Praktikum Echtzeit-Computergrafik
Assignment 5 – Lighting

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Christoph Neuhauser, Simon Niedermayr – SS 24

In Assignment 3, you already used some rudimentary form of Gouraud shading. In this assignment, you will learn how to implement different local lighting models in ShaderLabWeb (https://umwestermann10.in.tum.de/). You will get to know three lighting models (Lambertian, Phong, Blinn-Phong) and two shading interpolation models (per-vertex shading/Gouraud shading and per-fragment/per-pixel shading).

Tasks

Task 1: Lambertian Shading

Lambertian shading is one of the simplest shading models based on the cosine law[1]. Lambertian shading takes into account only so-called diffuse shading. The local light intensity for Lambertian shading can be computed using the normal vector n, the vector pointing from the point on the surface to the light source l and the diffuse lighting factor k_d ($= 1$ for the Lambertian shading model) as

$$I_d = k_d \langle n, l \rangle. \tag{1}$$

Your task is to pass the world-space position, the world-space normal and camera position from the vertex shader to the fragment shader and multiply the object color with I_d.

Hints:

- In Assignment 3, we used the object space normals for shading. In this assignment, the normals need to be transformed from object space to world space using $(\texttt{mMatrix}^{-1})^T$. For this, you can use the GLSL functions `inverse`[2] and `transpose`[3].
- The camera position can be computed as $\texttt{vMatrix}^{-1} \cdot (0, 0, 0, 1)^T$. For computing the inverse of a matrix, you can use the GLSL function `inverse`. Alternatively, you can also define a `uniform vec3` variable and attach `Camera Position` to it in the `Uniforms` tab.
- For the light direction l, you can either use a point light source (i.e., specify the light position as a uniform and compute $l = \|p_{light} - p_{frag,world}\|_2$) or a directional light source (i.e., specify directly l as a uniform variable and normalize it if necessary).

[1] https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/diffuse-lambertian-shading

[2] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/inverse.xhtml

[3] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/transpose.xhtml

Note: In the standard configuration, we have $(\texttt{mMatrix}^{-1})^T = \texttt{mMatrix}$, thus we could also transform the normals with `mMatrix` when not changing, e.g., the scaling and shearing component of the matrix. However, for the sake of getting more experience with GLSL, we recommend to compute the transposed inverse nonetheless. Also please note that, when having access to CPU-side code, it would be beneficial to pre-compute the transpose and inverse operations on the CPU and upload the resulting matrix to the GPU to avoid having to compute the matrix operations in every shader invocation.

Figure 1: Result using per-fragment Lambertian shading.

Task 2: Gouraud Phong Shading

On old hardware, lighting computations were usually performed at the vertex level to improve performance. This per-vertex lighting is referred to as Gouraud shading. In this task, you will implement the Phong shading model in the vertex shader. The Phong reflection model adds three lighting contributions to get the final light intensity at a surface point.

- The constant ambient lighting term $I_a = k_a$ adding background brightness using a constant factor k_a.
- The diffuse lighting term $I_d = k_d\langle n, l\rangle$, as already used in the Lambertian shading model.
- The specular lighting term $I_s = k_s\langle r, v\rangle^s$ for the specular exponent s, the reflection vector $r = \frac{2\langle n,l\rangle n - l}{\|2\langle n,l\rangle n - l\|_2}$ of the light direction by the normal vector, and the view vector v pointing from the vertex (per-vertex/Gouraud shading) or fragment (per-fragment/per-pixel shading) position to the camera.

Usually, I_a and I_d are normalized to sum up to 1 and are multiplied with the surface and light color, while I_s is sometimes only multiplied with the light color to get specular reflections using only the color of the light. Finally, the product of the intensities and colors are summed up to get the final color. Your task is to implement the Phong model in the vertex shader stage and pass the resulting color to the fragment shader stage.

Hint: Clamp I_d and I_s to the range $[0, 1]$ using the GLSL function `clamp`, as adding negative light intensities makes no physical sense.

Figure 2: Result using per-vertex Phong shading.

Task 3: Phong Shading

Now implement Phong shading in the fragment shader stage, i.e., per-fragment/per-pixel lighting. What differences do you see compared to Gouraud Phong shading?

Figure 3: Result using per-fragment Phong shading.

Task 4: Blinn-Phong Shading

The Blinn-Phong model is an adapted form of Phong shading using a different, more realistic term for the specular lighting. Instead of the reflection vector r, it uses the halfway vector $h = \frac{l+v}{\|l+v\|_2}$, and the term $\langle r, v \rangle$ is replaced by $\langle h, n \rangle$. Your task is to implement per-fragment Blinn-Phong shading.

Hint: You may need to adapt the exponent for specular lighting to again get similar results as for Phong shading. A good heuristic is to use $s' = 4s$.

Figure 4: Result using per-fragment Blinn-Phong shading.

Task 5: Flat Shading

When passing colors (Gouraud shading) or normals (per-fragment shading) from the vertex shader stage to the fragment shader stage, the hardware interpolates the per-vertex attributes using barycentric interpolation. Flat shading is usually achieved by not sharing vertices between triangles by referencing them using the same index, but by duplicating the vertices instead. Thus, the per-vertex values are not interpolated between different triangles and we can specify flat per-triangle normals.

In this assignment, we will simulate the same effect using fragment shader derivatives. In order to get the surface normal of the triangle or quad our pixel belongs to, we need to compute the slope/tangents of the primitives in screen space using the GLSL functions `dFdx` and `dFdy`[4]. We know that the two vectors `dFdx(fragmentPositionWorld)` and `dFdy(fragmentPositionWorld)` both must be tangents of our triangle, as they are local derivatives of the fragment locations, which all lie in one plane. Thus, the vector that is orthogonal to them, which can be computed using the (normalized) cross-product, must be orthogonal to the surface, i.e., its normal vector.

Your task is to replace the normal from Task 4 using the concept described above to get flat shading.

Figure 5: Result using per-fragment Blinn-Phong shading.

[4] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/dFdx.xhtml