CG1 WS 22/23 - Exercise 3: Shading

Technische Universität Berlin - Computer Graphics
Date 24. November 2022 **Deadline** 07. December 2022
Prof. Dr. Marc Alexa, Dimitris Bogiokas, Ugo Finnendahl, Max Kohlbrenner, Markus Worchel

Shading (7 points)

In order to render geometry and materials, we usually use a dedicated graphics processing unit (GPU). It provides a rendering pipeline that does most of the calculations highly parallelized. Many parts of this pipeline can be configured and some are programmable. The programmable parts are driven by (small) programs running on the GPU, called *shaders*. Two important shaders are the *vertex shader* and the *fragment shader*. The vertex shader first operates vertex-wise and later the fragment shader is executed per fragment/pixel. Depending on the API used, one is restricted to a specific programming language. In our case, we use *WebGL*, which uses *GLSL ES* – a `C`-like language – for the shaders. Other APIs, for example DirectX, OpenGL, Vulkan and Metal, use similar shader languages. In this exercise sheet, you have to implement different shaders. The shaders used for rendering in `Three.js` are defined object-wise in the material.

As the CPU and the GPU do not share memory we need to pass all relevant data to the GPU. There are two different concepts for variables that we can pass: `uniform` and `attributes`. A `uniform` variable is defined for an object and is valid for all its vertices and rendered pixels. Therefore it is accessible and constant in both the vertex and fragment shaders. An `attribute` variable is defined per vertex and is only passed to the vertex shader as an `in` variable. Variables can be passed from vertex to fragment shaders by declaring them as `out` (vertex shader) and `in` (fragment shader) variables sharing the same name in both shader files. Its value in the fragment shader is interpolated between the vertex values (see exercise slides for more detail).

The tasks in detail are:

1. **Basic Ambient Shader** We have given you a basic vertex and fragment shader which together output a constant vertex and color. Complete them to render the scene using a basic ambient shader. The vertex shader needs to calculate the normalized device coordinates like in the last exercise. The fragment shader computes the resulting pixel color using the light color and the ambient reflectance, both should be adjustable using the *Ambient color* picker and *Ambient reflectance* slider in the gui. (*1.5 points*)

   Restriction: You have to use `RawShaderMaterial` in this and all following tasks, **not** `ShaderMaterial`.

   Hint: `Three.js` already passes a lot of useful `uniforms` and `attributes`, those are listed and commented in the basic shader examples. The light color and reflectance needs to be passed additionally as `uniform`.

2. **Normal Shader** Implement a simple normal shader that visualizes the normal direction of the surface in world space as a color. Map the normals bijective from the unit sphere to the RGB color space. Note that the *normal* `attribute` that is predefined by `Three.js` is defined in local coordinates. (*1.5 points*)

   Hints: 1. When a linear transformation is applied to a surface, its normals need to be transformed correctly using the inverse transpose of the transformation, you might have to pass this matrix as an additional `uniform`.[1] 2. In the pipeline, the normals are defined per vertex and need to get (transformed and) passed to the fragment shader as an interpolated quantity using `in` and `out` variables.

3. **Toon Shader** Implement a toon shader. The color is based on discrete thresholds of the angle between the viewing direction and the surface normal. The larger the angle, the darker the surface color. You need to define four different shades of a color you like. Use pixel wise color calculation. (*1 point*).

[1] There are two transformed spheres in the scene. One is scaled using the local coordinate system, the other is scaled in the geometry and has therefore already corrected normals. On the screen both spheres should look the same.

4. **Diffuse Light Shader** Implement a diffuse (*Lambert*) illumination shader with a single point light source. The point light source should be movable using the gui and add a visualization in form of a sphere to the scene. Changes to the diffuse reflectance slider and color picker should lead to the correct update of pixel colors. You can use either gouraud or phong shading. Whether you add an ambient light component is your choice and does not affect the grading.

   Hint: You do not need any `Light` object provided by `Three.js`, as they only work on `Three.js Materials`. Just use a simple 3D vector representing the light. (*1.5 point*).

5. **Specular Light Shaders** Now implement the complete *Phong* illumination model by adding a specular component to the lambert and ambient calculations. Implement the two common interpolation methods for it: *Gouraud* and *Phong* using separate shaders. Again all changes to the according reflectance sliders and color pickers should lead to the correct update of pixel colors. The magnitude slider should control the falloff parameter m of the Phong light model. The point light source should be movable using the gui. (*2 points*)

6. **Blinn-Phong Specular Light Shader** Implement a *Blinn-Phong* shader using the Phong interpolation technique. The reflectance sliders and color pickers should update like before. The magnitude slider should now control the falloff parameter n of the Blinn-Phong light model. The point light source should be movable using the gui. (*0.5 points*)

Requirements

- Exercises must be completed individually. Plagiarism will lead to exclusion from the course.
- Submit a `.zip` file of the `src` folder of your solution through ISIS by **07. December 2022, 23:59**.
- *Naming convention*: {firstname}_{lastname}_cg1_ex{#}.zip (for example: jane_doe_cg1_ex3.zip).
- You only hand in your `src` folder, make sure your code works with the rest of the provided skeleton.