Interactive Computer Graphics Coursework – Task 3 (*assessed)

Bernhard Kainz, Antoine Toisoul, and Daniel Rueckert

January 31, 2017

Task 3: Illumination and Shading

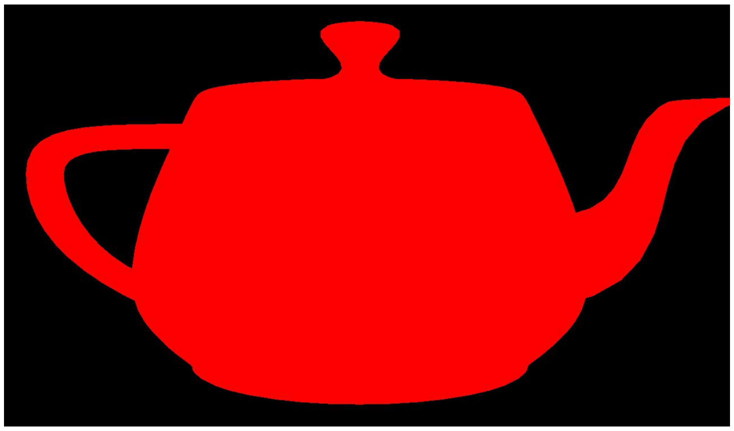
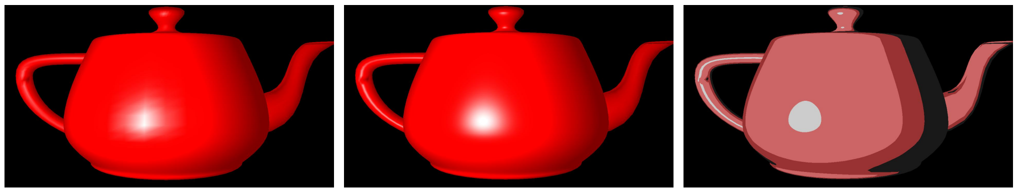


Figure 1: Default scene shown by the framework: an unshaded teapod model.

In this exercise you will learn how to use vertex and fragment shaders for vertex-wise and pixel-wise scene illumination. We use the *per-polygon* vertex and fragment shaders for this task. The framework shows you an unshaded read teapod model per default as shown in Figure 1. At the end of this task you will be able to program different shading models as shown in Figure 2.



(a) Gouraud shading (b) Phong shading (c) Toon shading

Figure 2: Results of Exercise 2

Task 3a: Per Vertex Gouraud shading*

For this exercise you will need to edit the file *per-polygon* vertex shader. This shader and performs operations on scene vertices in object space. It is already provided by the editor.

In this exercise you will implement *Gouraud shading* as discussed during the lecture. Gouraud shading is an interpolation scheme for the illumination based on the viewer's, the vertex' and the light position within the scene.

In the *per-polygon* vertex shader we have already defined a basic interface to the other shaders to pass on specific information about the currently processed vertex:

```
out VertexData{
  vec4 position_camSpace;
  vec3 normal_camSpace;
  vec2 textureCoordinate;
  vec4 color;
} VertexInOut;
```

The keyword `out` specifies, that this struct will be passed on to the following shaders (the geometry shader and the fragment shader).

The function `main()` defines already a simple pass-trough vertex shader. This means, that this shader does exactly the same as what would be done during the static rendering pipeline except that it defines the output color to a constant red value:

```
vec4 vertex_camSpace = mvMatrix*vertex_worldSpace;
vertexInOut.position_camSpace = vertex_camSpace;
vertexInOut.normal_camSpace = normalize(normalMatrix*normal_worldSpace);
vertexInOut.color = vec4(1.0, 0.0, 0.0, 1.0);
vertexInOut.textureCoordinate = textureCoordinate_input;
gl_Position = pMatrix * vertex_camSpace;
```

`gl_Position` is the only output that is expected to be set by the glsl compiler in most OpenGL versions. Everything else can be freely defined. `mvMatrix` is the ModelView matrix and `pMatrix` is the projection matrix. These are updated by the main program using `uniform` variables. Please not that previous versions of OpenGL and GLSL had intrinsic variables for values like the ModelView and Projection matrix. Modern OpenGL is freely programmable and defines the use of the intrinsics as deprecated. Many examples on the internet might use old style OpenGL and intrinsics.

We also provide a definition for a simple light source. Its properties are defined by:

```
vec4 diffuse;
vec4 ambient;
vec4 specular;
float shininess;
```

You can access the position of the single light source in this scene by `vec4 lightPosition_camSpace`. To convert a vector from `vec4` to `vec3`, you can either cast the vector, e.g., `vec3 vec = vec3(lightPosition_camSpace),`, or access the vector elements individually

`vec3 vec = lightPosition_camSpace.xyz;`

`float x = lightPosition_camSpace.x;` etc..

Your task for this exercise is to redefine `VertexInOut.color` according to *Gouraud shading.* Use the provided *Utah Teapot* model for testing. (Scene Tab → Object → Teapot). An example output for this task is shown in Figure 2(a).

Note that you only need to define one color per vertex. The interpolation between these vertices's is done by the rendering pipeline. Note the illumination problems at the border of the specular highlight.

Task 3b: Per Pixel Phong shading*

In this part you will implement *Phong shading* as discussed during the lecture. Phong shading is an interpolation scheme for the illumination based on interpolated normal vectors for each fragment instead of interpolated colors as done for the previous task . The shading effect depends on the viewer's, the fragment's and the light position. In contrast to Exercise , this exercise operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader.

You can use the same interface definitions as provided in Exercise . However, note that in the fragment shader you get an input fragment instead of an input vertex:

```
in fragmentData
{
vec4 position_camSpace;
vec3 normal_camSpace;
vec2 textureCoordinate;
vec4 color;
} frag;
```

Your task for exercise is to redefine `colorOut` according to *Phong shading.* Use the provided *Utah Teapot* model for testing. (Scene Tab → Object → Teapot)

You can again access the light source position via `vec4 lightPosition_camSpace;`. If you test your programme, the result should look similar to Figure Figure 2(b). Note that the quality of the specular highlight is better than in Figure Figure 2(a).

Task 3c: Per Pixel Toon shading*

In this part you will implement *Toon shading*. Toon shading is a simple lighting scheme, which allows you to achieve effects similar to hand drawn cartoons. This exercise also operates directly on fragments and needs therefore the extension of the *per-polygon* fragment shader. You can use preprocessor definitions as common in C-like language to switch between the shading types.

A toon shader can be defined per fragment as done in equation 1 and equation 2.

$$I_f = \frac{l}{||l||} \cdot \frac{n}{||n||} \tag{1}$$

$$I = \begin{cases} (0.8, 0.8, 0.8, 1.0), & \text{if } I_f > 0.98 \\ (0.8, 0.4, 0.4, 1.0), & \text{if } I_f > 0.5 \text{ and } I_f <= 0.98 \\ (0.6, 0.2, 0.2, 1.0), & \text{if } I_f > 0.25 \text{ and } I_f <= 0.5 \\ (0.1, 0.1, 0.1, 1.0), & \text{if } else \end{cases} \tag{2}$$

Your task for exercise is to redefine `colorOut` according to *Phong shading.* Use the provided *Utah Teapot* model for testing. (Scene Tab → Object → Teapot)

The final result should look similar to Figure 2(c).

Implement this task first using the provided light source position. The specular and illumination will stay constant relative to the position of the light source. Try to replace the static light source with a *head light*, *i.e.*, set the light source position equal to the position of the camera in camera coordinate space.

Task 3d: Blinn-Phong

Phong shading is not the most efficient way to approximate illumination. Blinn-Phong is a more efficent modification of Phong shading using halfway-vectors.

Your task for exercise is to redefine `colorOut` according to *Blinn-Phong shading* as discussed during the lecture. Use the provided *Utah Teapot* model for testing. (Scene Tab → Object → Teapot) Figure 3 shows and example output.

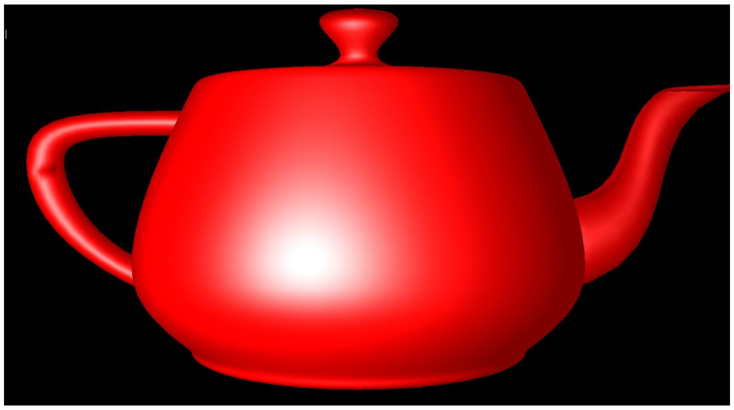


Figure 3: Result for Task 3b: Blinn-Phong shading

HAVE A LOT OF FUN!!