent each triangle's normal, and the blue arrows represent the curved line normal at the same positions.

Gouraud's proposal suggests the usage of the blue arrows as normals for each vertex. This implies that, in a triangle, each vertex can have a different normal. On the other hand, vertices shared amongst triangles will have the same normal vector.

If the surface we're approximating is based on a known equation, we can compute the normal analytically. This is not necessarily true for all models. Some models are just a "polygon soup", i.e. a bunch of triangles and there is no information about the underlying surface.

In this latter case the solution requires that we initially compute the normals of all triangles. Then for each vertex we compute the normal as the average of the normals of all the triangles that share the said vertex.

Once the normals are available, the intensities are computed at each vertex, and for each point inside the triangle these intensities are interpolated. This is the shading model used by default in OpenGL fixed functionality, i.e. without using shaders.

The images below show interpolation shading (left) and Gouraud shading (right).

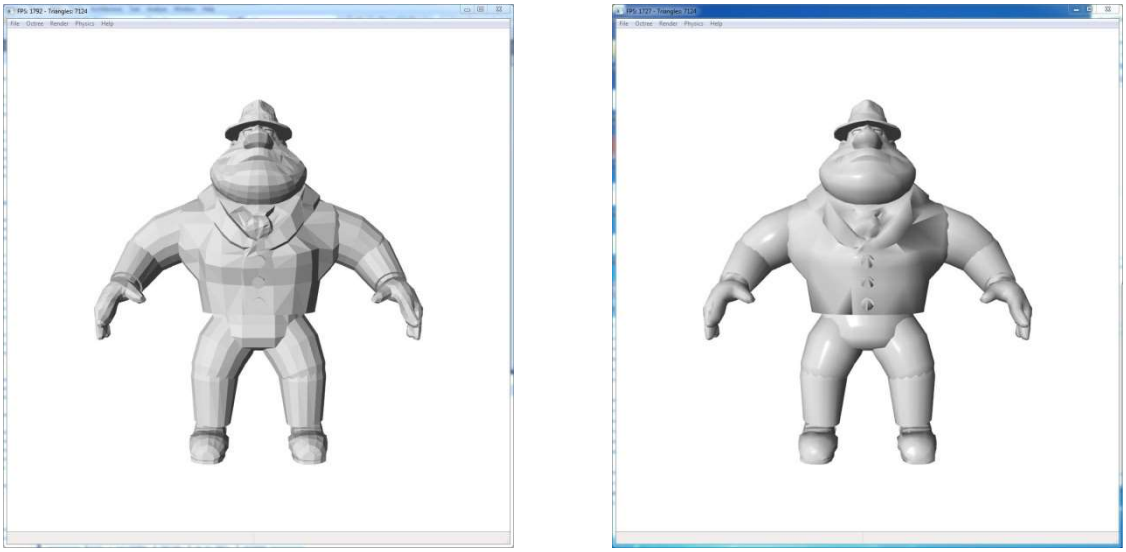

Figure 4.7 – Interpolation vs. Gouraud's shading model

As seen in Figure 4.7, using this latter approach, the discontinuities near the triangle edges do tend to disappear. Nevertheless, this shading model is not perfect. There are some issues that this model does not deal with appropriately.

For instance, what happens if one is using a light that does not reach any of the vertices, but if reaches the center of a triangle? For example, a spotlight aimed at the center of a triangle without hitting any of its vertices? Since no vertex receives light, the vertices will all have zero intensity (this is OK), but the center will also be black (this is not OK) because its intensity will be based on the interpolation of zero values.

Another issue is the interpolation itself. Note that the interpolation assumes that the area is flat, and does not take into account the curvature of the surface. The specular highlights are also an issue with this model as they are high frequency in nature, i.e. they can vary rapidly inside a triangle. Again interpolation based on the vertex intensities is far from ideal in this scenario.

4.5 Phong Shading

Phong shading builds on the proposal by Gouraud solving its issues, essentially by reordering the operations. Phong proposes a subtle change to the Gouraud shading regarding the interpolation phase.

Let's recap. Gouraud proposes that:

- **For each vertex** we **compute a normal** (either analytically or as an approximation to the underlying surface's normal) and **compute the lighting equation once per vertex**;
- **For each pixel** we **compute its intensity by interpolation**, as the weighted average of the intensities computed at vertices.

Phong suggests that, instead of computing the intensity for each vertex and interpolate these values per pixel, we should interpolate the normal itself for each pixel, and then compute the lighting equation on every pixel, with the interpolated normals.

So Phong's proposal is:

- **For each vertex compute a normal as in Gouraud**;
- **For each pixel interpolate the normal and compute the lighting equation using the interpolated normal**.

Phong shading is therefore lighter regarding vertex computation, and heavier for each pixel.

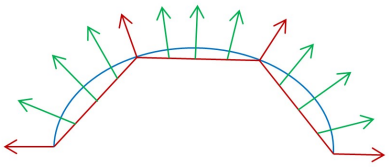

Figure 4.8 – Phong shading model interpolates normals for every inner point

The figure above shows the normals per vertex (in red) and interpolated normals (in green).

This solution is closer to what we are looking for when lighting a polygonal approximation to a curved surface. Note that interpolating intensities is not equivalent to interpolation normals and then computing intensities based on those normals.

The result is presented in Figure 4.9, Gouraud on the left, Phong on the right.

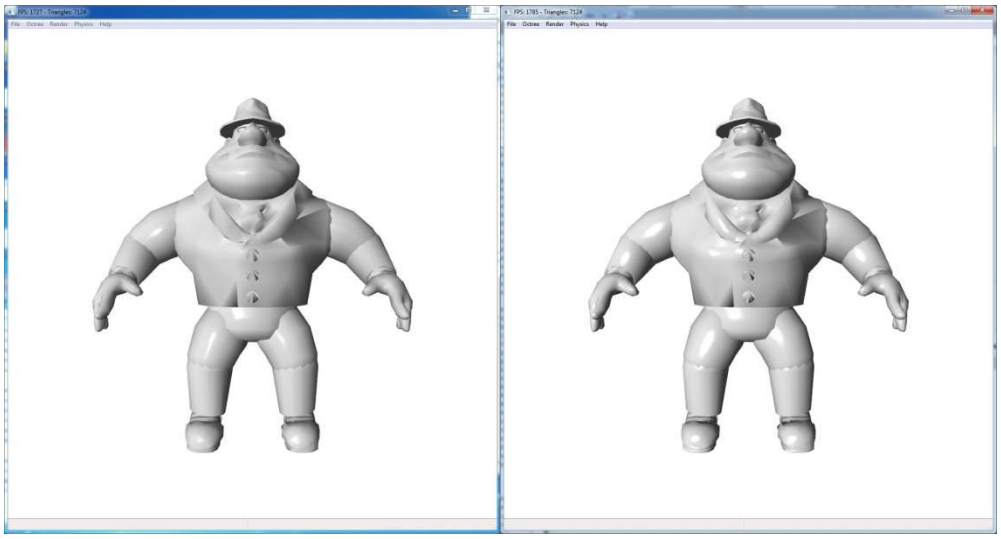

Figure 4.9 – Gouraud vs. Phong

Figure 4.10 presents a close-up of the face (left – Gouraud; right: Phong), where differences in the highlights can be clearly seen.

Figure 4.10 – Closeup of Gouraud vs. Phong

Phong model is not available in compatibility OpenGL by default. To use it we must write shaders.

4.6 Shading Model Summary

Here goes an overview of the main features of the shading models presented previously.

Flat model

Per vertex:

- Single **normal per triangle**
- Computes lighting equation **once per triangle**

Per Pixel:

- Applies the **same color** to all pixels inside the triangle

Interpolation

Per vertex:

- Single **normal per triangle**
- Computes lighting equation **once per vertex**

Per Pixel:

- **Interpolates colors** for pixels inside the triangle

Gouraud

Per vertex:

- **Per vertex normal** (approximation of the underlying surface's normal)
- Computes lighting equation **once per vertex**

Per Pixel:

- **Interpolates colors** for pixels inside the triangle

Phong

Per vertex:

- **Per vertex normal** (approximation of the underlying surface's normal)

Per Pixel:

- **Interpolates normals** for pixels inside the triangle
- Computes **lighting equation per pixel** with the interpolated normal