Lab Assignments
Computer Graphics DT3025, HT2016

Martin Magnusson and Daniel Ricão Canelhas
November 4, 2016

2 Materials

In this lab you will implement and experiment with different material models.

As a first step, you will implement the traditional Phong reflectance model (which used to be the *only* model before OpenGL 2 and Direct3D 8).

Then you will set up a more flexible framework for the appearance of materials in your fragment shader and implement more physically-based material models, and experiment with how to use them to render different kinds of materials, as well as the limits and virtues of the different models.

In this lab, you will not need to touch the vertex shader at all, but will write a lot of fragment shader code.

Learning goals

- Understanding the principles of material reflectance models.
- Understanding the effects of the parameters of common models.
- Managing normal-vector vertex attributes in OpenGL.

2.1 A Phong shader

The goal of this task is to add a basic material model to your object, and to model how light is reflected off its surfaces.

The Phong model has been immensely popular since its inception in the 1970s until today. Even if it, in its basic form, has several drawbacks, it is still widely used, so it is good to be familiar with it.

Tasks

Additional geometry setup In addition to the *position* vertex attribute that you used in Lab 1, you will now also need to specify the *normal vector* as an attribute for each vertex. Before, you were using only 12 vertices together with an index array to generate all the 20 triangular faces of the object. But now you will have to use 20×3 vertices, because the normal at a vertex depends on which face it belongs to, and a vertex cannot have more than one normal. In other words, create a new array of vertex positions that has 12×5 vertices (with 5 copies of each of your vertices from before). Update your index list to use your new vertex array to create the faces of the icosahedron. Once you have done that, loop over the faces and compute the surface normal for each.[1] Make sure that the normals are pointing outwards. (This is the only way of specifying what is the "inside" and "outside" of a triangle, and OpenGL may skip rendering triangles where only the back side is visible, for efficiency reasons.) You can

[1] See, for example, https://www.opengl.org/wiki/Calculating_a_Surface_Normal

store the normals in a `std::vector<glm::vec4>`, setting the w coordinate to 0, because it is a direction and not a point.

When setting up your VBOs, you can either have one buffer object that stores both vertex positions and normals, or you can use a separate buffer object for each vertex attribute. If you use a single VBO, you can choose between interleaving the vertex attributes, or having all the positions first followed by all the normals, for example. This can be tricky. Look closely at the documentation for `glBufferData` and `glVertexAttribPointer` to see how to specify where and in which order the attributes are stored.

Adding light sources Now that you have updated the geometry of your object, you should also instantiate one or several point light sources. Each light source should have a position and a "colour."[2] Pass them to the shader program with `uniforms`.

That should be all on the CPU side of the program.

Getting down to business In your fragment shader, set up a for loop that loops over all light sources. The loop should implement the Phong reflectance model as specified below.

$$C = C_\mathrm{a} I_\mathrm{a} + \sum_I \left(C_\mathrm{d} I_\mathrm{d} (\mathbf{n} \cdot \boldsymbol{\omega}_{\mathrm{i},I}) + C_\mathrm{s} I_\mathrm{s} (\boldsymbol{\omega}_{\mathrm{r},I} \cdot \mathbf{v})^f \right) \tag{1}$$

where $\boldsymbol{\omega}_{\mathrm{i},I}$ is the direction to light I and $\mathbf{v}$ is the direction to the camera. (This equation is a version of (27.20) in the book, but without attenuation, and using the mirror vector instead of the halfway vector.) As a starting point, you can use the following constants.

$$\begin{aligned} C_\mathrm{a} = C_\mathrm{d} &= (0.8, 0.8, 0.5)^\mathrm{T} \quad C_\mathrm{s} = (1.0, 1.0, 1.0)^\mathrm{T} \quad f = 20.0 \\ I_\mathrm{a} &= (0.2, 0.2, 0.2)^\mathrm{T} \quad I_\mathrm{d} = (0.8, 0.8, 0.8)^\mathrm{T} \quad I_\mathrm{s} = (1.0, 1.0, 1.0)^\mathrm{T} \end{aligned}$$

To compute the reflection vector $\boldsymbol{\omega}_{\mathrm{r},I}$ you can use the built-in function `reflect`, or use the expression below.

$$\boldsymbol{\omega}_\mathrm{r} = 2(\boldsymbol{\omega}_\mathrm{i} \cdot \mathbf{n})\mathbf{n} - \boldsymbol{\omega}_\mathrm{i} \tag{2}$$

Make sure that all your vectors are normalised! This can be done by the built-in GLSL function `normalize`. Also make sure that you only consider light from angles within 90° of the normal (that is, not light from behind the surface). In other words, check that the dot products are not negative. Otherwise you will not get the expected result.

The result should look similar to Figure 1.

✏ What can you say about the relation between I_a, I_d, and I_s? Are there any constraints on the relative values of these colours?

2.2 Your first BRDF

The goal of this task is to implement implement a more systematic model for how light is reflected, setting the basis for more elaborate material models in Task 2.4.

[2] Later on, we will specify lights in a more physically correct way, defining how much energy they emit, but for now let's say that a bright white light has colour $[1, 1, 1]$ without a unit.

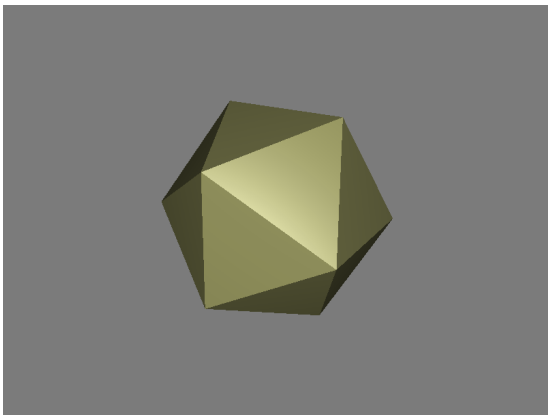

Figure 1: Glossy icosahedron from Task 2.1.

More precisely, you will implement a Lambertian BRDF, which should provide the effect of "flat"/matte paint.

A central part of the general rendering equation, which will be covered later in the course, is the following equation that describes how much light is reflected off a surface in a specific direction as a function of the incoming light and the surface normal.

$$L_{\text{out}}(\boldsymbol{\omega}_\text{o}) = \underbrace{f_\text{r}(\boldsymbol{\omega}_\text{i}, \boldsymbol{\omega}_\text{o})}_{\text{BRDF}} \cdot \overbrace{L_\text{in}(\boldsymbol{\omega}_\text{i})}^{\text{incoming irradiance}} \cdot \underbrace{(\boldsymbol{\omega}_\text{i} \cdot \mathbf{n})}_{\text{incident angle}} \tag{3}$$

In the above equation, $\boldsymbol{\omega}_\text{i}$ and $\boldsymbol{\omega}_\text{o}$ are the direction from a surface point to the light source and the camera, $\mathbf{n}$ is the surface normal at the point, and f_r is the BRDF at the point.

For now (until we get to Lab 4) we will approximate the full rendering equation with summing up the output of one such equation per light source.

Tasks

Replace the fragment shader from Task 2.1 with the skeleton `lab2-2_fs.glsl`, with a for loop that adds the contribution of each light source by summing up one instance of Equation 3 for each light source. In other words, multiply the light that comes from the light source with the BRDF at this fragment's position and the cosine between the normal vector and the unit vector pointing to the light. The only output direction $\boldsymbol{\omega}_\text{o}$ of interest is towards the camera. Just as in Task 2.1, remember to normalise your vectors and consider the sign of the cosine of the incident angle.

The fragment shader in the template code includes an empty BRDF evaluation function that takes three arguments and returns a colour. All of the BRDF functions to be implemented in this lab use that interface. (The vertex shader will not have to be changed in this lab. You can copy your final vertex shader from Task 1.)

Refer to Chapter 14.9 in the book for guidance on how to implement this task.

Figure 2: A metallic Stanford bunny with multicoloured lights.

Questions

✏ • What value does your BRDF return? Why?

✏ • Does the result look as you expected? Compare the appearance to the result you had in Task 2.1. Can you explain and motivate the difference?

2.3 Loading a more detailed object

It might be difficult to judge the difference in shading and highlights with an object with perfectly flat sides, like the icosahedron that you have used so far. Update your code to load a more detailed 3D model from a file. To do this, you can use `tiny_obj_loader.h` that is provided with the skeleton code. In the lab directory (under common/data) there is also a model of the Stanford bunny[3] You can use this, or some other model in Wavefront's .obj file format. Note that you might have to scale up the bunny, or move your camera closer, to see it better.

This task does not have to be demonstrated or covered in the report.

2.4 Physically-based reflection models

While the phenomenological Phong model of Task 2.1 is flexible enough to allow for modelling several different materials, everything tends to look rather like plastic. It falls short on realistically rendering metallic and ceramic surfaces; and in particular, it is not physically correct, which will be an important drawback in Lab 4.

In this task you will also implement one or several physically-based reflection models.

Figure 3: A few examples of materials with different scattering properties. Metallic and glossy dark plastic dice, a paper crane, a velvet dress. How can these be modelled in a shader?

Tasks

Add a new function to your fragment shader that implements Cook-Torrance reflection instead of Phong. The result might look like Figure 2.

Refer to Chapter 27.8.2–3 (and also the beginning of Chapter 27.8). When implementing these functions, be aware that the book contains a couple of errors. Refer to the errata sheet at http://cgpp.net. (In particular, there is an extra π in Equation 27.36 and R_F is sometimes written F_R instead on page 728.)

Additional tasks for grade 4 In order to get grade 4, you should also implement Oren-Nayar (Chapter 27.8.4).

When implementing Oren-Nayar, note that Equation 27.40 is the *radiance* function of the Oren-Nayar model, which is different from the BRDF. E_0 is the irradiance, and $\cos(\theta_i)$ is the incident angle, and both of these factors are multiplied with the BRDF in your main loop. (There is also an error in the book, where ϕ should be 0 in the first example on page 733.)

Additional tasks for grade 5 In order to get grade 5, you should also implement a normalised Blinn-Phong shader (Chapter 27.5.3), and provide a systematic *evaluation and comparison* of the classic Phong model, Blinn-Phong, Cook-Torrance and Oren-Nayar. Cover the following points in your report.

- Explore different parameters for the models.
- Try to mimic the appearance of some real-world objects (see Figure 3 for examples.)
- What are the limits of the models, in terms of which materials they can mimic?
- What are sensible ranges for the parameters? (Sensible as in meaningful for representing a physical material.)

The Oren-Nayar roughness parameter σ^2 is modelled from the variance of a statistical *normal distribution*. As a rule of thumb, normal-distributed variables are said to be distributed on the interval $[-3\sigma, 3\sigma]$.

- What effect does that have for very rough materials?
- What is a sensible limit for the σ^2 roughness parameter?

[3] From the Stanford 3D Scanning Repository, along with a history of the bunny.

Assessment criteria

The criteria for passing this lab with grade 3 are the same as for passing Lab 1. The criteria for grade 4 and 5 are given in Task 2.4.

Deadline

The deadline is **November 20** at noon. If you hand in your report after this, it will still be graded, but only when there is time.