MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session VII: Lighting

Konstantin Tretyakov, Ilya Kuzovkin

October 21, 2013

Today we shall study the basics of OpenGL lighting, and the use of fragment shaders. As usual, the base code is provided in the `practice07.zip` archive on the course website or via the course github page. Download, unpack and open it. You will need to submit your solution as a zipped archive file.

1 OpenGL pre-3.x Lighting

As you should remember from the lecture, OpenGL versions prior to 3.1 provide a bunch of convenient functions to implement the *per-vertex Phong lighting* model. To use it you have to do the following:

1. Turn on lighting computations using `glEnable(GL_LIGHTING)`, and one or more light sources using `glEnable(GL_LIGHT`x`)`,
2. Configure position/direction for each light source using `glLightfv(GL_LIGHT`x`, GL_POSITION, ...)`. Note that it is important what coordinate frame this command is executed in, as the provided coordinates are transformed using the model-view matrix.
3. Configure *ambient*, *diffuse* and *specular* light emission parameters of the Phong model for each light source using `glLightfv(GL_LIGHT`x`, GL_AMBIENT, ...)`, etc. For the ambient light it usually more reasonale to specify just a single overall *ambient* term via `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_light)` rather than provide it separately for each light source (which is possible, but makes less sense).
4. Configure *ambient*, *diffuse* and *specular* light diffusion parameters for each surface using either the `glMaterial` command or the combination of `glEnable(GL_COLOR_MATERIAL)`, `glColorMaterial` and `glColor`.

Exercise 1 (1pt). Open the project `1_Lighting`. The project renders three objects (the "ground", a "blue sphere" and a "flying cube"). Your goal is to add lighting to the scene as described below.

1. First enable lighting computations using `glEnable`. Once this is done, the scene will become dark. The `glColor` commands that were used to specify pixel color before play no role any more.

2. Now start adding lights one by one. First enable the ambient light using `glLightModel(GL_LIGHT_MODEL_AMBIENT ..)`. Let the ambient light intensity be $(0.1, 0.1, 0.1)$.

3. To have objects react to this light, you need to specify material parameters for them. You can usually do it by using the `glMaterial` function, or by enabling `GL_COLOR_MATERIAL`, and using `glColorMaterial` together with `glColor`. In this particular exercise you are requested to *not* use `GL_COLOR_MATERIAL`. Please, specify all material parameters using `glMaterial`. This will allow you to continue to the next exercise later on.

4. Specify the following material properties:

   (a) **Ground**. Specular $(0, 0, 0)$, Ambient & Diffuse: $(0.1, 1.0, 0.1)$.

   (b) **Cube**. Specular $(0, 0, 0)$, Ambient & Diffuse: $(1, 0, 0)$.

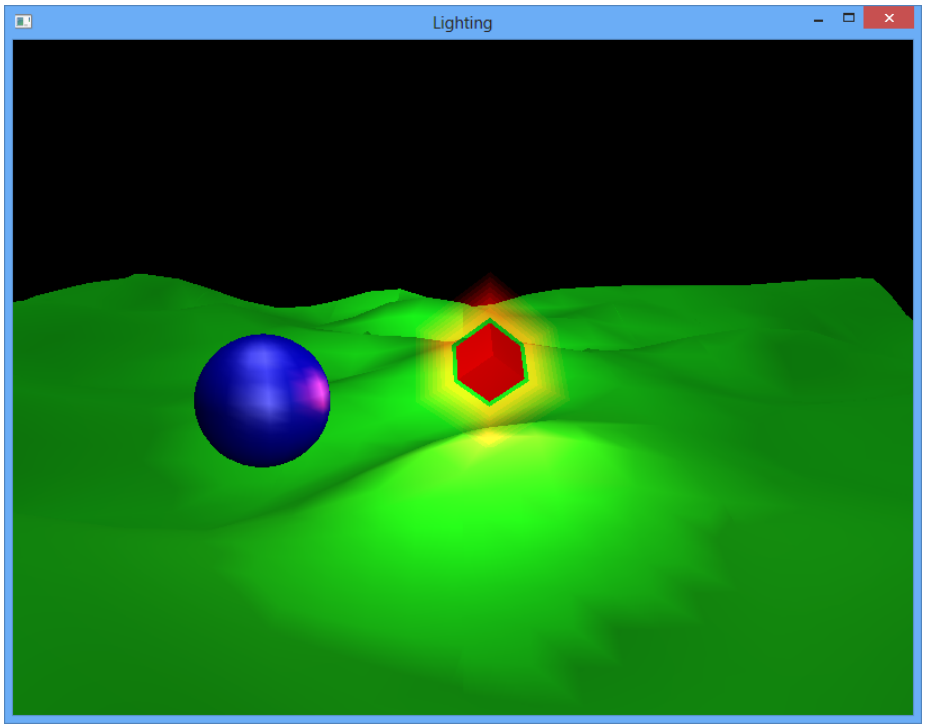
   (c) **Sphere**. Specular $(1, 1, 1)$, Ambient & Diffuse: $(0, 0, 1)$. Specular exponent (`GL_SHININESS`) 20.

5. Now let us start adding actual light sources. First add the "sun". It will be a directional light source shining straight from above the ground with diffuse/specular intensity equal to $(0.4, 0.4, 0.4)$.

6. Next, add a positional light source, located exactly at the position of the flying cube. Let the diffuse & specular intensity be $(0.5, 0.1, 0.1)$. Configure attenuation for this light source to be $\frac{1}{0.1d^2}$. (Hint: see `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, `GL_QUADRATIC_ATTENUATION`).

7. Finally, add a spotlight shining from the camera position straight forward. Set the diffuse & specular intensity to $(1.0, 1.0, 1.0)$. Set the spotlight cutoff angle (`GL_SPOT_CUTOFF`) to 20 degrees, and the spotlight fall-off exponent (`GL_SPOT_EXPONENT`) to 30.

In the end you should obtain an image like the one below.



Note that OpenGL does not do gamma correction in its lighting computations. Some graphics cards, however, can perform on-the-fly sRGB-gamma-encoding as pixels are written to the framebuffer. To switch it on, use `GLUT_SRGB` in your `glutInitDisplayMode` call and do a `glEnable(GL_FRAMEBUFFER_SRGB)`. See whether it makes any difference[1]

2 GLSL, Per-vertex and Per-fragment Lighting

Now let us try implementing the whole lighting logic manually in the shader program. We shall continue working with the `1_Lighting` project.

Exercise 2 (1pt). Uncomment the line

```
per_vertex_lighting.use();
```

which is located somewhere near the end of the `main` function. Once you do it, the built-in OpenGL lighting computations will be disabled in favor of the shader programs `lighting1_vertex.glsl` and `lighting1_fragment.glsl`. Your task is to implement the lighting computations in the shader manually, so that the result would exactly match the `GL_LIGHTING` solution you configured in the previous exercise using OpenGL's (now deprecated) fixed pipeline.

[1] E.g. my laptop does not react to `glEnable(GL_FRAMEBUFFER_SRGB)`.

You should keep the fragment shader intact. The vertex shader shows the implementation of the spotlight to get you started.

Exercise 3 (0.5pt). So far we did *per-vertex shading*, i.e. performing lighting computations for each vertex and interpolating the resulting colors. The result looks nice, but you can clearly see some ugly artifacts, in particular in how grainy the "ground" is lit. Let us now do per-fragment shading.
Comment out the line `per_vertex_lighting.use();` that you used in the previous exercise, and instead uncomment the next line:

```
per_fragment_lighting.use();
```

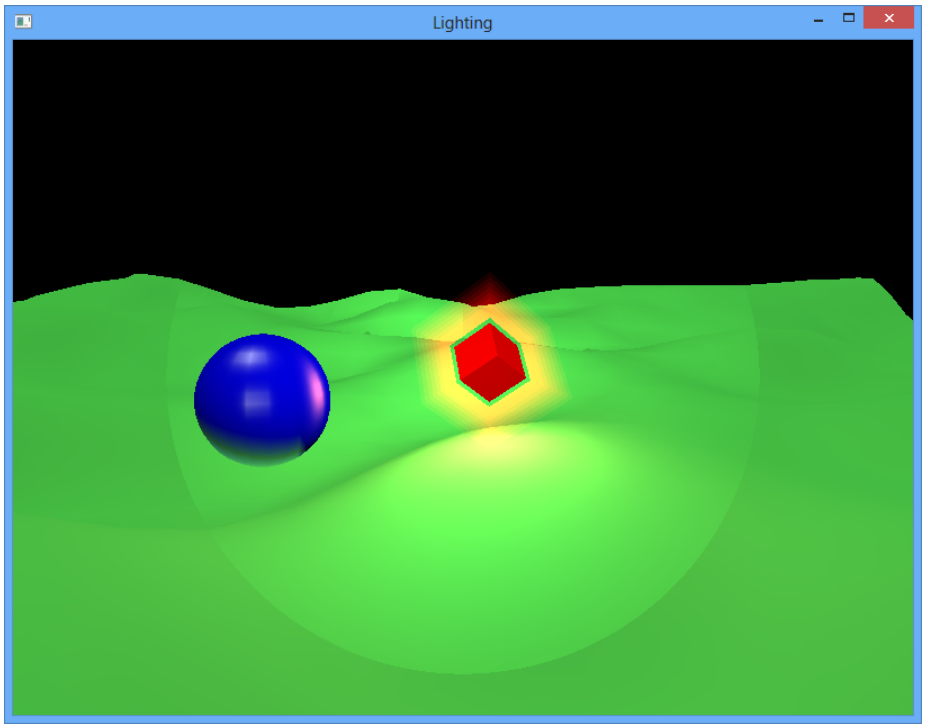
This will enable the use of the shader programs `lighting2_vertex.glsl` and `lighting2_fragment.glsl`. Your task is to modify the code in the fragment shader to implement the same lighting logic as before (but now applied to each fragment).

In the end, add the code line that performs gamma correction for the computed color. Recall that gamma correction means transforming the color from linear RGB colorspace to non-linear sRGB colorspace (used on the display) via the transformation:

$$\text{color}_{\text{sRGB}} := (\text{color}_{\text{RGB}})^{\frac{1}{2.2}}$$

Hint: Solving this task is all about copy-pasting the solution of your previous task to the fragment shader, fixing a couple of variable names and making sure you return the result in the `gl_FragCoord` variable.

Below is the result you should expect to see:



3 Fragment Shaders

In general, a fragment shader is a very powerful concept, the application of which extends far beyond basic lighting computations. Fragment shaders are particularly well-suited for implementing hardware-accelerated raytracing techniques. You can get a good overview of what kinds of effects can be implemented in a single fragment shader by browsing the *ShaderToy* website[2]. Now, although understanding the techniques behind demos like `https://www.shadertoy.com/view/lsXGWl` in full detail is somewhat beyond the scope of our course[3], an easy starting point is trying to use a fragment shader to implement a simple Phong-lighted orthogonal projection of a sphere.

Exercise 4* (1pt). We shall be using the "ShaderToy" website in this exercise. Open the website and click "New Shader". You will be presented with the `main` function of an empty fragment shader. Recall that the shader must produce the value for the `gl_FragColor` variable – color of the pixel located at position `gl_FragCoord`.

[2] `https://www.shadertoy.com/`

[3] It is not too far beyond this course, though. Take a look here for a list of useful keywords, if you are interested: `http://www.iquilezles.org/www/articles/proceduralgfx/inspire2008.pdf`

1. Suppose we are trying to render an orthogonal projection of a sphere. Our viewport is positioned so that the sphere is centered in the window and its height is exactly half the window's height. Let us start coding our shader by determining the pixels that belong to such a sphere.

```
void main(void) {
    // Center and normalize coordinates
    vec2 uv = gl_FragCoord.xy − iResolution.xy∗0.5;
    uv /= iResolution.yy∗0.5;

    // Write red pixels for all points belonging to the sphere
    if (length(uv) < 0.5) {
        gl_FragColor = vec4(1, 0, 0, 1);
    }
    else {
        gl_FragColor = vec4(1, 1, 1, 1);
    }
}
```

2. Now for every pixel that does belong to the sphere you have to implement the following:

   (a) Compute the position of the corresponding point on the sphere in 3D. Assume that sphere is centered at $(0, 0, 0)$ in camera coordinates.
   Hint: Every point on the sphere satisfies
   $$x^2 + y^2 + z^2 = R^2.$$

   (b) Compute the normal at this point.

   (c) Assume the direction to the viewer is $(0, 0, 1)$ as usual.

   (d) Assume that the sphere is illuminated by a positional light source located at $(5, 2, 5)$. Compute the light direction and the reflection vectors.

   (e) Assume the sphere's diffuse color is red $(1, 0, 0)$, specular is white $(1, 1, 1)$, shininess is 15, ambient light is $(0.1, 0.1, 0.1)$. Let the light source intensity be $(0.7, 0.7, 0.7)$ for both diffuse and specular components. Assume no attenuation. Given this information and the previously computed vectors, evaluate the Phong lighting model at the corresponding point on the sphere.

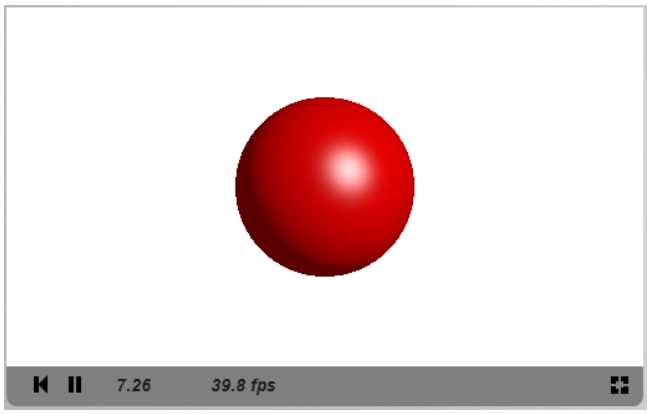
   (f) Apply gamma correction.

   (g) Apply HDR, if you find it necessary.

   (h) Finally, make the light position move around the sphere in a circle. I.e. do something like

```
vec3 light_pos = vec3(5.0*cos(iGlobalTime), 2.0,
                      5.0*sin(iGlobalTime));
```

3. As a result you will have implemented an animated rendering of a complete sphere done fully within a fragment shader (i.e. no "standard graphics pipeline" involved). The resulting image should look more or less as follows:



4. Submit the solution to this exercise in a text file `sphere.glsl` added to the archive. Note in the submission comment that you did solve this task.