OpenGL Assignment 2: Shading and Texture Mapping

February 20, 2022

Computer Graphics

1 Theoretical question (1 point)

This theoretical question can be submitted with a PDF file. We recommend using LaTeX, but we allow other submissions (including handwritten ones) as well.

This question is parametrised by the digits of one of your group member's student number. If your student number is $s1234567$, then $a = 1$, $b = 2$, $c = 3$, $d = 4$, $e = 5$, $f = 6$, $g = 7$.

In this assignment we will compute the light intensity of a vertex in two dimensions. Suppose the vertex has the following properties:

- Coordinates: $V = (f, g)$
- Normal: $N = (0, 1)$
- Color (RGB): $V_{col} = (0, 1, 0)$
- Ambient coefficient: $k_a = 1$
- Diffuse coefficient: $k_d = 0.6$
- Specular coefficient: $k_s = 0.4$
- Specular exponent: $p = 1$

We have a point light source with the following properties:

- Coordinates: $L = (d + 10, e + 20)$
- Color (RGB): $L_{col} = (1, 1, 0)$
- Ambient intensity: $I_a = 0.2$
- Diffuse intensity: $I_d = 0.8$
- Specular intensity: $I_s = 0.3$

The camera has coordinates $C = (b - 10, c + 10)$.

1. Compute the resulting color of the vertex as seen by the camera using the Phong illumination model.
2. In the Gouraud shading model we first calculate the resulting color of the vertices and then interpolate to obtain the color of the pixels of a triangle. In the Phong shading model we first interpolate the properties of the surrounding vertices and then compute the resulting color for a given pixel. Suppose we have another vertex at $W = (d, e)$ with a normal of $M = (1, 0)$ and color $W_{col} = (1, 0, 0)$. Sketch and explain how you interpolate the color and the normal between the vertices V and W.
3. Propose an addition to the Phong shading model to make the lighting/shading more realistic.

2 OpenGL implemenation

For this assignment, either extend your code from the previous OpenGL assignment or use the framework (`OpenGL_2.zip`) provided on Nestor.

In this assignment you will implement a set of shading techniques, and will implement them in such a way that your program can easily swap between them. After this, you are given the opportunity to not only work with your own models, but also add your own textures!

3 Shaders

To make your previous assignment work, you already implemented a minimal shader program to start with. Now, we will create a set of *additional* shader programs that will render your mesh differently. In your UI, you will be able to switch between shader programs in real-time.

Normal Shader (1 point)

The first new shading method your program will support is a simple shader that will visualize the normals of your mesh. This method calculates the colour at each vertex using the normal of that vertex. These colours are then interpolated across the surface of the triangle. The fragment shader will receive the interpolated colour as a user-defined input.

1. As we will be using normals instead of colours, change the naming of your vertex colour related variables in all your C++ and GLSL code to refer to normals instead. Test to see if your application still works (the changes should only be semantic).
    - Remember that you can pass variables from the vertex shader to the fragment shader by setting the **out** in the vertex shader and **in** in the fragment shader, they *must* have the same name.
2. To render the normals of your mesh, we will need to actually pass this additional data to the GPU. In other words, we will need to add a *normal* attribute to our vertices. The model loader can give you these normals by calling its `getNormals()` method. Append this to your mesh data instead of the random colours you assigned in the previous assignment.

3. Create a QMatrix3x3 inside `Mainview::paintGL()`. We will use this matrix for transforming the normals, which needs to be done in order to preserve their relative angles when your model gets transformed.

4. Create a uniform variable for this matrix in your vertex shader. Do not forget to also add a GLint for that uniform, such that `Mainview` will be able to refer to it.

5. Make sure the normal matrix is properly updated along with the other matrices. Please refer to the lecture slides in order to understand how your normal matrix should be constructed differently. *Hint*: you may use the provided `normalMatrix()` member from `QMatrix4x4`.

6. In the vertex shader, take the normal at each vertex and transform it with the normal matrix. Pass the result to the fragment shader.

7. The fragment shader should now receive an interpolated normal. *Normalize* it, map it to a colour (see the lecture slides on how to do this), and return it.

8. The result should look like Figure 3.1 (depending on the model you use). Test with and without the normal matrix and describe the difference in your `README` file!

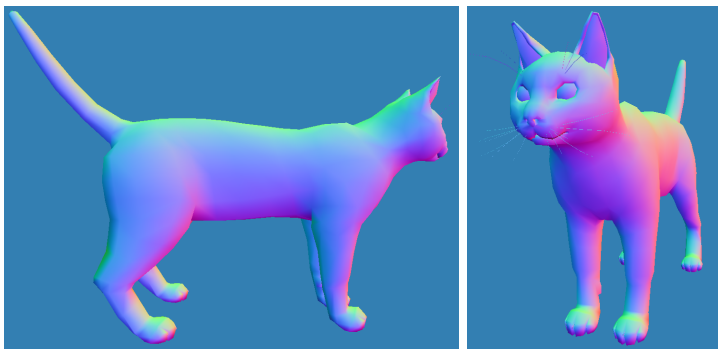

Figure 3.1: *The visualized normals of a cat.*

Selecting Shaders (0.5 point)

Before implementing any additional shaders, we should make sure we can properly switch between them without problems.

1. Go to the folder containing your shaders, and rename `vertshader.glsl` and `fragshader.glsl` to the more appropriate `vertshader_normal.glsl` and `fragshader_normal.glsl` respectively. Then, make two duplicates of them and rename these to the following:

   - `vertshader_gouraud.glsl` and `fragshader_gouraud.glsl`.
   - `vertshader_phong.glsl` and `fragshader_phong.glsl`.

2. Change the output color of each fragment shader such that you can easily make a distinction between the different shader programs (for example, divide the colour of Gouraud by 2 and Phong by 3, making their output slightly darker).

3. Import the new set of shaders into Qt. You can do this by right-clicking the `shaders` folder and pressing "add existing files".

4. Create two new shading programs and update the renamed paths of the normal shader, each using their corresponding normal, Gouraud or Phong shaders.

5. Create unique uniforms pointers to refer to the uniforms of each shader program, make sure they are all updated.

6. Allow for selecting the different shader programs in your UI. There should already be a set of toggles that correspond to a `ShadingMode`. You can use this `enum` to select the right shading program before painting to the screen.

7. Switch between the shader programs to check if your addition was successful. If so, you can continue to the next part of this assignment.

Gouraud Shader (1 point)

The second new shading method your program will support is called Gouraud shading. This method calculates the colour intensity at each vertex using the normal of that vertex and the direction to a light. These intensities are then interpolated before they are received by the fragment shader.

1. Implement Gouraud shading in `vertshader_gouraud.glsl` and `fragshader_gouraud.glsl`. Keep the following in mind:

   - Refer to the lecture slides for a definition of the *Phong Illumination Model* you can use.
   - You will need an extra uniform for the light position, the material and optionally you may add a uniform for the color of the light. If you want, you can add widgets in your UI to modify these uniforms.
   - Keep the light fixed with respect to the camera. When rotating the model, you should be able to see the "dark" side.
   - Calculate the colour using the Phong Illumination Model in the vertex shader and send it to the fragment shader.

2. The result should look like Figure 3.2 (depending on your model and material properties).

Phong Shader (1.5 point)

Phong shading improves on Gouraud shading by calculating the lighting not just for each vertex, but for each triangle fragment itself.

1. Implement Phong shading in `vertshader_phong.glsl` and `fragshader_phong.glsl`. In theory, you only need to move your illumination model from the vertex shader to the fragment shader.

2. The result should look like Figure 3.3 (depending on your model and material properties).

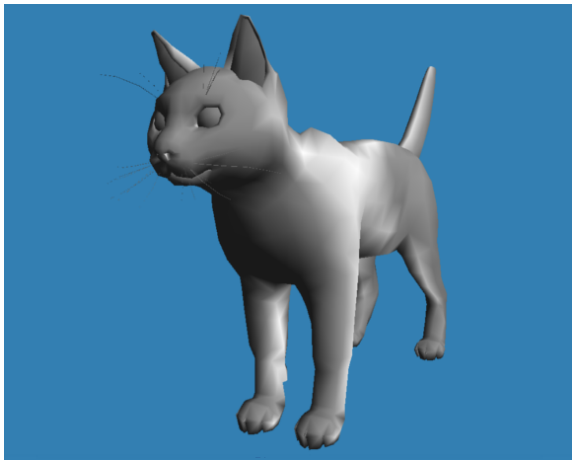

Figure 3.2: *A shiny Gouraud shaded cat.*

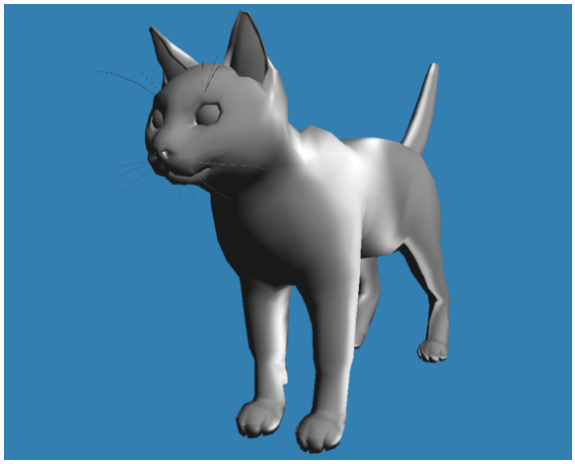

Figure 3.3: *A shiny Phong shaded cat.*

Suggestions for the competition

If you want to add some additional fun features, you might want to do the following:

- Implent a Z-Buffer shader: https://en.wikipedia.org/wiki/Z-buffering
- Implement a Cel shader: https://en.wikipedia.org/wiki/Cel_shading

4 Texture mapping

Using textures is an easy way to create more detail in objects. Instead of using a single colour, an image is mapped to an object, such as fur colour on a cat.

You may use the cat, cube and/or the sphere model provided in the previous assignment, but you may also use your own models and textures or models/textures available online (see today's slides for a couple of links to repositories). Please use texture images of size 1024 pixels by 1024 pixels. Using sizes that are powers of 2 are generally preferred for performance reasons. You can use a tool like Blender (https://www.blender.org/) to export other models to the `.obj` file format. **Important!** Remember to set the export options to *triangulate faces*.

Basic method (4 points)

Map a square *diffuse texture* (also referred to as a *colour texture*) to a mesh.

1. To use a `.png` as a texture, we first need to be able to read this file into memory. Included with this document you will have found a file named `utility.cpp`, which contains the method `QVector<quint8> MainView::imageToBytes(QImage image)`. Add this file to your project sources, it will allow you to convert a `QImage` (constructable with the path to an image) to a simple vector of values. Do not forget to also declare this method in your `mainview.h`!

2. You will need to generate a texture for the GPU to store your image data, and you will need to remember a pointer to this texture so you can bind it later. First, create a `GLuint` in your `mainview.h`, then generate a texture in `mainview.cpp` by calling `glGenTextures`.

3. To modify or use your generated texture, you need to bind it using `glBindTexture`.

   - Now might be a good time to remind you of the benefits gained from seperating your concerns into different methods. For example, binding a particular texture and pushing data to it might be neatly implemented in a method such as `void MainView::loadTexture(QString file, GLuint texturePtr)`.

4. How OpenGL will interpret your texture can be set using `glTexParameteri(GL_TEXTURE_2D, <Parameter Name>, <Parameter Value>)` where `<Parameter Name>` is one of the following:

   - `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T` define the wrapping
   - `GL_TEXTURE_MIN_FILTER` defines the minifying filter
   - `GL_TEXTURE_MAG_FILTER` defines the magnifying filter

   See the OpenGL RefPages for more info on possible values for `<Parameter Value>`.

5. With all the parameters set we can upload the data using the `glTexImage2D()` function. It takes the following parameters:

   (a) Target texture, `GL_TEXTURE_2D` for simple 2D textures.

   (b) Mipmap level, use 0 for now.

   (c) Internal data format, the format OpenGL will store the pixels in on the GPU. Use `GL_RGBA8` for RGBA values, each 8 bits.

(d) Texture width, width of the texture image.

(e) Texture height, height of the texture image.

(f) Border, always use 0 here.

(g) Format, the format of the pixels your are sending. Use `GL_RGBA` (*not* `GL_RGBA8`).

(h) Type of each of the red, green, blue, and alpha channels, use `GL_UNSIGNED_BYTE`.

(i) Pointer to the data to be uploaded, `<yourvector>.data()` where `<yourvector>` is the QVector of bytes you stored the image in using `imageToBytes()`.

6. **Important!** If you use mipmapping, call `glGenerateMipmap(GL_TEXTURE_2D)` after uploading the image or your texture might not show up.

7. To properly map your texture to the mesh, your mesh data will need to provide an additional attribute for its *texture coordinates*. These 2D coordinates can be retrieved using `QVector<QVector2D> Model::getTextureCoords()`. If your coordinates and normals are at location 0 and 1 respectively, you might use location 2 for your texture coordinates.

8. Next, make sure the *Phong* and *Gouraud* vertex shaders receive and pass the incoming texture coordinates to their corresponding fragment shader. In most cases you do not have to perform any transformations on the texture coordinates, since they need to be interpolated directly.

9. In both fragment shaders, create a uniform of the type **sampler2D**. This will be used to retrieve the colour (a **vec4**) from the texture at the provided coordinates. You also need to store the location of these uniforms in `createShaderPrograms()`.

10. In order to set the colour of the fragment to the colour of the texture, use the following line:
`textureColor = texture(samplerUniform, textureCoords)`, where `samplerUniform` is the name of your sampler uniform and `textureCoords` is a vec2 of your texture coordinates you retrieved from the vertex shader. Do not forget to combine this colour with the calculated light intensity!
Note: You may come across the use of the function `texture2D` instead of `texture`. The funtion `texture2D` takes the same parameters and may produce the same result, but is deprecated and should not be used in GLSL 3.30.

11. In `paintGL()`, call `glActiveTexture(GL_TEXTURE0)` and bind the texture using `glBindTexture(GL_TEXTURE_2D, texPtr)` before your call to your drawing function. Replace `texPtr` with your GLuint for your texture.

12. All sampler2D uniforms are bound to `GL_TEXTURE0` by default. To specify to which texture they are bound, call
`glUniform1i(<samplerUniformLoc>, <texture number>)` to specify the texture for each sampler (`glUniform1i(uniTex,1)` for `GL_TEXTURE1` for example). Also see the competition suggetions on using multiple textures.

13. Experiment with different wrapping and filter methods in your final application and describe what each one does in your `README` file.
Tip! For experimenting with wrapping methods try to multiply the vertex coordinates with (2,2) in the vertex shader to see the effects.

An example of a textured cat is illustrated in Figure 4.1.

Figure 4.1: *A textured cat.*

Suggestions for the competition

If you want to add some additional fun features, you might want to do the following:

- Use multiple texture maps (*diffuse* and *bump/normal* maps, or more advanced, *displacement* and *ambient occlusion* or *specular* maps). You need multiple calls to `glActiveTexture` and `glBindTexture` with a different texture number. Also make sure you set the right location of your sampler uniforms!

- Use *anisotropic filtering* on your texture. This will prevent blurring of the texture when your viewing vector is almost tangent to the surface of the textured object. This can be enabled by placing the following code snippet in `loadTexture()` before you upload your texture.

  ```
  GLfloat f;
  glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT,&f);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAX_ANISOTROPY_EXT,f);
  ```

- Implement the function `unitize()` in the `Model` class. As you might have encountered, the scale of a model is determined by the creator of the model. The provided cube has sides of length 2, whereas the sphere has a diameter of 50. It is also possible that the creator of the model did not center it around the origin of the model. By pre-scaling each model, you can make your program more robust and show each model at a similar scale and set the real size with a scaling transformation. Try to determine the size of the bounding box of your model and uniformly scale it to fit inside the unit cube (a cube with sides of length 1) and center it around the origin.