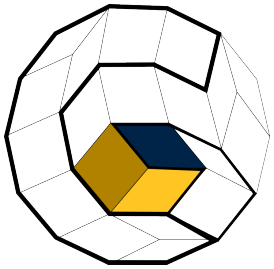
TU BRAUNSCHWEIG
Dr.-Ing. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)

May 10, 2012



Echtzeit Computergraphik SS 2012
Assignment 5

Present your solution to this exercise on Monday, May 21st, 2012.

The exercises are going to take place in the CIP pool, room G40 in Mühlenpfordstraße 23. Please make sure your solutions compile and run on the CIP pool computers. Note that you need a y-number, which can be obtained at the Gauß-IT-Zentrum, to use the computers. If for some reason you are not able to attend the exercise, you may send your solution to rogge@cg.tu-bs.de instead.

In this assignment you will start using GLSL to program custom shaders. The shader program of this exercise will use a per-fragment-lighting approach to render specified materials under variable lighting conditions.

5.1 Lights and Material in GLSL (40 Points)

In this tack, you will have to implement the missing shader code of the provided shader files `material_and_light.vert` and `material_and_light.frag`.
In the vertex shader file, there are already two structs defined. `LightSource` encloses all important parameters of a point light source, being its position and the color value for the ambient, diffuse and specular light components. The material struct `Material` holds all important material parameters, being the color value for the ambient, diffuse and specular components as well as the shininess exponent (k in the lecture), controlling the glossyness of the rendered object.
The very first thing you want to do in your shader code, is, to define output variables, which will be passed to the fragment program after execution of the vertex program. The needed components are:

- The properly transformed vertex normal
- A vector from the current vertex towards the camera position
- A vector from the current vertex towards the light source position
- The ambient color component for this vertex
- The diffuse color component
- The specular color component
- The shininess exponent

These values will be passed to the fragment shader and will be interpolated for every fragment.
Within the `main()` method of the vertex shader you will need to precompute the combined color components of the provided light source and the material. Simply multiply the color components and assign them to your previously defined output variables.
Transform the current vertex using the provided matrices and assign the final vector to the inbuild variable `gl_Position`.
To properly transform the vertex normal, invert and transpose the given `modelview` matrix and assign the transformed vertex to your vertex normal output variable.

Two more thing left to compute. In the fragment program we will need information about the vectors between pixel and light source and between pixel and camera position. Therefor compute these vectors given the current vertex position, because the resulting vectors will be interpolated properly for every fragment later on. Compute these vectors as presented in the lecture.

In the fragment program you will need to implement the per fragment lighting based on these precomputed values. Start by defining the necessary input variables. These are exactly the same as in your vertex program, except that here you will need to use the `in` qualifier.
Implement the per-fragment-lighting in the `main()` of your fragment program. First normalize all interpolated vectors, which may have been deformed due to the linear interpolation.
Compute the half-vector H used to compute the specular appearance of the material.
Using all given input vectors and colors, compute the different color components for this fragment, being an ambient component, a diffuse component and a specular component. We do not use any emissive material component just jet. Finally combine these color components for the final fragment color and assign it to the provided output variable `color`. Note that the color has a fourth component, being the alpha value. Since we do not use transparency, simply set this value to 1.

No that you have completed the shader code, we will need to provide the data for lights and materials in your OpenGL code. Complete the missing parts of `initShader()`, `initScene()`, and `renderScene()`.

Within the first method you need to get the locations of the uniforms defined in your shader program. The given code already provides you with a `std::map` named `uniformLocations`, allowing for convenient access to the uniform locations later on. For every uniform location defined in your shader code, insert its location value into this map by assigning it a proper name. Since we are using structs in the shader code, you need to get the uniform location not for the whole struct but for every component within the struct. Having a shader code like this

```
#version 330
...
struct MyStruct {
  vec3 value_0;
  ...
  float value_N;
};
uniform MyStruct myUniform;
...
void main() { ... }
```

you can retrieve the proper location of the first component in that struct by requesting the uniform location of `myUniform.value_0`.
Insert all uniform locations into the map and give them proper names.

Within `initScene()` you will need to define the available point light sources and materials.
The given code already provides vectors for the used lights and materials, being `lights` resp. `materials`. As used in the shader code, the c++ code also provides structs for light and material definitions. For simplicity the syntax is kept the same. Create the following lights and materials and insert them into the vectors.

Lights:

L1: Ambient: dark gray (~ 0.1); Diffuse, Specular: white; Position: $(4, 4, 4)^T$

L2: Ambient: dark blue; Diffuse, Specular: light blue; Position: $(0, 2, 0)^T$

Materials:

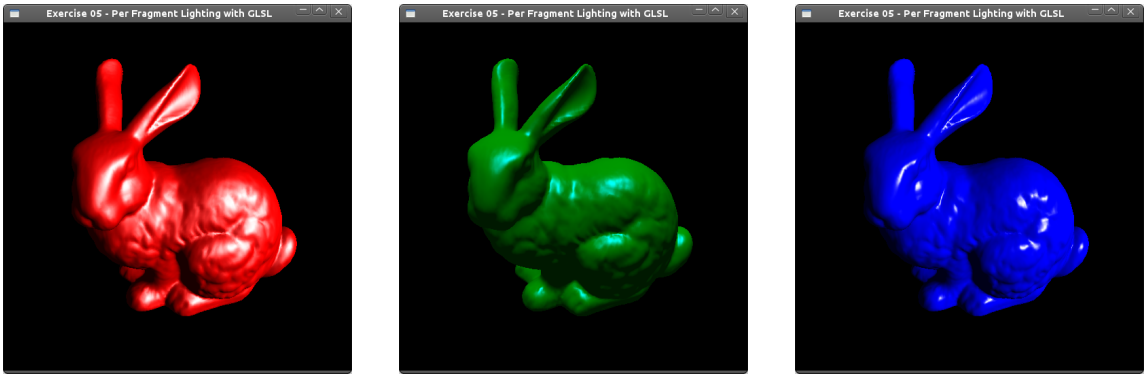
M1: Ambient, Diffuse: red; Specular: white, soft reflection

M2: Ambient, Diffuse: green; Specular: white, medium reflection

M3: Ambient, Diffuse: blue; Specular: white, sharp reflection

M4: Ambient, Diffuse: bright yellow; Specular: white, very soft reflection

The keyboard bindings M and L allow to cycle through all defined materials resp. light sources. The currently chosen material is defined by the variable `materialIndex` and the currently active light source can be identified through `lightIndex`.
As the last step, upload all information of the active light source and material to your shader program. Using the given light and material vectors and the provided index variable, your can easily access the currently active parameters. For example, use `materials[materialIndex].diffuse_color` to access the diffuse component of the currently active material. Upload all necessary information of the light source and the material to your shader. When uploading these values as vectors, you need to use `glm::value_ptr(...)` to get a proper reference to the data of a `glm::vec` vector.
When you finished completing the code within `ex05.cpp` you should be able to create views like the ones in figure 1.



(a) Light L1, Material M1 (b) Light L2, Material M2 (c) Light L1, Material M3

Figure 1: Exemplar renderings using different materials and lights.