Programming Assignment 4: Shader programming
Computer Graphics, Autumn 2016

Submission deadline: Thursday, November 17th, 12:00

This project is about adding shading and texturing functionality to your rendering engine. You will learn to work with GLSL shader programs. Contrary to the last assignment, where you wrote your own rasterizer, we will return to using OpenGL (but convince yourself that your software-renderer could do all of this too, just slower). Create separate eclipse projects for each task so you can demonstrate them separately. You can copy the project "simple" and extend it for each task.

This assignment needs to be handed in as usually on Ilias by the date indicated above. Additionally, the assignment must be demonstrated to a teaching assistant on a computer, as usual. Inscribe your name in the online list on Ilias, as always.

Before you start

Examine the different shaders already contained in the base code. You can find them in the project `jrtr` in the directory `shaders`. The most relevant shader is the shader `diffuse.vert/.frag`, which implements diffuse lighting with one light source and a texture. A short explanation for the special variable types in the shaders:

- Type `uniform`: Basically, `uniform` variables in the vertex or fragment shader are passed from the Java code using the function `glUniform`. All variables which are constant for all vertices and/or pixels are passed this way. Typically, such variables include transformation matrices, information about light sources or textures. The call of `glUniform` needs to be performed after the shader has been activated but before the vertex data is drawn via `glDrawArrays` (see below).

- Type `in` in the vertex shader: The `in` variables in the vertex shader are passed from Java to the shader via so-called "Vertex Buffer Objects". The corresponding Java code is already implemented in `GLRenderContext.draw`. Rendering the data is triggered by the call to the function `glDrawArrays`. "Vertex Buffer Objects" contain all data assigned to the triangle vertices such as "positions", "normals" or "texture coordinates". It is important to note that none of this data is interpreted by OpenGL – your shader will make sense of the data. Index buffers contain indices into the list of vertices to form the actual triangles. This is mostly an optimization: Vertices will have to be stored and transformed only once thanks to this. It is also conceptually important, because it indicates "watertight" meshes...

- Type `in` in the fragment shader, or type `out` in the vertex shader: The `in` variables of the fragment shader correspond to the `out` variables of the vertex shader. These variables are forwarded by OpenGL from the vertex- to the fragment shader and are also automatically interpolated at each pixel.

- Type `out` in the fragment shader: These variables are written to the frame buffer. Typically, this is the color of the pixels, but many advanced techniques write out all sorts of data to different buffers. The shader does not need to write the depth values explicitly to the frame buffer, they are automatically managed. It can be done manually if desired though.

In the Java code of `jrtr` each 3D object of the type `Shape` has a reference to a `Material`, which contains the properties of the material and a reference to a shader program, with which the object is to be rendered. Examine the method `setMaterial` in the class `glRenderContext`, where the shader is activated and the material properties are passed to the `uniform` variables of the shader, as explained above. In this method, the parameters of the light sources are also passed to the shader. In the Java code, light sources are saved as objects of the class `Light`, and a list of light sources is managed in `SimpleSceneManager`.

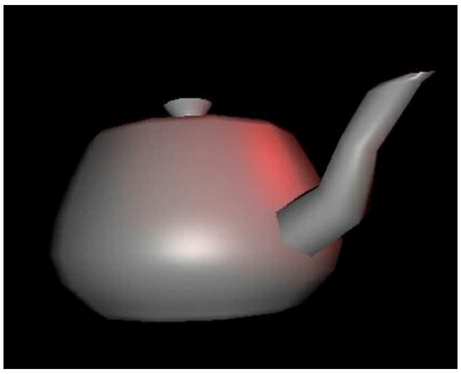
A good summary for GLSL: GLSL Quick Reference Card. Use the Ilias forum if you have questions or problems.

1 Shading with Multiple Light Sources (2 points)

Develop a shader which computes diffuse lighting with multiple point light sources by extending (that means copying and modifying, i.e. starting from) the already existing shader `diffuse.vert/.frag`. The parameters of the light sources are saved in `uniform` variables in the shader. Use arrays to accept the data of multiple light sources. You may set the length of the arrays to a fixed number, so the maximum number of lights the shader can handle is limited. These parameters need to be passed to the shader from the host program using the correct variants of the function `glUniform*`.

Basically, there is already code passing light source parameters to the shader in `setMaterial` of `glRenderContext`. Do note, however, that the existing code only passes the directon of directional light sources. For point light sources, you will need to pass the position of the source additionally.

Demonstrate your shader by constructing a scene with various objects with different material properties (different diffuse reflection coefficients and/or textures). Extend the cylinder from the first assignment by assigning texture coordinates to it and show it in the scene, too. The texture coordinates can be assigned similarily to the texture coordinates of the cube in the class `simple` of the basecode.


Teapot model with specular reflection of 2 light sources with different colors

2 Per-Pixel Phong Shading (3 points)

Develop a vertex- and fragment-shader program which additionally implements specular reflection using per-pixel Phong shading with multiple point light sources. This shader simply adds the contributions of the diffuse and the specular reflections.

Demonstrate your shader by constructing a scene with two light sources of different colors as shown in the figure above. You can best distinguish the two light sources if you choose very shiny material properties.

3 Texturing with Phong Shading (2 points)

Extend your Phong shader to support texturing and lighting in the same shader. In order to do that, simply use the colors of the texture as diffuse and ambient reflection coefficients. Also implement a "gloss map", which means using texture values to determine the specular reflection coefficient. You can for example use the brightness (the sum of red, green and blue) of the texture as specular coefficient.

Demonstrate this functionality using a scene with at least two objects (or surfaces) with different textures. You can find many interesting textures by performing an image search with the queries "grass texture", "wood texture" etc. For testing, there is a teapot model with texture coordinates in the folder `obj`, which you can use.

4 Experimenting with Shaders (3 points)

The goal of this task is to experiment with any other shader. You may expressly use shader code you found on the internet, or you can implement a shader of your choice. Possible keywords for simple shaders are "toon shading", "procedural brick shader", "procedural stripe shader", or (if you are feeling brave) "procedural noise shader". Take into account that a lot of shaders you can find online have maybe been written for older versions of GLSL and need to be modified to work with our code (you will recognize this by no longer supported keywords such as `attribute` and `varying`). You will also have to make sure you use the same names in the Java and GLSL code for all `in` and `uniform` variables and the same names for the variables passed from vertex- to pixel shader.

You should demonstrate the shader in a scene of your choice in our Java renderer, and you should be able to explain the code. (If you have copied it from online).